QuantumATK X-2025.06 Documentation

QuantumATK Team

Jul 08, 2025

CONTENTS

1 Getting started with QuantumATK **1**
- 1.1 Installation Guide . . . 1
- 1.2 Introductory Tutorials . . . 20
- 1.3 Feature List . . . 85
- 1.4 Referencing QuantumATK . . . 85

2 NanoLab Guides **87**
- 2.1 Overview of NanoLab . . . 87
- 2.2 Projects in NanoLab . . . 96
- 2.3 Organize your data in the Nanolab data view . . . 98
- 2.4 Builder Manual . . . 102
- 2.5 Workflows Manual . . . 126
- 2.6 Jobs manual . . . 181
- 2.7 Plots . . . 205
- 2.8 Manage 3D View and Select Atoms . . . 239
- 2.9 Importing and exporting files . . . 247
- 2.10 Molecular builder . . . 261
- 2.11 Using the Crystal Builder . . . 276
- 2.12 The Builder Console . . . 284
- 2.13 AddOns . . . 288

3 Tutorials **291**
- 3.1 New or Updated . . . 291
- 3.2 Semiconductors . . . 434
- 3.3 Batteries & Energy Storage . . . 729
- 3.4 Complex Interfaces . . . 779
- 3.5 1D Nanostructures . . . 886
- 3.6 2D Materials . . . 1019
- 3.7 Phonons & Thermal Transport . . . 1097
- 3.8 Molecular Dynamics . . . 1156
- 3.9 Molecular Electronics . . . 1287
- 3.10 Spintronics . . . 1342
- 3.11 Materials, Surfaces and Chemistry . . . 1426
- 3.12 Miscellaneous . . . 1673

4 Manual **1833**
- 4.1 General . . . 1833
- 4.2 Atomic-Scale Calculators . . . 1835
- 4.3 Python in QuantumATK . . . 1940
- 4.4 QuantumATK Reference Manual . . . 1956

4.5 Technical Notes 4288
4.6 Atomic data 4399
4.7 Sentaurus Materials Workbench Reference Manual 4438

5 Publications **4511**
5.1 Referencing 4511
5.2 List of Publications 4511

Bibliography **4513**

Python Module Index **4515**

CHAPTER

ONE

GETTING STARTED WITH QUANTUMATK

This page contains the resources that will help you get started with QuantumATK as a new user.

1.1 Installation Guide

Version: X-2025.06

This guide describes how to install and configure QuantumATK on Windows and Linux. Note that you can skip some of the points if they do not apply in your case. We will describe the conditions for that in each section.

Note: Site information such as Site ID will be asked and checked by the installer, so it should be at hand when starting the process. You can find your site ID at the top of your license file or by contacting your site admininistrator.

If you experience any problems during the installation, first search the User Forum to see if a solution is posted there. If not, please file a case on the SolvNetPlus website (see below) or contact Synopsys QuantumATK support (quantumatk-support@synopsys.com).

QuantumATK can be expected to run out-of-the-box on Windows and Linux platforms (Windows 10+11 and all recent Linux distributions). In some cases, on Linux, there are a few libraries that may need to be installed, but this can always be done post-installation. Future support for platforms is outlined in the Synopsys Compute Platforms Roadmap

1.1.1 Access to SolvNet

The very first thing is to ensure that you have access to SolvNet, which is Synopsys software portal. Go to the SolvNetPlus website and create an account if you do not have one already. Note the following for account creation:

- During the registration process you will need to provide your Synopsys Site ID. If you do not know your site ID, obtain it from your site administrator or from your sales representative.
- You must provide your professional email address. GMail, Yahoo, and other general emails will not be approved.
- Account activation can take up to 48 hours to be processed. If you have not received a confirmation email after this period, please contact securelogin@synopsys.com for assistance.

When you have successfully logged in, please proceed to the next step.

1.1.2 Access to a License Server

You now need to ensure that you have access to a license server. If there is already a license server running at your site, you may skip the section on installation of a license server. If you do not have access, you need to install a license server yourself to use the license. Proceed to either the guide for Windows or Linux, depending on the OS of the machine where you wish to install the license server.

•license_server_linux

•license_server_windows

Note: **For Network or Nodelocked Licenses**: Purchased licenses are issued as Network licenses. That means that a central server will handle the license, and any computer which can connect over the network to the server can check out a license and run the software. Evaluation licenses are nodelocked. The software is then constrained to running on a particular computer only.

1.1.3 Linux only: Download and use Synopsys Installer

In order to install EST images on your Linux system, a working installation of **Synopsys Installer** is required. The main reference should be Synopsys official documentation, here we summarize the most common steps:

•In the official documentation linked above, find the link to the EFT directory and download the latest version of **Synopsys Installer**.

•Inside the folder you will find a file named *SynopsysInstaller_vX.X.run* (where *X.X* is the version), execute it in a terminal with `./SynopsysInstaller_vX.X.run`.

–The process will ask for the desired installation path and will quickly install **Synopsys Installer** at that location.

•Execute **Synopsys Installer**:

–If the *text-based interface* is sufficient, the installation is completed. **Synopsys Installer** executable is located at `<installation_path>/installer` and can be executed in a terminal.

–If the *graphical interface* is desired, execute `<installation_path>/installer -gui` and make sure a window is shown. If there appears any error related to a missing `libXss.so.1` library (possible on a minimal RHEL/CentOS 7.3 installation), proceed to install the *libXScrnSaver* package (for RHEL/CentOS 7.3 this direct download can be used: libXScrnSaver-1.2.2-6.1.el7.x86_64.rpm).

The text-based and graphical interface will provide the same information and ask the same questions. There is also a *batch mode* (or *unattended mode*) available, to install EST images from a command line with limited human interaction. This mode can be useful when automating the installation on several machines and/or several products at the same time. For more information, see below the paragraph *Batch Install on Linux*.

Note: The EST image files should not be renamed, otherwise **Synopsys Installer** will not recognize them as valid product files. Therefore, the original name of the installation file downloaded from SolvNet should always be preserved.

1.1.4 Downloading, Installing and Configuring QuantumATK

You are now ready to install QuantumATK. First step is downloading the QuantumATK installation file.

Downloading QuantumATK

•Log on to SolvNet

•From the SolvNet front page click Downloads or point directly to https://solvnet.synopsys.com/DownloadCenter

•Under **My Product Releases** choose the QuantumATK package assigned to your Site, i.e. a name of the form *QuantumATK xxxxxx*

•Choose the latest product version from the list of available versions. i.e. the X-2025.06 version.

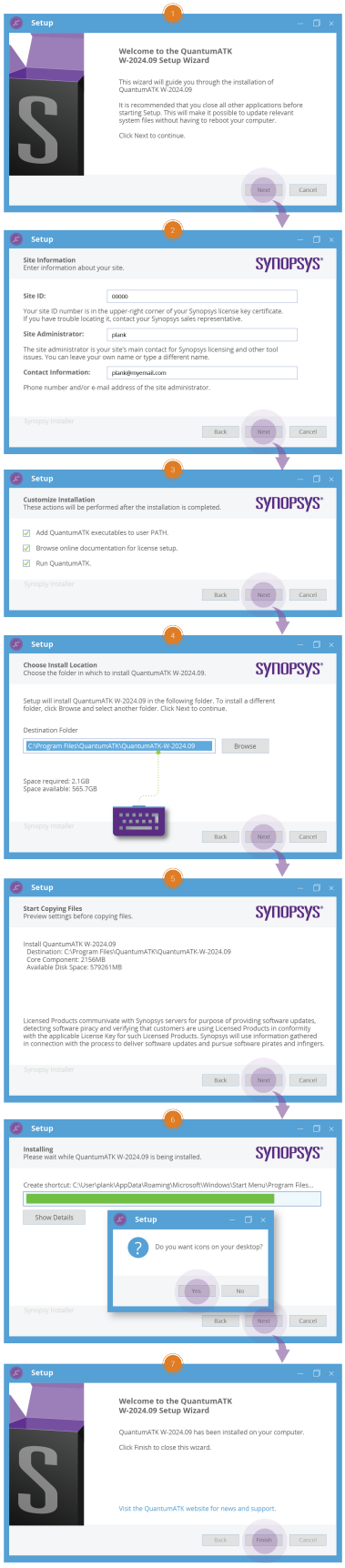
•After reading the legal page, Click on Yes I agree to the above terms.

•Click the Download button next to the QuantumATK installation file corresponding to your operating system.

Installing QuantumATK

Windows

The installation file downloaded from SolvNet is a standalone executable. Right-click and choose *Run as administrator*, as QuantumATK must be installed with *Administrator* rights. Follow the step-by-step instructions to read and approve the license agreement, choose the installation directory, etc. From the final page of the installer you can open QuantumATK, and/or the license configuration tool.


1
Setup
Welcome to the QuantumATK W-2024.09 Setup Wizard
This wizard will guide you through the installation of QuantumATK W-2024.09
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next
Cancel
2
Setup
Site Information
Enter information about your site.
SYNOPSYS
Site ID:
00000
Your site ID number is in the upper-right corner of your Synopsys license key certificate. If you have trouble locating it, contact your Synopsys sales representative.
Site Administrator:
plank
The site administrator is your site's main contact for Synopsys licensing and other tool issues. You can leave your own name or type a different name.
Contact Information:
plank@myemail.com
Phone number and/or e-mail address of the site administrator.
Synopsy Installer
Back
Next
Cancel
3
Setup
Customize Installation
These actions will be performed after the installation is completed.
SYNOPSYS
Add QuantumATK executables to user PATH.
Browse online documentation for license setup.
Run QuantumATK.
Synopsy Installer
Back
Next
Cancel
4
Setup
Choose Install Location
Choose the folder in which to install QuantumATK W-2024.09.
SYNOPSYS
Setup will install QuantumATK W-2024.09 in the following folder. To install a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\Program Files\QuantumATK\QuantumATK-W-2024.09
Browse
Space required: 2.1GB
Space available: 565.7GB
Synopsy Installer
Back
Next
Cancel
5
Setup
Start Copying Files
Preview settings before copying files.
SYNOPSYS
Install QuantumATK W-2024.09
Destination: C:\Program Files\QuantumATK\QuantumATK-W-2024.09
Core Component: 2156MB
Available Disk Space: 579261MB
Licensed Products communivate with Synopsys servers for purpose of providing software updates, detecting software piracy and verifying that customers are using Licensed Products in conformity with the applicable License Key for such Licensed Products. Synopsys will use information gathered in connection with the process to deliver software updates and pursue software pirates and infringers.
Synopsy Installer
Back
Next
Cancel
6
Setup
Installing
Please wait while QuantumATK W-2024.09 is being installed.
SYNOPSYS
Create shortcut: C:\User\plank\AppData\Roaming\Microsoft\Windows\Start Menu\Program Files...
Show Details
Setup
Do you want icons on your desktop?
Yes
No
Synopsy Installer
Back
Next
Cancel
7
Setup
Welcome to the QuantumATK W-2024.09 Setup Wizard
QuantumATK W-2024.09 has been installed on your computer.
Click Finish to close this wizard.
Visit the QuantumATK website for news and support.
Back
Finish
Cancel

Linux

The installation file downloaded from SolvNet is an EST image, with file extension `.spf`. A working installation of **Synopsys Installer** is needed, in order to install an EST image. If you do not have that available, jump to section *Linux only: Download and use Synopsys Installer* for additional instructions on this.

You can run the **Synopsys Installer** in text mode with the `installer` command, or in graphical mode with the `installer -gui` command. The installer will guide you through the installation process by asking for the site information (site ID, administrator and contact), the location of the EST image (the `.spf` file cited above), the installation path and possibly additional options.

Note: On machines without a dedicated GPU and vendor-provided drivers, which might be the case for **Linux** and **Virtual Machines**, some users experience issues with the 3D graphics. For these machines, we recommend choosing Software rendering when installing and upgrading to a higher quality graphics setting after installation, if possible.

Silent Install on Windows

A silent install will not allow any of the customization steps of a graphical install apart from the choice of installation path. Open a CMD shell with administrator rights, it is important to use a CMD shell because the next step might fail in a PowerShell shell. Inside the shell run the following command:

```
.\path\to\QuantumATK-installer.exe /S /LICENSE="I accept"
```

The above command will install QuantumATK in the default location, you can choose a custom location with an additional command line option: `/D=C:\path to\custom\location`. Note that this must be the last option on the command line and must not contain any quotes.

Batch Install on Linux

The main reference should again be Synopsys official documentation, but we summarize here the most common steps. In order to use the batch installer, a configuration file is needed. An initial template can be generated running the following command from the folder containing the EST image:

```
batch_installer -list -o <template_name>
```

Where `<template_name>` is the desired name for the configuration file and `batch_installer` is an executable shipped as part of **Synopsys Installer**. The command above generates a template configuration file that can be customized. The post-install arguments values can be modified as needed, since the default values are aimed at the interactive installation process. Once the configuration file is ready, the batch install can be started with the command:

```
installer -batch_installer -config <template_name>
```

If the command complains about a missing `-target` argument, it can be provided followed by the chosen installation path.

Here is an example of how the generated configuration file will look.

```
SourceDir: /synopsys/downloads
SiteID: 000
SiteAdmin: admin
SiteContact: admin@your_company.com
```

(continues on next page)

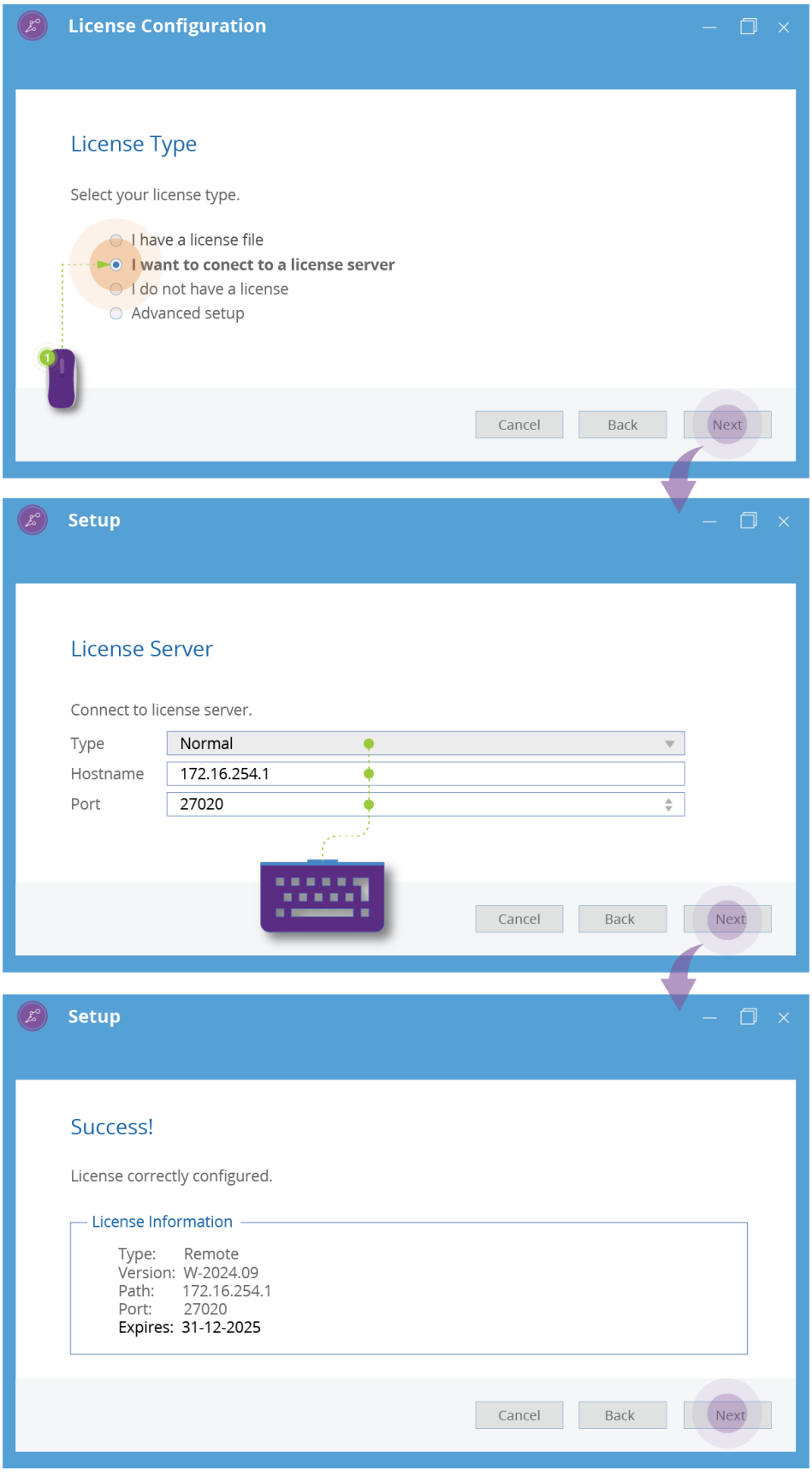
(continued from previous page)

```
PRODUCTS: quantumatk
RELEASES: X-2025.06

quantumatk, X-2025.06 {
  DESCRIPTION: Synopsys QuantumATK
  TARGETDIR: /home/user/synopsys
  POSTARGVALUE: softwareRendering (no)
  POSTARGVALUE: addToPath(yes)
  POSTARGVALUE: openLicenseDoc(no)
  POSTARGVALUE: runProgramAfterInstall(no)
}
```

Note: The batch installer will keep track of previous installations in a hidden folder at `<installation_path>/.installer`. If there is a warning about an existing installation, but the actual QuantumATK installation folder no longer exists, try removing this hidden folder and execute the batch installer again.

Configuring the QuantumATK License

If no valid license is detected when starting QuantumATK, then a step-by-step wizard will pop up and guide you through the license setup process. To connect to the license server, choose the corresponding option and enter the IP address or hostname of the license server. You only need to change the port number if the server is using a different port than the default 27020.


License Configuration
License Type
Select your license type.
I have a license file
I want to conect to a license server
I do not have a license
Advanced setup
Cancel
Back
Next
Setup
License Server
Connect to license server.
Type
Normal
Hostname
172.16.254.1
Port
27020
Cancel
Back
Next
Setup
Success!
License correctly configured.
License Information
Type: Remote
Version: W-2024.09
Path: 172.16.254.1
Port: 27020
Expires: 31-12-2025
Cancel
Back
Next

Note: If configuration is not successful you may want to verify the firewall of your machine or the one of the license server are not blocking port 27020 or whichever other port is used by the license server.

It is also possible to configure the license using environment variables on all platforms, though we recommend to configure the license system via the QuantumATK tool when possible. Using environment variables is the only option on machines without any access to graphics, for example in some Linux-based clusters. The license configuration can be controlled by the environment variable `SNPSLMD_LICENSE_FILE`. Example for a Unix environment: `export SNPSLMD_LICENSE_FILE="27020@mylicenseserver.edu"`

1.1.5 System Requirements

For additional information about the technical characteristics of the system or platform that you want to install the software on, see the technical_hardware section.

1.1.6 Using QuantumATK from the command line

Starting from QuantumATK X-2025.06, the installer will not add the QuantumATK executables to the system PATH on Linux and Windows.

The effect is that the newly installed QuantumATK executables are not available from the command line by default. You can still run QuantumATK from the command line by using the full path to the executable, e.g. `<installation_path>/bin/quantumatk` or `<installation_path>/bin/atkpython`. You can also opt to manually add `<installation_path>/bin` to the system path after the installation. For example, on Linux using the Bash shell, one can add `export PATH=<installation_path>/bin:$PATH` to the `~/.bashrc` file.

You can also use the QuantumATK environment script (`activate_env`) to prepare your command line environment. Users accustomed to Python virtual environments will find this approach familiar. More on the QuantumATK environment script can be found in the *Command Line Usage with QATK Shell Environments* section.

1.1.7 Troubleshooting

Slow Startup on Windows

If you experience slow startup of QuantumATK on Windows, the antivirus software on your system might be the reason. In particular when the application is started for the first time after a reboot of the system, starting QuantumATK might take a few minutes, because the antivirus software scans all files loaded by the program. The easiest way to address this issue is to add the QuantumATK installation directory to the list of directories that are excluded from real-time scanning by the antivirus software. Talk with your system administrator and consider the security implications before proceeding with any changes in antivirus configuration.

If you are using Windows Defender, you can add the QuantumATK installation directory to the exclusion list by following these steps:

1. Open the Windows Security app by clicking on the Start menu and typing Windows Security.
2. Click on Virus & threat protection.
3. Scroll down and click on Manage settings under the Virus & threat protection settings section.
4. Scroll down to the Exclusions section and click on Add or remove exclusions.
5. Click on Add an exclusion and select Folder.

6. Navigate to the QuantumATK installation directory (e.g., `C:\Program Files\QuantumATK\QuantumATK-<VERSION>`) and select it.
7. Click on Select Folder to add the exclusion.
8. Restart the system and start QuantumATK to see if the startup time has improved.

If you are using a different antivirus software, please refer to the documentation of that software for instructions on how to add an exclusion for a directory.

If you are experiencing slowness in relation to your project folder, you can also try adding the project folder to the exclusion list of your antivirus software. Alternatively, if you are using Windows Defender, you can use the above steps but add a Process type exclusion for the entire QuantumATK installation folder instead of a Folder type exclusion. This will make sure that all files opened by QuantumATK binaries at runtime are excluded from real-time scanning.

libGL.so: Cannot Open Shared Object File

The error message:

```
OSError: libGL.so: cannot open shared object file: No such file or directory
```

may appear on Linux systems when you try to launch QuantumATK for the first time.

To solve it, create a new link:

```
$ sudo ln -s /usr/lib/libGL.so.1.2 <quantumatk_installation_path>/lib/libGL.so
```

If you have a 64-bit Linux, use */usr/lib64* instead.

However, first check that you actually have *libGL.so.1.2* in */usr/lib* or */usr/lib64*. It might be called *libGL.so.1* or something else instead, or located in another directory, depending on the video drivers. You can use:

```
$ locate libGL
```

to find the correct file to link to. On some systems we have seen that the same link needs to be created for *libGLU.so* as well. Note that in this case, the error message may still complain about *libGL.so*!

Up to date details can be found in technical_hardware.

libGLU.so.1: Cannot Open Shared Object File

The error message:

```
ImportError: libGLU.so.1: cannot open shared object file: No such file or directory
```

may appear on Linux systems when you try to launch QuantumATK for the first time. To solve it, install the related OpenGL utility library:

- on Ubuntu the package name is *libglu1-mesa*
- on openSuSe the package name is *libGLU1*
- on CentOS/RHEL/AlmaLinux/Fedora the package name is *mesa-libGLU*

Could Not Find or Load the Qt Platform Plugin xcb

If you get the following message when trying to run QuantumATK on certain Linux platforms

```
This application failed to start because it could not find or load the Qt platform␣
↪plugin "xcb".
```

then you might need to install some additional xcb libraries. In order to find out what is missing exactly, one can add some debug output:

```
QT_DEBUG_PLUGINS=3 /path/to/installation/bin/quantumatk
```

This error may also prevent you from starting QuantumATK. In this case, especially if you double-clicked the Desktop icon, no error message is shown, but the text above can be found in the crashlog.txt file, which will be created in your user profile (aka home) directory at `<user_profile_dir>/.quantumatk/crashlogs`.

Read more in technical_hardware.

ATKError: Exceeded Maximum Number of Self-Consistent Iterations

As the error says, the calculation did not converge within the specified maximum number of iterations. If it looks like it is anyway about to converge, you can increase the maximum number of steps and rerun it, although you can save a lot of time by restarting the calculation, as described in the *Manual*.

Failure to Initialize

QuantumATK and atkpython may not be able to run on security-hardened systems. In particular, you should, after discussion with your system administrator, check for the following:

- Verify that the Security Enhanced Linux status is set to Disabled by invoking the utility program *getenforce*, generally available in your Linux distribution package system. You can disable Security Enhanced Linux by running *setenforce 0*
- Verify that resource usage limits are sufficiently high by issuing the command *ulimit -a*.
- Verify your hard disk quota limits with the program *quota*.
- Verify your firewall configuration for rules that may prevent your system to connect to the license server.

ImportError: No Module Named

If you are using the Job Manager to run a script which imports other scripts they must reside in the same folder as the main script or in another directory that is part of the Python path. The paths included by default in the Python path can be printed in a terminal with the command *atkpython -c import sys; print(sys.path)*. If ATKPython or QuantumATK are executed from a shortcut, the run location specified in the shortcut is usually included in the Python path as well. Thus, either move the script to be imported to a relevant folder from where it can be imported, or include the directory where the script resides explicitly in the Python path by inserting something like this in your script:

```
import sys
sys.path.append("c:/my files/scripts")
```

A simple solution is otherwise to run the script from the command line instead; as long as the script to be imported resides in the same directory as you are running from, it will work fine.

Pulay Mixing Inversion Failed. Using Only Last Step

This is an indication of a problem with the geometry of the system. Perhaps atoms are too close to each other. Check the configuration carefully!

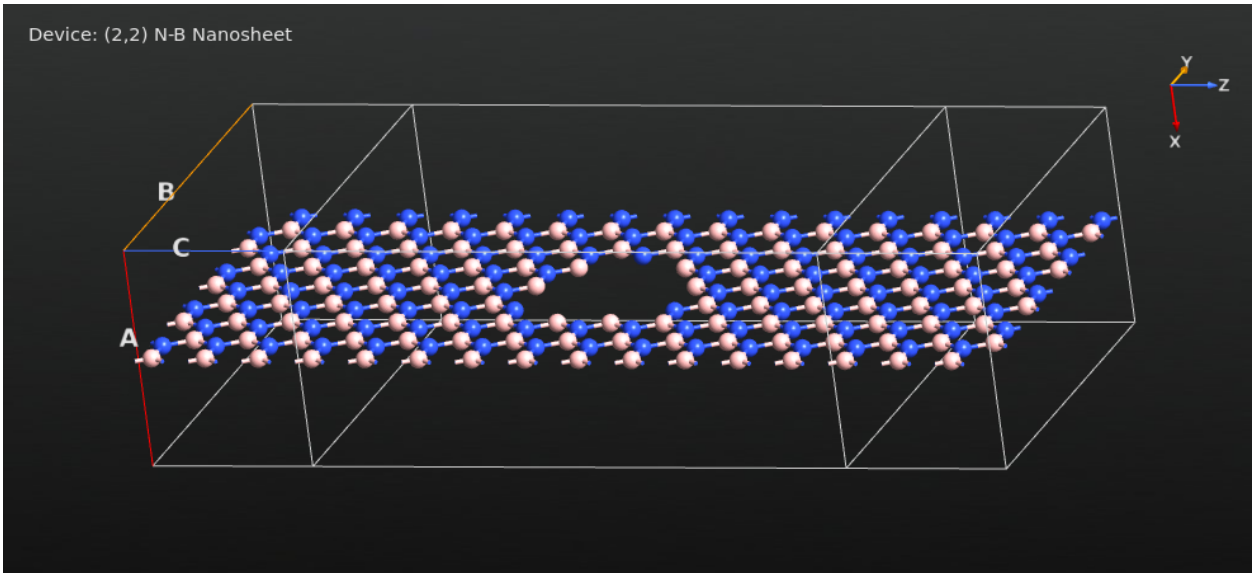
Why are so many k-points needed in the transport direction in a device calculation?

QuantumATK users often ask us why so many k-points are needed in the transport direction in a device calculation. Here is an example from the Forum:

For a calculation of a system like (metallic electrode)-(molecule)-(metallic electrode) with a molecule along the Z axis, some literature references mention k-point parameters like 1x1x100 or 3x3x100 for the Brillouin zone integration. It is surprising, because in a pure interface system like a self-assembled monolayer, reasonable k-point parameters are like 8x8x1. Can you explain this difference?

This is a rather subtle point, so we here address it carefully.

Note: The transport direction is always C || Z, and the X/Y plane is spanned by the A/B lattice vectors. Thus C and Z can be used interchangeably to denote the transport direction.



The key point of the NEGF transport model in **QuantumATK** is that the boundary conditions are not periodic in the C direction. Instead, open boundary conditions are used, in order to apply a voltage across the structure, so that one can compute the electron current under finite bias.

Since the system has no periodicity in C, one should therefore not need more than one **k**-point in this direction. And this is indeed the case - there is no **k**-point sampling used in the device calculation. This is reflected in the specification of the **k**-point sampling for the transmission calculation and other transport analysis quantities: The option for Brillouin zone integration is simply (k_A, k_B).

We do, however, need **k**-point sampling in all directions for the electrodes! These are still assumed to be periodic in C, and the electronic states of the electrodes are obtained from a standard bulk calculation, with **k**-points in all directions. But that still does not fully explain the unusual **k**-point sampling needed for device calculations, because for the typical electrode sizes used, it would appear to be sufficient to use 5 or 12 k_C-points - not 100.

Assume for a moment that you have an ideal system, i.e. the electrodes are identical to the central region. Let H_L, H_{CR}, and H_R be the Hamiltonians of the left electrode, central region, and right electrode, respectively. H_L and

H_R are calculated under periodic boundary conditions, while H_{CR} is evaluated with open boundary conditions using self-energies obtained from H_L and H_R. Now, clearly you will only get perfect transmission in this system if $H_L = H_R = H_{CR}$. To obtain this, the same **k**-point sampling should be used in the A and B directions in the electrodes and in the open boundary condition calculation for the central region. This point is quite trivial, and must always be obeyed in a calculation. The actual number of k_A, k_B points required is a matter of balancing performance vs. accuracy, just like for any other calculation.

In the C-direction, however, the self-energy calculation effectively corresponds to an infinite number of **k**-points, and you will need a lot of k_C-points in the electrode calculation to match the electronic structures of the electrodes and the central region. The primary quantity that needs to be closely matched is the Fermi level. A difference here will lead to slow or bad SCF convergence, and in the worst case inaccurate results due to artificial scattering introduced by the Fermi level mismatch. Such scattering cannot really be distinguished from actual scattering in a complicated system, but it is noticeable in a perfect system. The **QuantumATK** default value for the k-point density of 150 Å is usually on the safe side. But it is relatively inexpensive since these **k**-points are only used in the electrode calculation, which is the smaller and faster part of the full computation.

Note: The short message: Stick to the default value along the C direction, and you should usually be safe.

ATKError: St9bad_alloc

This error usually mean that the calculation went out of memory. Try reducing the number of MPI processes and use threading instead. Otherwise, calculation accuracy or system size have to be reduced.

Runtime Terminates an Application

The full error message is:

```
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

and although it can, in principle, mean many different things, most likely the calculation has requested more memory than is available on the computer.

Killed by Signal 9

When running in parallel, sometimes you may see an error like:

```
rank 1 in job 34  n7_3767
caused collective abort of all ranks
exit status of rank 1: killed by signal 9
```

In most situations this is a case of out of memory. Be careful not to run too many MPI processes on the same node. Sometimes threading can help to reduce memory usage and still parallelize on the available cores.

Not Enough Space to Allocate vblock

This likely means that the calculation went out-of-memory error. Try again with fewer MPI processes and with some threading instead. Otherwise, reduce the calculation accuracy or the system size.

Cant Allocate Memory for Array

As the error message says, the calculation has used all the available memory on the computer. Reduce the number of MPI processes and use threading instead. Otherwise, you have to reduce the calculation accuracy or the system size.

ATKError: Bad Allocation

This message likely means that the calculation has run out of memory. Be careful not to run too many MPI processes on the same node. Sometimes threading can help to reduce memory usage and still parallelize on the available cores.

ATKError: Inverse(DZMatrix const&) : Could Not LU Factorize!

In the majority of cases this error occurs when something is wrong with the geometry, e.g. two atoms sit on top of each other (or are too close), or outside the unit cell (at equivalent positions, perhaps). Check your geometry carefully!

Cannot Run QuantumATK P-2019.03

If you receive the license error below or in general have any issue starting with QuantumATK P-2019.03 please verify the following:

- You must use the Synopsys Common Licensing (SCL) software, version 2018.06 or later.
- You must have the license key files issued later than December 10, 2018.

You can retrieve both SCL installers and license key from the SolvNet portal

License errors:

```
FlexNet Licensing checkout error: SIGN= keyword required but missing from the license␣
↪certificate.
This is probably because the license is older than the application
You need to obtain a SIGN= version of this license from your vendor
```

For more information please visit the License Installation Guide.

Network or Nodelocked Licenses

Purchased licenses are issued as **Network** licenses. That means that a central server will handle the license, and any computer which can connect over the network to the server can check out a license and run the software.

Evaluation licenses are **nodelocked**. The software is then constrained to running on a particular computer only.

License for QuantumATK Parallel Run

You do not need any special license to take advantage of OpenMP threading on multicore computers. To run QuantumATK in parallel using MPI, on a cluster or a single machine, you need so-called DP licenses. The number of DP licenses determines how many MPI processes can be used for QuantumATK parallelization. For more information on running QuantumATK in parallel, see the *Parallel Guide*.

Basic License Troubleshooting

If you have a very general license problem, here are some tips that might lead to actually generating one of those error messages, or solve the issue in a simple way. It will also be much easier for us to help you troubleshoot the license problem if you go through these steps first and let us know the result.

Automated License Setup

The easiest way to manage the license configuration is to use Virtual NanoLab (VNL), which has a built-in tool both for setting up the license environment and for changing it. This tool will either pop up when you start VNL and no license can be found, or you can start it from the Help menu. For more information, see the detailed guide here. If you to inspect the license setup in more detail, look in the file licenses.licconf, located in the directory .quantumwise, inside your home directory. This directory also holds any nodelocked or demo license you have, in the licenses subdirectory.

Manual License Setup

In some cases, for instance on a cluster, it is not possible or convenient to use VNL to configure the license. In this case, the easiest option is to use environment variables, as discussed here. To see how the license environment is currently set up, you can run the following commands in a terminal on Linux, or on Windows in Cygwin or similar:

```
echo $QUANTUM_LICENSE_PATH
echo $QUANTUM_AUTOMATIC_SERVER_DISCOVERY
```

or on Windows (in a regular Windows cmd terminal):

```
echo %QUANTUM_LICENSE_PATH%
echo %QUANTUM_AUTOMATIC_SERVER_DISCOVERY%
```

If you think that the license variables are set correctly, you can check what licenses are available on the system by using (on Windows or Linux):

```
lmxendutil -licstat
```

This will report where QuantumATK looks for licenses and what it found, and is particularly useful for checking **floating licenses** (when you are using a license server).

Some Further Particular Tips

If you have a **standalone license** file, check that this file is readable by the current user, by using the following command:

```
cat $QUANTUM_LICENSE_PATH     (Linux/Cygwin)
type %QUANTUM_LICENSE_PATH%   (Windows cmd)
```

If this doesnt produce the expected output (which is, that it prints out the contents of the license file), try

```
ls -l $QUANTUM_LICENSE_PATH    (Linux/Cygwin)
dir %QUANTUM_LICENSE_PATH%     (Windows cmd)
```

This will show if the license file actually exists in the specified location. If it does, but you cant see its contents - check the permissions of the file, maybe only root/Administrator can read it? A lot of useful information about the license setup can be found in the Installation Guide and the LM-X End-User Guide.

Advanced License Troubleshooting

When troubleshooting license problems, a very convenient tool is to define the environment variable **QUANTUM_EXTENDEDLOG=license.log**, then start QuantumATK, and afterwards inspect the file license.log which will contain a detailed account of all license-related system settings, and very importantly how LM-X/ATK tried to locate a license (file or server) and whether or not it was successful. For example, on **Linux** (or Windows/Cygwin), open a terminal and type (this syntax is for bash, see Notes below for tcsh)

```
QUANTUM_EXTENDEDLOG=license.log atkpython nothing.py
```

The file nothing.py need not exist, its just a placeholder for this test. Now, if you get a message saying

```
can't open file 'nothing.py': [Errno 2] No such file or directory
```

your license is actually working properly. However, if you see

```
License Error: (Internal: 719 Feature: ATKPython)
(Err: 9) Feature not found
```

you need to check license.log for more information. Make sure to provide this file to QuantumWise support when asking for help about license issues. On Windows, the corresponding approach would be to press WinKey+R and type cmd (or use the Start Menu), and then in the terminal window which opens type

```
set QUANTUM_EXTENDEDLOG=license.log
atkpython nothing.py
```

Note: The file license.log will be created in the current directory, so make a note of where that is, so you can locate the file afterwards. The test above only checks that atkpython can be started, and for that you only need the feature ATKPython in the license file. If you can launch atkpython and/or VNL without problems, but have license troubles when running a particular calculation, try running the calculation again with the environment variable set globally (in your login file .bashrc on Linux, or via the Control Panel on Windows). Note that on Linux, you cannot use this approach if you start VNL from the Desktop icon, unless you first log out and back in again. Its easier to run vnl from a terminal window for the test purpose. If your shell is csh/tcsh, the syntax is

```
env QUANTUM_LICENSE_FILE=license.log atkpython nothing.py
```

Maintenance, Updates and Support

As a customer of QuantumWise, you have access to support via email (and some other benefits, like participation in user meetings etc) provided that you hold an active maintenance contract.

Note: Always state your customer ID when communicating with support staff, so that we can identify your affiliation.

- For **annual licenses**, the maintenance period normally coincides with the validity of the license itself, unless otherwise specified.
- For **perpetual licenses**, the maintenance period is specified separately on your contract/invoice. When it expires, the contract can be prolonged for continued access to support, as well as upgrades.

Updates and Upgrades

If you hold a perpetual license, you can always update QuantumATK if a new minor version is released (typically bug fixes) free of charge, even if your maintenance period has expired. Minor versions are for instance 2016.1 and 2016.2, which are updates to 2016.0.

When a new major version of QuantumATK is released, you may upgrade free of charge if the new version is already included in your contract. You can check your license file or invoice to find out which version you may upgrade to; if the license file contains a feature label 17.0 you can upgrade to version 2017 and all updates (2017.1, etc) to it.

If you have an older version in your contract, you can purchase an upgrade as part of the maintenance (for perpetual licenses) or by prolonging your annual license.

License and QuantumATK Upgrading

Whether or not you can upgrade for free to a newer version of QuantumATK depends on which version is included in your license file, and correspondingly in your contract. Either check your latest invoice, or start QuantumATK and open Help>About, where the highest support version is written out as e.g. 17.0, in which case you are able to upgrade to version 2017 (if it says 16.0 or 16.5 or anything else with 16, your maximum version is 2016, and so on).

You can also inspect the license file (its just a text file). The highest supported version is written in the file, on the VENDOR line, for instance as

```
VENDOR=QUANTUM VERSION=17.0 END=2017-08-14 SHARE=VIRTUAL|TERMINALSERVER
```

If the new version is supported by your license, you can just download and install it. If your license does not support the new version, please contact us for upgrade options

Updating the License File

If you receive an updated **standalone license** file, for instance a prolonged demo/trial license, you dont need to reinstall the software. All you need to do is run the **license configuration tool** provided with QuantumATK, as described in detail in the Installation Guide. If you are running a **license server** and receive a new license file, the location of the license file should be obvious from the way the server is started. In this case, just replace the license file and restart the license server.

License Server Cannot Be Started

If the license server refuses to start on Linux, check the output of

```
ldd lmx-serv-quantum
```

If you see something like

```
./lmx-serv-quantum: /lib64/tls/libc.so.6: version 'GLIBC_2.4' not found (required by ./
↪lmx-serv-quantum)
libpthread.so.0 => /lib64/tls/libpthread.so.0 (0x0000003ca4700000)
libm.so.6 => /lib64/tls/libm.so.6 (0x0000003ca3f00000)
libc.so.6 => /lib64/tls/libc.so.6 (0x0000003ca3c00000)
/lib64/ld-linux-x86-64.so.2 (0x0000003ca3800000)
```

it means that this machine can not be used to run the license server, since it requires GLIC 2.4 or newer, and this machine uses GLIBC 2.3 (or lower).

SSH License Server Access

Normally ATK will locate active license servers automatically, as long as you are on the same network segment and there are no firewalls between your computer and the server. If there are firewalls which cant be opened for whatever reason, or you wish to access the license server remotely, e.g. from home or while traveling, a very convenient possibility is to set up an SSH tunnel to the license server. To make this work, you need:

•An SSH client. On Linux you will have this for sure, on Windows you can consider using Cygwin with OpenSSH, PuTTy, or another 3rd party SSH application. Below you will find instructions for different options.

•The hostname or IP address of the license server - ask your system administrator if you are unsure.

•An account on a machine which is able to connect to the license server, and to which you can login over SSH. This may be the license server itself, but it can really be any machine which can run ATK using the license server (like a cluster headnode, your office desktop, etc).

Below we will denote

•the hostname of the license server as LICSERV

•the machine you SSH into, which can run QuantumATK, as HOST

•your username on this machine as USER

All these should of course be replaced with the actual hostnames (e.g. server2.uni.edu) or IP number, and username.

Note: Make sure you verify first that HOST can indeed use the license server, else it may be hard to troubleshoot problems further on.

Step 1: Open the SSH Tunnel

Linux / Windows with Cygwin Open a terminal window and type

```
ssh -vvv -N -L 6200:LICSERV:6200 USER@HOST
```

Lots of information will be printed to the screen. After you have provided your password, the license tunnel should be open - proceed to step 2. The process will run in the foreground, it will be waiting for active connections. You will not, however, get a prompt on the remote host (due to the option -N). You need to keep this window open as long as you want the license tunnel to work.

Windows with PuTTY

1.Download and install PuTTY from here. The easiest option is to choose the Windows installer (putty-0.62-installer.exe, or a newer version if available).

2.Start PuTTY (if you cant find a Start Menu entry for it, its in C:Program Files (x86)PuTTYPUTTY.EXE).

3.Under Host Name, enter **HOST**. Replace HOST by the actual IP address or hostname of the computer you have an account on.

4.Then, in the *Category* tree in the left-hand panel, click the + next to SSH to expand the options. Go to Tunnels. - Make sure **Local** is selected. - Source port: **6200** - Destination: **LICSERV:6200** (replace LICSERV by actual license server hostname) - Click **Add**.

5.Do not click Open yet! Instead, go back to the first page (scroll up in the Category tree to find Session). Enter a name for the session, for instance License Tunnel and click Save.

6.Now click **Open** to initiate the connection. After you have provided your username and password, the license tunnel should be open - proceed to step 2.

Note: Under Connection>Data you can set your username so you dont have to type it in each time. Dont forget to Save the session profile. Next time you want to open the license tunnel: - Start PuTTY - Select the License Tunnel session and click Load - Then click Open You can also load this profile directly by providing the command line argument -load License Tunnel to putty.exe.

Step 2: Client License Environment Configuration

1. Start the QuantumATK license configuration (as described in the Installation Guide); the easiest way is to open a new terminal and type **atk_license_config**, or on Windows press **WinKey+R** and type **atk_license_config**.
2. Click the Next and OK buttons until you get to the page with license options
3. Select to use a **Floating license**, click Next, and on the next page set the server to **localhost** (the port number will be automatically added, do not enter it).
4. Complete the configuration utility to save the changes by clicking Next>Next>Finish.
5. Proceed to step 3 to test the license tunnel.

   Once you are able to connect to the license server directly again (say, when you return from your travel), you must rerun this configuration utility to change the configuration back to the original state, which most likely is Floating but with the server setting left empty, for autoconfiguration.

   If you are familiar with environment variables, you can also just type in the terminal

```
export QUANTUM_LICENSE_PATH=6200@localhost
```

   In this case the setting only applies to the current session (to this terminal), but it saves you from reconfiguring the license environment when you have direct access to the license server again. You must of course start both vnl and atkpython from this terminal to have access to the license.

Step 3: Testing

To test that the license configuration works, try to run (in a terminal):

```
atkpython nothing.py
```

If the license checkout is **successful**, there will be an error message like

```
can't open file 'nothing.py': [Errno 2] No such file or directory
```

Thats ok - the file does indeed not exist, but the point was to check whether QuantumATK manages to find the license or not, and this message only appears if it could.

If the license checkout in unsuccessful, you will instead see a license error message

```
License Error: (Internal: 719 Feature: ATKPython)
(Err: 9) Feature not found
```

In this case, first make sure - again - that the HOST computer can run QuantumATK and that there are licenses available, and then try to look for messages in the SSH window; the -vv verbosity option will print lots of debug messages that will be helpful for troubleshooting.

Even in the successful case, you may want to check the SSH messages, especially in case you notice that it takes a rather long time before the message about nothing.py appears. This may indicate that instead of actually using the license tunnel, QuantumATK is using the 2 week built-in trial license. In the truly successful case, you should see a message of the following form in the SSH window:

```
debug1: Connection to port 6200 forwarding to multiserver port 6200 requested.
debug1: channel 2: new [direct-tcpip]
debug1: channel 2: free: direct-tcpip: listening port 6200 for multiserver port 6200,␣
↪connect from ::1 port 50269, nchannels 3
```

Note: The license tunnel is only open as long as the SSH process is running. If the network connection times out, or you put the computer to sleep or reboot, you just run the same command again. - It is possible to put the license tunnel into the background by adding the -f option. This saves you from having an extra terminal open for this process, but its also harder to restart the tunnel if the network connection is broken, since you must first kill the initial process, since it binds port 6200. Thus we recommend keeping the tunnel running in the foreground unless you are very experienced with SSH tunnels.

If you have a HAL (high-availability license) solution with 3 redundant license servers, you need some special considerations. You need to set up a tunnel to the first two HAL servers (the 3rd one cannot be used to check out a license from anyway), on separate local ports (for instance 6200 and 6201), and then set the environment variable manually, in .bashrc (Linux/Cygwin) or via the Control Panel (Windows) as follows:

```
QUANTUM_LICENSE_PATH=6200@localhost;6201@localhost    (on Windows)
QUANTUM_LICENSE_PATH=6200@localhost:6201@localhost    (on Linux)
SmallBestOriginalRemove
```

Restricting Access to a License Server

By using a configuration file, it is possible to restrict access to the license features to certain computers, users, etc. The access is controlled using ALLOW and DENY statements, a bit like in a .htaccess file. The rules work a bit differently, however, so below we will present some simple examples to make it clearer.

The essential logic of the access restrictions is:

- You can ALLOW or DENY access to the license features all, or a single, or multiple specified users, IP numbers, hosts.
- The rules can apply to all license features, or only some specific ones.
- The rules are evaluated in the order they appear in the configuration file. The first matching rule is applied - the following rules are generally ignored.
- If no rule matches, the default is to ALLOW.

Point 4 is very important, because it means that if you wish to restrict access to the license in any way, you must explicitly ALLOW those with access and then DENY all others. So, in order to only allow computers in the IP number range 192.168.0.0 to 192.168.0.255 access to the license, you will need to use the statements

```
ALLOW_IPADDR_ALL = 192.168.0.*
DENY_IPADDR_ALL = *
```

Another typical option is to only allow a particular user to access the license. You can then use

```
ALLOW_USER_ALL = niceguy1 niceguy2
DENY_USER_ALL = *
```

Point 3 is also crucial - if we reverse the ALLOW/DENY statements in the two examples above, nobody will have access to the license server because everybody will match the DENY rule and thus be rejected. You only want the DENY rule to apply to those not matching the ALLOW rule!

There are multiple options for the ALLOW/DENY statements - for more details please see the extensive examples inside the configuration file **lmx-serv.cfg** which resides in the same directory as the license server binary. For example, you may just want to block a particular user, then its enough to use

```
DENY_USER_ALL = badguy
```

In this case all other users will be allowed by virtual of Point 4 above. To use the configuration file the license server must be started with the option **-c lmx-serv.cfg**.

Note: In general you should always use this option, and make sure to change the server password in the file from the default value.

1.2 Introductory Tutorials

Introductory tutorials below will help you get started and enable you to set up and analyze your first calculations using the **QuantumATK NanoLab graphical user interface**.

1.2.1 Geometry Optimization

Version: X-2025.06

In this tutorial, you will learn how to use QuantumATK NanoLab GUI to

- Find and use structures from the NanoLab Internal Database.
- Setup a basic geometry optimization simulation for SiO_2 (quartz) using Density Functional Theory (DFT).
- Inspect the optimized geometry.

All the workflow, input and output files can be found in the **Example Project - X-2025.06 → SiO2-quartz-geom-opt folder.**

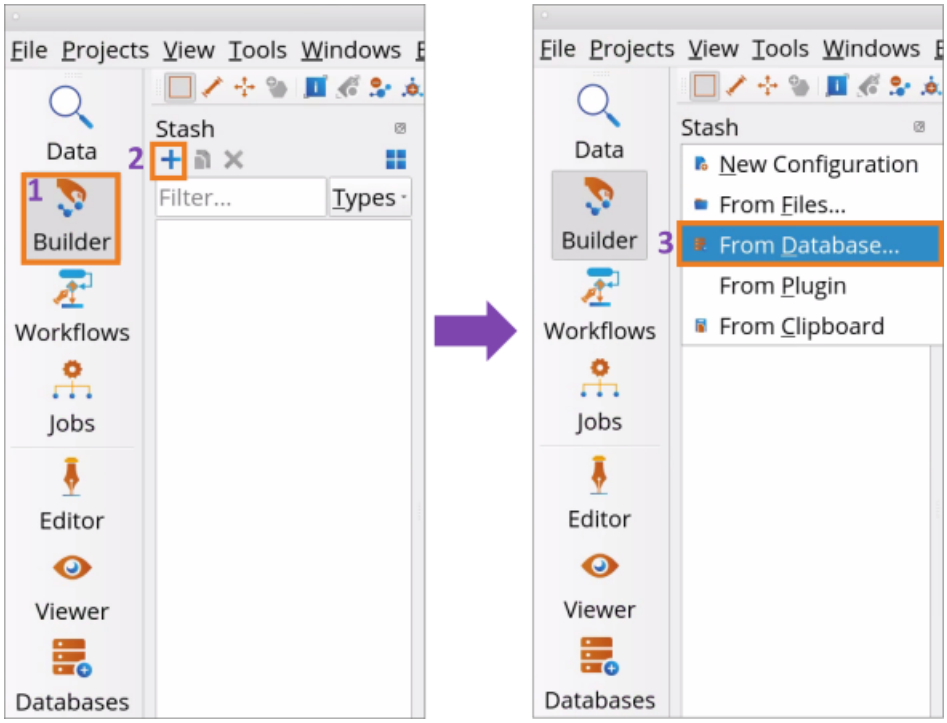
Prerequisites

- Video Introduction to QuantumATK NanoLab GUI
- Video tutorial on the Workflow Builder tool
- Video tutorial on the Job tool

Importing Structures from the NanoLab Database

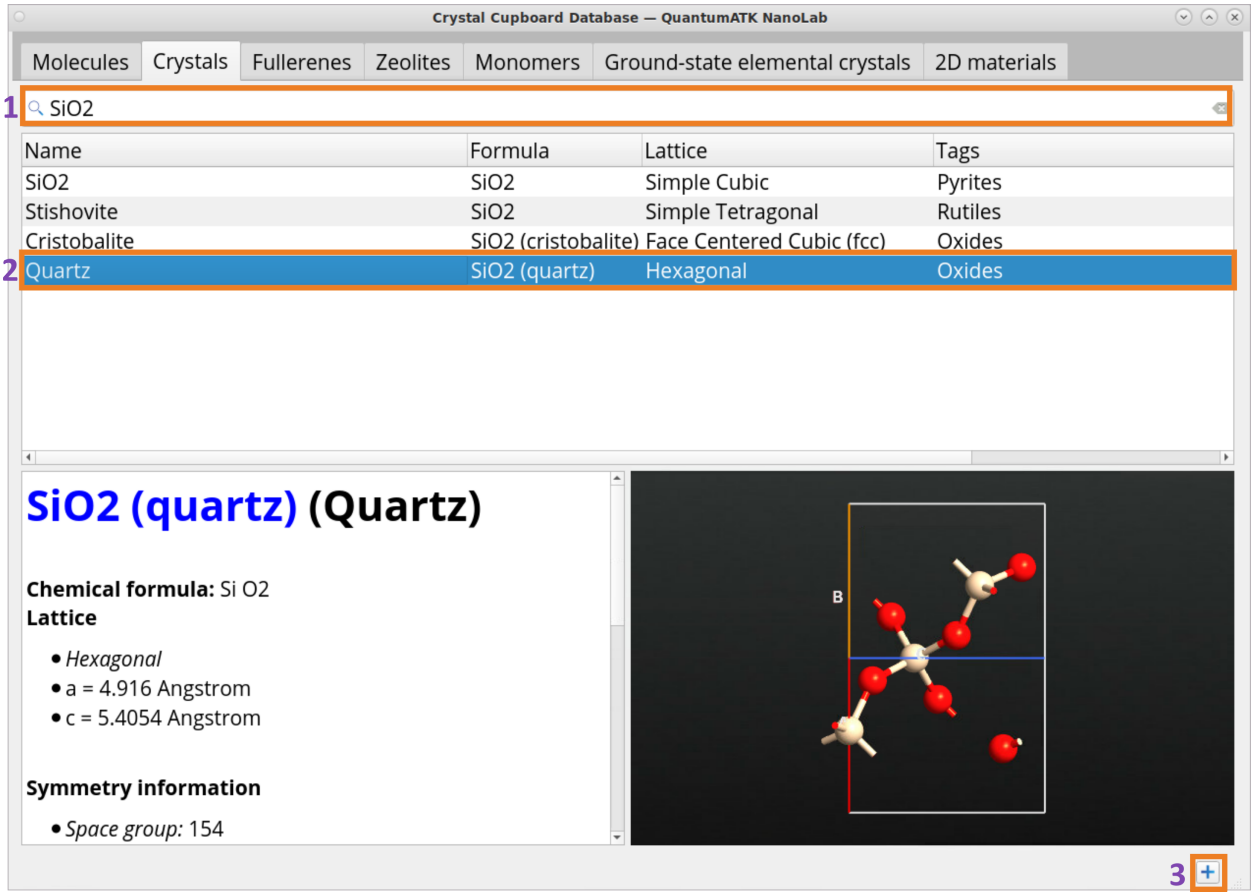
To import a SiO_2 (quartz) structure from the NanoLabs Internal Database, follow the step-by-step protocol described below.

1. In the **Builder** *Stash* (1), click on the *Add* button (2) and choose *From Database* (3) to open NanoLabs *Crystal Database*.



2. Use the *Search* bar (1) to find SiO_2 (quartz) structure (2) and click on the *Add* button (3) to add the structure to the **Builder** *Stash*.

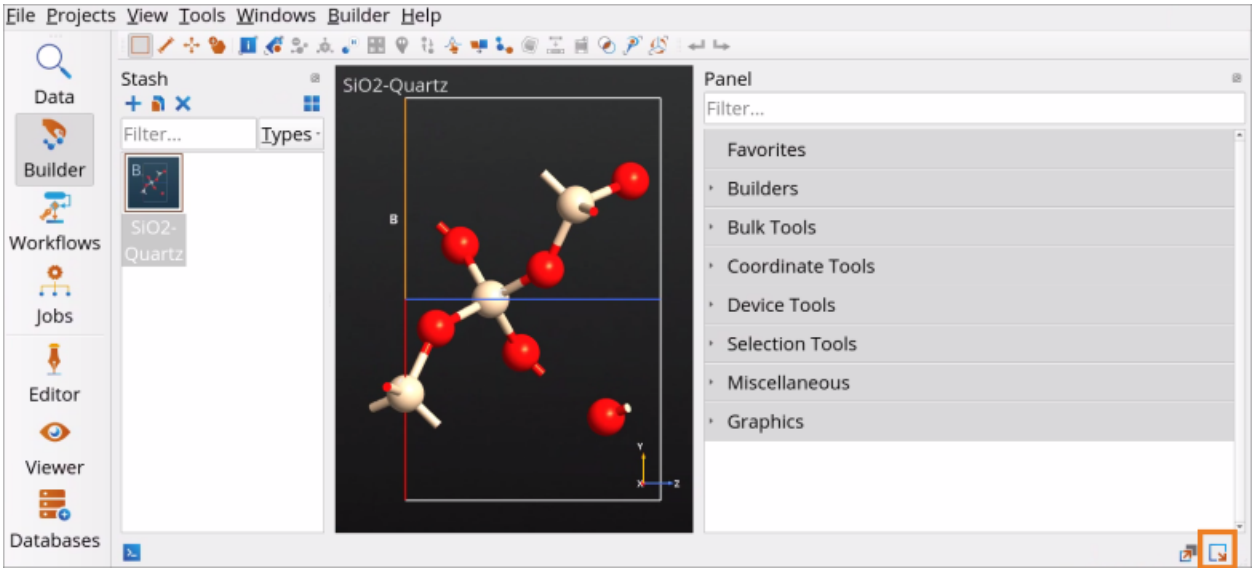
Tip: Internal NanoLab structure library contains several hundred basic molecule, crystal, fullerene, zeolite, monomer and 2D material structures. If you cannot find there a structure of your interest, click on **Databases tool** to use NanoLabs interface to query online databases such as Crystallography Online Database and Materials Project.


Crystal Cupboard Database — QuantumATK NanoLab
Molecules
Crystals
Fullerenes
Zeolites
Monomers
Ground-state elemental crystals
2D materials
1
SiO2
Name
Formula
Lattice
Tags
SiO2
SiO2
Simple Cubic
Pyrites
Stishovite
SiO2
Simple Tetragonal
Rutiles
Cristobalite
SiO2 (cristobalite)
Face Centered Cubic (fcc)
Oxides
2
Quartz
SiO2 (quartz)
Hexagonal
Oxides
SiO2 (quartz) (Quartz)
Chemical formula: Si O2
Lattice
Hexagonal
a = 4.916 Angstrom
c = 5.4054 Angstrom
Symmetry information
Space group: 154
B
3

Setting up Geometry Optimization Workflow

To create a workflow for *OptimizeGeometry* calculations in the **Workflow Builder**, follow the step-by-step protocol described below. Here, we will use DFT with *LCAOCalculator* basis sets and a hybrid HSE06 DFT functional, as it is optimized for bulk semiconductor and insulator materials.

1. In the **Builder**, use the *Send content to other tools* button to send the imported SiO_2 (quartz) structure to the **Workflow Builder**.



2. Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* category on the right hand panel and drag-and-drop the *SetLCAOCalculator* block to the middle *Workflow Area* → *Build* tab.

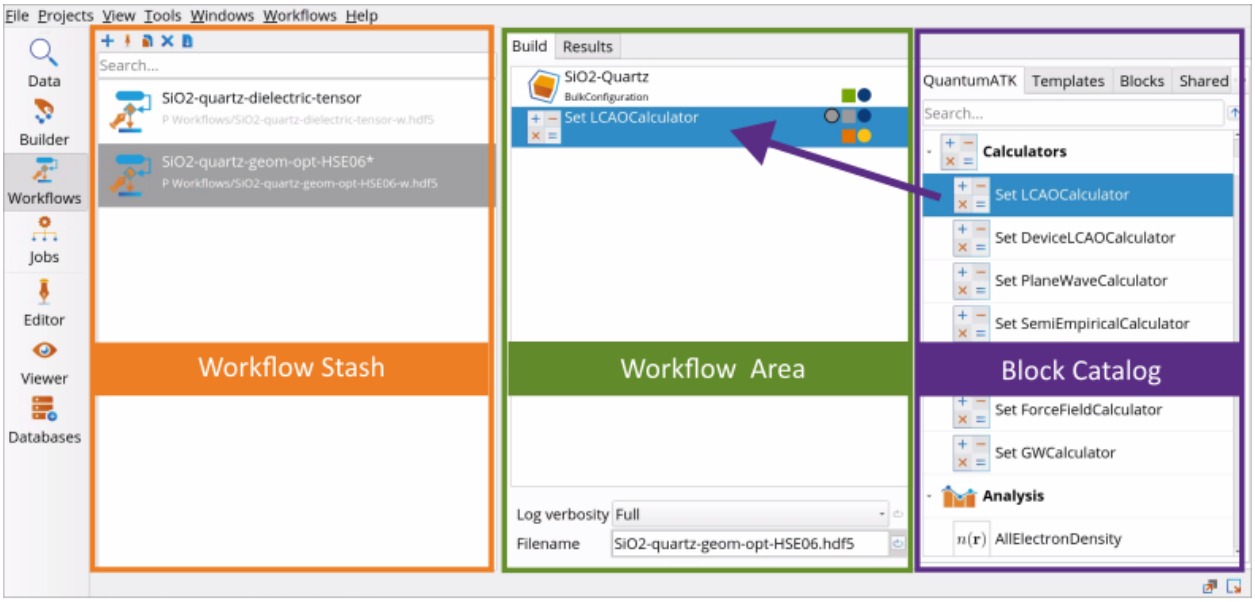
3. Double-click on the *SetLCAOCalculator* block and edit the *Main* tab:

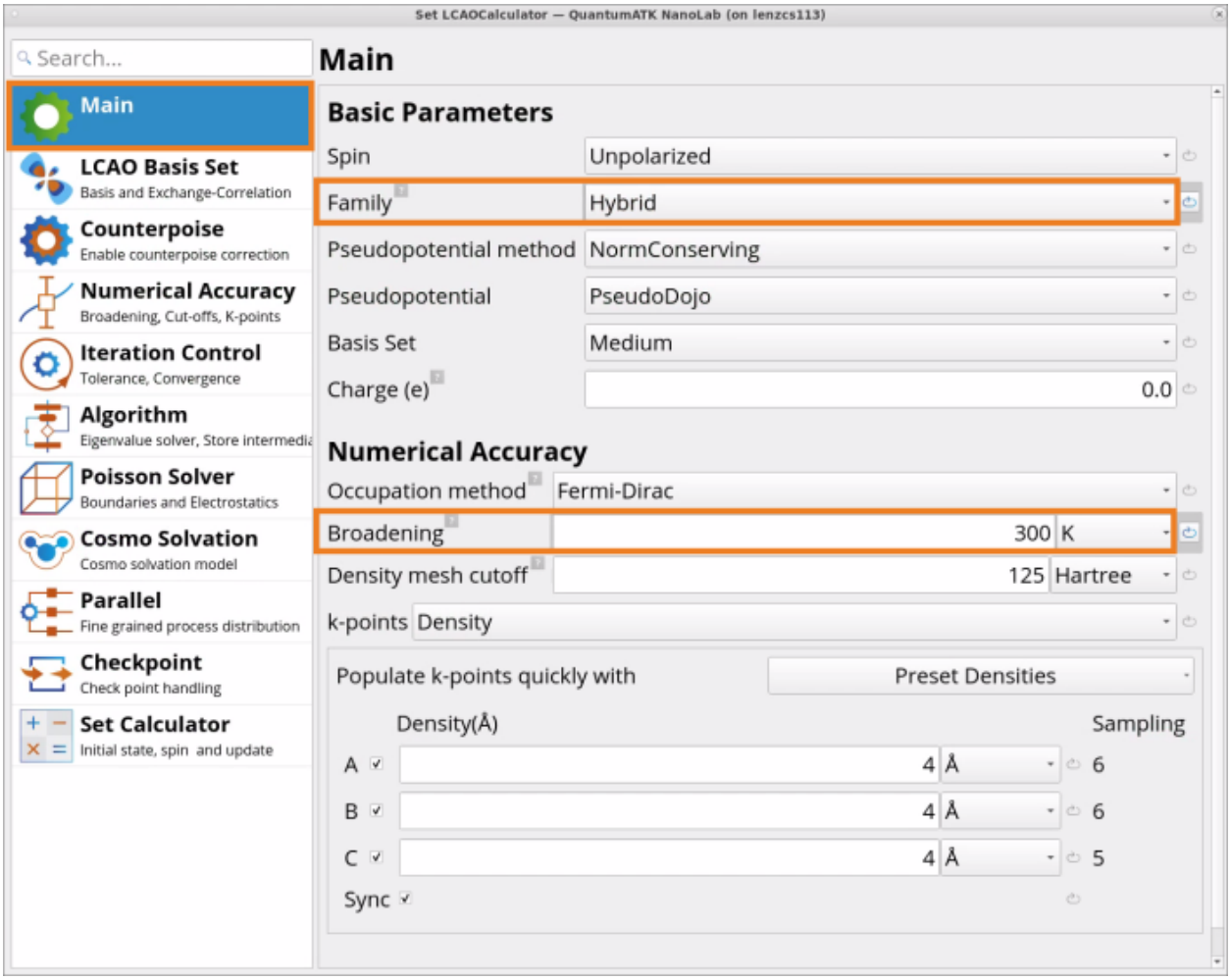
- Change the *Exchange Correlation Family* from the default *GGA* to *Hybrid* with a predefined default *HSE06 (HybridGGA.HSE06)* functional (functional can be changed under the *LCAO Basis Set* tab).
- Change the numerical accuracy paremeter *Broadening* from 1000 K to 300 K (recommended for semiconductor materials).

Tip:

1. *SetLCAOCalculator* block sets medium k-point sampling, medium basis set and 125 Ha density mesh cutoff (system-dependent) by default, typically providing good accuracy results with modest computational resources. It is recommended though to check the convergence of your results with these parameters.

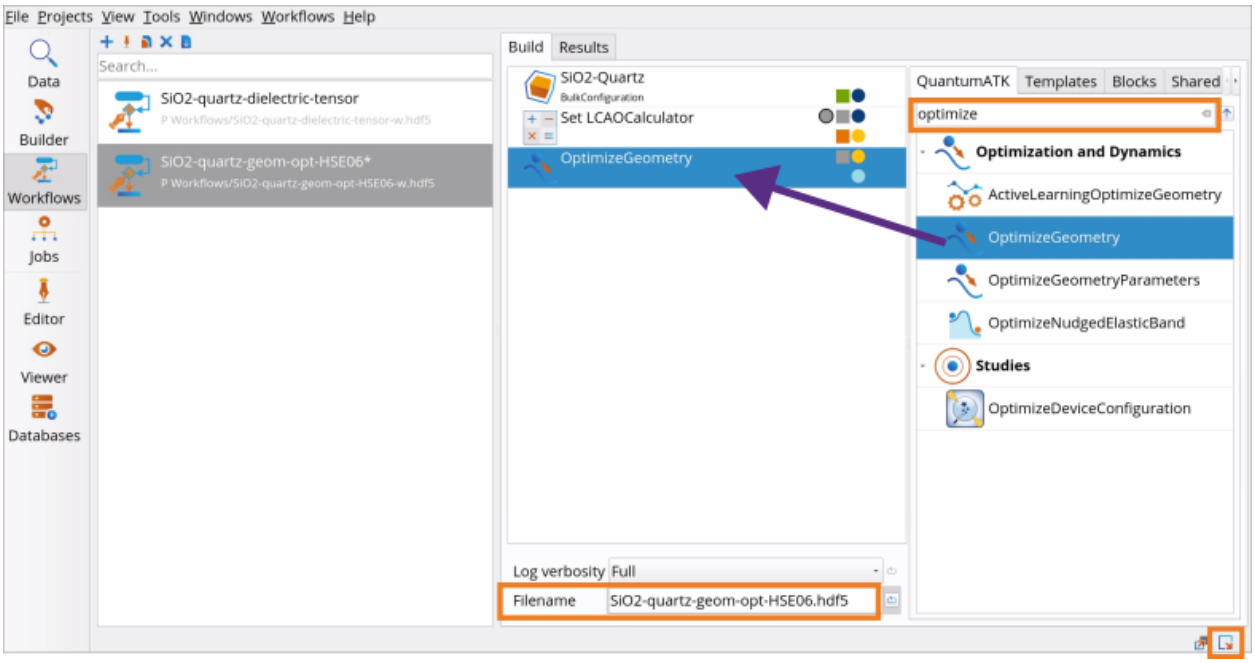
2. *SetLCAOCalculator* block uses a default *Fermi-Dirac Occupation Method* with 1000 K *Broadening*, which generally works reasonably well across various systems. Another setting may be more appropriate for your system, and you can adjust these settings according to the recommendations in *Occupation Methods* for semiconductor, insulator, metal, and molecule systems.


File Projects View Tools Windows Workflows Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Databases
Search...
SiO2-quartz-dielectric-tensor
SiO2-quartz-geom-opt-HSE06*
Workflow Stash
Build
Results
SiO2-Quartz
Set LCAOCalculator
Log verbosity Full
Filename SiO2-quartz-geom-opt-HSE06.hdf5
Workflow Area
QuantumATK
Templates
Blocks
Shared
Calculators
Set LCAOCalculator
Set DeviceLCAOCalculator
Set PlaneWaveCalculator
Set SemiEmpiricalCalculator
Block Catalog
Set ForceFieldCalculator
Set GWCalculator
Analysis
AllElectronDensity


Set LCAOCalculator — QuantumATK NanoLab (on lenzcs113)
Search...
Main
LCAO Basis Set
Basis and Exchange-Correlation
Counterpoise
Enable counterpoise correction
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Algorithm
Poisson Solver
Boundaries and Electrostatics
Cosmo Solvation
Cosmo solvation model
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
Main
Basic Parameters
Spin Unpolarized
Family Hybrid
Pseudopotential method NormConserving
Pseudopotential PseudoDojo
Basis Set Medium
Charge (e) 0.0
Numerical Accuracy
Occupation method Fermi-Dirac
Broadening 300 K
Density mesh cutoff 125 Hartree
k-points Density
Populate k-points quickly with Preset Densities
Density(Å)
Sampling
A 4 Å 6
B 4 Å 6
C 4 Å 5
Sync

3. For slabs, surfaces and interfaces (modelled as slabs), it is recommended to set *FFT2D* Poisson Solver (*Dirichlet/Neumann*) in the out-of-plane (C) direction instead of the default *FFT* (*Periodic*) in the *SetLCAOCalculator* block, *Poisson Solver/Solver Type* section. For neutral molecules, stick to the default *FFT* Poisson solver, whereas for charged molecules, set the *Multigrid/Multipole* Poisson Solver. Read these technical notes on *Poisson solvers* to learn more.

4. Go to the *Block Catalog* → *QuantumATK* tab → *Optimization and Dynamics* category on the right hand panel and drag-and-drop the *OptimizeGeometry* block to the middle *Workflow Area* → *Build* tab.



5. Double-click on the *OptimizeGeometry* block and change from *Constrain Bravais Lattice* to *Constrain space group*.

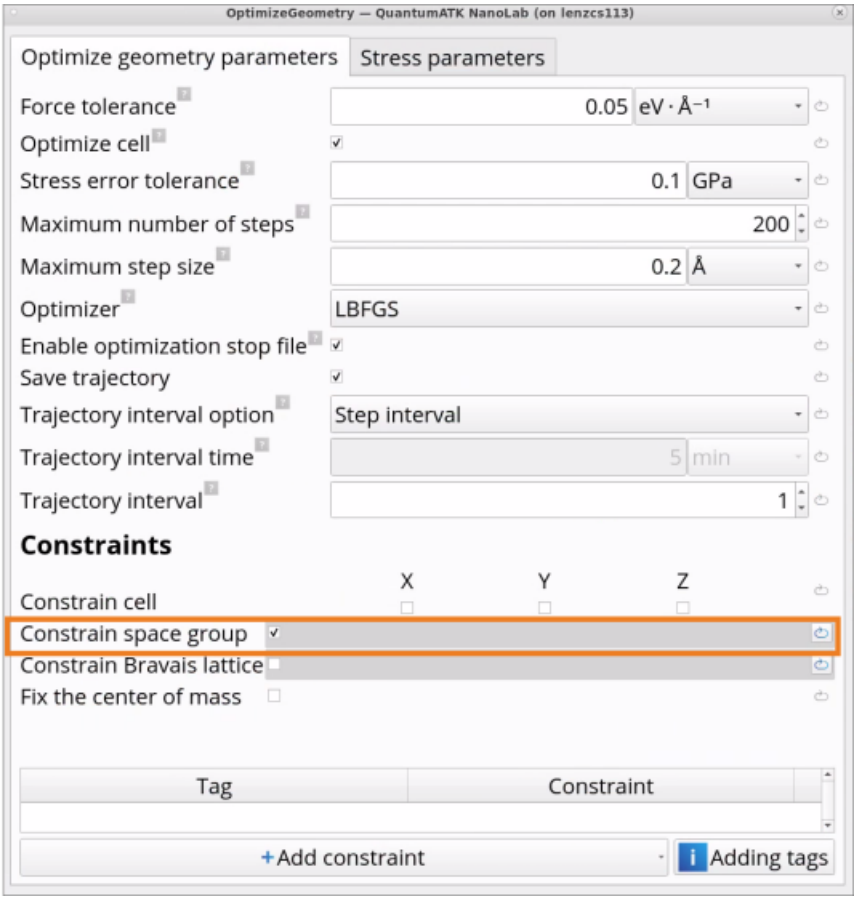
6. Change the *Workflow* name in the *Workflow Stash* and *Filename* as needed (e.g., `SiO2-geom-opt-HSE06.hdf5`).

7. Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool in order to submit this job on a local machine or a remote cluster. It is recommended to parallelize over both MPI processes and threads in DFT simulations. Read the technical notes on *Parallelization of QuantumATK calculations* to learn more.

8. This also generates a Python script `SiO2-geom-opt-HSE06.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool.

Tip:

1. *Constrain space group* relaxes atom positions, unit cell volume and shape and preserves crystal symmetry during the geometry optimization, whereas the default *Constrain Bravais Lattice* allows to relax to a different crystal symmetry. *Constrain Bravais lattice* can also be used when optimizing the geometry of alloys, amorphous materials or crystals with defects with a preserved Bravais lattice type.


OptimizeGeometry — QuantumATK NanoLab (on lenzcs113)
Optimize geometry parameters
Stress parameters
Force tolerance
0.05
eV · Å⁻¹
Optimize cell
Stress error tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer
LBFGS
Enable optimization stop file
Save trajectory
Trajectory interval option
Step interval
Trajectory interval time
5
min
Trajectory interval
1
Constraints
X
Y
Z
Constrain cell
Constrain space group
Constrain Bravais lattice
Fix the center of mass
Tag
Constraint
+Add constraint
Adding tags

2. Unit cell volume and shape can be constrained by unchecking the box *Optimize cell* in the *OptimizeGeometry* block, thus allowing for local relaxation of atoms only.

3. When doing geometry optimization of a bare surface, or, e.g., surface covered by molecules, it is possible to *Add constraints* on selected (tagged) atoms or monolayers in the *OptimizeGeometry* block to, e.g., fix the bottom substrate layer.

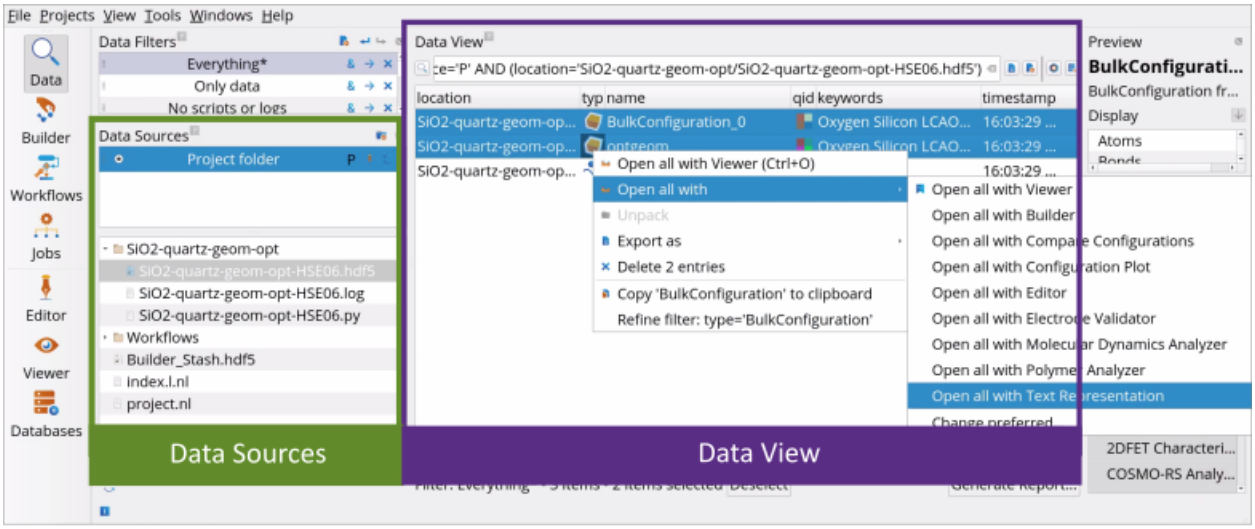
4. In case of calculating *PhononBandstructure*, *VibrationalMode*, *DielectricTensor* (or other properties that require *DynamicalMatrix* calculations), it is recommended to pre-optimize structure with a tighter force (e.g., 0.001) and stress (e.g., 0.01) tolerances than default.

Analyzing Results

Once the calculation has finished, we are ready to analyze the results.

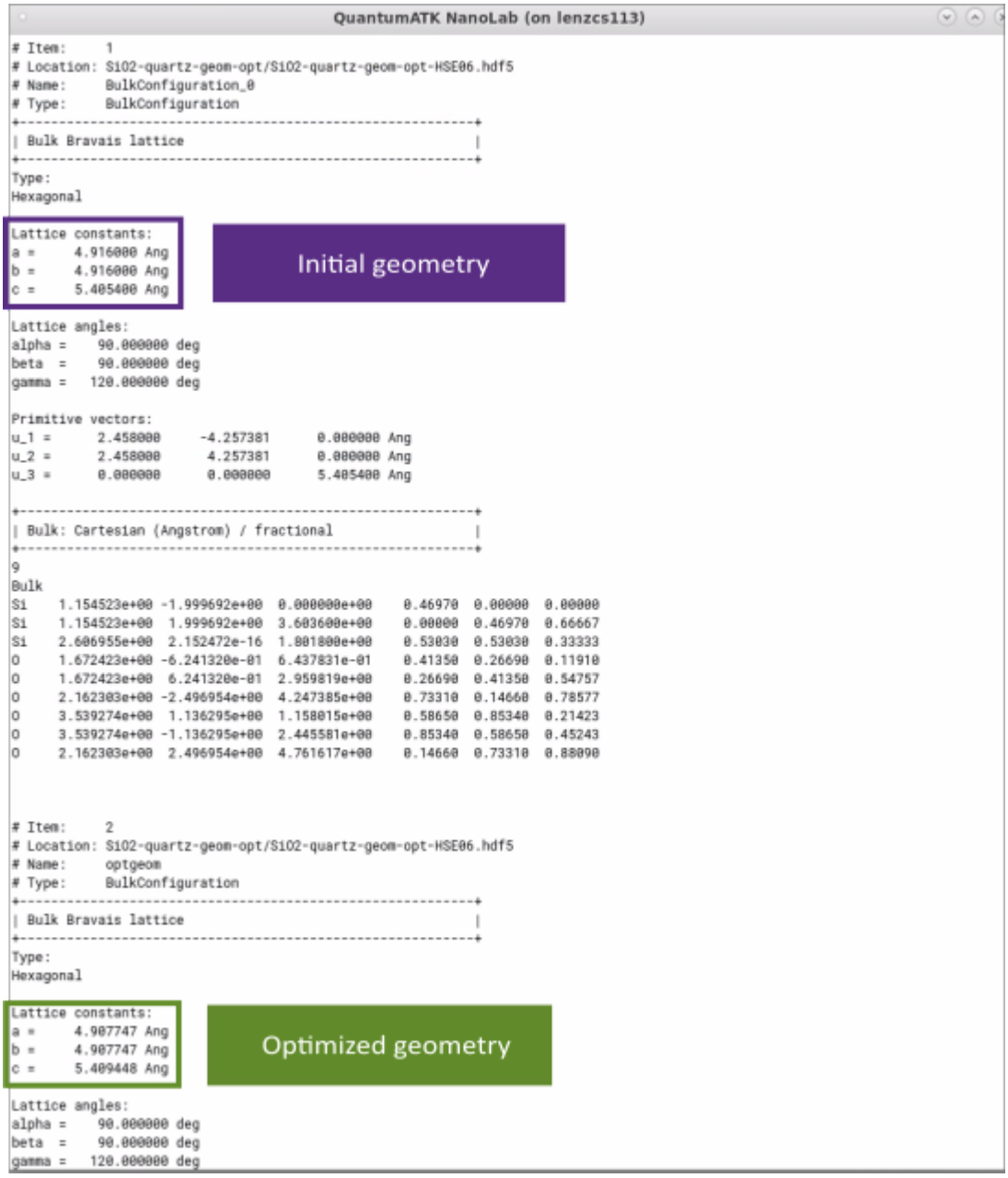
1. Go to the **Data View** tool.

2. In the *Data Sources* select the file `SiO2-geom-opt-HSE06.hdf5` and then in the *Data View* window select both, *BulkConfiguration_0* (initial geometry) and *optgeom* (optimized geometry), followed by the right-click to *Open All with Text Representation*.



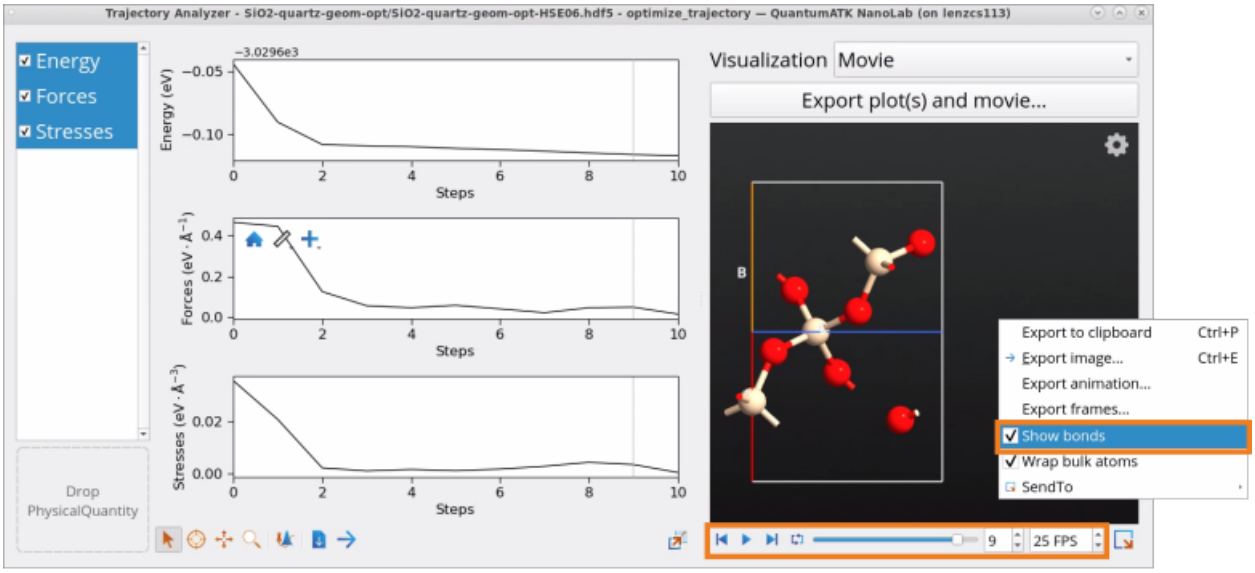
3. Inspect and compare lattice constants of initial and optimized geometries in text representation. The optimized lattice constants of SiO_2 (quartz) , a=4.908 Å and c= 5.409 Å, with the hybrid HSE06 functional are in a good agreement with experimental results a=4.913 Å and c=5.405 Å[1].

[1] R.ăW.ăG. Wyckoff and R.ăWyckoff, editors. *Crystal structures*. Interscience publishers New York, 1963.


QuantumATK NanoLab (on lenzcs113)
Item: 1
Location: SiO2-quartz-geom-opt/SiO2-quartz-geom-opt-HSE06.hdf5
Name: BulkConfiguration_0
Type: BulkConfiguration
| Bulk Bravais lattice |
Type:
Hexagonal
Lattice constants:
a = 4.916000 Ang
b = 4.916000 Ang
c = 5.405400 Ang
Initial geometry
Lattice angles:
alpha = 90.000000 deg
beta = 90.000000 deg
gamma = 120.000000 deg
Primitive vectors:
u_1 = 2.458000 -4.257381 0.000000 Ang
u_2 = 2.458000 4.257381 0.000000 Ang
u_3 = 0.000000 0.000000 5.405400 Ang
| Bulk: Cartesian (Angstrom) / fractional |
9
Bulk
Si 1.154523e+00 -1.999692e+00 0.000000e+00 0.46970 0.00000 0.00000
Si 1.154523e+00 1.999692e+00 3.603600e+00 0.00000 0.46970 0.66667
Si 2.606955e+00 2.152472e-16 1.801800e+00 0.53030 0.53030 0.33333
O 1.672423e+00 -6.241320e-01 6.437831e-01 0.41350 0.26690 0.11910
O 1.672423e+00 6.241320e-01 2.959819e+00 0.26690 0.41350 0.54757
O 2.162303e+00 -2.496954e+00 4.247385e+00 0.73310 0.14660 0.78577
O 3.539274e+00 1.136295e+00 1.158015e+00 0.58650 0.85340 0.21423
O 3.539274e+00 -1.136295e+00 2.445581e+00 0.85340 0.58650 0.45243
O 2.162303e+00 2.496954e+00 4.761617e+00 0.14660 0.73310 0.88090
Item: 2
Location: SiO2-quartz-geom-opt/SiO2-quartz-geom-opt-HSE06.hdf5
Name: optgeom
Type: BulkConfiguration
| Bulk Bravais lattice |
Type:
Hexagonal
Lattice constants:
a = 4.907747 Ang
b = 4.907747 Ang
c = 5.409448 Ang
Optimized geometry
Lattice angles:
alpha = 90.000000 deg
beta = 90.000000 deg
gamma = 120.000000 deg

4. Right-click on the *optimize_trajectory* object in the *Data View* window and open it with the *Movie Tool* to investigate the convergence of energy, forces and stresses throughout the simulation. It took 10 steps to converge energy, forces and stresses to the default force and stress tolerance values.

5. Right-click on the structure in the *Movie tool* to check the box *Show bonds* and use the *Play* button to see the evolution of structure throughout the geometry optimization simulation.



6. Right-click on the the optimized geometry *optgeom* in *Data View* and choose to *Open with Builder*. The optimized geometry will be automatically saved in the **Builder** *Stash*, so you could use it for other simulations, such as *Band Structure, Projected Density of States and Effective Mass Calculations*, *Optical Property Calculations*, etc.

Note: Note that lattice parameters optimized with the default, computationally-cheaper GGA PBE functional instead, a=4.966 Å and c=5.466 Å, deviate by ~1% from the measured values, compared to ~0.1% when optimized with the more sophisticated hybrid HSE06 functional used in this tutorial example for SiO_2 (quartz).

In the next tutorial on *Band Structure, Projected Density of States and Effective Mass Calculations* you will learn how to set up and analyze band structure, projected density of states and effective mass calculations for the optimized structures of SiO_2 (quartz) and Si.

References

1.2.2 Band Structure, Projected Density of States and Effective Mass Calculations

Version: X-2025.06

In this tutorial, you will learn how to use the QuantumATK NanoLab GUI to

- Set-up band structure, projected density of states (PDOS) and effective mass calculations for bulk Si and SiO_2 (quartz) using Density Functional Theory (DFT).
- Analyze obtained results.

All the workflow, input and output files can be found in the **Example Project - X-2025.06** → **SiO2-quartz-bandstructure-pdos** and **Si-bandstructure-effective-masses folders.**

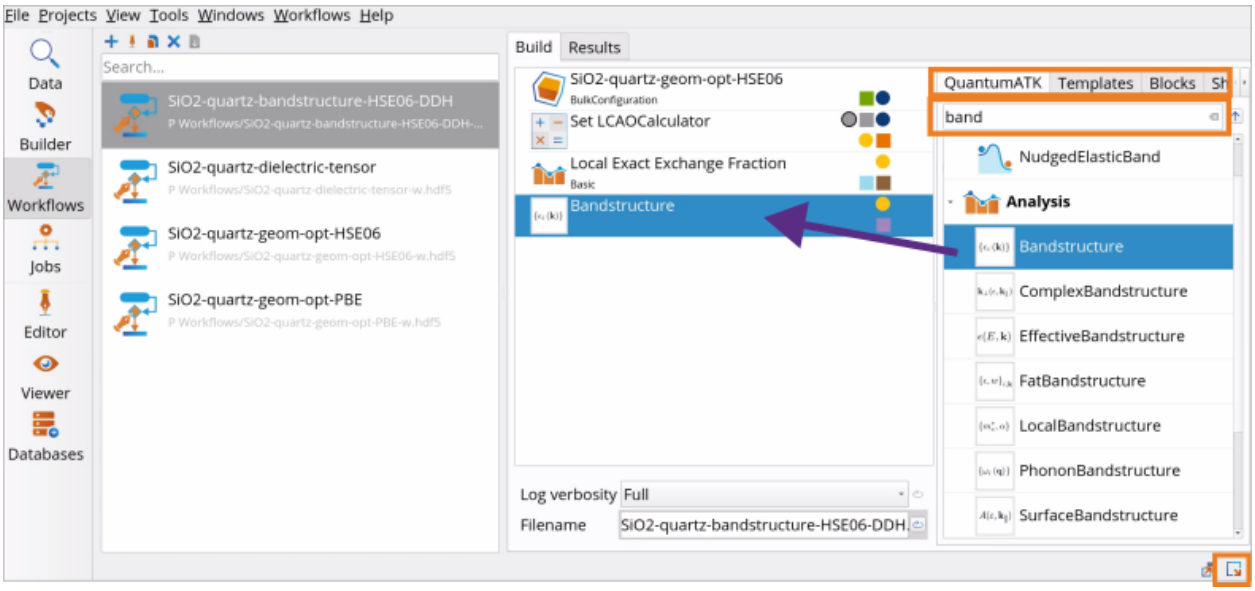
Prerequisites

- Video Introduction to QuantumATK NanoLab GUI
- Video tutorial on the Workflow Builder tool
- Video tutorial on the Job tool
- Tutorial *Geometry Optimization*

Setting up Band Structure Calculation Workflow

To create a workflow for *Bandstructure* and *ProjectedDensityOfStates* calculations in the **Workflow Builder**, follow the step-by-step protocol described below. Here, you will use DFT with *LCAOCalculator* basis sets. For the DFT functionals, choose a hybrid HSE06 DFT functional for bulk Si (exact exchange fraction of 0.25 optimized for Si) and a dielectric dependent hybrid HSE06-DDH functional (*DDHCustomExchangeCorrelation*) for SiO_2 (quartz) to find an improved material specific exact exchange fraction for more accurate band structure and band gap.

1. Follow the *Geometry Optimization* tutorial to optimize the structure of bulk Si and SiO_2 (quartz).
2. In the **Builder**, click on the *Send content to other tools* button to send the optimized structures from the **Builder** Stash to the **Workflow Builder**.
3. Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* and *Analysis* categories on the right hand panel and drag-and-drop the *SetLCAOCalculator* and *Bandstructure* blocks, respectively, to the middle *Workflow Area* → *Build* tab.
4. For SiO_2 (quartz) case, add a *LocalExactExchangeFraction* block from the *Block Catalog* → *QuantumATK* tab → *Analysis* category. It will return the calculated improved material specific exact exchange fraction for SiO_2 (quartz).
5. Double-click on the *SetLCAOCalculator* block and make the following changes:



•In the *LCAO Basis Set* tab of the *SetLCAOCalculator* block, change the *Exchange Correlation Family* from the default *GGA* to *Hybrid* with predefined *HSEO6 (HybridGGA.HSE06)* functional for Si and *HSE06DDH (Hybrid.GGA.HSEO6DDH)* for SiO_2 (quartz).

•In the *Numerical Accuracy* tab of the *SetLCAOCalculator* block, change the numerical accuracy paremeter *Broadening* from 1000 K to 300 K (recommended for semiconductor materials).

Tip:

1. *SetLCAOCalculator* block sets medium k-point sampling, medium basis set and 125 Ha density mesh cutoff (system-dependent) by default, typically providing good accuracy results with modest computational resources. It is recommended though to check the convergence of your results with these parameters.

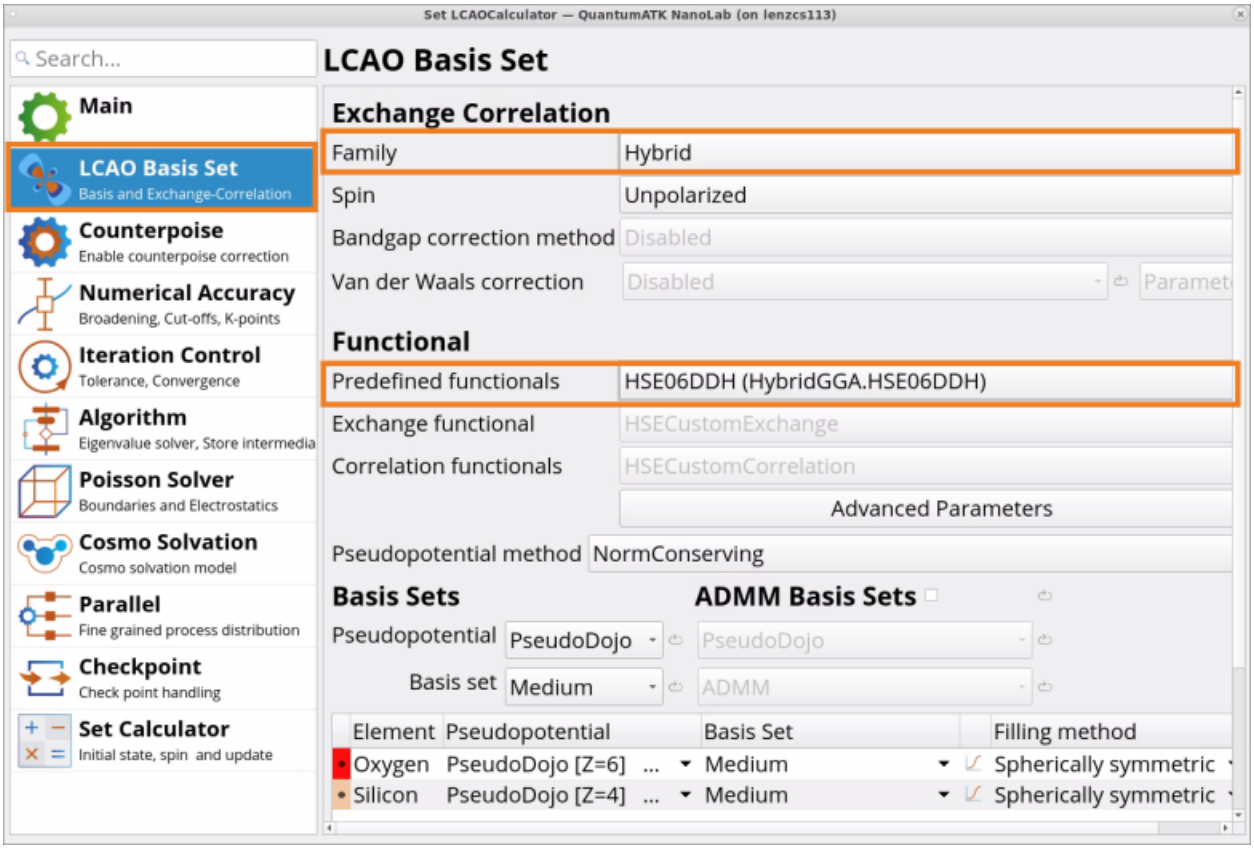
2. *SetLCAOCalculator* block uses a default *Fermi-Dirac Occupation Method* with 1000 K *Broadening*, which generally works reasonably well across various systems. Another setting may be more appropriate for your system, and you can adjust these settings according to the recommendations in *Occupation Methods* for semiconductor, insulator, metal, and molecule systems.

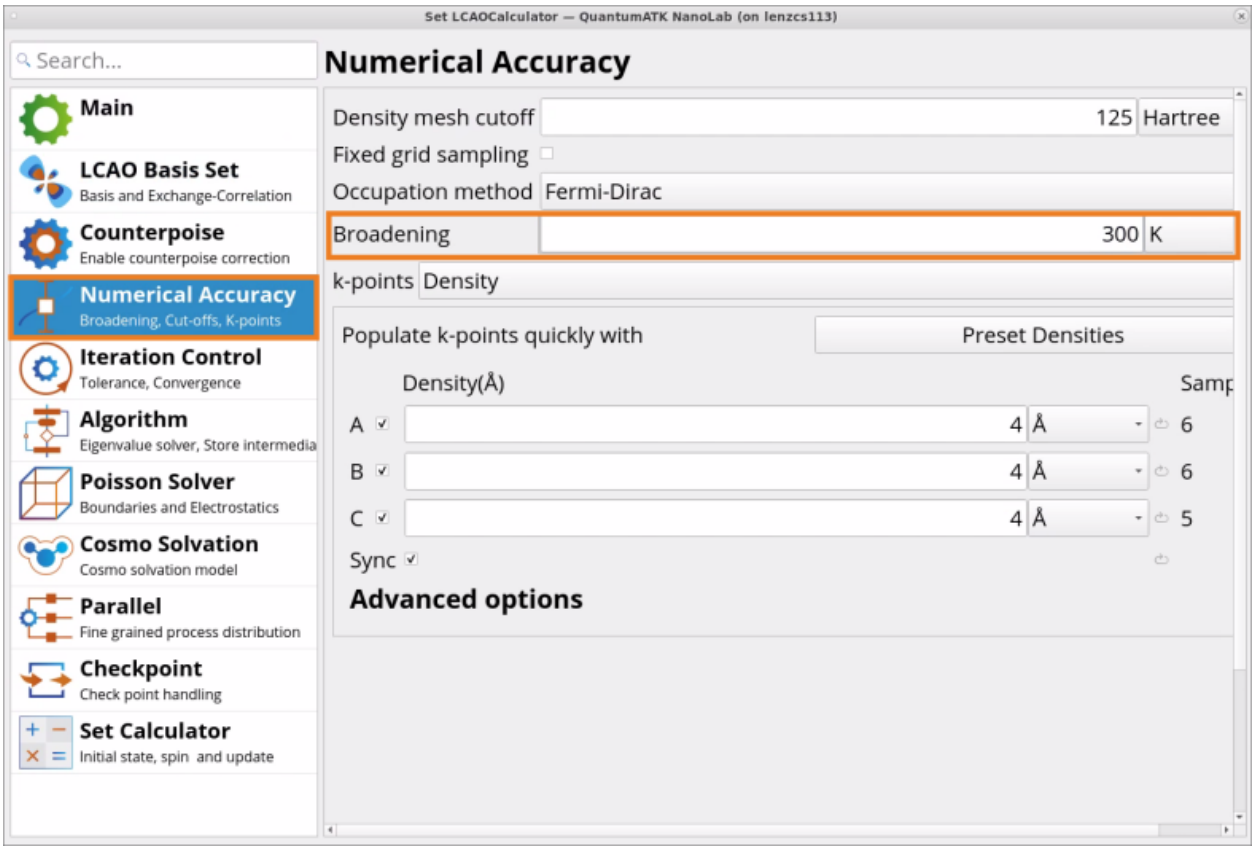
3. For slabs, surfaces and interfaces (modelled as slabs), it is recommended to set *FFT2D* Poisson Solver (*Dirichlet/Neumann*) in the out-of-plane (C) direction instead of the default *FFT* (*Periodic*) in the *SetLCAOCalculator* block, *Poisson Solver* section. For neutral molecules, stick to the default *FFT* Poisson solver, whereas for charged molecules, set the *Multigrid/Multipole* Poisson Solver. Read these technical notes on *Poisson solvers* to learn more.

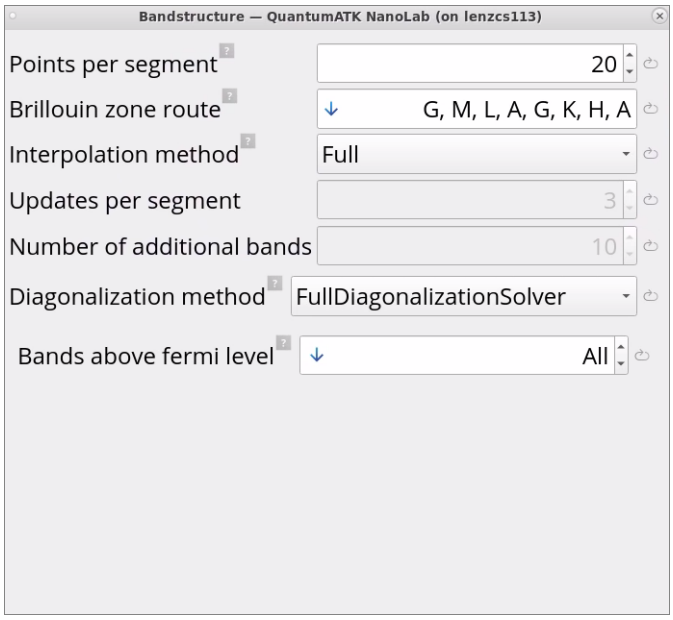
---

6. Double-click on the *Bandstructure* block. Here, the Brillouin zone route is automatically detected, and there is an option to modify it. There is also a possibility to increase the number of points per segment for more fine sampling of bandstructure. Stick to the default settings.

Tip:


Set LCAOCalculator — QuantumATK NanoLab (on lenzcs113)
Search...
Main
LCAO Basis Set
Basis and Exchange-Correlation
Counterpoise
Enable counterpoise correction
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Algorithm
Eigenvalue solver, Store intermedia
Poisson Solver
Boundaries and Electrostatics
Cosmo Solvation
Cosmo solvation model
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
LCAO Basis Set
Exchange Correlation
Family
Hybrid
Spin
Unpolarized
Bandgap correction method
Disabled
Van der Waals correction
Disabled
Functional
Predefined functionals
HSE06DDH (HybridGGA.HSE06DDH)
Exchange functional
HSECustomExchange
Correlation functionals
HSECustomCorrelation
Advanced Parameters
Pseudopotential method
NormConserving
Basis Sets
ADMM Basis Sets
Pseudopotential
PseudoDojo
PseudoDojo
Basis set
Medium
ADMM
Element
Pseudopotential
Basis Set
Filling method
Oxygen
PseudoDojo [Z=6]
Medium
Spherically symmetric
Silicon
PseudoDojo [Z=4]
Medium
Spherically symmetric


Set LCAOCalculator — QuantumATK NanoLab (on lenzcs113)
Search...
Main
LCAO Basis Set
Basis and Exchange-Correlation
Counterpoise
Enable counterpoise correction
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Algorithm
Eigenvalue solver, Store intermedia
Poisson Solver
Boundaries and Electrostatics
Cosmo Solvation
Cosmo solvation model
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
Numerical Accuracy
Density mesh cutoff
125
Hartree
Fixed grid sampling
Occupation method
Fermi-Dirac
Broadening
300
K
k-points
Density
Populate k-points quickly with
Preset Densities
Density(Å)
Samp
A
4
Å
6
B
4
Å
6
C
4
Å
5
Sync
Advanced options



1. When calculating band structure of very large systems (> 1000 atoms), there is an option to choose *IterativeDiagonalizationSolver* instead of *FullDiagonalizationSolver* in the *Bandstructure* block to speed-up simulations, in particular with Semi-Empirical Thigh-Binding models for more than order of magnitude faster calculations.

2. Perform *LocalBandstructure* calculations to get insights into the local properties of the band structure such as effective masses and band unparabolicity for specified k-point, bands and directions.

3. When setting up bandtructure calculations for low symmetry crystal or systems with atomic disorder (e.g., alloys), one needs to define k-routes in the Brillouin zone in a script *input.py*. For example, for SiGe alloy specify inequivalent X and L points, resulting in degeneracy lifting, i.e., alloy broadening of bands.

```
# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
def mysym():
    return {
            "G"   : numpy.array((0., 0., 0.)),
            "X1"  : numpy.array((1./2., 0., 0.)),
            "X2"  : numpy.array((0., 1./2., 0.)),
            "X3"  : numpy.array((0., 0., 1./2.)),
            "L1"  : numpy.array((1./2., 1./2., 1./2.)),
            "L2"  : numpy.array((-1./2., 1./2., 1./2.)),
            "L3"  : numpy.array((1./2., -1./2., 1./2.)),
            "L4"  : numpy.array((1./2., 1./2., -1./2.)),
        }

bulk_configuration.bravaisLattice().symmetryPoints = mysym

bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['X1', 'G', 'X2', 'G', 'X3', 'L1', 'G', 'L2', 'G', 'L3', 'G', 'L4'],
    points_per_segment=400,
    bands_above_fermi_level=All
    )

nlsave('SiGe_bandstructure.hdf5', bandstructure)
```

7. Change the *Workflow* name in the *Workflow Stash* and *Filename* as needed (e.g., `SiO2-quartz-bandstructure-HSE06-DDH.hdf5`).

8. Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool in order to submit this job on a local machine or a remote cluster. It is recommended to parallelize over both MPI processes and threads in DFT simulations. Read the technical notes on *Parallelization of QuantumATK calculations* to learn more.

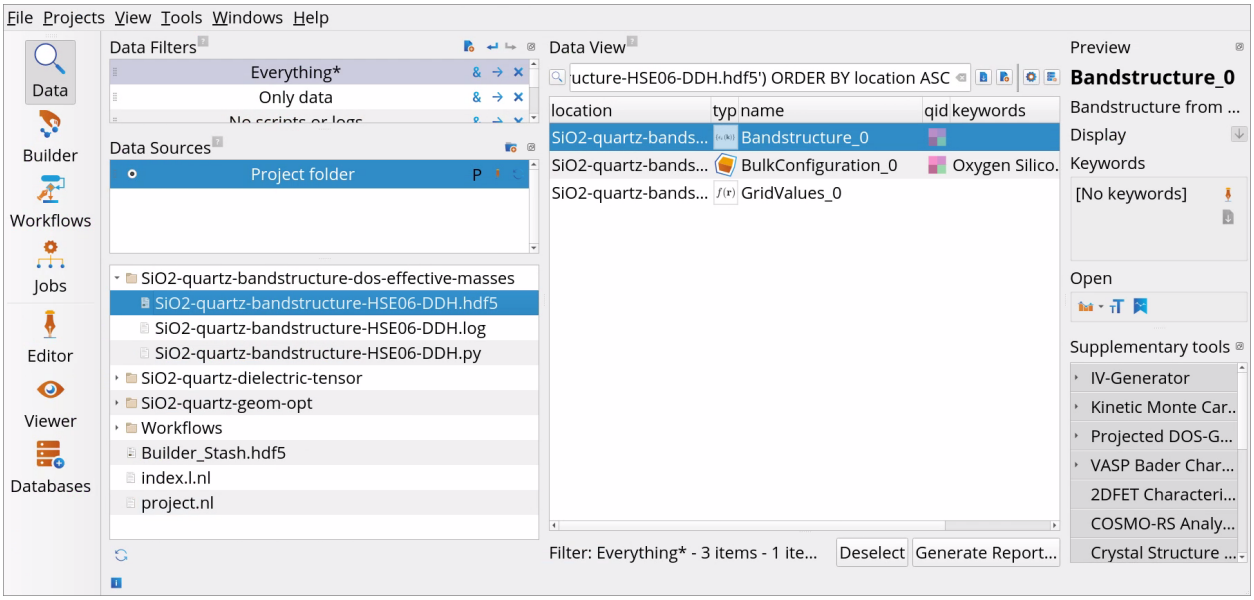
9. This also generates a Python script `SiO2-quartz-bandstructure-HSE06-DDH.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool.

Analyzing Band Structure Results

Once the calculation has finished, we are ready to analyze the results.

1. Go to the **Data View** tool.

2. In the *Data Sources* select the file `SiO2-quartz-bandstructure-HSE06-DDH.hdf5` and then in the *Data View* window double-click on the *Bandstructure_0* object to open it with the *Bandstructure Analyzer*.

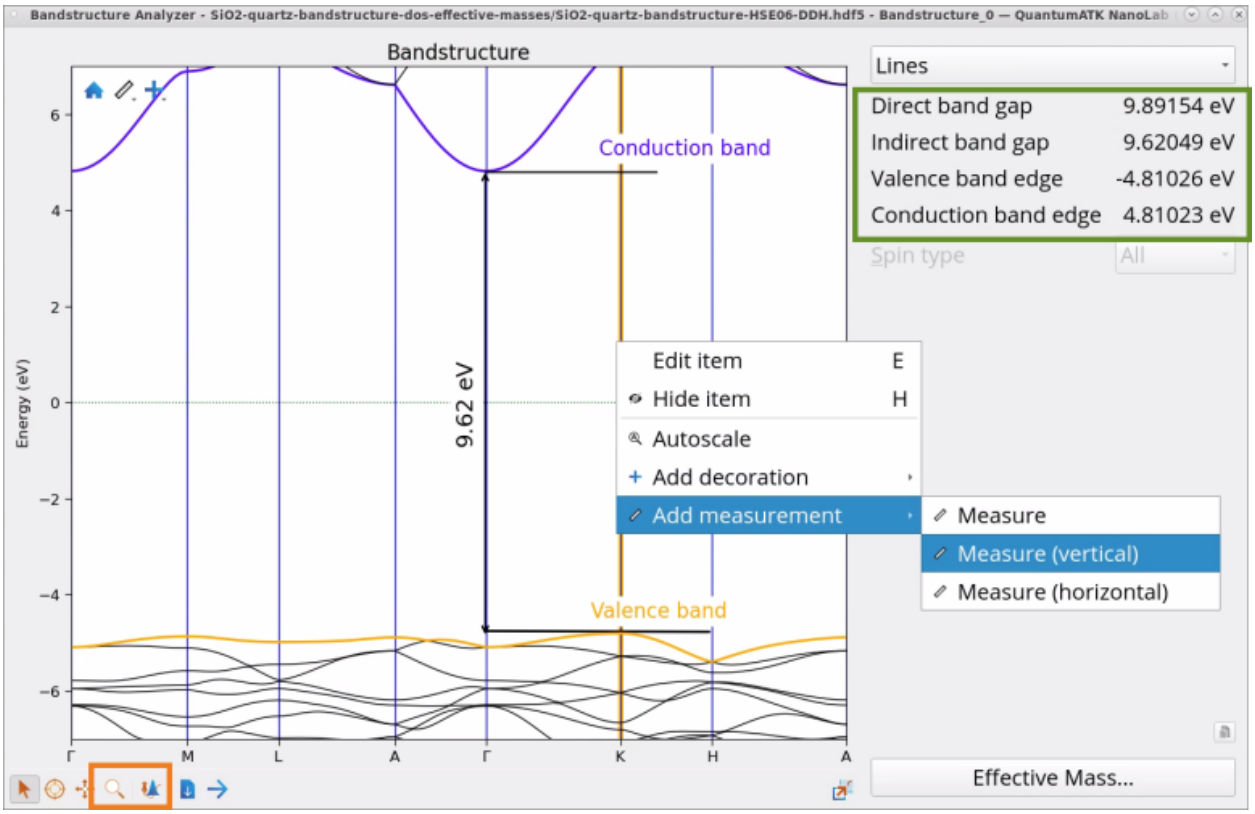


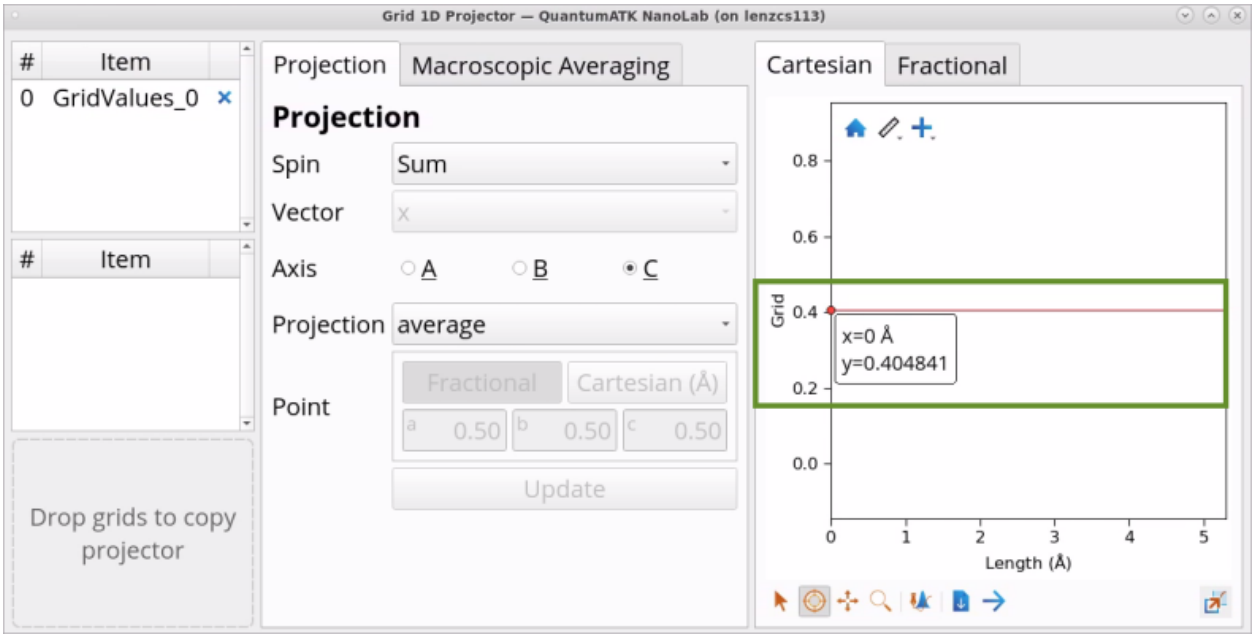
3. Examine *Direct band gap*, *Indirect band gap*, *Valence band edge* and *Conducion band edge* reported in the *Bandstructure Analyzer*.

4. Zoom into the band structure using *Zoom* and *Plot editor* tools and use measuring tools (right-click to *Add measurement*) to measure direct and indirect bandgaps yourself.

5. Double-click on *GridValues_0* object in *Data View* to open it with the *Grid 1D Projection Analyzer*. Examine the automatically calculated exact exchange parameter in HSE06-DDH, which is 0.4 for SiO_2 (quartz) - quite different from the pre-set 0.25 exact exchange parameter in the HSE06 functional.

Note:


Bandstructure Analyzer - SiO2-quartz-bandstructure-dos-effective-masses/SiO2-quartz-bandstructure-HSE06-DDH.hdf5 - Bandstructure_0 — QuantumATK NanoLab
Bandstructure
Lines
Direct band gap 9.89154 eV
Indirect band gap 9.62049 eV
Valence band edge -4.81026 eV
Conduction band edge 4.81023 eV
Spin type All
Conduction band
Valence band
9.62 eV
Energy (eV)
Edit item E
Hide item H
Autoscale
Add decoration
Add measurement
Measure
Measure (vertical)
Measure (horizontal)
Effective Mass...


Grid 1D Projector — QuantumATK NanoLab (on lenzcs113)
Item
0 GridValues_0
Drop grids to copy projector
Projection
Macroscopic Averaging
Projection
Spin Sum
Vector x
Axis A B C
Projection average
Point
Fractional
Cartesian (Å)
Update
Cartesian
Fractional
Grid
x=0 Å
y=0.404841
Length (Å)

1. Calculated indirect band gap of SiO_2 (quartz) 9.62 eV with the HSE06-DDH functional is in a good agreement with experimental (9.65 eV)[1] and many-body GW method (9.68 eV)[2] results. Indirect band gap obtained with HSE06 is 8.39 eV.

2. When simulating interfaces between different materials, choose *HSE06LocalDDH* for the predefined functional in the *SetLCAOCalculator* block, which automatically calculates material specific exact exchange fractions for each material in the interface.

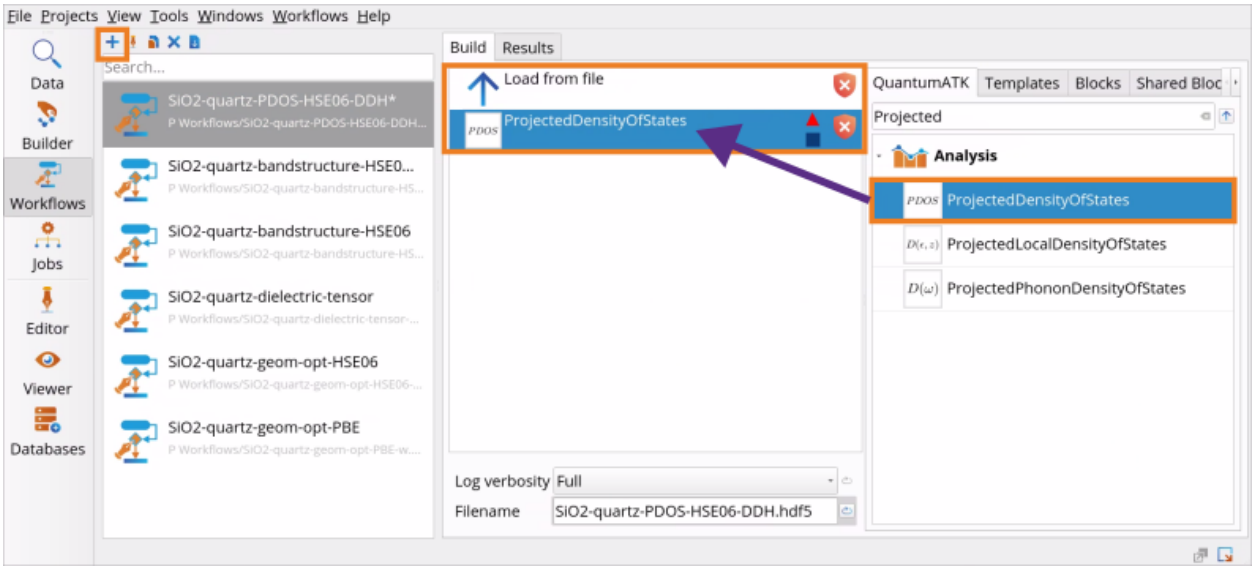
Restarting to Obtain Projected Density of States

In QuantumATK it is easy to restart from a previous simulation and calculate additional properties, for example, projected density of states, without re-doing the SCF convergence part. Follow this step-by-step protocol to set up additional projected density of state calculations.

1. Go back to the **Workflow Builder** and click on the the *Add* button to add a new workflow.

2. Change the *Workflow* name and *Filename* as needed (e.g., `SiO2-quartz-PDOS-HSE06-DDH.hdf5`).

3. Go to the *Block Catalog* → *QuantumATK* tab → *Algorithm* and *Analysis* categories on the right hand panel and drag-and-drop the *Load from file* and *ProjectedDensityOfStates* blocks, respectively, to the middle *Workflow Area* → *Build* tab.

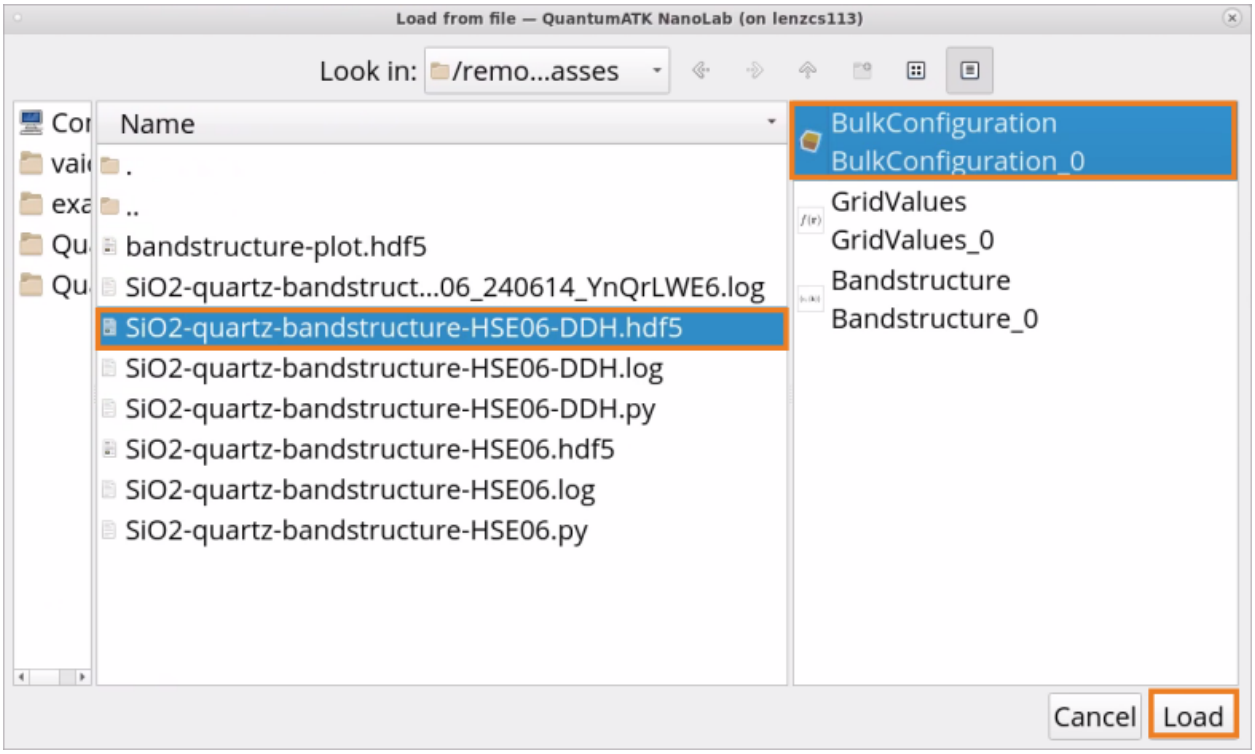


4. Double-click on the *Load from file* block to choose from which file to restart the calculation:

- Click on *Load from file*.
- Choose to restart from file `SiO2-quartz-bandstructure-HSE06-DDH.hdf5` → *BulkConfiguration*.
- Click *Load*.

[1] K.ăL. Milton, T.ăR. Durrant, T.ăCobos Freire, and A.ăL. Shluger. Difference in Structure and Electronic Properties of Oxygen Vacancies in -Quartz and -Cristobalite Phases of SiO2. *Materials*, 16(4):1382, 2023. doi:10.3390/ma16041382.

[2] T.ăMarkussen, R.ăFaber, P.ăA. Khomyakov, B.ăVerstichel, and A.ăBlom. Band Alignment in GAA Nanosheet Structures from Density Dependent Hybrid Functional and Many-Body GW Methods. *SISPAD*, 2024.



5.Double-click on the *ProjectedDensityOfSates* block and make the following edits:

- Define an *Energy range*: default -5 eV œ +5 eV is good for Si, but it should be increased for SiO_2 (quartz) to -15 eV œ +10 eV as it has a much larger band gap than Si.
- Choose *Projections* on *Elements and Shells* from various options, such as *Spin*, *Sites*, *Sites and Shells*, *Sites and Orbitals*, *Elements*, *Elements and Shells*, *Elements and Orbitals*.

Tip: Choose *Projections* on *Sites* to get local density of states / band diagram across interfaces, stacks or inhomogeneous systems in general.

6.Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool in order to submit this job on a local machine or a remote cluster.

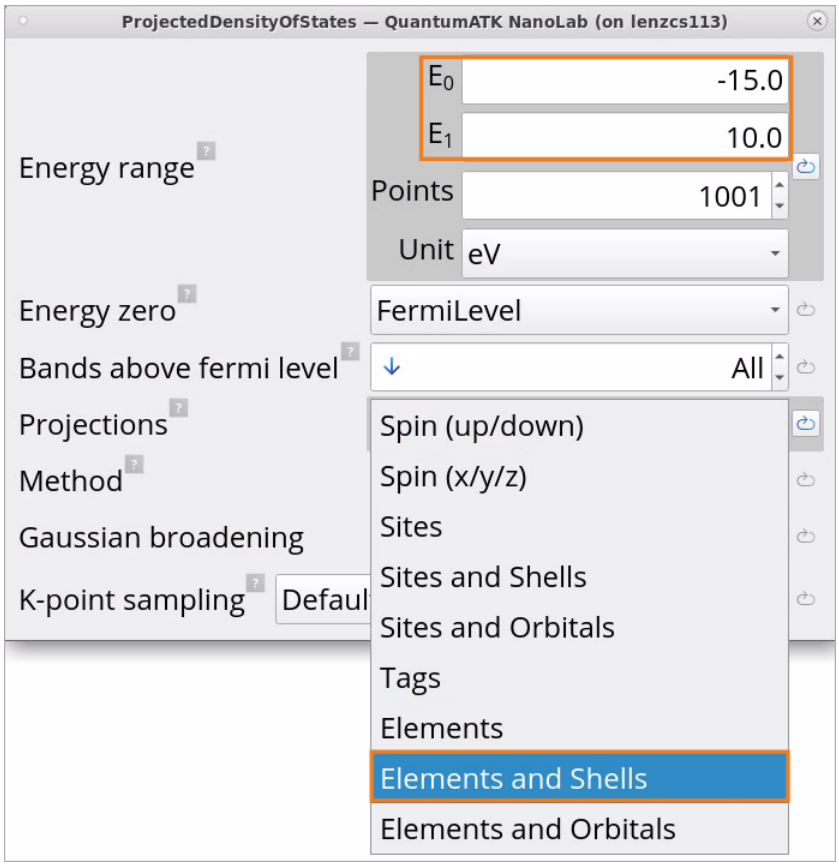
Analyzing Projected Density of State Results

Once the calculation has finished, we are ready to analyze the results.

1.Go to the **Data View** tool.

2.In the *Data Sources*, select the file `SiO2-quartz-PDOS-HSE06-DDH.hdf5` and then double-click on the *ProjectedDensityOfStates_0* object in the *Data View* window to open it with the *Projected DOS Analyzer*.

3.Use *Projected DOS Analyzer* to visualize and analyze Projected DOS on elements and shells in SiO_2 (quartz). As shown in the image below, valence bands are dominated by O *p* shell (l=1), whereas conduction bands are dominated by Si *p* shell (l=1).



4.Combine band structure and projected density of states plots for additional insights:

•Drag-and-drop handle from the *Projected DOS Analyzer* to the *Bandstructure Analyzer* plot to combine the two plots.

•Choose combination type: *Flipped and side-by-side with shared y axis*.

•Click *Merge*.

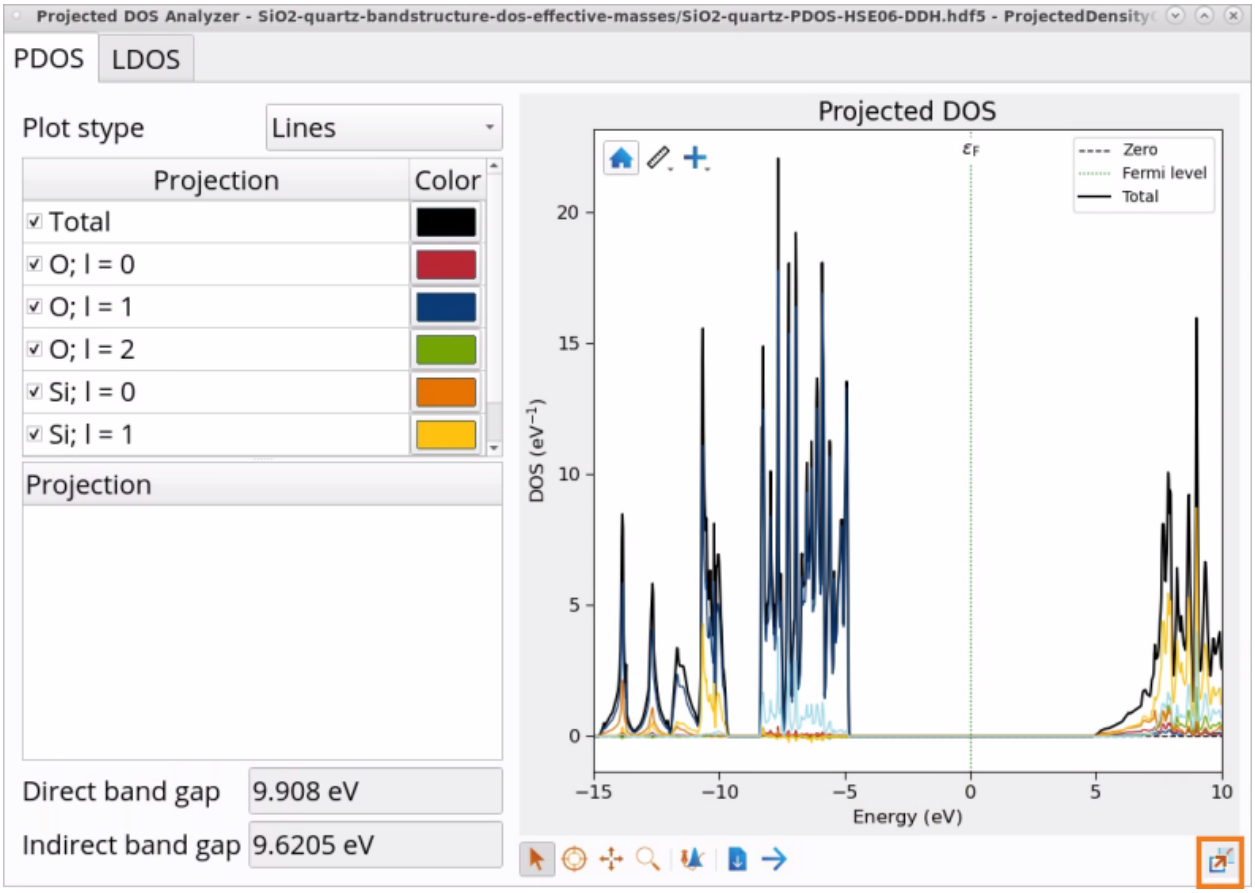
Calculation of Effective Masses

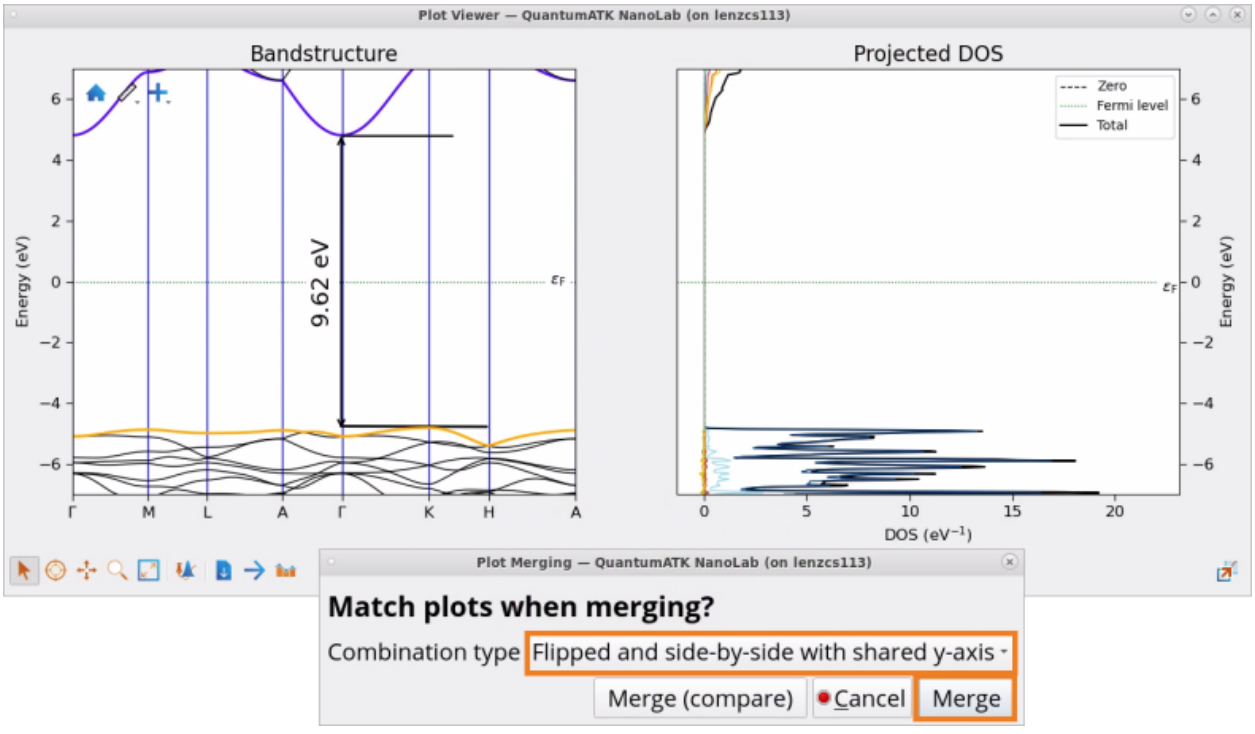
Lets see how to estimate effective mass of electrons in bulk Si from the band structure calculated with the hybrid HSE06 functional. The *EffectiveMass* of holes and electrons can be estimated by fitting a parabola to the minimum/maximum of the conduction/valence bands.

1.Go to the **Data View** tool.

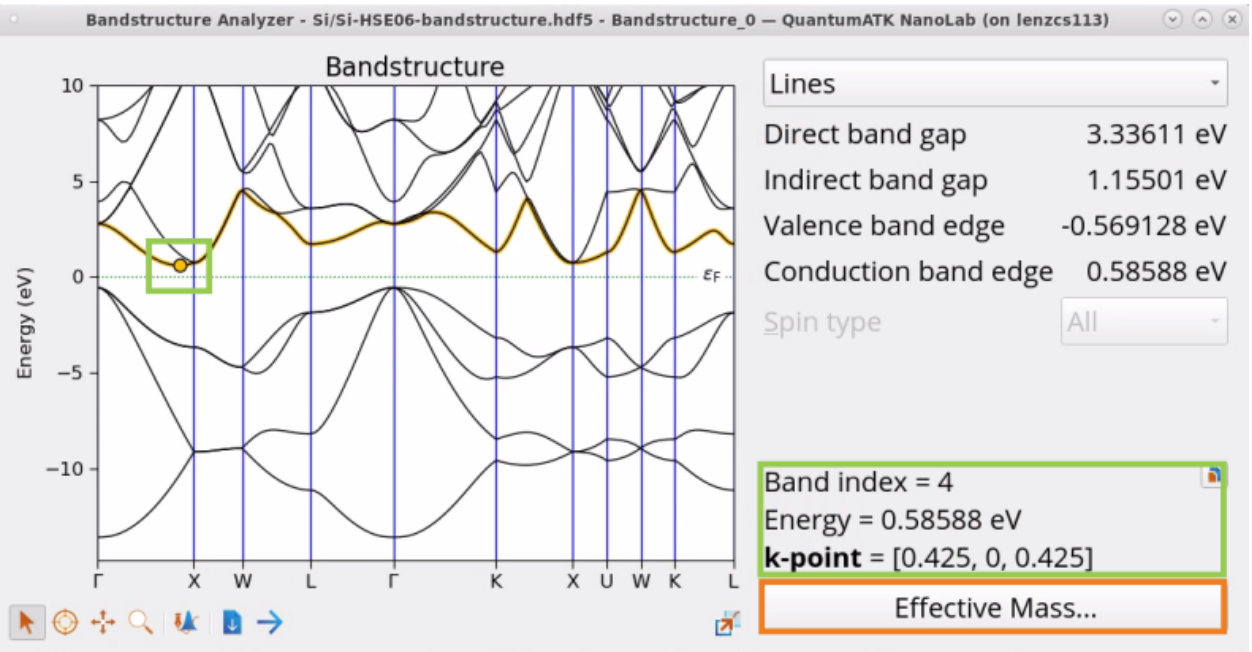
2.In the *Data Sources*, select the file `Si-bandstructure-HSE06.hdf5` and then double-click on the *Bandstructure_0* in the *Data View* window to open it with the *Bandstructure Analyzer*.

3.Find the lowest conduction band of Si and determine the minima. As shown in the image below, the minimum is located at k-point (0.425, 0, 0.425) with 0.585 eV energy, where 0.425 is 85% of the distance to the first Brillouin zone boundary at X=(1/2,0,1/2). At this point, the energy isosurfaces are ellipsoids, thus you get different values for the effective mass depending on if you look in the longitudinal (along) or transverse (perpendicular to) directions.


Projected DOS Analyzer - SiO2-quartz-bandstructure-dos-effective-masses/SiO2-quartz-PDOS-HSE06-DDH.hdf5 - ProjectedDensity
PDOS
LDOS
Plot stype
Lines
Projection
Color
Total
O; l = 0
O; l = 1
O; l = 2
Si; l = 0
Si; l = 1
Projection
Direct band gap
9.908 eV
Indirect band gap
9.6205 eV
Projected DOS
Zero
Fermi level
Total
DOS (eV⁻¹)
Energy (eV)


Plot Viewer — QuantumATK NanoLab (on lenzcs113)
Bandstructure
9.62 eV
Energy (eV)
Γ
M
L
A
Γ
K
H
A
Projected DOS
Zero
Fermi level
Total
DOS (eV⁻¹)
Plot Merging — QuantumATK NanoLab (on lenzcs113)
Match plots when merging?
Combination type
Flipped and side-by-side with shared y-axis
Merge (compare)
Cancel
Merge

4. Click on the *Effective Mass* widget to open the *Effective Mass* block and set up an effective mass calculation at this conduction band minima k-point (0.425, 0, 0.425).



5. In the *Effective Mass* block, the parameters such as band index and k-point coordinates corresponding to the clicked point will be automatically set. Manually change the *Direction* to *Fractional* and set a=1, b=0, c=1, corresponding to the longitudinal direction.

6. Click on the *Send content to other tools* button to add this calculation as a block to the *Workflow* located in the *Workflow Area* → *Build* tab.

7. The workflow already has a *Load BulkConfiguration* block, which restarts the calculation from the former band structure calculation on Si: file `Si-bandstructure-HSE06.hdf5`.

8. Right-click on the *Effective Mass* block in the *Workflow Area* → *Build* tab and choose to *Duplicate* the block.

9. In the duplicated *Effective Mass* block, change *Direction* to a=1, b=1, c=0, corresponding to one of the transverse directions.

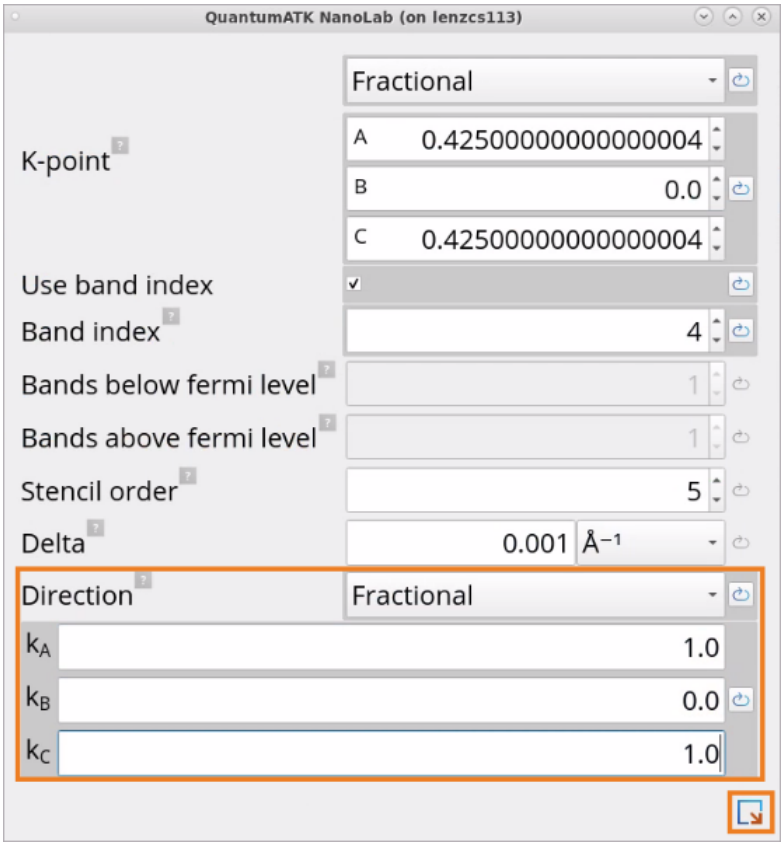
10. Optionally right-click on *Effective Mass* blocks and choose to rename the blocks to *Effective Mass: 101* and *Effective Mass: 110*, respectively.

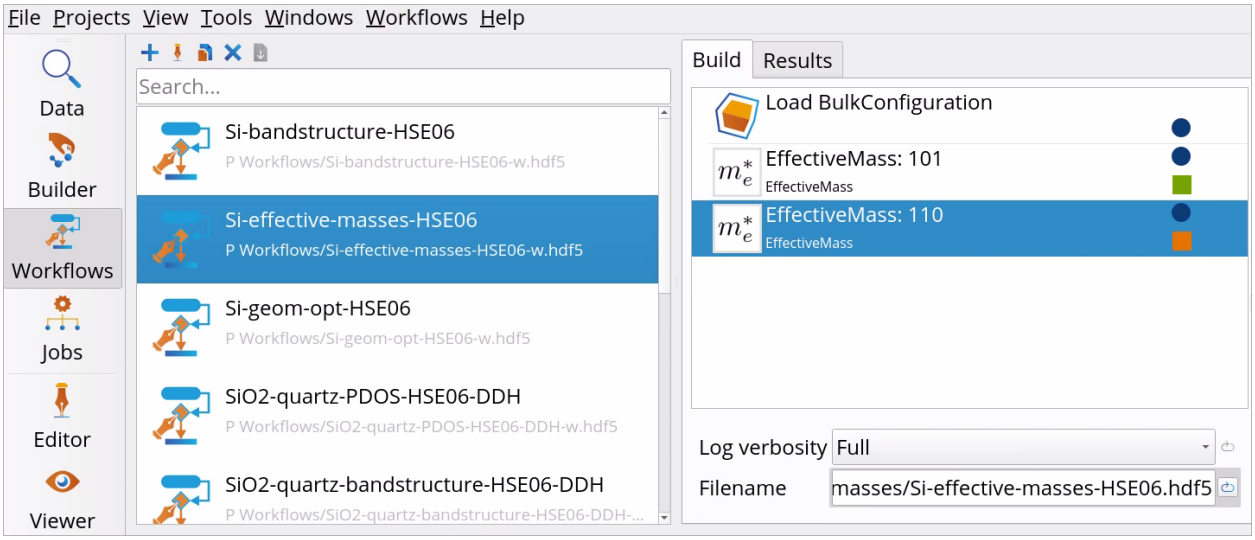
11. Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool in order to submit this job on a local machine or a remote cluster.

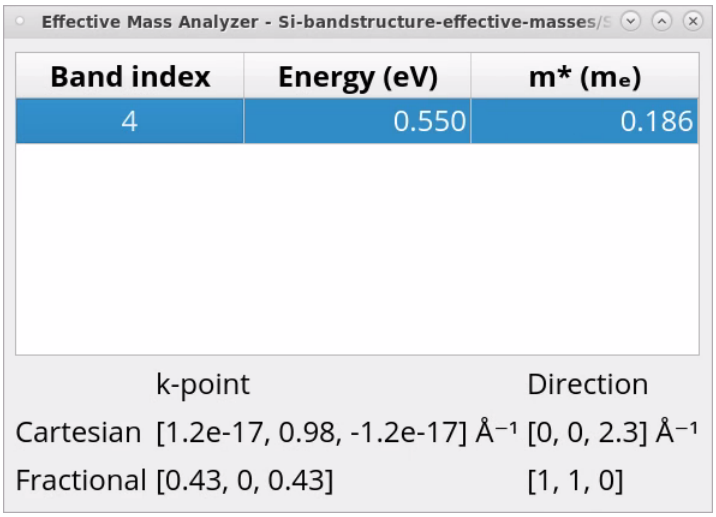
12. Once the calculation is finished, in the *Data Sources*, select the file `Si-effective-masses-HSE06.hdf5` and then double-click on the *EffectiveMass_0* and *EffectiveMass_1* objects in the *Data View* window to open them with the *Effective Mass Analyzer*.

13. Here, the effective mass corresponding to the specified band and direction is reported. From the two *EffectiveMass* objects you will obtain:

- Longitudinal effective mass, [1,0,1] direction, $m^* = 0.85\, m_e$.


QuantumATK NanoLab (on lenzcs113)
Fractional
K-point
A 0.42500000000000004
B 0.0
C 0.42500000000000004
Use band index
Band index 4
Bands below fermi level 1
Bands above fermi level 1
Stencil order 5
Delta 0.001 Å⁻¹
Direction Fractional
kA 1.0
kB 0.0
kC 1.0


File Projects View Tools Windows Workflows Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Search...
Si-bandstructure-HSE06
P Workflows/Si-bandstructure-HSE06-w.hdf5
Si-effective-masses-HSE06
P Workflows/Si-effective-masses-HSE06-w.hdf5
Si-geom-opt-HSE06
P Workflows/Si-geom-opt-HSE06-w.hdf5
SiO2-quartz-PDOS-HSE06-DDH
P Workflows/SiO2-quartz-PDOS-HSE06-DDH-w.hdf5
SiO2-quartz-bandstructure-HSE06-DDH
P Workflows/SiO2-quartz-bandstructure-HSE06-DDH-...
Build
Results
Load BulkConfiguration
EffectiveMass: 101
EffectiveMass
EffectiveMass: 110
EffectiveMass
Log verbosity Full
Filename masses/Si-effective-masses-HSE06.hdf5



•Transverse effective mass, [1,1,0] direction, $m^* = 0.186\, m_e$.

14. With these values we can now compute the density of states mass m^*_{DOS} and effective mass for conductivity m^*_C:

$$m^*_C = 3/(1/m^*_L + 2/m^*_T)$$
$$m^*_{DOS} = 6^{2/3}(m^*_L(m^*_T)^2)^{1/3}.$$

We get $m^*_{DOS} = 1.02$ and $m^*_C = 0.25$, very close indeed to the reference values 1.08 and 0.26 (see https://www.colorado.edu/ecee/sites/default/files/attached-files/sd_constants_sheet.pdf). Thus DFT actually does reproduce the effective mass of the electrons in the Δ valley very well.

References

1.2.3 Optical Property Calculations

Version: X-2025.06

In this tutorial, you will learn how to use QuantumATK NanoLab GUI to

•Set-up optical property simulations of dielectric constant, refractive index, IR spectra, absorption spectra, etc. for SiO_2 (quartz) using Density Functional Theory (DFT).

•Analyze obtained results.

All the workflow, input and output files can be found in the **Example Project - X-2025.06 → SiO2-quartz-dielectric-tensor folder.**

Prerequisites

•Video Introduction to QuantumATK NanoLab GUI

•Video tutorial on the Workflow Builder tool

•Video tutorial on the Job tool

•Tutorial *Geometry Optimization*

•Tutorial *Band Structure, Projected Density of States and Effective Mass Calculations*

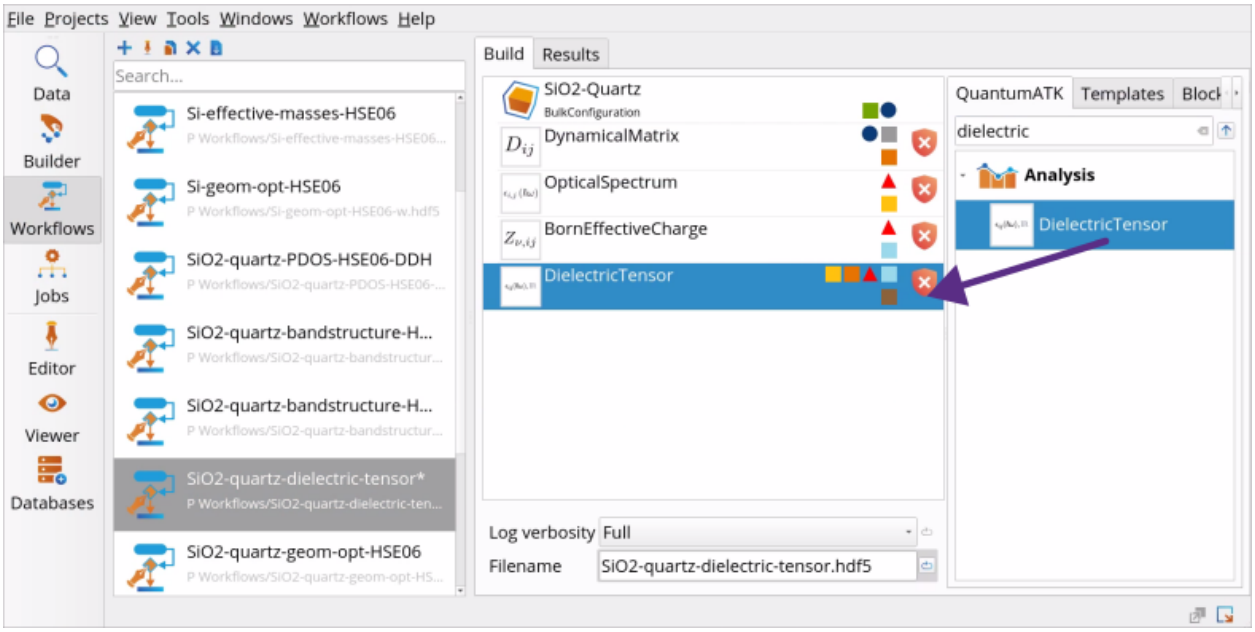
Setting up an Optical Property Calculation Workflow

QuantumATK can calculate many different optical properties from the *DielectricTensor*, including both electronic and ionic contributions, such as dielectric constant, refractive index, reflectivity, extinction coefficient, optical absorption coefficient, IR spectra, susceptibility. Follow the step-by-step protocol described below to set up these simulations for SiO_2 (quartz) in the **Workflow Builder**.

1. Follow the tutorial *Geometry Optimization* to import the structure of bulk SiO_2 (quartz) from NanoLabs *Crystal Database* to the **Workflow Builder**.

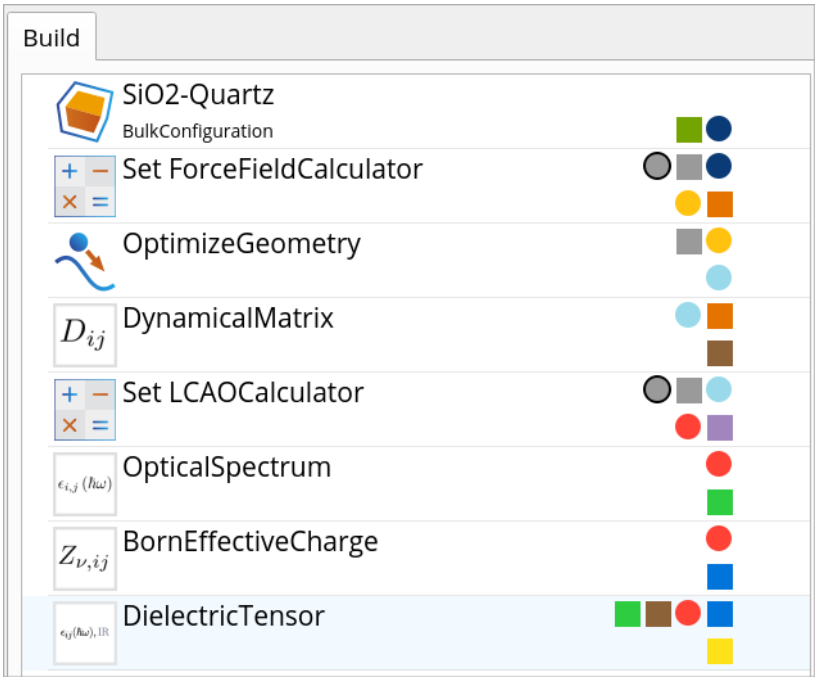
2. Go to the *Block Catalog* → *QuantumATK* tab → *Analysis* category on the right hand panel and drag-and-drop the *DielectricTensor* block to the middle *Workflow Area* → *Build* tab.

3. **Workflow Builder** automatically adds-in the *Optical Spectrum* block (for the electronic part of the dielectric tensor), and then *Dynamical Matrix* and *Born Effective Charge* blocks (for the ionic part of the dielectric tensor) to the workflow. You see the warning signs, because the calculators are not yet specified for these blocks (see the next steps).

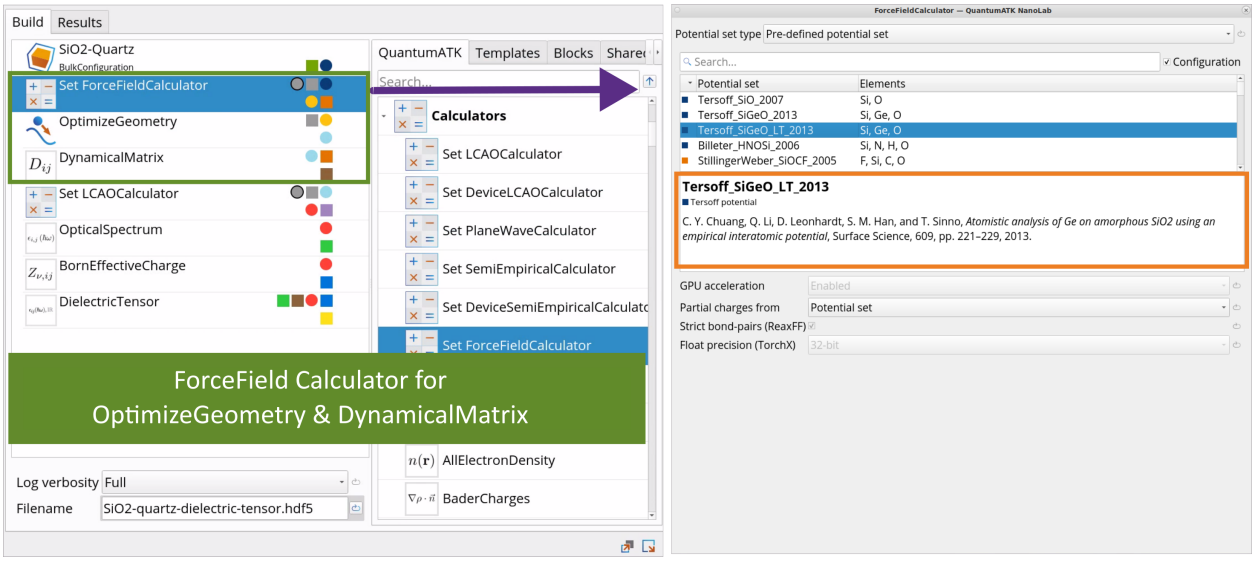


4. Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* and *Optimization and Dynamics* categories on the right hand panel and drag-and-drop the *SetForceFieldCalculator* and *OptimizeGeometry* blocks, respectively, to the middle *Workflow Area* → *Build* tab, above the *DynamicalMatrix* block.

5. Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* category and drag-and-drop the *SetLCAOCalculator* block to the middle *Workflow Area* → *Build* tab, below the *DynamicalMatrix* block, to define the DFT *LCAOCalculator* that will be used for *OpticalSpectrum* and *BornEffectiveCharge* simulations. The full workflow for optical property calculations is shown below.



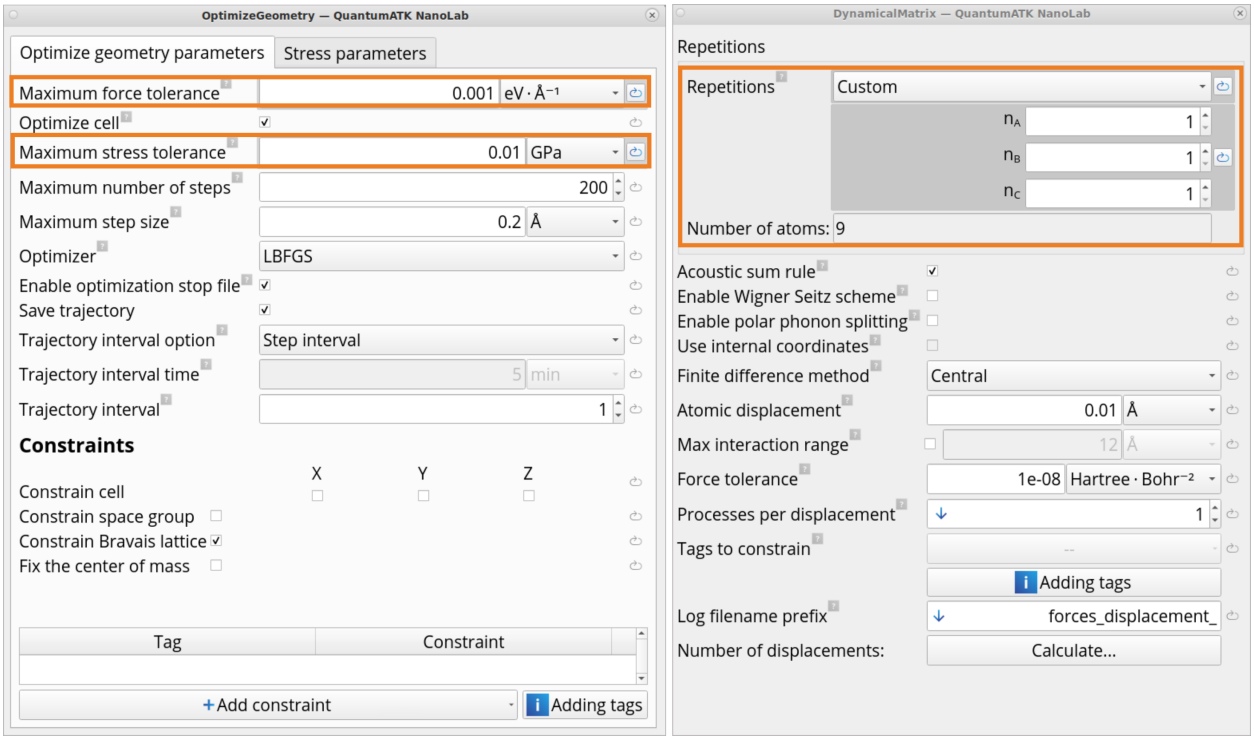
6. Double-click on the *SetForceFieldCalculator* block and choose a *Tersoff_SiGeO_LT_2013* Force Field for *OptimizeGeometry* and *DynamicalMatrix* calculations. It is important to use the same calculator for both geometry optimization and dynamical matrix simulations.



7. Double-click on the *OptimizeGeometry* block and set a tighter than default force (*0.001*) and stress (*0.01*) tolerances, as it is important to have a very well converged geometry for the subsequent dynamical matrix simulations.

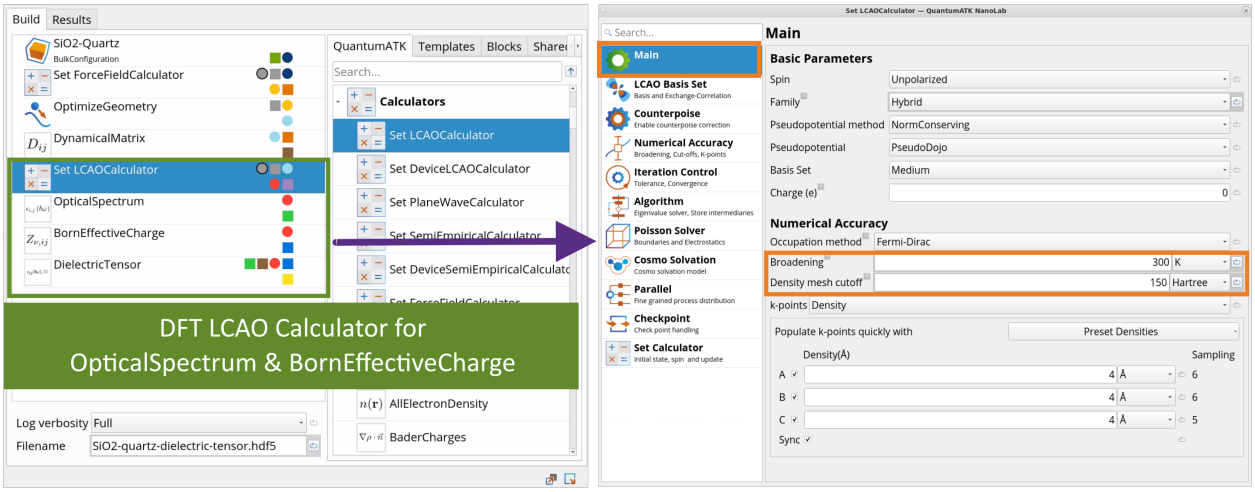
8. Double-click on the *DynamicalMatrix* block and change the *Repetitions* to *Custom*: *1, 1, 1*, as you need to include vibrations only at the point and do not need a repeated super cell. This makes the calculation computationally much

cheaper (and manageable with DFT if it is chosen instead of Force Fields) as now you need to calculate displacements for 9 atoms instead of 2205 atoms in the default 7 x 7 x 5 repetition.



9.Double-click on the *SetLCAOCalculator* block and make the following edits:

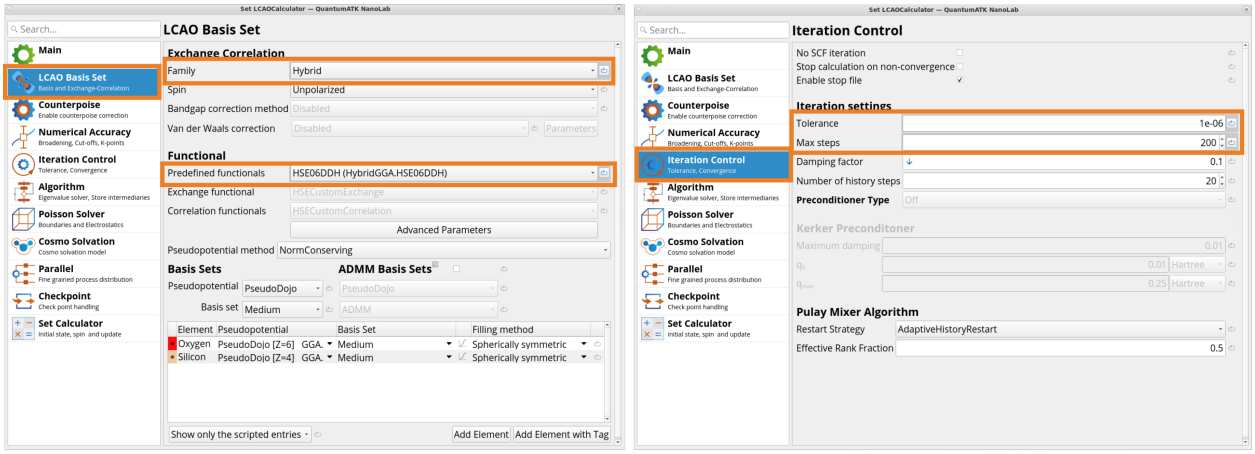
•Under the *Main* tab, increase the *Density Mesh Cutoff* from the default *125 Ha* to *150 Ha* as this is necessary for obtaining accurate and converged Born effective charges and thus optical properties, in particular the dielectric constants ionic part. Change the numerical accuracy parameter *Broadening* from *1000 K* to *300 K* (recommended for semiconductor materials).



•Under the *LCAO Basis Set* tab, change the *Exchange Correlation Family* from the default *GGA* to *Hybrid* and for the *Predefined functional* choose *HSE06DDH (Hybrid.GGA.HSEO6DDH)*. Here, we will use a dielectric dependent

hybrid functional HSE06-DDH (*DDHCustomExchangeCorrelation*) as it provides accurate electronic structure for SiO_2 (quartz), as shown in the Tutorial *Band Structure, Projected Density of States and Effective Mass Calculations*, which is important for optical properties.

- Under the *Iteration Control* tab, set a tighter than default SCF convergence tolerance (*1e-06* vs. *1e-04*), which is also crucial for calculating accurate and converged Born effective charges. In addition, increase the *Max steps* for SCF from the default *100* to *200* so the SCF cycle converges even if it needs more steps due to the tighter tolerance.



Tip:

1. *SetLCAOCalculator* block sets medium k-point sampling and medium basis set by default, typically providing good accuracy results with modest computational resources. It is recommended to check the convergence of your results with these parameters.

2. *SetLCAOCalculator* block uses a default *Fermi-Dirac Occupation Method* with 1000 K *Broadening*, which generally works reasonably well across various systems. Another setting may be more appropriate for your system, and you can adjust these settings according to the recommendations in *Occupation Methods* for semiconductor, insulator, metal, and molecule systems.

10. In the *OpticalSpectrum* block, change the following settings to calculate *OpticalSpectrum* accurately:

- Change the default number of *Bands below/above Fermi level* to arbitrary *1000* (as the default *4* is too few for SiO_2 (quartz)).

- For the *Transition matrix operator* method, select the *VelocityOperator* (which is also the default method for the DFT-LCAO calculator with Norm Conserving PPs), as it gives a more accurate optical spectrum compared to the *MomentumOperator* method.

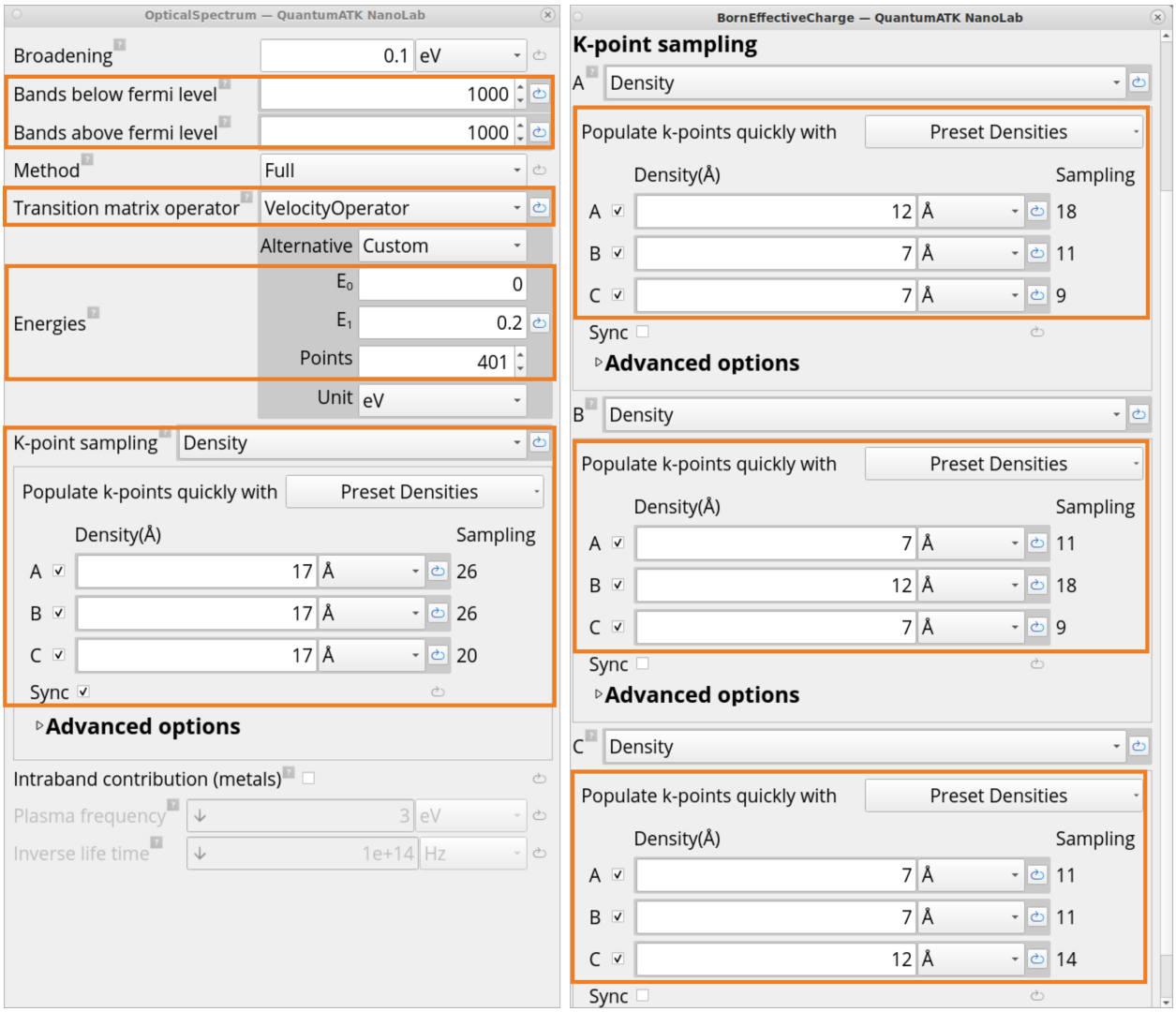
- Select the energy range *0 – 0.2 eV* and change to *401* points for finer sampling. This will ensure a high-resolution IR spectrum, which typically features many narrow and closely spaced peaks corresponding to different vibrational modes.

- For the *k-point sampling*, choose *Density* and click on *Preset Densities* to increase the k-point density from *medium* to *higher*, i.e., *17Å x 17Å x 17Å*. It is important to have a large k-point sampling for properties that depend on the conduction band minimum, for example, dielectric constant and electron transport properties.

Tip:

- When calculating absorption spectra, it is recommended to select a wider energy range and use default (more coarse) sampling. This is because an absorption spectrum typically covers a broader range of wavelengths and can include various types of transitions, often resulting in broader peaks compared to those in IR spectra.

11. In the *BornEffectiveCharge* block, for the *k-point sampling*, choose *Density* and click on *Preset Densities* to increase the k-point density from *medium* to *high*, i.e., A: *12Å x 7Å x 7Å*, B: *7Å x 12Å x 7Å*, C: *7Å x 7Å x 12Å*.



12. The *DielectricTensor* block has no adjustable paramaters. It takes in calculated *DynamicalMatrix*, *OpticalSpectrum* and *BornEffectiveCharges* in previous steps to compute the *DielectricTensor*.

13. Change the *Workflow* name in the the *Workflow Stash* and *Filename* as needed (e.g., `SiO2-quartz-dielectric-tensor.hdf5`).

14. Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool to submit this job on a local machine or a remote cluster.

15. This also generates a Python script `SiO2-quartz-dielectric-tensor.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool.

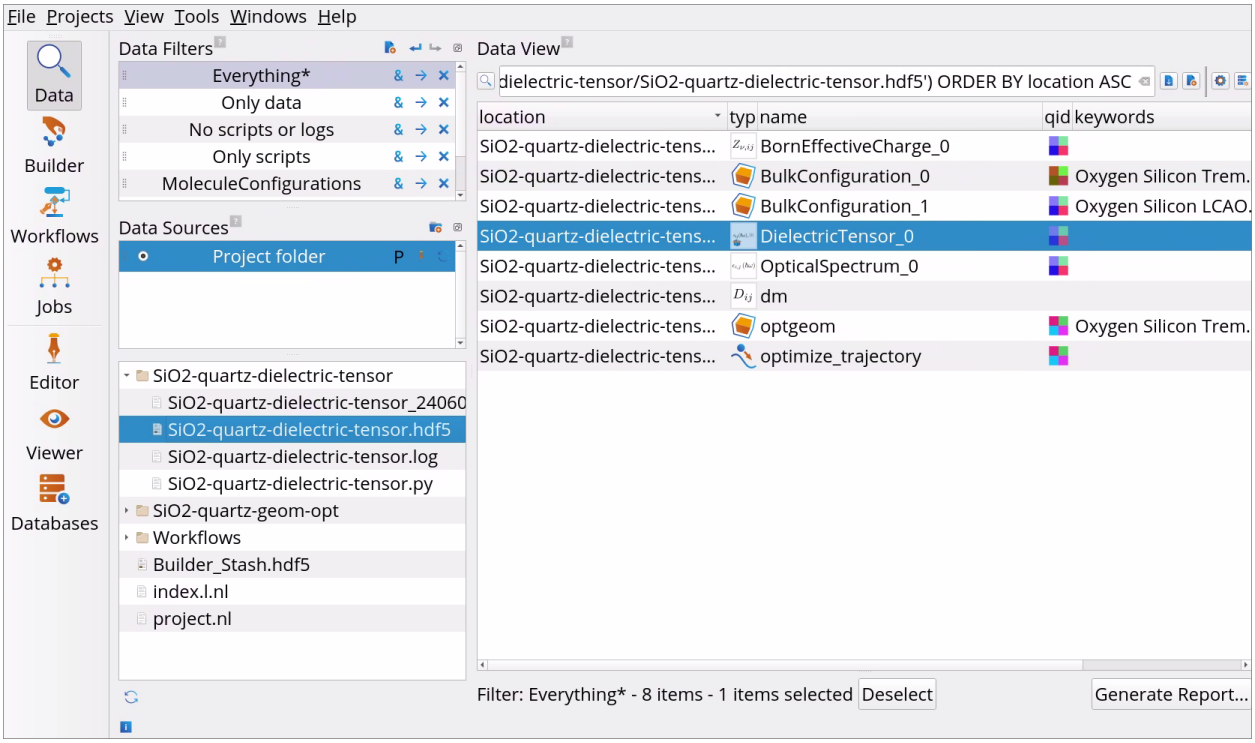
Note:

1. *DynamicalMatrix*, the second derivative of the energy, corresponding to the first derivative of the forces, is an object used to calculate many properties related to dynamics of the material lattice, e.g., *DielectricTensor*, *PhononBandstructure*, *Mobility*, *RamanSpectrum*, *DeformationPotential*.
2. The *BornEffectiveCharge* tensor is defined as the derivative of the polarization with respect to the atomic coordinate and is used for *DielectricTensor* calculation.

Analyzing Results

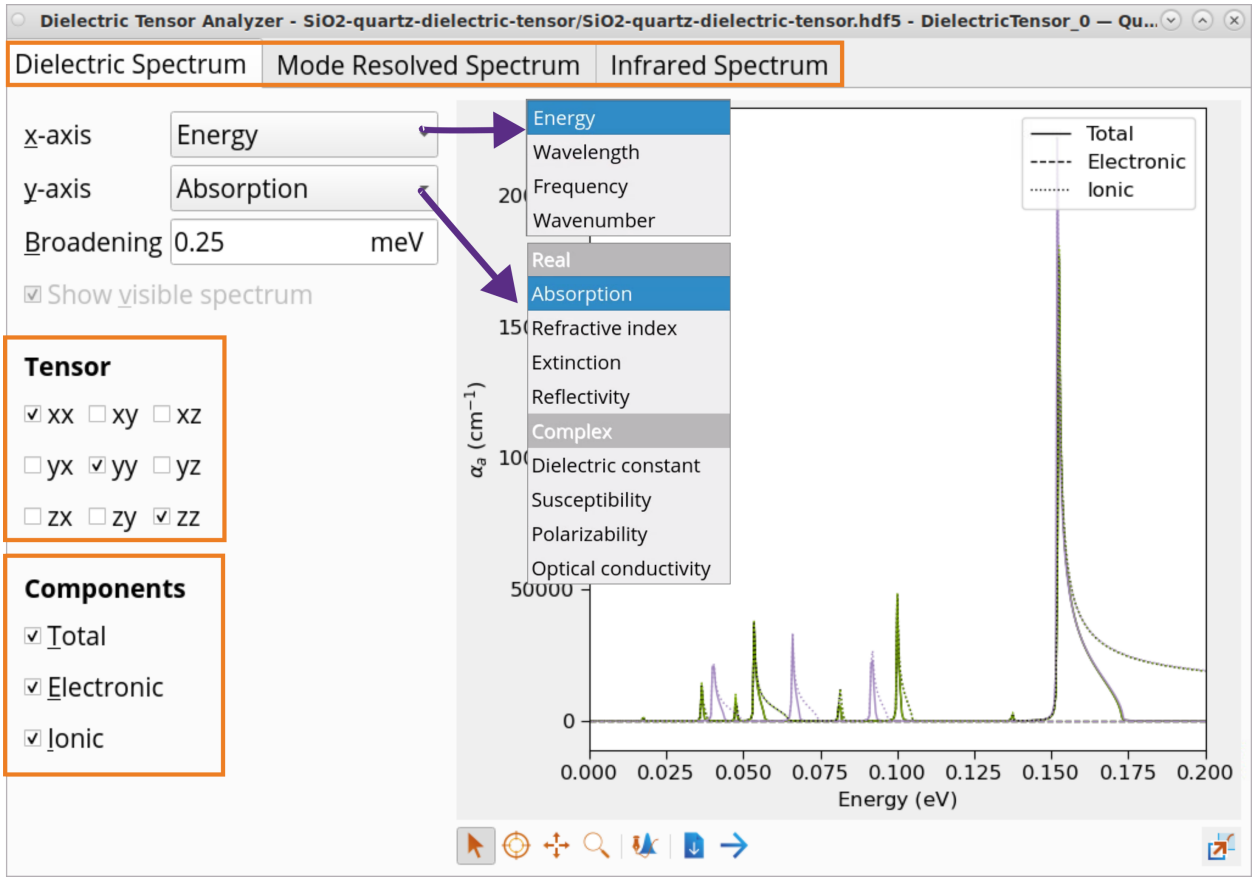
Once the calculation has finished, we are ready to analyze the results.

1. Go to the **Data View** tool.
2. In the *Data Sources*, select the file `SiO2-quartz-dielectric-tensor.hdf5` and then double-click on the *DielectricTensor_0* object in the *Data View* window to open it with the *Dielectric Tensor Analyzer*.



Tip:

1. The *Dielectric Tensor Analyzer* can plot a range of optical properties. In the *Dielectric Spectrum* panel, click on *y-axis* to see the list: *Absorption*, *Refractive index*, *Extinction*, *Reflectivity*, *Dielectric constant*, *Suscpetibility*, *Polarizability*, *Optical conductivity* as a function of *Energy*, *Wavelength*, *Frequency* or *Wavenumber* (x-axis).
2. Plot *Mode resolved spectrum* and *Infrared Spectrum*.
3. Plot differet tensor components (xx, yy, zz) and total, electronic and ionic contributions to the optical properties.



3.To plot and analyze the dielectric constant:

•Select the *Dielectric constant* (y-axis) as a function of *Energy* (x-axis).

•Click on the x axis and then on the *Plot editor* button.

•Adjust the *Lower limit* and *Upper limit* to x and y axes to zoom into the $Re[]$ at E = 0 eV to determine the static dielectric constant, $_0$. For the x axis, select a range from 0 to 2.5e-05, and for the y axis, select a range from 0 to 10.

•Use the *Inspect* tool to examine dielectric constant values for SiO_2 (quartz):

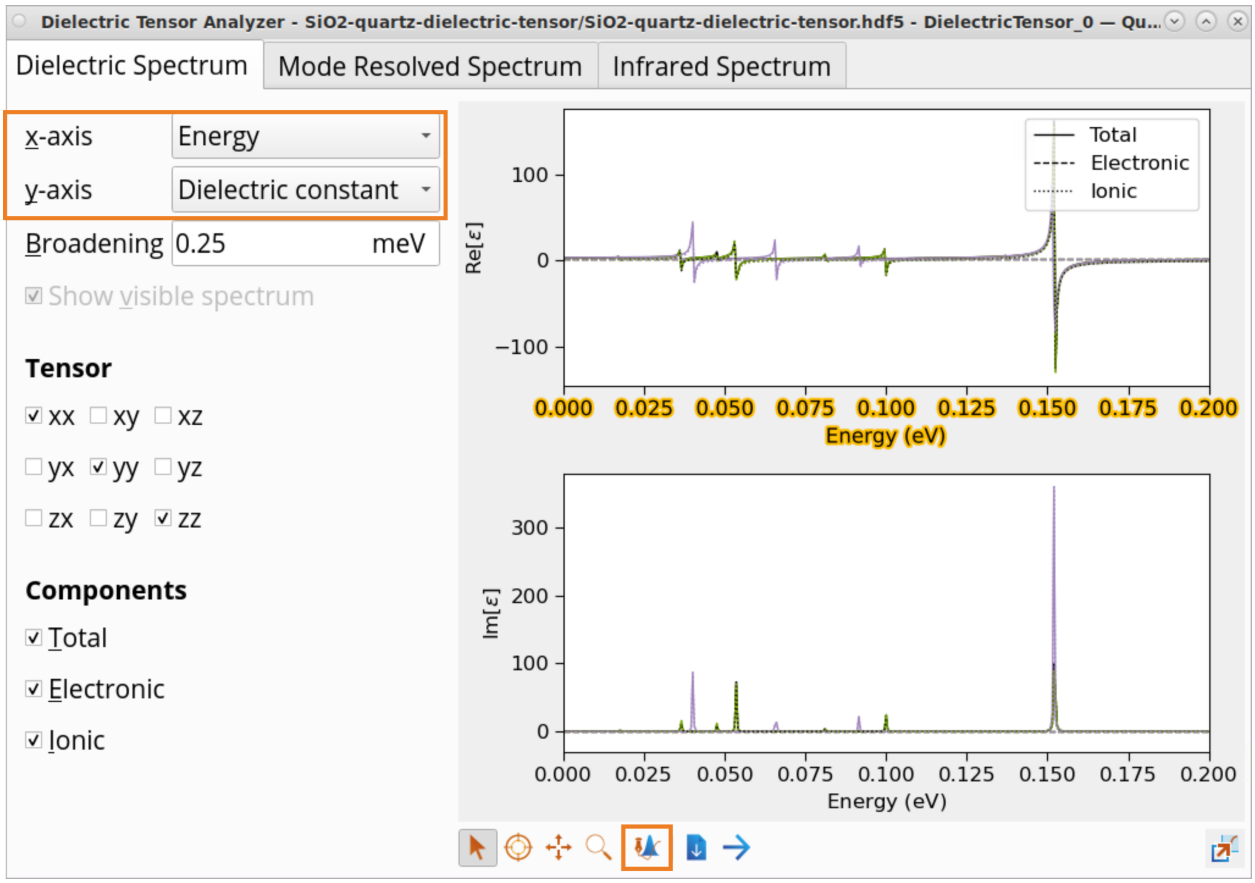
–total: $_{0_{xx/yy}}$ = 3.72 and $_{0_{zz}}$ = 3.92

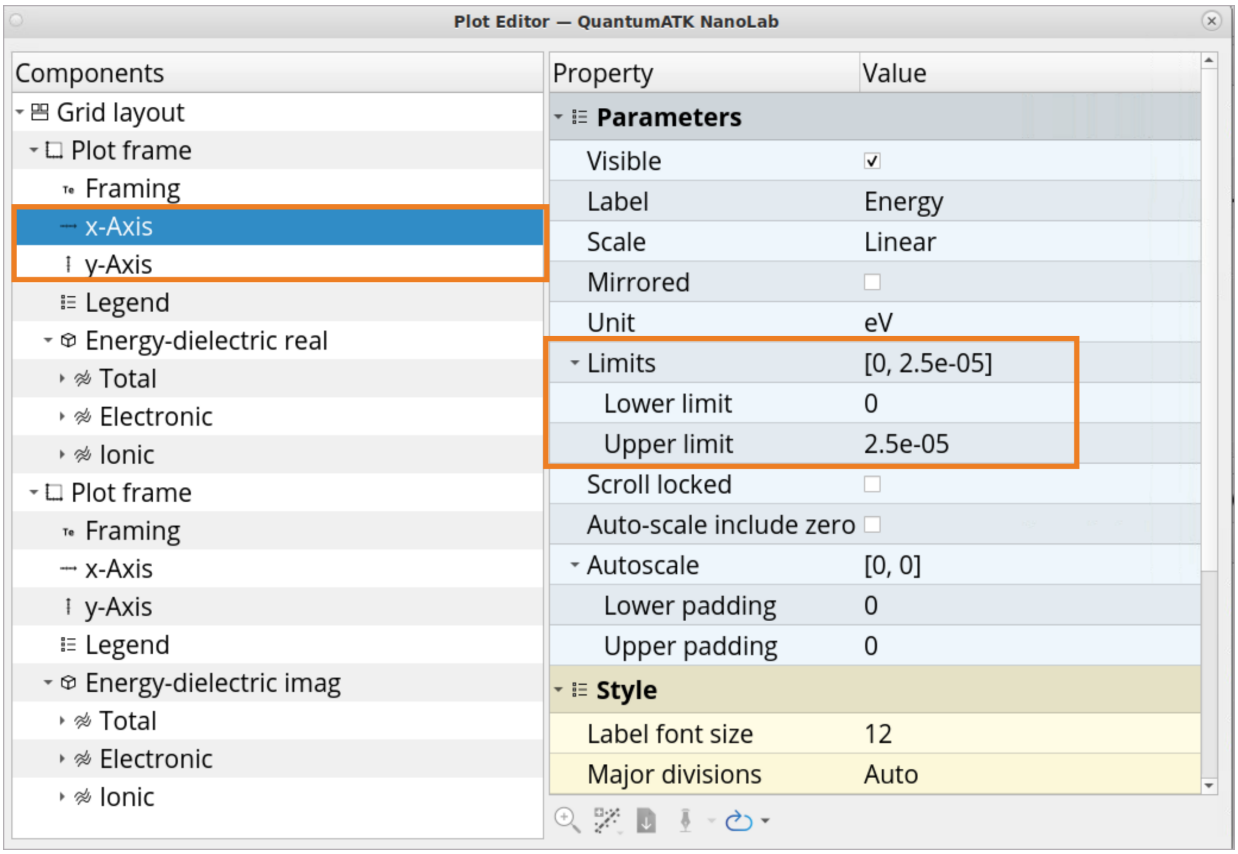
–electronic contribution: $_{el_{xx/yy/zz}}$ = 2.11

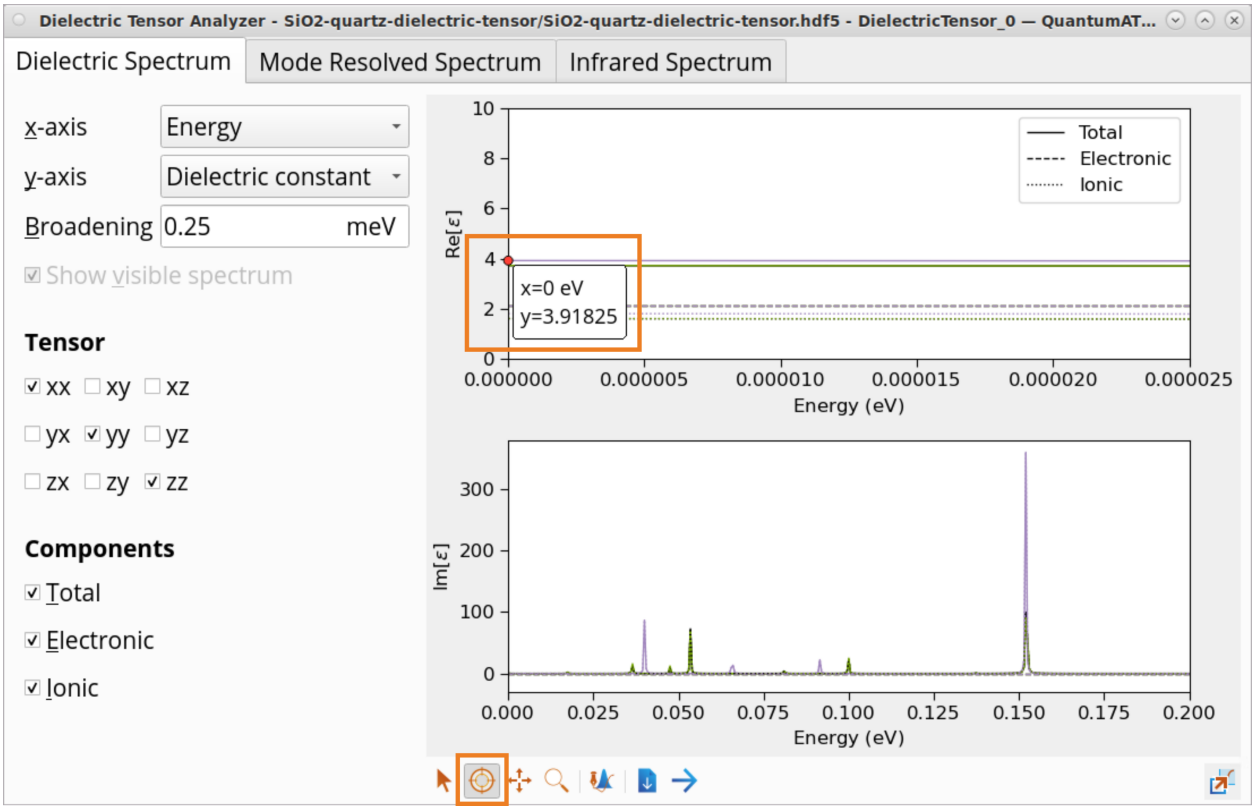
–ionic contribution: $_{ion_{xx/yy}}$ = 1.61 and $_{ion_{zz}}$ = 1.81

Note:

1.Ionic contribution to the SiO_2 (quartz) static dielectric constant is significant and cannot be neglected.


Dielectric Tensor Analyzer - SiO2-quartz-dielectric-tensor/SiO2-quartz-dielectric-tensor.hdf5 - DielectricTensor_0 — Qu...
Dielectric Spectrum
Mode Resolved Spectrum
Infrared Spectrum
x-axis
Energy
y-axis
Dielectric constant
Broadening
0.25
meV
Show visible spectrum
Tensor
xx
xy
xz
yx
yy
yz
zx
zy
zz
Components
Total
Electronic
Ionic
Total
Electronic
Ionic
Re[ε]
100
0
−100
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
Energy (eV)
Im[ε]
300
200
100
0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
Energy (eV)


Plot Editor — QuantumATK NanoLab
Components
Grid layout
Plot frame
Framing
x-Axis
y-Axis
Legend
Energy-dielectric real
Total
Electronic
Ionic
Plot frame
Framing
x-Axis
y-Axis
Legend
Energy-dielectric imag
Total
Electronic
Ionic
Property
Value
Parameters
Visible
Label
Energy
Scale
Linear
Mirrored
Unit
eV
Limits
[0, 2.5e-05]
Lower limit
0
Upper limit
2.5e-05
Scroll locked
Auto-scale include zero
Autoscale
[0, 0]
Lower padding
0
Upper padding
0
Style
Label font size
12
Major divisions
Auto


Dielectric Tensor Analyzer - SiO2-quartz-dielectric-tensor/SiO2-quartz-dielectric-tensor.hdf5 - DielectricTensor_0 — QuantumAT...
Dielectric Spectrum
Mode Resolved Spectrum
Infrared Spectrum
x-axis
Energy
y-axis
Dielectric constant
Broadening
0.25
meV
Show visible spectrum
Tensor
xx xy xz
yx yy yz
zx zy zz
Components
Total
Electronic
Ionic
Total
Electronic
Ionic
Re[ε]
x=0 eV
y=3.91825
Energy (eV)
Im[ε]
Energy (eV)

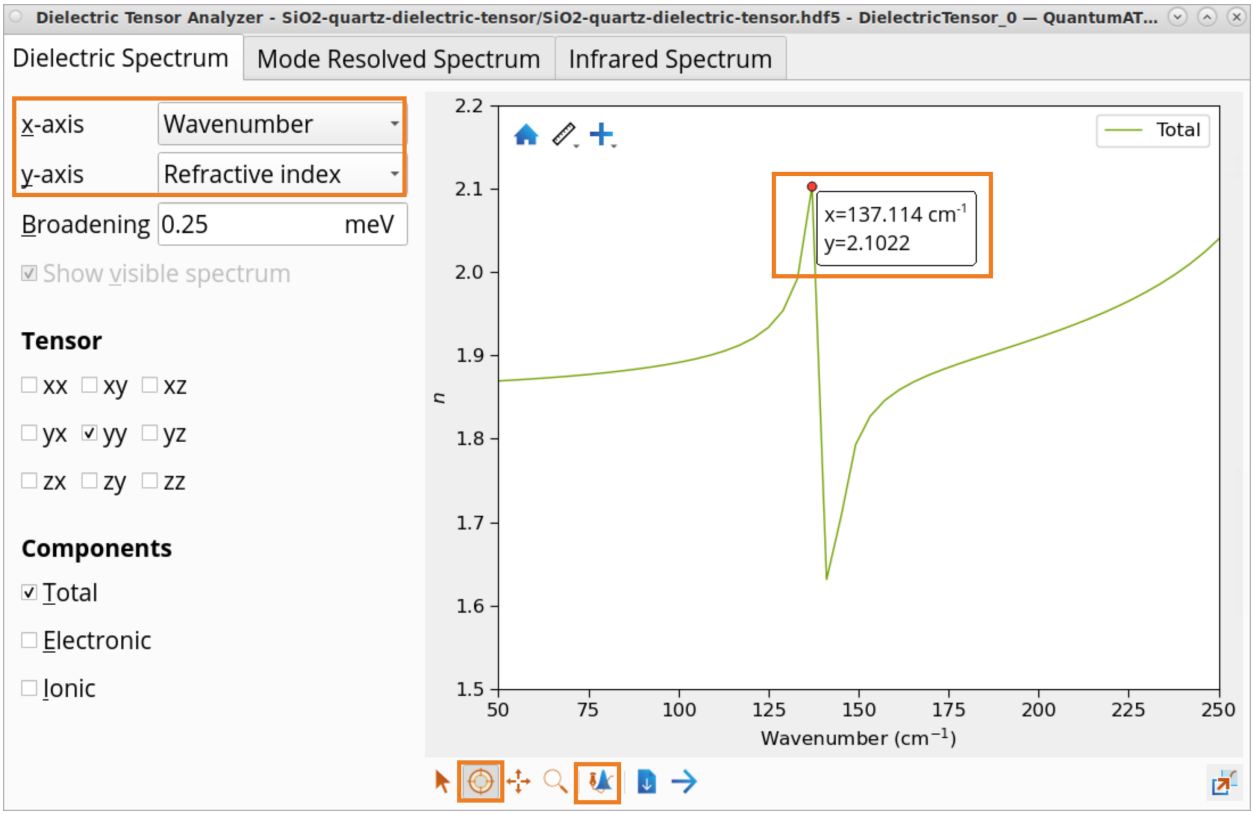
2.The calculated dielectric constant, $_0$, and symmetry break are in a reasonably good agreement with the experimental results at a room temperature: $_0$ = 4.43 and $_0$ = 4.64, in the directions perpendicular to and along the trigonal axis, respectively[1].

4.To plot and analyze refractive index:

•Select the *Refractive index* (y-axis) as a function of *Wavenumber* (x-axis).

•Click on the *Plot editor* to zoom in on the refractive index n at around 50-250 cm^{-1}.

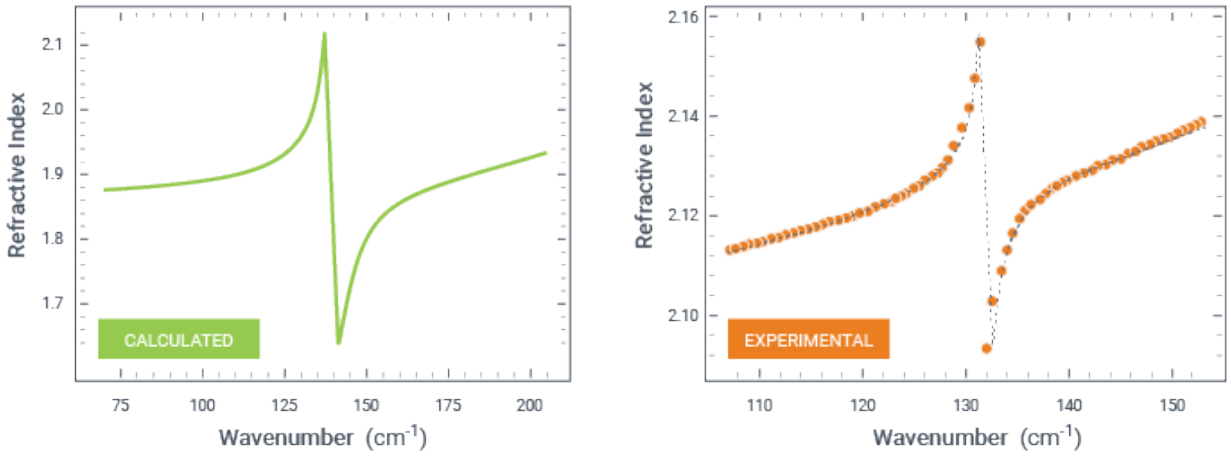
•Use the *Inspect* tool to examine refractive index values at the peaks: e.g., $n_{yy}(137cm^{-1})$ = 2.10 and $n_{yy}(141cm^{-1})$ = 1.63.



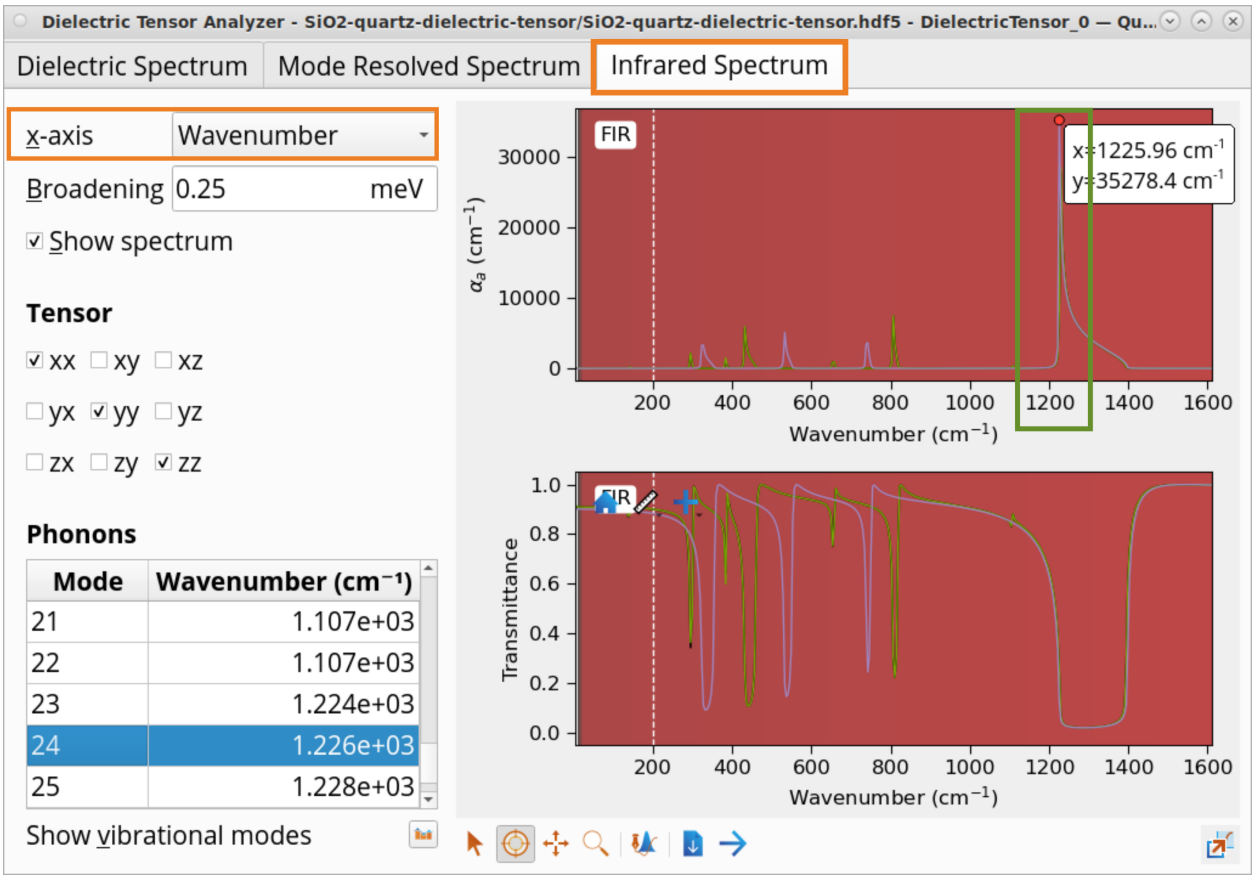
Note: The simulated refractive index at 0 K is in a qualitative agreement with the experimental results at 10 K[2] as shown in the Figure below.

[1] F.ăGervais and B.ăPiriou. Temperature Dependence of Transverse and Longitudinal Optic Modes in the and Phases of Quartz. *Physical Review B*, 11(10):3944–3950, 1975. doi:10.1103/PhysRevB.11.3944.

[2] C.ăL. Davies, J.ăB. Patel, C.ăQ. Xia, L.ăM. Herz, and M.ăB. Johnston. Temperature-Dependent Refractive Index of Quartz at Terahertz Frequencies. *Journal of Infrared, Millimeter, and Terahertz Waves*, 39:1236–1248, 2018. doi:10.1007/s10762-018-0538-7.



5.Click on the *Infrared Spectrum* tab to examine the calculated IR spectrum.



•The highest peak in the IR spectrum at ~1226 cm^{-1} corresponds well to the experimentally measured IR peak at ~1100-1200 cm^{-1} for powder SiO_2 (quartz). See https://webbook.nist.gov/cgi/cbook.cgi?ID=C14808607&Contrib=IARPA-IR-S&Type=IR-SPEC&Index=0#IR-SPEC).

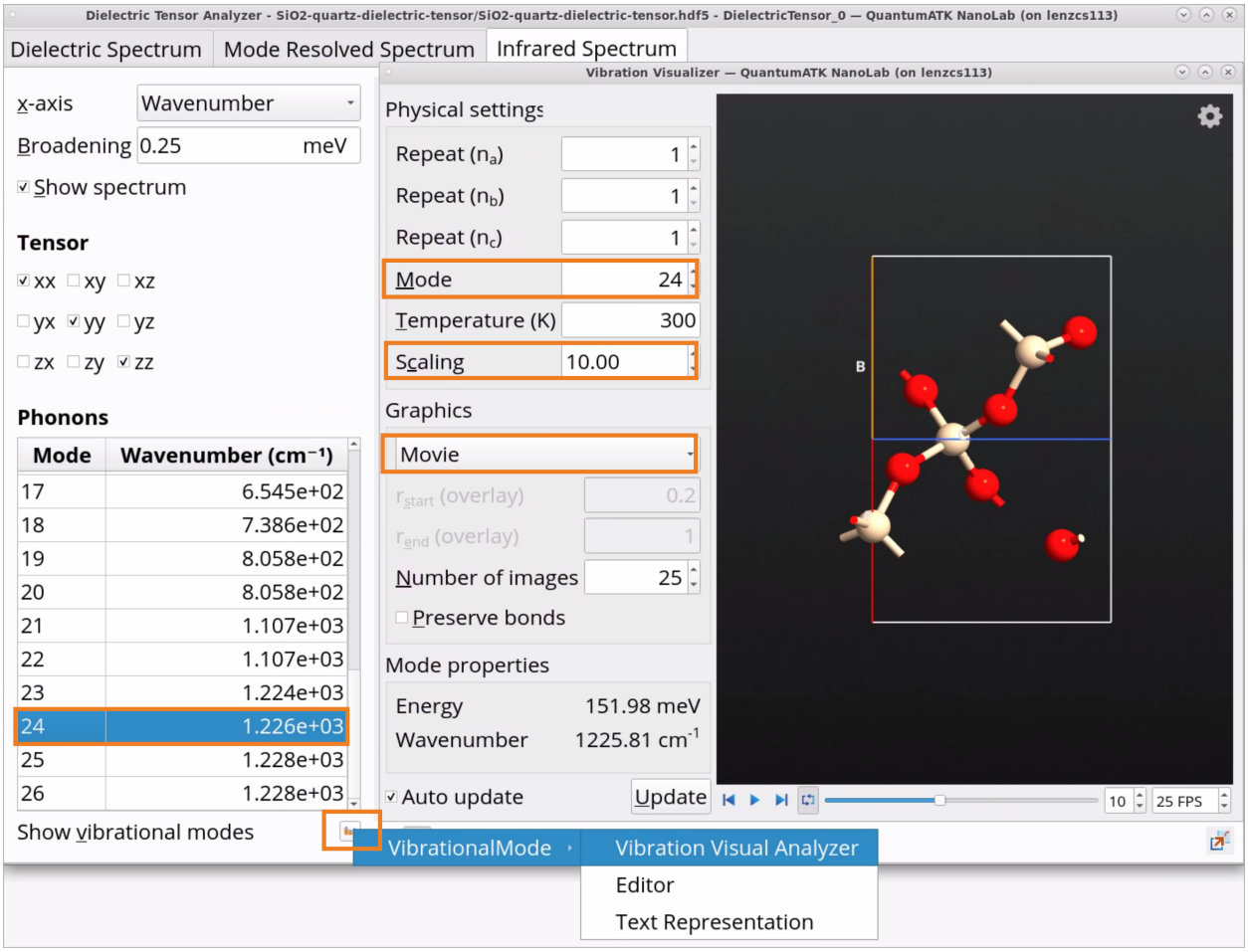
Note: Due to the temperature effects and random crystal orientations in the powder, the experimentally measured IR peak is broader than the calculated one for the single crystal direction at 0K.

6.Click on the *Show vibrational modes* icon and choose *Vibration Visual Analyzer* to visualize chosen vibrational modes.

•In the *Vibration Analyzer*, choose a vibrational mode, e.g., *24*.

•Set *Scaling* to *10*.

•Under *Graphics*, choose *Movie* to see the actual vibration for mode 24 at 1226 cm^{-1} corresponding to the highest peak in the IR spectrum.



Outlook

1. If you wish to explore optical properties further, QuantumATK has additional optical analyses like *RamanSpectrum*, *SecondHarmonicsGenerationSusceptibility*, and *ElectroOpticalTensor*.
2. For polar materials, it is possible to apply polar-phonon (LO-TO) splitting to the vibrational modes beyond the workflow presented here. This effect can be included by adding the optical properties and Born effective charge to the dynamical matrix analysis, and doing so will increase the LO mode vibrational frequency (observable in Raman spectroscopy for instance).

References

1.2.4 Molecular Dynamics Simulations for Generating Amorphous Structures

Version: X-2025.06

In this tutorial, you will learn how to use QuantumATK NanoLab GUI to

- Prepare an initial structure for amorphous SiO_2 using *Amorphous Prebuilder* based on *Packmol*.
- Set-up Molecular Dynamics (MD) melt-quench and subsequent MD equilibration simulations to generate a final equilibrated amorphous SiO_2 structure using classical Force Fields.
- Analyze obtained results, such as radial and angular distribution functions, RDF and ADF, to verify the quality of the generated amorphous SiO_2 structure.
- Set-up a geometry optimization simulation for amorphous SiO_2 in order to obtain a 0 K structure, representing the most stable configuration. This optimized structure can serve as a reference state for further calculations of electronic properties, phonons and other thermodynamic properties.

All the workflow, input and output files can be found in the **Example Project - X-2025.06 → Am-SiO2-MD folder.**

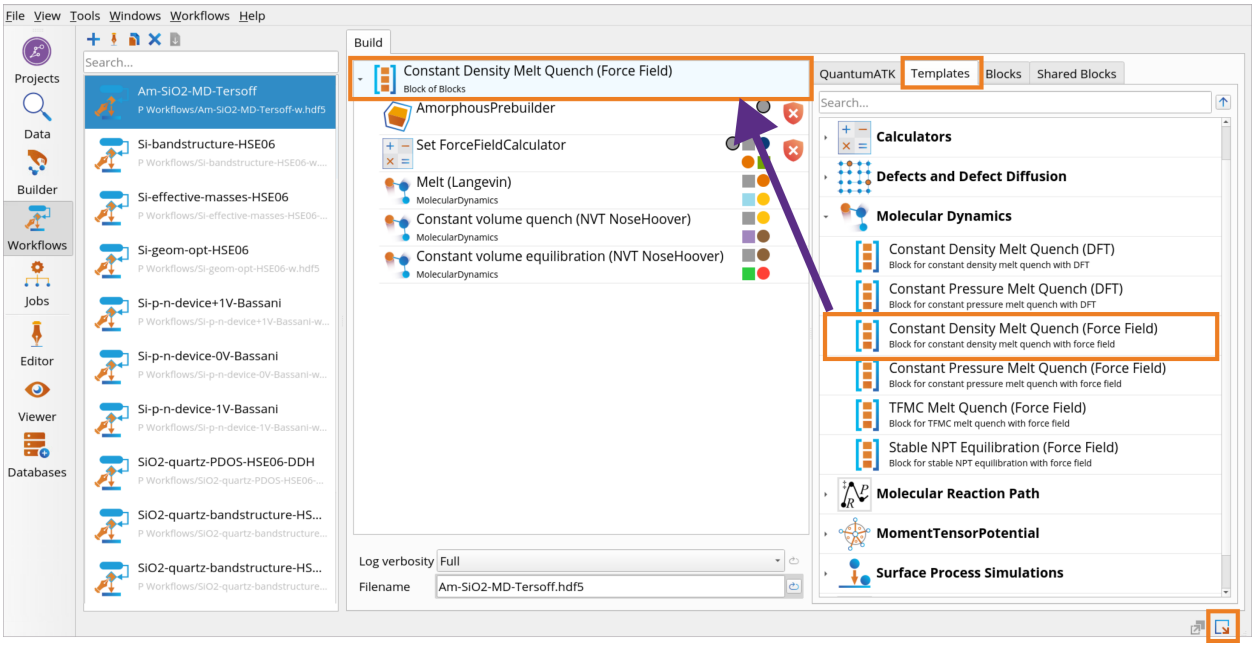
Prerequisites

- Video Introduction to QuantumATK NanoLab GUI
- Video tutorial on the Workflow Builder tool
- Video tutorial on the Job tool
- Tutorial *Geometry Optimization*

Setting-Up Molecular Dynamics Simulations

To create a workflow for generating an initial amorphous structure, performing *MolecularDynamics* melt-quench and equilibration simulations in the **Workflow Builder**, follow the step-by-step protocol described below. Here, we will use a classical Force Field *Tersoff_SiO_2007*.

1. Go to the *Block Catalog → Templates* tab → *Molecular Dynamics* category on the right hand panel and drag-and-drop the *Constant Density Melt Quench (Force Field)* template to the middle *Workflow Area → Build* tab. You see the warning signs, because the Force Field calculator and the initial amorphous structure are not yet specified for these blocks (see the next steps).



2.Click on the *AmorphousPrebuilder* block to set up a generation of an initial amorphous structure from selected elements on-the-fly. It will use *Packmol* to randomly pack atoms into a bulk configuration given a list of elements stoichiometry and material density.

•Click on *Add row* button.

•In the opened *Periodic Table* window, click on *Si* to add it to the *AmorphousPrebuilder*.

•Repeat the steps to add *O* to the *AmorphousPrebuilder*.

•Set *Stoichiometry 1* to *Si* and *2* to *O*.

•Set *Size multiplier* to *100* to generate an initial amorphous cell with 100 Si and 200 O atoms.

•For *Density option*, choose *By atomic mass* and specify *2.2* gcm^{-3} *Mass density* for the amorphous SiO_2 structure. This will automatically determine the *Lattice constant* of *16.55* $\AA$.

•The generated initial amorphous structure will automatically be passed into the 2nd part of the simulation workflow, which includes melt-quench and equilibration MD simulations.

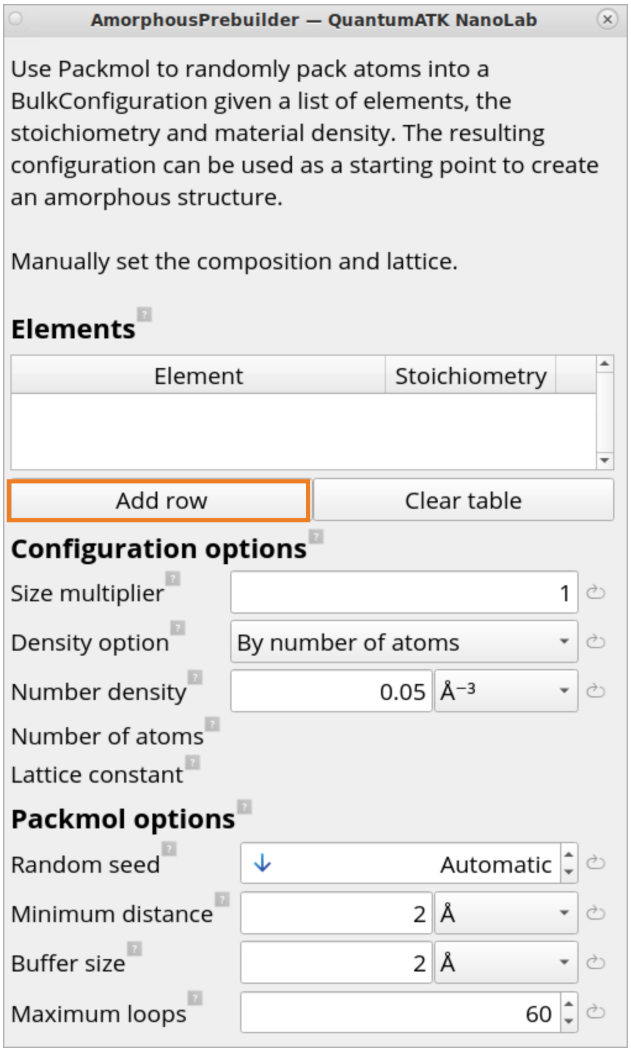
3.Click on the *Set ForceFieldCalculator* block and choose *Tersoff_SiO_2007* Classical Force Field.

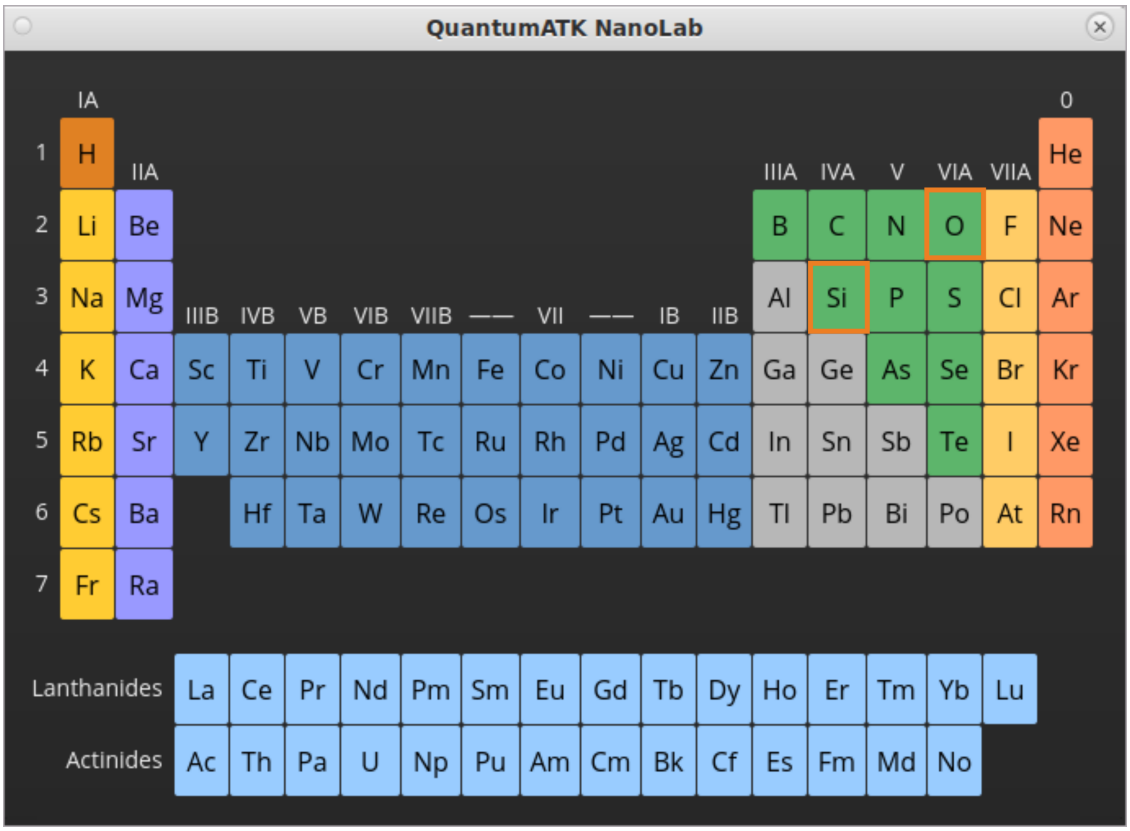
Tip:

•For systems for which there is no conventional Force Field available or the quality is not good, we recommend to use Machine Learned Force Fields (ML FFs). Consider:

1.Using pre-trained custom Moment Tensor Potentials (MTPs) for a specific target system and application or pre-trained universal MACE or M3GNet Neural Network (NN) potentials. The list of available ML FFs can be found in the *SetMachineLearnedForceFieldCalculator* block.

2.Training and employing custom Moment Tensor Potentials (MTPs) for a specific target system and application. Learn about the MTP training here: https://docs.quantumatk.com/tutorials/mtp-training-c-am-TiSi/mtp-training-c-am-TiSi.html.


AmorphousPrebuilder — QuantumATK NanoLab
Use Packmol to randomly pack atoms into a BulkConfiguration given a list of elements, the stoichiometry and material density. The resulting configuration can be used as a starting point to create an amorphous structure.
Manually set the composition and lattice.
Elements
Element
Stoichiometry
Add row
Clear table
Configuration options
Size multiplier
1
Density option
By number of atoms
Number density
0.05
Å⁻³
Number of atoms
Lattice constant
Packmol options
Random seed
Automatic
Minimum distance
2
Å
Buffer size
2
Å
Maximum loops
60



3. Training new or finetuning existing MACE NN potentials. Learn about training MACE potentials here: https://docs.quantumatk.com/tutorials/mace-training-c-am-TiSi/mace-training-c-am-TiSi.html.

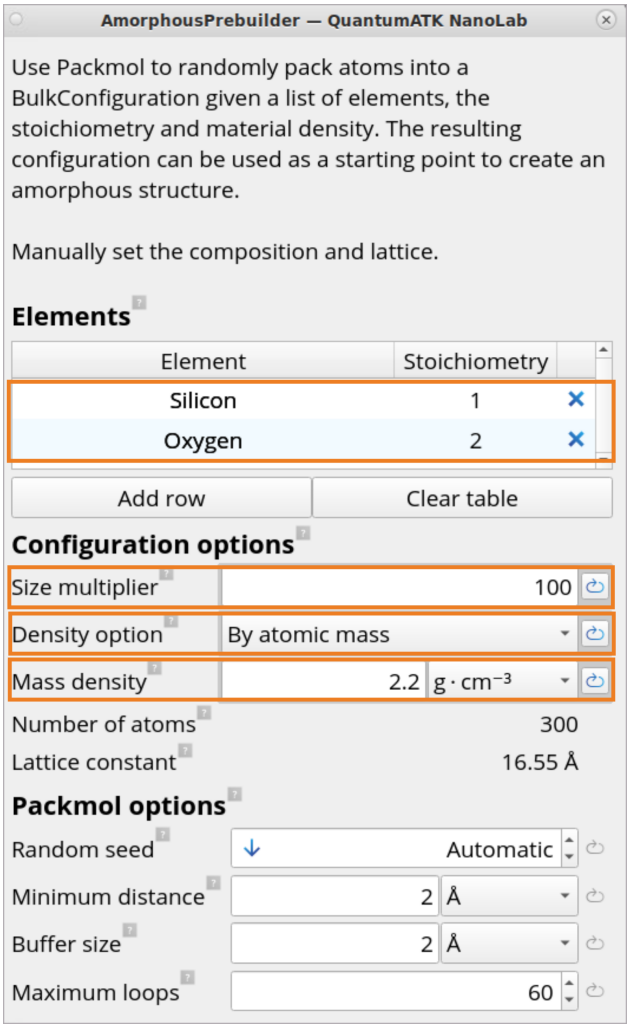
4. Click on the *Melt (Langevin)* block, which contains settings for the Langevin type MD simulations at high (i.e., *5000 K*) temperature to melt the structure. In this example, you will run *30 000* MD steps with a time step of *1 fs*, resulting in a total MD melt simulation time of 30 ps. Use the settings set by the template.

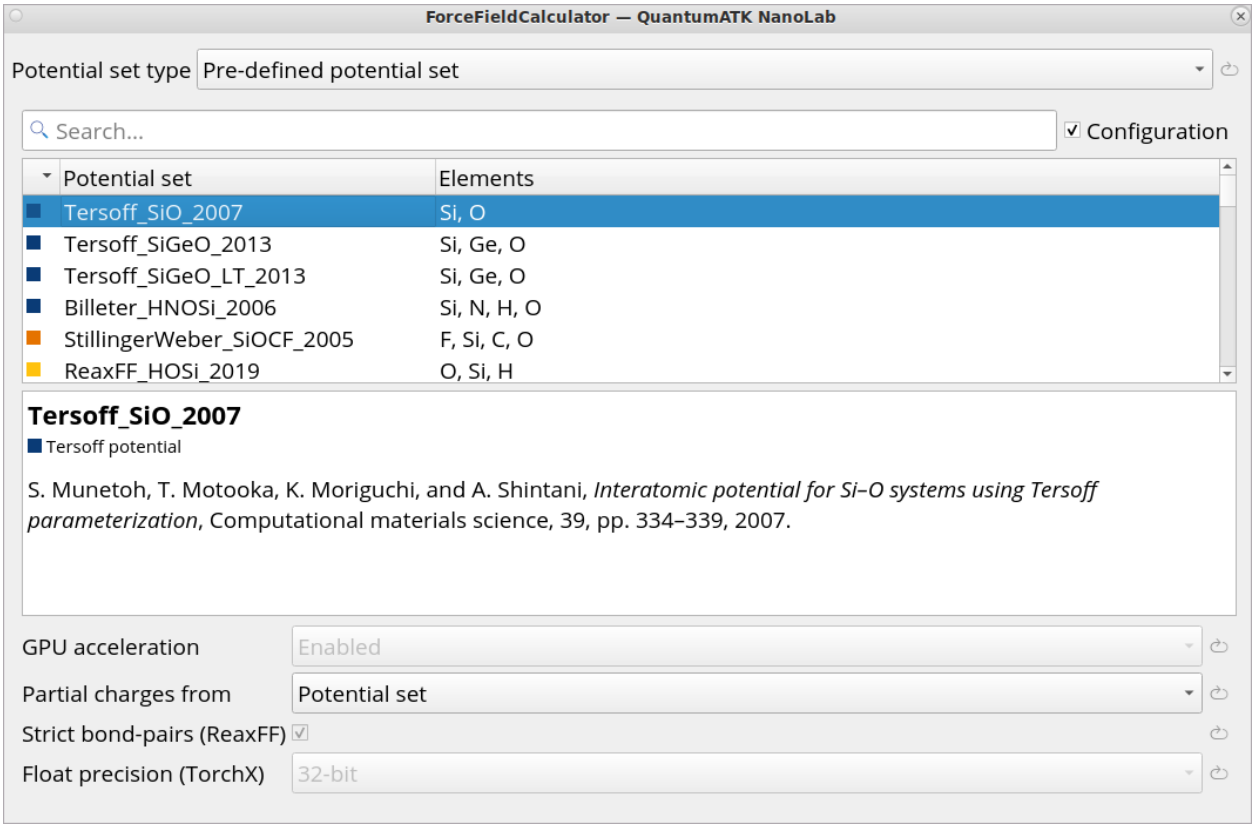
5. Click on the *Constant volume quench (NVT NoseHoover)* block, which contains settings for the NVT Nose Hoover type MD simulations to quench the structure from *5000 K* to *600 K* using *-88* Kps^{-1} *Heating rate* within 50 ps (*50 000* MD steps with a time step of *1 fs*). Use the settings set by the template.

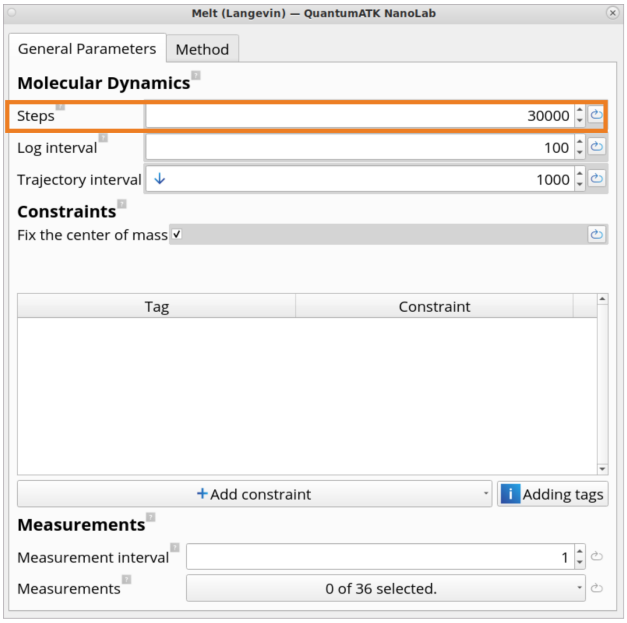
6. Click on the *Constant volume equilibration (NVT NoseHoover)* block, which defines settings for the NVT Nose Hoover type MD equilibration simulations. In this example, you will equilibrate the system for 100 ps at *600 K* in order to reach a stable state where systems temperature and energy are stable, thus allowing us to obtain meaningful RDF and ADF values.

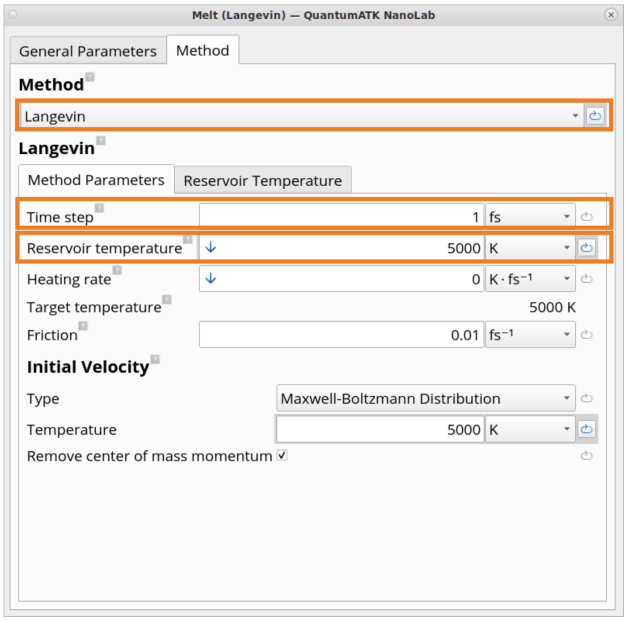
Tip:

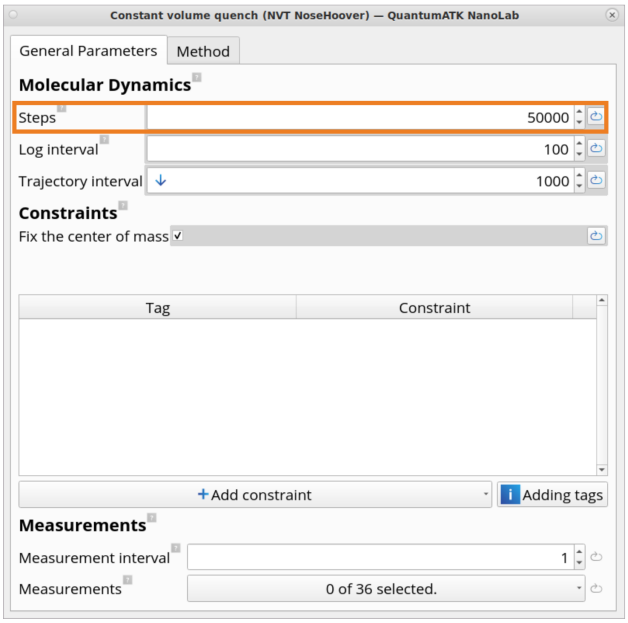
1. In the *Molecular Dynamics* block, one can choose between various Molecular Dynamics methods: *NVE Velocity Verlet*, *NVT Berendsen*, *NVT Bussi Donadio Parrinello*, *NVT Nose Hoover*, *Langevin*, *NPT Berendsen*, *NPT Bernetti Bussi*, *NPT Martyna Tobias Klein*, *Non Equilibrium Momentum Echange* and *Stress Strain*. Read these technical notes on *Molecular Dynamics* to learn more.

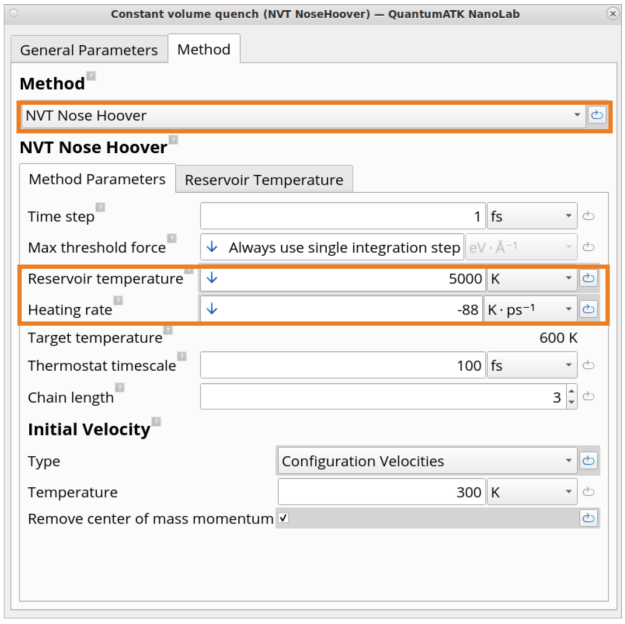

AmorphousPrebuilder — QuantumATK NanoLab
Use Packmol to randomly pack atoms into a BulkConfiguration given a list of elements, the stoichiometry and material density. The resulting configuration can be used as a starting point to create an amorphous structure.
Manually set the composition and lattice.
Elements
Element
Stoichiometry
Silicon
1
Oxygen
2
Add row
Clear table
Configuration options
Size multiplier
100
Density option
By atomic mass
Mass density
2.2
g · cm⁻³
Number of atoms
300
Lattice constant
16.55 Å
Packmol options
Random seed
Automatic
Minimum distance
2
Å
Buffer size
2
Å
Maximum loops
60

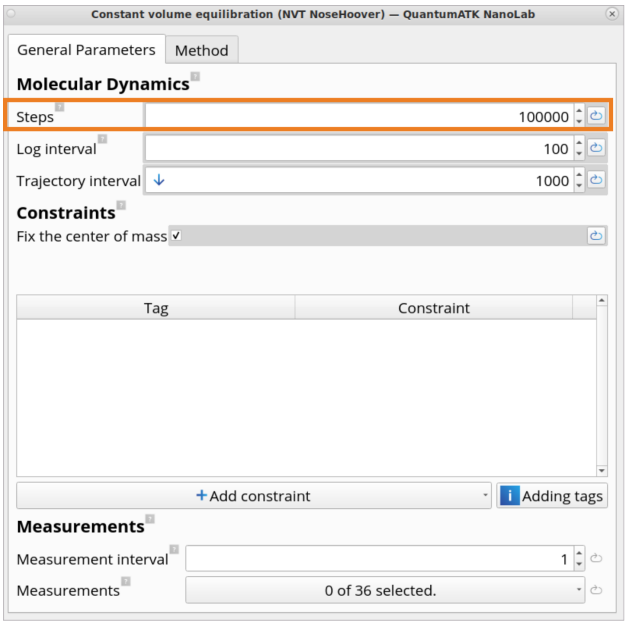

ForceFieldCalculator — QuantumATK NanoLab
Potential set type
Pre-defined potential set
Search...
Configuration
Potential set
Elements
Tersoff_SiO_2007
Si, O
Tersoff_SiGeO_2013
Si, Ge, O
Tersoff_SiGeO_LT_2013
Si, Ge, O
Billeter_HNOSi_2006
Si, N, H, O
StillingerWeber_SiOCF_2005
F, Si, C, O
ReaxFF_HOSi_2019
O, Si, H
Tersoff_SiO_2007
Tersoff potential
S. Munetoh, T. Motooka, K. Moriguchi, and A. Shintani, Interatomic potential for Si–O systems using Tersoff parameterization, Computational materials science, 39, pp. 334–339, 2007.
GPU acceleration
Enabled
Partial charges from
Potential set
Strict bond-pairs (ReaxFF)
Float precision (TorchX)
32-bit

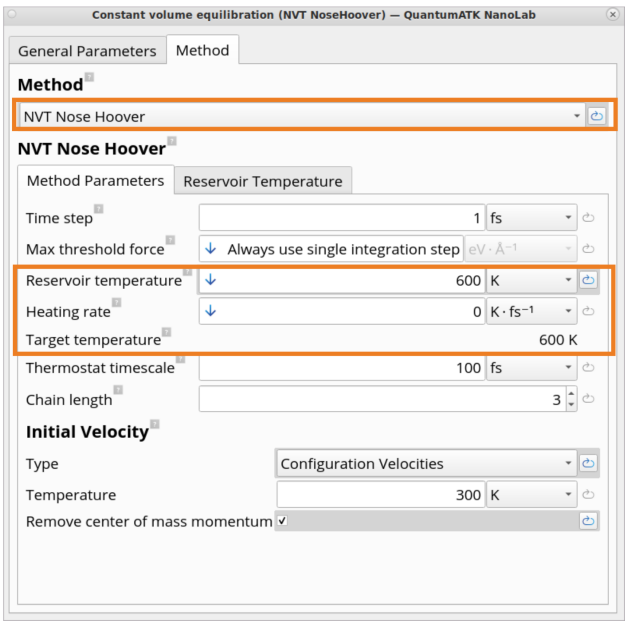

Melt (Langevin) — QuantumATK NanoLab
General Parameters
Method
Molecular Dynamics
Steps
30000
Log interval
100
Trajectory interval
1000
Constraints
Fix the center of mass
Tag
Constraint
Add constraint
Adding tags
Measurements
Measurement interval
1
Measurements
0 of 36 selected.


Melt (Langevin) — QuantumATK NanoLab
General Parameters
Method
Method
Langevin
Langevin
Method Parameters
Reservoir Temperature
Time step
1
fs
Reservoir temperature
5000
K
Heating rate
0
K · fs⁻¹
Target temperature
5000 K
Friction
0.01
fs⁻¹
Initial Velocity
Type
Maxwell-Boltzmann Distribution
Temperature
5000
K
Remove center of mass momentum


Constant volume quench (NVT NoseHoover) — QuantumATK NanoLab
General Parameters
Method
Molecular Dynamics
Steps
50000
Log interval
100
Trajectory interval
1000
Constraints
Fix the center of mass
Tag
Constraint
Add constraint
Adding tags
Measurements
Measurement interval
1
Measurements
0 of 36 selected.

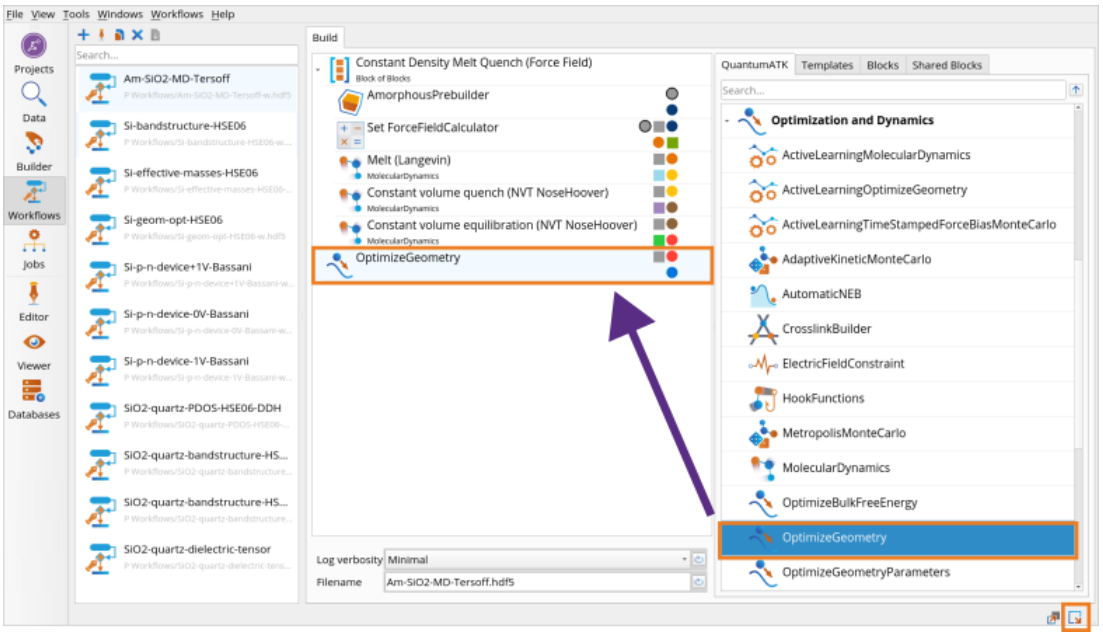

Constant volume quench (NVT NoseHoover) — QuantumATK NanoLab
General Parameters
Method
Method
NVT Nose Hoover
NVT Nose Hoover
Method Parameters
Reservoir Temperature
Time step
1
fs
Max threshold force
Always use single integration step
Reservoir temperature
5000
K
Heating rate
-88
K · ps⁻¹
Target temperature
600 K
Thermostat timescale
100
fs
Chain length
3
Initial Velocity
Type
Configuration Velocities
Temperature
300
K
Remove center of mass momentum


Constant volume equilibration (NVT NoseHoover) — QuantumATK NanoLab
General Parameters
Method
Molecular Dynamics
Steps
100000
Log interval
100
Trajectory interval
1000
Constraints
Fix the center of mass
Tag
Constraint
Add constraint
Adding tags
Measurements
Measurement interval
1
Measurements
0 of 36 selected.


Constant volume equilibration (NVT NoseHoover) — QuantumATK NanoLab
General Parameters
Method
Method
NVT Nose Hoover
NVT Nose Hoover
Method Parameters
Reservoir Temperature
Time step
1
fs
Max threshold force
Always use single integration step
Reservoir temperature
600
K
Heating rate
0
K · fs⁻¹
Target temperature
600 K
Thermostat timescale
100
fs
Chain length
3
Initial Velocity
Type
Configuration Velocities
Temperature
300
K
Remove center of mass momentum

2.In the *Molecular Dynamics* block, it is possible to edit a variety of MD parameters, such as a number of steps, time step, reservoir temperature, heating rate, target temperature, friction, velocity initialization type and temperature.

7.Go to the *Block Catalog* → *QuantumATK* tab → *Optimization and Dynamics* category on the right hand panel and drag-and-drop the *OptimizeGeometry* block to the middle *Workflow Area* → *Build* tab.



8.Click on the *OptimizeGeometry* block to set the *OptimizeGeometry* to obtain the 0K structure of amorphous SiO_2 using the same *Tersoff_SiO_2007* Classical Force Field.

- Set a tighter than default *Force tolerance* (*0.001* eV^{-1}).
- Increase *Maximum number of steps* to *2000*, as many more optimization steps are needed when performing a geometry optimization with classical Force Fields.
- Constrain cell in *x*, *y* and *z* directions and un-click *Constrain Bravais lattice*.

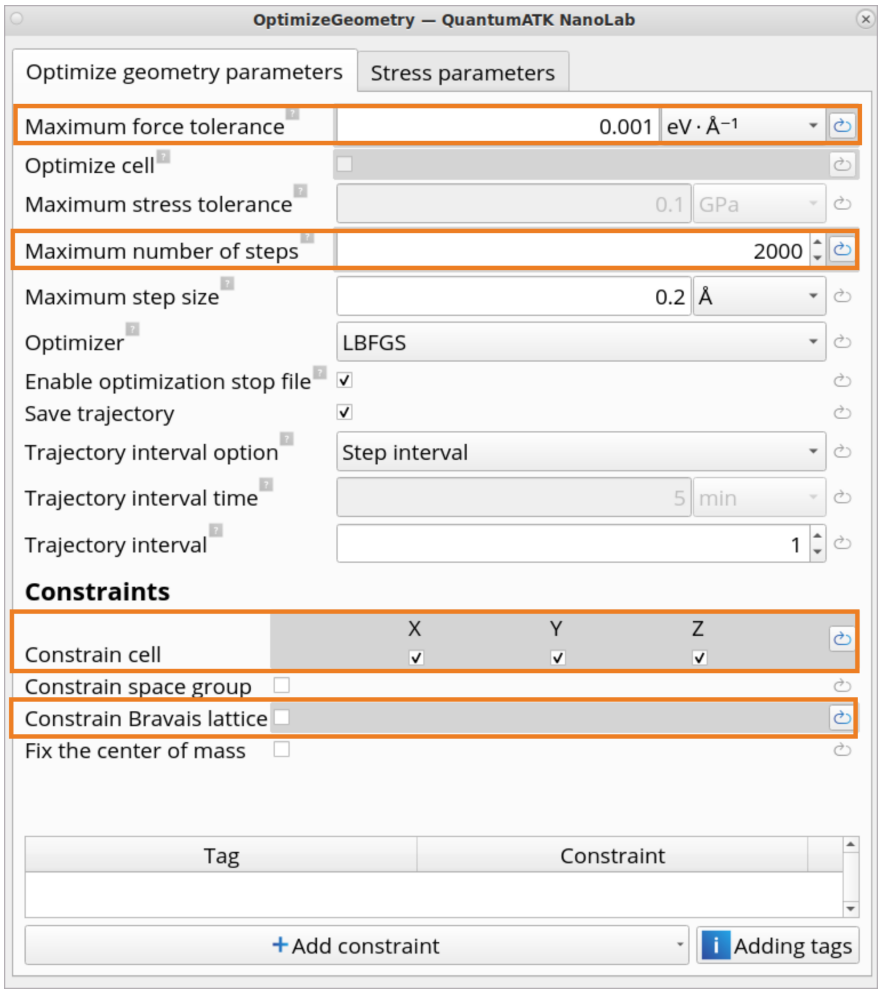
Note:

- Note, that when calculating electronic properties, it is important to perform a prior geometry optimization with DFT to get accurate electronic properties. In such a case, drag-and-drop the *SetLCAOCalculator* and additional *OptimizeGeometry* blocks to the workflow to set up the simulation.

9.Change the *Workflow* name in the *Workflow Stash* and *Filename* as needed (e.g., `Am-SiO2-MD-Tersoff.hdf5`).

10.Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool in order to submit this job on a local machine or a remote cluster. It is recommended to use threading for Force Field simulations.

11.This also generates a Python script `Am-SiO2-MD-Tersoff.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool.

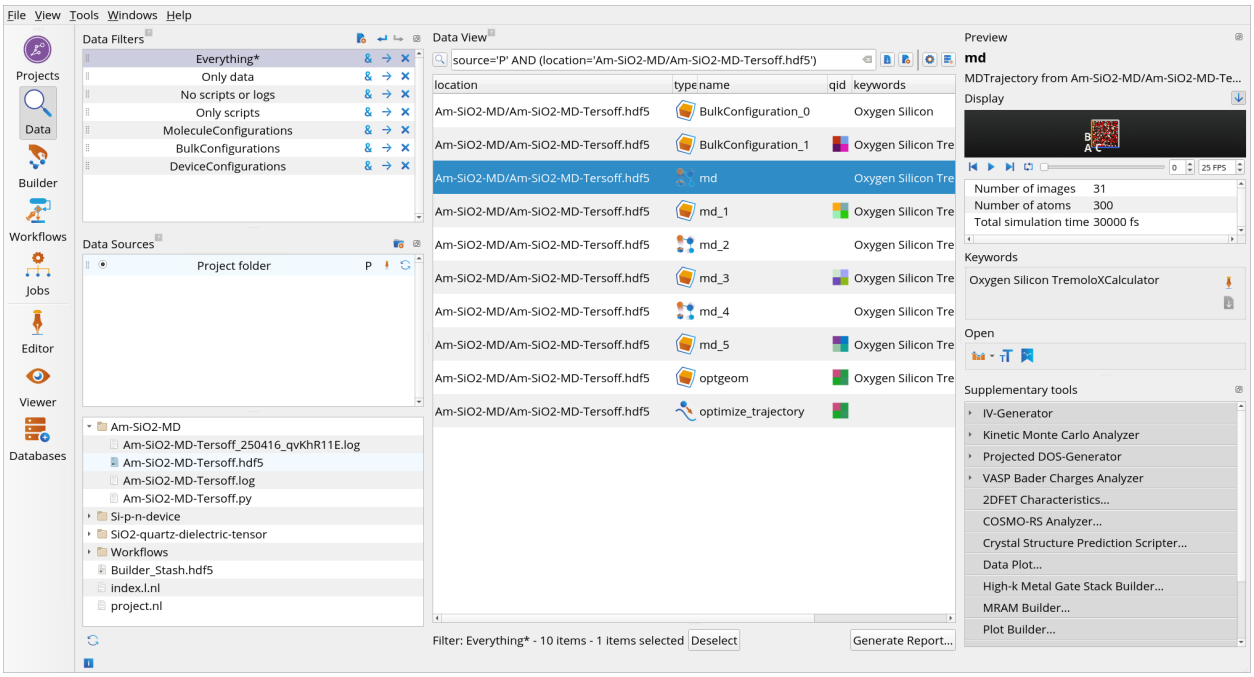

OptimizeGeometry — QuantumATK NanoLab
Optimize geometry parameters
Stress parameters
Maximum force tolerance
0.001
eV · Å⁻¹
Optimize cell
Maximum stress tolerance
0.1
GPa
Maximum number of steps
2000
Maximum step size
0.2
Å
Optimizer
LBFGS
Enable optimization stop file
Save trajectory
Trajectory interval option
Step interval
Trajectory interval time
5
min
Trajectory interval
1
Constraints
X
Y
Z
Constrain cell
Constrain space group
Constrain Bravais lattice
Fix the center of mass
Tag
Constraint
Add constraint
Adding tags

Analyzing Molecular Dynamics Simulations

Once the calculation has finished, we are ready to analyze the results.

1. Go to the **Data View** tool.

2. In the *Data Sources*, select the file `Am-SiO2-MD-Tersoff.hdf5` and then in the *Data View* window double-click on the *md* object from the melt Langevin MD to open it with the *Movie Tool*.



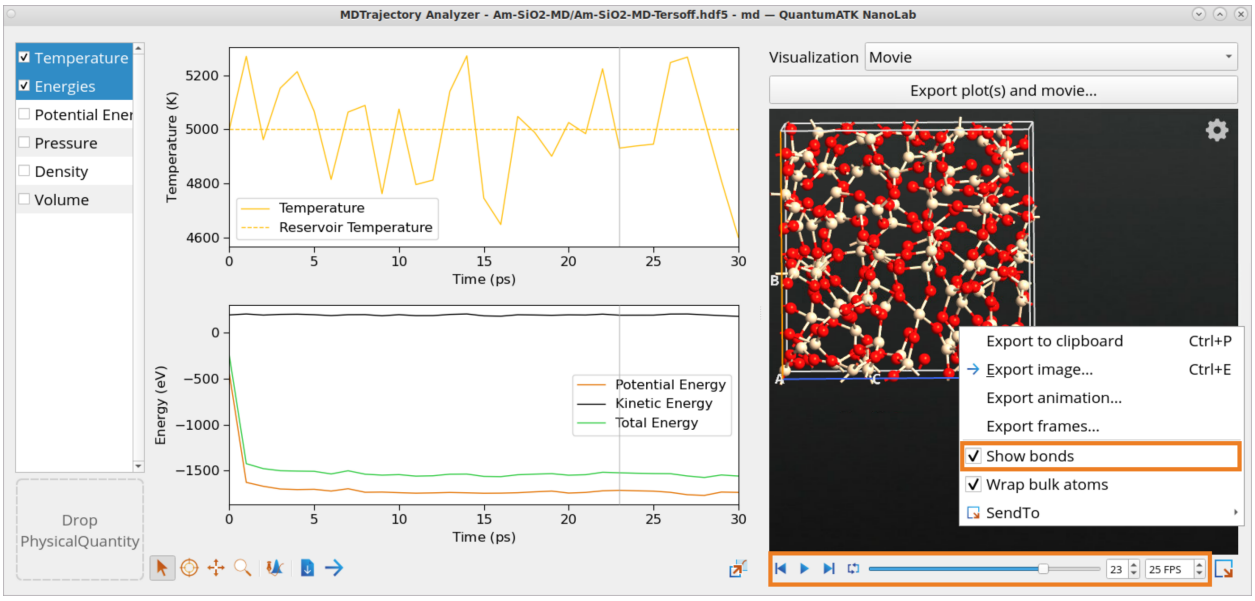
3. Use the *Movie Tool* to investigate the temperature as well as the kinetic, potential and total energies, pressure, density and volume of the system as a function of simulation time, 30 ps in this case. You will observe the temperature oscillating around the set temperature (5000 K).

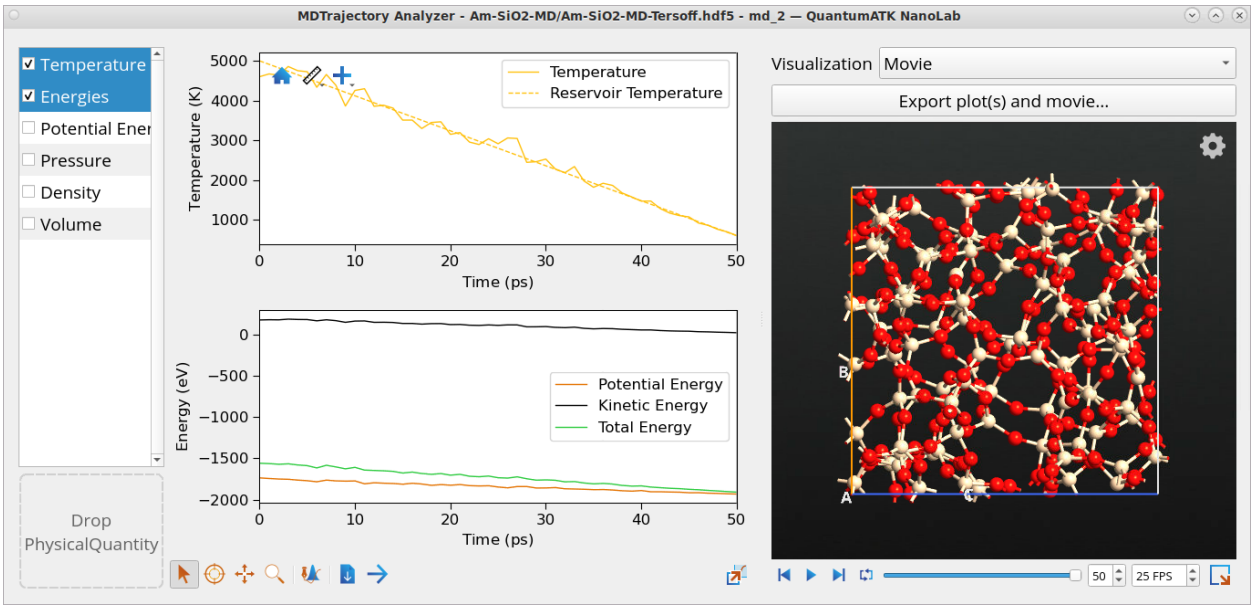
4. Simultaneously, an animation of the motion of the atoms is displayed. Right-click on the structure in the *Movie tool* to check the box *Show bonds* and use the *Play* button to see the evolution of structure throughout the MD simulation.

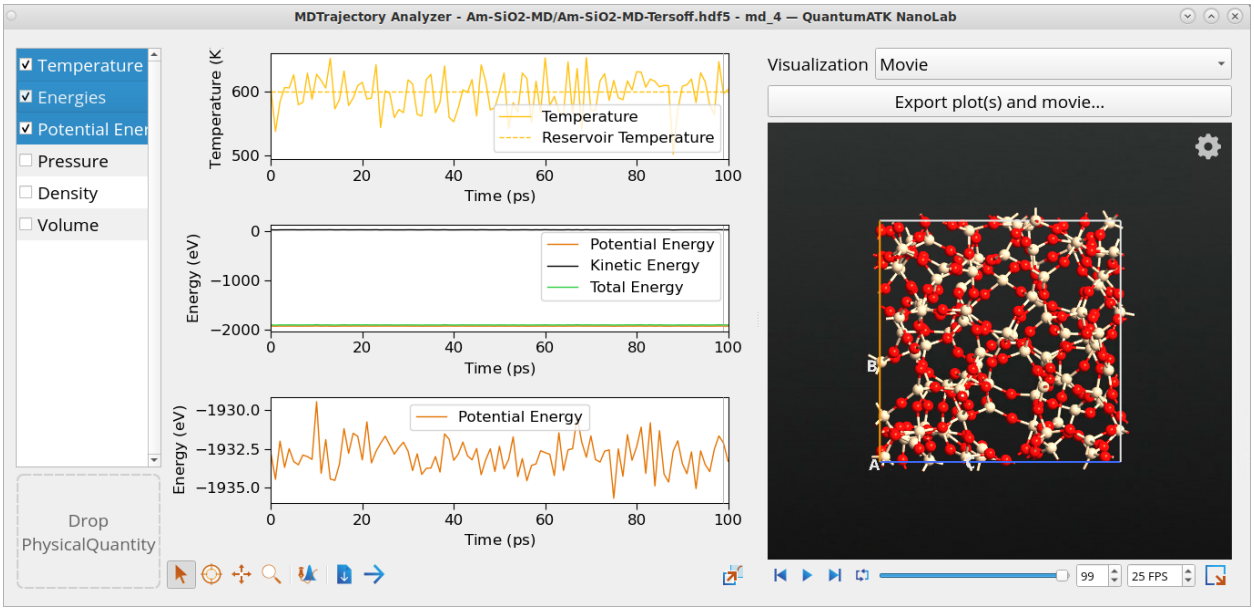
5. Come back to the *Data View* window and double-click on the *md_2* object from the quench NVT Nose Hoover MD to open it with the *Movie Tool*.

6. Use the *Movie Tool* to investigate the temperature as well as the kinetic, potential and total energies, pressure, density and volume of the system as a function of simulation time, 50 ps in this case. The temperature is gradually reduced from 5000 K to ~ 600 K as set in the workflow.

7. Come back to the *Data View* window and double-click on the *md_4* object from the NVT Nose Hoover type MD equilibration simulations to open it with the *Movie Tool*. The temperature fluctuates around the set value of 600 K, and the energies also fluctuate around their equilibrium values, indicating that the system is stable. This allows us to obtain meaningful RDF and ADF values from this trajectory.


MDTrajectory Analyzer - Am-SiO2-MD/Am-SiO2-MD-Tersoff.hdf5 - md — QuantumATK NanoLab
Temperature
Energies
Potential Ener
Pressure
Density
Volume
Drop PhysicalQuantity
Temperature (K)
Temperature
Reservoir Temperature
Time (ps)
Energy (eV)
Potential Energy
Kinetic Energy
Total Energy
Visualization
Movie
Export plot(s) and movie...
Export to clipboard Ctrl+P
Export image... Ctrl+E
Export animation...
Export frames...
Show bonds
Wrap bulk atoms
SendTo
23
25 FPS


MDTrajectory Analyzer - Am-SiO2-MD/Am-SiO2-MD-Tersoff.hdf5 - md_2 — QuantumATK NanoLab
Temperature
Energies
Potential Ener
Pressure
Density
Volume
Drop PhysicalQuantity
Temperature (K)
Temperature
Reservoir Temperature
Time (ps)
Energy (eV)
Potential Energy
Kinetic Energy
Total Energy
Visualization
Movie
Export plot(s) and movie...
50
25 FPS



8. Come back to the *Data View* window and right-click on the *md_4* object from the NVT Nose Hoover type MD equilibration simulations to open it with the *Molecular Dynamics Analyzer* to plot calculated RDF and ADF for amorphous SiO_2.

9. In order to plot RDF:

- Click on *Analysis type* selection box and choose to plot *Radial Distribution*.
- Click on *Selection 1* and *Selection 2* boxes to choose for which element combinations to plot RDF.
- Click *Plot* to generate RDF plots for Si-O, Si-Si and O-O one-by-one.
- Click on the *Inspect* tool to examine the exact RDF values.

Note: Calculated radius values for the 1st coordination sphere obtained from the RDF plot: r_{Si-O} = 1.675 Å, r_{Si-Si} = 3.175 Å and r_{O-O} = 2.675 Å are in a reasonably good agreement with experimental values for fused silica at 300 K: $r_{Si-O_{exp}}$ = 1.62 Å, $r_{Si-Si_{exp}}$ = 3.12 Å and $r_{O-O_{exp}}$ = 2.65 Å[1].

---

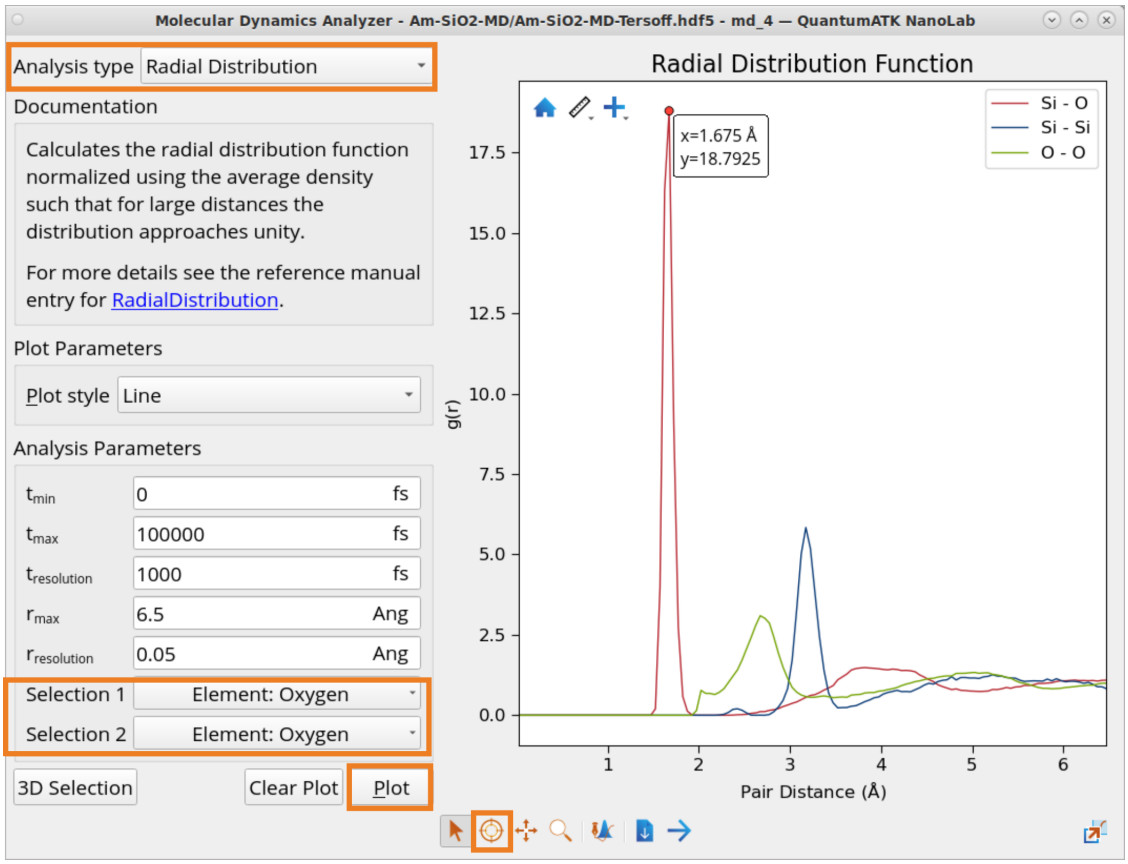
10. In order to plot ADF:

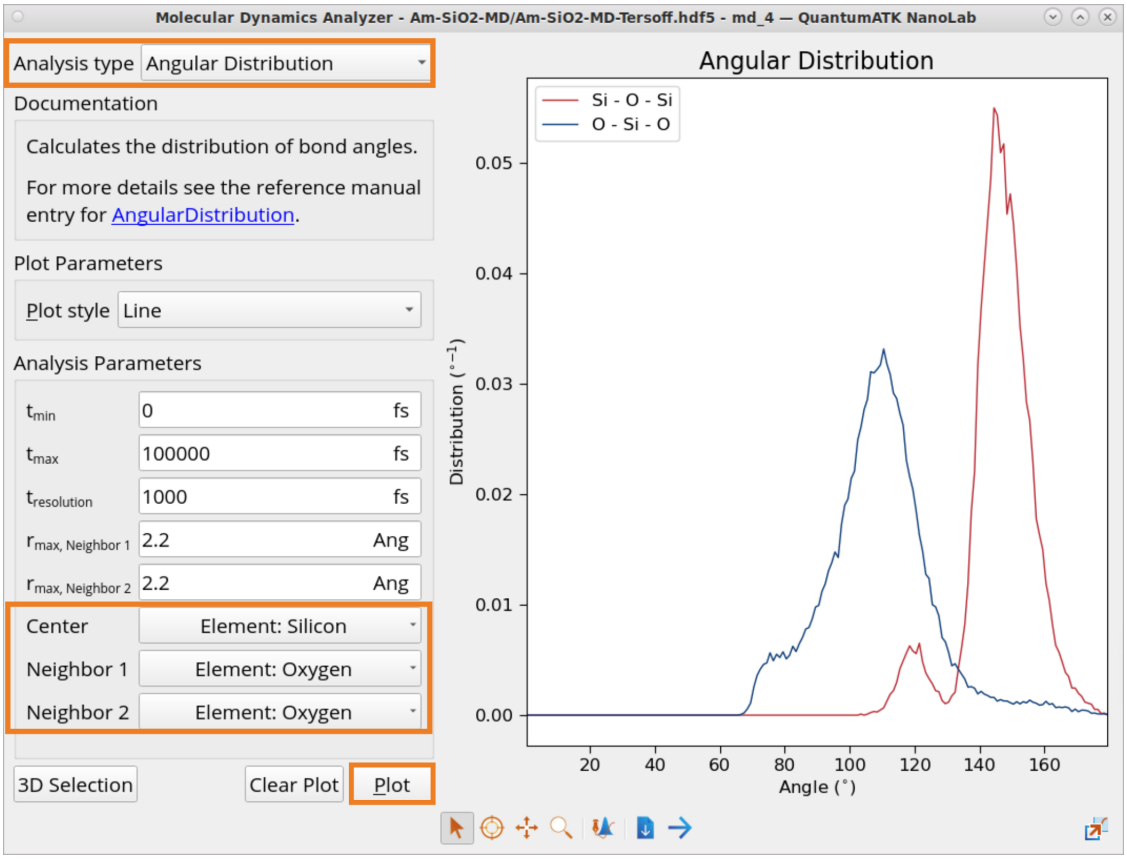
- Click on *Analysis type* selection box and choose to plot *Angular Distribution*.
- Click on *Center*, *Neighbor 1* and *Neighbor 2* boxes to choose for which element combinations to plot ADF.
- Click *Plot* to generate ADF plots for Si-O-Si and O-Si-O one-by-one.

Note: Calculated angle values for all the coordination spheres obtained from the ADF plot: θ_{O-Si-O} = 110.5ř and $\theta_{Si-O-Si}$ = 144.5ř are also in a reasonably good agreement with experimental values for fused silica at 300 K: $\theta_{O-Si-O_{exp}}$ = 109.7ř and $\theta_{Si-O-Si_{exp}}$ = 152.0ř[2].

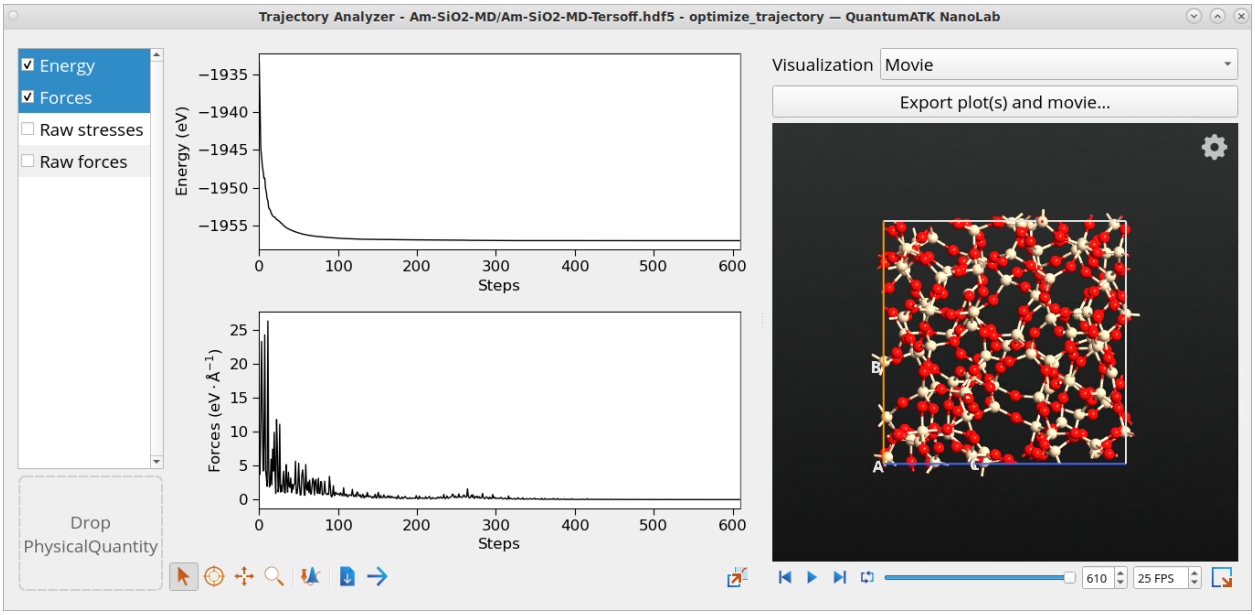
---

[1] O.ăV.ăMazurin, M.ăV.ăStreltsina and T.ăP. Shvaiko-Shvaikovskaya, editors. *Handbook of Glass Data*. Elsevier, 1983.
[2] R.ăZallen, editor. *The Physics of Amorphous Solids*. John Wiley and Sons, New York, 1983.


Molecular Dynamics Analyzer - Am-SiO2-MD/Am-SiO2-MD-Tersoff.hdf5 - md_4 — QuantumATK NanoLab
Analysis type
Radial Distribution
Documentation
Calculates the radial distribution function normalized using the average density such that for large distances the distribution approaches unity.
For more details see the reference manual entry for RadialDistribution.
Plot Parameters
Plot style
Line
Analysis Parameters
t_min 0 fs
t_max 100000 fs
t_resolution 1000 fs
r_max 6.5 Ang
r_resolution 0.05 Ang
Selection 1
Element: Oxygen
Selection 2
Element: Oxygen
3D Selection
Clear Plot
Plot
Radial Distribution Function
x=1.675 Å
y=18.7925
Si - O
Si - Si
O - O
g(r)
Pair Distance (Å)


Molecular Dynamics Analyzer - Am-SiO2-MD/Am-SiO2-MD-Tersoff.hdf5 - md_4 — QuantumATK NanoLab
Analysis type
Angular Distribution
Documentation
Calculates the distribution of bond angles.
For more details see the reference manual entry for AngularDistribution.
Plot Parameters
Plot style
Line
Analysis Parameters
t_min 0 fs
t_max 100000 fs
t_resolution 1000 fs
r_max, Neighbor 1 2.2 Ang
r_max, Neighbor 2 2.2 Ang
Center
Element: Silicon
Neighbor 1
Element: Oxygen
Neighbor 2
Element: Oxygen
3D Selection
Clear Plot
Plot
Angular Distribution
Si - O - Si
O - Si - O
Distribution (°⁻¹)
Angle (°)

11. Come back to the *Data View* window and right-click on the *optimize_trajectory* object from the final geometry optimization at 0 K and to see the evolution of structure throughout the geometry optimization simulation. The geometry converged in 611 steps.



12. Right-click on the the optimized geometry *optgeom* in *Data View* and choose to *Open with Builder*. The optimized geometry will be automatically saved in the **Builder** *Stash*, so you could use it for other simulations.

References

1.2.5 NEGF Simulations of Electron Transport in Devices

Version: X-2025.06

In this tutorial, you will learn how to use QuantumATK NanoLab GUI to

- Set-up and analyze electron transport simulations in the Si p-n junction device using the Non-equilibrium Greens function (NEGF) method with the Semi-Empirical Tight-Binding model.
- Analyze results, such as Hartree difference potential, projected local density of states (PLDOS), transmission spectrum and spectral current.

All the workflow, input and output files can be found in the **Example Project - X-2025.06 → Si-p-n-device folder.**

Prerequisites

•Video Introduction to QuantumATK NanoLab GUI

•Video tutorial on the Workflow Builder tool

•Video tutorial on the Job tool

•Tutorial *Geometry Optimization*

•Tutorial *Band Structure, Projected Density of States and Effective Mass Calculations*

•Manual *NEGF: Device Calculators*

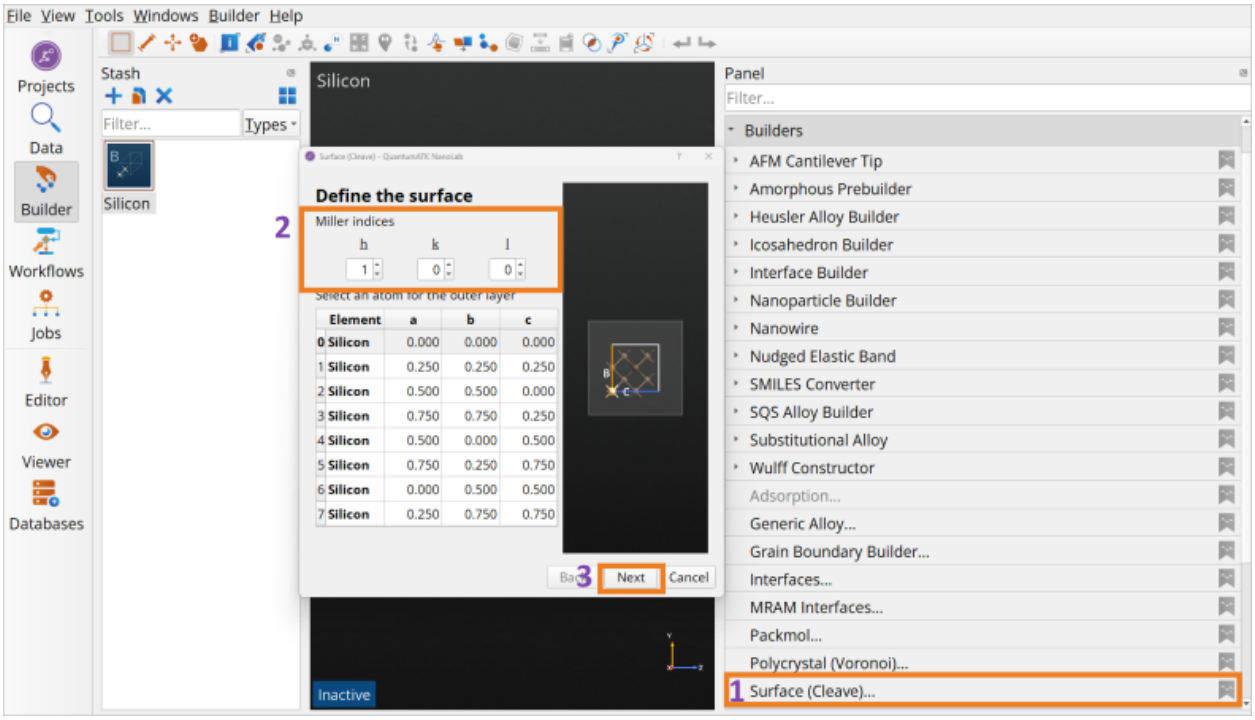
Building Si p-n Junction Device

1.Follow the tutorial *Geometry Optimization* to import the structure of bulk Silicon (alpha) from NanoLabs *Crystal Database* to the **Builder**.

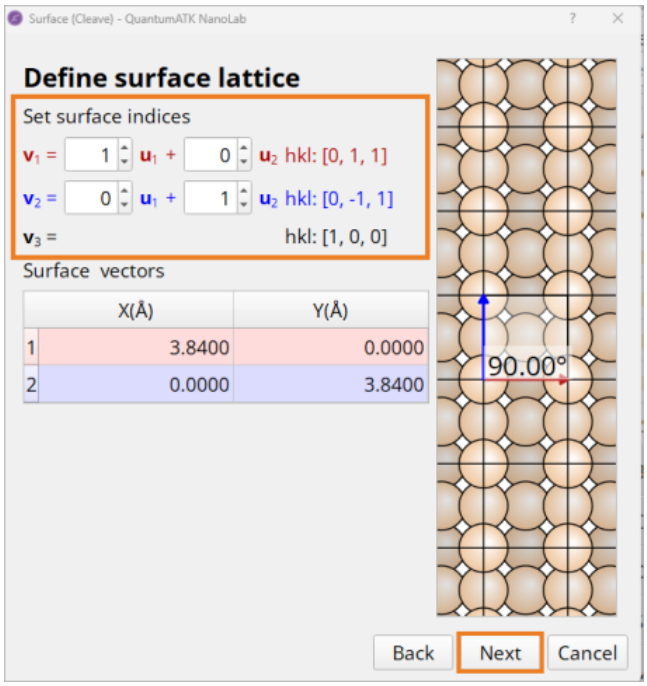
2.Create a Si(100) surface using the **Builder** tool.

•Navigate to *Builders* → *Surface(Cleave)* in the *Builder Panel* (1).

•In the *Surface(Cleave)* window, choose the [1,0,0] Miller indices (2) and click *Next* (3).



•Keep the 1Œ1 surface unit cell and click *Next*.



•Select a thickness of 78 layers and click *Finish*.

Note: The 78 layers will constitute the central region of your device as explained in the NEGF manual *NEGF: Device Calculators*. You need such a relatively long device (~20 nm) because of the long screening length in the silicon semiconductor.

3.Construct a device with the transport direction along (100).

•Navigate to *Device Tools* → *Device From Bulk* in the *Builder Panel*.

•Keep the default parameters for minimal electrode lengths and electrode extension lengths.

•Click *OK*.

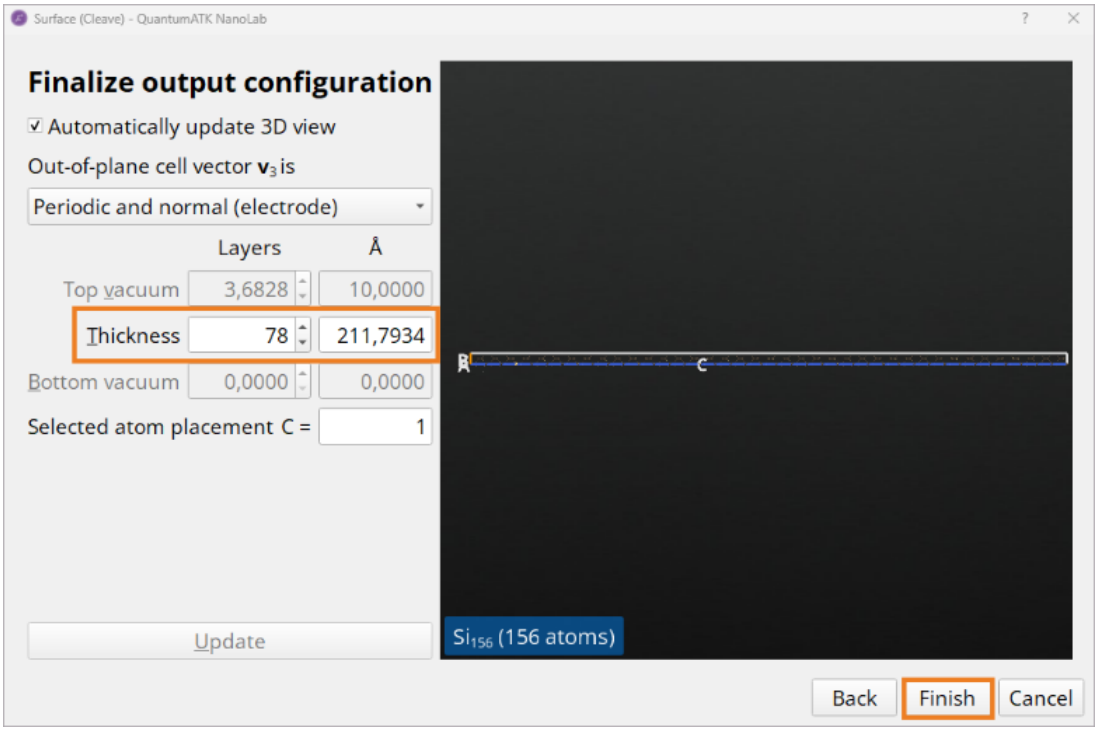
•The created device configuration is automatically saved in the *Builder Stash* and has the central region, left/right electrodes and lef/right electrode extensions.

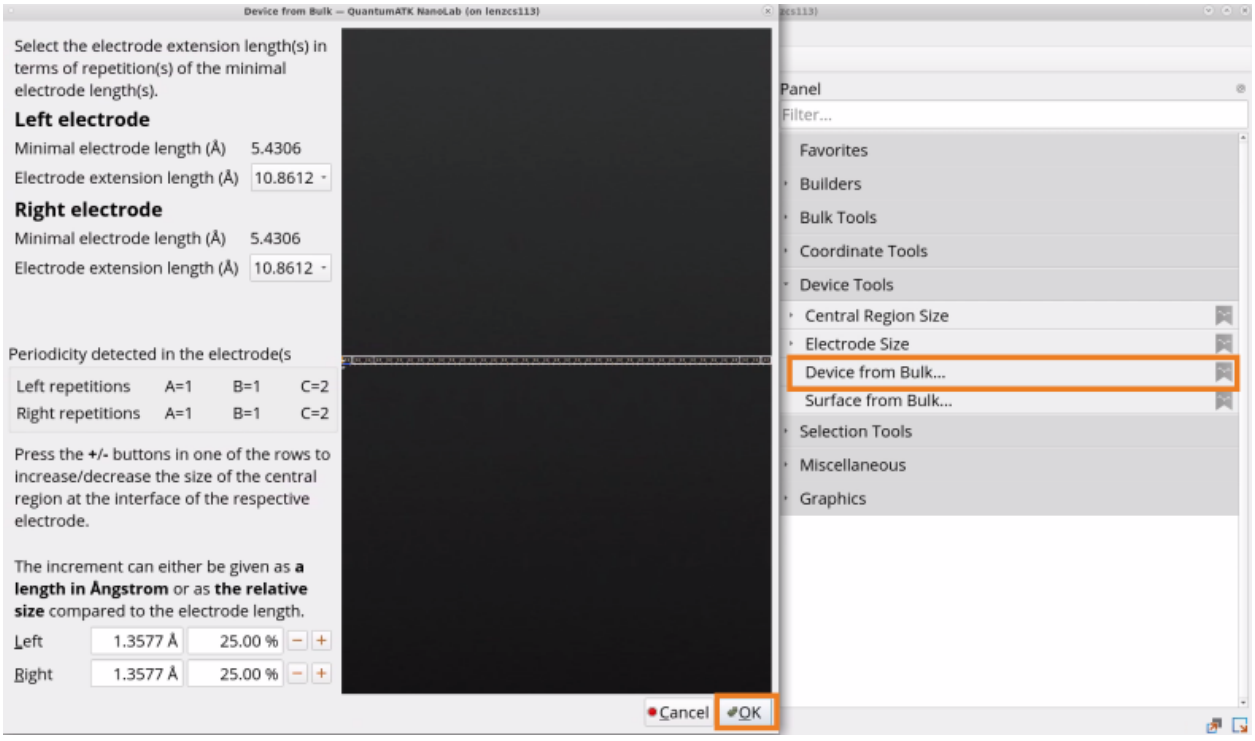
Tip:

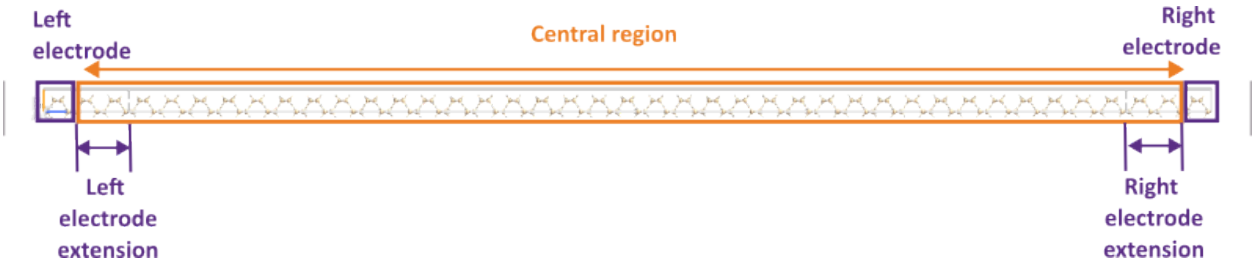
1.When using a Semi-Empirical Tight-Binding models for NEGF electron transport, one can use the electrode extensions as small as possible, in particular for Nearest Neighbor models, such as Bassani.

2.When performing DFT-NEGF simulations, it is recommended to use longer electrode extensions. The required length will depend on the basis set and xc-functional. In particular when using hybrid DFT functionals - electrode extensions should be 2-3 times longer than for non-hybrid functionals.

3.QuantumATK will print a warning if the elecrode extension is too short.

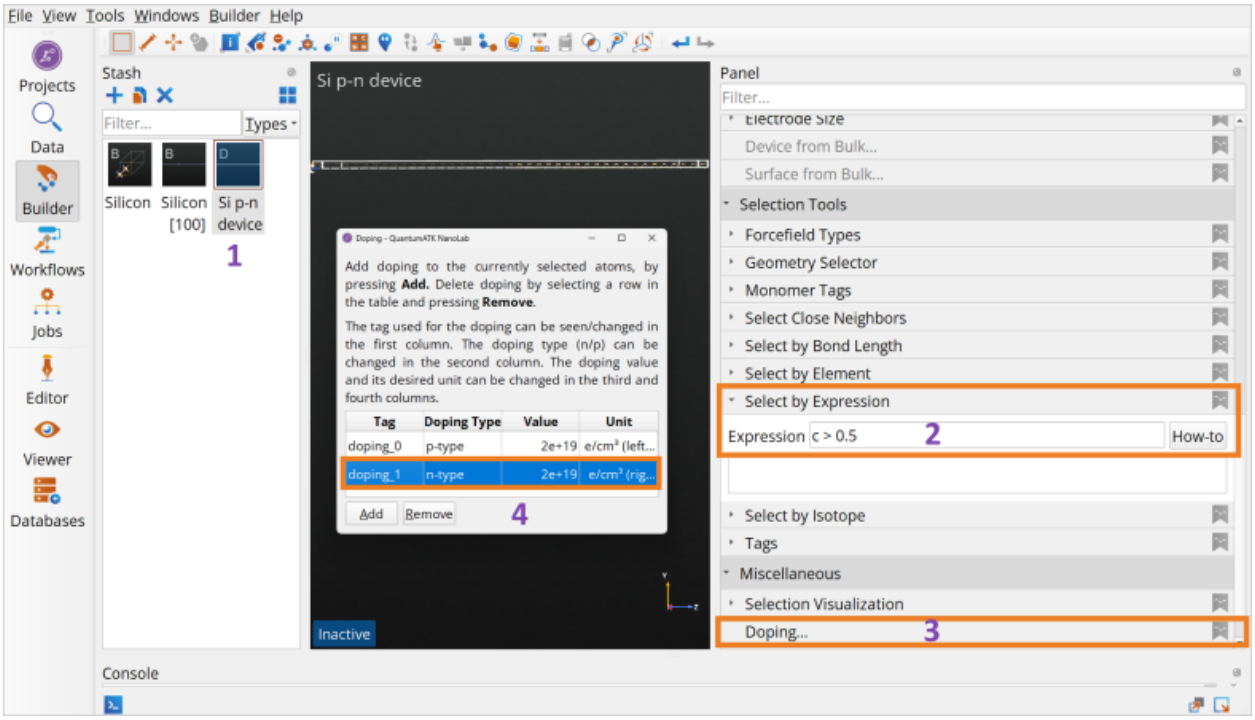

Surface (Cleave) - QuantumATK NanoLab
Finalize output configuration
Automatically update 3D view
Out-of-plane cell vector $\mathbf{v}_3$ is
Periodic and normal (electrode)
Layers
Å
Top vacuum
3,6828
10,0000
Thickness
78
211,7934
Bottom vacuum
0,0000
0,0000
Selected atom placement C =
1
Update
B
C
A
Si_{156} (156 atoms)
Back
Finish
Cancel


Device from Bulk — QuantumATK NanoLab (on lenzcs113)
Select the electrode extension length(s) in terms of repetition(s) of the minimal electrode length(s).
Left electrode
Minimal electrode length (Å) 5.4306
Electrode extension length (Å) 10.8612
Right electrode
Minimal electrode length (Å) 5.4306
Electrode extension length (Å) 10.8612
Periodicity detected in the electrode(s
Left repetitions A=1 B=1 C=2
Right repetitions A=1 B=1 C=2
Press the +/- buttons in one of the rows to increase/decrease the size of the central region at the interface of the respective electrode.
The increment can either be given as a length in Ångstrom or as the relative size compared to the electrode length.
Left 1.3577 Å 25.00 %
Right 1.3577 Å 25.00 %
Cancel
OK
Panel
Filter...
Favorites
Builders
Bulk Tools
Coordinate Tools
Device Tools
Central Region Size
Electrode Size
Device from Bulk...
Surface from Bulk...
Selection Tools
Miscellaneous
Graphics



4.Set up p/n doping to have a Si p-n junction.

- Highlight the device configuration in the *Builder Stash* (1).
- Go to *Selection Tools* → *Select By Expression* in the *Builder Panel* (2).
- Type in $c < 0.5$ and press *Enter* to select all atoms in the left hand side of the device (2).
- Go to *Miscellaneous* → *Doping* in the *Builder Panel* (3).
- Click *Add* to add doping for the left side, set the doping type (p-type) and the doping concentration (Value) of $2 \cdot 10^{19}$ cm^{-3} (4).
- Follow the same procedure to select atoms with $c > 0.5$ to select all atoms in the right hand side of the device (2) and set the doping type (n-type) and doping concentration of $2 \cdot 10^{19}$ cm^{-3} to those atoms (4).



Setting up Electron Transport Simulations for the Si p-n Junction at 0 V Bias

You will use the Non-equilibrium Greens function (NEGF) method *NEGF: Device Calculators* with Semi-Empirical Tight-Binding Bassani model parametrized for Si to calculate the *ProjectedLocalDensityOfStates* (PLDOS), *HartreeDifferencePotential* and *TransmissionSpectrum* for the p-n doped silicon junction at 0 bias. Calculating these properties will give you insights into the Si band gap on the p- and n-type sides of the device, band bending, depletion layer width and potential barrier at the p-n interface, left and right electrode chemical potentials and transmission spectrum, which can then be related to the current in the device.

To create a workflow for electron transport NEGF calculations at 0 V bias in the **Workflow Builder**, follow the step-by-step protocol described below.

1. Click on the *Send content to other tools* button to send the created device configuration from the **Builder** to the **Workflow Builder**.

2. Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* category on the right hand panel and drag-and-drop the *Set DeviceSemiEmpiricalCalculator* block to the middle *Workflow Area* → *Build* tab. Set the following parameters:

   - Use the default *Slater-Koster Hamiltonian*.
   - Leave the default *Bassani.Si Parameter set*
   - Uncheck *No SCF Iteration* box.
   - Use the default medium k-point density of 4 Å x 4 Å x 150 Å, which will result in a sampling of 7 x 7 x 174.
   - Under *Poisson Solver*, leave the default *FFT2D Solver*, which applies *Periodic* boundary conditions in A and B directions and open *Dirichlet* boundary conditions in the transport C direction.
   - Under *Electrode*, leave the default 0V bias voltages and 300 K temperatures for electrodes.

Tip:

1. Many k-points are needed in the transport (Z/C) direction in the electrode calculation to match the electronic structure of the electrodes and the central region. But it is relatively inexpensive since these k-points are only used in the electrode calculation, which is the smaller and faster part of the full computation.
2. Often you can use Semi-Empirical models in NEGF simulations, which are faster than DFT-NEGF, but DFT is more generally applicable and allows simulation of devices without the restriction of an available Semi-empirical model.

---

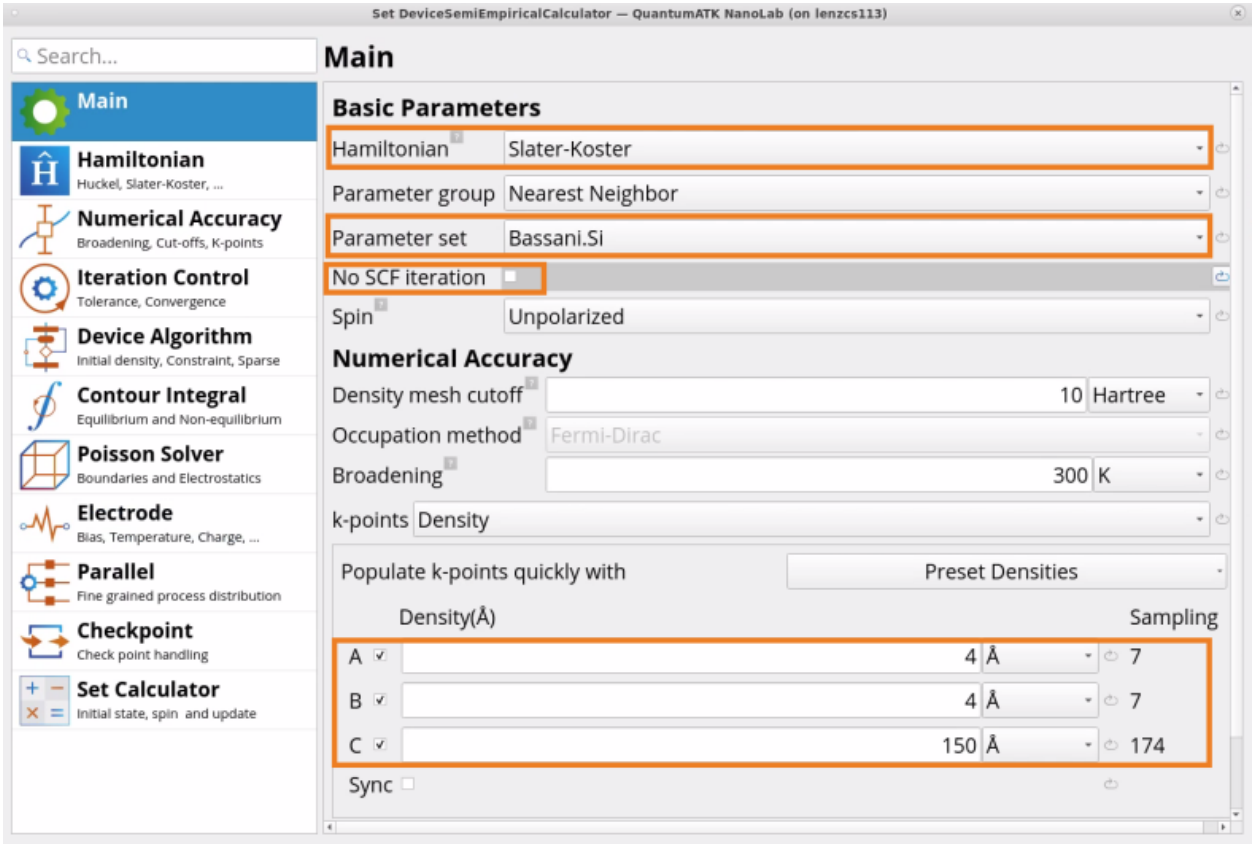
3. Go to the *Block Catalog* → *QuantumATK* tab → *Analysis* category on the right hand panel and drag-and-drop the following analysis object blocks to the middle *Workflow Area* → *Build* tab.

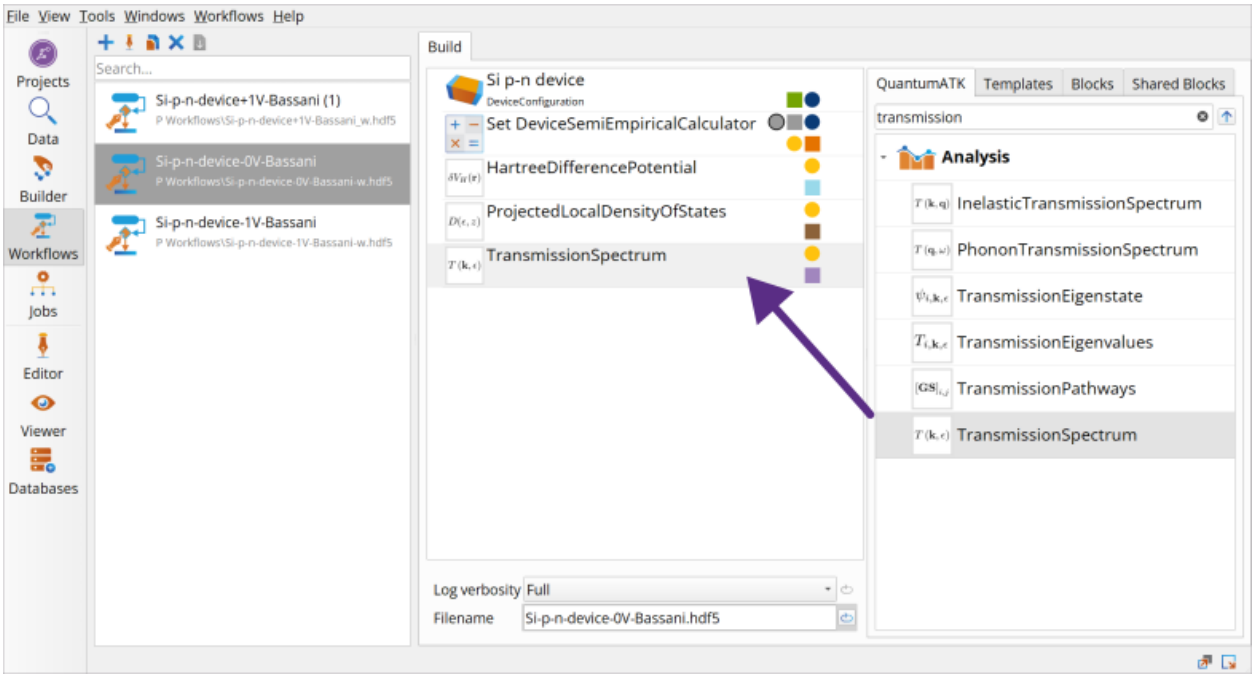
   - *HartreeDifferencePotential* block
   - *ProjectedLocalDensityOfStates* block
   - *TransmissionSpectrum* block.

4. *HartreeDifferencePotential* has no adjustable parameters.

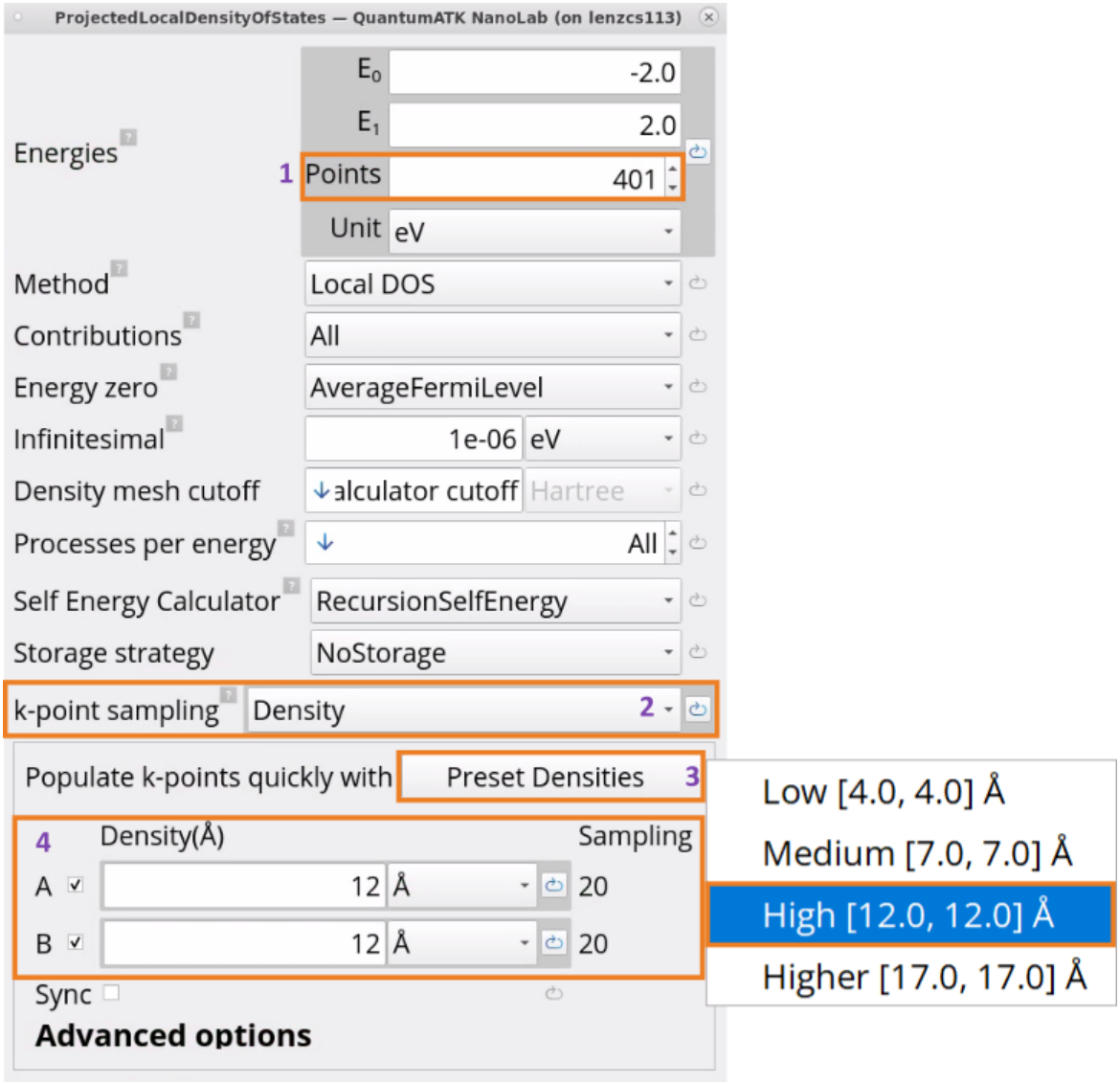
5. Edit the *ProjectedLocalDensityOfStates* block parameters:

   - For the *Energies*, increase the number of points from 200 to 401 in the -2.0 to +2.0 eV energy range for finer sampling.


Set DeviceSemiEmpiricalCalculator — QuantumATK NanoLab (on lenzcs113)
Search...
Main
Hamiltonian
Huckel, Slater-Koster, ...
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Device Algorithm
Initial density, Constraint, Sparse
Contour Integral
Equilibrium and Non-equilibrium
Poisson Solver
Boundaries and Electrostatics
Electrode
Bias, Temperature, Charge, ...
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
Main
Basic Parameters
Hamiltonian
Slater-Koster
Parameter group
Nearest Neighbor
Parameter set
Bassani.Si
No SCF iteration
Spin
Unpolarized
Numerical Accuracy
Density mesh cutoff
10
Hartree
Occupation method
Fermi-Dirac
Broadening
300
K
k-points
Density
Populate k-points quickly with
Preset Densities
Density(Å)
Sampling
A
4
Å
7
B
4
Å
7
C
150
Å
174
Sync


File View Tools Windows Workflows Help
Projects
Data
Builder
Workflows
Jobs
Editor
Viewer
Databases
Search...
Si-p-n-device+1V-Bassani (1)
Si-p-n-device-0V-Bassani
Si-p-n-device-1V-Bassani
Build
Si p-n device
DeviceConfiguration
Set DeviceSemiEmpiricalCalculator
HartreeDifferencePotential
ProjectedLocalDensityOfStates
TransmissionSpectrum
QuantumATK
Templates
Blocks
Shared Blocks
transmission
Analysis
InelasticTransmissionSpectrum
PhononTransmissionSpectrum
TransmissionEigenstate
TransmissionEigenvalues
TransmissionPathways
TransmissionSpectrum
Log verbosity
Full
Filename
Si-p-n-device-0V-Bassani.hdf5

•For the *k-point sampling*, choose *Density* and click on *Preset Densities* to increase the k-point density from *medium* to *high*, i.e., 12 Å x 12 Å, which will result in a sampling of 20 x 20.



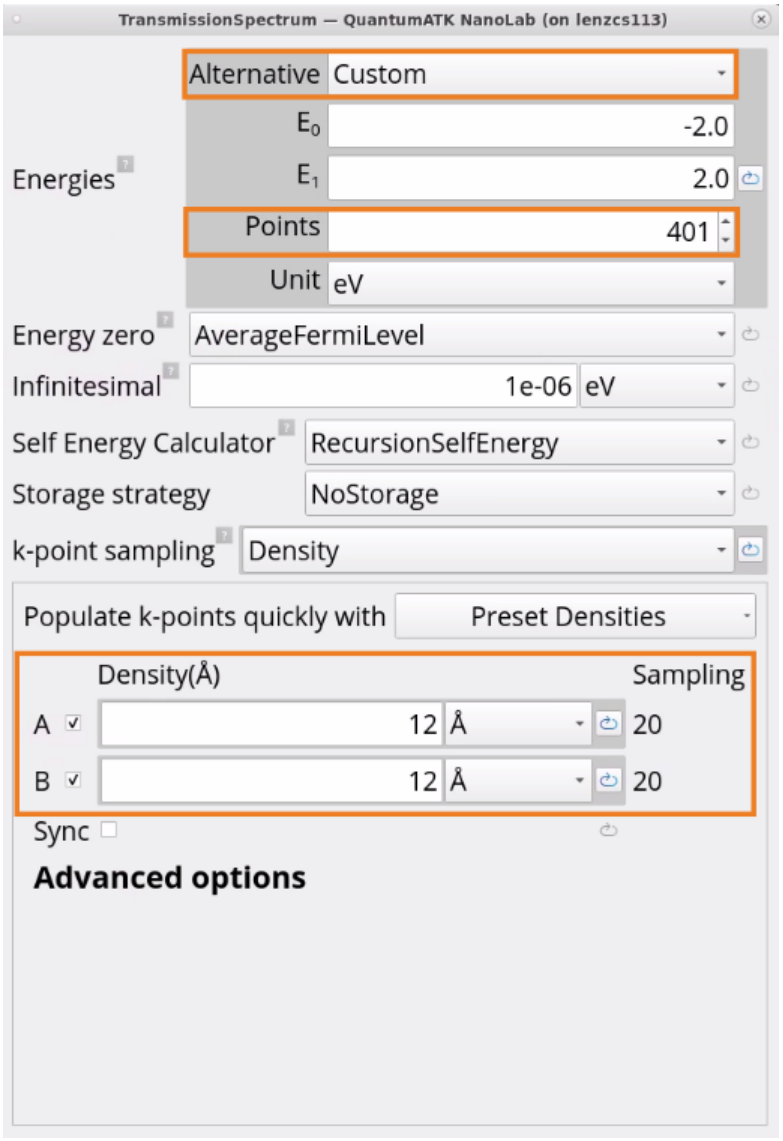
6.Edit the *TransmisionSpectrum* block paramaters:

•For the *Energies*, choose *Custom* and increase the number of points from 200 to 401 in the -2.0 to +2.0 eV energy range for finer sampling.

•For the *k-point sampling*, choose *Density* and click on *Preset Densities* to increase the k-point density from *medium* to *high*, i.e., 12 Å x 12 Å, which will result in a sampling of 20 x 20.

Note: It is important to have a large k-point sampling for electron transport properties, such as projected local density of states and transmission spectrum, as they depend on the conduction band minimum.

7.Change the *Workflow* name and *Filename* as needed (e.g., Si-p-n-device-0V-Bassani.hdf5)


TransmissionSpectrum — QuantumATK NanoLab (on lenzcs113)
Alternative
Custom
Energies
E0
-2.0
E1
2.0
Points
401
Unit
eV
Energy zero
AverageFermiLevel
Infinitesimal
1e-06
eV
Self Energy Calculator
RecursionSelfEnergy
Storage strategy
NoStorage
k-point sampling
Density
Populate k-points quickly with
Preset Densities
Density(Å)
Sampling
A
12
Å
20
B
12
Å
20
Sync
Advanced options

8. Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool to submit this job on a local machine or a remote cluster. Read these technical notes on *Parallelization of QuantumATK calculations* to learn how to best run NEGF simulations in parallel.

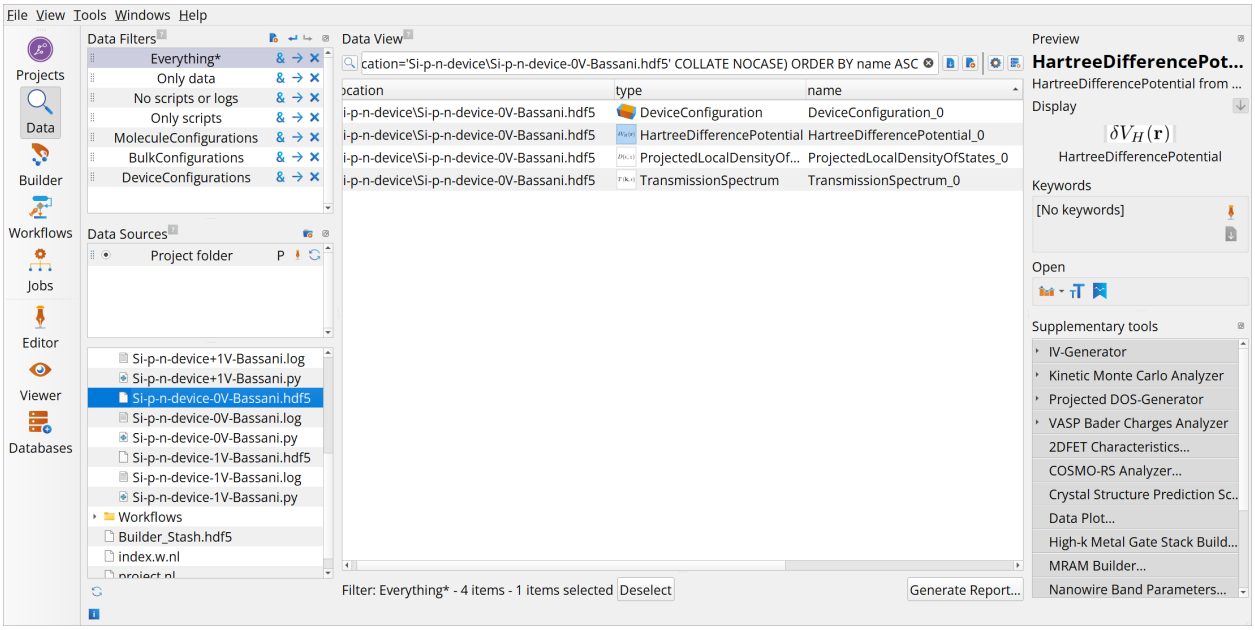
9. This also generates a Python script `Si-p-n-device-0V-Bassani.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool.

Analyzing Results at 0 V Bias

Once the calculation has finished, we are ready to analyze the results.

1. Go to the **Data View** tool.

2. In the *Data Sources*, select the file `Si-p-n-device-0V-Bassani.hdf5` to see the list of calculated objects, *HartreeDifferencePotential*, *ProjectedLocalDensityOfStates* and *TransmissionSpectrum* in *Data View*.



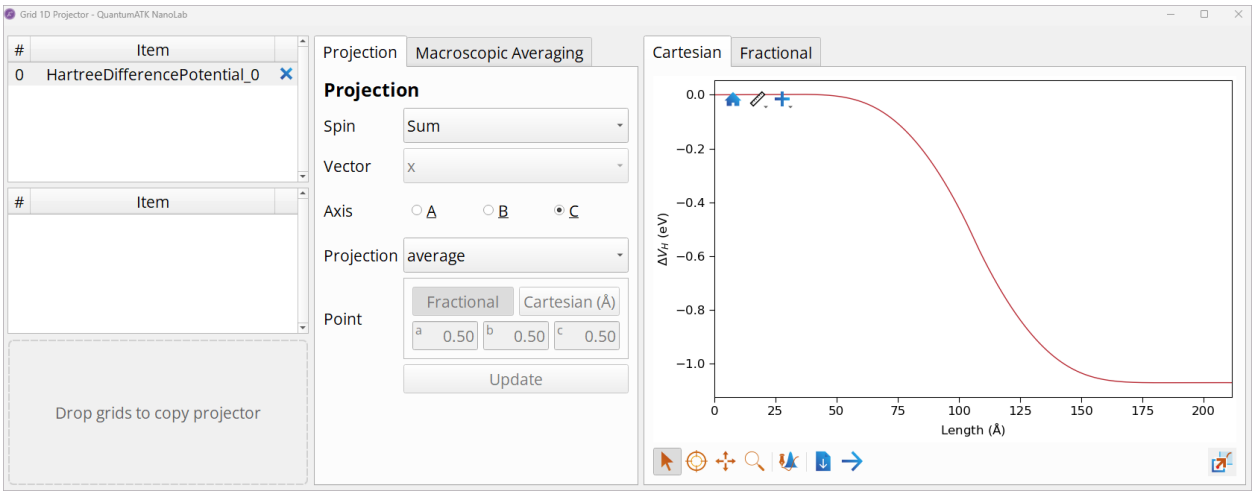
3. Double-click on the *HartreeDifferencePotential* to open it with the *Grid 1D Projector* and observe Hartree difference potential plotted along the devices central region, which is ~20 nm long.

Note: The Hartree difference potential in the simulated Si p-n junction is flat towards the electrodes, indicating that the device is long enough, i.e., the number of Si layers in the left and right ends of the central region provide sufficient screening (the results are converged wrt. the length of the device). See the NEGF manual *NEGF: Device Calculators* for more info.

---

4. Double-click on the *ProjectedLocalDensityOfStates* object to open it with the *Projected Local DOS Analyzer* and investigate the projected local DOS of the Si p-n junction along the transport direction of the device.

- Check the box *Band Edges* to visualize band edges.



•Right-click to *Add measurement* → *Measure (vertical)* or *Add measurement* → *Measure (horizontal)* to measure a band gap, E_g, along the devices central region, depletion-layer width, W_{dep}, and the built-in potential, ϕ_{bi}.

Note:

1.*Projected Local DOS* shows that far from the p-n interface, we have a p-type semiconductor on the left side (with valence band maximum, E_V, close to the Fermi level, $_L$) and the n-type semiconductor on the right side (with conduction band minimum, E_C, close to the Fermi level, $_R$).

2.Measured Si band gap on the left (p-type) and right (n-type) sides of the device is ~ 1.18 eV, which is in a good agreement with the band gap calculated for bulk Si in Tutorial *Band Structure, Projected Density of States and Effective Mass Calculations* with the HSE06 hybrid DFT functional, GW[1] and experimental results[2]. This confirms that the Bassani tight-binding model parametrized for Si model gives reliable results in this case.

3.At the p-n interface (middle of the device) there is a depletion layer with W_{dep} 16.5 nm, where E_C and E_V are not flat (band bending) due to different doping types, resulting in the built-in potential across the junction (potential barrier at the junction), ϕ_{bi} 1.05 eV, which opposes both the flow of holes and electrons across the junction.

5.Double-click on the *TransmissionSpectrum* object to open it with the *Transmission Analyzer* to analyze transmission spectrum $T()$ at 0 V bias.

Note:

1.The transmission $T()$ is zero in the band gap.

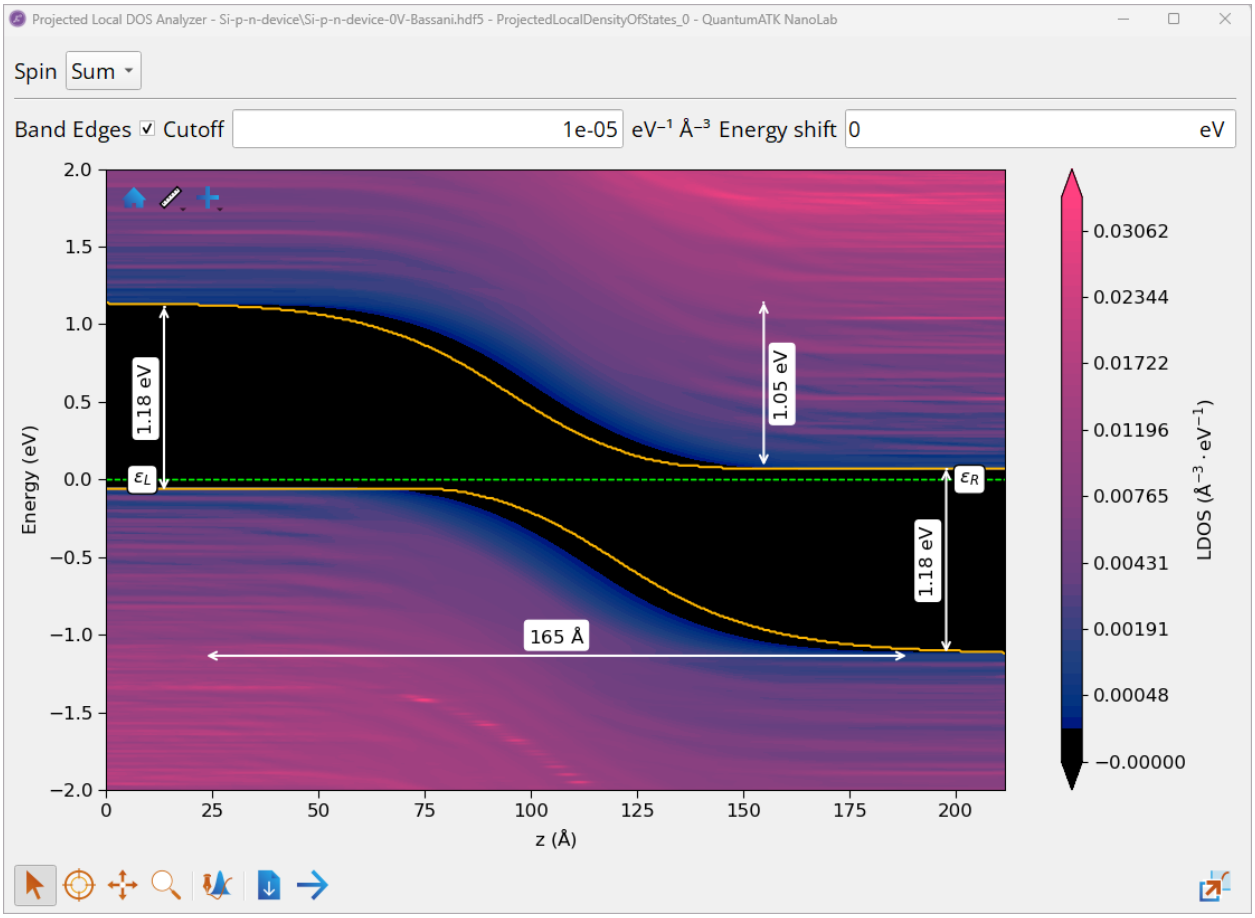
2.The electrical current I is calculated using the Landauer formula by intergating T across the energy range: $I = 2e/h \int_{-} dT()[f(-\mu_L)/(k_B T_L) - f(-\mu_R)/(k_B T_R)]$, where is electron energy and μ_L and μ_R are the left and right electrode chemical potentials, respectively. See the manual on *TransmissionSpectrum* to learn more.

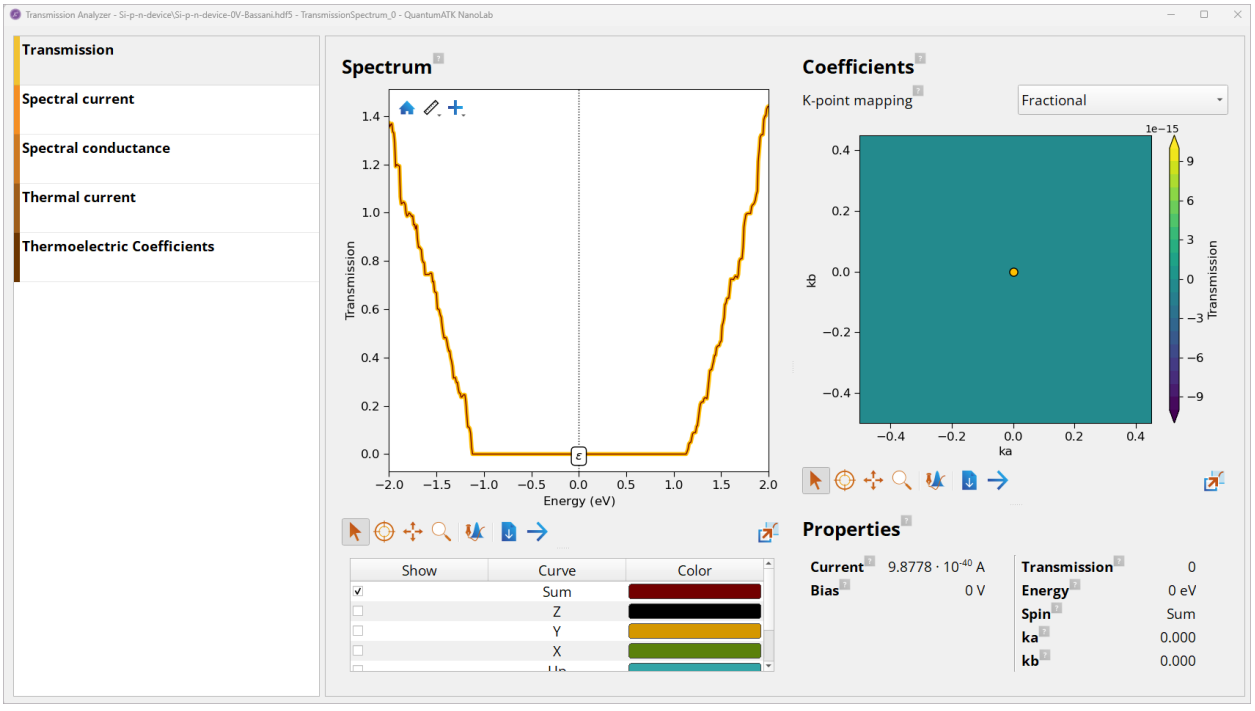
3.At 0V bias the $\mu_L = \mu_R = 0$, and so the current is 0.

---

[1] T.ăMarkussen, R.ăFaber, P.ăA. Khomyakov, B.ăVerstichel, and A.ăBlom. Band Alignment in GAA Nanosheet Structures from Density Dependent Hybrid Functional and Many-Body GW Methods. *SISPAD*, 2024.

[2] K.ăL. Milton, T.ăR. Durrant, T.ăCobos Freire, and A.ăL. Shluger. Difference in Structure and Electronic Properties of Oxygen Vacancies in -Quartz and -Cristobalite Phases of SiO2. *Materials*, 16(4):1382, 2023. doi:10.3390/ma16041382.


Projected Local DOS Analyzer - Si-p-n-device\Si-p-n-device-0V-Bassani.hdf5 - ProjectedLocalDensityOfStates_0 - QuantumATK NanoLab
Spin
Sum
Band Edges
Cutoff
1e-05
eV⁻¹ Å⁻³
Energy shift
0
eV
1.18 eV
1.05 eV
1.18 eV
165 Å
ε_L
ε_R
Energy (eV)
z (Å)
LDOS (Å⁻³ · eV⁻¹)
0.03062
0.02344
0.01722
0.01196
0.00765
0.00431
0.00191
0.00048
−0.00000


Transmission Analyzer - Si-p-n-device\Si-p-n-device-0V-Bassani.hdf5 - TransmissionSpectrum_0 - QuantumATK NanoLab
Transmission
Spectral current
Spectral conductance
Thermal current
Thermoelectric Coefficients
Spectrum
Transmission
Energy (eV)
Show
Curve
Color
Sum
Z
Y
X
Coefficients
K-point mapping
Fractional
kb
ka
Transmission
Properties
Current
9.8778 · 10⁻⁴⁰ A
Bias
0 V
Transmission
0
Energy
0 eV
Spin
Sum
ka
0.000
kb
0.000

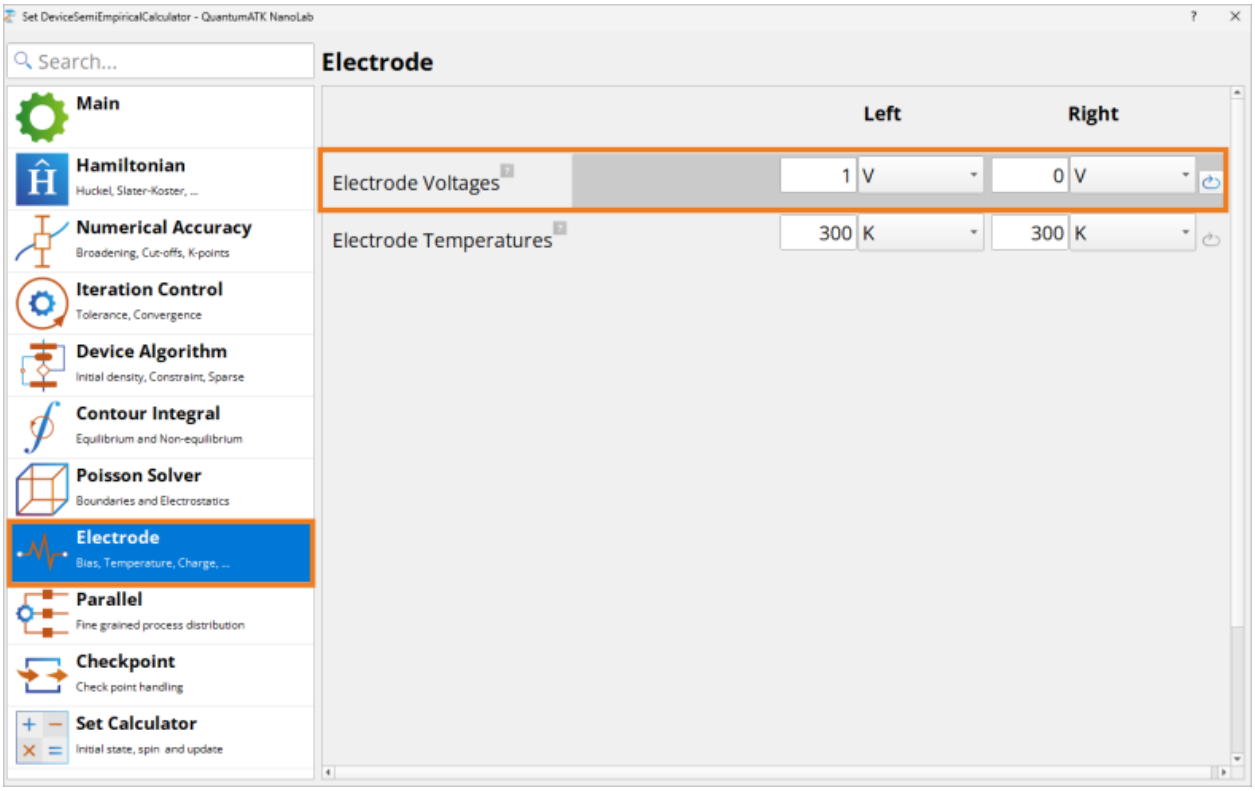
You have just finished with analyzing Hartree difference potential, potential barrier, left and right electrode chemical potentials from the projected local density of states, and transmission spectrum for the Si p-n junction at 0 V bias. You will next perform the finite bias simulations to see the impact of the forward (+1 V) and reverse (-1 V) bias on the potential barrier, left and right electrode chemical potentials and transmission spectrum, which can then be related to the calculated current in the device.

Setting up Electron Transport Simulations for the Si p-n Junction at Finite Bias

To create a workflow for electron transport NEGF calculations at finite bias in the **Workflow Builder**, follow the step-by-step protocol described below.

1. To set up finite bias (V_{bias} = -1 and 1 V) simulations:

- Go back to the **Workflow Builder**.
- In the *Workflow Stash*, right-click on the 0 V bias workflow *Si-p-n-device-0V-Bassani*, choose *Duplicate* and do this twice.
- Rename the two duplicates to *Si-p-n-device-1V-Bassani* for -1V and *Si-p-n-device+1V-Bassani* for +1V.
- Double-click on the *Set DeviceSemiEmpiricalCalculator* block.
- Under *Electrode*, change *Electrode Voltages* to -1V (left electrode) and 0 V (right electrode) to set up V_{bias} = -1 V.
- Under *Electrode*, change *Electrode Voltages* to 1 V (left electrode) and 0 V (right electrode) to set up V_{bias} = +1 V.
- Leave the rest of the settings unchanged.



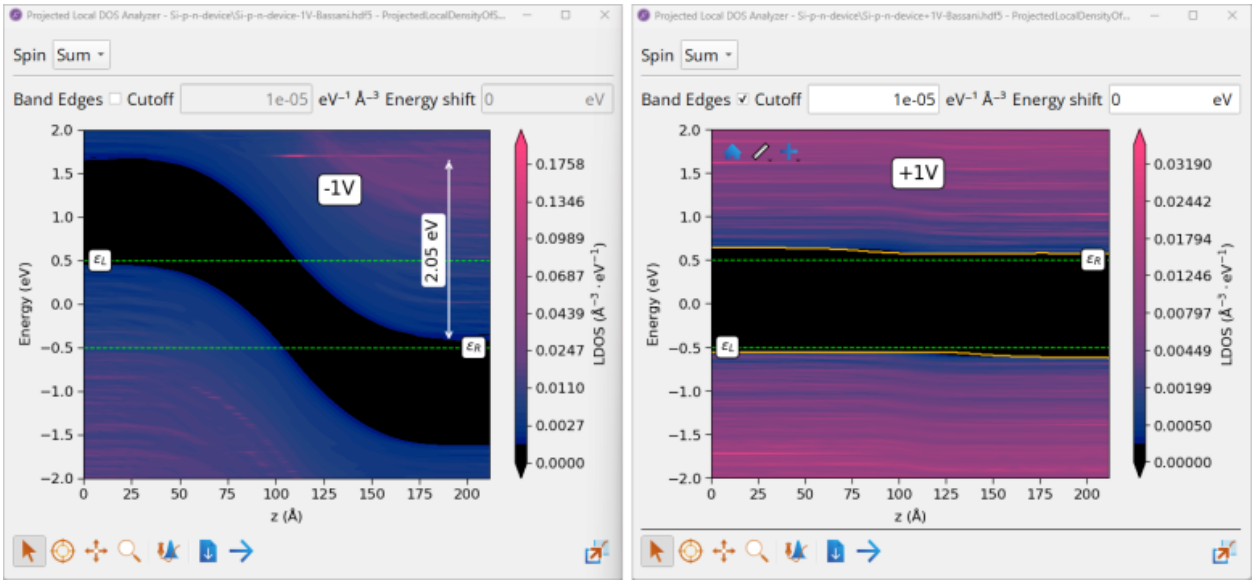
Tip: When running finite bias calculations, there is a possibility to restart from 0 V bias or lower bias calculations to improve and speed-up the SCF convergence. To do this, use *Load from file* block in the workflow to set up from which calculation to restart and check *Set with initial state* box in the *Set DeviceSemiEmpiricalCalculator* block → *Set Calculator* tab.

Analyzing Results at Finite Bias

Once the calculation has finished, we are ready to analyze the results.

1.To analyze projected local density of states at finite bias (V_{bias} = -1, +1 V):

- Go to the **Data View** tool.
- In the *Data Sources* window, select a relevant hdf5 file (Si-p-n-device-1V-Bassani.hdf5 for -1V and Si-p-n-device+1V-Bassani.hdf5 for +1V).
- In the *Data View* window, double-click on the *ProjectedLocalDensityOfStates* object to inspect the PLDOS plot for each bias value.
- Right-click to *Add measurement* → *Measure (vertical)* to measure the built-in potential, ϕ_{bi}.
- Right-click to *Add decoration* → *Label* to add labels.



Note:

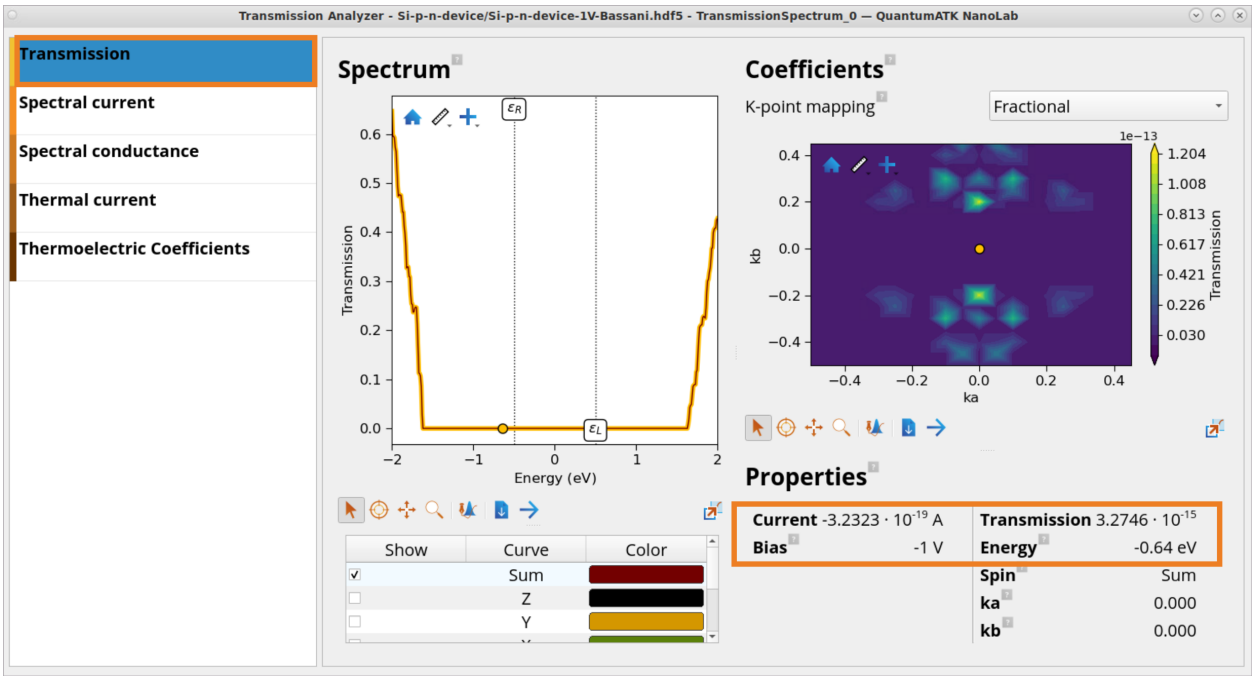
1.PLDOS at -1V (reverse bias) shows that $_L$ = +0.5 eV and $_R$ = -0.5 eV and the potential barrier is increased beyond ϕ_{bi} by 1eV (2.05 eV in total).

2.PLDOS at +1 V (forward bias) shows that $_L$ = -0.5 eV and $_R$ = +0.5 eV and the potential barrier is decreased beyond ϕ_{bi} by 1eV (to negligible ~ 0.05 eV in total).

2.To analyze transmission spectrum at finite bias (V_{bias} = -1, +1 V):

•In the *Data Sources*, select a relevant hdf5 file (Si-p-n-device-1V-Bassani.hdf5 for -1V and Si-p-n-device+1V-Bassani.hdf5 for +1V) and in *Data View* double-click on the *TransmissionSpectrum* object.

•Click on the *Transmission* panel to inspect the transmission spectrum for each bias value. *Spectrum* window shows the transmission plotted as a function of energy, whereas *Properties* panel reports the total current and a transmission value at a chosen energy in the *Spectrum* plot.



Note:

1.According to the Landauer formula shown above, electrical current I is calculated by intergating T across the energy range, and so the size of transmission has an impact on the observed current.

2.Under reverse -1 V bias, due to the increased potential barrier observed in the PLDOS plot, the transmission T around the bias window is close to 0 (e.g., 3.3 10^{-15} at -0.64 eV) and so is the current, I = -3.23 10^{-19} A.

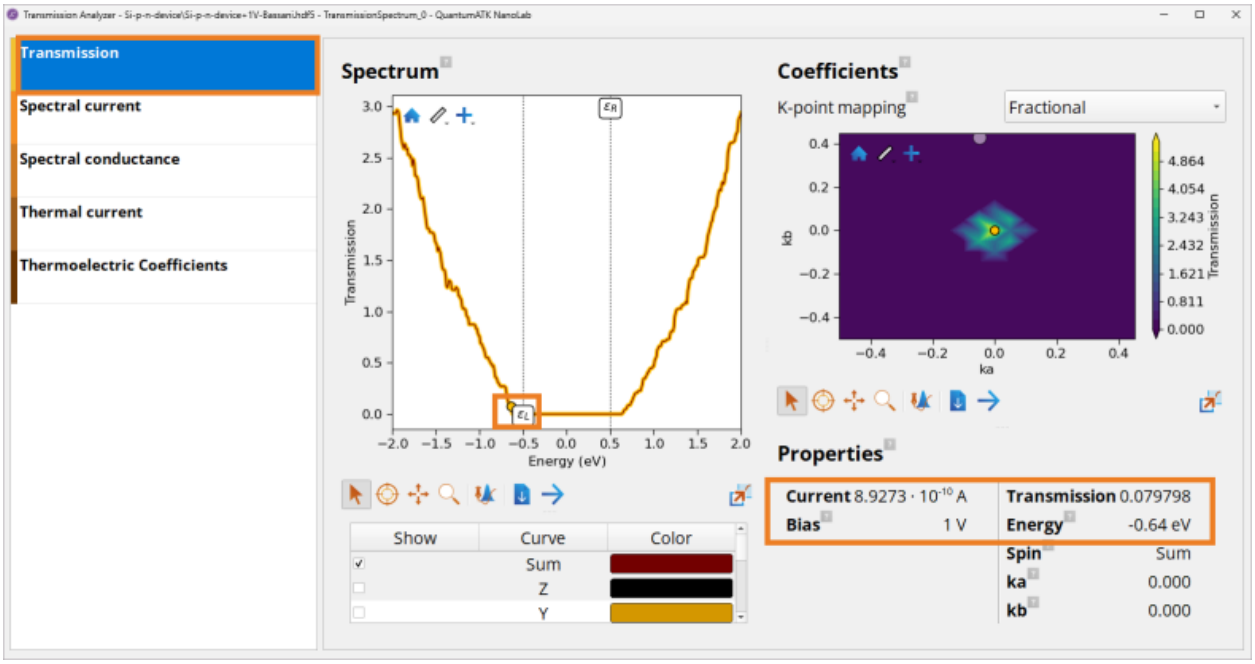
3.Under forward +1 V bias, due to the decreased potential barrier observed in the PLDOS plot, the transmission T around the bias window is much higher (e.g., 0.08 at -0.64 eV and rapidly increases when going to larger energy values) and so the current is much larger: I = 8.9 10^{-10} A.

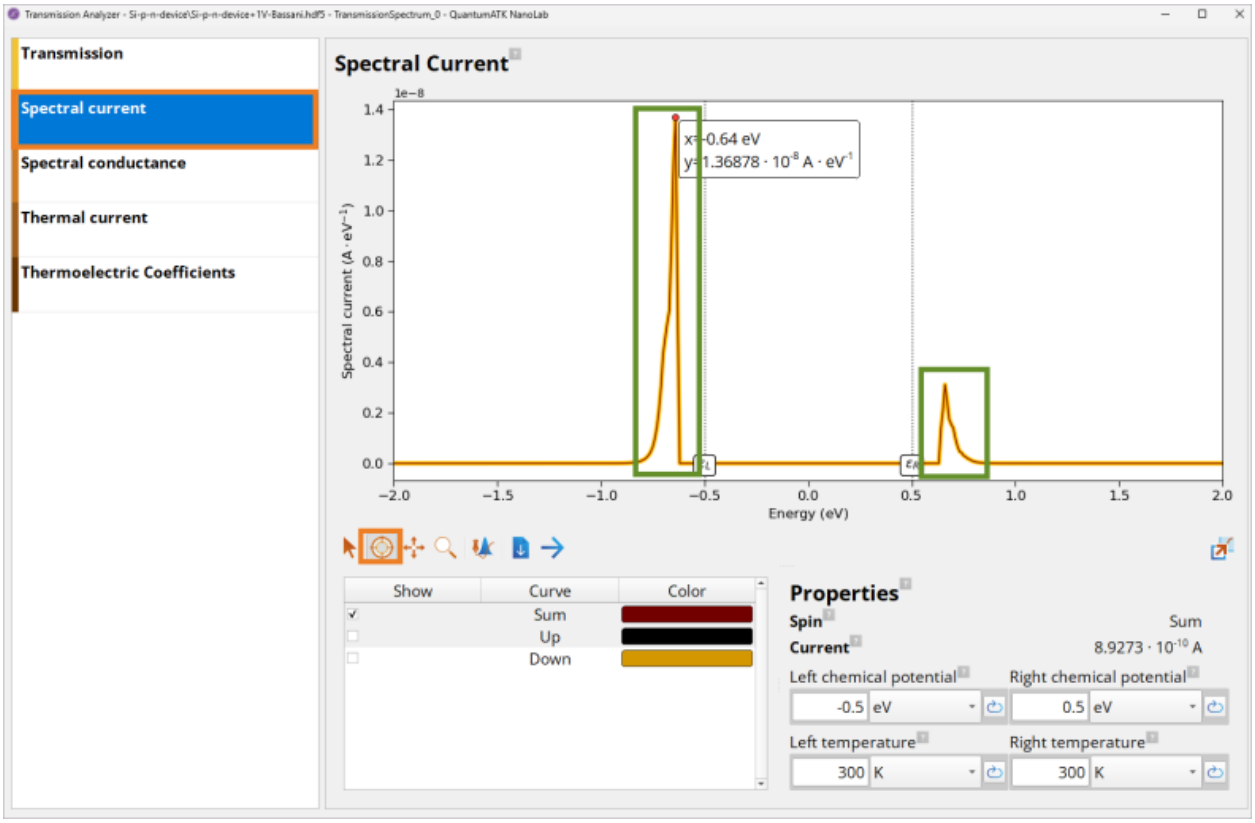
4.This shows as expected, that the Si p-n device acts as a diode, which allows current to flow easily in one direction, but severely restricts current from flowing in the opposite direction.

3.Click on the *Spectral current* panel to investigate which energy states contribute most to the current, I, at forward V_{bias} = +1 V bias, as the current depends not only on the size of transmission, but also available states that can transmit the current.

Note:

1.Energy states at ~ -0.64 eV and 0.66 eV contribute most to the current, I


Transmission
Spectral current
Spectral conductance
Thermal current
Thermoelectric Coefficients
Spectrum
Transmission
Energy (eV)
Show
Curve
Color
Sum
Z
Y
Coefficients
K-point mapping
Fractional
kb
ka
Transmission
Properties
Current 8.9273 · 10^-10 A
Bias 1 V
Transmission 0.079798
Energy -0.64 eV
Spin Sum
ka 0.000
kb 0.000


Transmission
Spectral current
Spectral conductance
Thermal current
Thermoelectric Coefficients
Spectral Current
Spectral current (A · eV^-1)
Energy (eV)
x=-0.64 eV
y=1.36878 · 10^-8 A · eV^-1
Show
Curve
Color
Sum
Up
Down
Properties
Spin Sum
Current 8.9273 · 10^-10 A
Left chemical potential
Right chemical potential
-0.5 eV
0.5 eV
Left temperature
Right temperature
300 K
300 K

2. One can obtain *TransmissionEigenstate* and *TransmissionEigenvalues* by adding *TransmissionEigenstate* and *TransmissionEigevalues* blocks to the workflow.

3. It is also possible to vary the temperature of the Fermi-Dirac distribution in the left and right electrodes to see the temperature impact on the current, I, in this case of an ideal structure. Note that this does not include electron-phonon coupling, which is beyond the scope of this tutorial.

Tip: Though there were no convergence issues in electron transport NEGF simulations of the Si p-n junction, sometimes one can encounter challenges in converging NEGF calculations to obtain the electronic ground state. In these cases, read the *NEGF Convergence Guide* which provides a list of action points for troubleshooting poorly converging NEGF calculations.

Summary and Outlook

In this tutorial, you have learnt how to set up and analyze projected local density of states, transmission spectrum and spectral current for a few bias points to demonstrate the principles of NEGF calculations of electron transport in devices. For more realistic studies, such as full I_dV_d and I_dV_g curves, we recommend to perform *IVCharacteristics* simulations. This is a combined framework for running multiple source-drain/gate voltage calculations and collecting and analyzing the results with smart SCF restart functionality from lower bias simulations for faster and better SCF convergence.

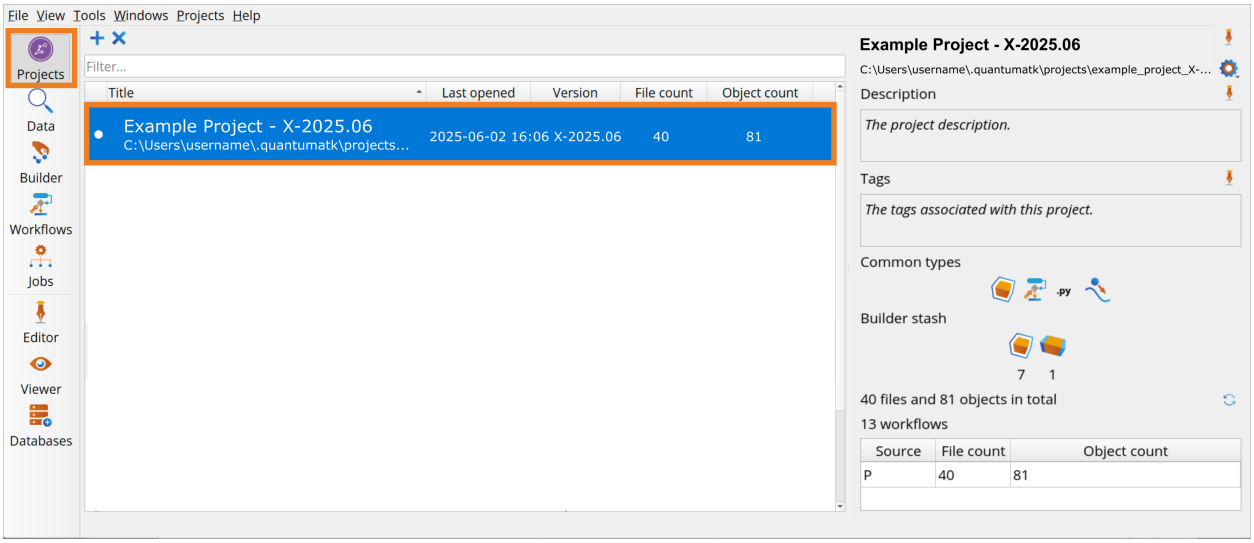
References

Workflow, input and output files from the tutorials above can be found in the **Example Project** of the QuantumATK X-2025.06 release or can be downloaded from the tutorial pages.

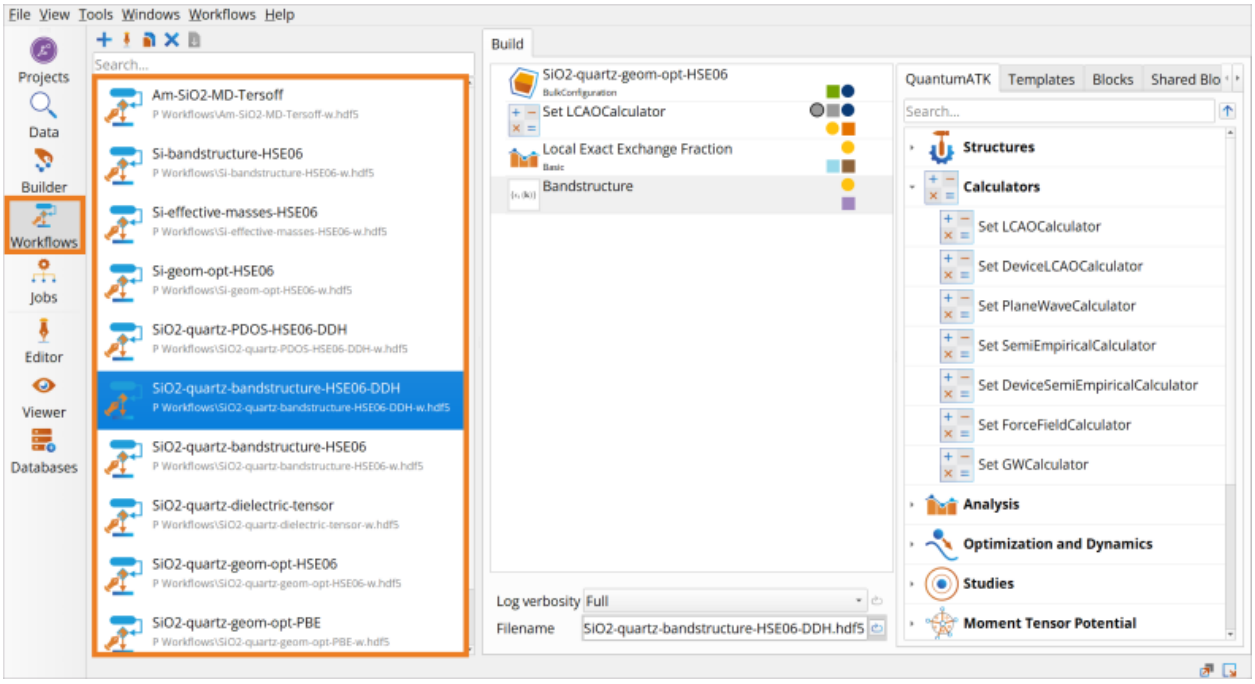
1. Open **QuantumATK NanoLab**.

2. Click on **Projects**.

3. Double-click on the *Example Project - X-2025.06*.



4.Click on the **Workflow Builder** to inspect/rerun all the simulation workflows used in the above introductory tutorials.



5.Click on the **Data View** -> *Data Sources* to find dedicated folders for each tutorial, containing input (.py) and output files (.hdf5).

1.3 Feature List

1.4 Referencing QuantumATK

QuantumATK is an *integrated platform of electronic and atomic-scale modeling tools* written in Python and C++. It gives you access to a powerful set of simulation engines for R&D at the atomic scale:

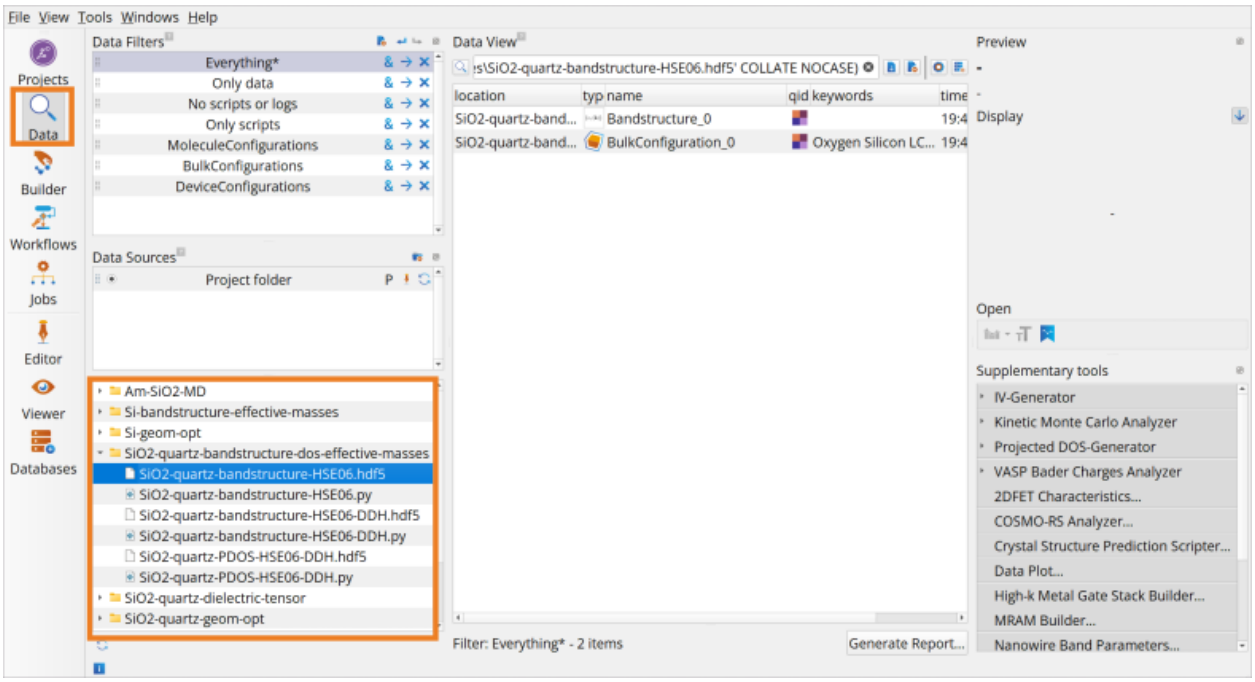
•Density Functional Theory (DFT)

•Many-Body Perturbation Theory Method GW

•Semi-Empirical Tight Binding

•Classical Conventional and Machine-Learned Force Fields

•Non-Equilibrium Greens Functions (NEGF) Method for device simulations



A QuantumATK installation comes with two executables – **atkpython** and **quantumatk**. The first one is an enhanced Python interpreter that includes the QuantumATK simulation engines. It is meant for running QuantumATK simulations via Python scripts directly from a terminal. The second executable launches **QuantumATK NanoLab** which provides an intuitive graphical user interface (GUI) on top of the QuantumATK engine.

The recommended way of using QuantumATK is via **NanoLab**. It helps you focus on your modeling task rather than technical details, by assisting you with the preparation steps. **NanoLab** also offers important tools for data analysis.

CHAPTER

TWO

NANOLAB GUIDES

Here you will find an overview of the user interface in *QuantumATK NanoLab*. When a new version is released, we recommend checking out this section to get acquainted with the latest updates and newly added features.

The *QuantumATK NanoLab* section will get you familiar with the user interface layout and tools such as *Data*, *Builder*, *Jobs*, and *Plots*. You will learn how to organize, search, import, and export files and structures using the *Data* tool. You will get an overview of the *Builder* tool and its plugins with which you can create, load, or edit various atomic-scale structures. The *Jobs* tool overview will show you how to execute scripts on a local or remote machine and get calculation results, which you can then plot with the *Plots* tool.

In the *QuantumATK NanoLab* section, you will also find guides on the newly added features and updates of the current tools.

If you have already installed *QuantumATK NanoLab* and you are familiar with its most recent features, we recommend you to continue to the *Tutorials* category, where you will learn how to use the program and its vast number of features in a meaningful way through many concrete examples starting from a beginner to an expert level.

Guides to QuantumATK NanoLab

2.1 Overview of NanoLab

Downloads & Links

Installation Guide Builder Database Workflows Jobs Data `SiC.hdf5` `analysis_SiC.hdf5`

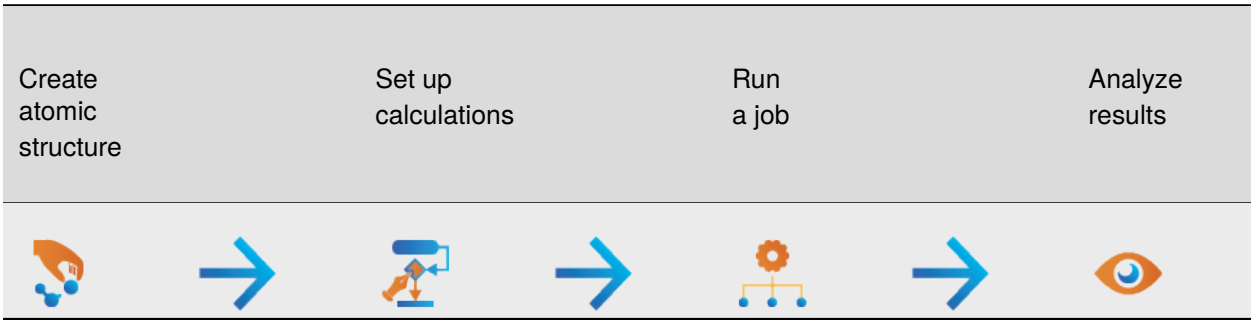
Note: We will assume that a working copy of QuantumATK has already been installed, as described in the *Installation Guide*.

Table of Contents

- *Typical workflow*
- *Create atomistic structure*
- *Set up calculation*
- *Run a job*
- *Analyze results*

2.1.1 Typical workflow

The recommended way of using QuantumATK is through NanoLab. It helps you focus on solving your problem, by assisting you with the preparation steps. A typical workflow in NanoLab consists of the following steps:



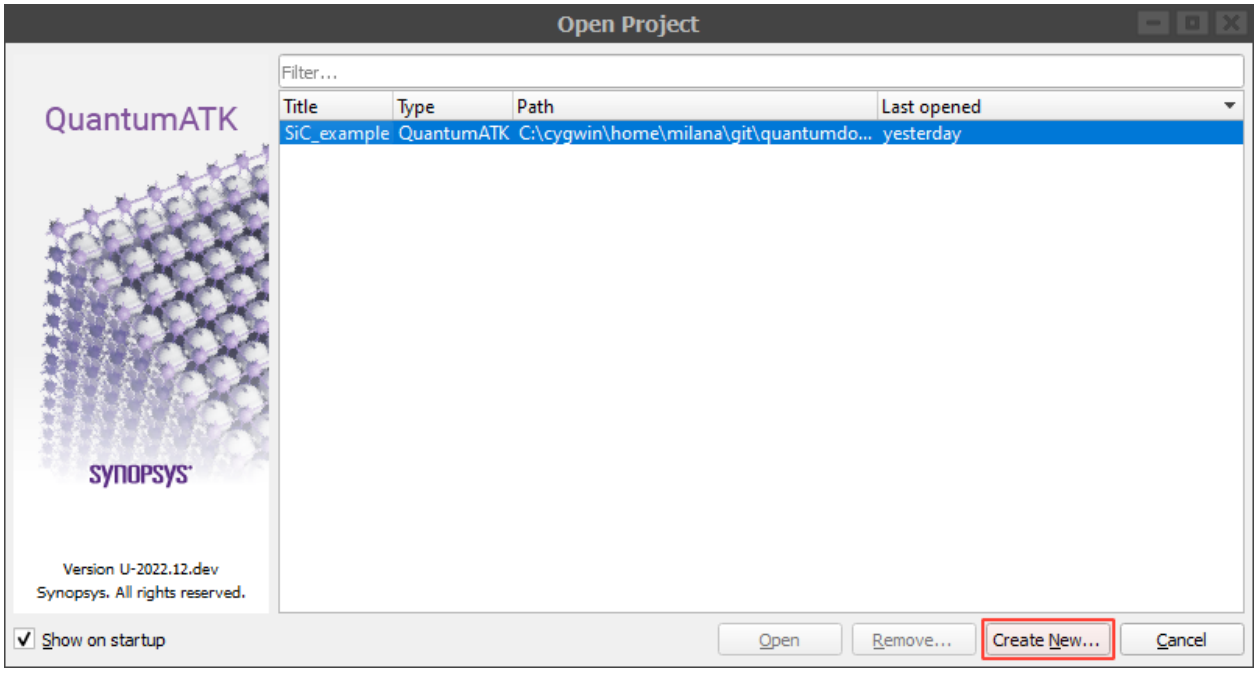
We will demonstrate each workflow step through a very simple example in which we will be calculating the **bandstructure of a SiC crystal**. In the next tutorials, we will show how to use QuantumATK to calculate many other properties and teach you how to choose the simulation parameters to best suit your goals.

2.1.2 Create atomistic structure

Lets start by opening NanoLab. There are two different ways to do it:

1. by clicking the QuantumATK NanoLab icon on the Desktop;
2. by executing the **quantumatk** command in a terminal.

The *Open Project* window will appear where you can create *new projects* and open existing ones. We will start by creating a new *QuantumATK project* with the title *SiC_example*.

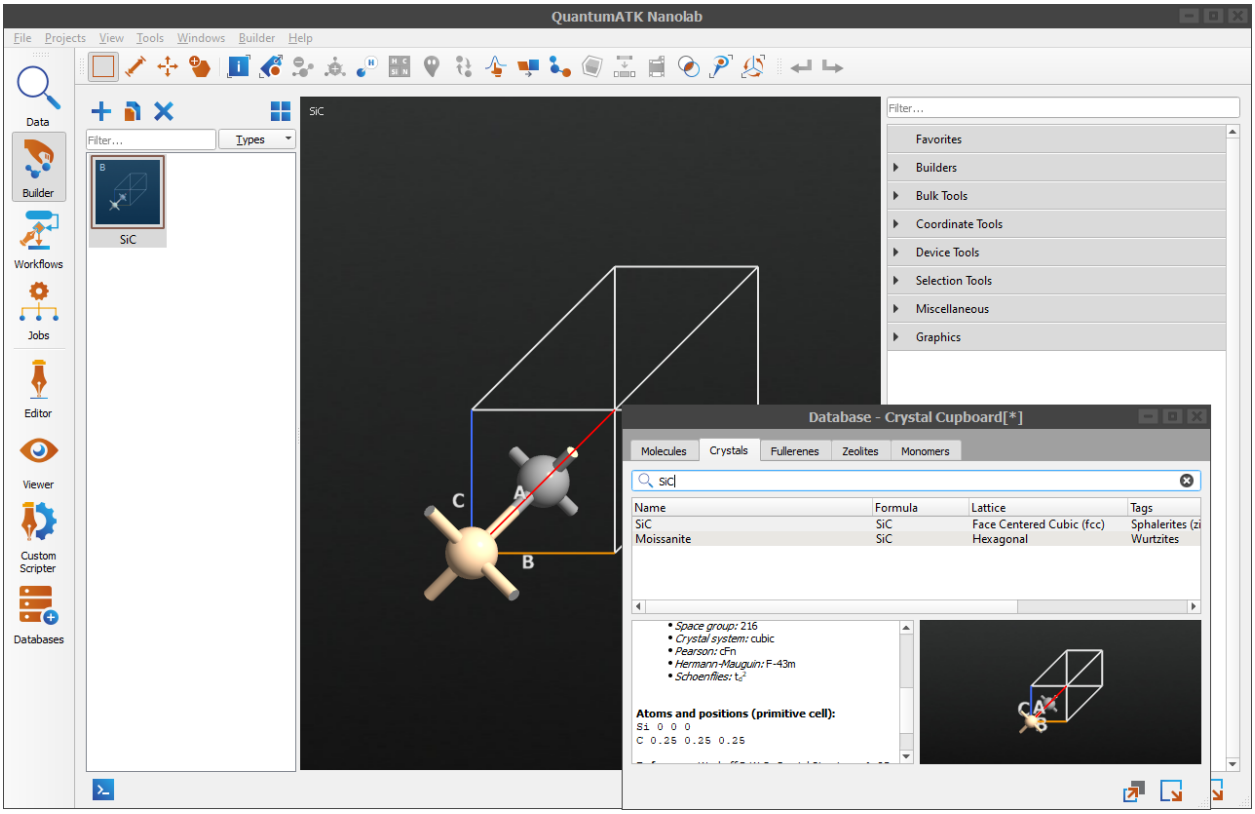


Once you open a *new project*, you will see the main window of NanoLab showing the contents of the new *SiC_example* project folder. The project is currently empty.

We will start by loading a SiC `BulkConfiguration` using the *Builder* tool. The *Builder* tool is used to load, build, edit, visualize and inspect atomic configurations. It contains a plethora of built-in plugins for manipulating atomic configurations, as well as databases with built-in atomic configurations.

Lets load a SiC crystal in the *Builder* tool. To open the *Builder* tool, click the icon in the vertical toolbar on the left. Once you are in the *Builder* tool, lets load a SiC crystal from the *Database*.

To open the *Database* click the and select *From Database*. Make sure you are at the *Crystals* tab, and type *SiC* in the text search. Double click the *SiC* configuration from the list to load it to the *Builder*.



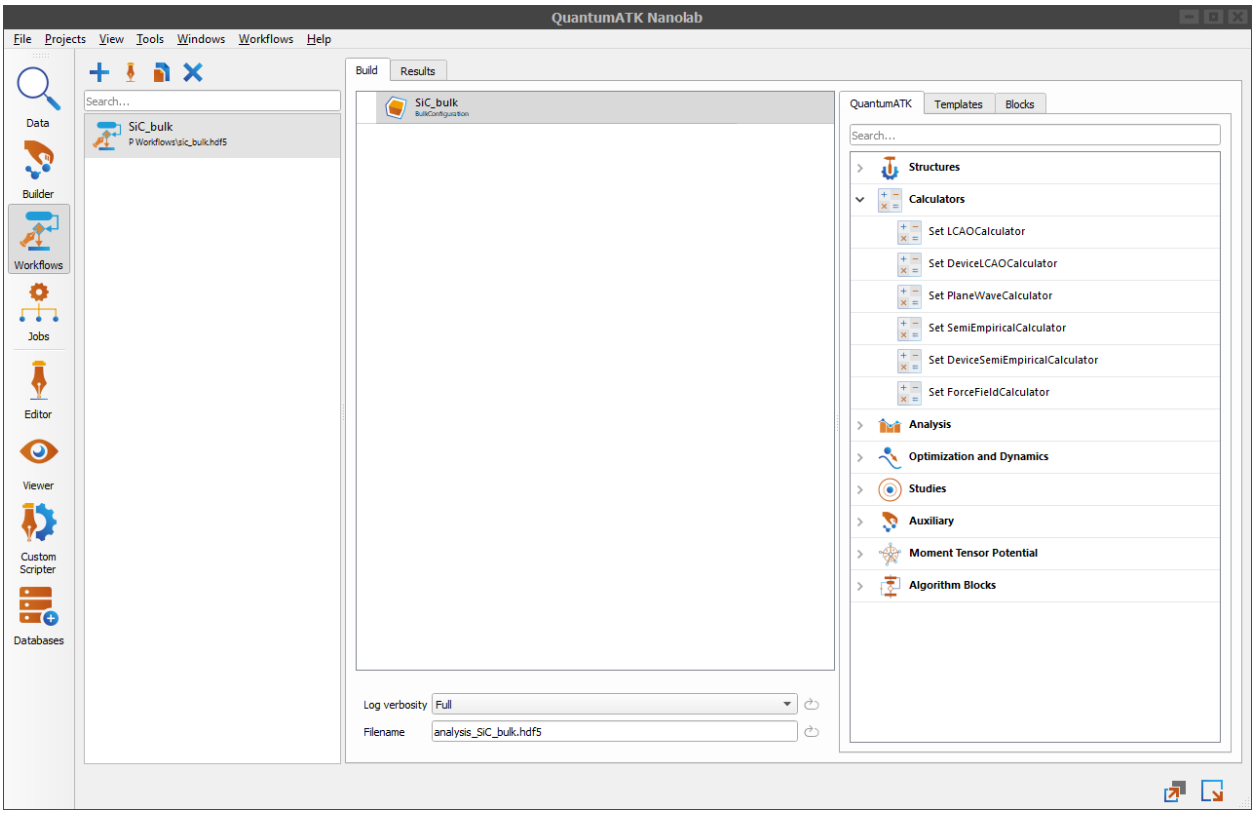
The SiC crystal should now be visible in the *Builder*. In the window on the right, you will see a small icon corresponding the SiC crystal. This window is also called *Builder stash*. It contains all the atomic configurations that you have created or loaded in the *Builder*.

In the central window of the *Builder* you will see a 3D graphical model of the SiC crystal. The central window is also referred to as the *3D Viewer*. From here you can click, rotate, zoom or pan the atomic configuration selected in the stash window.

2.1.3 Set up calculation

Lets create a bandstructure calculation for the SiC crystal loaded in the *Builder* tool. We will start by sending the SiC crystal from the *Builder* to the *Workflows* tool.

To send the SiC crystal from the *Builder* to the *Workflows* tool, click the **Send to** button and choose **Workflows** from the drop down menu. The *Builder* window is automatically replaced by the *Workflows* window.



The *Workflows* tool is used to assemble calculation steps, which can be run through the *Jobs* tool. Each calculation step can be one QuantumATK object. A catalog of built-in blocks, or calculation steps, are available under the *QuantumATK* tab in the right window.

We can see that the *Workflows* tool already contains the SiC crystal. This will be our first calculation step. Lets complete the calculation steps in the current workflow by adding a calculator and an analysis object.

In the *QuantumATK* tab on the right find the *Set LCAOCalculator* block and a *Bandstructure* analysis block. Drag the blocks from the list of QuantumATK blocks to the central window.

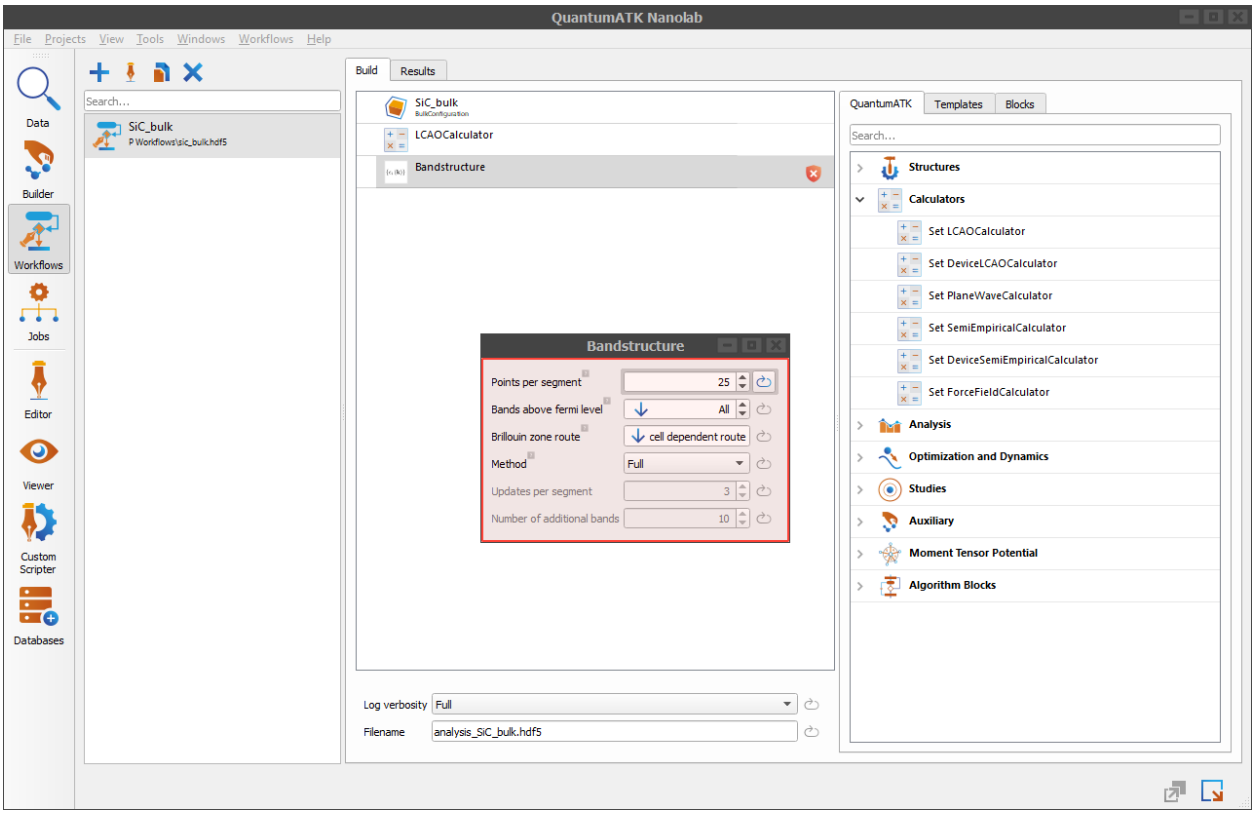
We have now created a workflow tree with one configuration, a calculator, and an analysis block. The *Set LCAOCalculator* block will get the SiC crystal as an input, which is a `BulkConfiguration` object. The output of the block will be the `BulkConfiguration` object with an attached `LCAOCalculator` object. The *Bandstructure* block will get the `BulkConfiguration` with the attached `LCAOCalculator` as an input, and will output a `Bandstructure` object. After we run the workflow, the `Bandstructure` object will be the final result.

The created workflow was automatically saved as the `SiC.hdf5` file in the `Workflows` folder. The file has extension `.hdf5` and is available in the *stash* of workflows on the right, as well as in the *Data*.

Note: **HDF5** is a data model, library, and file format for storing and managing data and is the format used for files storing QuantumATK objects.

The calculator computes the electronic ground state of the SiC crystal with DFT. It contains a wide range of settings, all of which have sensible default values. However, you will most often need to change at least a few of them in order to make the calculation match your scientific goals. Double-click the dropped LCAOCalculator block when you wish to edit it.

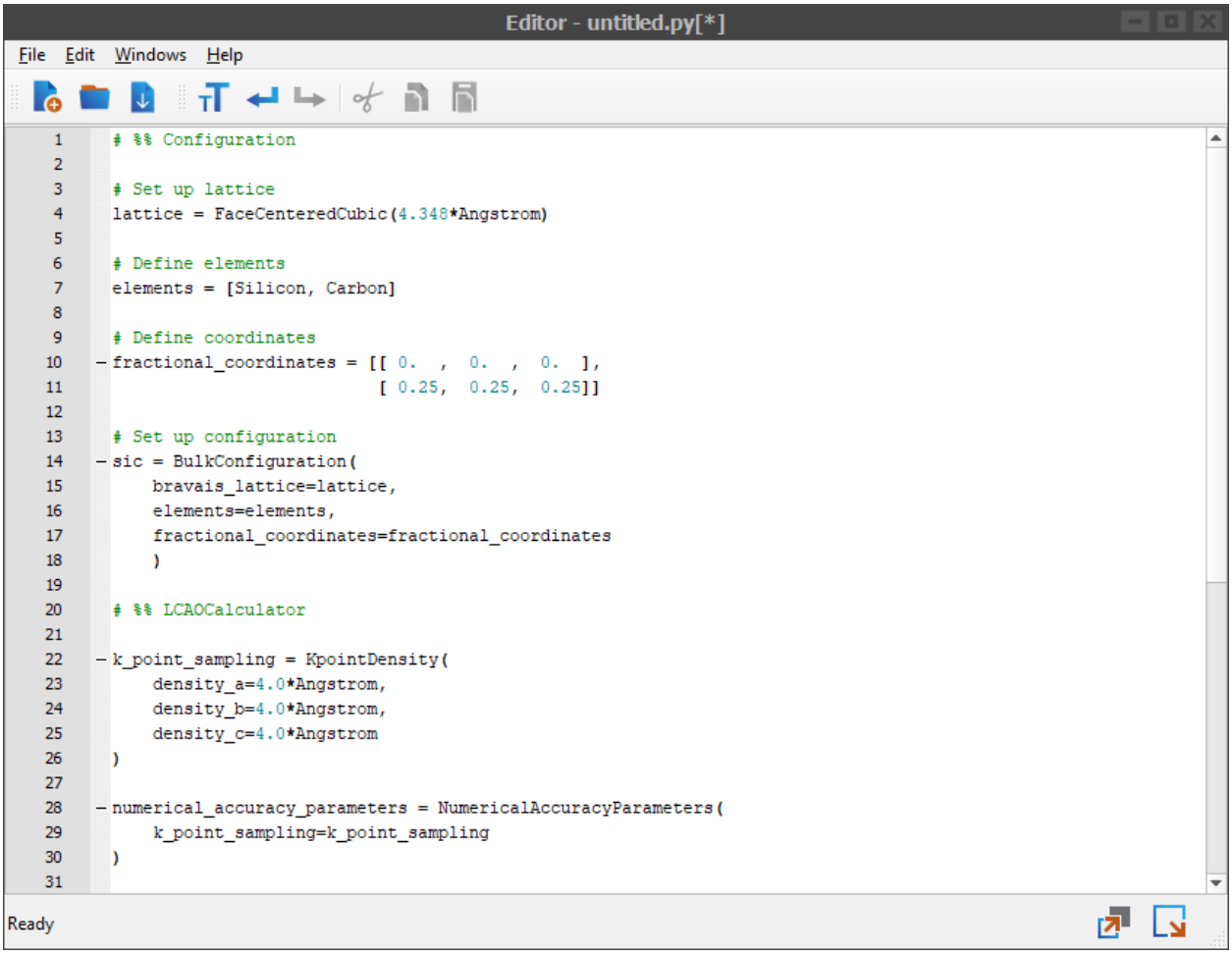
The Bandstructure block performs electronic ground state analysis of the SiC crystal. Double click the Bandstructure block to open its editor and to increase *Points per sampling* to *25*.



Note: When changing the value of a field in the Bandstructure block editor, make sure to confirm the change by hitting *Enter* before closing the dialog. Otherwise, the change will not get saved.

At this step we can generate a Python script based on this workflow by clicking and sending it to the **Editor** tool.

We are now prepared to run the calculation on our local machine or a supercomputing cluster.

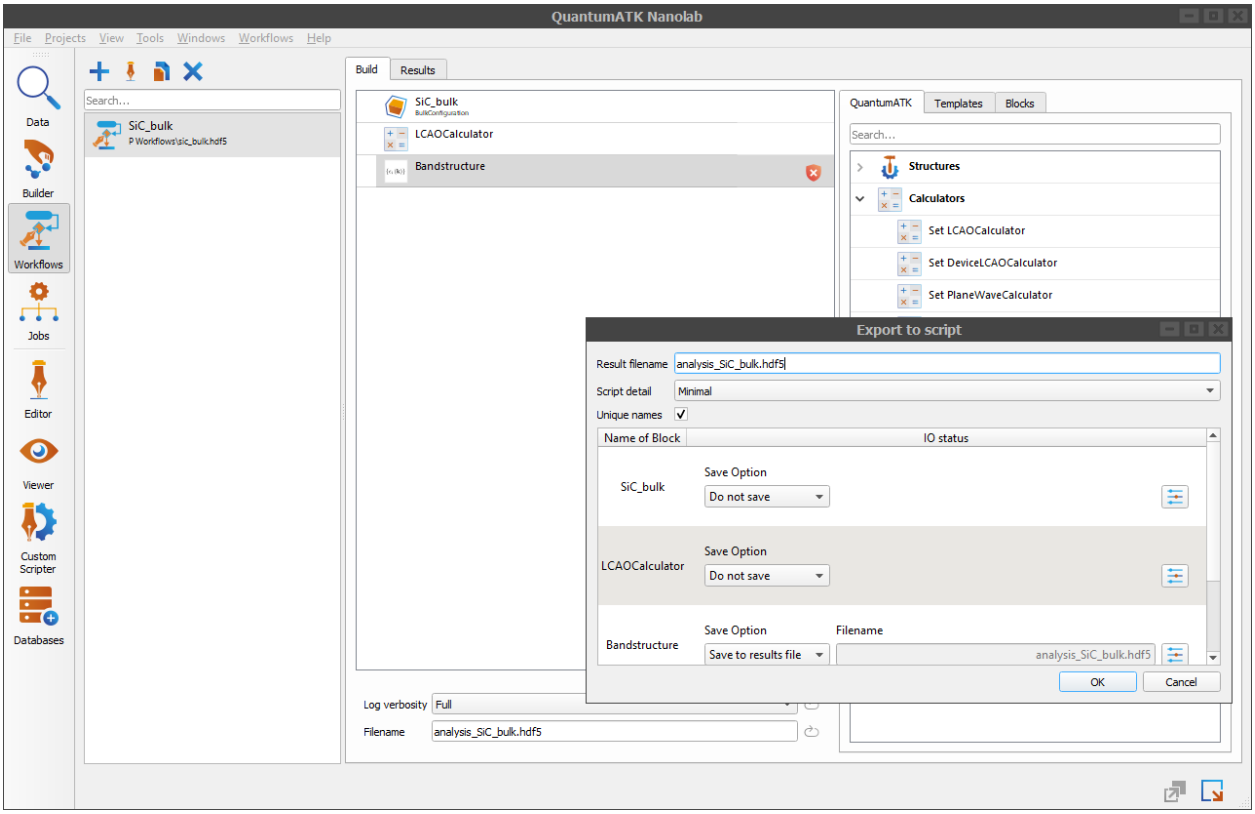

Editor - untitled.py[*]
File Edit Windows Help
# %% Configuration
# Set up lattice
lattice = FaceCenteredCubic(4.348*Angstrom)
# Define elements
elements = [Silicon, Carbon]
# Define coordinates
fractional_coordinates = [[0. , 0. , 0.],
[0.25, 0.25, 0.25]]
# Set up configuration
sic = BulkConfiguration(
bravais_lattice=lattice,
elements=elements,
fractional_coordinates=fractional_coordinates
)
# %% LCAOCalculator
k_point_sampling = KpointDensity(
density_a=4.0*Angstrom,
density_b=4.0*Angstrom,
density_c=4.0*Angstrom
)
numerical_accuracy_parameters = NumericalAccuracyParameters(
k_point_sampling=k_point_sampling
)
Ready

2.1.4 Run a job

There are different ways to run the created workflow. For example, we could export it to a script and run it from a terminal:

```
atkpython analysis_SiC_bulk.py > analysis_SiC_bulk.log
```

A very convenient alternative is to use the *Jobs* tool to run the script directly from inside NanoLab:



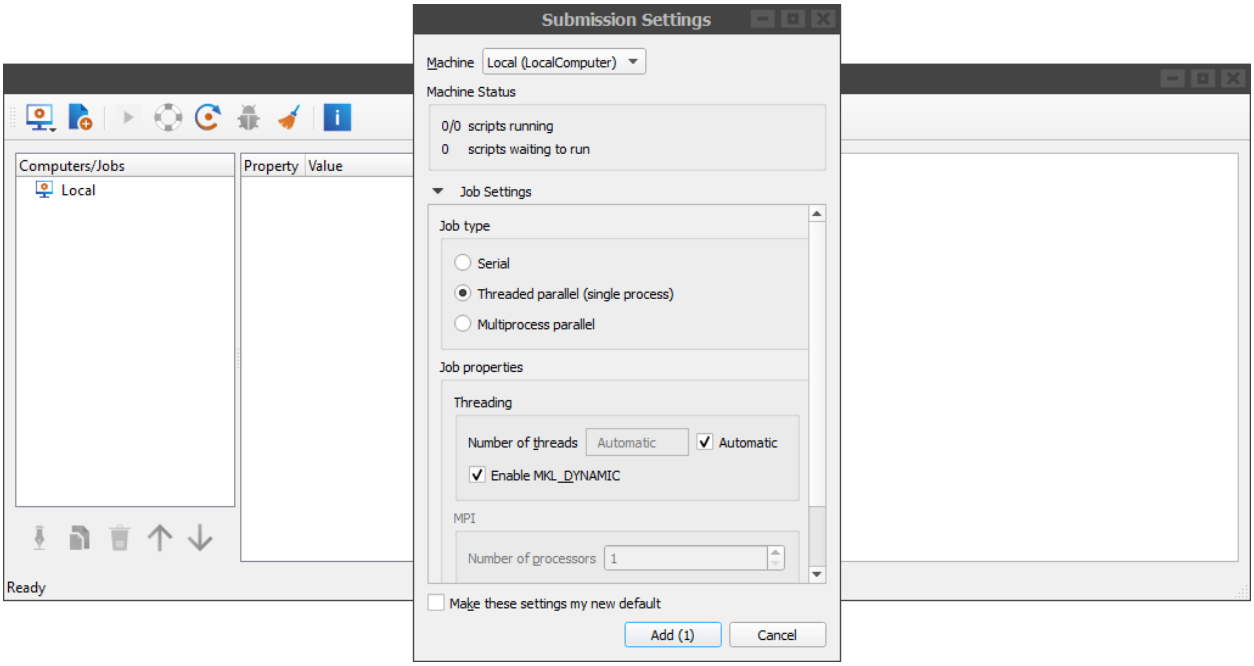
•Click the in the lower-right corner of the *Workflows*, and select *Jobs as script* from the menu.

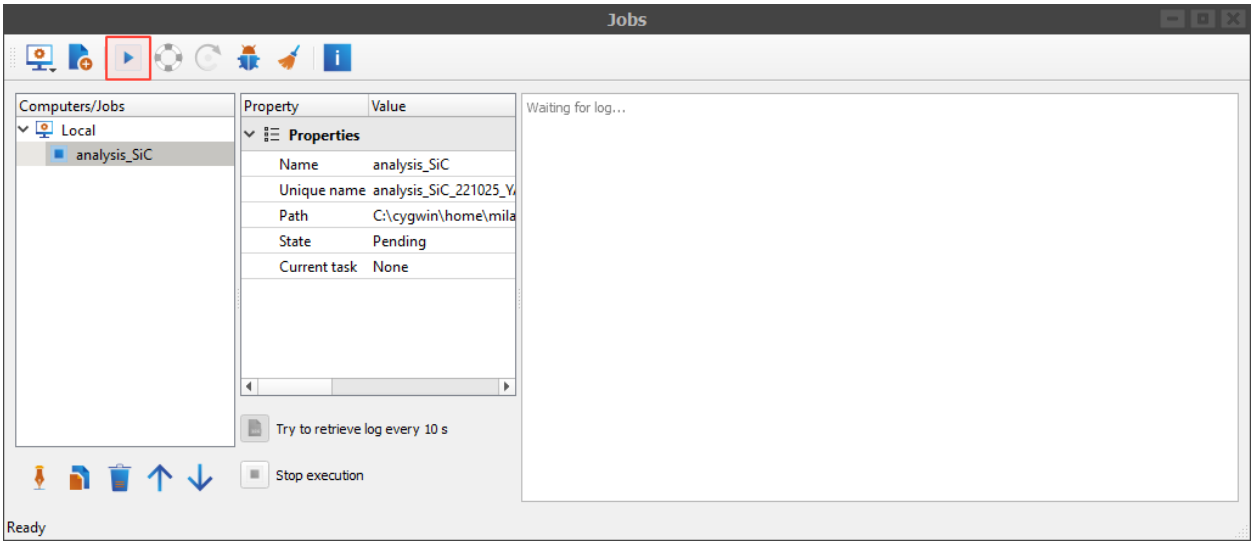
•The *Export to script* dialog appears, prompting you to export the workflow to a Python script and make sure it is saved. After saving the script, the *Jobs* window will open followed by the *Submission Settings* dialog.

•Choose a *Local* machine type. This is your current machine. This option is automatically created after the installation of NanoLab. When you click *OK*, the QuantumATK Python script will be added to the job queue of the chosen machine.

•Select your script and click the button to start executing the calculation.

Tip: See the *Jobs* tutorial for more instructions on using the Jobs for multi-CPU execution of QuantumATK Python scripts on your local machine and on remote supercomputers.


Submission Settings
Machine
Local (LocalComputer)
Machine Status
0/0 scripts running
0 scripts waiting to run
Job Settings
Job type
Serial
Threaded parallel (single process)
Multiprocess parallel
Job properties
Threading
Number of threads
Automatic
Automatic
Enable MKL_DYNAMIC
MPI
Number of processors
1
Make these settings my new default
Add (1)
Cancel
Computers/Jobs
Local
Property
Value
Ready


Jobs
Computers/Jobs
Local
analysis_SiC
Property
Value
Properties
Name
analysis_SiC
Unique name
analysis_SiC_221025_Y
Path
C:\cygwin\home\mila
State
Pending
Current task
None
Waiting for log...
Try to retrieve log every 10 s
Stop execution
Ready

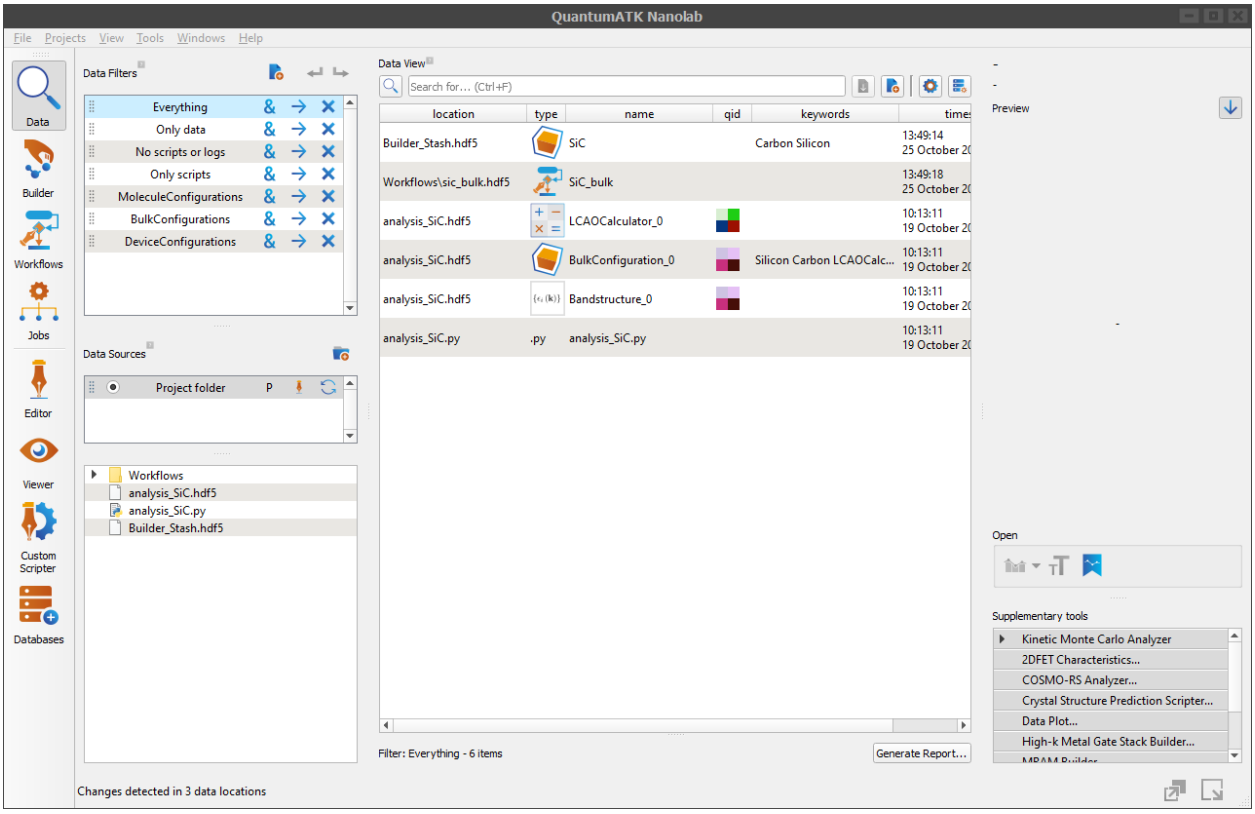
2.1.5 Analyze results

The calculation should last for about 20s. After it has finished, a new `.hdf5` result file will appear in the *Data* tool. New `.hdf5` files will be automatically scanned and their contents displayed in the central window of the *Data*.

You should be able to search and see the contents of the result file, `analysis_SiC.hdf5`. The content should include *Bandstructure_0*, *BulkConfiguration_0* and *LCAOCalculator_0*, which correspond to the calculated *Bandstructure* object, the SiC crystal and the same SiC crystal with the attached *LCAOCalculator*.

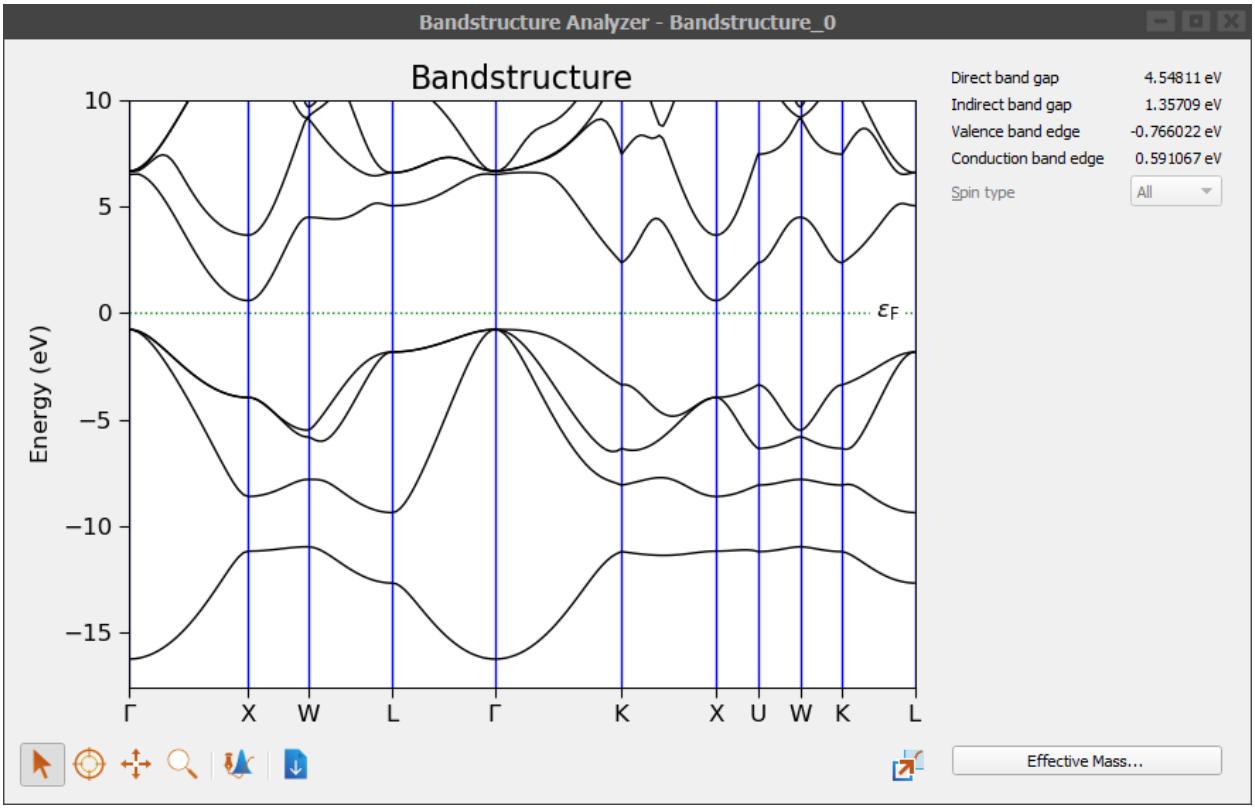
Other files you will see include the `Builder_Stash.hdf5`, the `sic_bulk` workflow and the Python script `analysis_SiC.py`.

The `Builder_Stash.hdf5` contains atomic configurations from the *Builder* stash. Currently there is only one *BulkConfiguration*. The workflow `sic_bulk` is the workflow which we have just created. The `analysis_SiC.py` is the `sic_bulk` workflow exported as a Python script.



To analyze the bandstructure results, search a bandstructure item in the *Data* and double click it to launch the *Bandstructure analyzer*. Both the direct and indirect band gaps are given by the analyzer.

Looking at the band gap results you can clearly see that they are inaccurate. This is because we were using default parameters in the *LCAOCalculator* and the *Bandstructure* blocks. To calculate the band gaps of SiC crystal the right way, follow the tutorial *Calculate the band structure of a crystal*.



2.1.6 Whats next

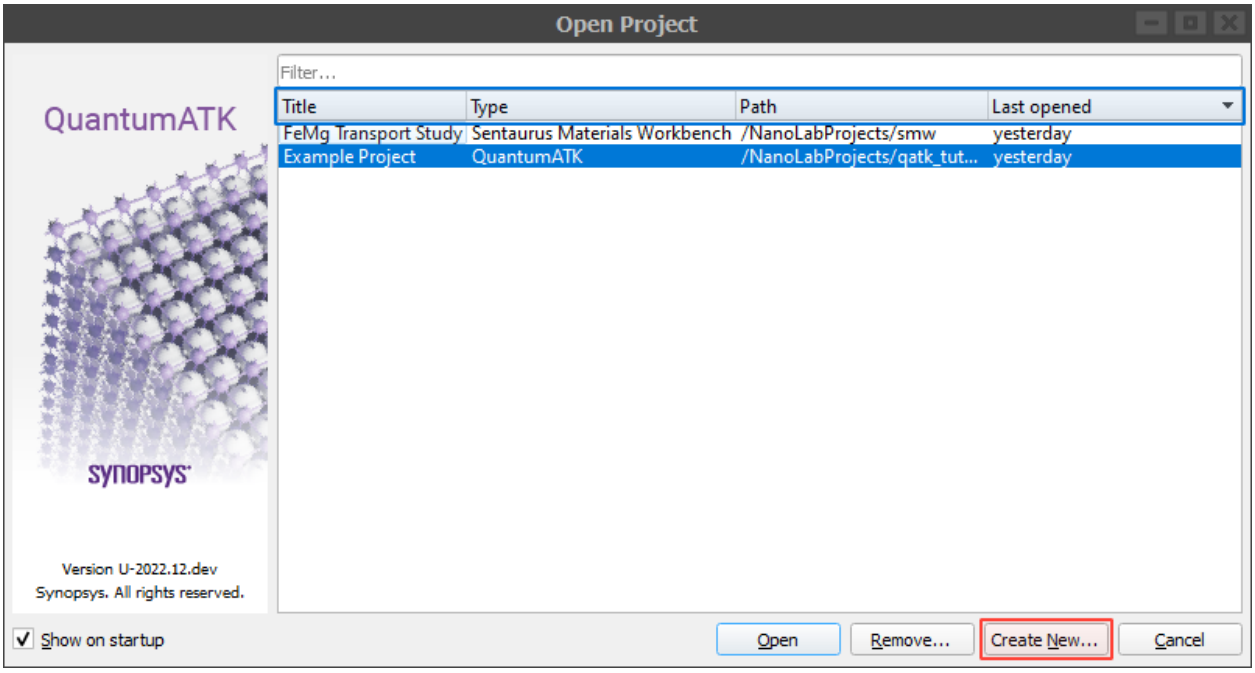
In the Introductory Tutorials we will show how to use QuantumATK to calculate many other properties and teach you how to choose the simulation parameters to best suit your goals. You can find them here: *Introductory Tutorials*.

2.2 Projects in NanoLab

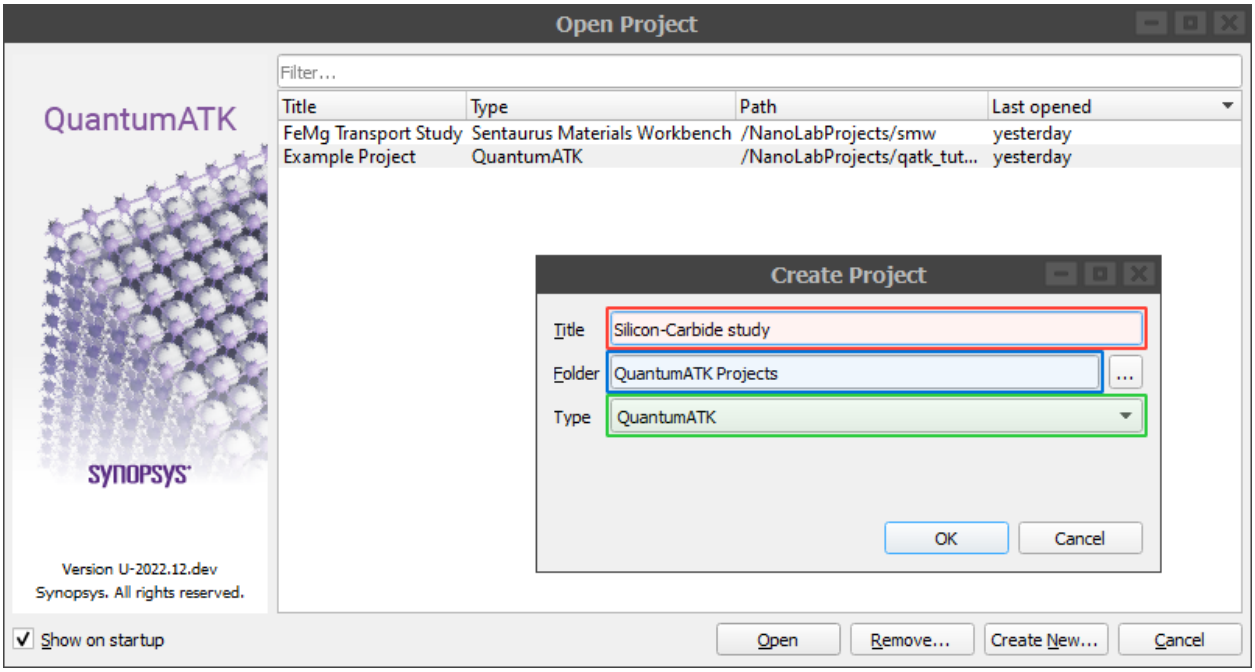
Here you will learn how to create a new project in *QuantumATK NanoLab*.

Lets start *QuantumATK NanoLab* by clicking the QuantumATK NanoLab icon on the Desktop. You will briefly see a splash screen followed by an *Open Project* window. Here you can manage projects, open one of them or create a new one.

In the *Open Project* window you will see a list of your existing projects, specifically their titles, types, paths and the last time they were opened. To create a new project click the *Create New* button.



A new dialog *Create Project* will appear. Each new project must be given a suitable title, type and a local path for storing project files.



Give the project a *Title* (here: Silicon-Carbide study), select the folder where data will be stored (here: QuantumATK Projects), select the *QuantumATK* type and then click **OK** and **Open** to start the project.

After creating a new project, *QuantumATK NanoLab* will generate a new file called `project.nl`. You will find it in the folder of your project. The *project.nl* will store general information about your project, such as the *data sources* and the *data filters* imported and created in the *Data* tool, atomic configurations from the *Builder* tool or workflows from the *Workflows* tool.

The following sections will introduce the main parts of *QuantumATK NanoLab* and the functionalities each part offers.

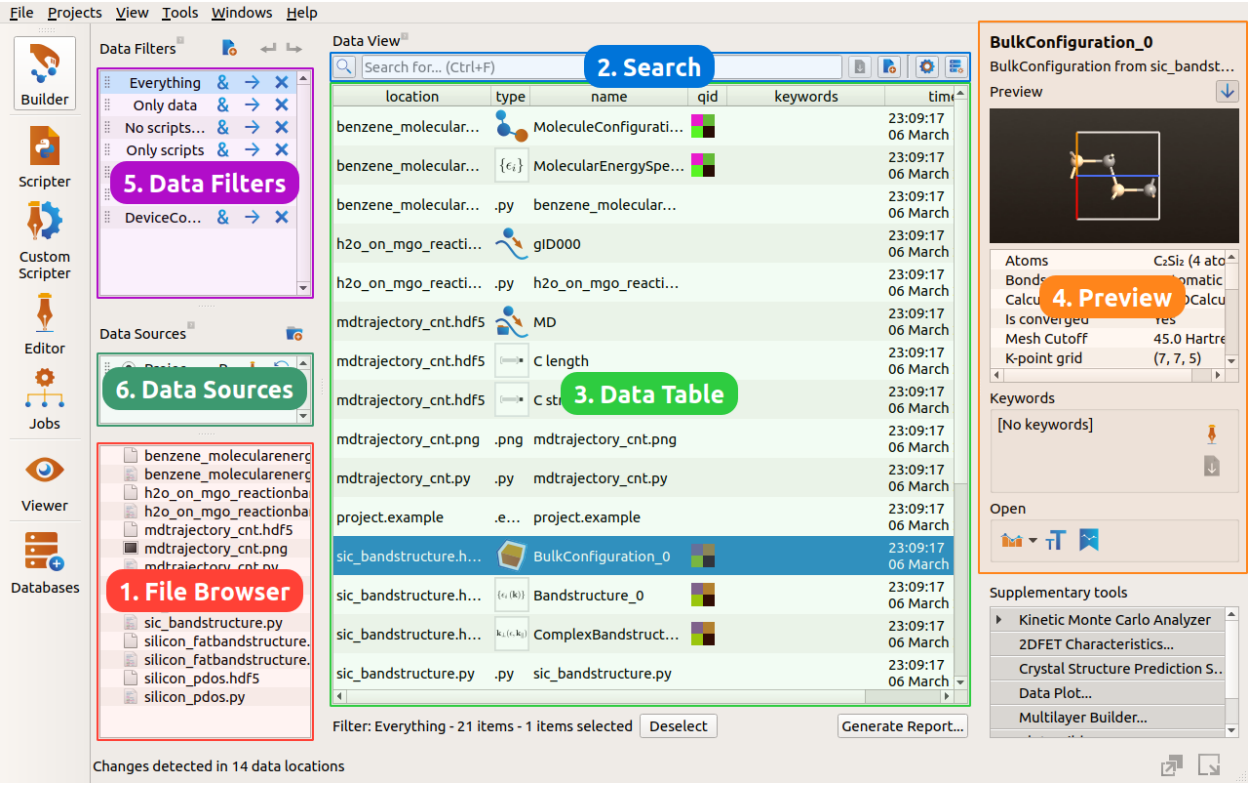
2.3 Organize your data in the Nanolab data view

Working with ab-initio simulations require a lot of data and it can be challenging to manage and organize data from many different simulations.

When opening Nanolab. the first window that shows up, is the Nanolab data view. The Nanolab data view makes it possible to filter, analyze and summarize all the constructed structures and calculated results.

2.3.1 The Nanolab Data View

This introduction to the Nanolab data view, will explain the 6 main parts of the data view:



2.3.2 1. The File Browser

The tutorial starts in the lower left corner, where you will find a familiar file browser showing the contents of the current project folder.

Select a file, and the data table (to the right of the file browser) will display the structures and results saved in that file. Select multiple files by holding down `ctrl` or `shift` while clicking the files with the mouse.

2.3.3 2. The Search Field

It is also possible to search through your data using the search bar at the top of the Nanolab data view. Type in search queries to quickly locate your data. Note that it will search across several columns in the data view – including folders, filenames, data type, object ids and keywords.

This makes it possible to quickly find, e.g. all **Bandstructures** or **MDTrajectories** within a relevant folder.

It is also possible to use more advanced SQL queries to create custom views on your data. It make take some effort to create the most powerful queries, and therefore the queries can be saved as what we call Data filters. Such saved filters can quickly be applied and combined with other queries in order to properly limit your search.

The advanced SQL search

In addition to normal free-text search terms, the search field also accepts SQL queries. It is even possible to combine text search and SQL search together in the search field. Here are some examples of how the search field can help locate relevant data.

This query search for all results of type **Bandstructure**:

```
type='Bandstructure'
```

This query searches for all data items that have a name starting with Electron. Note the use of the `%` sign to signify wild card matching.

```
name LIKE 'Electron%'
```

In addition to searching, the data can be sorted by e.g. the last changed time. Here the specifier `ASC` sorts the timestamp in ascending order. Similarly sorting can be done in descending order by using the `DESC` specifier.

```
name LIKE 'Electron%' ORDER BY timestamp ASC
```

Multiple queries can be joined by OR or AND. Here we search a specific subfolder for all bandstructures.

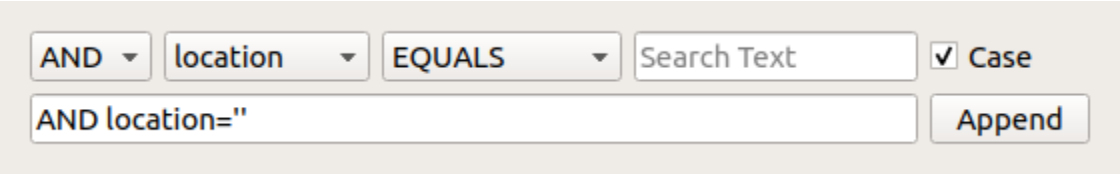
```
location LIKE 'my_subfolder%' AND type="Bandstructure" ORDER BY timestamp ASC
```

Also, normal search terms can be combined with SQL. Let us search for anything related to silicon of type Bandstructure and ordered by timestamp.

```
type='Bandstructure' AND silicon ORDER BY timestamp ASC
```

The Query Builder

Advanced SQL-searches can be difficult to master, and as a help Nanolab provides the advanced query builder. Click the cog-button to show the query builder. Fill in menus and text fields to construct the relevant queries.



2.3.4 3. The Data Table

The main data table shows all the data that matches the current search query. Each piece of data is represented as a single row in the table (in the following called a data item).

The columns displays the data location (folder and filename), data type, object id, Quantum ID (qid), keywords and last changed time.

Each data item can be double clicked. This will open a window with a relevant analysis, that will display the scientific data contained in the data item. Try double-clicking an atomic structure to open the 3D Viewer, a **Bandstructure** to show a plot of the band diagram or an **MDTrajectory** to show the time-evolution.

For more detailed options, right-click a data item. This will open a detailed menu with several options:

- **Open**. This will open the default analysis.
- **Open with**. Displays a menu of all available analysis for this data type.
- **Delete**. Some items support deletion. Note that deletions cannot be undone.
- **Unpack**. Show nested data (see explanation below).
- **Export** the selected data items to specific file formates (.hdf5, .xyz, .mol and so forth).
- **Refine** the search field based on the right-clicked data item, by choosing Refine

The Quantum ID (qid)

Calculations done with LCAOCalculator, PlaneWaveCalculator or SemiEmpericalCalculator are all marked with a unique number called a quantum ID (qid). The number can be cumbersome to work with, so in the data view it has been encoded as 4 colors. The four-colored squares can quickly be used to distinguish different calculations, since analysis done with the same calculator all share the same quantum ID.

Unpack

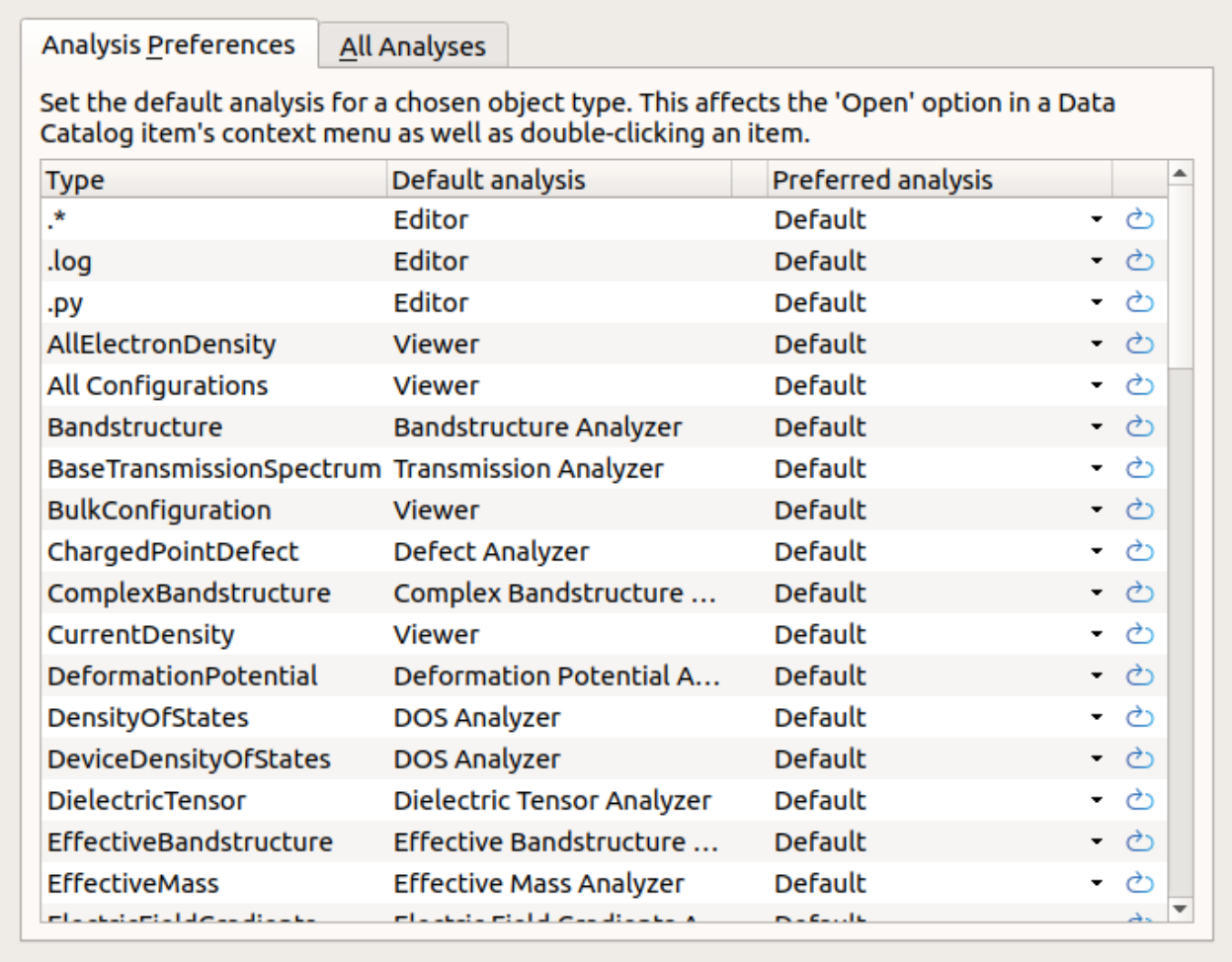
Some data may contain nested data. MDTrajectories will for example sometimes contain custom measurements of stresses and strain. All data items containing nested data are marked with a folder-like unpack icon .

Right click one of these items and choose **Unpack** to show the contents of the data. The nested data can then be inspected and analyzed like all other data.

Analyzers and Preferences

Double-click a data item to open it with its default analyzer. Bandstructures will open to reveal the band diagram, atomic structures will open in the 3D Viewer and so forth. Additional analyzers can always be found in the right-click menu.

It is possible to change the default analysis for a given type of data. Open the analysis preferences by clicking the bookmark button and you are presented with a list of types and analyzers. For each analyzer you can see the default and make a user-defined change. Those changes apply across all projects.



2.3.5 4. The Data Preview

Clicking a data item will display a preview of the relevant data in the preview panel to the right of the data table.

For atomic structure configurations the preview includes a 3D view of the atomic structure along with relevant data like bonds and unique elements. If the structure has a calculator attached, it is also possible to inspect the calculator settings. Many different data types items expose some results or settings in the preview pane.

The 3D view can be disabled by clicking the arrow-down button. This may be convenient when working on slower machines or with very large structures.

The Keywords

Data items can also be annotated by keywords that can be used for categorization and search. Atomic structures will also automatically embed the unique elements and its calculator name as keywords.

The preview panel also contains a keyword editor, where it is possible to add custom keywords to your data items. This can be a convenient way to attach additional data to your data items, or for enabled special searches or filters.

2.3.6 5. The Data Filters

In the upper left corner is a list of saved searches known as **Data Filters**. Per default each project starts with a few standard filters that has been given appropriate names. The **Only molecules** filter for example corresponds to the advanced search query `type='MoleculeConfiguration'`.

Each filters can be applied by clicking their arrow button . This will replace the currently active search query with the filter query. It is also possible to combine multiple filters by using the button.

This & feature makes it very easy to perform advanced searches like for example all molecules within the MyStructures folder:

1. Click the MyStructures folder in the folder view.
2. Click the next to the Only molecules default filter.

2.3.7 6. The Data Sources

The data sources specify what data belong in the current project. The table shows the list of current sources.

Each project always a contains single folder (the project folder) as the main data source, but it is possible to add additional folders as sources. Simply click the button and navigate to the desired folder in order to add it to the current project.

Each source has a unique name. The project folder is always named `P` while subsequent sources are named `S0`, `S1`, etc. Clicking the source name will limit the search to the corresponding source by changing the search query to e.g.,

```
source='S0'
```

The settings for each data source can be accessed by clicking the pen button . In these settings the current source folder can be changed along with the activated data inspectors.

Data Inspectors

A data inspector can turn files into data in the data table. A standard source is capable of automatically import data from QuantumATK files, .xyz files, .cif files, LAMMPS files, VASP files and QuantumEspresso files.

However, NanoLab supports many different file format inspectors that can be activated in the settings menu. Simply tick off the desired inspectors to enable them, and wait a few seconds for Nanolab to re-index the files.

2.4 Builder Manual

2.4.1 Introduction

The **Builder** is an essential part of the QuantumATK graphical user interface to the QuantumATK calculator engines. The Builder provides a range of powerful tools for building advanced atomic-scale structures, e.g. bulk crystals, 1D nanowires, 2D nanosheets, clusters, and device configurations with either one or two probes. After a structure is built, it can be used either in QuantumATK calculations or be exported to external file formats for use in other atomic-scale simulation packages.

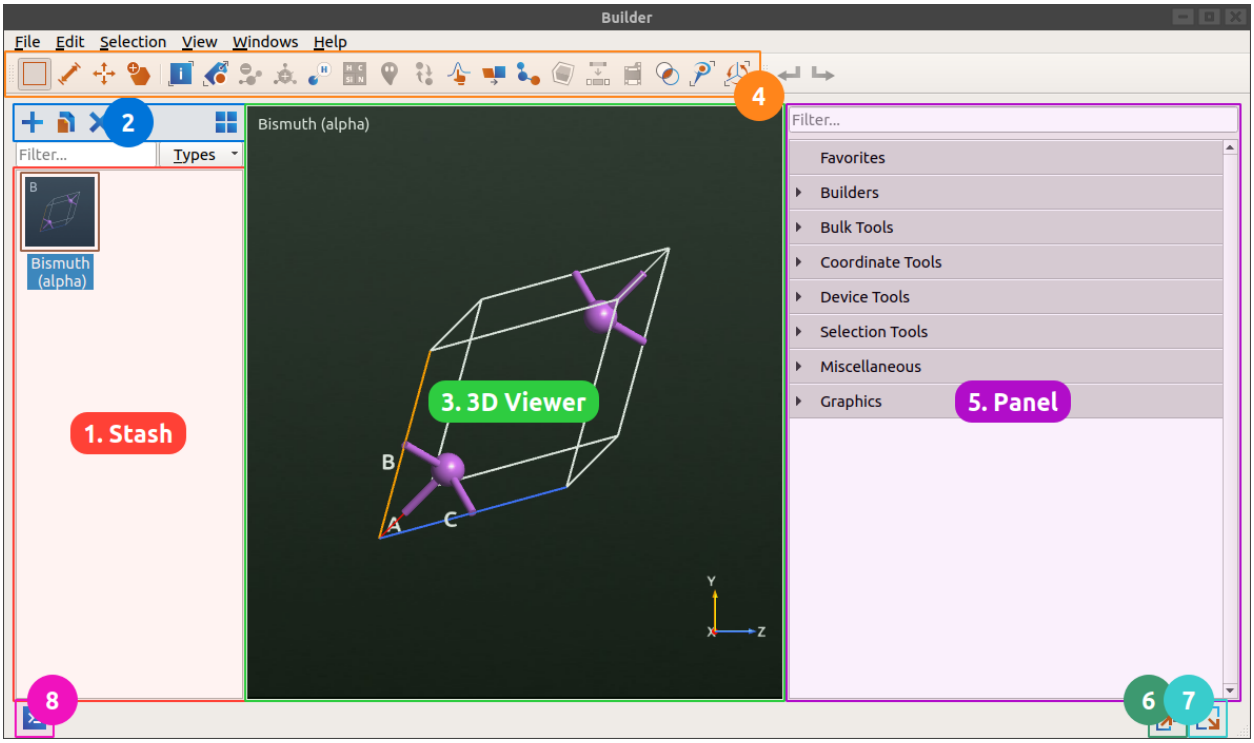
This manual provides an overview of the Builder functionalities. For specific advice and tips for building atomistic structures, please refer to the QuantumATK Tutorials website and visit the public QuantumATK Forum, where specific questions may be answered by experienced users.

Layout and basic operations

When the Builder starts up for the first time in a QuantumATK session, it is empty. You will soon learn how to add structures to it, but let us first get an overview of the Builder main window.

As illustrated below, the main parts of the Builder are:

1	Stash	Storage for all created structures.
2	Stash toolbar	Tools for adding items to the Stash, and for copying and deleting Stash items.
3	3D viewer	Visualization of active Stash items.
4	Toolbar	Simple plugins for modifying Stash items, e.g. moving a particular atom.
5	Panelbar	Advanced plugins for editing Stash items, e.g. creating a device configuration.
6	Drag-and-Drop	Use the mouse to drop the active Stash item onto another QuantumATK tool.
7	Send-to button	Use a menu list to send the active Stash item to another QuantumATK tool.
8	Console button	Use a console to execute python code.



Stash

All structures you are working on in the Builder are stored in the Stash. Each Stash item is represented by an icon, and corresponds to a single configuration.

Stash items can be **selected** and/or **active** – there is a difference: You can select more than one structure, but only one can be active. Some Builder plugins operate on more than one Stash item at a time (e.g. to combine two configurations into one), and you can therefore select other items than the currently active one.

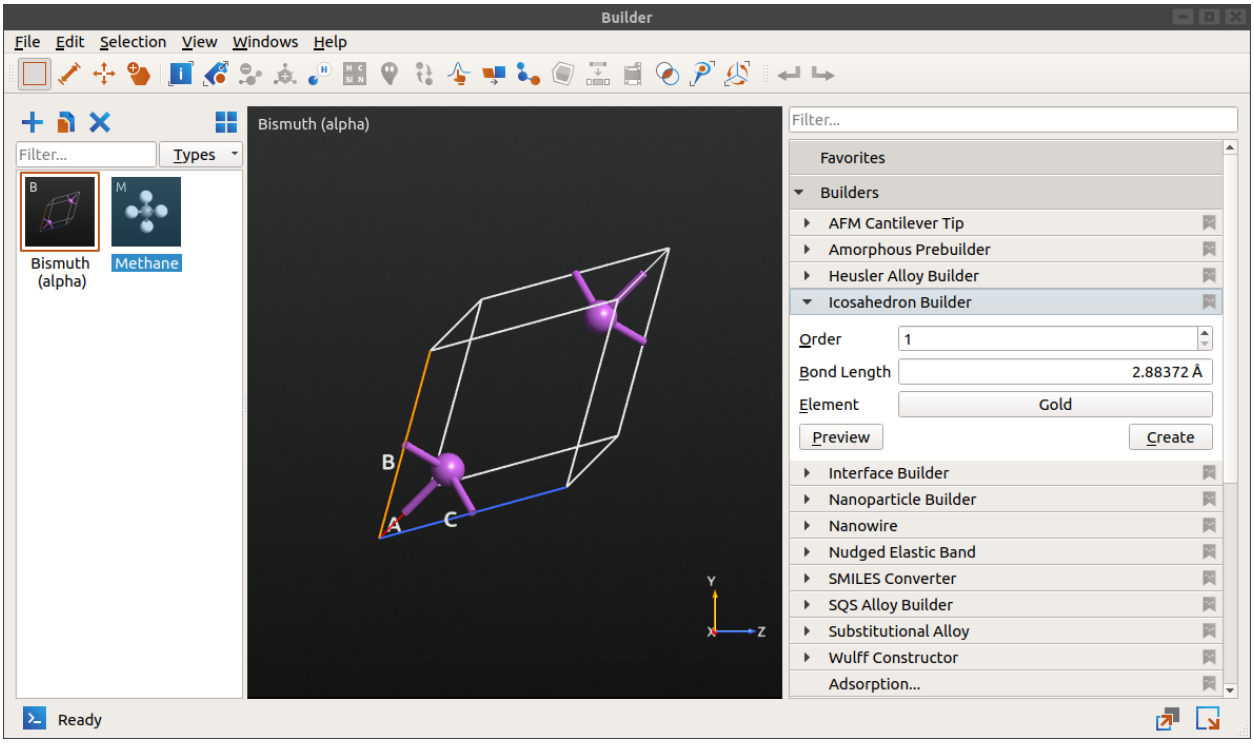


Fig. 2.1: The Stash item named Bismuth (alpha) is **active**, as indicated by the red line around the icon. The active item is always the one shown in the 3D viewer, and operations like delete atom or change lattice constant will be carried out on this structure. The Stash item named Methane is merely **selected**, which is indicated by the gray background.

Stash operations

There are several ways to add new structures to the Stash:

- You can simply **drop** configurations from the QuantumATK main window onto the Builder. The structure is automatically added to the Stash. See the tutorial introduction_quantumatk0 for an introduction to the QuantumATK main window.
- You can use the **Stash toolbar** to add completely new structures to the Stash. The **Add** menu contains a range of tools for creating configurations from scratch.
- You can also use **Add** menu to import structures from predefined databases and saved files.

The Stash toolbar also allows you to **Copy** and **Delete** active items. When you copy an item, a new and identical item is added to the Stash.

Right-click a Stash item to open its context menu. This allows you to rename, copy, delete, and in various ways save/export the structure to files, or send the structure to another QuantumATK tool.

Tip: The keyboard shortcut **F2** may be used for renaming Stash items, while pressing the **Delete** key is a shortcut to the Delete operation.

Plugin tools

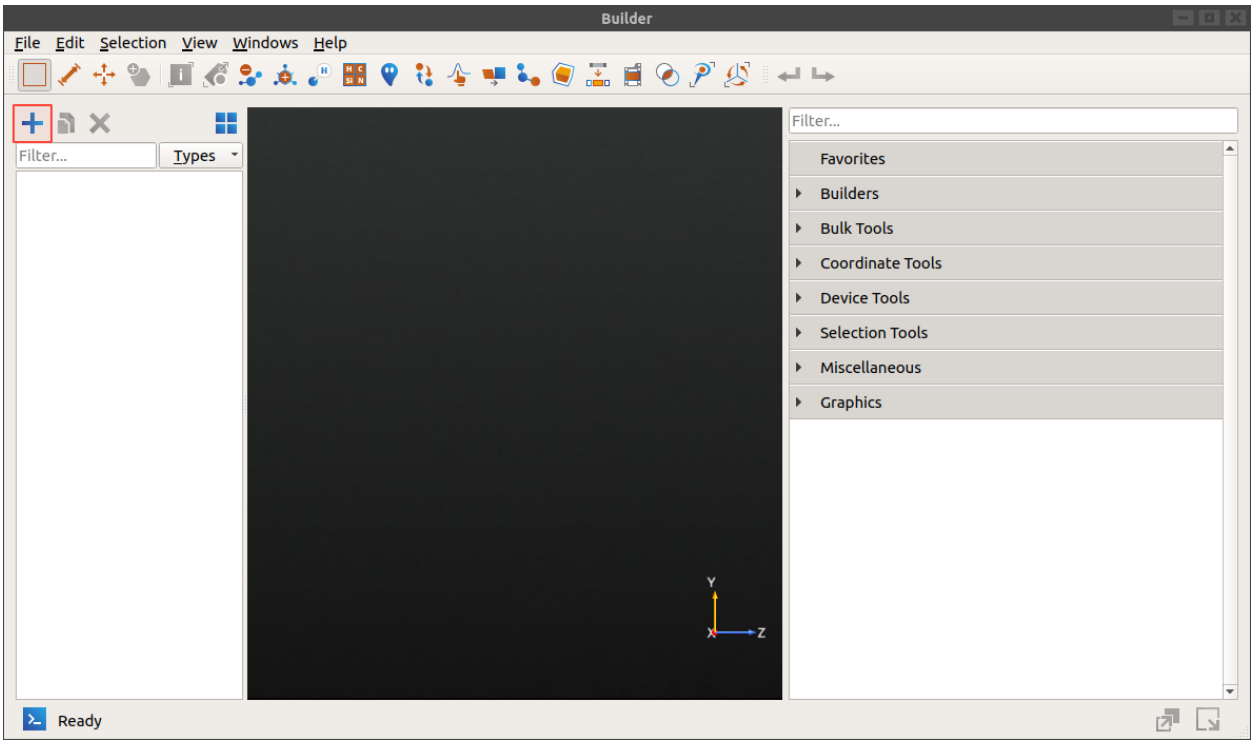
As indicated in Fig. 2.1, the horizontal **Toolbar** above the 3D viewer and the vertical **Panelbar** to the right of the 3D viewer both contain plugin tools for interacting with Stash items, and for applying more or less complex operations on the active Stash item, e.g. change the lattice constant of a bulk configuration. Note that each plugin tool usually operates on the active Stash item only.

The horizontal Toolbar plugins are applied by clicking the icon. Hover the mouse over each icon to see a tooltip describing it.

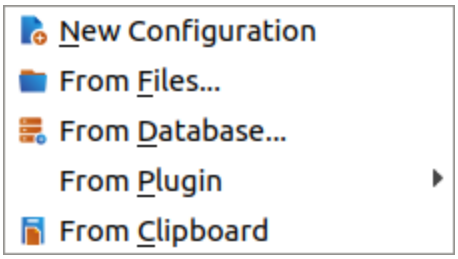
The vertical Panelbar plugins are grouped in panels, e.g. *Bulk Tools* and *Device Tools*. Click a panel to expand it – a list of relevant plugins is revealed. You can also read more about each plugin in our *Overview of built-in Builder plugins*.

2.4.2 Importing, adding, and exporting structures

There are several ways to **insert new structures** into the Builder. Most of them can be accessed selecting **Add** highlighted in red on the Stash menu.



The window highlighted in red, that opens when clicking the button, shows the different options to add a new structure.



Adding an already existing structure

Add a structure from a file

A structure can be **added** from a file by choosing *Add* → *From Files*. For a list of **supported file formats**, see the *Plugins section*.

If the file is a **natively supported file format** (currently QuantumATK Python/HDF5, XYZ, CIF, CAR, VASP) you can also **drag and drop the file from a file browser** (like Explorer in Windows, or the built-in file browser in QuantumATK) onto the open Builder window or onto the Builder icon on the toolbar in the main QuantumATK window to import the file.

It is also possible to **drag and drop a Python script (as text)**, e.g. from an Editor, onto the Builder. Finally, you can **paste a Python code from the clipboard** into the Builder by choosing *Add* → *From Clipboard*.

Tip: If you select multiple files, they will be imported as individual Stash Items.

Structure databases

QuantumATK contains an **internal database** with several hundred crystals, molecules and fullerenes. To add a structure from this database, click *Add* → *From Database*, and select the relevant database from the tabs (for example, *Crystals*).

In the *databases* window, type the name of the desired structure in the search field to filter the list.

Once you have located and selected the desired structure, click the icon highlighted in red to add it as a new Stash Item. You can also double-click selection to add that structure.

Creating a new structure

New configuration

To **create a brand new structure**, click *Add* → *New Configuration*. This will insert a molecule with a single hydrogen atom, and then you can manipulate this further by using a combination of the plugins available in the Builder such as the *Molecular Builder*.

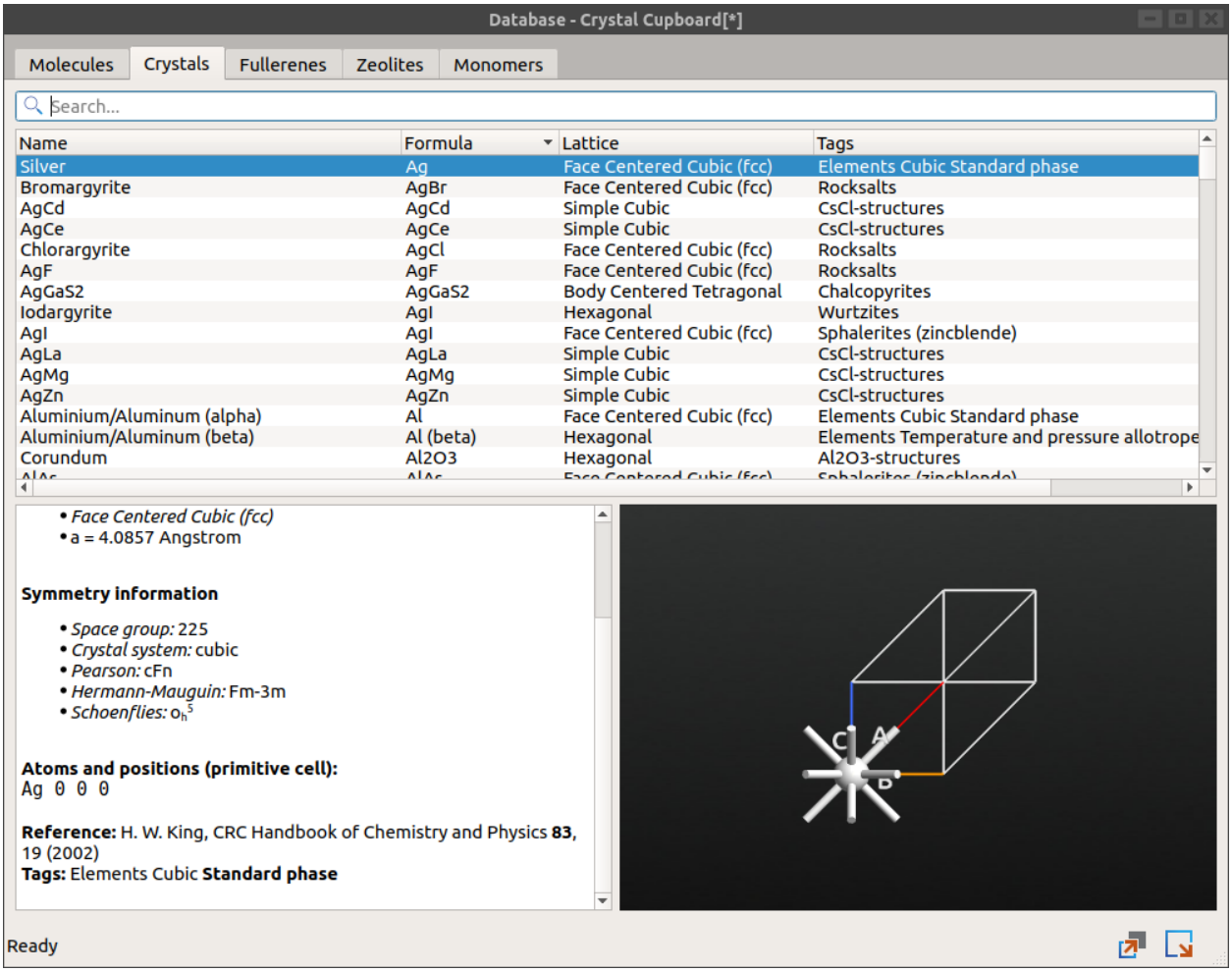


Fig. 2.2: Use the tabs to select between the Crystal, Molecule, Fullerene, Zeolite and Monomer databases.

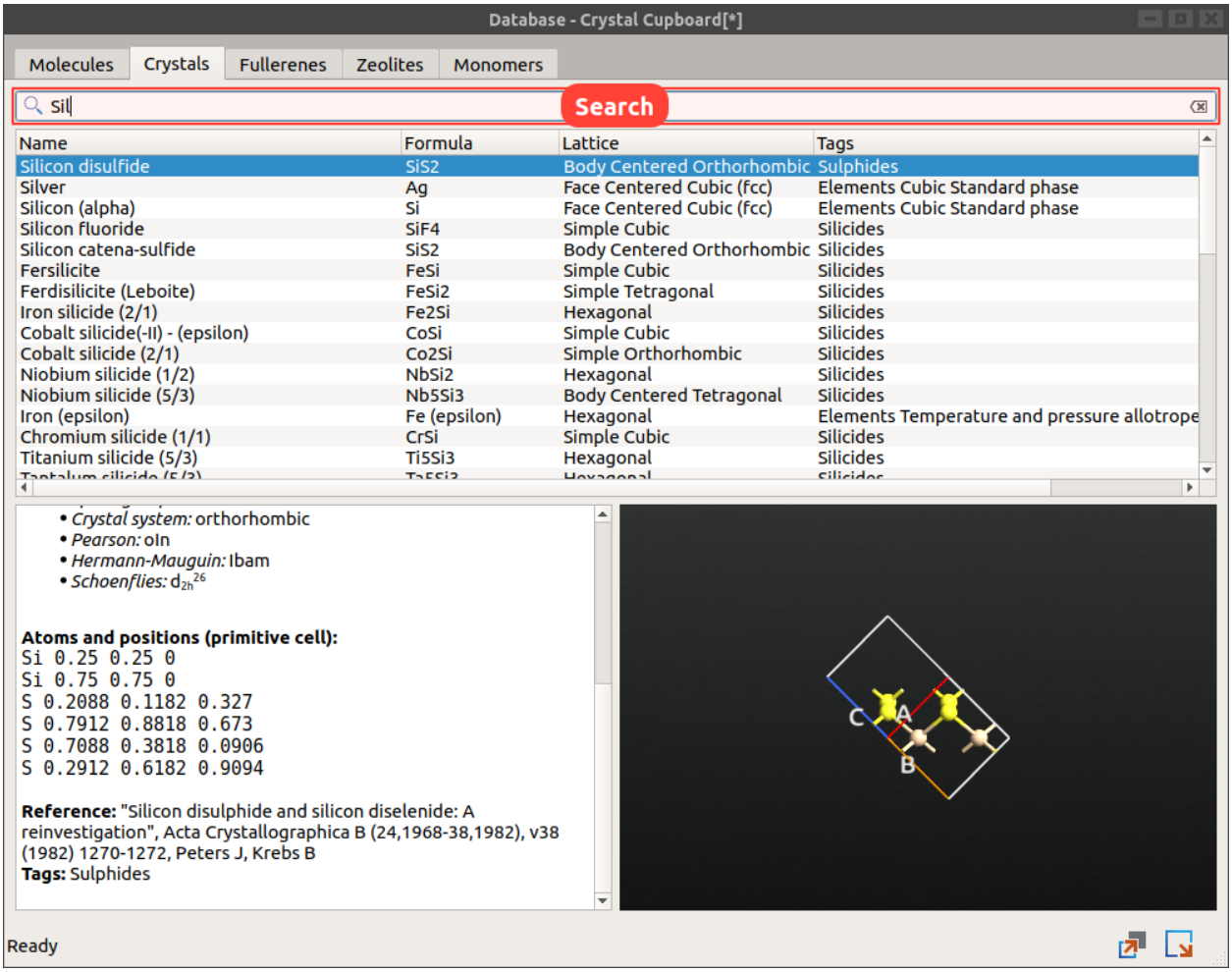


Fig. 2.3: The Crystal database. Use the search field to look for your desired crystal structure.

Custom builders

Each entry in the *Add → From Plugin* menu is associated with a plugin. These plugins can be simple and just take a few parameters, or use a very elaborate interface to create complicated structures.

Note: While other plugins normally operate on existing Stash Item, these plugins create a new Stash Item.

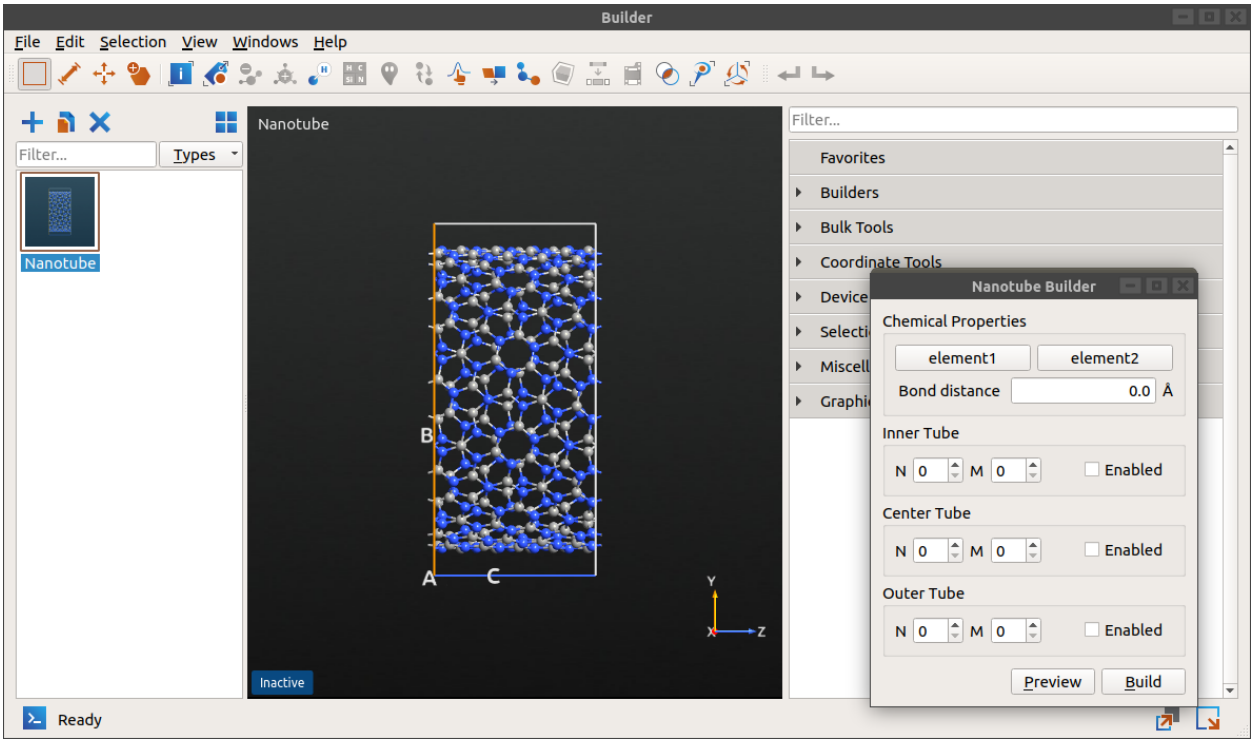


Fig. 2.4: Building a nanotube, also multiwall, in QuantumATK is very easy - just use *Add → From Plugin → Nanotube*.

The Builder comes with the following **built-in plugins**:

•Nanotube

•Nanoribbon

•Nanosheet

•*Crystal builder*

•Magnetic Tunnel Junction

Tip: In addition to the built-in plugins, you can also *write your own plugin or download and install new addons* to extend the list, to make it easier to build a particular new type of structure.

Combining, modifying and converting structures

In many cases it is necessary to **combine** two structures into one, or to **convert** one type of configuration to another, e.g. bulk to device.

In the Builder there are plugins that can be used to modify or combine structures for special purposes. They are located in the plugin panel on the right-hand side of the Builder. Two examples are:

- the *Interface Builder*, for combining two structure to form an interface, which can be accessed by clicking on *Builders* → *Interface*;
- the **Surface (cleave)** plugin for cleaving a crystal, which can be accessed by clicking on *Builders* → *Surface (Cleave)*.

Another possibility to is to **drop a Stash Item from the Stash onto the 3D window**; this will add the atoms of the dropped item to the active configuration (only the atomic positions are used, not unit cells or other properties). When you do this, the **Builder** activates the **Move Tool**, which is a powerful way to interactively translate and rotate fragments. For more details, see the dedicated *Move tool section*.

To **convert a bulk structure to a device**, use *Device Tools* → *Device From Bulk*. Similarly, to **convert a bulk structure or slab to a surface**, use *Device Tools* → *Surface From Bulk*. To go back, click the button on the toolbar. You can also **convert a periodic structure to a molecule** (removing the unit cell) using the button.

Saving and exporting structures

Once you have created a structure, you may want to **save it as an QuantumATK Python/NetCDF file** or **export it to a non-ATK format**.

Save

To save the active Stash Item into a Python or HDF5 file, press `Ctrl + S` (or use the menu *File* → *Save as*) and use the file dialog to specify the file type and location.

> **Warning:** When saving into HDF5 files, the structure is appended if the file exists.

Export

To export the active Stash Item to a non-ATK file format (like XYZ, CIF, etc), press `Ctrl + E`, or use the menu *File* → *Export configuration*.

Note: To save or export another Stash Item than the active one, right click it and choose *Save as* or *Export* from the context menu.

Sending the structure to other QuantumATK tools

As all instruments in **QuantumATK**, the **Builder** window has a **script drag** button and **Send To** button in the lower right-hand corner (see Fig. 2.1). Using these, you can transfer the active structure to e.g. the **Script Generator** to set up a calculation. To send a particular Stash Item to e.g. the Editor, right-click it and choose *Send To* from the context menu.

2.4.3 Mouse and key operations

In this section, you will learn how to operate the 3D view and manipulate atomic structures in a simple way, including how to select a specific atom or a group of atoms.

Controlling the 3D view

Using the **right button** and **scroll wheel** of the mouse, you can perform the following operations for the 3D view.

- **Rotate** can be done by moving the mouse while holding down the right mouse button.

  If the mouse pointer is near the center of the 3D view, the rotation will be done as a full 3D view rotation.

  Having the mouse pointer at the 3D view edge constrains the rotation to the view plane. If you first align a particular axis with the screen normal, this function will then allow you to rotate the structure around this axis in a controllable manner.

- **Zoom** can be easily done by rolling the **scroll wheel** of the mouse.

  If you do not have the scroll wheel, you can do **Zoom** by holding down the **Ctrl** key, simultaneously pressing the right mouse button, and then moving the mouse back and forth to actually zoom the structure.

- **Pan** can be easily done by pressing the **scroll wheel** down, and then moving the atomic structure with the mouse.

  If you do not have the scroll wheel, you can **Pan** by holding down the **Shift** key, simultaneously pressing the right mouse button, and then moving the mouse to actually pan the structure.

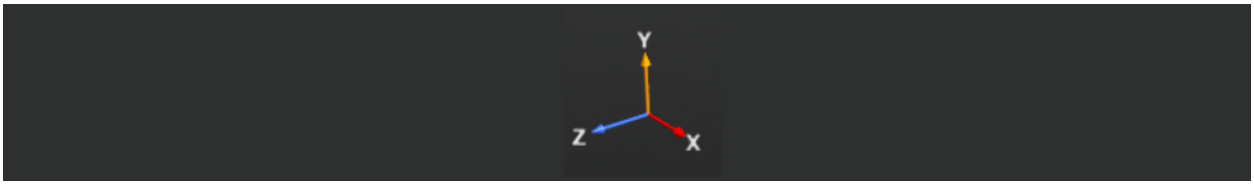


Fig. 2.5: The compass in the upper right-hand corner helps you to orient yourself when rotating the 3D view.

Press **Ctrl+R** to reset the 3D view to the default orientation and zoom.

For a periodic system, the crystal axes labeled as **A**, **B** and **C** are given with the red, yellow and blue sides of the periodic cell, respectively. The coordinate origin is placed at the cell vertex where all the three crystal axes do intersect.

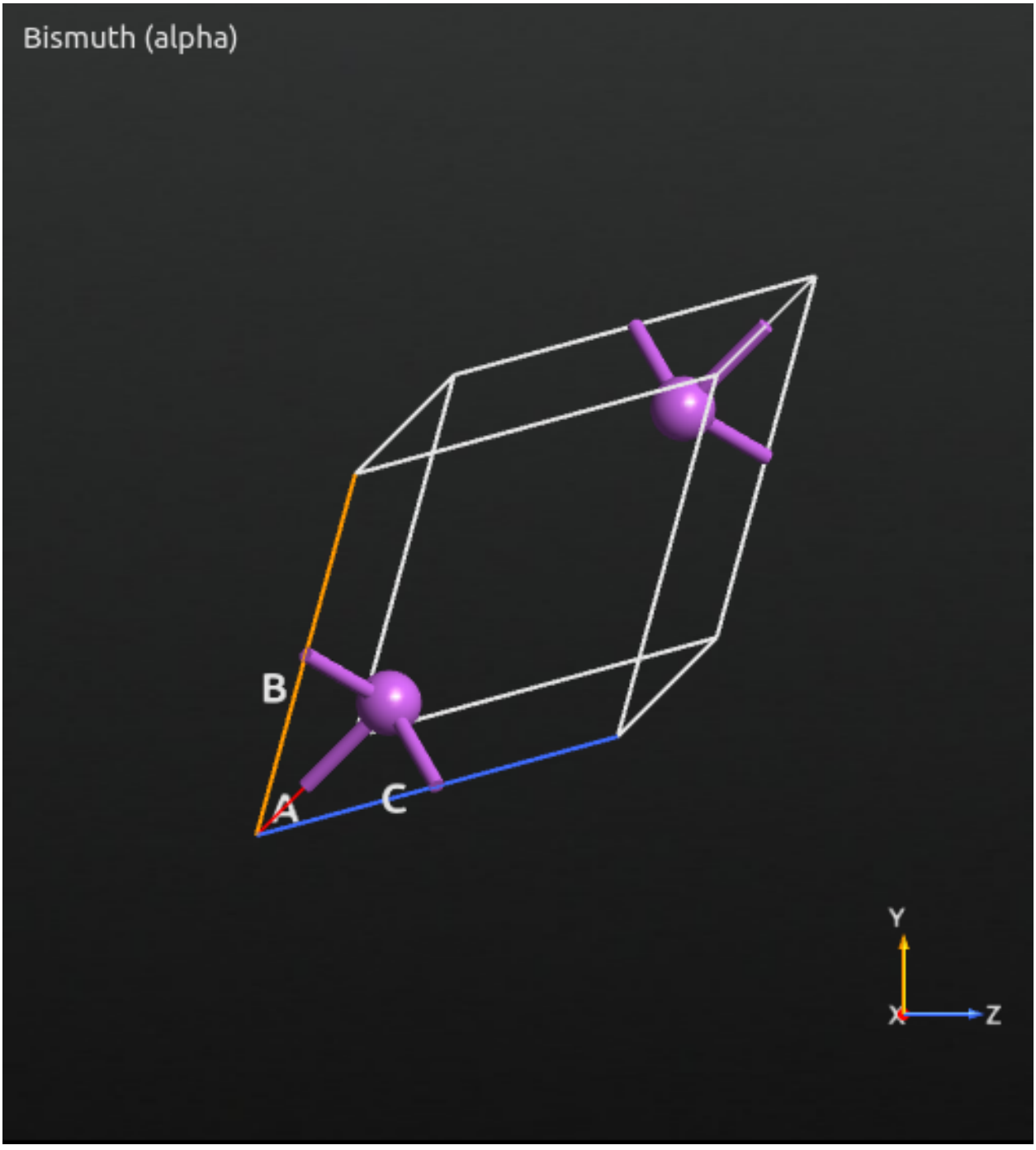


Fig. 2.6: The red, yellow and blue box edges are used to indicate the A, B, and C crystal axes for a periodic structure, respectively. These are the same colors as used on the *compass* for the X, Y, and Z axes of the Cartesian coordinate system. In general, the crystal axes are not parallel to these coordinate axes, even for a cubic cell.

Preview mode

Some of the plugins have a **Preview** button, and may even update the preview when some of the parameters are changed. The preview state is indicated by the red background color and a text in the upper left corner of the 3D window.

Important: Keep in mind that the stash item has not been modified yet. To confirm the operation, you actually need to press the relevant button, often called **Apply**.

To dismiss the preview, you can just click on the 3D window. You can then bring it back by clicking on the plugin **Preview** button again.

Note: All the other plugins become unavailable when a preview is shown. It is meant to avoid confusion about which structure you are operating on.

Selection

Selecting an atom or a group of atoms is often an initial step for those plugins that allow us to manipulate the atomic structure in the Builder. A number of plugins apply their operations on selected atoms only. There exist some plugins that act on the whole structure if no atoms are selected.

The Builder therefore has many ways to select atoms and manipulate selections. The selected atoms are indicated by a yellow halo.

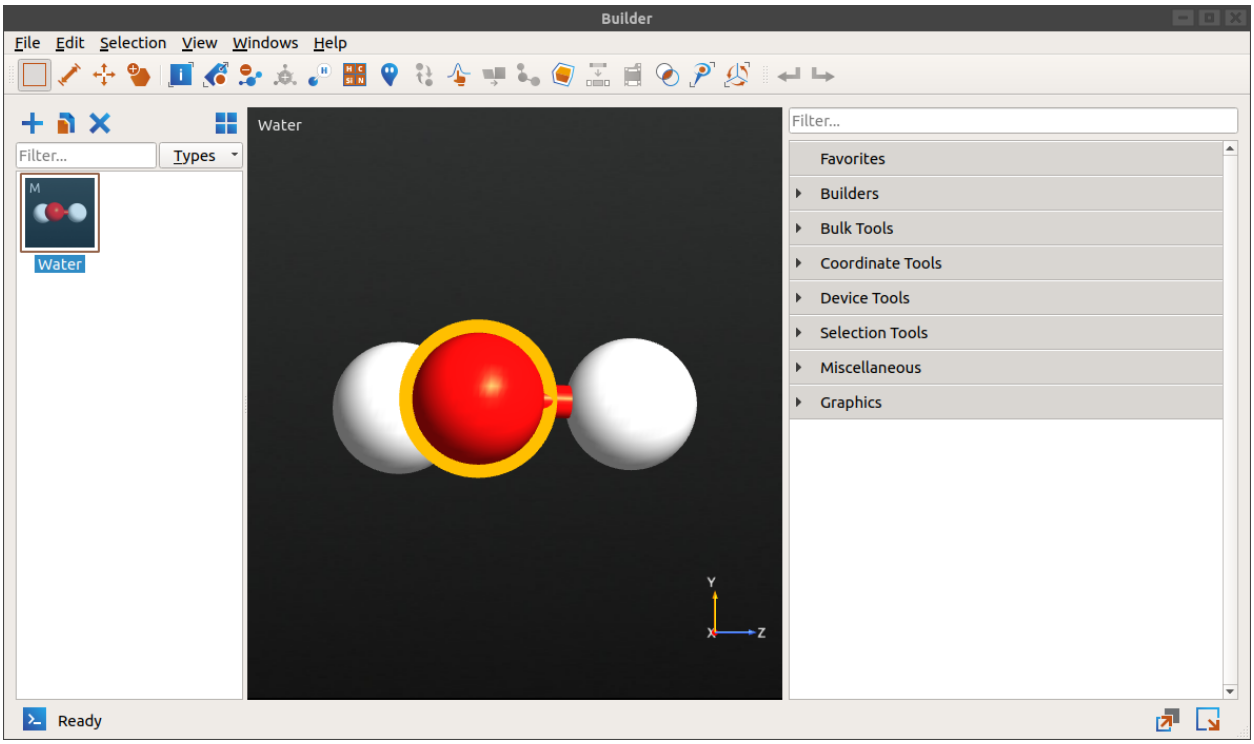


Fig. 2.7: The oxygen atom in this water molecule is selected, as indicated by the yellow halo.

To **select an individual atom**, click on it with the **left mouse button**. Note that this operation will cancel any existing selection of atoms that has already been made in the structure.

To **add an atom** to an existing selection, hold down the **Ctrl** key and then click on this atom. If you click on an already selected atom, this atom will be removed from the selection.

To **select a group of atoms**, you may make use of the **shape select tool** by first clicking on the icon in the top toolbar, and then choosing one of the following shapes for the selection area.

- **Rectangle** .
- **Circle** . The circle is centered at the initial mouse position.
- **Lasso** (free-form) .

Subsequently, the **left mouse button** can be exploited to draw a figure of the chosen shape to enclose the atoms of interest.

Note that the shape select tool operates on all the atoms that are visually inside the shape drawn by the mouse as well as on those atoms that may be hidden behind the other atoms, but are virtually projected on the shape. The single-click mouse select operation only applies on the atom you have clicked on.

The selection operations can be modified in various ways using the **Ctrl** and **Shift** keys.

- To **add a group of atoms** to an existing selection, hold down the **Ctrl** key and select the atoms of interest with the shape select tool. In this case, the already selected atoms that may appear inside the new selection area will not be deselected.
- To **undo the selection** for the already selected atoms, hold down the **Shift** key and draw a shape with the shape select tool to enclose the atoms to be deselected.
- **Double-click** on an atom in a **fragment** (a fragment is a set atoms connected by bonds) with the **left mouse button** to select all the atoms of this fragment. If you hold down the **Ctrl** key and double-click on an atom of the fragment, all the fragment atoms will be added to the existing selection, or removed if the clicked atom was already selected.
- By holding down the **Shift** key and **double-clicking** on an atom, you can **select** all the atoms of **the same chemical element** as the doubled-clicked atom in the atomic structure. If you simultaneously hold down the **Ctrl** and **Shift** keys, and then **double-click** on the atom in the fragment, the fragment atoms will then be added to the existing selection, or removed from the selection if the double-clicked atom of that fragment was already selected.

There exist several other useful keyboard tricks.

- To **deselect** all atoms, left-click on the background of the 3D window, or press the **Esc** key.
- Press the **Ctrl+A** keys to select all atoms.
- Press the **Ctrl+I** keys to invert the selection. In this case, the atoms of the original selection will be deselected, wheres all the other atoms of the atomic structure will be included in a new selection group.

Advanced select

Sometimes the atoms you want to select are not easily accessible with the mouse, or you need a detailed control of the precise atoms to select. In this case, you can use a set of the following plugins available under *Selection tools*.

- **By Expression** plugin allows you to select atoms based on logical expressions.

For example, you want to select all the oxygen atoms with a Z coordinate larger than 1.5 Å or an X coordinate smaller than -0.5 Å. This can be easily achieved by typing

$$e = 0 \quad and \quad (z > 1.5 \quad or \quad x < -0.5)$$

in the *Expression Select* input field, and then pressing the **Enter** or **Tab** key.

•Using **By Element** plugin, you can select atoms of a particular chemical element.

•**Close Neighbors** plugin is designed to locate completely or almost overlapping atoms.

The overlapping atoms may occur, for example, as a result of a mirror operation, or a translation. This kind of atoms are hard to spot visually, but they must be removed to perform any further calculation for the atomic structure constructed in the Builder.

Select by tags

In the *Selection tools*, there exists an additional selection feature **Tags** that allows for assigning one or multiple tags to an atom or a group of atoms. So, each atom can then have many tags, and the same tag can apply to many atoms in the atomic structure. A tag can be any label that identifies an atom or a set of atoms.

One of the purposes of having tags is that you can assign them to a set of atoms when you build the structure, and then use them later on, e.g., in the Script Generator, to assign a special basis set, or initial spin to those atoms. Instead of making a list of the atom indices for all those atoms of special interest, you would just say *use basis set X for atoms with tag Y*. For example, for a work function calculation one might use a basis set with a longer range for the surface atoms than for the bulk atoms in the structure.

To **add, remove and modify tags** on atoms, use the plugin *Selection* → *Tags* plugin. This placement of the plugin indicates that tags are also a handy tool for selecting atoms. The fastest way of selecting atoms with tags is to click on the icon in the top toolbar.

Deleting atoms

To **delete atoms**, you should first select them in the structure, using any selection method discussed above, and then press the **Delete** button on the keyboard.

Tip: It is possible to delete atoms from the structure, using *Coordinate Tools→Coordinate List*. You can also **insert atoms** with this tool.

Undo

Any operation (except selection) in the Builder can be undone by pressing in the top toolbar. The undone operation can also be redone using the icon, if needed. The keyboard shortcut for undo is the **Ctrl+Z** keys, while the **Ctrl+Y** keys stand for redo.

Note: Some of the plugins do not change the atomic structure in the stash item in which you have initiated the change, but they rather create a new stash item for the structure with the change implemented. In this case, there is obviously nothing to undo but delete the new item if it is not what you wanted.

2.4.4 Using the Move Tool

The Move Tool in the Builder is used to move, rotate, and align atoms, using either the mouse or exact coordinate and vector control. This section will explain the basic features and how to use them.

To **activate** the Move Tool, click the button on the toolbar. The Move Tool will also automatically be activated if you drop a system from the stash onto the 3D window (see *below*).

When the Move Tool is active, a widget appears in the panelbar in the right side of the window, where you can set some parameters that control the operations.

There are four modes of move interaction - Translate, Rotate, Align and Advanced Move. Each mode can be activated either by selecting buttons available in the 3D window, or by selecting one of the tabs in the widget.

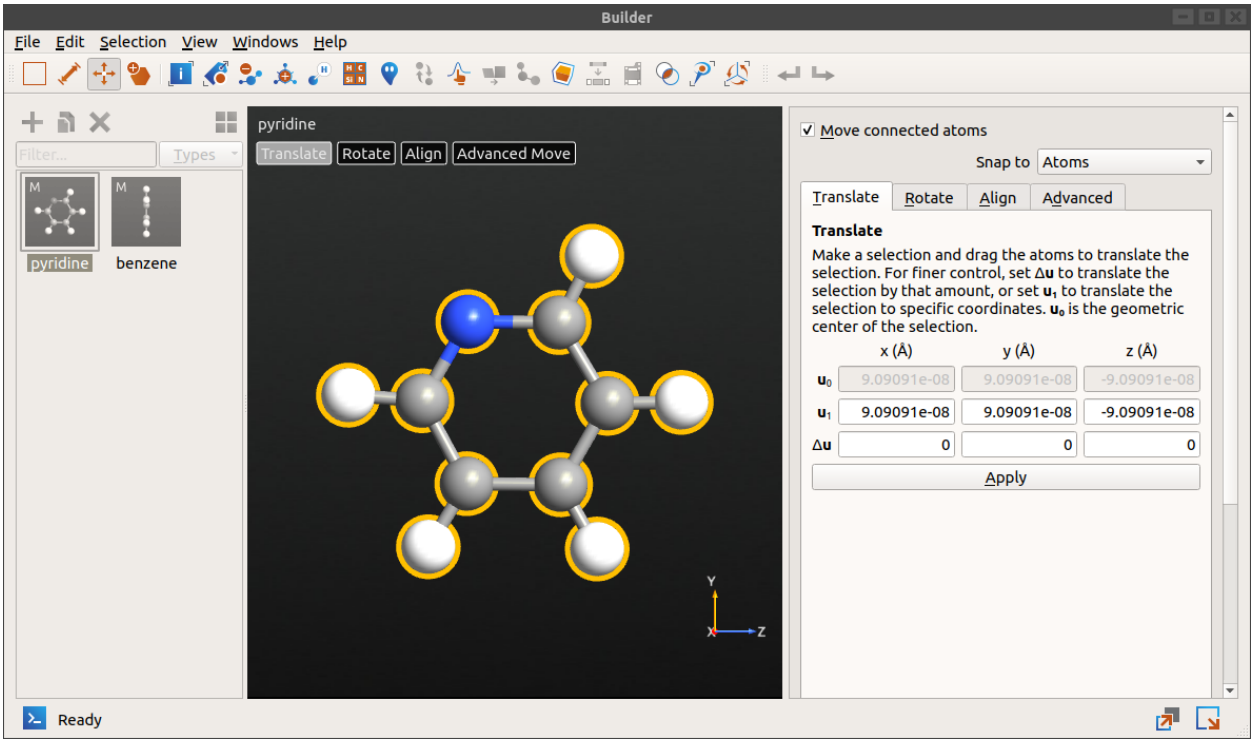


Fig. 2.8: Benzene molecule as an active Stash Item. Move Tool was activated by clicking .

Also note that when the Move Tool is active, all other plugins are unavailable, to avoid interference (in the same way other plugins are disabled in *Preview mode*). To **use any other plugin**, you must first **exit** the Move Tool, either by closing the widget, or by clicking the shape select button on the toolbar at the top of the window.

Selection control

The Move tool **operations** are applied to the atoms which are selected. In Fig. 2.8 all atoms in benzene molecule are selected.

Tip: If you **hold** your mouse on an atom or a bond in the window, you can get detailed **information** on that atom or bond.

Translation

To **move (translate)** the selection **drag** one of the atoms in the selection group to a new position, holding down the left mouse button. If multiple atoms are selected the entire selection group will follow rigidly, i.e. the same translation vector will be applied to each selected atom.

The translation is carried out in the view/screen plane (except for *snapping*, as explained below).

Position or move the center of the selected group by entering its new cartesian coordinates in the table given to the right of the window. Change the row for u_1 to set a new position, or row u to move the center of selected group for a specific amount. The center of the group is given in the row u_0.

Anchor atoms

Another way to activate the Move Tools is to drop a structure from the Stash onto the 3D window. In the Fig. 2.9, benzene molecule has been dropped with the pyridine. In a addition, three anchor atoms have been selected (red markers).

When the Move Tool is active, you can **select** up to three atoms within the selection group, which are used to guide the actual **move operations**. These atoms will be indicated by **red selection markers**, and will also be marked by the numbers 0, 1, and 2 to show the selection sequence. These are called **anchor atoms**. In Fig. 2.9 you can see that anchor atoms have been selected.

You can also see in Fig. 2.9 that when the anchor atoms are selected, the three **table rows in the widget** are populated. The three rows show:

- r_0, the position of the first anchor atom (index 0).
- $r_1 r_0$, i.e. the vector from the first to the second anchor atom.
- $r_2 r_1$, i.e. the vector from the second to the third anchor atom.

The significance of these vectors, or rather how to use them in a powerful way, will become more obvious in the following sections. As a general rule, select:

- **One** anchor atom to **translate (move) the fragment**,
- **Two** anchor atoms to **adjust the angle** between two bonds,
- **Three** anchor atoms to **adjust a dihedral angle**.

To **deselect** all anchor atoms, click the **background** with the left mouse button. To deselect one anchor atom, just left-click it again (if you deselect the first anchor atom, all anchor atoms are deselected, and if you deselect the second anchor atom, the third one is also deselected).

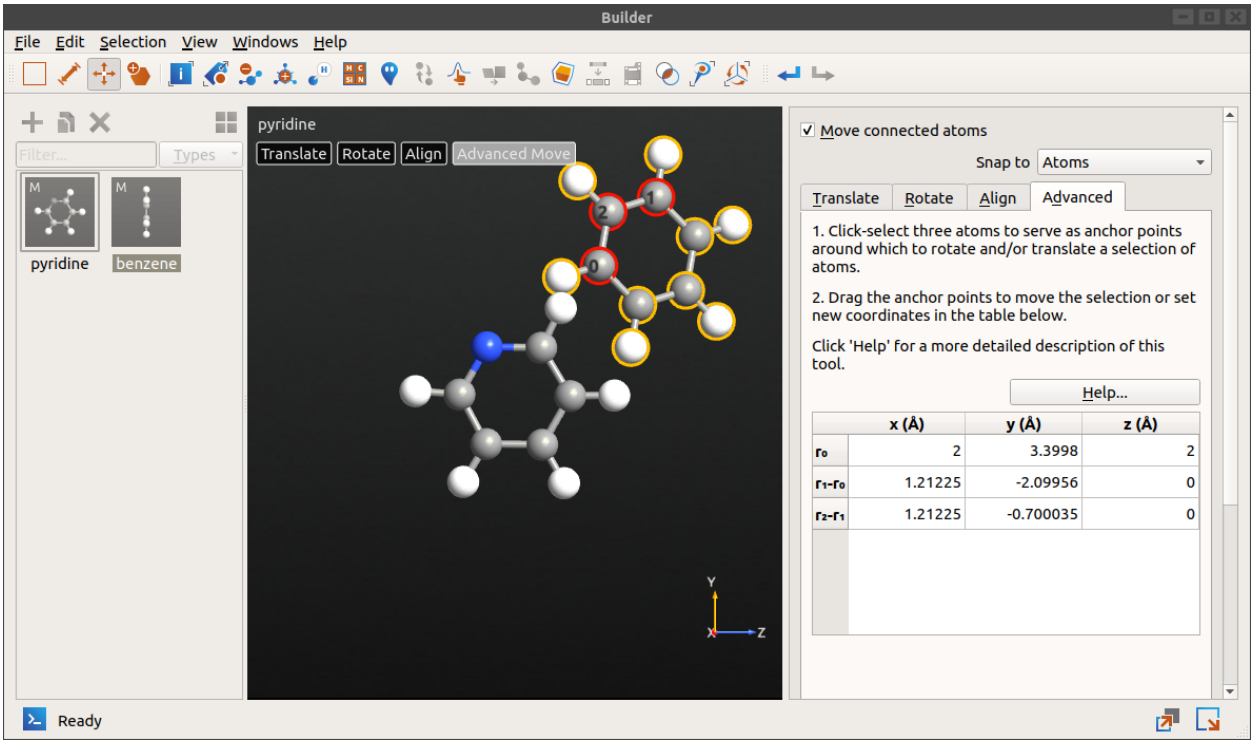


Fig. 2.9: The three anchor atoms on the pyridine molecule are indicated by an index and red markers.

Snap and fuse

If *Snap* is checked when you translate by mouse, the anchor atom will snap to the positions of other atoms, if you come close enough to them. This can be used to place the anchor atom exactly at the position of another atom, which does not belong to the selection group.

Important: The configuration obtained just after snapping is not valid, which will be indicated by the label showing that there are overlapping atoms. There may be more than one atom pair overlapping as a result of the translation. Click *Fuse* to delete the overlapping atoms not belonging to the selection group.

The *snap* option defines the translation vector that is applied to the entire selection group in all 3 directions; **X**, **Y**, and **Z**. Thus, using the *Snap* option may bring the selection out of the screen plane if the snap atom does not have the same **Y** coordinate as the anchor atom (in the example case of translation in the **XZ** plane).

Tip: A very powerful way to use snapping is to add alignment atoms at the positions where you want to place an atom or molecule. This alignment atom could be in a high-symmetry position, e.g. an fcc site a certain distance above a crystal surface. The alignment atom itself is sacrificial, and will be removed by the *Fuse* operation.

Note: It is common that there are **atoms which are close, but not exactly overlapping**, after a move operation. In this case *Fuse* cannot be used, but instead the plugin *Selection Tools* → *Close Neighbors* is useful to find and delete atoms which are close to each other. Make a habit of **checking the structure** with this tool after a complex operation, e.g. with the Move Tool, or after mirroring or rotating structures.

Advanced Move

If you select **two anchor atoms** and start moving either one with the mouse, the selection group will rotate, using the **first** anchor atom as the **rotation center** and the **vector** between the two anchor atoms as the **rotation axis**. During the rotation, you can **snap** to any other atom for more exact control.

Alternatively, you can **type in** the direction vector between the two anchor atoms $r_1 r_0$ in the widget, to **realign** the structure with regard to the cartesian directions. Only the direction of this vector matters, not its magnitude. For example, if you want the group to be realigned such that the vector between the two anchor atoms becomes parallel to the Z-direction, enter $r_1 r_0 = (0, 0, 1)$.

Finally, if you select **three anchor atoms**, you can use the mouse to control the dihedral angle between the plane spanned by two of the anchor atoms and the current mouse position. *Snap* works as above, as well as editing directly in the widget.

To **align** the group such that the three anchor atoms lie in the **XY**-plane, enter $r_1 r_0 = (0, 1, 0)$ and $r_2 r_1 = (1, 0, 0)$. The cross product of these two vectors is $(0, 0, 1)$, which is the **XY**-plane normal.

2.4.5 Overview of built-in Builder plugins

The Builder contains a lot of tools. Each tool allows one to do an isolated operation, for instance, repeating a periodic structure, rotating a selected atom around a specified axis, passivating selected atoms with hydrogen, and many other operations on the atomic structure. The true power of the Builder comes from the possibility of combining these operations in numerous ways to build complex atomic structures.

In the following, we give a quick overview of the basic plugins shipped with the QuantumATK package. This overview provides a broad picture of the functionalities of the Builder. The plugins are designed to be simple to use, and many of them have instructions built into the QuantumATK interface. You may also take advantage of the tooltips to find out about a particular plugin parameter.

Builder Toolbar plugins

Shape select uses the mouse to paint a shape to select atoms. Hold the mouse over the toolbar at the icon to switch between the three shape select modes: rectangle, circle and lasso.

Measure tool measures distances, angles and dihedrals between atoms.

Move tool manipulates an atomic structure by moving and/or rotating its fragments (groups of selected atoms) in a controllable way. See also the section *Using the Move Tool*.

Molecular builder builds a molecule from the library of chemical elements and fragments.

Information tool displays information about an active configuration.

Tags allow for a quick selection of atoms labeled with user-defined tags.

Delete atoms deletes selected atoms of an active configuration.

Centroid places an atom in the geometrical center of the group of selected atoms. This is especially handy for creating alignment points to be used with the *Move Tool*. Note that you should select at least two atoms for the Centroid operation to do anything.

Passivate adds hydrogen or pseudo-hydrogen atoms to passivate dangling bonds for selected atoms, or the whole structure if no selection has been made prior to passivation. This operation preserves the periodicity of the original unit cell structure.

Periodic table allows you to replace the selected atoms of a certain chemical element with atoms of some other chemical element from the Periodic Table, which is displayed when you click on the icon.

Ghost converts selected atoms to ghost atoms, or to regular atoms if the selected atoms were already ghost atoms.

Swap positions swaps positions of two selected atoms.

Rattle randomly displaces each of the selected atoms in the atomic structure, with a maximal deviation of 0.125 Å from the original atom position in any direction. If no atoms are selected prior to the Rattle operation, all the atoms of the atomic structure will be randomly displaced.

Niggli form reduces the unit cell and related atom positions to the Niggli form.

Convert to molecule converts the unit cell atomic structure to an isolated molecule-like structure, removing periodic boundary conditions of the unit cell. For a device structure, this operation removes the electrodes, converting the central region of the device to the corresponding non-periodic structure.

Convert to bulk converts an isolated molecule or any non-periodic atomic structure to a periodic unit cell structure, e.g., to describe the molecule in a periodically-repeated cell. For a device structure, this operation removes the electrodes, converting the central region of the device to the corresponding periodic structure.

Split device splits a device, creating separate stash items for the electrodes and the central region, and imposing periodic boundary conditions on each of the item structures.

Extract image extracts the active image from a Nudged Elastic Band (NEB) configuration, and then adds it to the stash as a new configuration.

Overlapping atoms tool checks for overlapping atoms in an active configuration.

POVRay Generator generates a POV-Ray processed image.

Camera controls the view plane of the currently active 3D window.

Builder Panelbar plugins

Builders

Basic plugins for converting or combining atomic structures into new complex structures.

AFM Cantilever Tip builds an AFM-Tip with a cuboid shaped block. This operation cuts the tip area, using a selected material on the stash by controlling the size and the orientation.

Amorphous Prebuilder creates an amorphous structure with a given density, according to a sum formula.

Icosahedron Builder creates an icosahedron crystal.

Interface creates an interface between two bulk structures, e.g., SiO_2 and Si, to represent the oxide on a Si surface. This operation automatically searches for an interface configuration with minimal average strain by doing various rotation and repetition operations of the two input bulk structures, using the coincidence lattice method.

Nanowire builds a nanowire with a given radius and surface energies in the main crystallographic directions, using the Wulff construction method.

Nudged Elastic Band creates a NEB path between two end configurations, using one of several different methods. The created path can then be manipulated in the tool to remove or insert images, or further refine the path between the two images.

Substitutional Alloy creates an alloy structure by replacing the selected atoms in the stash item structure with another element, by specifying a percentage (concentration) of the substitutional atoms in the alloy. The tool is also able to create vacancy sites with a certain concentration.

Wulff Constructor creates an atomic cluster with a given radius and surface energies in the main crystallographic directions, using the Wulff construction method.

Generic Alloy creates a random alloy by combining different, but similar, unit cells in a larger supercell. The percentage of each configuration in the alloy can be specified in the plugin as well as the size of the alloy that can be controlled by changing the number of repetitions in each direction. The tool has an option of choosing the algorithm, Normal distribution or Fixed fraction, for generating the alloy structure.

Packmol creates an amorphous configuration containing the molecules (or any non-periodic atomic structures) listed in the plugin table.

Polycrystal (Voronoi) generates the polycrystal from the crystal structures taken from the stash. It creates the polycrystal by controlling the grid points and a crystal type.

Surface (Cleave) cleaves a bulk structure to create a surface, device electrode, or a slab. This tool has a variety of options when creating the surface cell.

Bulk Tools

These plugins primarily operate on the unit cell of a periodic atomic structure.

Crystal Symmetry Info detects the symmetry of the atomic structure if one clicks on the *Detect* button. The tool gives information on the space group of the structure in Hermann-Mauguin, Hall and Schoenflies conventions. If one clicks on the *More information* button, more information can then be obtained about the symmetry properties of the atomic structure.

Fit Cell attempts to fit a cell to the structure by looking for periodic patterns.

Merge Cells merges two periodic structures in a simpler way than the Interface pluging in the Builders does. That can be of particular interest when merging 1D or 2D structures, e.g., merging two nanotubes to create a 1D junction.

Repeat repeats a periodic structure in A, B, and C.

Stretch Cell allows you to stretch the unit cell. One can then choose whether the selected atom positions should be either moved rigidly or stretched in the unit cell. This tool can be used to strain a structure, or make a 2D junction (e.g., bilayer graphene) with partial overlap.

Supercell applies a 3x3 matrix operation to a unit cell to make a supercell structure. The atoms are then repeated as needed to fill the supercell. This operation is of particular interest for converting the primitive cell to a conventional cell, or a hexagonal to an orthorhombic cell.

Swap Axes swaps the unit cell axes A, B and C, and/or the coordinate system axes X, Y, and Z. In most cases, this operation should be applied twice to end up with a right-handed cell, not a left-handed one.

Wrap wraps the atoms (which are outside of the unit cell) back into the cell. This option can be of interest if some atoms appear outside of the unit cell during atomic structure operations.

Brillouin Zone Viewer shows the Brillouin zone and reciprocal lattice vectors of the crystal in the Cartesian coordinate system.

Lattice Parameters allows one to change the lattice type as well as the actual lattice vectors and related lattice parameters of the crystal structure.

Coordinate Tools

These plugins primarily operate on the atom coordinates.

Center puts the geometrical center of the unit cell atomic structure (considered as a group of atoms) in the geometrical center of the corresponding unit cell, or with respect to the origin in the case of a molecule or non-periodic atomic structure such as an atomic cluster.

Custom Passivator passivates dangling bonds of undercoordinated atoms with hydrogen or pseudo-hydrogen atoms. This passivation tool has an option to manually set the bond distance between the hydrogen and passivated atom.

Mirror mirrors a group of selected atoms with respect to a given mirror plane. Optionally, one may copy the atoms from their original to their mirrored positions to have both the mirrored and original atoms in the structure.

Quick Optimizer runs a geometry optimization of the atomic structure using an ATKClassical potential, provided that there is a classical potential available in the classical potential database for each of the chemical elements in the structure.

Rotate rotates a group of selected atoms with respect to a given axis. Optionally, one may copy the atoms from their original to their rotated positions to have both the rotated and original atoms in the structure. Note that the *Move Tool* provides an alternative way to rotate the group of atoms with the mouse.

Sort sorts the atomic coordinates. This operation can also be done in the Coordinate List.

Transform by Expression applies a coordinate transformation given by a mathematical expression to the coordinates of selected atoms, e.g., $x \rightarrow x + \sin(y)$. This plugin supports all mathematical functions available in the Python math module, and interprets the variables x,y,z as atomic coordinates, and a,b,c as lattice vectors of periodic systems.

Translate moves a group of selected atoms by a given distance. Optionally, one may copy the translated atoms to their original (prior to translation) positions to have both the translated and original atoms in the structure. Note that the *Move Tool* provides an alternative way to move the group of selected atoms with the mouse.

Coordinate List allows one to directly edit the coordinates of atoms in a sortable coordinate list. You should right-click on the table to use the context menu to insert or delete atoms. Selecting rows in the table automatically selects the corresponding atoms.

Z-matrix is a direct way of editing the Z-matrix of selected atoms.

Non-default Coordinate Tools

If needed, other plugins for building very specific atomic structures can be added to the Coordinate Tools. These plugins are not available in the Coordinate Tools by default, but they can be installed from the AddOnManager, see *Help* $\rightarrow$ *AddOn Manager*. For example, the following plugin can be installed from the TubeWrapper and FunWithGraphene packages in the AddOnManager.

Buckler applies a buckling by modulating the x-coordinate of an initially flat structure.

Tube Wrapper wraps a periodic system onto a circular arc, or a tube if the angle is 360 degrees. Subsequently, the original B(Y) axis will be mapped onto the arc, while the tube axis will lie along C(Z). If helical rise is non-zero, a helix is then created such that a difference in the Z coordinate between the beginning and end is the specified helical rise.

Twister twists the structure by a given angle around the Z axis. The non-twisting zone defines a distance from the edges of the system in the Z direction, which are left unmodified to maintain periodicity.

Device Tools

These plugins create or modify a device geometry.

Central Region Size allows for changing the size of the central region after a device structure has already been created in the Builder, using the *Device from bulk* or *Device from surface* plugins. Using this tool, one may also add or remove atomic layers in the central region, enlarging the right and/or left electrode extensions.

Electrode Size allows for changing the size of the electrodes after a device structure has already been created in the Builder, using the *Device from bulk* or *Device from surface* plugins. This is similar to the Central Region Size plugin.

Device from Bulk is the main tool to automatically create a device in the Builder. The tool converts the bulk configuration item in the stash to the device central region structure with no periodicity in the C-transport direction unlike the original bulk structure. It then attaches the left and right electrodes of the device to its central region. The electrodes are constructed from the edge atomic layers (electrode extensions) of the central region. The electrode length can be set by default or manually in the plugin.

Surface from Bulk is the tool to automatically create the surface structure by converting the bulk configuration structure in the stash to the central region of a one-probe device. The tool then attaches the left electrode to the device central region. The electrode is constructed from the left edge atomic layers (electrode extension) of the central region. The electrode length can be set by default or manually in the plugin. To create a real surface, some of the atoms have to be manually deleted from the one-probe device central region (or original bulk structure) to create a vacuum layer above the surface of interest.

Graphics

Properties opens a window with the many graphical options available in QuantumATK.

Miscellaneous

Various plugins that do not fit in any other category.

Doping adds doping charge to the group of currently selected atoms. One may also undo doping by selecting a row in the table. This tool is of particular interest to modeling of doping effect in a atomistic semiconductor device without explicitly introducing dopant atoms.

Spatial Regions allows one to add and remove metallic and dielectric regions in the device structure. This tool is of particular interest to building a semiconductor device with the dielectric spacer between the gate electrode and channel region of the semiconductor device. Note that the metallic and regions are given by the continuum model in a TCAD fashion, unlike the channel region that is treated at the atomistic scale.

Select Tools

Plugins for advanced selection of atoms.

By Bond Length selects atoms by a range of separation distance between the atoms.

By Element selects atoms by a type of chemical element.

By Expression selects atoms using a logical expression, e.g., setting e=H and x<10 will select all the hydrogen atoms with X coordinate less than 10 Å.

Close Neighbors helps to identify overlapping atoms. It is a very useful tool to detect those atoms that are on top of each other, and then delete duplicate atoms. This must be done before sending the structure from the Builder to the Script Generator to avoid any numerical issues upon running the QuantumATK calculations.

Tags allow one to add, edit or remove tags on an atom or a group of selected atoms. One may also select atoms by this plugin, but the *Tag Select* button in the toolbar is a faster option for atom selection with tags.

Custom builder plugins

Under the Stash menu *Add* → *From Plugin*, you will find a list of template builders, which are plugins designed to build a particular structure based on a simple or complex set of rules, and add a new item to the Stash.

In the default installation of the QuantumATK, you will find the following plugins for adding new stash items.

Nanotube builds a nanotube with a given chirality, such as boron-nitride and carbon nanotubes as well as multiwall nanotubes with up to 3 walls.

Crystal Builder builds a crystal by selecting the space group and lattice parameters of the crystal.

Databases chooses molecules, crystals, fullerenes or monomers from an extensive list of predefined structures.

Nanoribbon builds one-dimensional ribbons cut out from a nanosheet. Optionally, one may passivate the ribbon edges with hydrogen atoms.

Nanosheet builds infinite two-dimensional sheets, such as graphene monolayers and similar 2D hexagonal materials, aligning the C-axis with a given chiral vector. Note that the C-axis is the default direction for QuantumATK transport calculations.

Magnetic Tunnel Junction (FeMgO-style) builds a magnetic tunnel junction (MTJ) such as the Fe|MgO|Fe MTJ, which is a two-probe device with a layer of insulator or semiconductor material placed between two ferromagnetic electrodes.

Supported file formats for import and export

In addition to QuantumATK Python and HDF5 files, the following file formats are supported in the QuantumATK package for import and export of atomic geometries.

•XYZ

•CAR

•CASTEP (.cell and .geom)

•CIF

•FHI-aims

•OpenBabel (.acr, .alc, .arc, .bgf, .box, .bs, .c3d1, .c3d2, .caccrt, .can, .ccc, .cdx, .car, .cssr, .dmol, .fract, .fs, .ins, .sy2)

•Quantum ESPRESSO

•ASE

•MBN Explorer

•VASP (POSCAR and CONTCAR, and their gz, gzip and zip versions)

Support for additional file formats, not listed above, becomes regularly available in the form of plugins that can be installed in the *AddOn Manager*.

Third-party plugins

When you add a new plugin to QuantumATK, either from add-ons that you have developed yourself, downloaded via the *AddOn Manager*, or obtained directly via third-party (or QuantumATK) as a zip-file, they are always seamlessly integrated in the proper place where they belong. It means that if you add your own import filter, it will then automatically appear in the dropdown list of file filters in the *Add → From Files* dialog.

2.4.6 Managing plugins using the AddOn Manager

Each functionality in the Builder is a **plugin**. A plugin is a small or large piece of Python code that defines its functionality.

An **add-on** is a collection of plugins; some contain just one, to define a single button/filter etc, while others contain a whole set of plugins which are related to each other.

Using the **AddOn Manager** (*Help→AddOn Manager*) you can automatically update installed add-ons, or install new ones to extend the functionality of QuantumATK, either by downloading them from the QuantumATK add-on server, or installing them from `.zip` files or a folder by clicking **Install zip/folder**. Normally you can just update the add-ons by clicking the Update or Update All buttons in the AddOn Manager in QuantumATK. If, however, you are behind a firewall that prevents QuantumATK from accessing the add-on server directly, you can instead download the add-on and install it manually by clicking the *Install zip/folder* button highlighted in red.

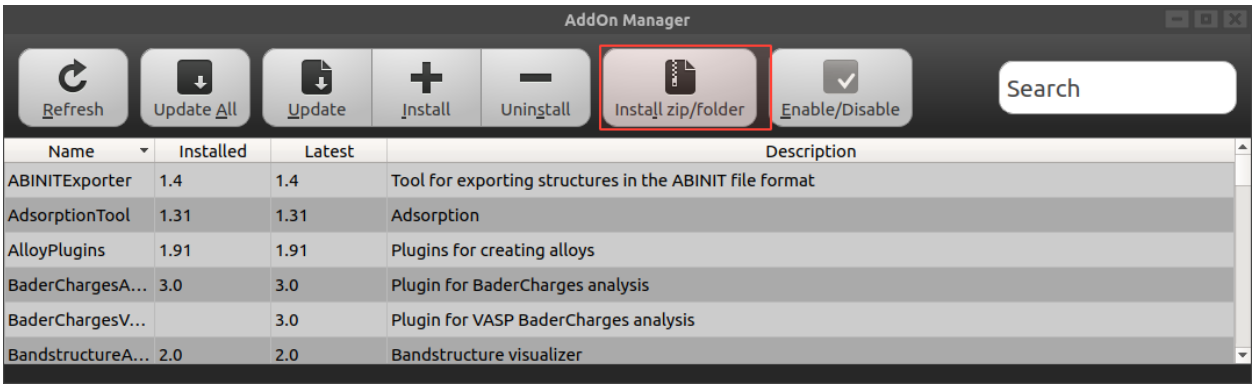


Fig. 2.10: The add-on can be manually installed by clicking the button *Install zip/folder* in the AddOn Manager in QuantumATK. Restart QuantumATK to activate the new add-on.

Tip: Make it a habit to check the AddOn Manager now and then, to see if there are any updates available.

Add-ons are installed in `quantumatk/lib/site-packages/AddOns` on Windows and `quantumatk/lib/python2.7/site-packages/AddOns` on Linux, relative to the QuantumATK installation directory. There you can inspect the source code of almost all plugins provided with QuantumATK, and use them for inspiration to create your own!

2.5 Workflows Manual

2.5.1 Introduction to the Workflow Builder

Downloads & Links

`bandstructure.hdf5` `bandstructure_rerun.hdf5` `multiple_calculators.hdf5` `multiple_configurations.hdf5` `block_of_blocks.hdf5`

The *Workflow Builder* is one of the core features of the *QuantumATK NanoLab*. It enables you to easily set up a calculation as a *workflow* of different calculation steps. These calculation steps can form one or more branches of execution, which can then be run though the *Jobs* tool.

With the Workflow Builder you will be able to create iteration loops and iterate over integers, floats, `PhysicalQuantity` values, and atomic configurations. You will be able to group multiple calculation steps and save it for later use, as well as save and share the entire workflow with your colleagues.

In the next sections you will learn more about the Workflow Builder and its user interface through a set of simple examples. You will learn how to create workflows with one or more atomic configurations, you will learn how to split a workflow into more branches of execution, as well as how to run workflows though the *Jobs* tool and how to reuse results from the previous calculations.

In the *Downloads & Links* sidebar you will find a list of all the workflows that are created in the following examples.

Note: Users familiar with the *Script Generator* will find the Workflow Builder to be very similar. The Workflow Builder replaces the Script Generator that was available in the previous versions of the *QuantumATK NanoLab*.

The Workflow Builder supports all the features of the *Script Generator*, and it also introduces new ones, such as support for multiple branches of execution or support for looping over custom `PhysicalQuantity` values or atomic configurations.

Table of Contents

- *What is the Workflow Builder?*
- *The Workflow Builder overview*
- *What is a workflow block and how to edit it*
- *Creating a simple workflow*
- *Re-running a workflow*
- *Workflow with multiple branches*
- *Block of blocks*
- *Common operations in the Workflow Builder*

What is the Workflow Builder?

The *Workflow Builder* is a powerful feature of *QuantumATK NanoLab* where you can easily set up and run calculations. It is placed in the side toolbar on the left in the main *QuantumATK NanoLab* window.

In this tool, you can create new objects called *Workflows*, which can be saved, shared, re-opened and edited within the Workflow Builder Valid workflow can be converted directly to one or multiple scripts that can be run through the *Jobs* tool.

Workflows are *automatically* saved as `.hdf5` files and are visible in both the Workflow Builder and the *Data* tool. To *share a workflow*, simply share its `.hdf5` file.

A workflow consists of one or more components called *Blocks*. A block represents one or more calculation steps and it exposes calculation parameters. Calculation parameters can be edited from the Workflow Builder. Creating a workflow is as easy as assembling blocks from a catalog of built-in blocks. To read more about blocks, jump to the section *What is a workflow block and how to edit it*.

A workflow can contain sets of calculations. This is because a workflow block can contain lists of objects and their parameter values. A workflow can then iterate over the given lists to perform the same calculation with different inputs. To increase the calculation throughput, you also can split a workflow into more execution branches.

You can export a workflow as a single Python script or as multiple Python scripts. Each script would represent one branch of execution in the workflow. Having multiple scripts enables you to run a workflow in parallel instead of sequentially, which can significantly improve the calculation speed.

The Workflow Builder overview

This guide assumes you are familiar with the *Data* tool and the *Builder* tool in the *QuantumATK NanoLab*. The *Data* tool is used for managing files and analyzing results. The *Builder* tool is used for creating, manipulating and inspecting atomic configurations.

Start by opening the *QuantumATK NanoLab* and creating a *new project*. After opening the program, the first window you will see is a *Data* tool window.

To select the *Workflow Builder*, click the icon located in the side toolbar on the left. A new *Workflow Builder* window will appear with three main areas - Workflow Stash, Workflow Area and Workflow Catalogs.

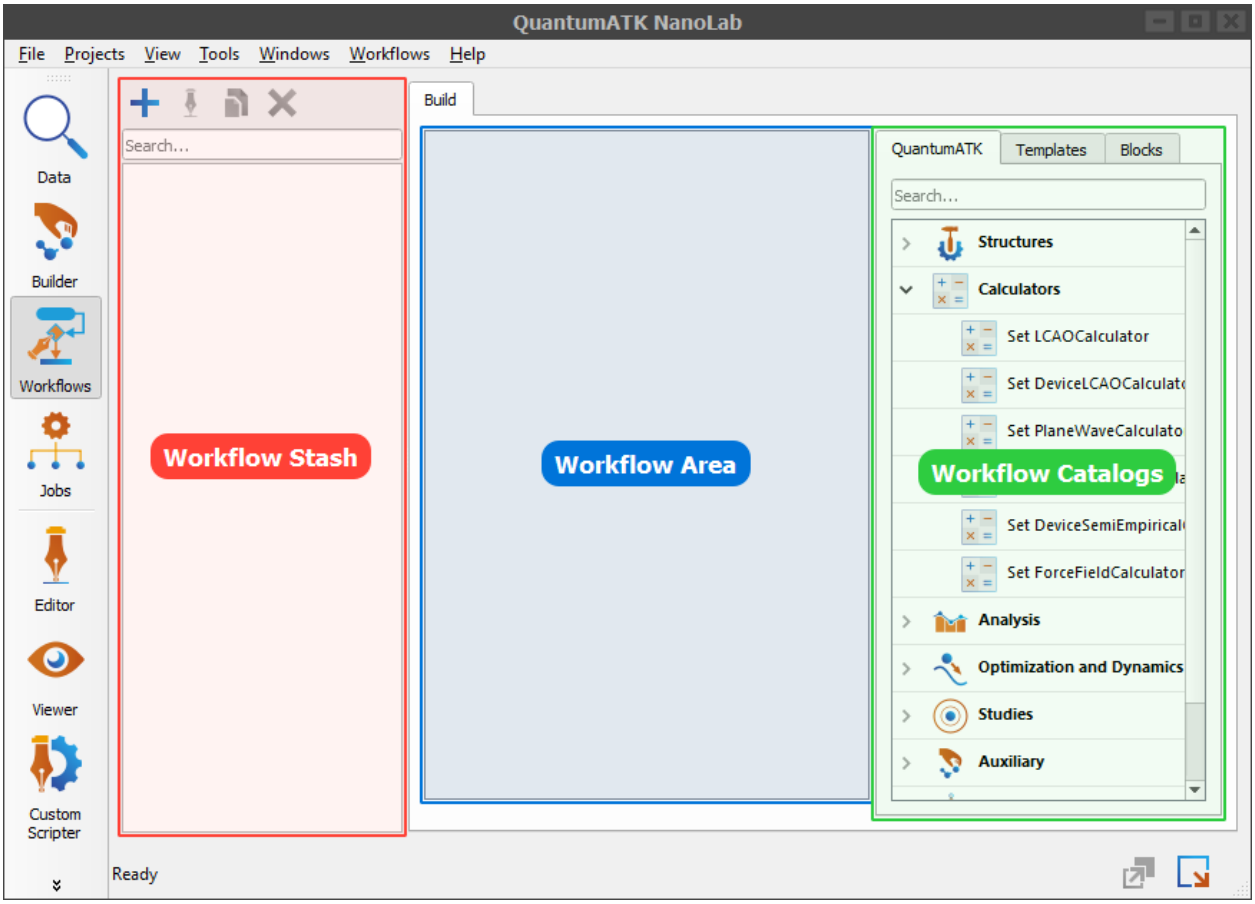
The *Workflow Stash* is on the left highlighted in red. Here you will find all the existing project workflows. From here you can also create new workflows or delete, rename or duplicate the existing ones. The Workflow Stash is currently empty.

The *Workflow Area* is in the center under the *Build* tab, highlighted in blue. It shows the contents of the currently selected workflow in the Workflow Stash.

Note: There are three levels of the log verbosity - *Full*, *Minimal* and *Silent*. Use the *Full* verbosity for troubleshooting, *Minimal* for inspecting only the most significant steps and *Silent* when you do not need any calculation feedback.

The *Workflow Catalogs* of pre-made workflow blocks is on the right highlighted in green. These are the building blocks of a workflow. In the catalog you will find three tabs – *QuantumATK*, *Templates* and *Blocks*. The blocks from the catalogs are available across all projects.

The *QuantumATK* tab contains a list of built-in blocks divided into different categories, such as *Structures*, *Calculators* or *Analysis*. This list of blocks cannot be edited.


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Custom Scripter
Search...
Workflow Stash
Build
Workflow Area
QuantumATK
Templates
Blocks
Search...
Structures
Calculators
Set LCAOCalculator
Set DeviceLCAOCalculat
Set PlaneWaveCalculato
Workflow Catalogs
Set DeviceSemiEmpirical
Set ForceFieldCalculator
Analysis
Optimization and Dynamics
Studies
Auxiliary
Ready

Note: If you are familiar with the Script Generator of previous *QuantumATK NanoLab* releases, you will recognize most of the blocks under the *QuantumATK* tab.

Each QuantumATK block can be a container of objects or a single object from the *QuantumATK Engine*, such as a `BulkConfiguration`, `LCAOCalculator` or a `Bandstructure`. There are also special blocks, such as the *Iterator* block or the *Custom* block. To read more about workflow blocks, jump to the section *What is a workflow block and how to edit it*.

The *Templates* tab contains built-in single blocks and *groups* of blocks chained together to calculate a specific property such as a *Constant Density Melt Quench (DFT)* available in the *Molecular Dynamics* sub-category. Among the *Templates* you will also find single blocks with specific parameters values, such as a *Metal (Low)* block which is an `LCAOCalculator` block with a low k-point density.

Note: If you are familiar with the Script Generator of previous *QuantumATK NanoLab* releases, you should note that the *Templates* play a similar role in the *Workflow Builder* as well. The difference is that Templates in the *Workflow Builder* cannot be modified and its list also includes single blocks.

The *Blocks* tab is a place where you can save your custom blocks across projects. Workflow Blocks can be grouped together in a container block called the *Block of Blocks*. Since Block of Blocks can can be saved, this allows for re-use of even very complex calculation setups. To learn how to save blocks under the *Blocks* tab, jump to the section *How to save a block from a workflow*.

What is a workflow block and how to edit it

A workflow block is a basic building unit of a workflow. It can be though of as one calculation step in a workflow. A block typically represents a single object from the *QuantumATK Engine*. It can require specific input *QuantumATK Engine* objects and can provide specific output *QuantumATK Engine* objects. Blocks are added one after the other to form a branch of execution in a workflow. If a block has an invalid input, it will show an error message.

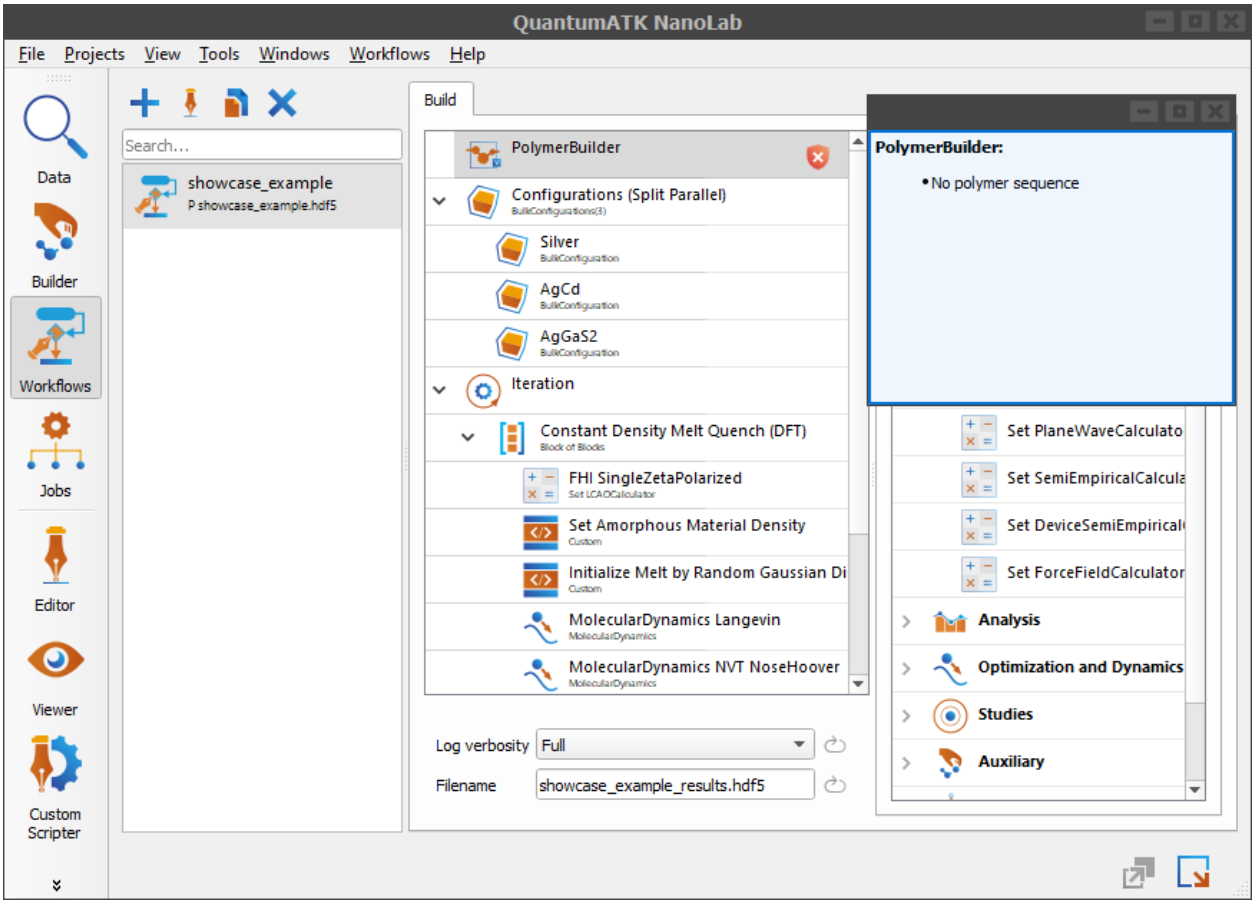
To edit a block in a workflow just double click it. All block editors indicate non-default values and allow for direct reset to default. Editors contain default settings of the underlying *QuantumATK Engine* object and can be changed by the user.

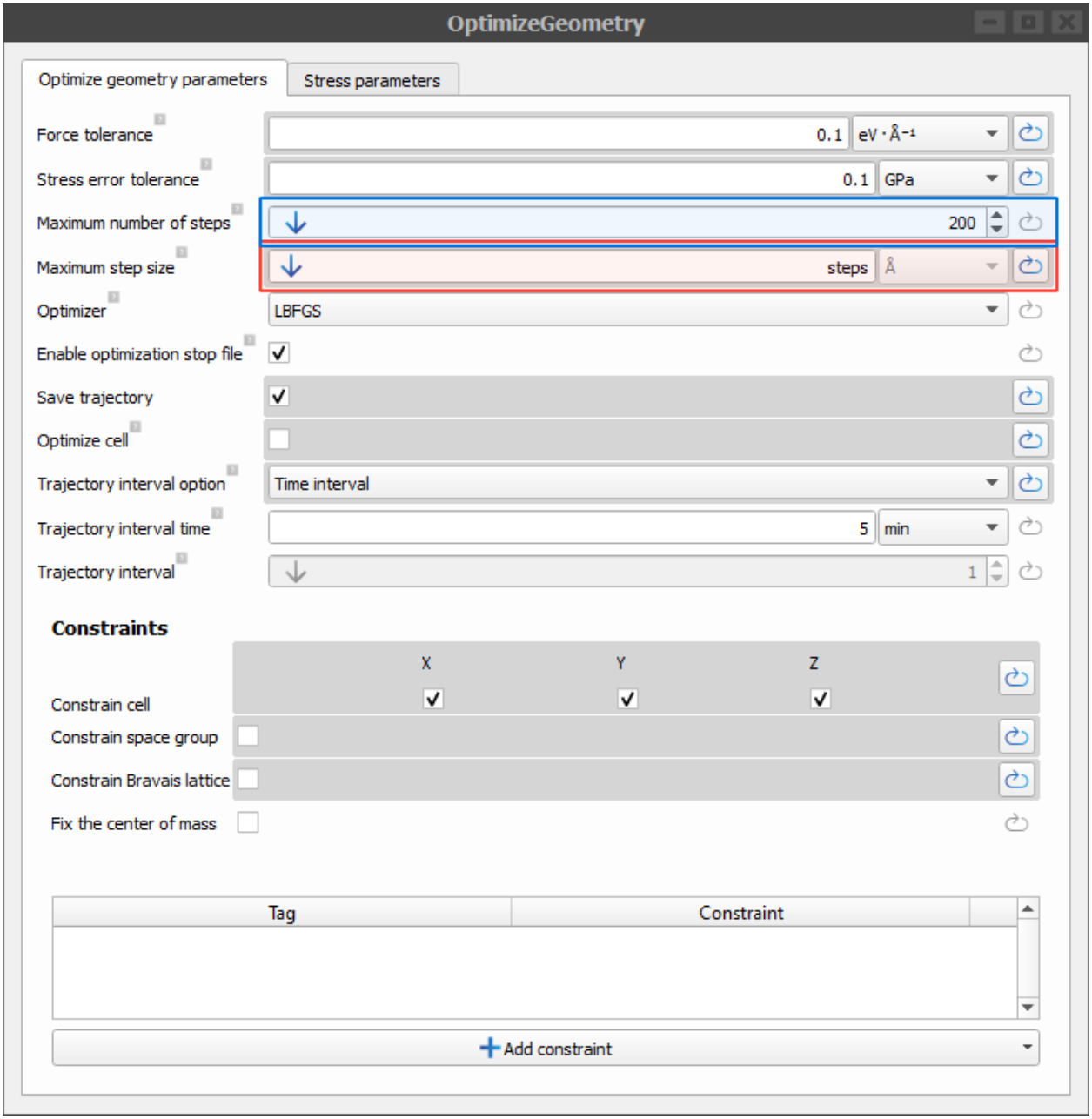
If you have made a change to some of the default settings and would like to revert to a default value instead, click the reset button near the value you were editing, like the *Maximum step size* parameter highlighted with red in the *Optimize Geometry* block (see the figure below).

You will know the value is default when the reset button becomes disabled (grayed out), like the *Maximum number of steps* parameter highlighted with blue.

Beside blocks which represent a single *QuantumATK Engine* object, there are also special blocks, such as *Configurations (Split Parallel)* and *Split Parallel* blocks. These blocks implement simple parallelization. They can contain a list of sub-blocks and then each sub-block in the list would belong to a separate branch of execution. The resulting workflow script will loop over each sub-block, hence producing simple parallelization mechanism akin to loop unrolling.

The simplest example is the Configurations (Split Parallel) block, that enables one to run the same (parallel) calculation for multiple atomic configurations.


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Custom Scripter
Search...
showcase_example
P showcase_example.hdf5
Build
PolymerBuilder
Configurations (Split Parallel)
Silver
AgCd
AgGaS2
Iteration
Constant Density Melt Quench (DFT)
FHI SingleZetaPolarized
Set Amorphous Material Density
Initialize Melt by Random Gaussian Di
MolecularDynamics Langevin
MolecularDynamics NVT NoseHoover
PolymerBuilder:
No polymer sequence
Set PlaneWaveCalculato
Set SemiEmpiricalCalcula
Set DeviceSemiEmpirical
Set ForceFieldCalculator
Analysis
Optimization and Dynamics
Studies
Auxiliary
Log verbosity
Full
Filename
showcase_example_results.hdf5


OptimizeGeometry
Optimize geometry parameters
Stress parameters
Force tolerance
0.1
eV · Å⁻¹
Stress error tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
steps
Å
Optimizer
LBFGS
Enable optimization stop file
Save trajectory
Optimize cell
Trajectory interval option
Time interval
Trajectory interval time
5
min
Trajectory interval
1
Constraints
X
Y
Z
Constrain cell
Constrain space group
Constrain Bravais lattice
Fix the center of mass
Tag
Constraint
Add constraint

Workflows with *Configurations (Split Parallel)* or *Split Parallel* blocks should be run in parallel as explained in the section *Workflow with multiple branches*.

Other special blocks include *Iteration*, *Block of Blocks*, *Custom* blocks as well as the *Load from file* block.

With the *Iteration* block you can specify integers, floats, `PhysicalQuantity` variables of different types, and atomic configurations. These values can then be used as inputs to some of the other blocks in the workflow. Using the *Iteration* block you can iterate over a different set of atomic configurations, basis sets, densities, etc

The *Block of Blocks* can group existing blocks in the workflow. Use this block to save a sequence of blocks specialized to calculate or analyze a set of properties, which you can later reuse in other workflows.

The *Custom* blocks are very powerful and flexible blocks. They support insertion of custom QuantumATK scripts. The script, desired inputs and the produced outputs can be defined directly in the *Workflow Builder*. The output from a custom block can also be used in subsequent blocks, thus allowing Custom blocks to serve as an interface to a complex calculation setup. The Custom Block can also be saved to the *Blocks* catalog for easy reuse.

The *Load from file* blocks supports the inclusion of already calculated results. All QuantumATK objects can be loaded and used directly within a workflow. The obvious example is the calculation of additional analyses from an already updated configuration.

Creating a simple workflow

Lets create a simple workflow which calculates a `Bandstructure` of a `BulkConfiguration` using an `LCAOCalculator`. To do this we would need to add three workflow blocks, each representing one of these *QuantumATK Engine* objects.

We will start by adding a `BulkConfiguration` from the database in the *Builder*.

Sending an atomic configuration from the Builder to the Workflow Builder

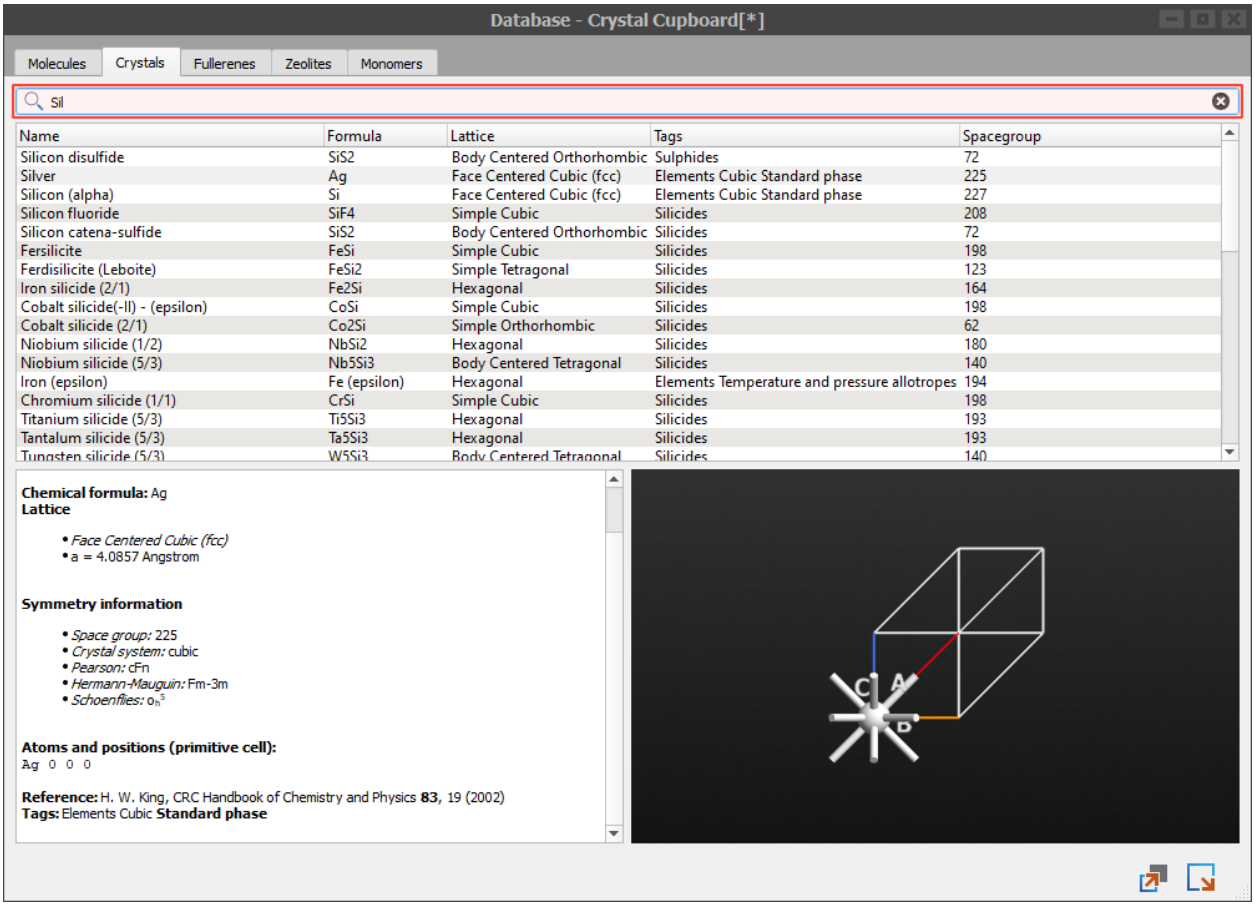
Lets load a `BulkConfiguration` from the *Builder* to the *Workflow Builder*.

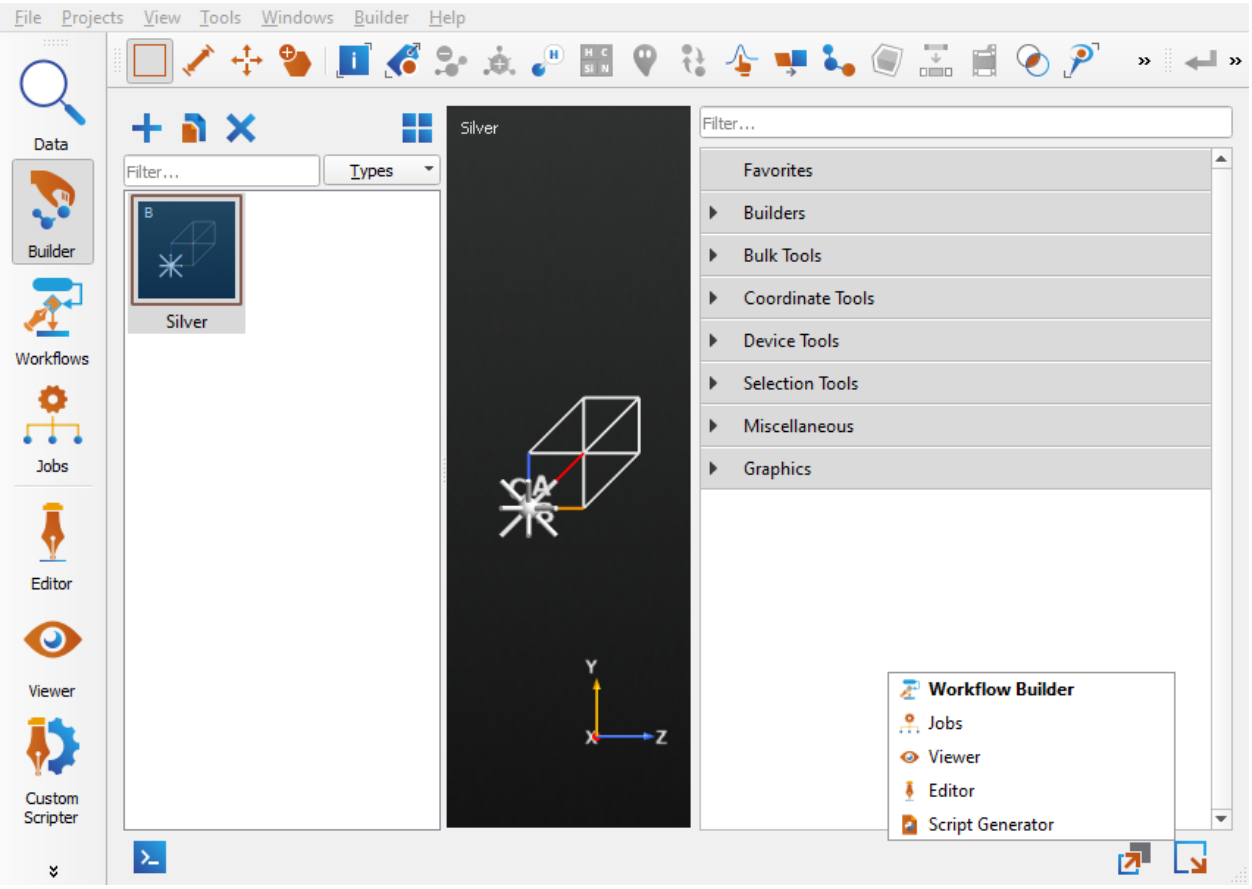
Start by opening the *Builder* by clicking the icon from the side toolbar on the left. This will open a new window.

Once we are in the *Builder* click the *Add* icon and select *From Database*. This will open a Database with a list of built-in atomic configurations we can load in the *builder stash*.

Lets load an atomic configuration called *Silver*. You will find *Silver* under the *Crystals* tab. To quickly find it type *Silver* in the *Search*. To add *Silver* to the *builder stash* just double click it. You will see the atomic configuration in the *builder stash* and its graphical representation in the *3D Viewer*.

To send the *Silver* configuration from the *Builder* to the *Workflow Builder*, click the in the lower right corner and select *Workflow Builder*.


Database - Crystal Cupboard[*]
Molecules
Crystals
Fullerenes
Zeolites
Monomers
Sil
Name | Formula | Lattice | Tags | Spacegroup
Silicon disulfide | SiS2 | Body Centered Orthorhombic | Sulphides | 72
Silver | Ag | Face Centered Cubic (fcc) | Elements Cubic Standard phase | 225
Silicon (alpha) | Si | Face Centered Cubic (fcc) | Elements Cubic Standard phase | 227
Silicon fluoride | SiF4 | Simple Cubic | Silicides | 208
Silicon catena-sulfide | SiS2 | Body Centered Orthorhombic | Silicides | 72
Fersilicite | FeSi | Simple Cubic | Silicides | 198
Ferdisilicite (Leboite) | FeSi2 | Simple Tetragonal | Silicides | 123
Iron silicide (2/1) | Fe2Si | Hexagonal | Silicides | 164
Cobalt silicide(-II) - (epsilon) | CoSi | Simple Cubic | Silicides | 198
Cobalt silicide (2/1) | Co2Si | Simple Orthorhombic | Silicides | 62
Niobium silicide (1/2) | NbSi2 | Hexagonal | Silicides | 180
Niobium silicide (5/3) | Nb5Si3 | Body Centered Tetragonal | Silicides | 140
Iron (epsilon) | Fe (epsilon) | Hexagonal | Elements Temperature and pressure allotropes | 194
Chromium silicide (1/1) | CrSi | Simple Cubic | Silicides | 198
Titanium silicide (5/3) | Ti5Si3 | Hexagonal | Silicides | 193
Tantalum silicide (5/3) | Ta5Si3 | Hexagonal | Silicides | 193
Tungsten silicide (5/3) | W5Si3 | Body Centered Tetragonal | Silicides | 140
Chemical formula: Ag
Lattice
Face Centered Cubic (fcc)
a = 4.0857 Angstrom
Symmetry information
Space group: 225
Crystal system: cubic
Pearson: cFn
Hermann-Mauguin: Fm-3m
Schoenflies: O_h^5
Atoms and positions (primitive cell):
Ag 0 0 0
Reference: H. W. King, CRC Handbook of Chemistry and Physics 83, 19 (2002)
Tags: Elements Cubic Standard phase
C
A
B


File
Projects
View
Tools
Windows
Builder
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Custom Scripter
Filter...
Types
Silver
Silver
Filter...
Favorites
Builders
Bulk Tools
Coordinate Tools
Device Tools
Selection Tools
Miscellaneous
Graphics
Workflow Builder
Jobs
Viewer
Editor
Script Generator

Selecting this option will open the *Workflow Builder* window. In the *Workflow Builder* you will notice a new workflow object called *Silver* in the *Workflow Stash* on the left. It was automatically created after sending the *Silver* configuration. The workflow contains one `BulkConfiguration` block which represents the *Silver* configuration.

Lets rename the created workflow. In the Workflow Stash, click the to change the name of the current workflow to *bandstructure*.

It is important to know that the workflows in the *Workflow Stash* are project-specific. You can still import workflows from other projects like you would import any other `.hdf5` file in the *QuantumATK NanoLab*. Jump to the section *How to share a workflow* to learn more about exporting and importing workflows.

Note: To see the file location of a workflow, right click it in the *Workflow Stash* and select *Open location*.

Currently, we have one workflow with one `BulkConfiguration` block called *Silver*. This will be our atomic configuration block. At this point we can inspect the configuration from the blocks editor.

Lets now open an editor of the configuration block in the *bandstructure* workflow. From there we can visualize and inspect the added atomic configuration. To open the editor, double click the *Silver* configuration block in the central window of the *Workflow Builder*, i.e. the Workflow Area.

The loaded atomic configuration will appear in the list of atomic configurations on the left and the 3D window will display its graphical representation. In the *Definition* list you will see a summary of its properties such as type, unique elements, tags, lattice type and regions.

To add new tags or regions to the atomic configuration, use the *Tags* or *Spatial Regions* plugins in the right window.

We have now added an atomic configuration to the workflow. The next step is to create a calculator and attach it to the loaded atomic configuration.

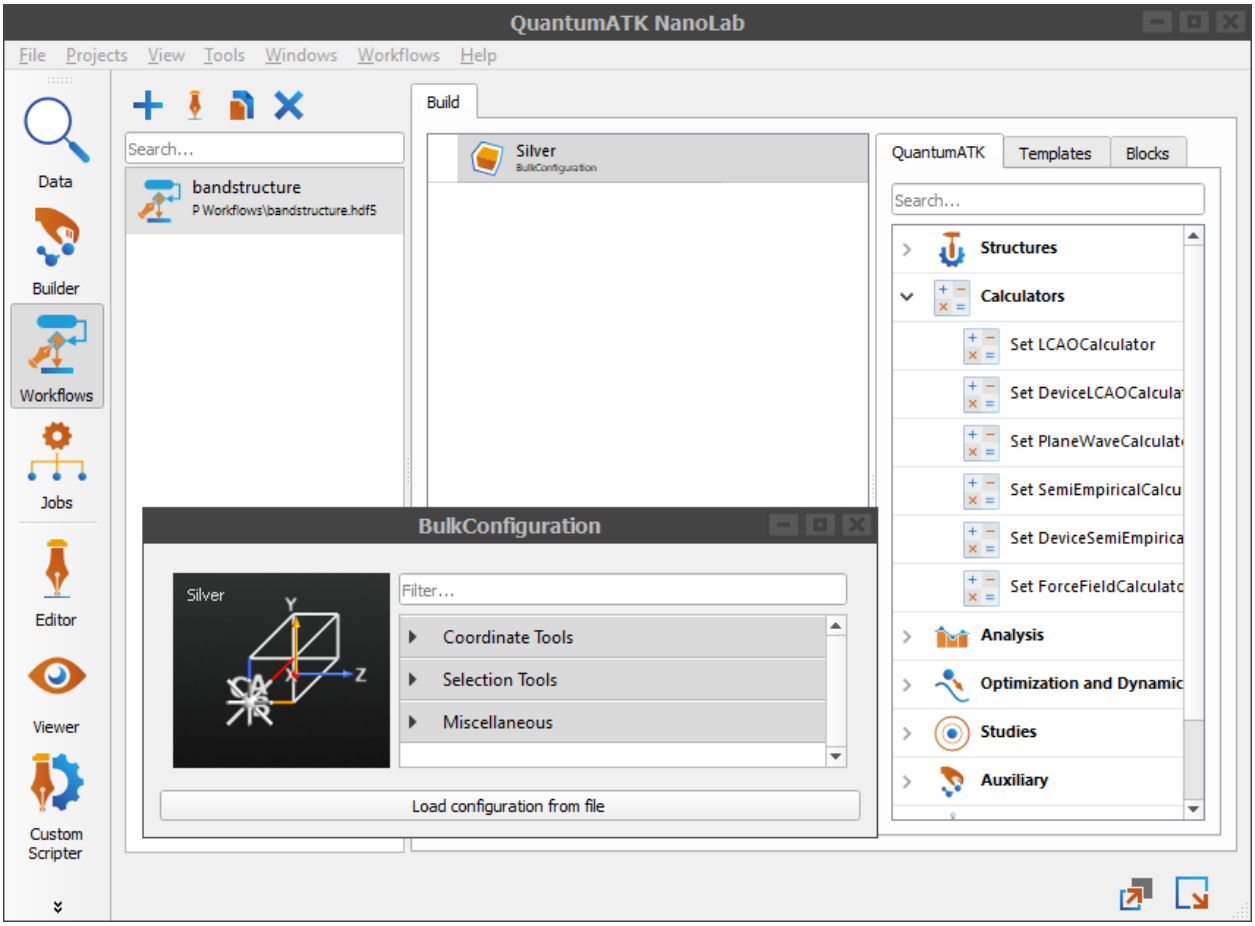
How to add a calculator

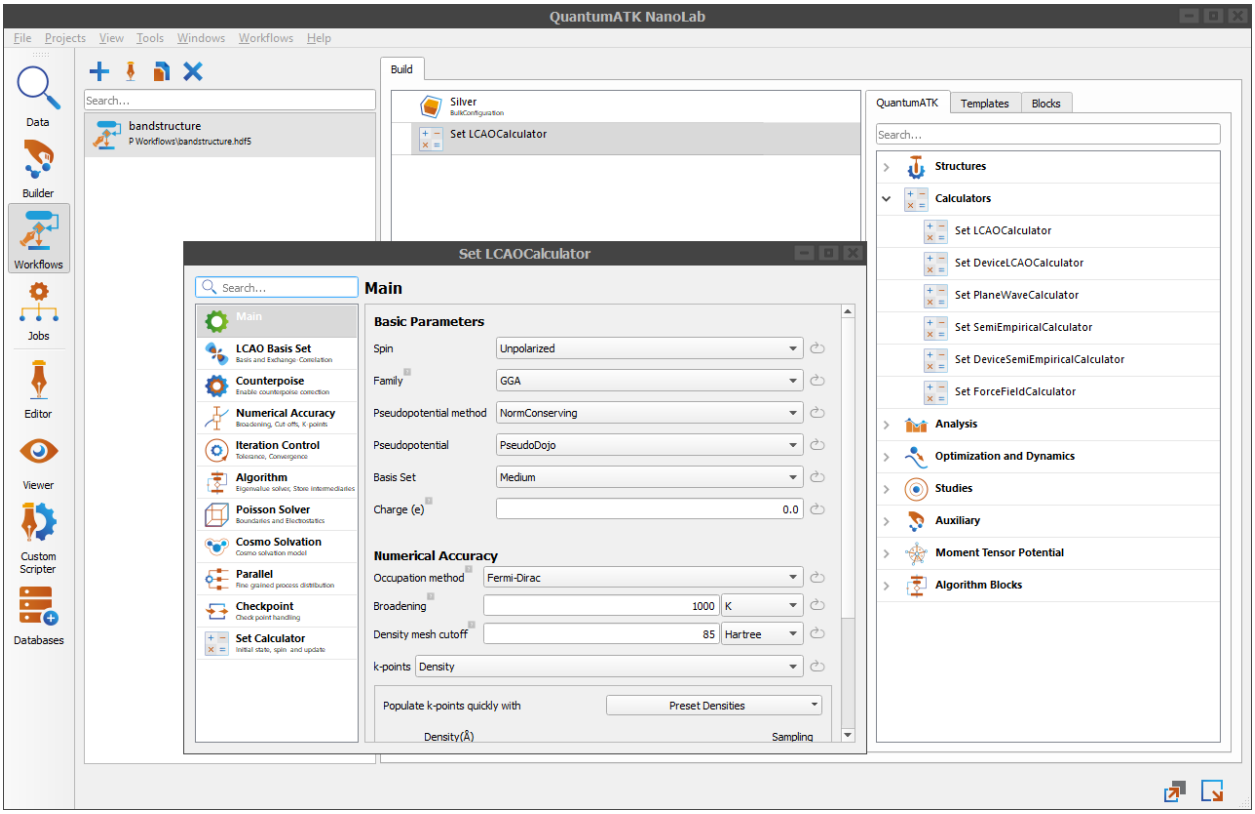
To add a calculator, go to the catalog of QuantumATK blocks, expand the *Calculators* category and choose *Set LCAOCalculator* block from the list of available calculators.

Adding the *Set LCAOCalculator* block requires an atomic configuration as an input. It outputs the same atomic configuration with an attached `LCAOCalculator`.

The calculator block uses the default parameters of the underlying `LCAOCalculator` object. Double click the block to see its default values. Here you can inspect or edit calculator properties. In this example, we will continue with the defaults.

Our workflow currently contains an atomic configuration and a calculator attached to it. The next step will be to perform an analysis on the given configuration.


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Custom Scripter
Search...
bandstructure
P Workflows\bandstructure.hdf5
Build
Silver
BulkConfiguration
QuantumATK
Templates
Blocks
Search...
Structures
Calculators
Set LCAOCalculator
Set DeviceLCAOCalcula
Set PlaneWaveCalculat
Set SemiEmpiricalCalcu
Set DeviceSemiEmpirica
Set ForceFieldCalculatc
Analysis
Optimization and Dynamic
Studies
Auxiliary
BulkConfiguration
Silver
Y
Z
Filter...
Coordinate Tools
Selection Tools
Miscellaneous
Load configuration from file



How to add an analysis object

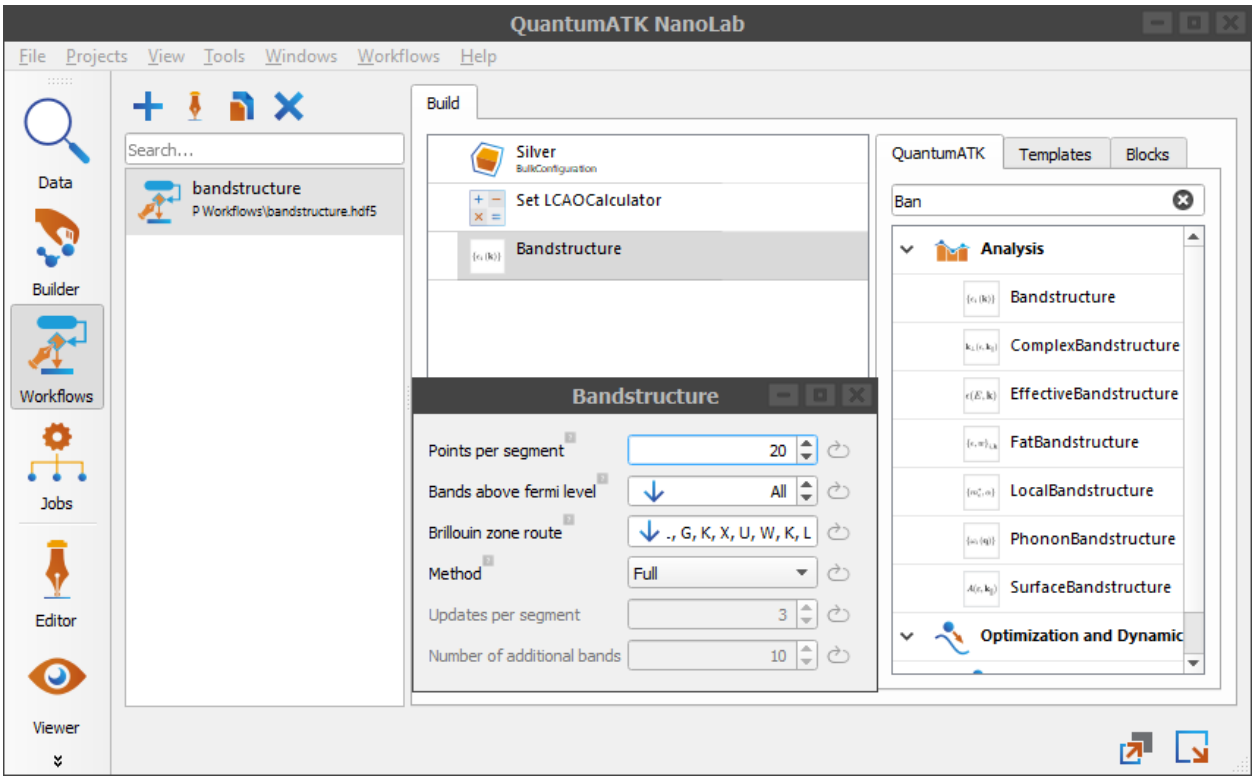
A complete list of all analysis blocks can be found in the catalog of QuantumATK blocks under the *Analysis* group.

Lets add a *Bandstructure* analysis block to the workflow.

To find a *Bandstructure* analysis block in the long list of analysis blocks, we recommend a text search. Then double click or drag and drop the *Bandstructure* block from the catalog onto the Workflow Area.

The *Bandstructure* block requires a *BulkConfiguration* with an attached calculator as an input. It outputs a *Bandstructure* analysis object. The block uses the default parameters of the underlying *Bandstructure* object. We will not edit the blocks parameters and will continue with the default values.

Since the block input is valid and it reports no errors, lets run the current workflow.



How to run a workflow

A workflow can be exported as a Python script, which can be run with the *Jobs* tool.

To execute the `bandstructure.hdf5` workflow from the current section, click the button in the lower right corner and select *Jobs as script*. This will launch an *Export to script* window where you can specify *Script exporting options*. Lets change the result filename to *bandstructure_results.hdf5*. Leave the rest of options to default and click *OK*.

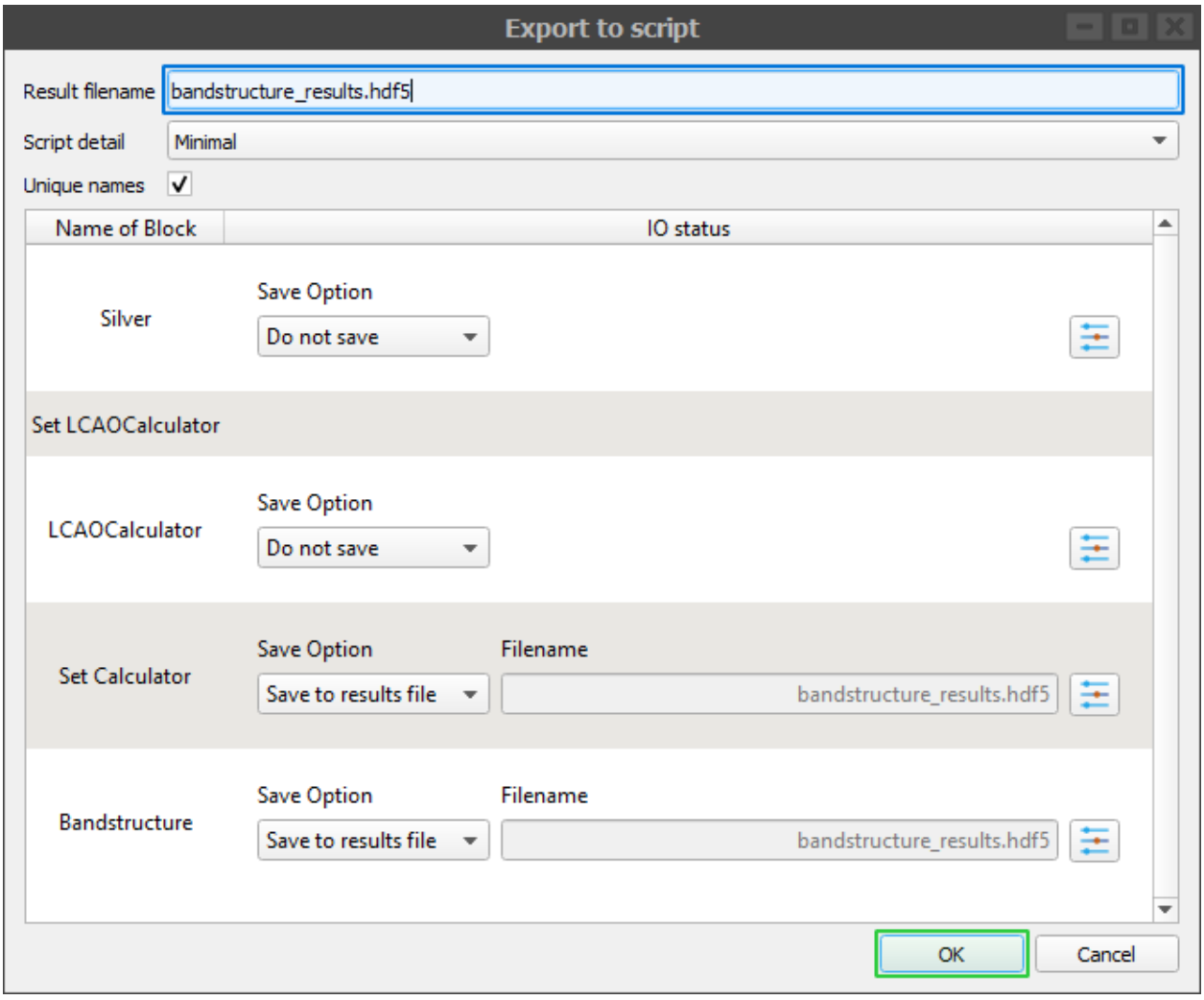
Note: Using the options highlighted in red at the bottom of the Workflow Area window you can specify a default name of the result file and the log verbosity of the calculation.

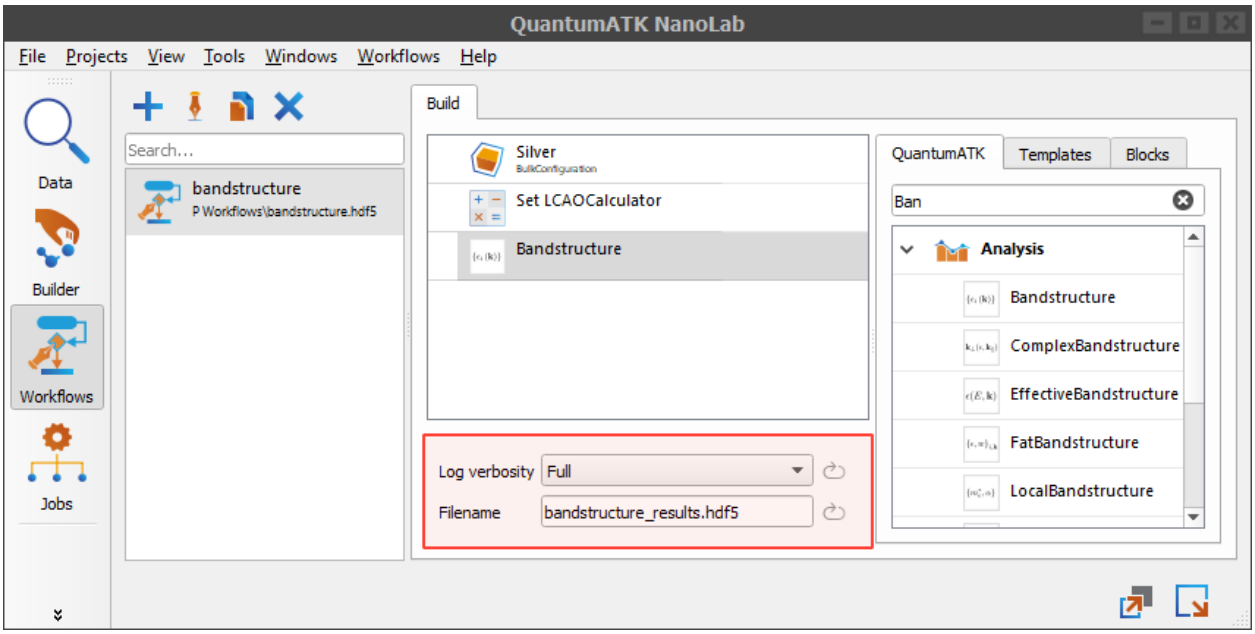
After setting the export options you will need to follow the steps to send a script to the *Jobs* tool.

Start by clicking the *Ok* button in the *Export to script* window. This will open a save dialog. Choose a place to save your script and click *Save*. Next you will see the *Submission Settings* window where you can select run options, such as the machine where the script will be run. After you have set the submission options, click the *Submit* button.

To learn more about running scripts through the *Jobs* tool, follow the *Jobs* tool guide.

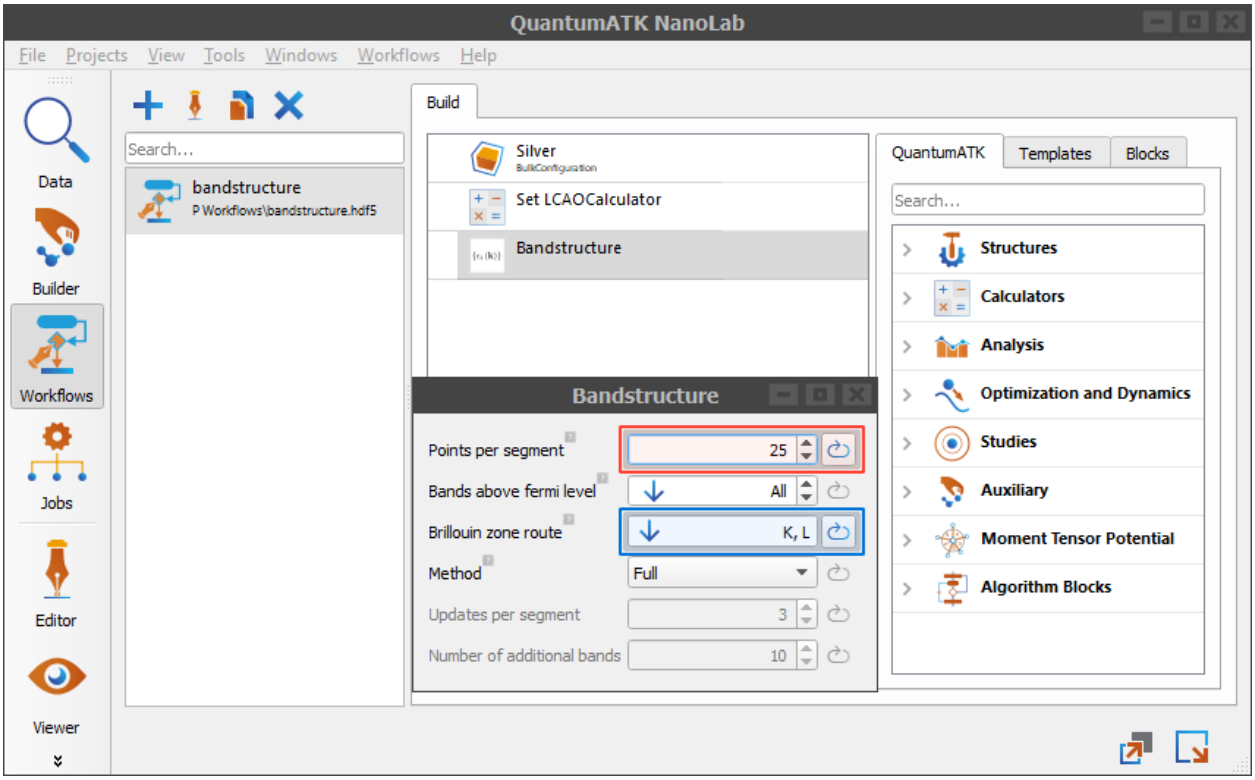
After the *Jobs* tool finishes, the result file `bandstructure_results.hdf5` should appear in the *Data* tool. The result file should contain two saved objects – a `BulkConfiguration` with an attached `LCAOCalculator` and a `Bandstructure` object.


Export to script
Result filename
bandstructure_results.hdf5
Script detail
Minimal
Unique names
Name of Block
IO status
Silver
Save Option
Do not save
Set LCAOCalculator
LCAOCalculator
Save Option
Do not save
Set Calculator
Save Option
Filename
Save to results file
bandstructure_results.hdf5
Bandstructure
Save Option
Filename
Save to results file
bandstructure_results.hdf5
OK
Cancel


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Search...
bandstructure
P Workflows\bandstructure.hdf5
Build
Silver
Set LCAOCalculator
Bandstructure
Log verbosity
Full
Filename
bandstructure_results.hdf5
QuantumATK
Templates
Blocks
Ban
Analysis
Bandstructure
ComplexBandstructure
EffectiveBandstructure
FatBandstructure
LocalBandstructure

Re-running a workflow

Lets make a change to the `bandstructure.hdf5` workflow from the previous section. Go ahead and double click the *Bandstructure* block to open its editor. Increase the sampling by setting *Points per segment* to 25 and set the *Brillouin zone route* to *K, L*.



After the change, there are two ways to re-run the workflow.

- Run the entire workflow from scratch
- Reuse some of the saved objects from the previous calculations

We can always run the entire workflow from scratch. This way we are not going to use any of the results saved in the previous run. This is the easiest way, but not a very economic one.

The other way is to use some of the saved objects from the previous calculation found in `bandstructure_results.hdf5`. It is crucial that the object we intend to reuse in a new calculation are not affected by the change.

The previous calculation saved two objects – a *BulkConfiguration* with an attached *LCAOCalculator* and a *Bandstructure* object. In this example, it is easy to see that the first object can be used again in the new workflow, because it is not affected by the changes in the *Bandstructure* block. To include this object in the workflow, we will reuse a *Load from file* block.

The second way is obviously economical and can be faster, but you need to be extra careful when cherry-picking object to use in the new calculation.

Load from file

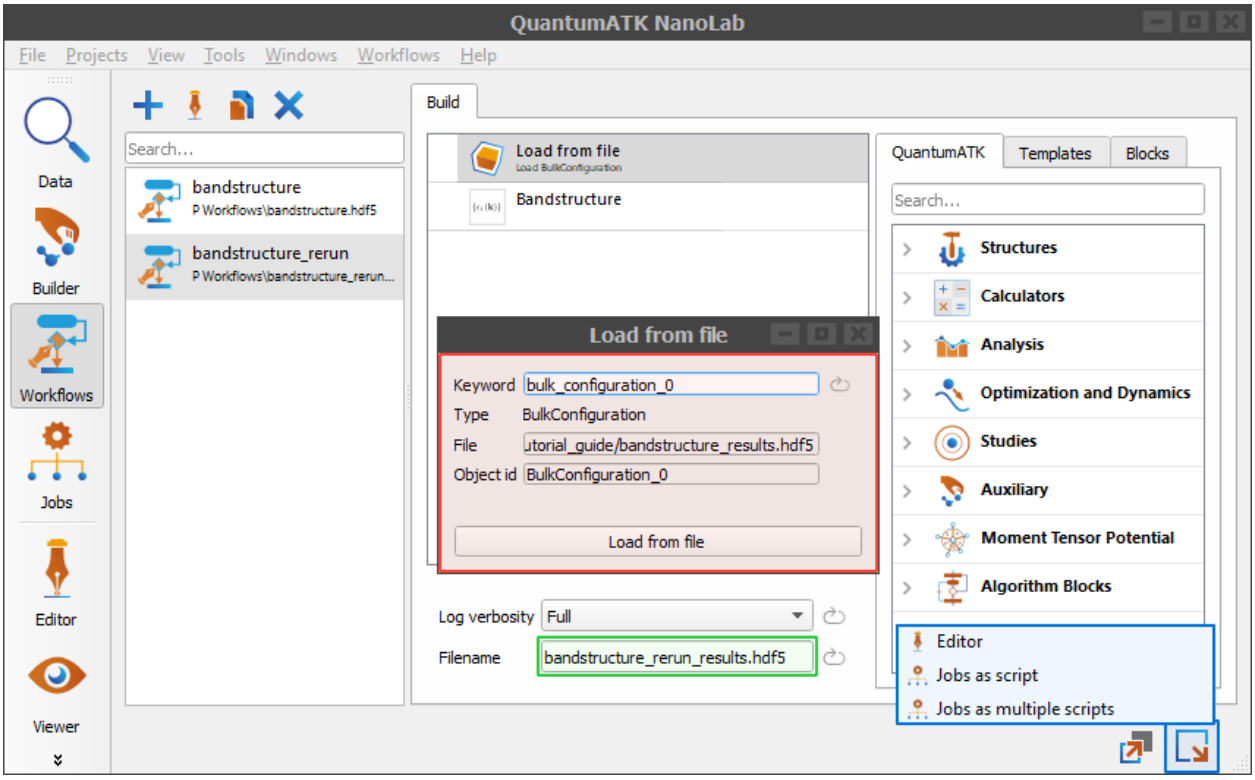
This block is very simple, but powerful. It enables you to reuse already calculated parts after making a change to the same workflow. With this block you can simply load an object from a result file.

In the previous section *Re-running a workflow*, we have modified the `bandstructure.hdf5` workflow by changing one of the properties in the *Bandstructure* block. Now we want to re-run the workflow using the same configuration object with the attached calculator, which were unaffected by the change.

Lets start by creating a duplicate of the `bandstructure.hdf5` workflow by clicking *Copy*. button in the Workflow Stash on the left. This will duplicate the current workflow. Change the name of the copy to *bandstructure_rerun*. Then go ahead and delete the existing configuration and calculator blocks. We will provide both objects from the result file of the previous calculation.

In the catalog of QuantumATK blocks under the *Algorithm Blocks* group find the *Load from file* block and place it as the first block in the workflow. You should now have only two blocks in the workflow - *Load from file* block and the *Bandstructure* block.

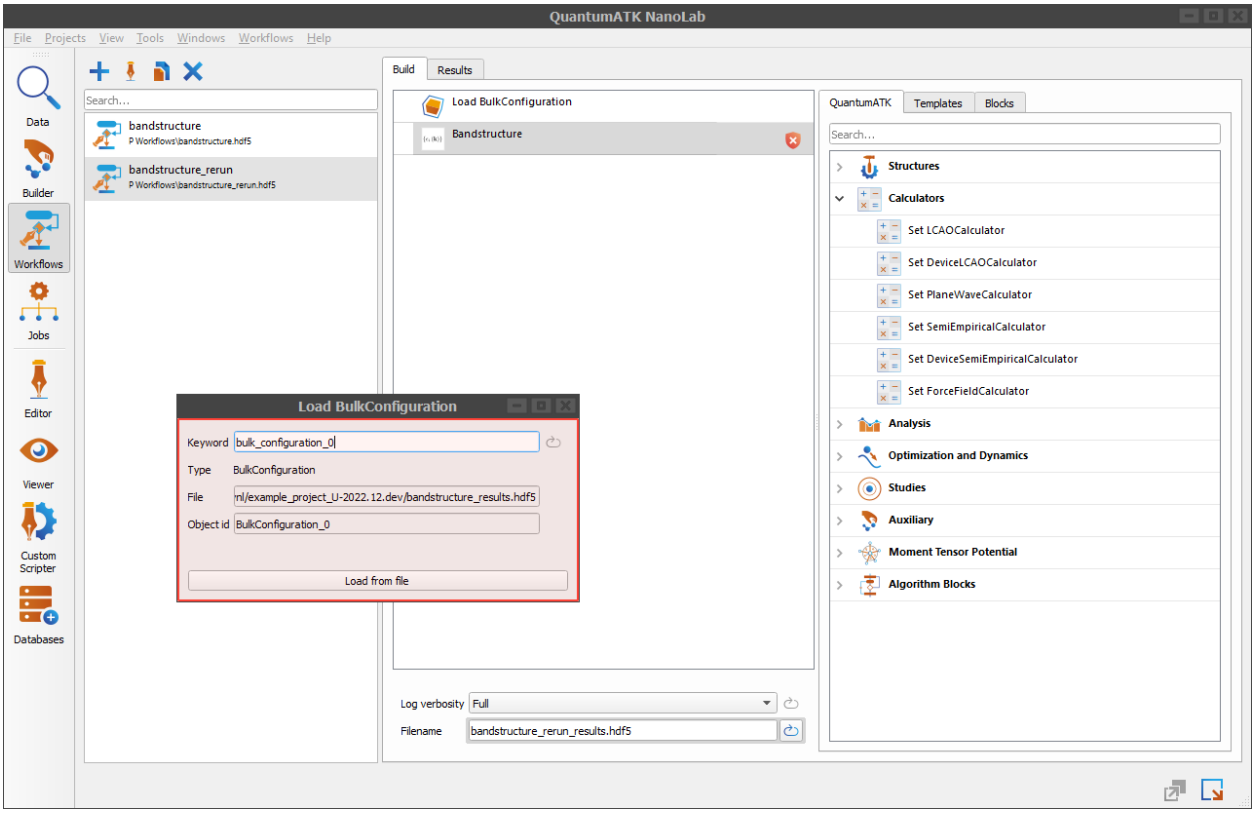
Note: To change the order of the blocks in the workflow, just drag and drop them within the Workflow Area window.



You will notice that both blocks have errors. A *Load from file* block is missing a file to load *QuantumATK Engine* objects from, and a *Bandstructure* block needs a `BulkConfiguration` with an attached calculator.

To resolve errors, we need to load a bulk configuration with an attached calculator using the *Load from file* block.

To open its editor, double click the *Load from file* block. To load objects click the *Load from file* button at the bottom of the editor. Select the result file from the previous calculation, `bandstructure_results.hdf5`, and load the bulk configuration with the attached calculator.



After loading the bulk configuration in the *Load from file* block, all the errors should be gone and you will see some details in the editor window like the bulk configuration type, its file location and its object id.

Change the result file name of the new workflow to the *bandstructure_rerun_results.hdf5*. The filename option is placed at the bottom of the Workflow Area window.

Run the workflow by clicking the button to send it to the *Jobs* tool. Select *Jobs as script* option and follow the script export steps and *Jobs* tool steps the same way as in the previous section.

After the *Jobs* tool finishes, you should get results similar to the `bandstructure_rerun_results.hdf5`.

Workflow with multiple branches

The new workflow feature makes it much easier to run several calculations with only a small difference, e.g., calculating the `Bandstructure` for the same atomic configuration, but with different calculator settings.

To do this we could make copies of the same workflow and then tweak calculator settings in each of the copies. This is not a very good way, because keeping the track of all the workflows can become very cumbersome.

What we should do instead is split the workflow into multiple branches of execution. To do this we will need to use the *Split Parallel* block from the *Algorithm Blocks* section in the QuantumATK catalog of blocks.

Lets start by creating a new workflow with one `BulkConfiguration` block and one `Bandstructure` block.

Send a *Silver* configuration from the *Builder* to the *Workflow Builder* as described in *Sending an atomic configuration from the Builder to the Workflow Builder*. This will create a new workflow with one configuration block.

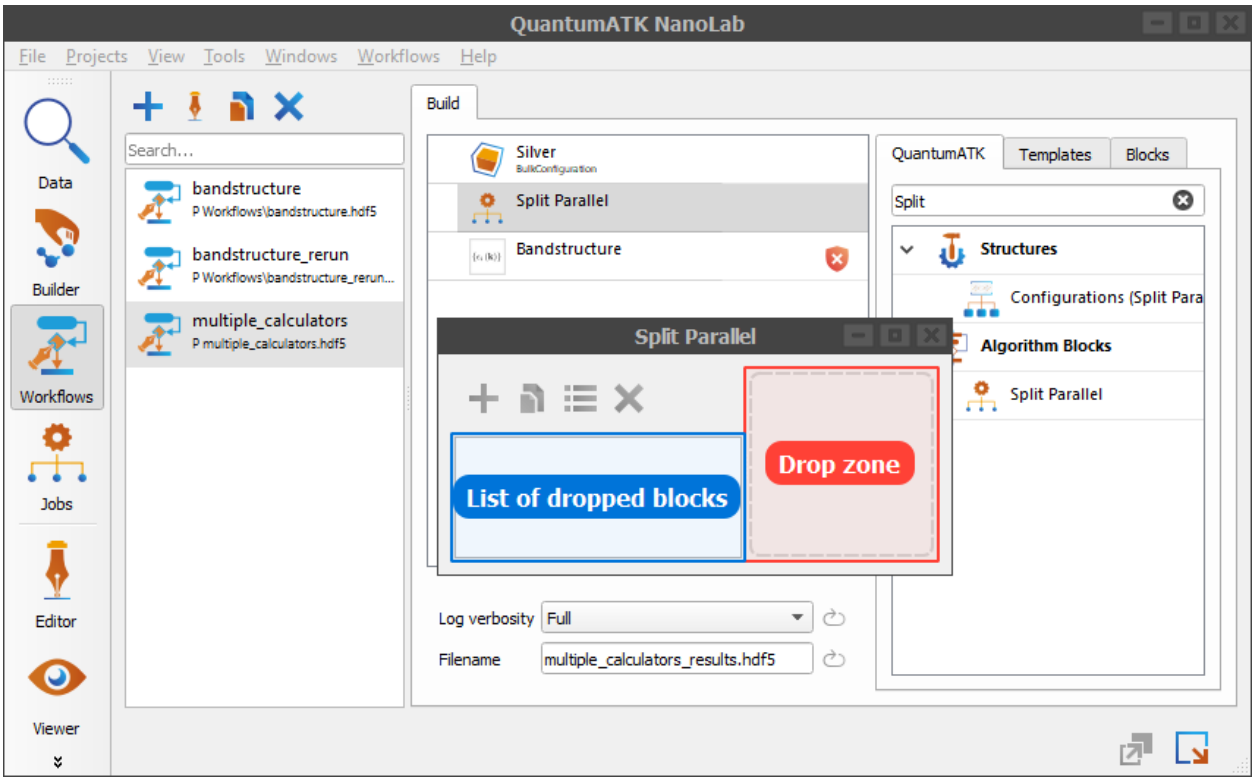
Now add a `Bandstructure` block as described in *How to add an analysis object*.

Rename the created workflow to *multiple_calculators*, since we will be using multiple calculators in the same workflow.

We want to have two separate branches of execution. Each branch of execution should calculate a `Bandstructure` of the same *Silver* configuration, but with different `LCAOCalculator`-s.

To split the execution of the workflow, go ahead and drop *Split Parallel* block from the QuantumATK catalog under the *Algorithm Blocks* group. In the workflow, the new block needs to be placed between the `BulkConfiguration` and the `Bandstructure` blocks.

Double click the added *Split Parallel* block in the Workflow Area to open its editor (see the figure below).

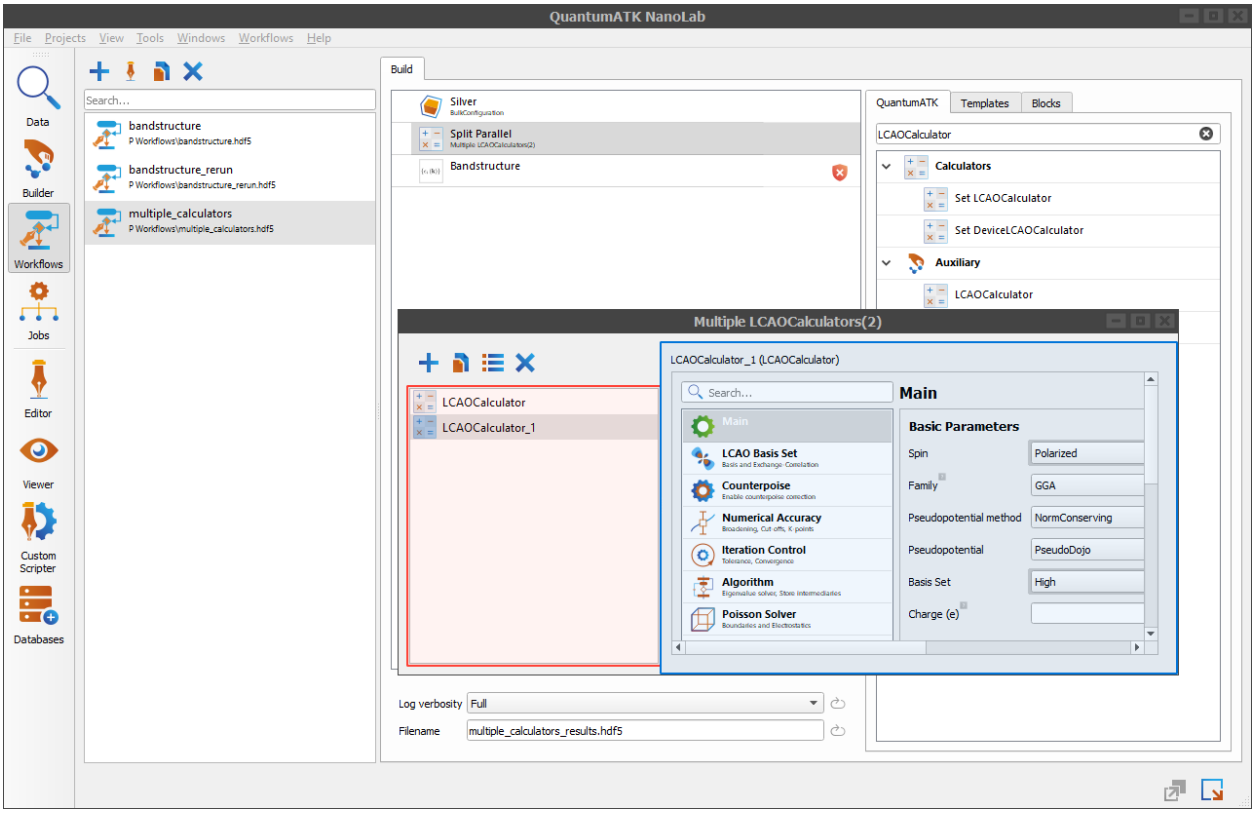


On the right highlighted in blue you will see a *drop zone* – an empty space to drop blocks from the catalog of blocks. On the left highlighted in red is a list of dropped blocks, which is currently empty.

Each block in the list will belong to a separate branch of execution. In other words, each block in the list will create one branch of execution. If we add one block in the list, there would be only one branch of execution. If we add ten blocks, we would be creating ten branches, etc

We want to have two separate branches of execution, each containing different `LCAOCalculator`-s. To do this, we will add two calculators to the *Split Parallel* block.

To add the first `LCAOCalculator` calculator, find an *LCAOCalculator* block in the QuantumATK catalog under the *Auxiliary* group and drop it to the drop zone. To add the second calculator, just click the + button. This will create a copy of the first calculator block (see the figure below).



In the list of blocks highlighted in red on the left, we now have two *LCAOCalculator* blocks, one named *LCAOCalculator* and the other named *LCAOCalculator_1*.

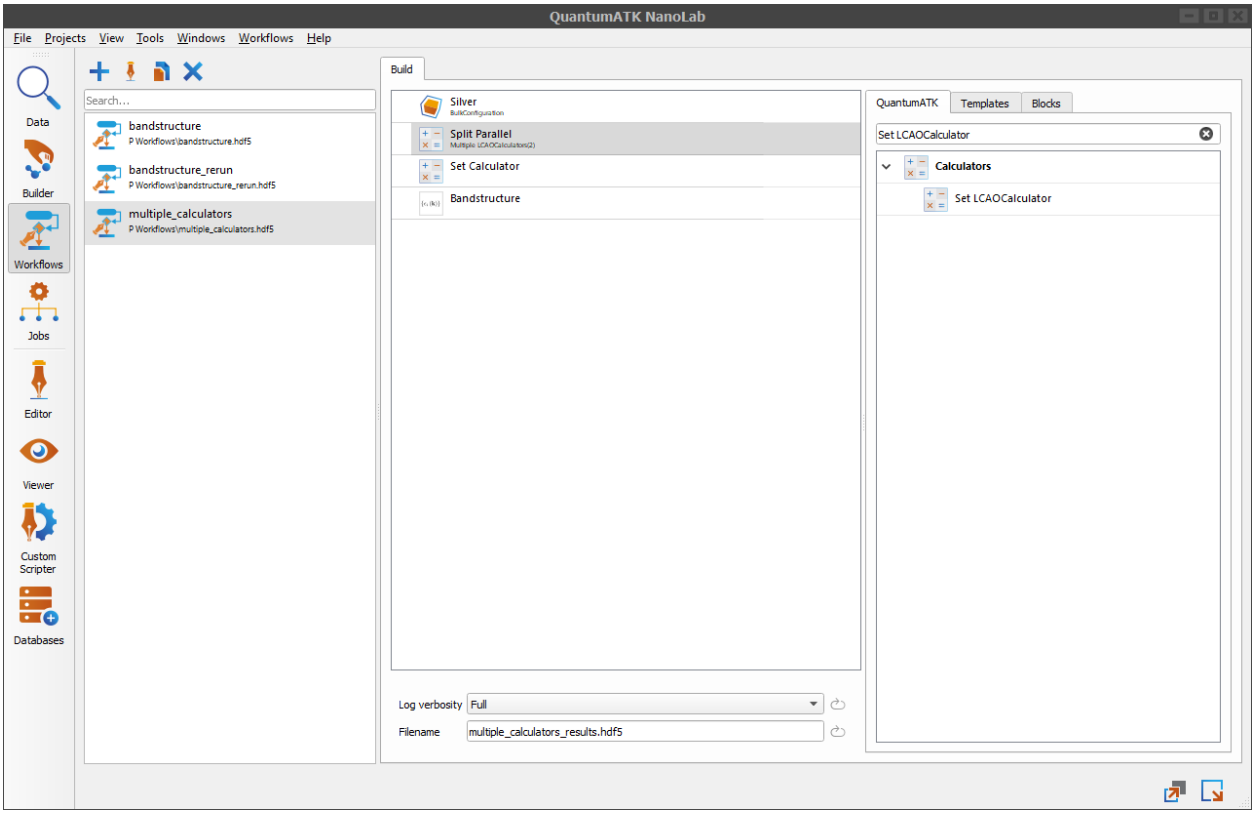
In the same window on the right highlighted in blue you can see an editor of the selected calculator block in the list. We can use the editor displayed on the right to change the settings of the calculators.

Lets change some of the settings in the second calculator. Select the second calculator in the list on the left and change its spin from *Unpolarized* to *Polarized* and the *Basis Set* from *Medium* to *High*. Both options are under the *Main* editor tab.

In the workflow you will notice some errors reported by the *Bandstructure* block saying that a `BulkConfiguration` with an attached calculator is missing. Even though we do have both the `BulkConfiguration` and the `LCAOCalculator`-s, we havent linked them. To resolve this problem, add a *Set Calculator* block right after the *Split parallel* block.

The *Set Calculator* block requires two inputs – an atomic configuration and a calculator. Its only purpose is to attach the calculator to the configuration.

Note: Note that there are *LCAOCalculator* and *Set LCAOCalculator* blocks. The first one just creates an `LCAOCalculator` and the other both creates the calculator and attaches it to a configuration.



Note: Note that the icon of the *Split Parallel* block has changed to , since the block has two calculators inside.

In the workflow you should now have four blocks - *BulkConfiguration* block, *Split Parallel* block, *Set Calculator* block, and `Bandstructure` block.

The workflow should now be ready for execution. There are different ways to execute it, as described in the section *How to run a workflow*.

The recommended way to run the workflow is to run it in parallel, as described in the next section *Running a workflow with multiple branches*.

After the *Jobs* tool finishes workflow execution, you will get results similar to `multiple_calculators_results.hdf5`.

Running a workflow with multiple branches

Workflows containing *Split parallel* blocks can have multiple branches of execution - one for each sub-block in the list. The recommended way to run such workflows is to export them as multiple scripts, send them to the *Jobs* tool and run them all at once. Each exported script will contain one branch of execution.

Note: The *Configurations (Split Parallel)* block also causes multiple branches of execution - each branch for every loaded atomic configuration.

In the lower right corner, click and select *Jobs as multiple scripts*. Follow the script export process, which is similar as exporting a workflow as a single script. See *Script exporting options* for more script exporting details.

Workflow with multiple configurations

You can increase the calculation throughput by increasing the number of configurations.

To show an example, lets first make a copy of the first workflow we have created: `bandstructure.hdf5`. Rename the copy to *multiple_configurations*.

Note: Alternatively you can create a new workflow with a default `LCAOCalculator` and a `Bandstructure` blocks. Follow the section *How to create a new workflow* to check how to create a workflow from scratch.

We will need to add more than one configuration to the workflow. The first block in the copied workflow is limited to hold only one configuration. Go ahead and delete that block.

To add more configurations to the workflow, we will need to use the *Configurations (Split Parallel)* block. You can find this block in the catalog of QuantumATK blocks under the *Structures* group.

Add the block by double clicking it or dragging and dropping it from the catalog to the Workflow Area. The block should be the first one in the workflow.

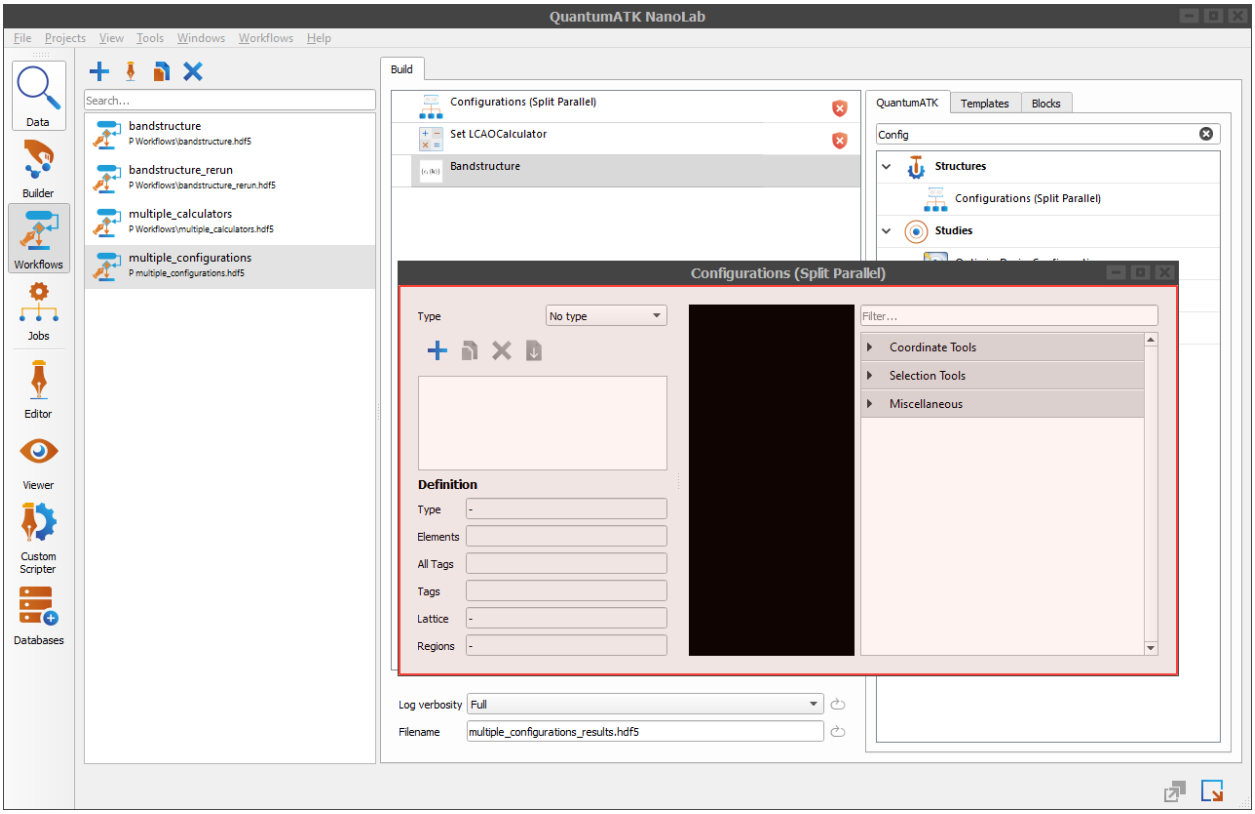
After adding the block, you will notice an error icon. Double click the icon to see the error message. The message should say that *a configuration is missing*. This is because the block is initially empty and does not have any atomic configurations, which is not very useful.

We will load two atomic configurations from the *Builder_Stash.hdf5* file.

Note: The *Builder_Stash.hdf5* file is automatically created by the *QuantumATK NanoLab*. It is project-specific and it contains atomic configurations from the *builder stash* in the *Builder* tool. You can find the file in the project folder. It is also present in the *Data* tool.

Before we can load atomic configurations from a *Builder_Stash.hdf5* file, we will need to create them in the *Builder* tool and add them to the *builder stash*.

To achieve this, we are going to use the *Crystal* database from the *Builder* tool. The steps to load a configuration from the *Crystal* database to the *builder stash* are similar to what we already did in the section *Sending an atomic configuration from the Builder to the Workflow Builder*.



In the *Builder*, in addition to the *Silver* configuration, load the *AgCd* configuration as well. After you load these configurations from the *Crystal* database, you should see the two configurations in the *builder stash* under names *Silver* and *AgCd*.

When you load the *Silver* and *AgCd* configurations in the *builder stash* return to the *Workflow Builder* window by clicking the icon located in the side toolbar on the left.

We are now ready to load two atomic configuration from a *Builder_Stash.hdf5* file.

To load configurations, open the blocks editor by double clicking the *Configurations (Split Parallel)* block in the Workflow Area. In the editor click the button. This will open a file browser within the current project directory.

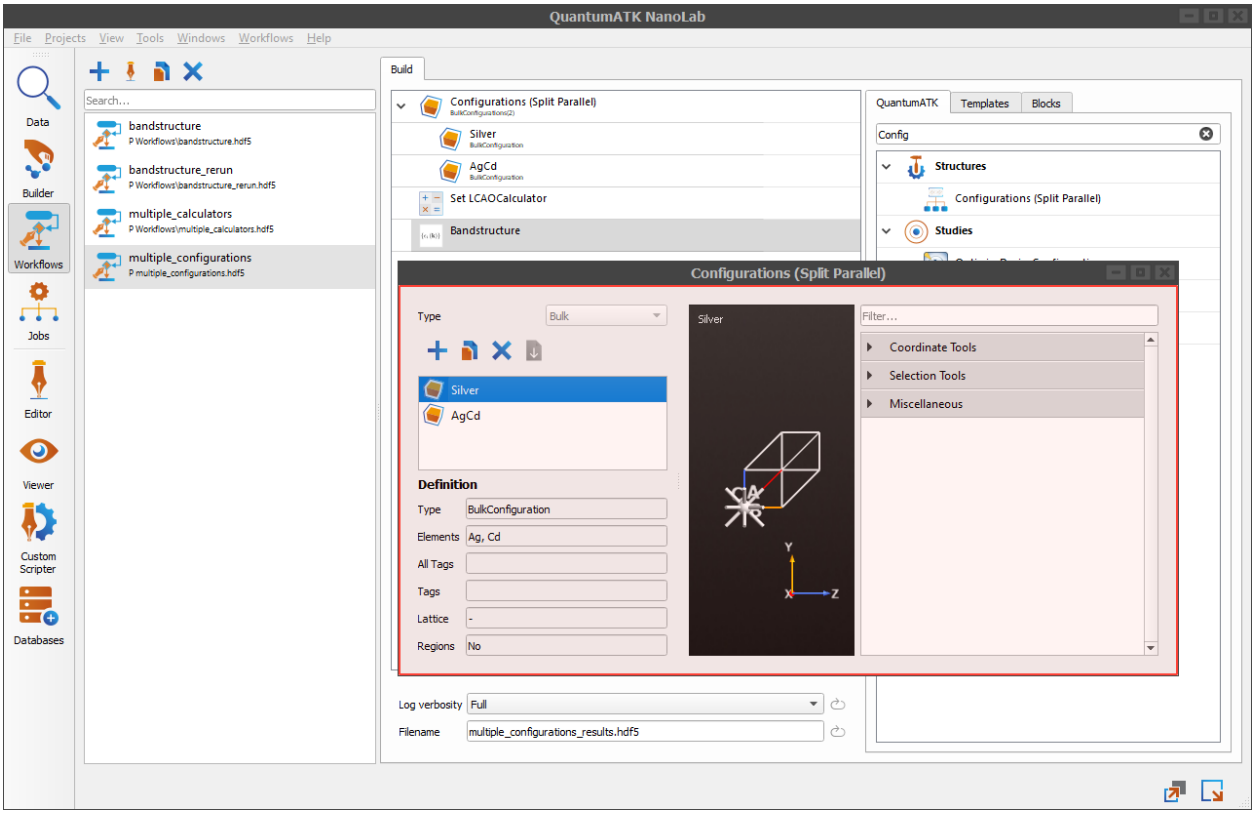
The central window of the file browser contains a list of files in the current directory. Note that the file list also includes the *Builder_Stash.hdf5* file. The window on the right displays the contents of a file selected in the central window of the file browser.

Select the *Builder_Stash.hdf5* file and choose two `BulkConfiguration`-s from its contents - the *Silver* and *AgCd*.

Click *Load* to load the selected atomic configurations.

Note: If you do not see these configurations in the *Builder_Stash.hdf5* file, go to the *Builder* and load the *Siliver* and *AgCd* configurations from its database. Follow the *Builder* guide if you are unfamiliar with the tool.

The loaded atomic configurations will appear in the list of atomic configurations on the left and the 3D window will display graphical representation of the currently selected one.



In the *Definition* list you will see a summary of the configuration properties common for both of the atomic configurations in the list. The type, tags, lattice and regions show common properties found in each configuration, while the elements and tags include unions of all the unique elements and tags.

To add new tags or regions to any of the atomic configurations from the list, use the *Tags* or *Spatial Regions* plugins in the right window.

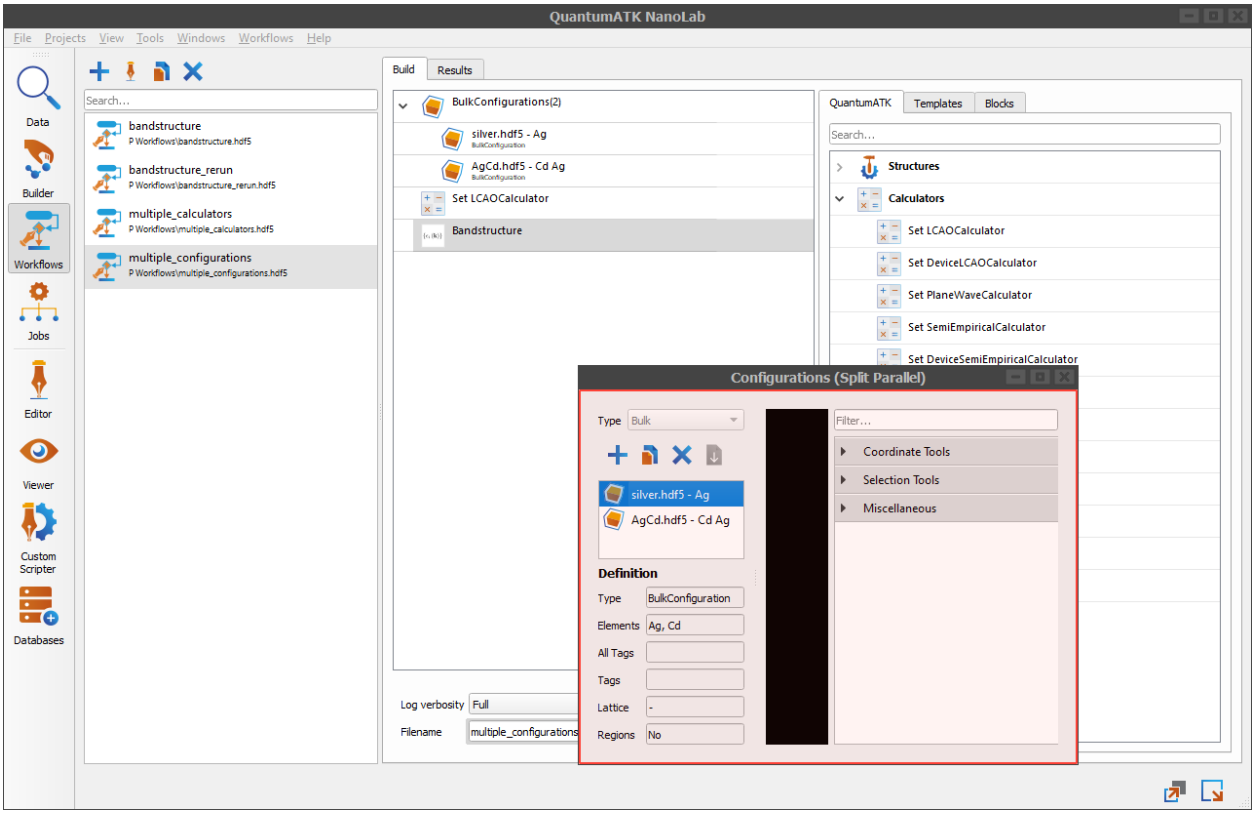
Adding additional atomic configurations in the same block is similar as before. The only constraint now is that subsequent atomic configurations have to be of the same type, i.e. `BulkConfiguration`.

If you now close the block editor and look at the Workflow Area, you will notice some changes in the block – the blocks icon has changed from to *BulkConfiguration*, and two sub-blocks, one called *Silver* and the other called *AgCd*, corresponding to the just loaded `BulkConfiguration`-s.

We can now run a `Bandstructure` calculation for each of the `BulkConfiguration`-s from the *Configurations (Split Parallel)* block. Adding two bulk configurations creates two branches of execution, which we can run in parallel for better efficiency.

Change the result file to *multiple_configurations_results*. Run the workflow in parallel as described in the section *Running a workflow with multiple branches*.

After the *Jobs* tool finishes execution of the workflow, you should get the same results as in the `multiple_configurations_results.hdf5` file.



Block of blocks

Lets append more analysis blocks in the `bandstructure.hdf5` workflow. In addition to the *Bandstructure*, lets also calculate *BornEffectiveCharge* and *BlochState*. Start by duplicating the `bandstructure.hdf5` and rename the copy to *block_of_blocks* - the name will make more sense a bit later.

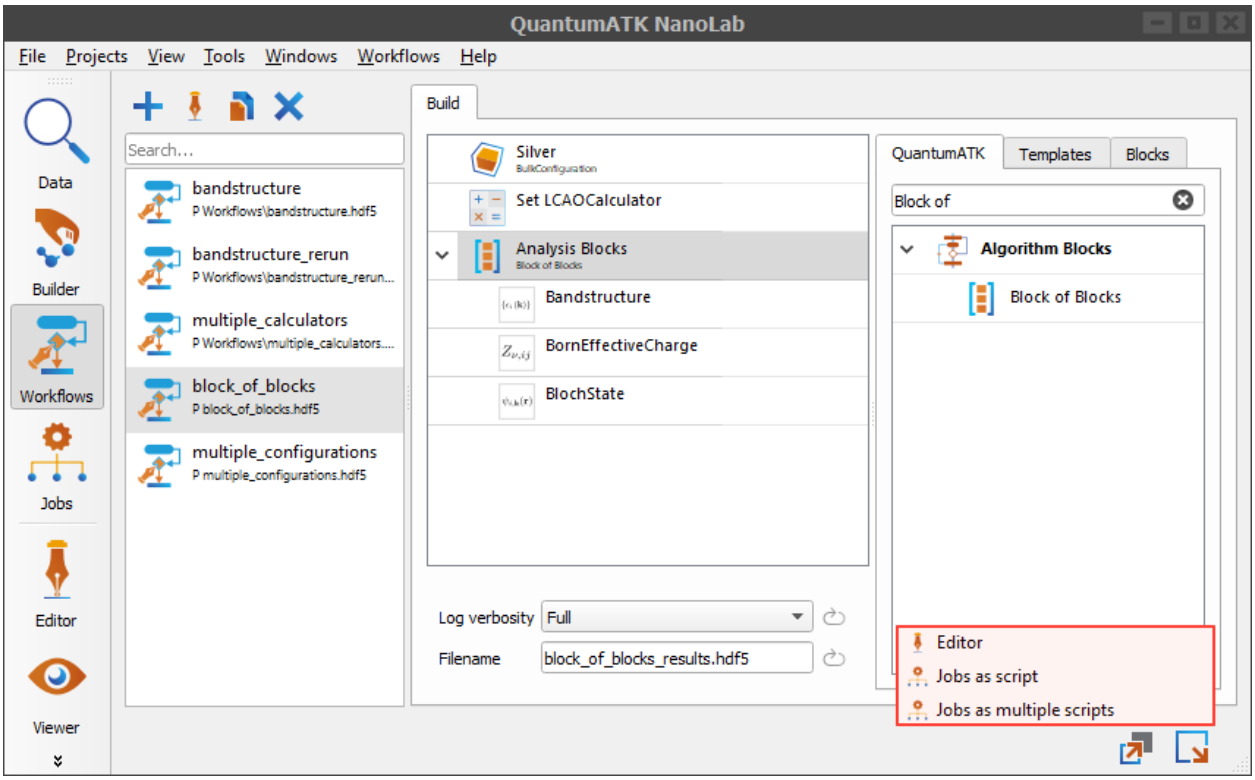
Now append the two new analysis blocks to the workflow. You can find both blocks in the catalog of QuantumATK blocks under the *Analysis* group. We can now look at the workflow as a three step calculation – bulk configuration loading, attaching an LCAO calculator and analyzing.

We can group the last three analysis blocks under a single block. To do that, we will use *Block of Blocks*. You can find it under the *Algorithm Blocks* category.

Add the *Block of Blocks* at the end of the workflow and then drag and drop analysis blocks under it. The analysis blocks should appear as three sub-blocks under the *Block of Blocks*. Rename the *Block of Blocks* to *Analysis Blocks* by right clicking it and selecting *Rename* from the drop down menu.

The *Block of Blocks* is used for grouping multiple blocks into one for easier managing. Use this block to save a sequence of blocks specialized to calculate or analyze a set of properties, which you can later reuse in other workflows. To learn how to save custom blocks for later reuse, see the section *How to save a block from a workflow*.

We can now run a calculation with multiple analysis objects. Change the result file to *block_of_blocks_results*. The workflow contains only one branch of execution, so we can only run it sequentially.



Lets send the workflow to the *Jobs* tool. Click the in the lower right corner and select *Jobs as script*. Follow the script export process. See *Script exporting options* for more script exporting details.

After the *Jobs* tool finishes execution of the workflow, you should get the same results as in the `block_of_blocks_results.hdf5` file.

Note: Note that the last workflow example can contain only one branch of execution. This is because the *Split Parallel* block requires blocks of the same type and the analysis objects in the workflow had different types.

Common operations in the Workflow Builder

In the next sections you will learn about the common operations you will use in the *Workflow Builder*, such as how to create a workflow from scratch, how to save a current workflow and a custom workflow block, as well as how to share saved blocks and workflows.

You will also learn about the workflow exporting options, as well as the ways to run a workflow through the *Jobs* tool.

How to create a new workflow

To create a new workflow click the in the Workflow Stash window. A new workflow should appear in the Workflow Stash with a generic name. Since the new workflow is empty, the Workflow Area does not show any contents yet. Click the to change the name of the workflow to *bandstructure*.

The workflows in the Workflow Stash are project-specific. You can import workflows from other projects like you would import any other `.hdf5` file.

Note: To see the file location of a workflow, right click it in the Workflow Stash and select *Open location*.

How to save a workflow

All workflows are saved automatically, and every change is permanent. There is no *Save* button. You can find workflows in the Workflow Stash in the *Workflow Builder* or in the *Data* tool.

It is important to know that the workflows in the *Workflow Stash* are project-specific. You can still import workflows from other projects like you would import any other `.hdf5` file.

How to share a workflow

Workflows are saved as `.hdf5` files, which means they can be exported and imported like all other QuantumATK objects or files.

You can find your workflows among files in the *Data* tool and in the Workflow Stash of the *Workflow Builder*. Workflow files can be easily spotted in the *Data* tool by the icon. To find a location of a workflow file, just right click it from the *Data* or from the Workflow Stash in the *Workflow Builder* and select *Open location*.

Once you know the location of the workflow file, you can import it in the current project by selecting *File -> Import File* from the top menu.

When you import the workflow file, it will automatically become available in the *Data* tool and in the *Workflow Builder*, as the *QuantumATK NanoLab* takes care of finding workflow files present in the current project folder.

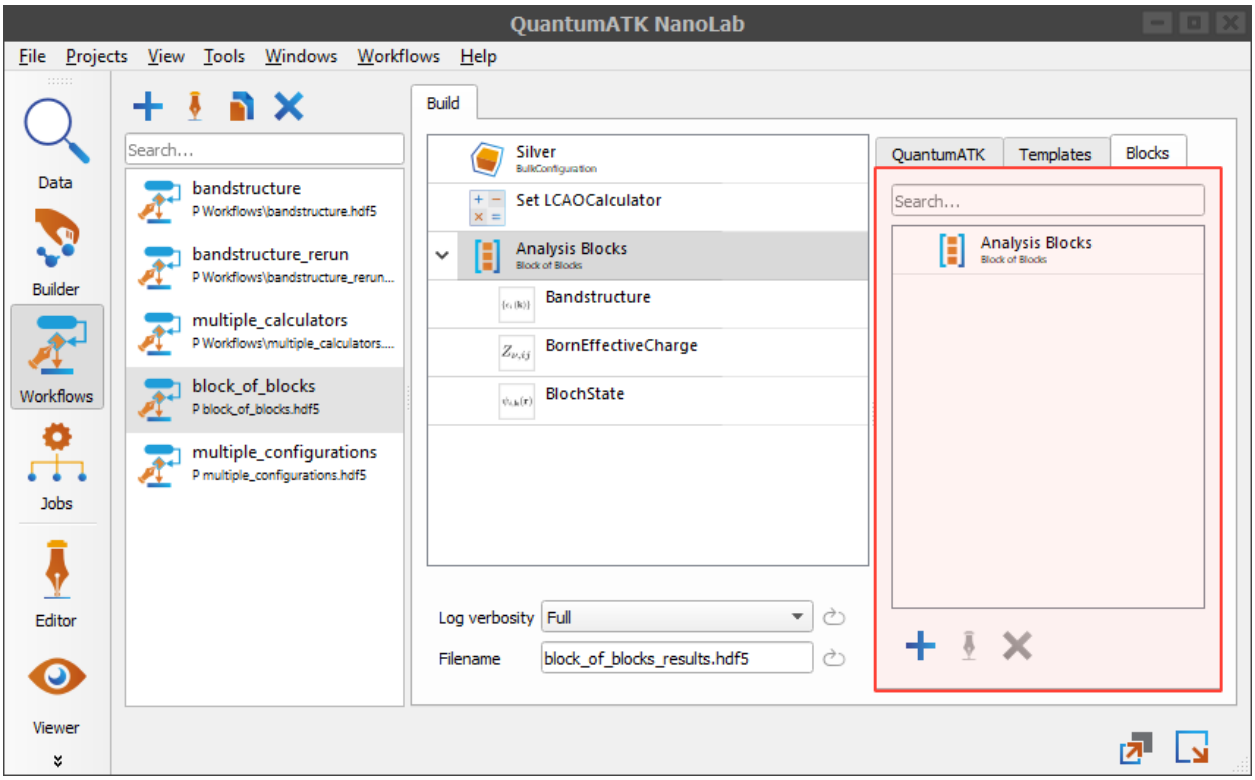
How to save a block from a workflow

Lets say you have a calculator block with a specific set of settings or a group of blocks which do some specific analysis you would like to reuse, like the *Analysis Blocks* block in the `block_of_blocks.hdf5` workflow.

To save a block, simply drag and drop it from the Workflow Area to the *Blocks* window in the catalog of blocks.

Lets save a block in the `block_of_blocks.hdf5` workflow. Drag and drop the *Analysis blocks* on the catalog window under the *Blocks* tab. You can rename the saved block, or add it under a new category. This way the block is permanently saved and can be reused in new workflows.

It is important to note that the saved blocks are shared across all projects.



How to run a workflow

A workflow can be sent to the *Editor* or the *Jobs* tool. To send a workflow to either of these two tools, click the send to button in the bottom right corner of the *Workflow Builder*. This will show a send-to-menu with three options:

- Editor
- Jobs as script
- Jobs as multiple scripts

The first option will open the workflow as a single Python script in the *Editor*. From here you can inspect, edit or save the workflow script.

Option *Jobs as script* and *Jobs as multiple scripts* would export a workflow as a single or multiple Python scripts. You can control the export options such as saving calculation objects at different calculation stages or keeping object unique names. Follow the section *Script exporting options* to read more about the exporting options.

The advantage of multi-script export versus single-script export is obvious in cases when a workflow contains multiple configurations. One script would be generated for each execution branch of the workflow containing a different configuration. These multiple scripts can then be executed in parallel using the *Jobs* tool.

Script exporting options

To generate a script based on the current workflow, click and select *Editor* from the drop menu. A new *Export to script* window will open where you can tweak some of the exporting options. In the following section you can read about script exporting options:

•Result filename

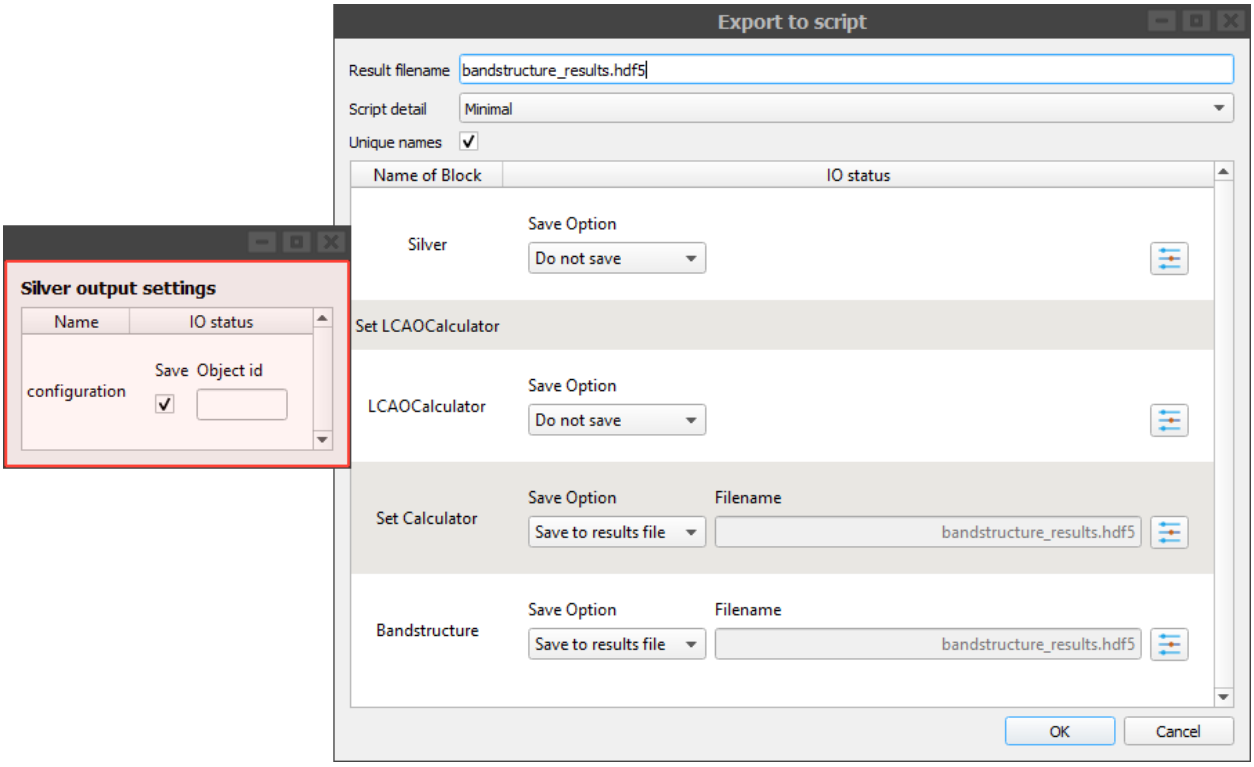
•Script details

•Unique names

•Save options

•Output savings

In the next sections we will refer to the figure below. It shows script exporting options available for the `bandstructure.hdf5` workflow in the section *How to run a workflow*.



Result filename

This will be the filename of the calculation results file. A default filename is the same as the *filename* field at the bottom of the Workflow Area window of the *Workflow Builder*.

Script details

A script can be generated with higher or lower level of detail. The *Report all* option will script all the default values, while the *Minimal* option should only script the values directly or indirectly changed by the user.

Minimal option will result in a more compact script with only the most important parameters visible, while the *Report all* option will give a long and detailed script, which can be useful for troubleshooting.

Unique names

If checked, the script will have unique object names. Otherwise, it will overwrite them once they are not needed anymore by the previous calculation. A typical scenario is when there is more than one configuration in a configuration list. With this option unchecked both configuration variables would have the same name.

Save options

During the calculation you might want to save objects from a workflow not only at the end of calculation, but also at specific calculation stages. This can help when analyzing results and re-starting a calculation with certain parameter changes.

For example, the result file in the `bandstructure.hdf5` workflow would include objects such as a `BulkConfiguration`, an `LCAOCalculator` and a `Bandstructure`. Beside saving the `Bandstructure` and a final `BulkConfiguration` object with an attached `LCAOCalculator`, you could also save the initial `BulkConfiguration` before attaching the calculator. This way you could reuse the same configuration for another calculation with a different calculator.

Saving objects during certain calculation stages can be useful and this is what the save options enable you to do. You can save an object in the final result file, a custom file or skip its saving.

For example, each row in the referenced figure corresponds to a block in the `bandstructure.hdf5` workflow, and each block corresponds to an *QuantumATK Engine* object at a certain stage:

- Silver - controls the saving of the *Silver* configuration right after its construction
- Set LCAOCalculator - a header for the next two rows
- LCAOCalculator – controls the saving of the calculator object
- Set Calculator - controls the saving of the *Silver* configuration after attaching the `LCAOCalculator` to it
- Bandstructure - controls the saving of a `Bandstructure` analysis object.

Output settings

When saving a *QuantumATK Engine* object to a file you can index it by specifying a unique identifier. This helps finding a specific object when reading the file later. To set the object identifier, click on the . This will open a *Silver* *output settings* window - highlighted in red in the referenced figure from this section. From this window you can set an *object id* field - the object identifier.

2.5.2 Using Tables in the Workflow Builder

Downloads & Links

`showcase_table.hdf5` `iteration_block_with_table.hdf5` `table_iteration.hdf5`

The *Table* is a powerful tool that can be used in a variety of situations. The main purpose is to collect data, for later analysis. *Tables* are also used as a container to pass initial training data to the ActiveLearningSimulation, etc.

In the next sections you will learn how to use the *Table* block and *TableIteration* block, through a set of simple examples:

- **Table**. The block that collect data in a table.
- **TableIteration**. The block that iterate over rows of a table.

These two blocks can be found in the Workflow Catalogs under *Algorithm*, and are very convenient ways to first collect data, and later process the collected data.

In the *Downloads & Links* sidebar you will find a list of all the workflows that are created in the following examples.

Table of Contents

- *How to collect data in a table*
- *How to collect data in a table using an Iteration block*
- *How to extract data from a table using a TableIteration block*

How to collect data in a table

The most simple workflow is to add a Configuration to a Table. Thus in this example we have placed a single *Configuration* block followed by a *Table* block.

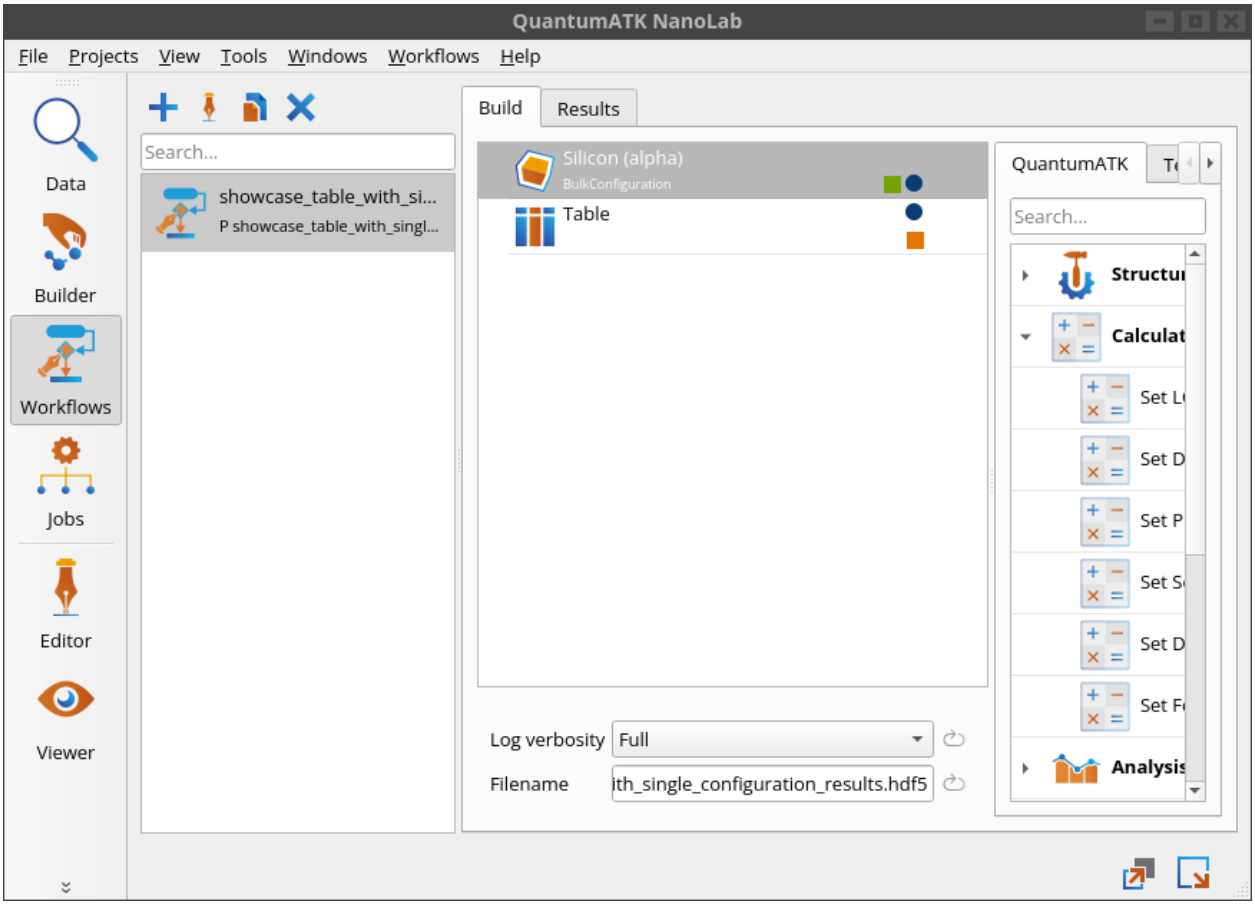
The purpose of the *Table* block is to collect data in columns and rows, thus we need to specify how many columns the table should have, and which data type the column should accept.

Double click to edit the *Table* block. In this case we want a single column that contain a BulkConfiguration. Thus, the data type of the first column should be BulkConfiguration. Click on the *Add* button to add a column to the table specification.

Note: A Table consists of columns with different types of data and rows for each entry. The different data types for a column will be explained below.

Several column options are available:

- **Type**. This is the most generic column type. You simply need to specify the type of the objects to collect.
- **PhysicalQuantity**. Collects a `PhysicalQuantity`, and requires the unit of the `PhysicalQuantity` to collect.
- **Configuration**. Collects a Configuration, and a large array of setting for further specification. Such as to only collect *`MoleculeConfiguration`*, or only collect *`BulkConfiguration`* with a calculator attached, or only collect *`BulkConfiguration`* with a Simple Cubic Bravais lattice type, etc.


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Search...
showcase_table_with_si...
P showcase_table_with_singl...
Build
Results
Silicon (alpha)
BulkConfiguration
Table
Log verbosity
Full
Filename
ith_single_configuration_results.hdf5
QuantumATK
Search...
Structu
Calculat
Set L
Set D
Set P
Set S
Set D
Set F
Analysis

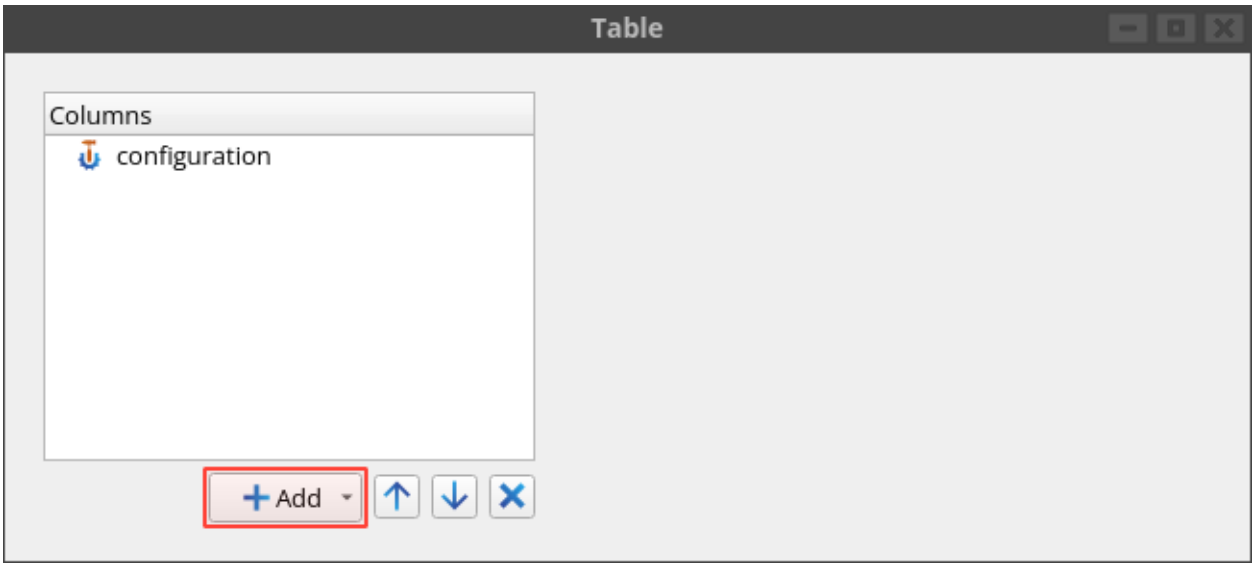
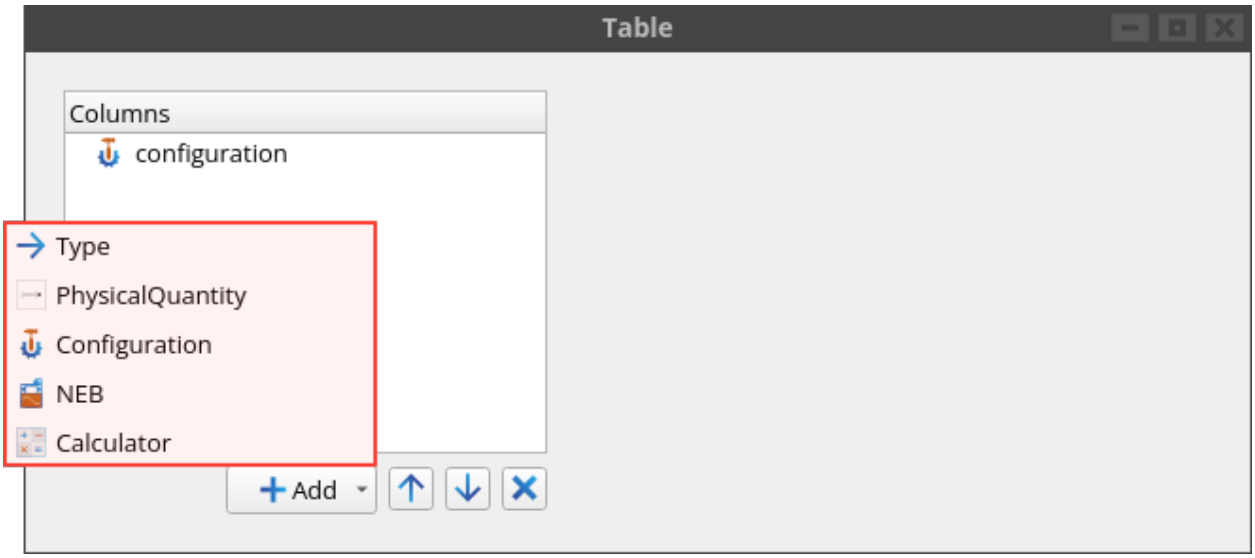


Table
Columns
configuration
Add

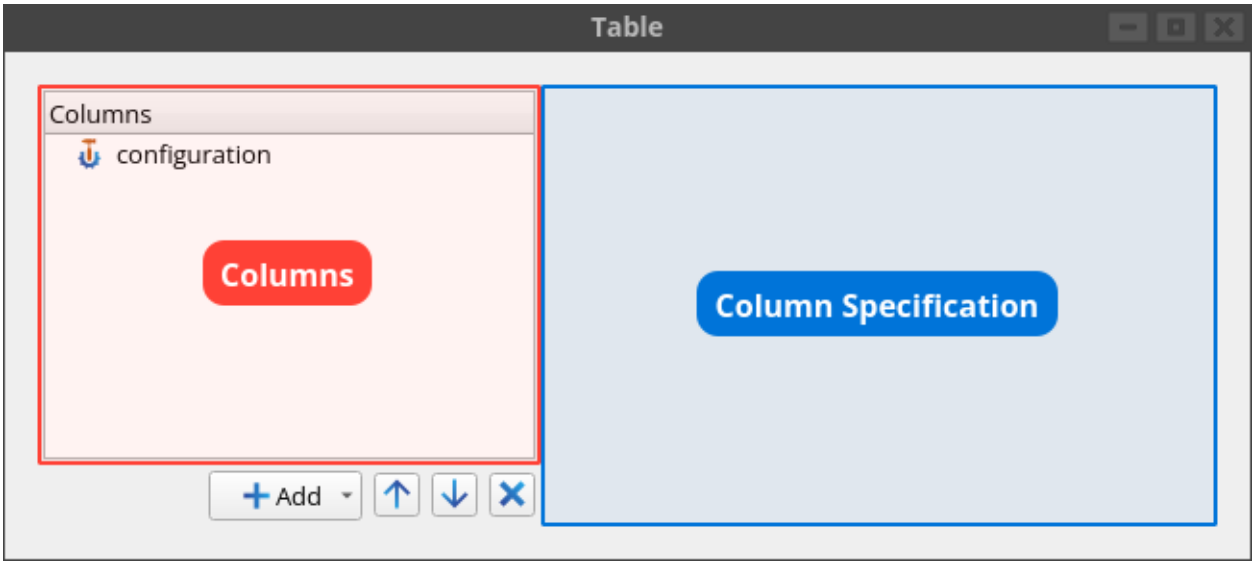
- **NEB**. Collects a `NudgedElasticBand`, and a large array of settings for further specification
- **Calculator**. Collects a Calculator, and a large array of settings for further specification, such as the type of calculator and supported elements.

Note: *Type* should only be used when one of the more specific column options cannot be used. Thus if we want to collect BulkConfigurations, we could do this using the *Type* column, but the *Configuration* column, allows you to specify more information, such as if the configuration have spatial regions, etc.

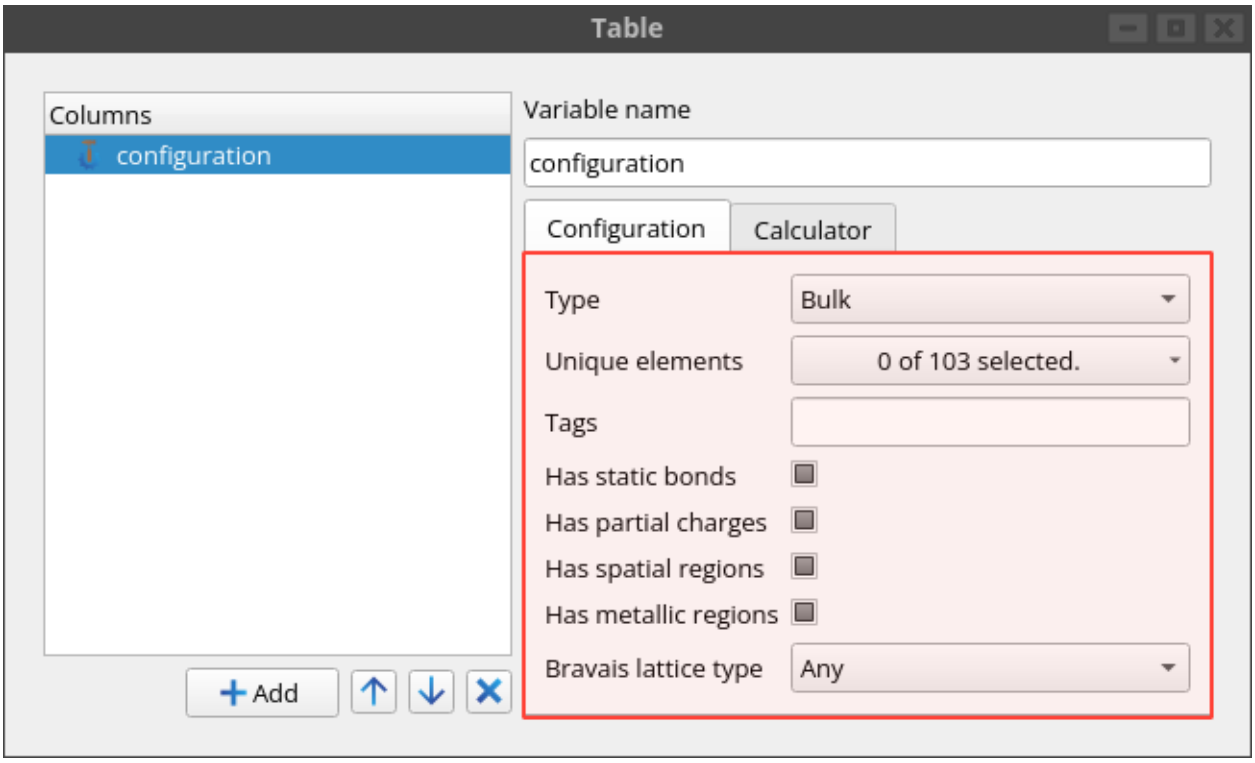
Select *Configuration* in the menu, in order to specify a column with a `BulkConfiguration`. We will later in this guide also use the *Type*, and the *PhysicalQuantity* column options.



Once the *Configuration* column have been added it should be visible in the list of columns, and it can be selected. Once selected the settings for the *Configuration* column become visible on the right in the *Column Specification*.



Specify `BulkConfiguration` as the configuration type in the widget.



The resulting workflow will place the provided bulk configuration into the table, resulting in a table with a single column and a single row.

Naturally this is a very simple example that is not very useful on its own. So in the next example we will go through a bigger more useful example.

How to collect data in a table using an Iteration block

A standard use case for the Table is to collect results using an *Iteration* block `iteration_block_with_table.hdf5`.

In this example we first define an *Iteration* block, where we loop over a k-point sampling parameter. The goal in this examples to determine if a k-point sampling of (9,9,9) is converged.

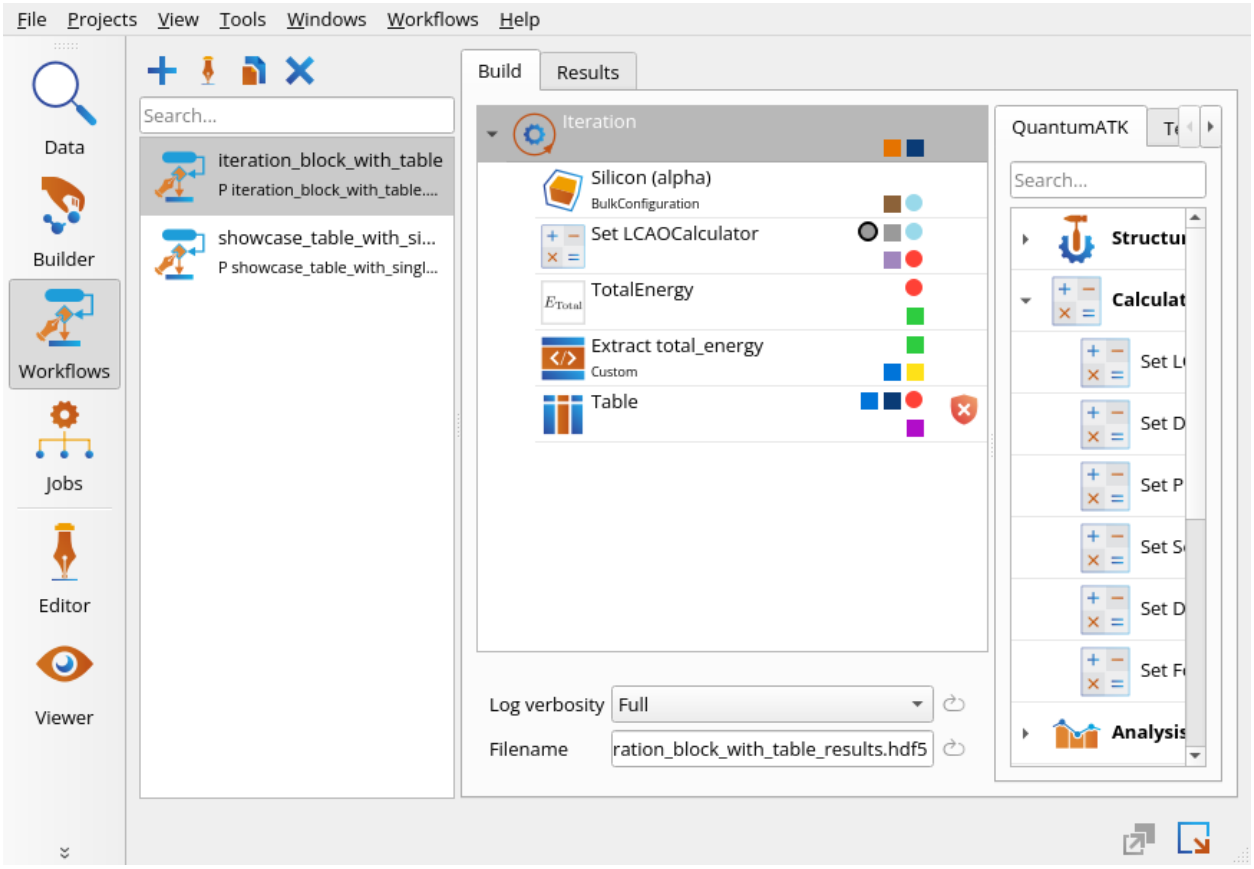
To specify the k-point sampling parameter we open the editor for the *Iteration* block. We click on the *Add* button to add an iterator to the *Iteration* block.

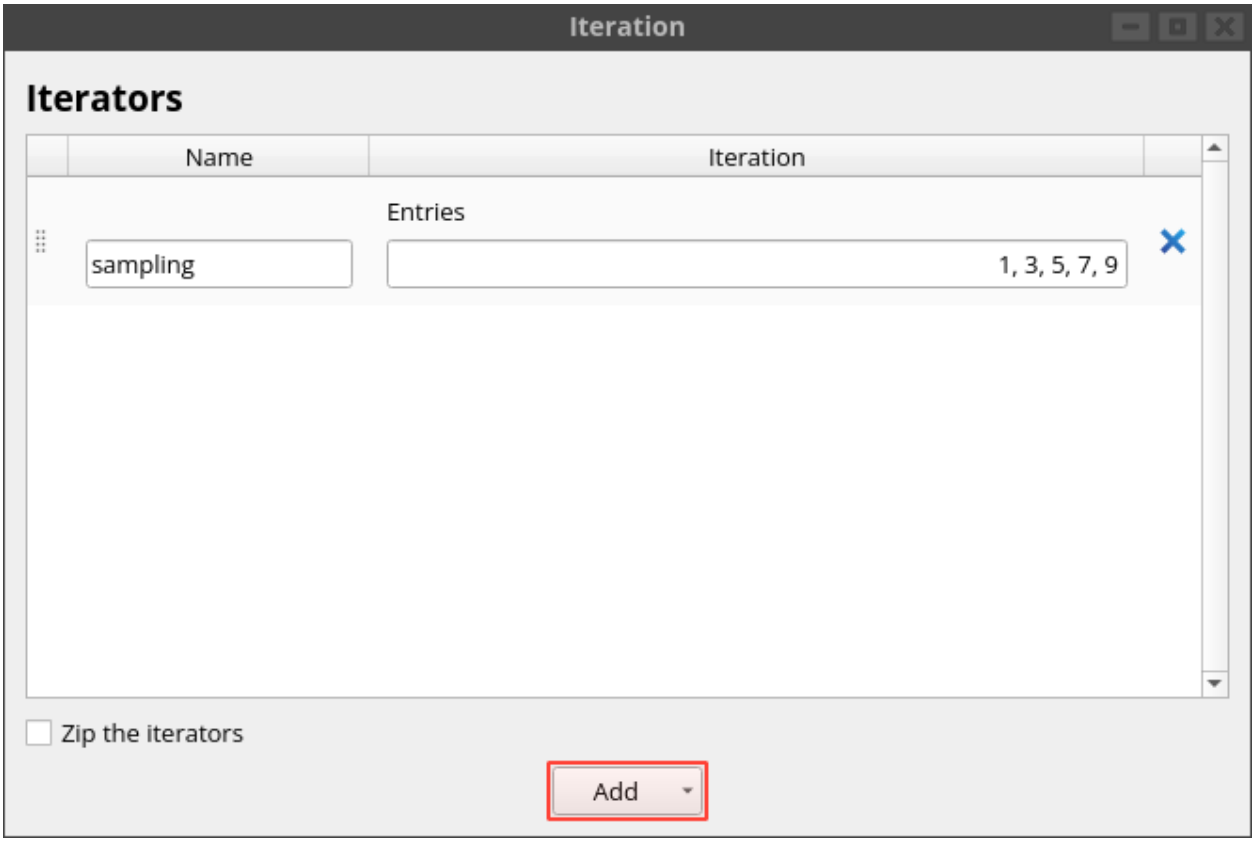
In this case we select the *List of integers* in the menu.

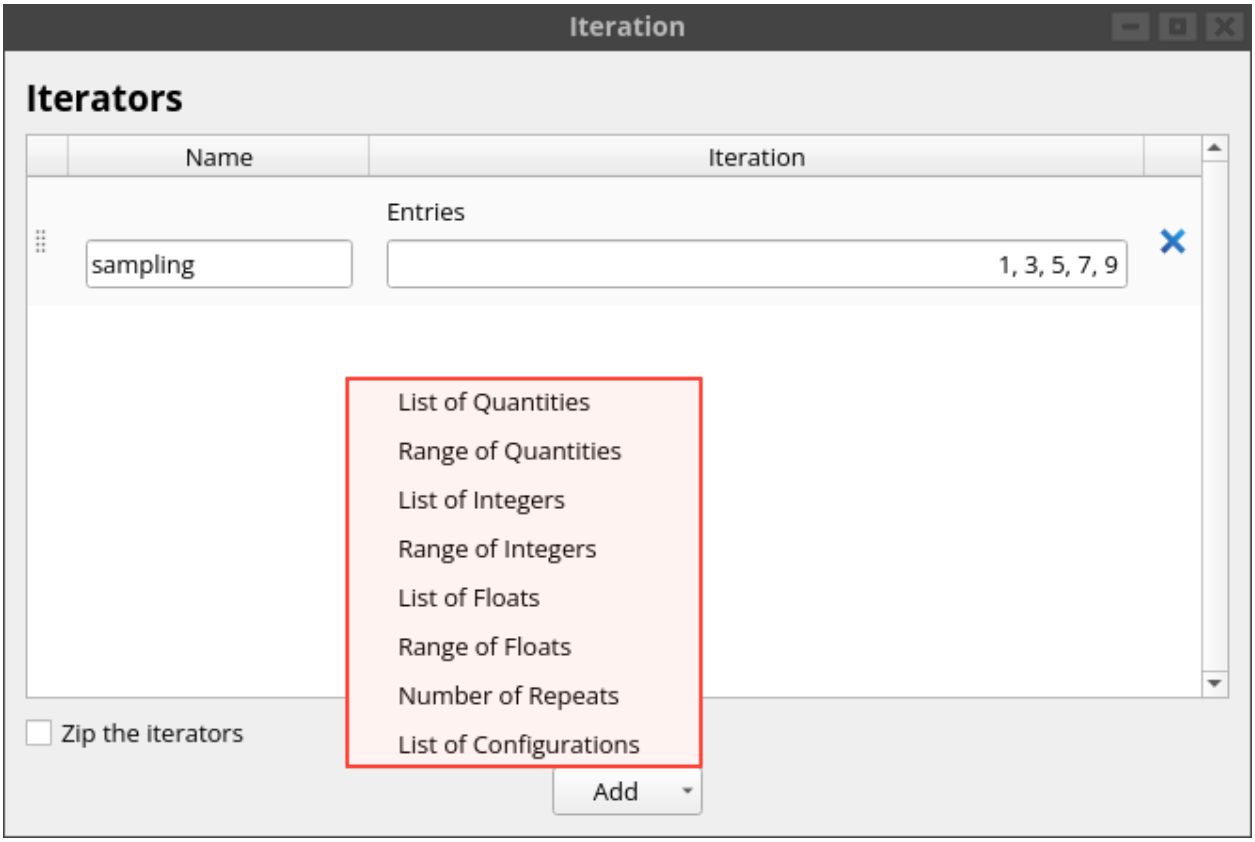
Fill in the name and values of the iterator. In this case we want to loop over the values (1,3,5,7,9).

Tip: *Range of integers* could also be used. For this example that would mean setting the *Start* to 1, *Step size* to 2, *Steps* to 4, resulting in a *Stop* of 9.

The attentive reader would notice, that the range specification is over determined. Thus changing *Start*, *Step size* or *Steps* will update *Stop*, while changing *Stop* will update *Start*.


File Projects View Tools Windows Workflows Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Search...
iteration_block_with_table
P iteration_block_with_table....
showcase_table_with_si...
P showcase_table_with_singl...
Build
Results
Iteration
Silicon (alpha)
BulkConfiguration
Set LCAOCalculator
TotalEnergy
Extract total_energy
Custom
Table
Log verbosity
Full
Filename
ration_block_with_table_results.hdf5
QuantumATK
Search...
Structu
Calculat
Analysis


Iteration
Iterators
Name
Iteration
Entries
sampling
1, 3, 5, 7, 9
Zip the iterators
Add



The *Iteration* block, works as a *Block of Blocks* (see the *Block of blocks*) block and in this case it consist of 5 blocks:

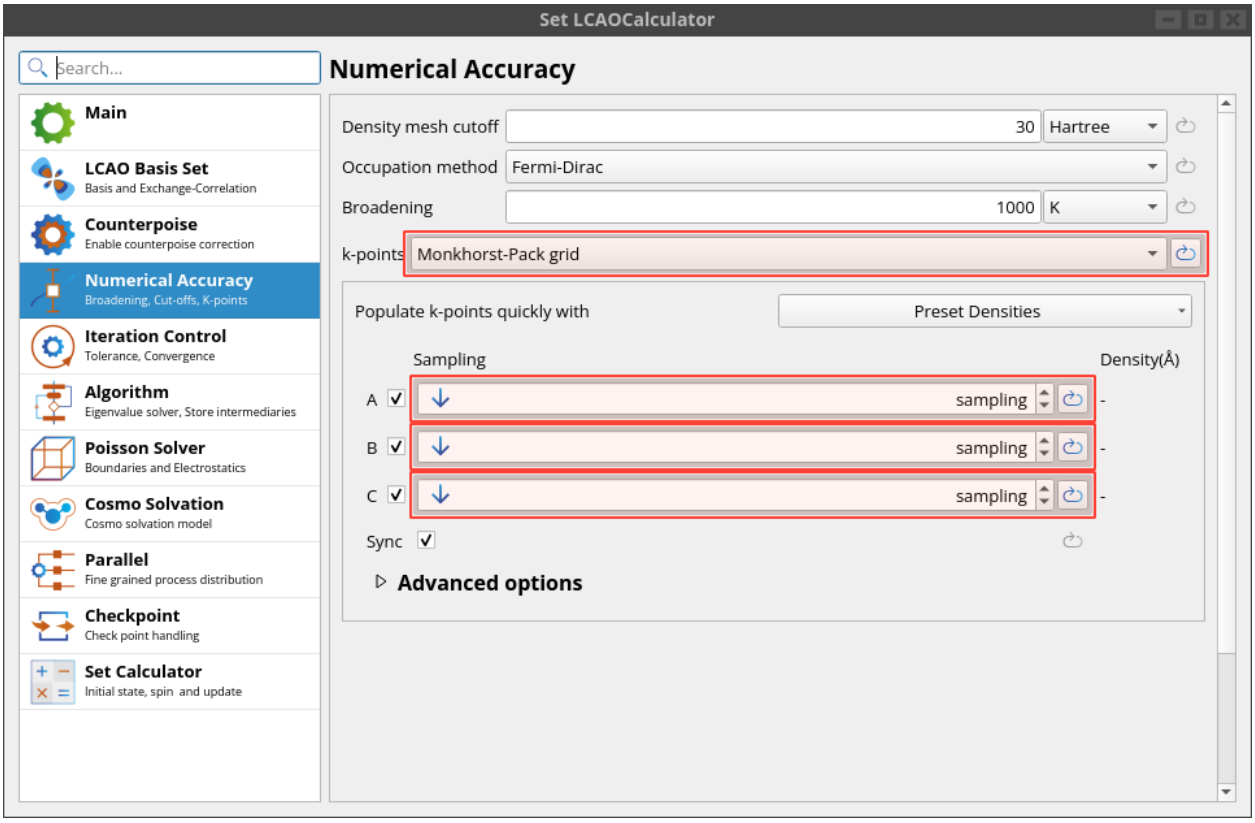
- **Configuration**. This will create a configuration.
- **Set LCAOCalculator**. This will create a `LCAOCalculator`, set it on the configuration and update the configuration.
- **TotalEnergy**. Calculates a `TotalEnergy` analysis object.

E_{Total}

- **Extract total_energy**. A Custom block that extract the total energy in eV from the TotalEnergy analysis object.
- **Table**. The table block.

First we modify the LCAO calculator to use the iterator. First a Monkhorst-Pack grid is selected and then the iterator *sampling* is selected using the in the *A*, *B*, *C* sampling fields

Secondly, we use a Custom block to extract the total energy from the `TotalEnergy` analysis object (see the *What is a workflow block and how to edit it*). The *Custom* blocks are very powerful and flexible blocks. They support



insertion of custom QuantumATK scripts, thus allowing Custom blocks to serve as an interface to a complex calculation setup.

To extract the total energy from the `TotalEnergy` analysis object we must specify in the *Inputs* tab that the *Custom* block requires a `TotalEnergy` analysis object as input.

In the *Outputs* tab we need to specify that the *Custom* block returns a quantity in units of eV.

Finally in the *Script* tab we write the script to extract the total energy from the *TotalEnergy* object.

Running the calculation results in a table, that can be viewed in the *Table Viewer*. (The *Table Viewer* can be open using the data tool *Analyzers and Preferences*)

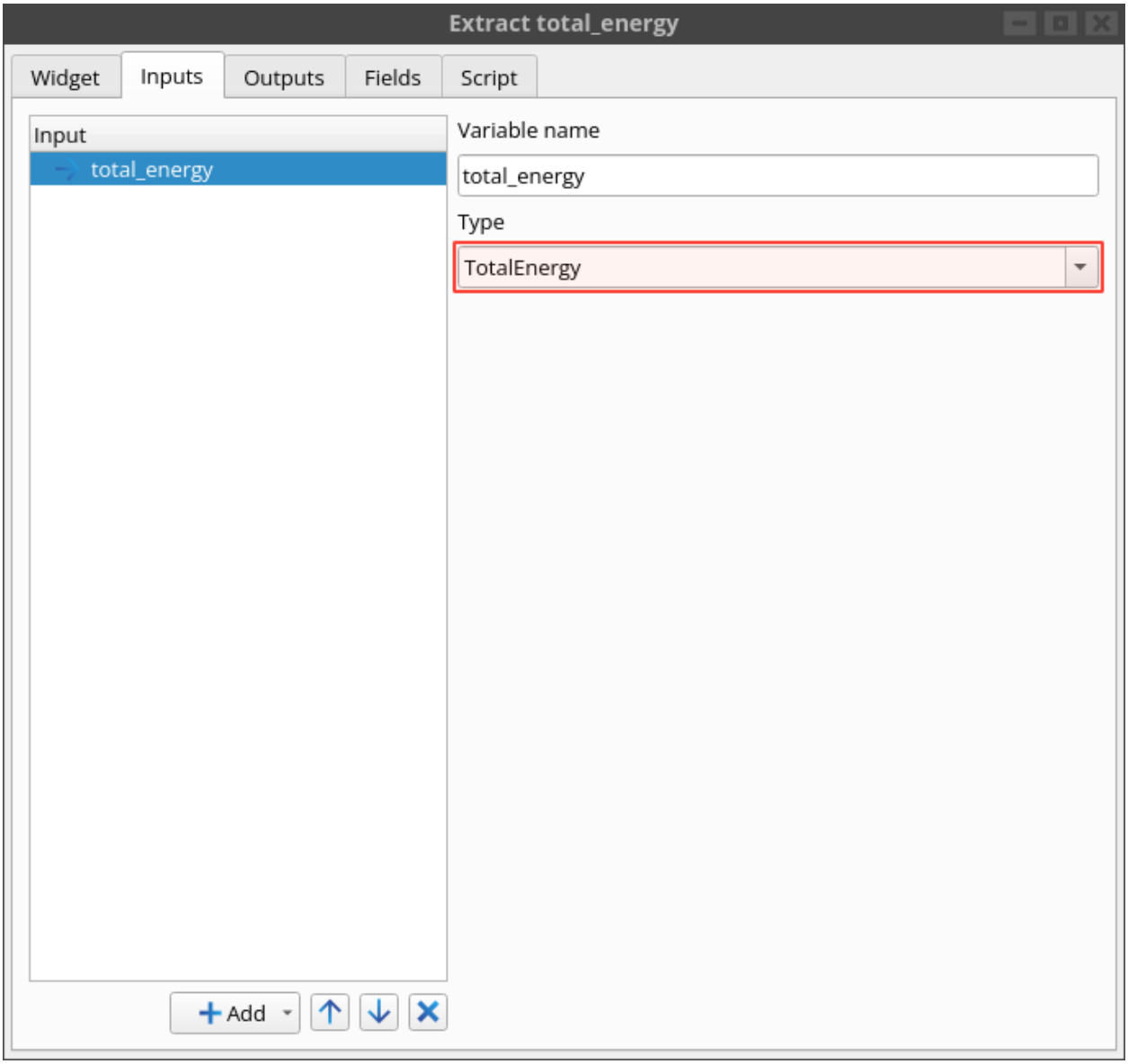
It is fairly easy to see that the total energy is converged to within 10 meV (or at least 4 significant digits).

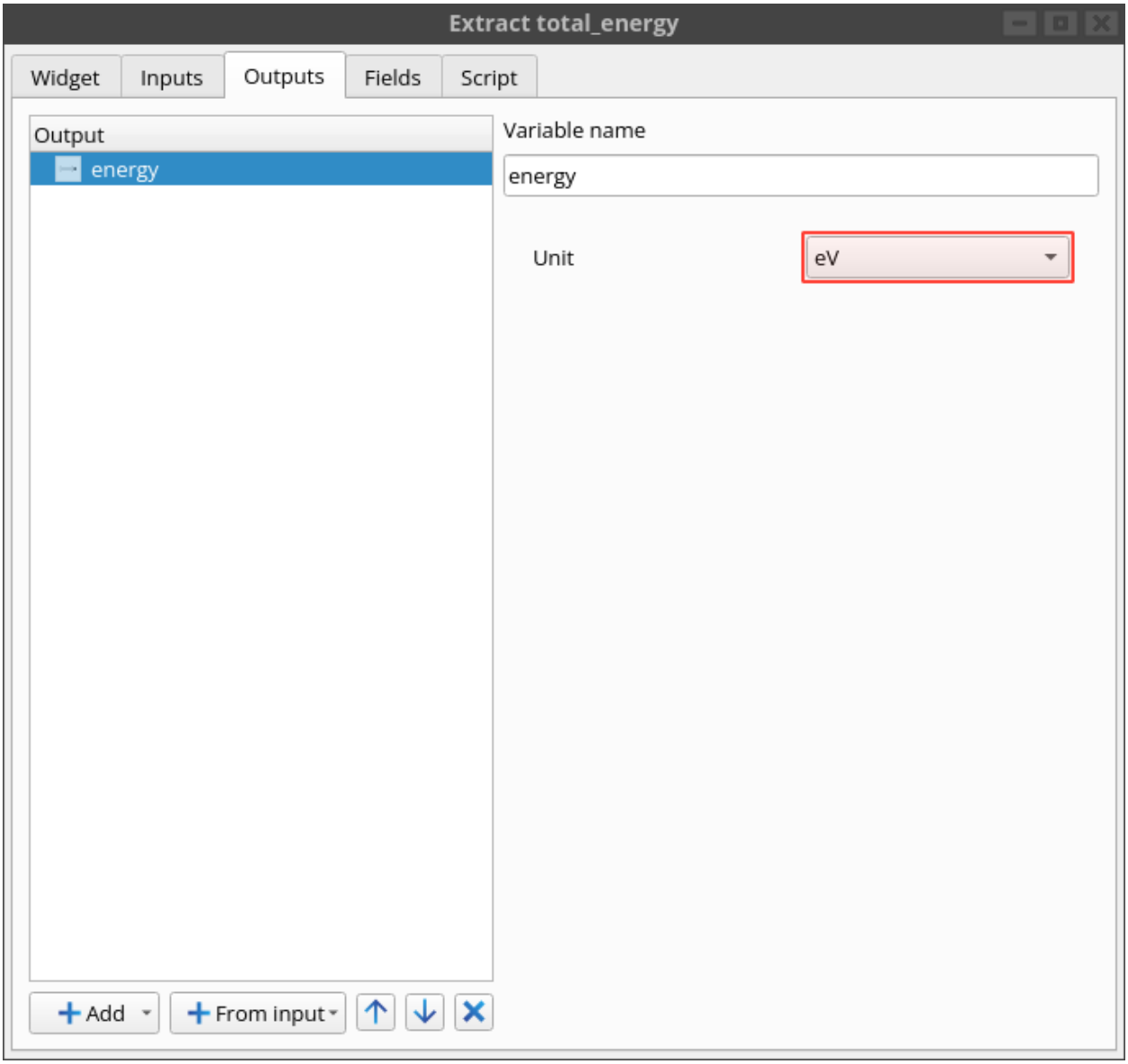
How to extract data from a table using a TableIteration block

Imagine the scenario where you have just done the previous calculation, but realize that you also need to check that the direct bandgap is converged.

You can now load the previous Table using a *Load from file* block, and use the *TableIteration* to calculate the bandgap for each row of the table `table_iteration.hdf5`.

Note: When the *Load from file* block is used to load a Table from file. It will recognize it, and adapt the name and the icon of the *Load from file* block to *Load Table* with the Table icon. *Load from file*


Extract total_energy
Widget
Inputs
Outputs
Fields
Script
Input
total_energy
Variable name
total_energy
Type
TotalEnergy
Add


Extract total_energy
Widget
Inputs
Outputs
Fields
Script
Output
energy
Variable name
energy
Unit
eV
Add
From input

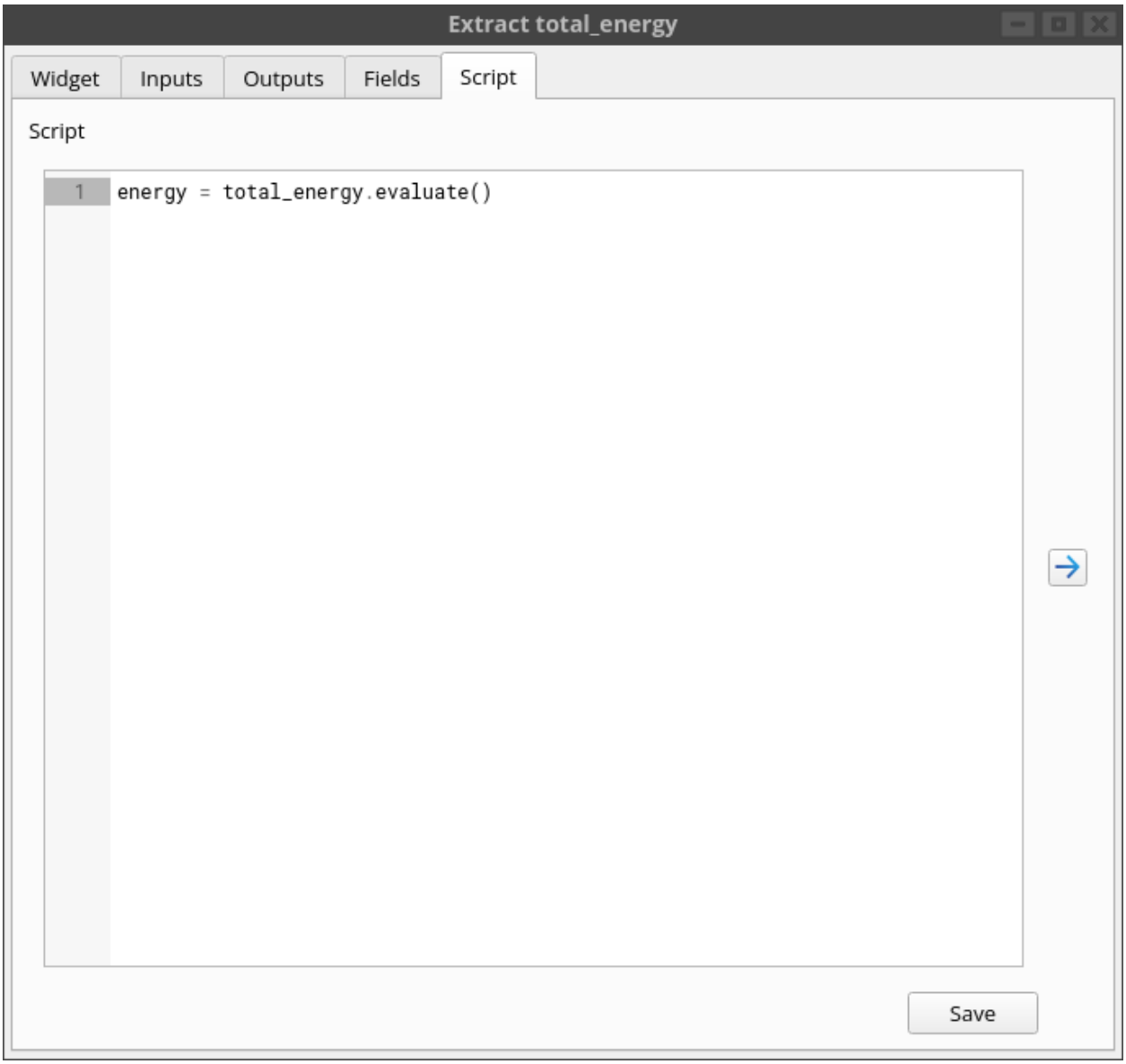
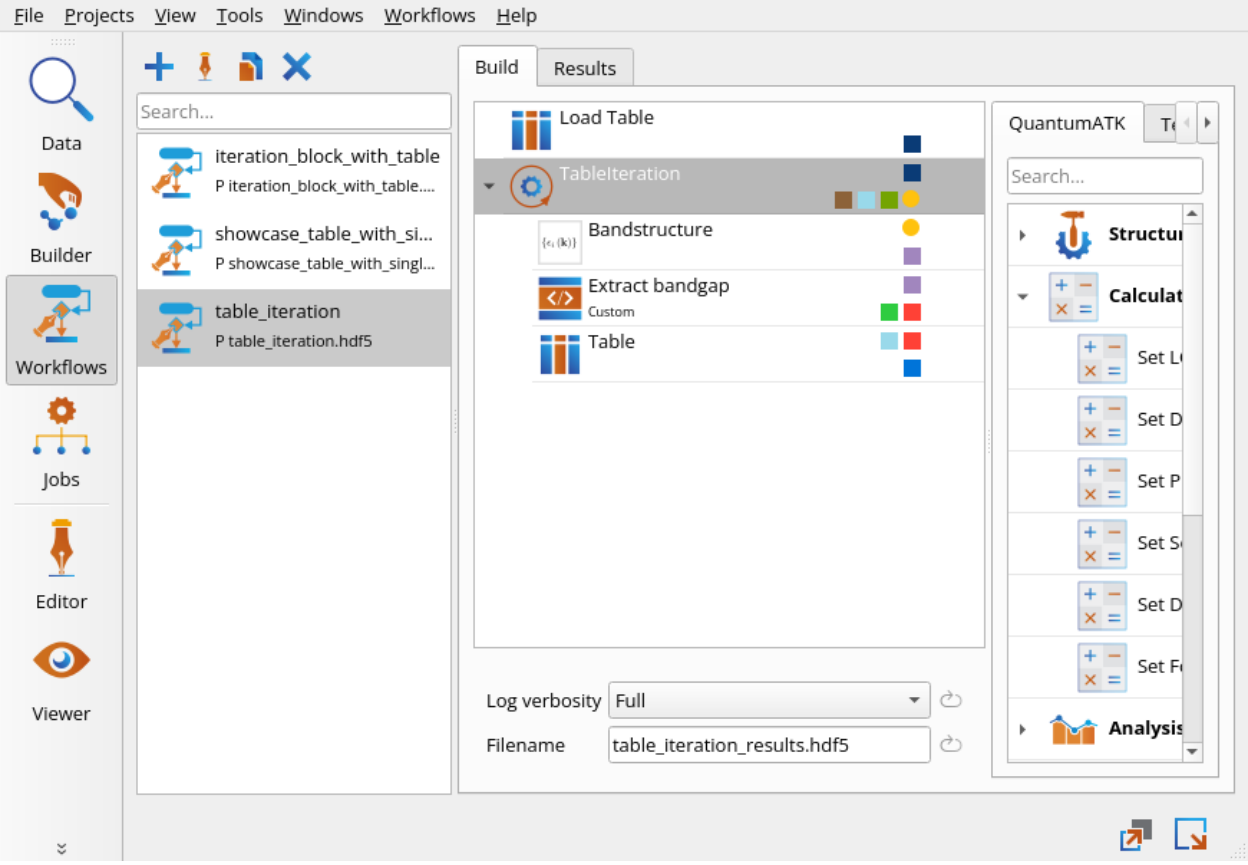


Table - table.hdf5 - table

	Configuration	Sampling	Total_Energy
0	<NL.Com...	1	-212.801968768 eV
1	<NL.Com...	3	-229.411070419 eV
2	<NL.Com...	5	-229.969695432 eV
3	<NL.Com...	7	-230.02498655 eV
4	<NL.Com...	9	-230.032665685 eV



The *Load from file* block is used to load the table. Use the *Load from file* button to specify the table.

The *TableIteration* block takes a table as input and provide the tables row information for each iteration. However, there could be many tables in the workflow and we thus need specify the columns we require to be present in the provided table. In other words we need to align the columns of the *TableIteration* block, with the columns of the Table.

The methodology is similar to specifying the *Table* block. Thus we need to specify a Configuration column, an integer column for the sampling, and a total energy column.

The Configuration column:

The integer column:

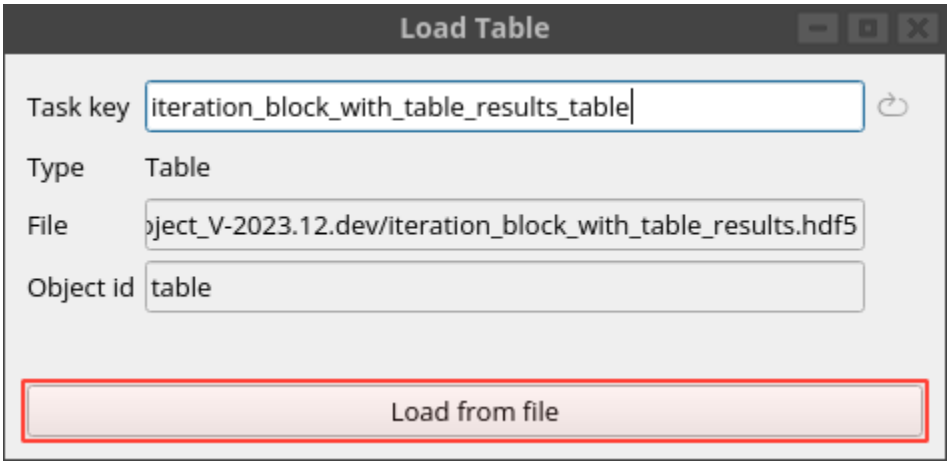
The total energy column:

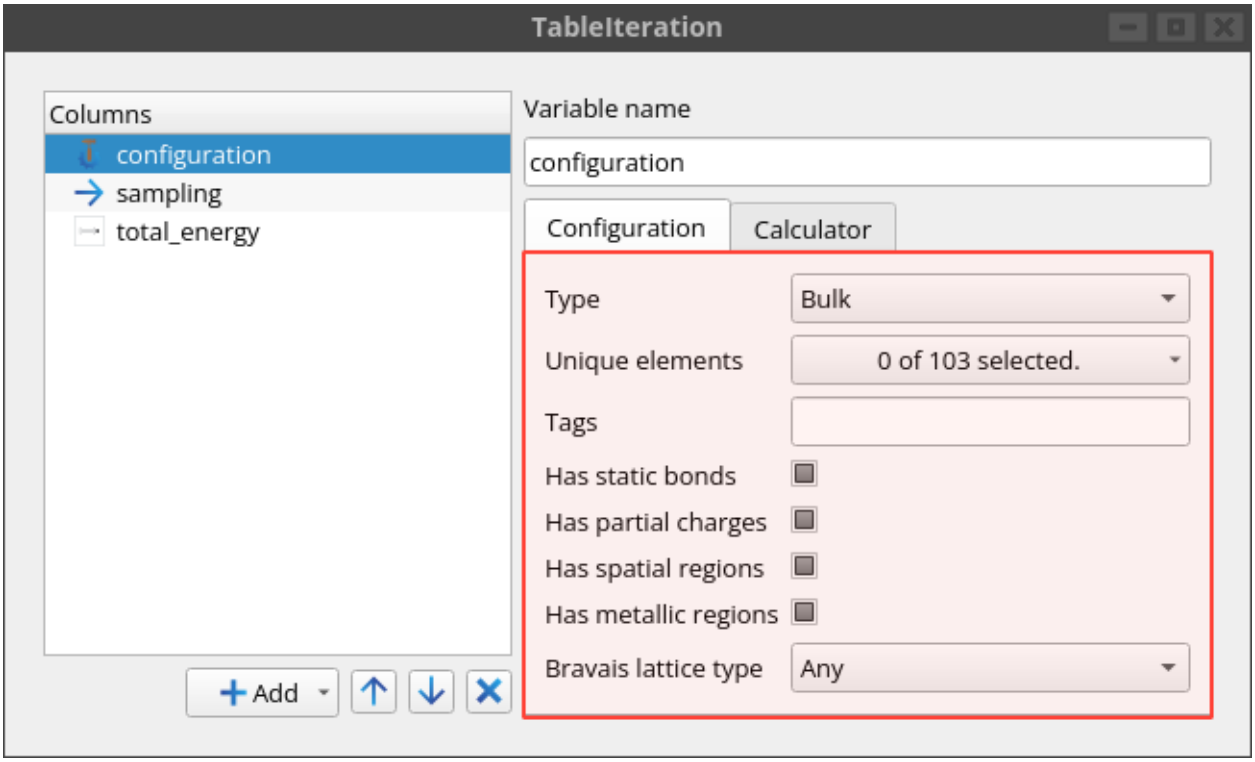
Similar to the last example, we need a Custom block, in this case to extract the direct bandgap of a `Bandstructure` object. Thus we specify the *Script* tab of the Custom block

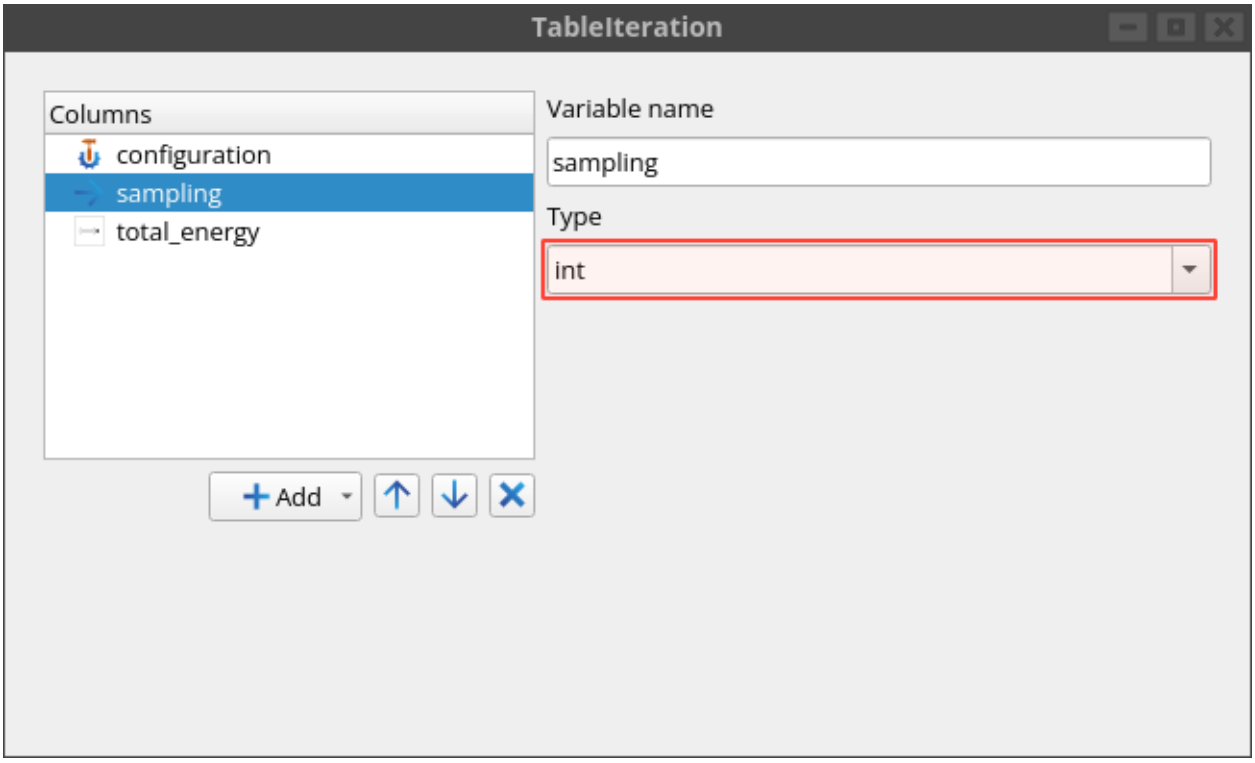
We furthermore need to specify in the *Inputs* tab of the Custom block, that a `Bandstructure` object is required as input.

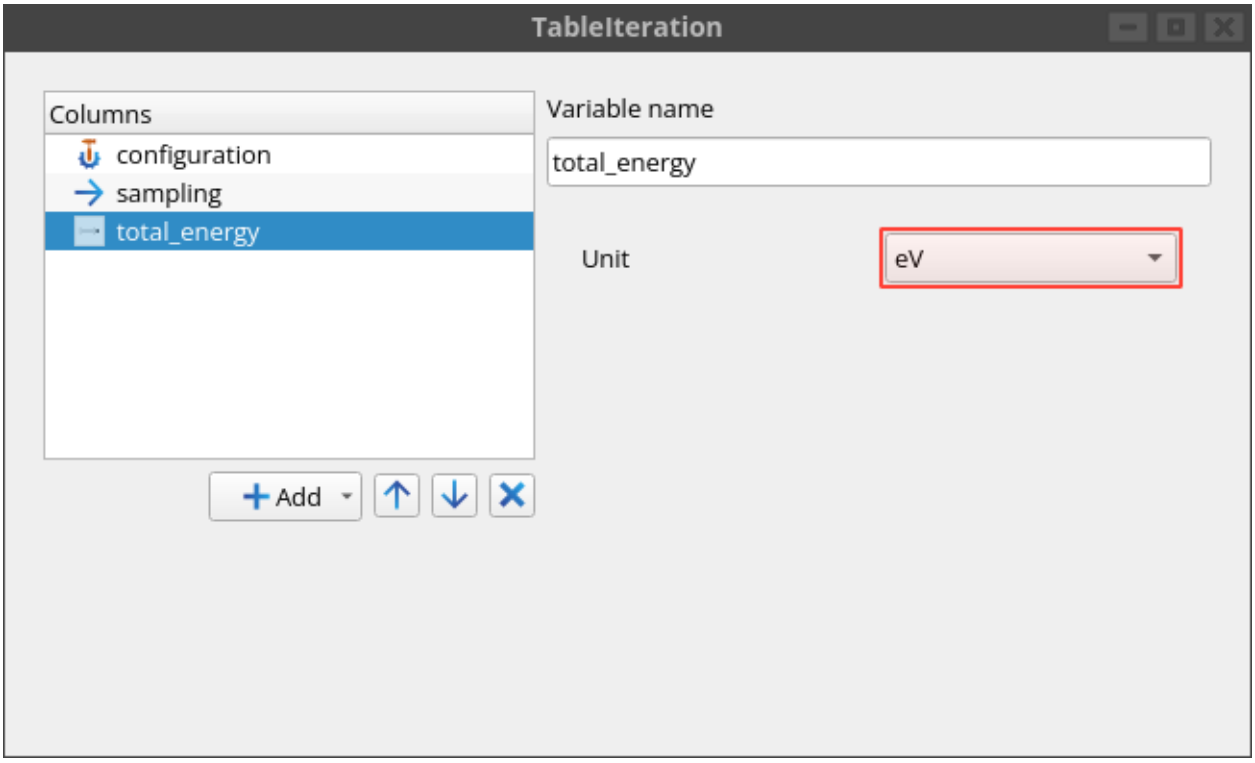
In the *Outputs* tab of the Custom block, we specify that a physical quantity with units of eV is provided as output.

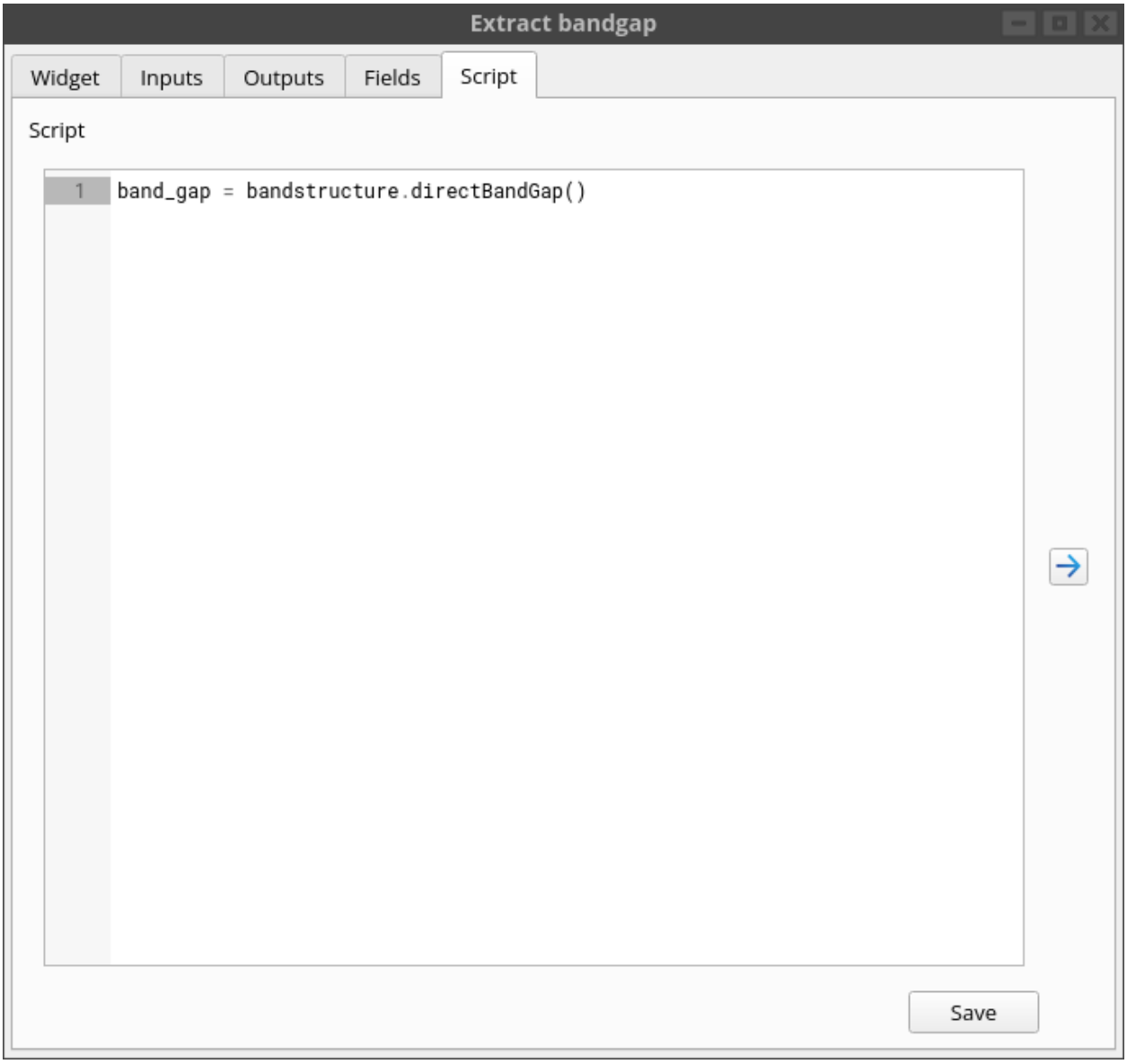
Finally we again collect the sampling and bandgaps in a table. Thus we again need to specify a number of columns to collect the data. In this case, lets collect the bandgap and the sampling.

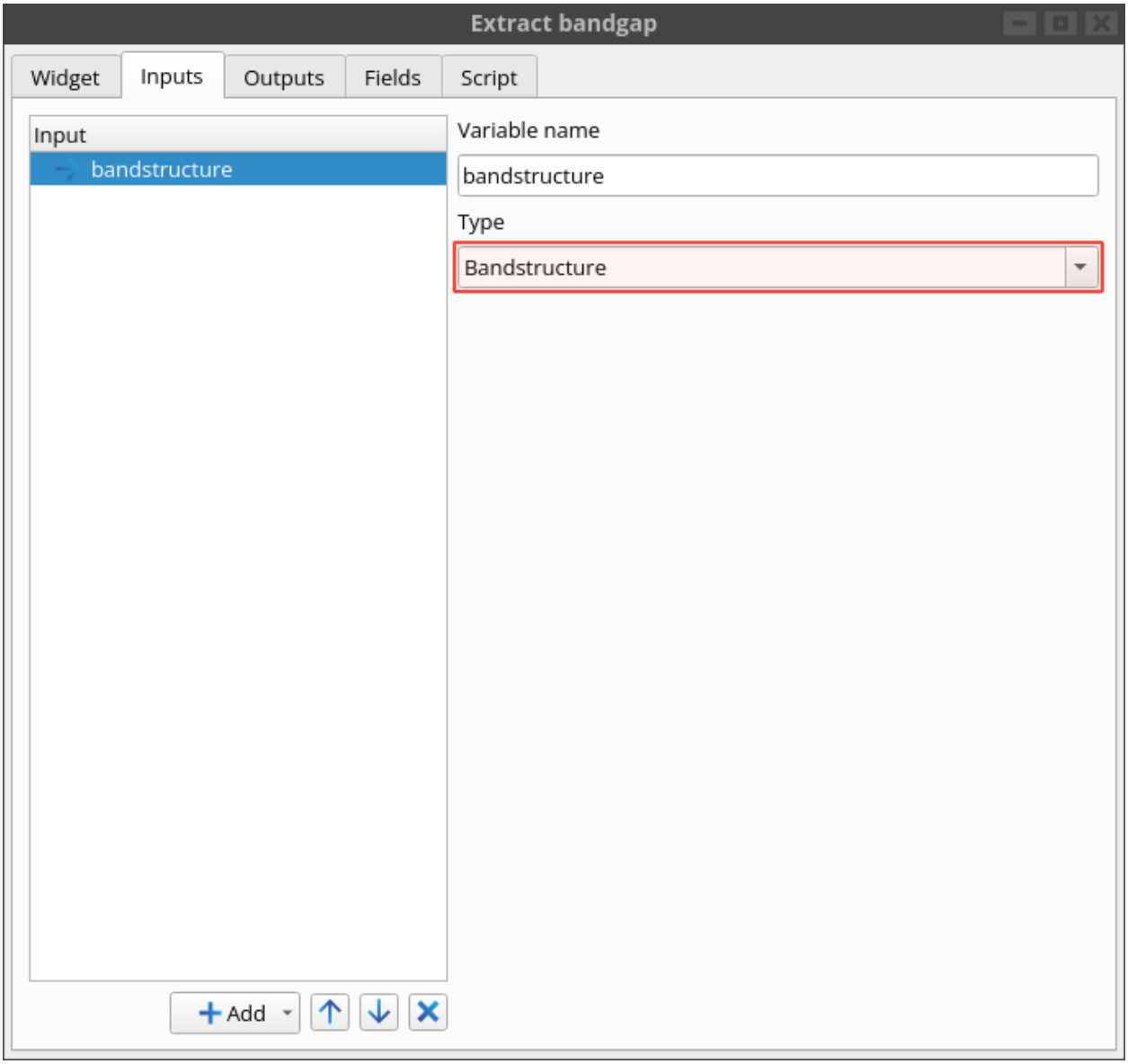

Load Table
Task key
iteration_block_with_table_results_table
Type
Table
File
oject_V-2023.12.dev/iteration_block_with_table_results.hdf5
Object id
table
Load from file

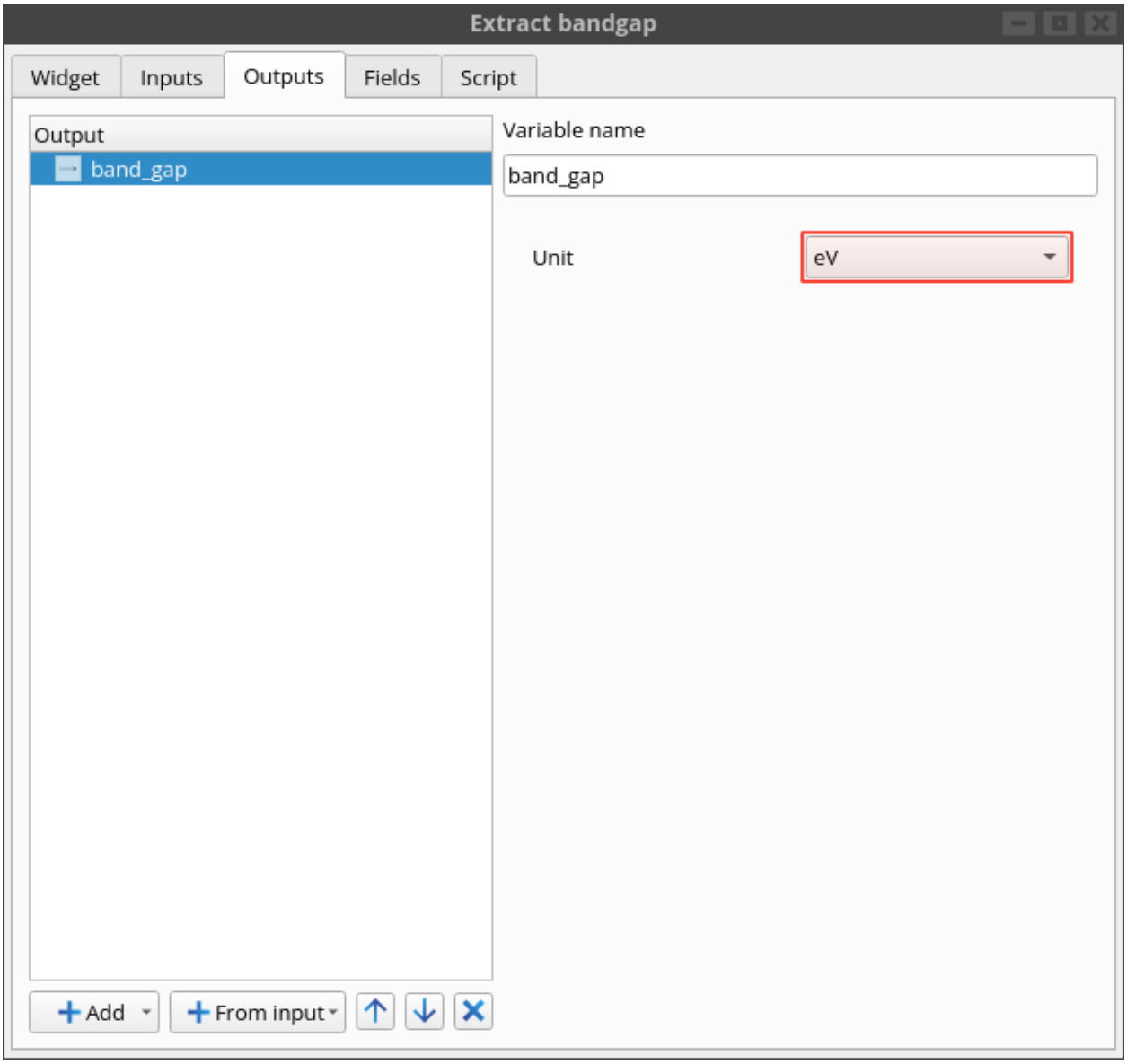

TableIteration
Columns
configuration
sampling
total_energy
Add
Variable name
configuration
Configuration
Calculator
Type
Bulk
Unique elements
0 of 103 selected.
Tags
Has static bonds
Has partial charges
Has spatial regions
Has metallic regions
Bravais lattice type
Any


TableIteration
Columns
configuration
sampling
total_energy
Add
Variable name
sampling
Type
int

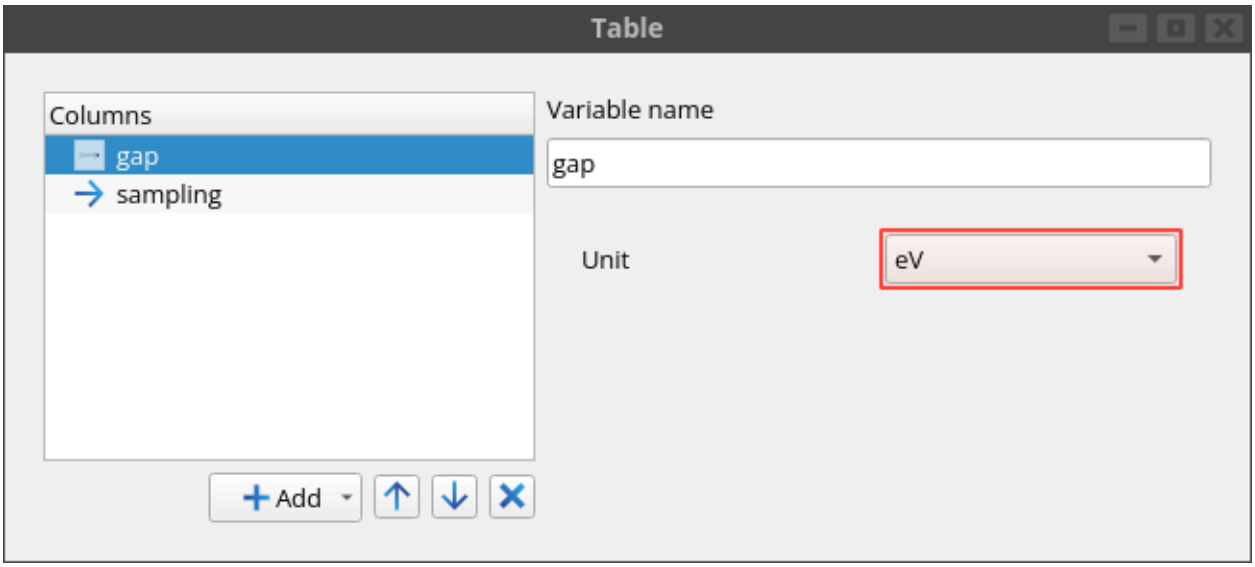

TableIteration
Columns
configuration
sampling
total_energy
Add
Variable name
total_energy
Unit
eV


Extract bandgap
Widget
Inputs
Outputs
Fields
Script
Script
1
band_gap = bandstructure.directBandGap()
Save

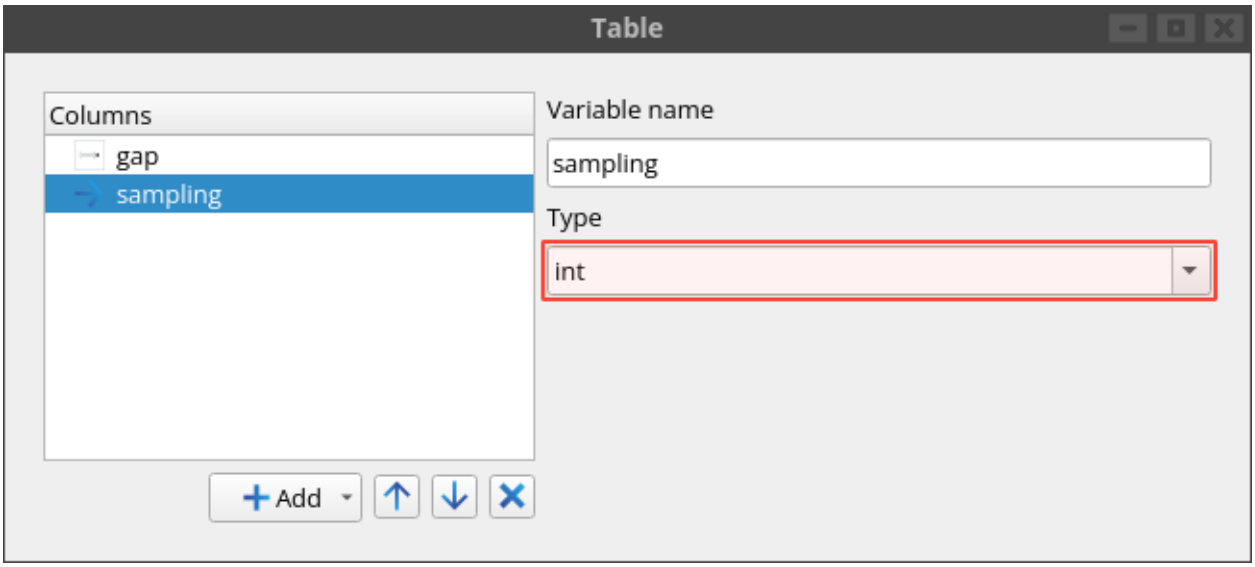

Extract bandgap
Widget
Inputs
Outputs
Fields
Script
Input
bandstructure
Variable name
bandstructure
Type
Bandstructure
Add


Extract bandgap
Widget
Inputs
Outputs
Fields
Script
Output
band_gap
Variable name
band_gap
Unit
eV
Add
From input

The bandgap specification, is a *PhysicalQuantity* Column with the bandgap in units of eV:



The sampling specification, is a *Type* Column with the type specified as integer.



Resulting in the table, where it is clear that the direct bandgap is converged to within 1 meV (atleast 3 significant digits). (The *Table Viewer* can be open using the data tool *Analyzers and Preferences*)

We have now used the *Table* and *TableIteration* blocks, and hopefully shown how convenient, and versatile these can be. You can collect almost anything in a table, you can easily load a table using a *Load from file* block, and you can easily iterate over rows of a table.

Table - table_bs.hdf5 - table

	Gap	Sampling
0	0.0 eV	1
1	2.53600226966 eV	3
2	2.57271991828 eV	5
3	2.57863756444 eV	7
4	2.57976535212 eV	9

2.5.3 Introduction to Array Jobs

Downloads & Links

`array_workflow.hdf5` `array_workflow_results.py` `array_workflow_results.hdf5`

Often, one wants to run many similar calculations in parallel. Perhaps one wants to scan over a specific set of parameters or calculate specific properties for many different structures.

One way to achieve this is to take advantage of the array jobs feature present in all standard HPC queueing systems such as PBS, LFS or SLURM. Instead of performing all the calculations in one single, huge job an array job allows the queuing system to manage the computation in a much more fine-grained way, where each of the many calculations are only submitted in smaller batches whenever sufficient resources are available. This often leads to much faster computation times, since the job does not have to wait for a single large reservation of computing resources to become available.

While running array jobs comes with a a lot of advantages, they are also much harder to set up and submit compared to regular jobs. Also, once the computation has finished, it is also much harder to gather the data of all the independent calculations.

The QuantumATK Workflow Builder and *Jobs* tool now takes away all the hassle of setup, submission and data gathering for array jobs. They are now essentially as easy to set up as normal computation jobs.

This small guide shows you how you can leverage the power of array jobs in QuantumATK.

Table of Contents

•*How to build an array script in the Workflow Builder*

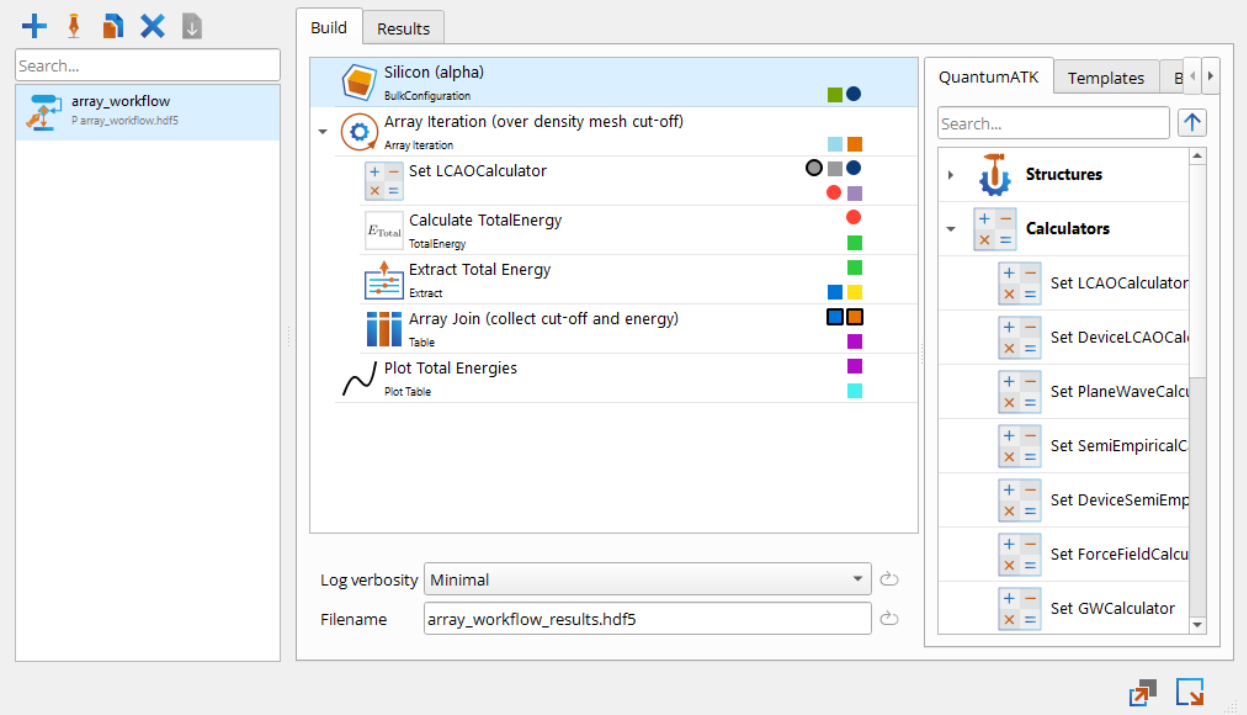
•*How to submit an array script*

How to build an array script in the Workflow Builder

We begin by using the Workflow Builder to set up an array script.

As a quick example, let us imagine that we are interested in Silicon and wish to investigate Silicon using a LCAO-Calcualtor. An important first step is then to converge the density mesh cut-off for the LCAOCalculator. We can do this by calculating the Total Energy of the Silicon system for different values of the density mesh cut-off, and all these independent calculations can be performed in an array job.

Let us set up the following workflow:



The array iteration block creates a density_mesh_cutoff variable that iterates over energies of 10, 20, , 110 Hartree.

Tip: Array jobs can be built using either the Array iteration block or the Array table iteration block.

The density_mesh_cutoff variable is then plugged into the LCAOCalculator block.

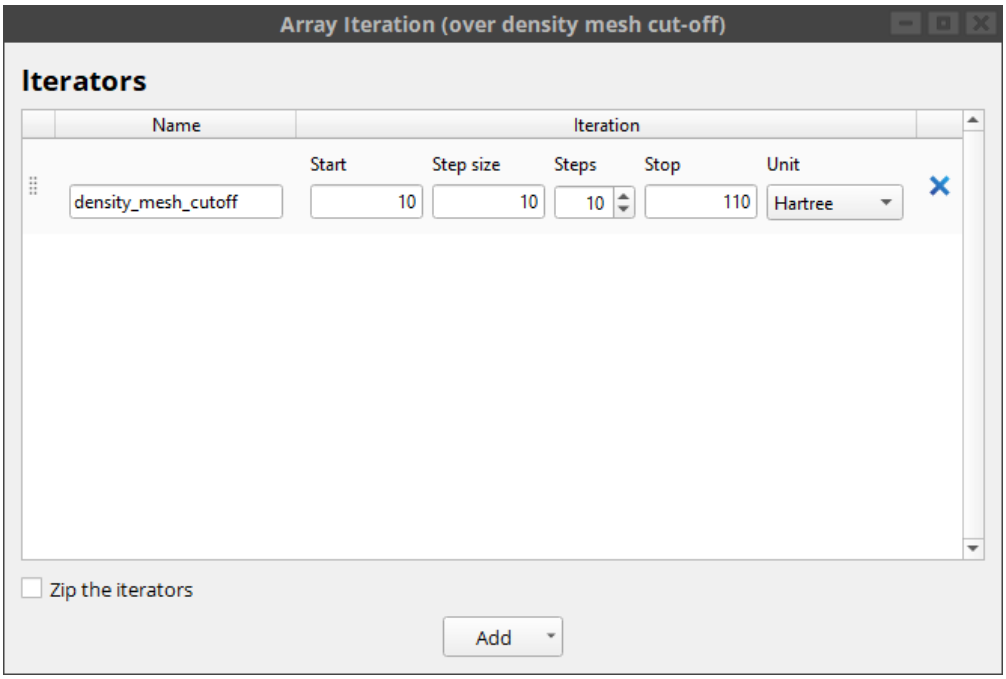
All array jobs must collect their result in a Table. In this case we save both the input density mesh cut-off and the resulting total energy in the table.

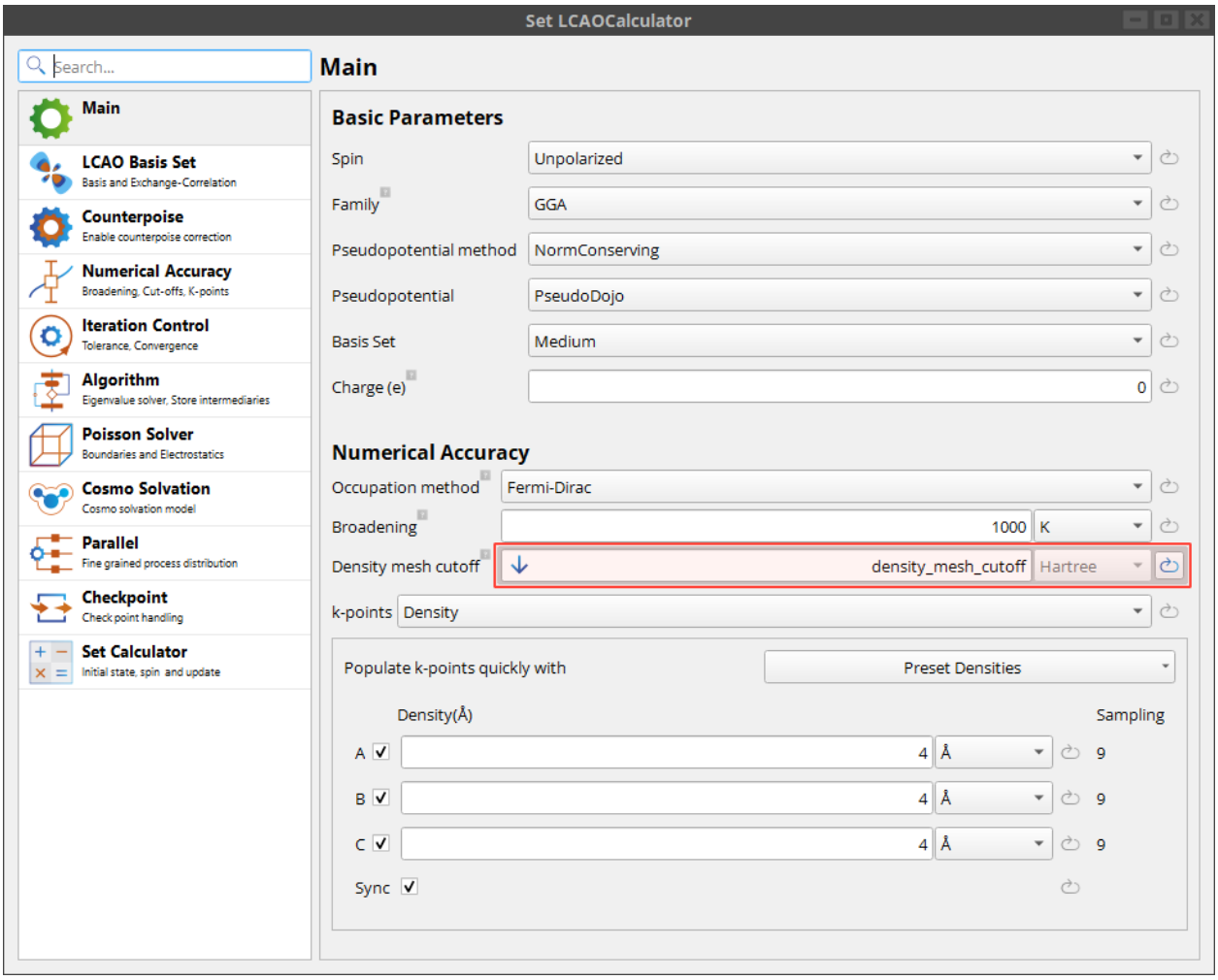
Note that the workflow also contains a plot of the total energy against the density mesh cut-off, so it is straight forward to gauge the convergence.

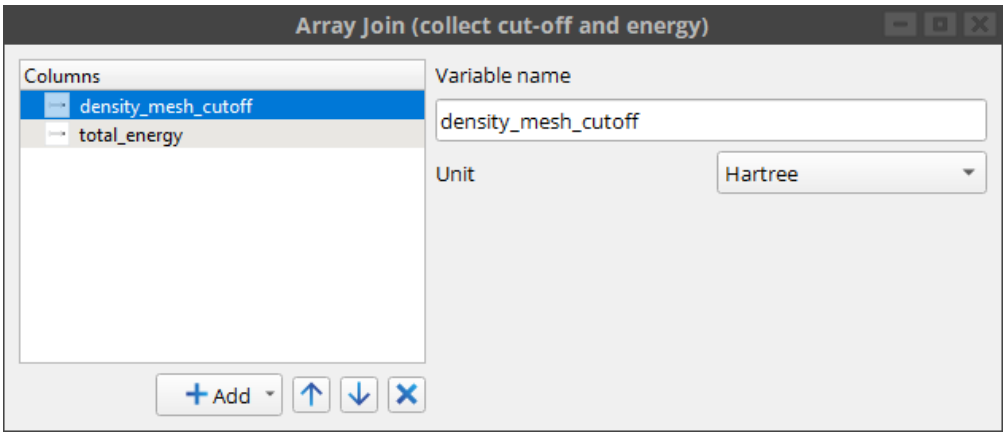
Next, in the lower right corner of the Workflow Builder we then click the Arrow and then select Editor as Array script. The resulting script looks like this:

```
# ARRAY prepare 11 finalize

import itertools

# Set minimal log verbosity
setVerbosity(MinimalLog)
```

(continues on next page)


Array Iteration (over density mesh cut-off)
Iterators
Name
Iteration
Start
Step size
Steps
Stop
Unit
density_mesh_cutoff
10
10
10
110
Hartree
Zip the iterators
Add


Set LCAOCalculator
Search...
Main
LCAO Basis Set
Basis and Exchange-Correlation
Counterpoise
Enable counterpoise correction
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Algorithm
Eigenvalue solver, Store intermediaries
Poisson Solver
Boundaries and Electrostatics
Cosmo Solvation
Cosmo solvation model
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
Main
Basic Parameters
Spin
Unpolarized
Family
GGA
Pseudopotential method
NormConserving
Pseudopotential
PseudoDojo
Basis Set
Medium
Charge (e)
0
Numerical Accuracy
Occupation method
Fermi-Dirac
Broadening
1000
K
Density mesh cutoff
density_mesh_cutoff
Hartree
k-points
Density
Populate k-points quickly with
Preset Densities
Density(Å)
Sampling
A
4
Å
9
B
4
Å
9
C
4
Å
9
Sync



(continued from previous page)

```
results_filename = 'array_workflow_results.hdf5'

# Extract array part (if present).
part = os.environ.get('PART')
common_folder = os.getcwd()

# Common setup.
if part in ('prepare', None):
    # Set up lattice
    lattice = FaceCenteredCubic(5.4306*Angstrom)

    # Define elements
    elements = [Silicon, Silicon]

    # Define coordinates
    fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                              [ 0.25,  0.25,  0.25]]

    # Set up configuration
    silicon_alpha = BulkConfiguration(
        bravais_lattice=lattice,
        elements=elements,
        fractional_coordinates=fractional_coordinates
        )

    silicon_alpha_name = "silicon_alpha"

    nlsave(results_filename, silicon_alpha, object_id='silicon_alpha')

# Parallelize.
if part in ('main', None):
    if part == 'main':
        # Extract array specifics.
        folder = os.environ['ARRAY_FOLDER']
        array_index = int(os.environ['ARRAY_INDEX'])
        array_size = int(os.environ['ARRAY_SIZE'])
```

(continues on next page)

(continued from previous page)

```
    else:
        folder = '.'
        array_index = None
        array_size = None

    # Move to the array folder.
    os.chdir(folder)

    silicon_alpha = nlread(
        os.path.join(common_folder, results_filename), object_id='silicon_alpha'
    )[0]

    # Array Iteration (over density mesh cut-off)(preparation)
    ############################################################
    #          Array Iteration (density_mesh_cutoff)           #
    ############################################################

    # Create iterator.
    def densityMeshCutoffsIterator():
        for i in range(11):
            v = (10.0 + i * 10.0) * Hartree
            yield v

    density_mesh_cutoffs = densityMeshCutoffsIterator()

    def arraySplit(generator, length, array_size, array_index):
        """
        Determine the start and end indices to distribute an generator over
        an array of calculations.

        :param length:          The length of the generator.
        :type  length:          int

        :param array_size:      The array length.
        :type  array_size:      int

        :param array_index:     The array index.
        :type  array_index:     int

        :returns:               An generator over the split indices.
        :rtype:                 iterator
        """
        # Fallback if no array parameters were set.
        if None in (array_size, array_index):
            return generator

        size = length // array_size
        extra = length % array_size

        start = array_index * size + min(array_index, extra)
        stop = (array_index + 1) * size + min(array_index + 1, extra)
```

(continues on next page)

(continued from previous page)

```
        return itertools.islice(generator, start, stop)

    for density_mesh_cutoff in arraySplit(
        density_mesh_cutoffs, 11, array_size, array_index
    ):

        # %% Set LCAOCalculator

        # %% LCAOCalculator

        k_point_sampling = KpointDensity(
            density_a=4.0 * Angstrom, density_b=4.0 * Angstrom, density_c=4.0 * Angstrom
        )

        numerical_accuracy_parameters = NumericalAccuracyParameters(
            density_mesh_cutoff=density_mesh_cutoff, k_point_sampling=k_point_sampling
        )

        calculator = LCAOCalculator(
            numerical_accuracy_parameters=numerical_accuracy_parameters,
            checkpoint_handler=NoCheckpointHandler,
        )

        # %% Set Calculator

        silicon_alpha.setCalculator(calculator)

        silicon_alpha.update()

        # %% Calculate TotalEnergy

        calculate_total_energy = TotalEnergy(configuration=silicon_alpha)

        # %% Extract Total Energy

        def extract_total_energy(total_energy):
            evaluate = total_energy.evaluate()
            return evaluate

        evaluate = extract_total_energy(calculate_total_energy)

        # %% Array Join (collect cut-off and energy)

        if 'array_join_collect_cutoff_and_energy' not in locals():
            array_join_collect_cutoff_and_energy = Table(
                results_filename, object_id='table'
            )
            array_join_collect_cutoff_and_energy.addQuantityColumn(
                key='density_mesh_cutoff', unit=Hartree
            )
            array_join_collect_cutoff_and_energy.addQuantityColumn(
                key='total_energy', unit=eV
```

(continues on next page)

(continued from previous page)

```
            )
            array_join_collect_cutoff_and_energy.setMetatext(
                'Array Join (collect cut-off and energy)'
            )

        array_join_collect_cutoff_and_energy.append(density_mesh_cutoff, evaluate)

# Collect all results into common table.
if part in ('collect', 'finalize'):
    # Extract array specifics.
    array_folder_prefix = os.environ['ARRAY_FOLDER_PREFIX']
    array_size = int(os.environ['ARRAY_SIZE'])

    # Collect all tables into one.
    if part == 'finalize':
        array_join_collect_cutoff_and_energy = Table(results_filename)
    else:
        array_join_collect_cutoff_and_energy = Table('partial_' + results_filename)

    for array_index in range(array_size):
        path = os.path.join(array_folder_prefix + str(array_index), results_filename)

        try:
            # Read the partial table.
            partial_table = nlread(path, object_id='table')[0]

            # Extend main table.
            array_join_collect_cutoff_and_energy.extend(partial_table)
        except Exception:
            print(f'No file found for index {array_index} at {path}.')

# Post-process results.
if part in ('collect', 'finalize', None):

    # Create PlotModel.
    plot_model = Plot.PlotModel(x_unit=Hartree, y_unit=meV)

    plot_model.framing().setTitle('Plot_Total_Energies')

    # Add line
    Total_energy = Plot.Line(
        array_join_collect_cutoff_and_energy[:, 'density_mesh_cutoff'],
        array_join_collect_cutoff_and_energy[:, 'total_energy'],
    )
    Total_energy.setLabel('Total energy')
    Total_energy.setColor('#b82832')
    Total_energy.setLineStyle('-')
    Total_energy.setMarkerStyle('')
    plot_model.addItem(Total_energy)

    # Auto-adjust axis limits.
    plot_model.setLimits()
```

(continues on next page)

(continued from previous page)

```
    # Save plot to file.
    Plot.save(plot_model, results_filename)
```

This script effectively splits the calculation up into three different parts. First a common prepare part that sets up the silicon structure, then an array part that calculates the total energy for the different density mesh cut-offs and lastly a finalize part, in which the array results are collected and plotted.

This script is specifically adapted to work with the QuantumATK *Jobs* tool. However, it is also made in such a way that it will still work when submitted as a normal job. Running it like a normal job will of course be a lot less efficient, as it will run entirely in a single process. It is also possible to prepare your own array scripts for submission with the *Jobs* tool by using e.g. the example script above as a template.

Once the script is created, save it in preparation for submitting it in the *Jobs* tool.

How to submit an array script

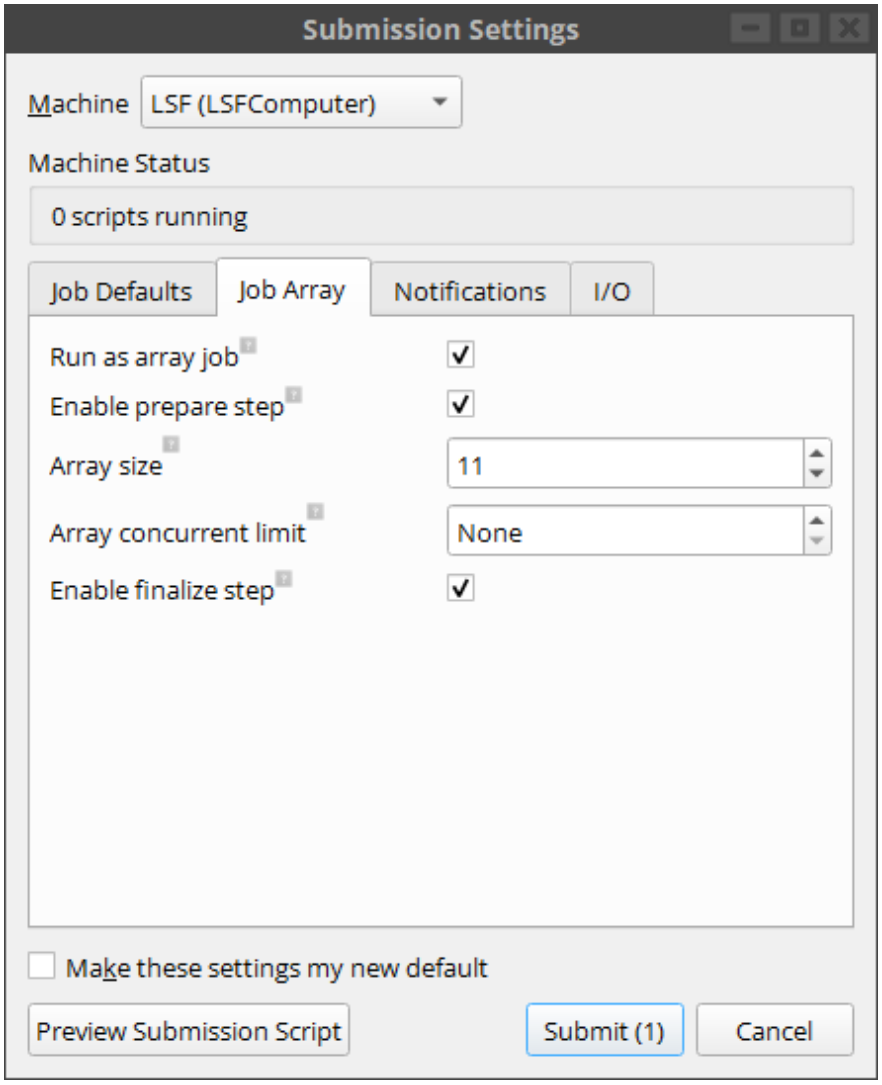
Start by sending the array script to the *Jobs* tool. In the job creation dialog choose a machine with a queueing system, e.g. LSF, SLURM or PBS.

The submission dialog contains a specific tab for array jobs settings. Note that this tab has already been filled out with the relevant information taken from the first comment line in the array script.

Creating the job and pressing submit, we can follow the progress of the job in log panel. In our case it would look like this:

```
+-------------------------------------------------------------------------------+
| Total Array Jobs                                                              |
+-------------------------------------------------------------------------------+
| Total jobs:                 13                                                |
| Queued jobs:                 1                                                |
| Running jobs:                7                                                |
| Completed jobs:              5                                                |
+-------------------------------------------------------------------------------+

+-------------------------------------------------------------------------------+
| Progress                                                                      |
+-------------------------------------------------------------------------------+
| Running      [**************************                      ]               |
| Completed    [******************                              ]               |
+-------------------------------------------------------------------------------+

+-------------------------------------------------------------------------------+
| Job status                                                                    |
+-------------------------------------------------------------------------------+
| Prepare                    Done                                               |
| Main 0                     Done                                               |
| Main 1                     Done                                               |
| Main 2                  Running                                               |
| Main 3                  Running                                               |
| Main 4                  Running                                               |
| Main 5                  Running                                               |
| Main 6                  Running                                               |
```

(continues on next page)

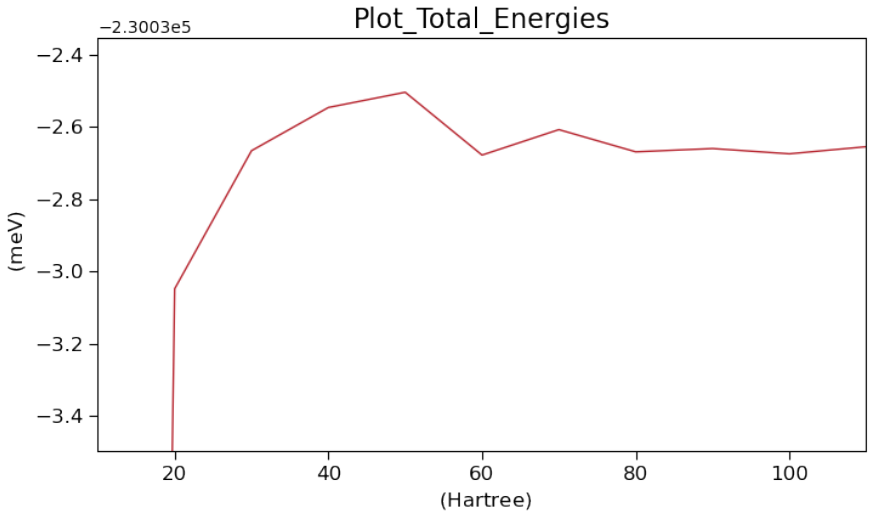

Submission Settings
Machine
LSF (LSFComputer)
Machine Status
0 scripts running
Job Defaults
Job Array
Notifications
I/O
Run as array job
Enable prepare step
Array size
11
Array concurrent limit
None
Enable finalize step
Make these settings my new default
Preview Submission Script
Submit (1)
Cancel

(continued from previous page)

```
| Main 7                    Running                                                  |
| Main 8                    Running                                                  |
| Main 9                       Done                                                  |
| Main 10                      Done                                                  |
| Finalize                   Queued                                                  |
+-----------------------------------------------------------------------------------+
```

Tip: The array log files are each put in separate folders, and they can be inspected in the special log files tab.

When the calculations have finished all the main results are downloaded. This includes all outputs from the main part of the workflow, i.e. all results form the prepare and finalize parts and the table from the array part. The created plot looks like this, showing a convergence to less then 1 meV at around 30 Hartree.



2.6 Jobs manual

2.6.1 Job Manager for local execution of QuantumATK scripts

Version: T-2022.03

Downloads & Links

`silicon.py` `cnt.py` `mpi_check.py`

Table of Contents:

•*Execute QuantumATK simulations via the Job Manager*

•*Serial execution*

•*Threading*

•*MPI parallelization*

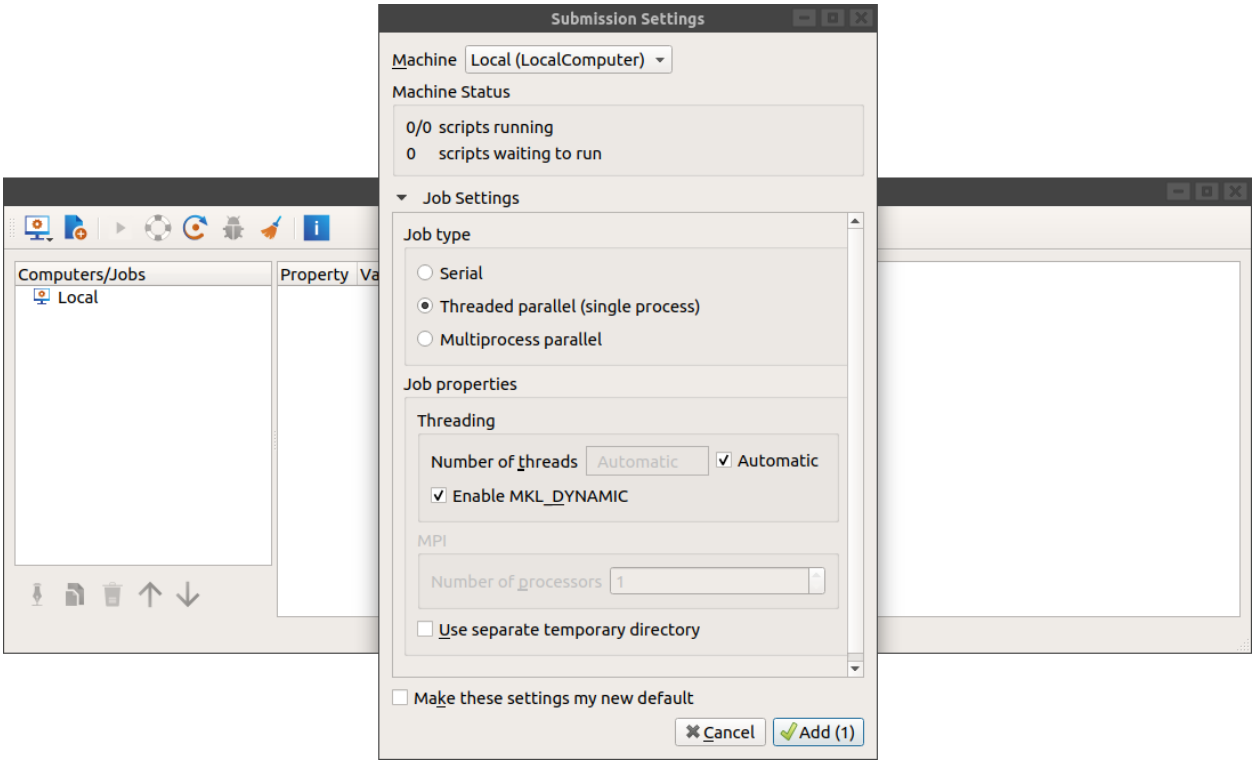
•*Adding Computers*

Execute QuantumATK simulations via the Job Manager

In this section you will learn how to use the **Job Manager** for local execution of QuantumATK scripts. Specifically, you will learn about queuing, running and managing QuantumATK jobs.

Create a new empty project and download the example script `silicon.py`, which runs an ATK-DFT calculation with very many k-points (31x31x31).

Drop it on the **Job Manager** and select a local computer for the job execution.



The **Submission Settings** widget has two basic panels:

•**Job type;**

–*Serial*

–*Threaded parallel (single process)*

–*Multiprocess parallel*

•**Job properties;**

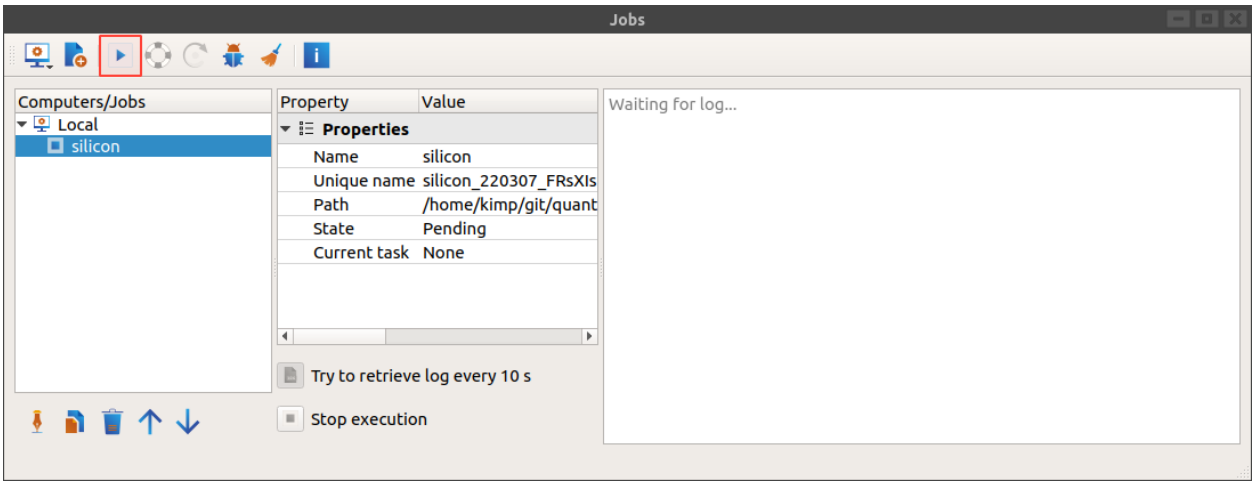
–Threading;

–MPI Parallelism;

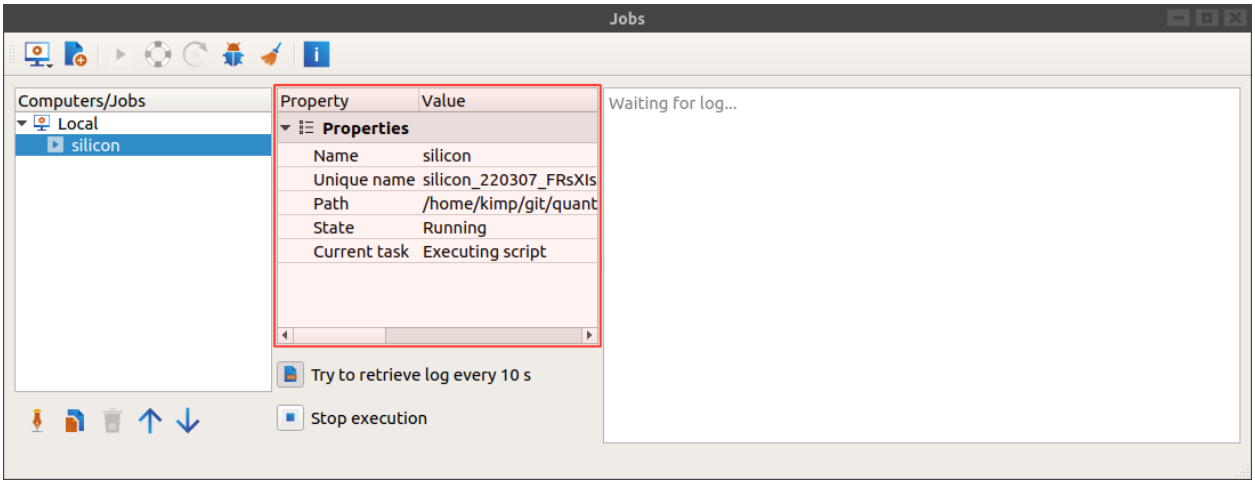
–**Use separate temporary directory** (set this to allow using a non-default working directory for running the job. Note, results will not be appended to existing hdf5 or nc files, but instead a new file will be created.)

Set these settings according to your needs, and click **Add**.

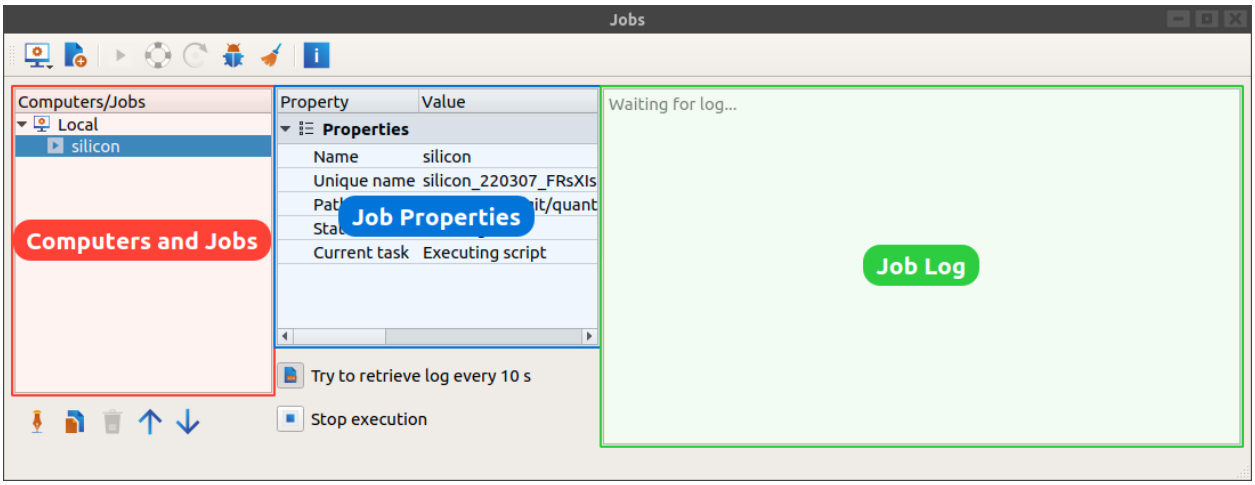
In the **Job Manager**, first select the job. Then click the highlighted **Run** button to start the job.



The task state changes from Pending to Running.



The Job Manager has three main parts. The overview of **computers and jobs** highlighted in red, the **properties of the job highlighted** in blue, and **the log** highlighted in green.



The properties of the job is a Property–Value list which shows all details of the settings used for job execution, including:

•Name of the Python script.

•State.

•Current task.

The job finishes after about 1 minute (2.5 GHz CPU). Note that the task state changes to Finished.

The job output (calculation results) of course appears in the QuantumATK **DataView** after job execution.

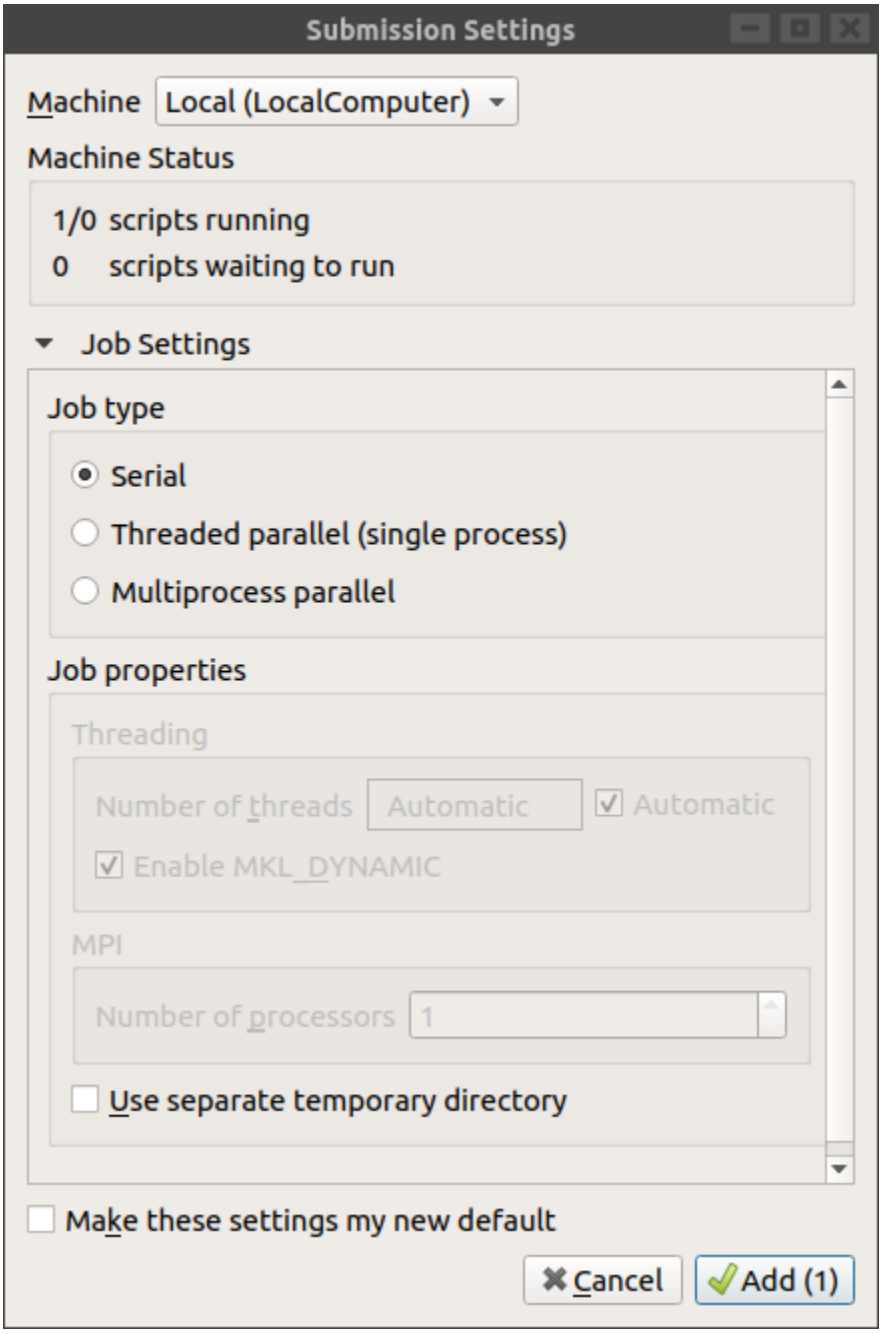
Note: Remember that the default job type is Threaded parallel (Single process). You can change this to Serial or Multiprocess parallel before starting the job.

Use the **Trash** button to remove jobs from the job queue.

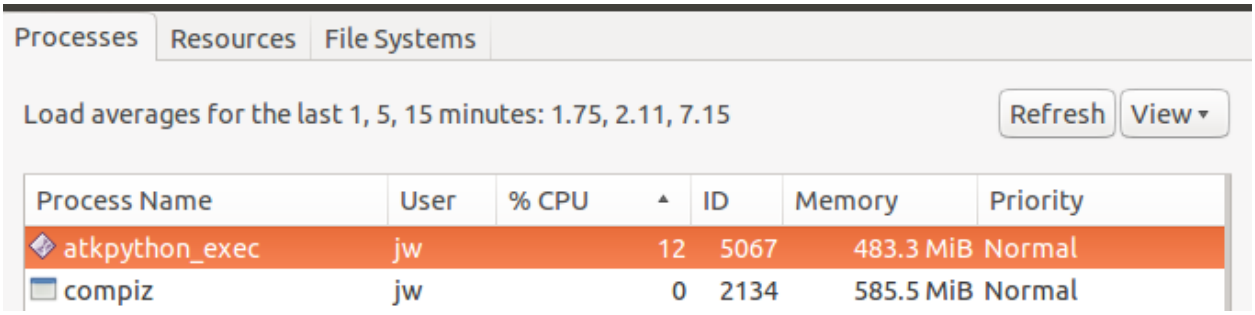
Serial execution

In the **Submission Settings** window select a **Serial** job type as shown in the below figure to run on a single process with no threading.

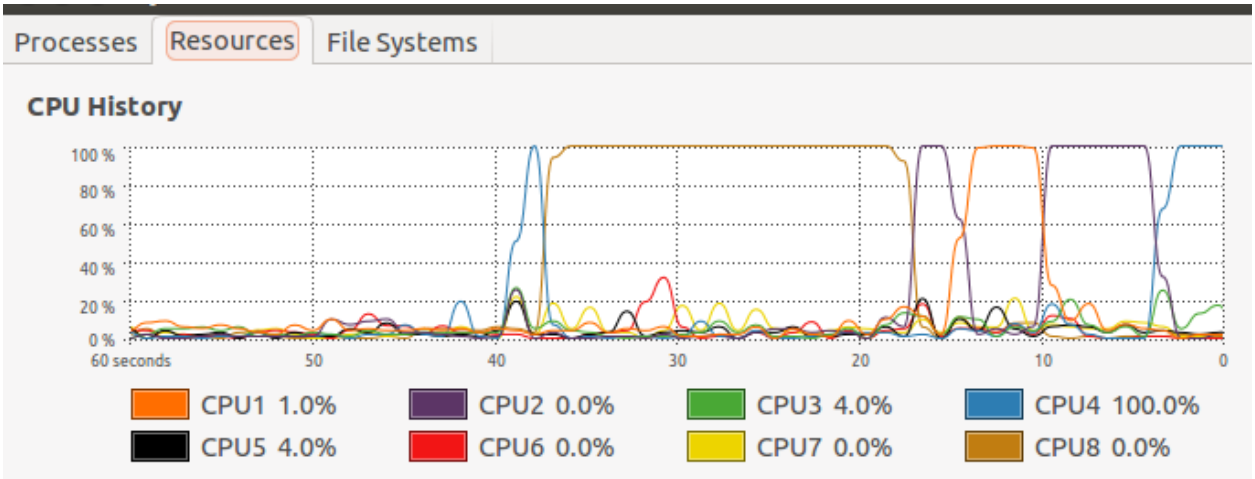
In fact, note that threading and MPI parallelization are not available.



If you check the system load during local execution in serial, you should see that the serial job launches only a single computing task on a single CPU core.

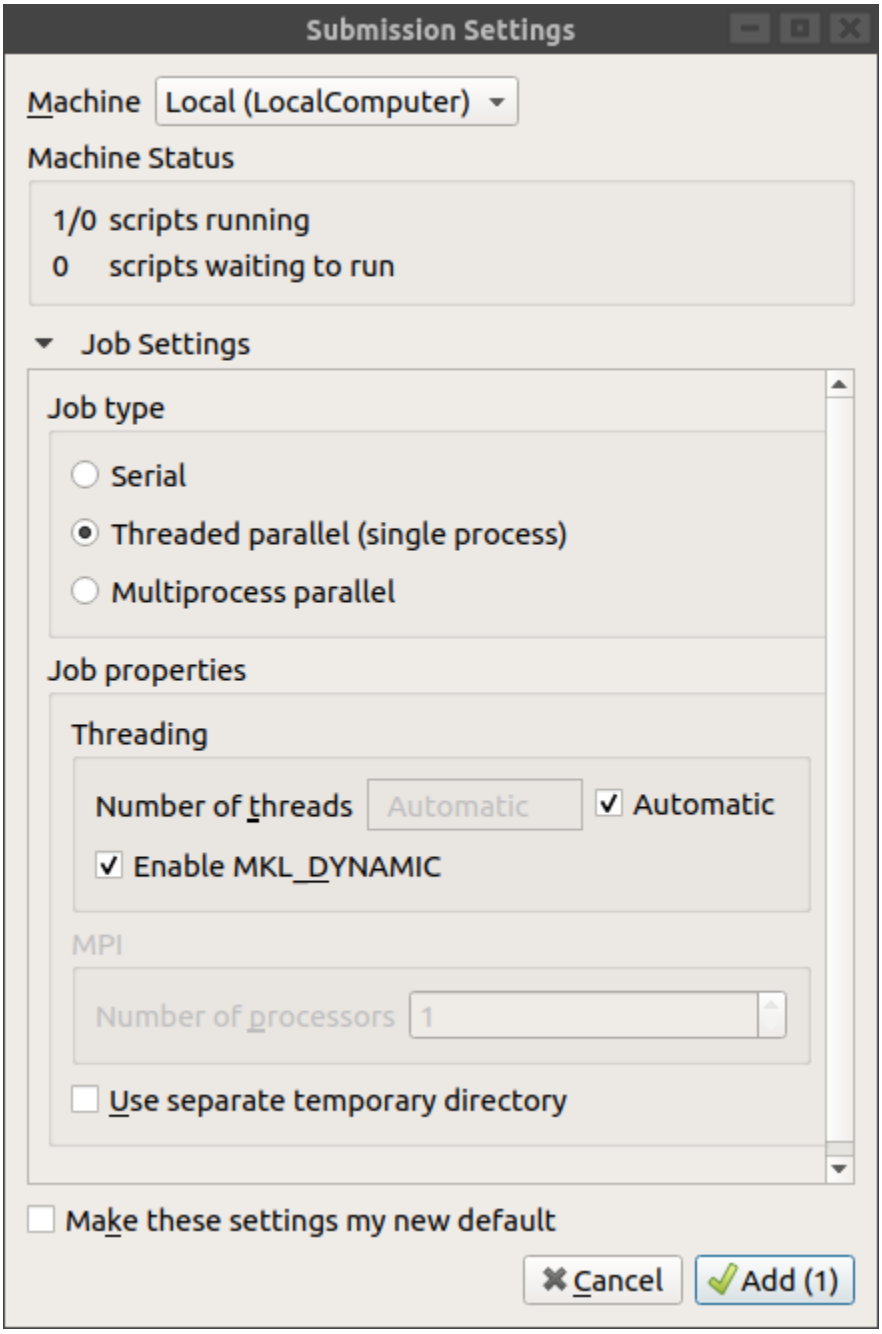


Only one core is used at a time, but the hardware process manager may move the task between cores from time to time.



Threading

In the **Submission Settings** window select a **Threaded parallel (single process)** job type as shown in the below figure to run on a single process with threading.

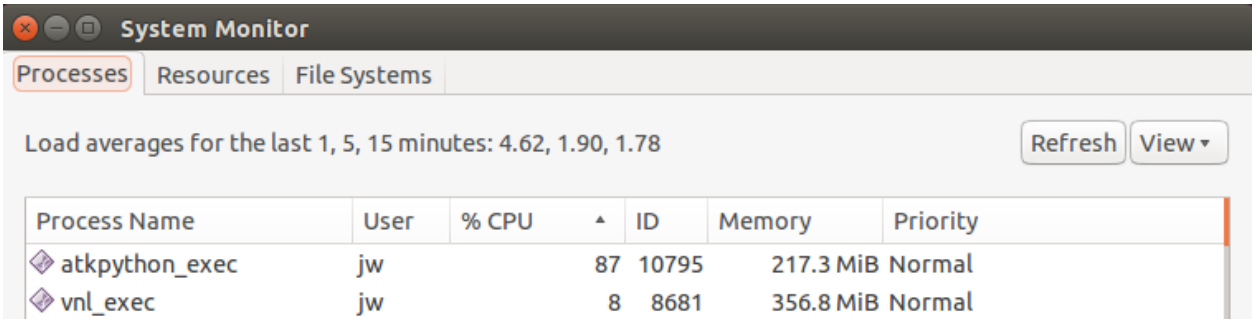


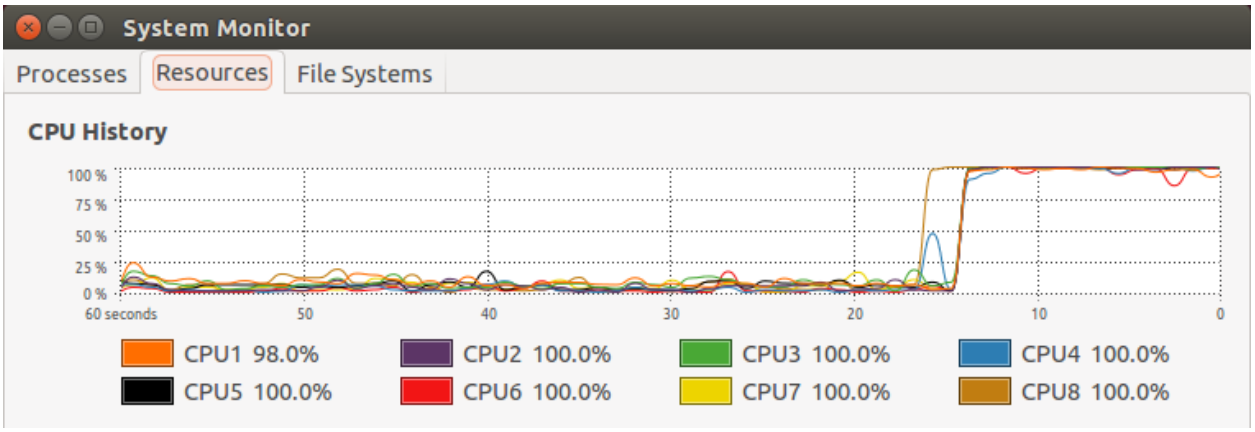
Threading is one way to parallelize a computational job. QuantumATK uses Intel MKL for openMP threading. Note that we do in general recommend *MPI parallelization* over threading for parallelizing DFT calculations. However, threading is often more efficient for parallelizing **ATK-ForceField** calculations, and Hybrid DFT calculations.

Download the script `cnt.py`, which uses ATK-ForceField to calculate the dynamical matrix of a multi-wall carbon nanotube.

Execute it using the **Job Manager**, and choose job type Threaded parallel (single process). It should be pretty fast.

If you check the system load during execution of the calculation, you should see that only a single **atkpython** process is started, even though several cores appear to be busy. This is because the work load of the one process is split into a number of threads that may be distributed on more cores.





Download the script `cnt.py` to test the performance of a ATK-ForceField simulations using threading. This specific example will calculate the dynamical matrix of a multi-wall carbon nanotube. If you also run the calculation in serial, you will see that the wall-clock time used for evaluating ATK-ForceField forces may decrease significantly when threading is switched on. In the example shown below, the time spent on force calculations is roughly halved.

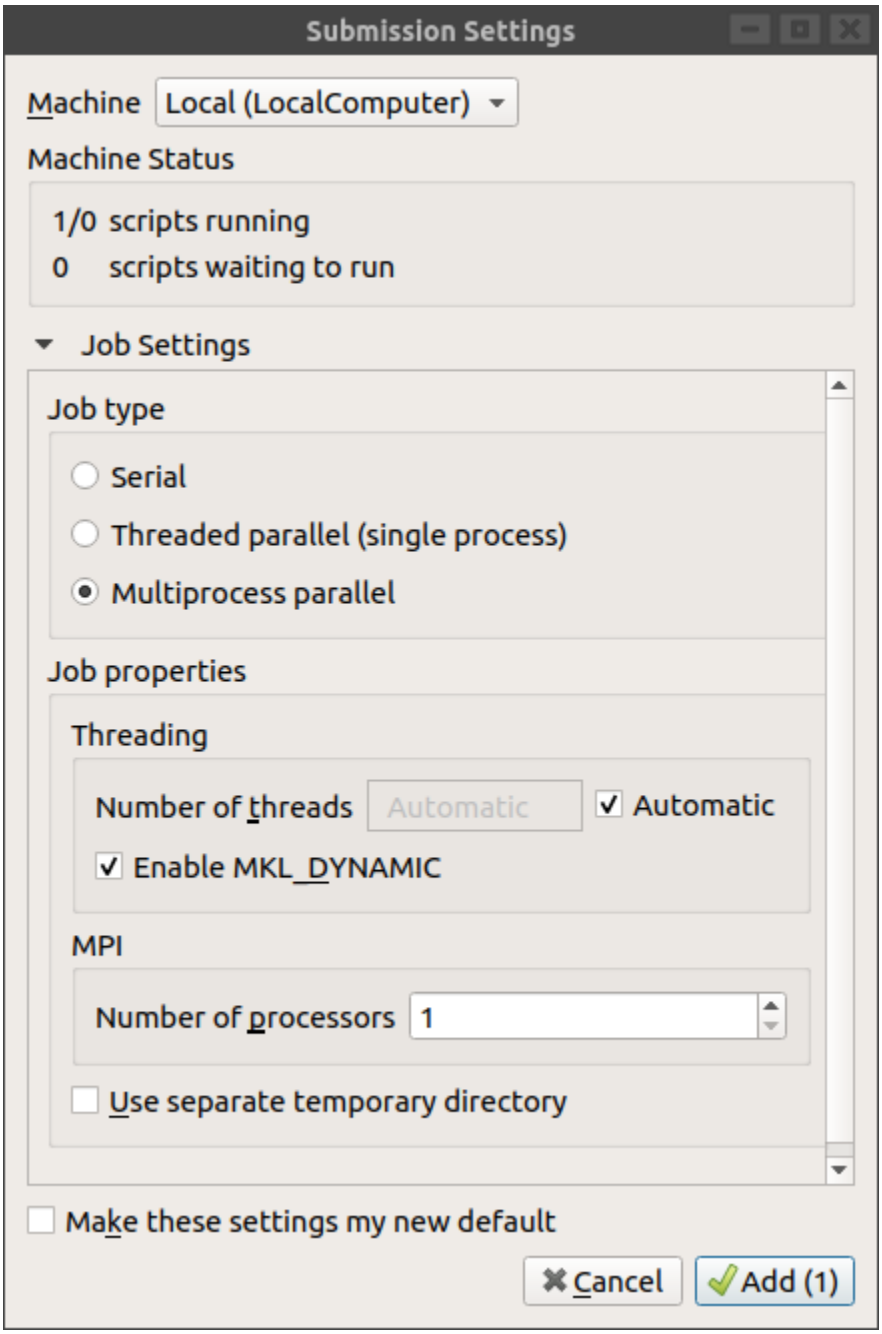
```
==> cnt_threading.log <==
Timing:
--------------------------------------------------------------------------------
Forces                  :      14.22 s       0.00 s      60.63% |=============|
Loading Modules + MPI   :       1.57 s       1.57 s       6.69% |=|
File IO, nlsave         :       0.10 s       0.05 s       0.43% |
--------------------------------------------------------------------------------

==> cnt_serial.log <==
Timing:
--------------------------------------------------------------------------------
Forces                  :      22.37 s       0.01 s      79.53% |=============|
Loading Modules + MPI   :       1.55 s       1.55 s       5.52% ||
File IO, nlsave         :       0.06 s       0.03 s       0.21% |
--------------------------------------------------------------------------------
```

MPI parallelization

Important: If you are running **ATK 2016 or earlier** you need to have MPI (Message Passing Interface) installed on your local computer.

Both the **Linux** and **Windows** versions are compiled against Intel MPI library. Since **ATK 2017** Intels `mpiexec.hydra` is provided on both Windows and Linux versions - **this is the recommended way to run QuantumATK in parallel**.

In the **Submission Settings** window select a *Multiprocess parallel* and e.g. 4 MPI processes.



Running from the command line

If you wish to run QuantumATK in parallel from the command line you can use the `mpiexec.hydra` binary shipped with QuantumATK and located in the folder `libexec/mpiexec.hydra` present in your installation folder.

In this case you can run parallel jobs with:

```
$ QW_INSTALLATION_PATH/libexec/mpiexec.hydra -n 4 atkpython atk_script.py
```

Hint: Prepend `QW_INSTALLATION_PATH/libexec/` in your `PATH`.

```
$ export PATH=QW_INSTALLATION_PATH/libexec:$PATH
```

This way you will automatically pick up the `mpiexec.hydra` binary shipped with QuantumATK:

```
$ mpiexec.hydra -n 4 atkpython atk_script.py
```

Adding Computers

It may sometimes be convenient to have a predefined local computer that is set up with MPI parallelization as default mode. You can easily add such a computer yourself.

In the Job Manager main window, click to add a new computer and select *New Local*.

Then edit the default job settings of the new computer in the **Job Settings** window that pops up:

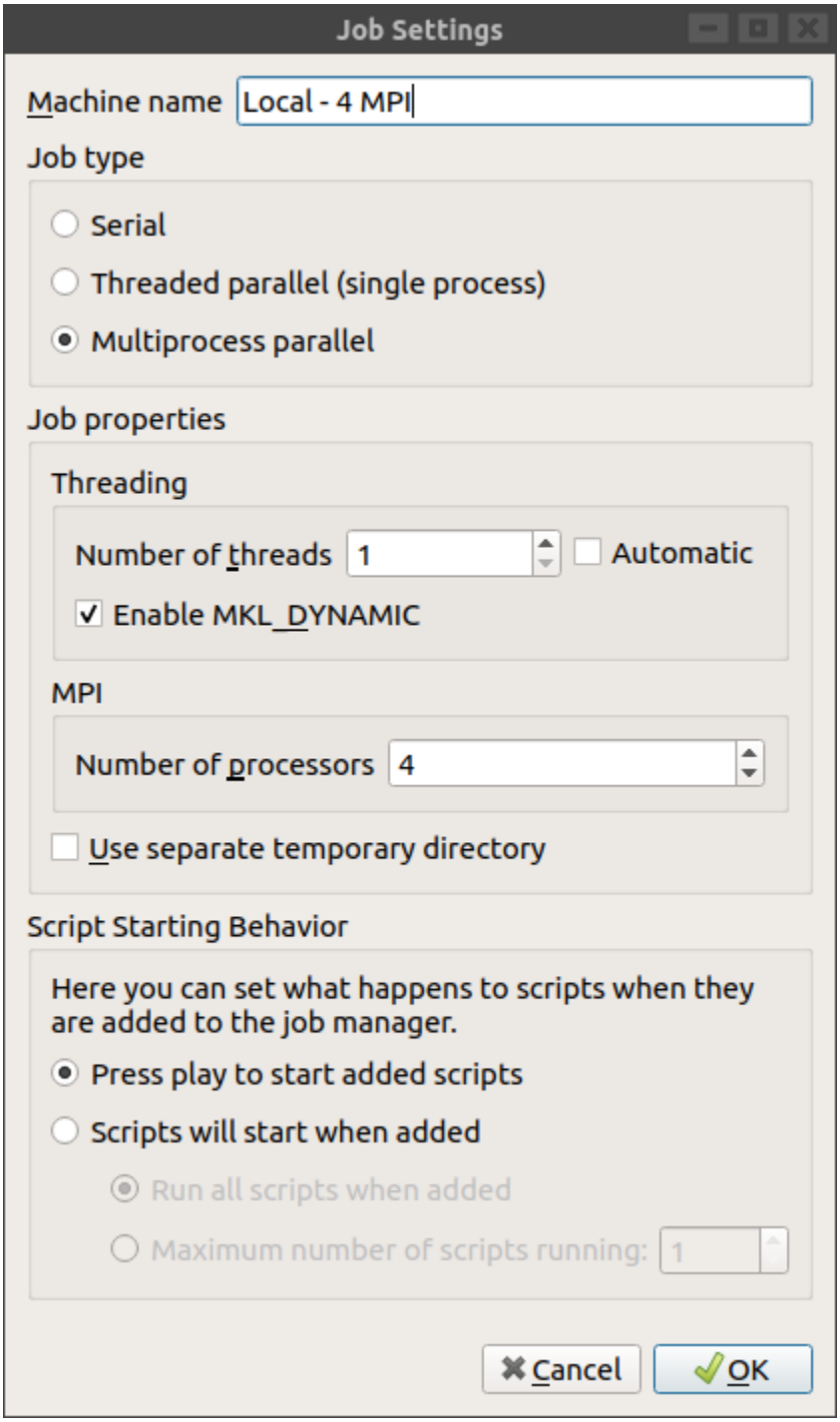
•Name the computer, e.g. Local - 4 MPI.

•Select *Multiprocess Parallel* as job type.

•Make sure threading is turned off (*Number of threads* = 1)

•Choose the default number of processors, e.g. 4.

•Click *OK* to add the new computer.



You can add as many custom computers as you like.

The **Job Settings** widget has an additional panel in addition to *Job type* and *Job properties*:

There are two options. Either you manually press play to run the jobs, or you can automatically start jobs submitted to the computer.

Set these settings according to your needs, and click **Add**.

2.6.2 Job Manager for remote execution of QuantumATK scripts

Downloads & Links

`mpi_check.py` `silicon.py`

In this tutorial you will learn how to use the Job Manager for execution of QuantumATK jobs on remote computing clusters. In particular, you will learn how to:

- add a remote machine to the Computers/Jobs;
- use custom Machine Settings for individual jobs;
- add several different machines and import/export machine settings.

Important: You will set up a remote machine for running jobs in parallel using MPI, as well as with threading. We strongly recommend you go through the tutorial *Job Manager for local execution of QuantumATK scripts* before continuing with this one.

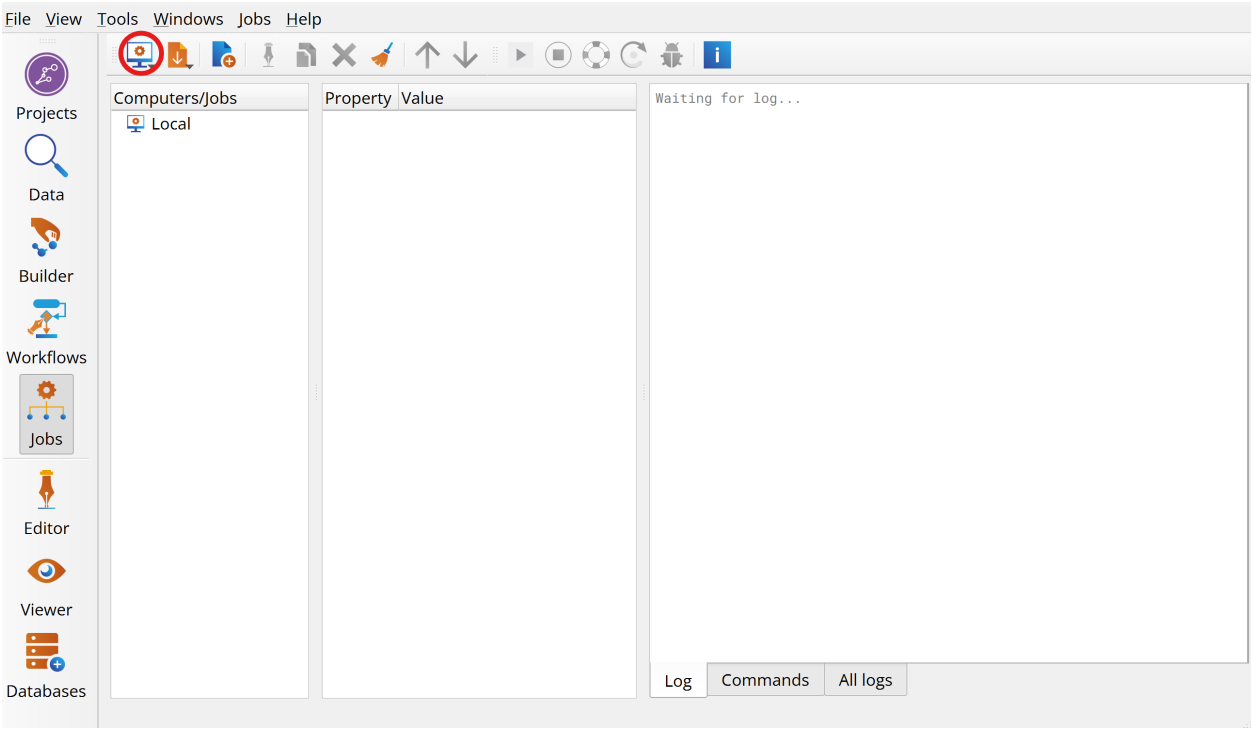
Note: There are two essential requirements when using the Job Manager for executing and managing QuantumATK jobs on a remote cluster. You need:

1. QuantumATK installed on the local machine **and** on the cluster;
2. an SSH connection from your local machine to the cluster.

Please refer to the tutorial ssh_keys if you need help setting up the SSH connection.

A single remote machine

Click on the **Create computer** to add a new machine.



The menu that appears has seven options, *New DirectRemote*, *New Local*, *New LSF*, *New PBS*, *New PBSPro*, *New SGE* and *New SLURM*,.

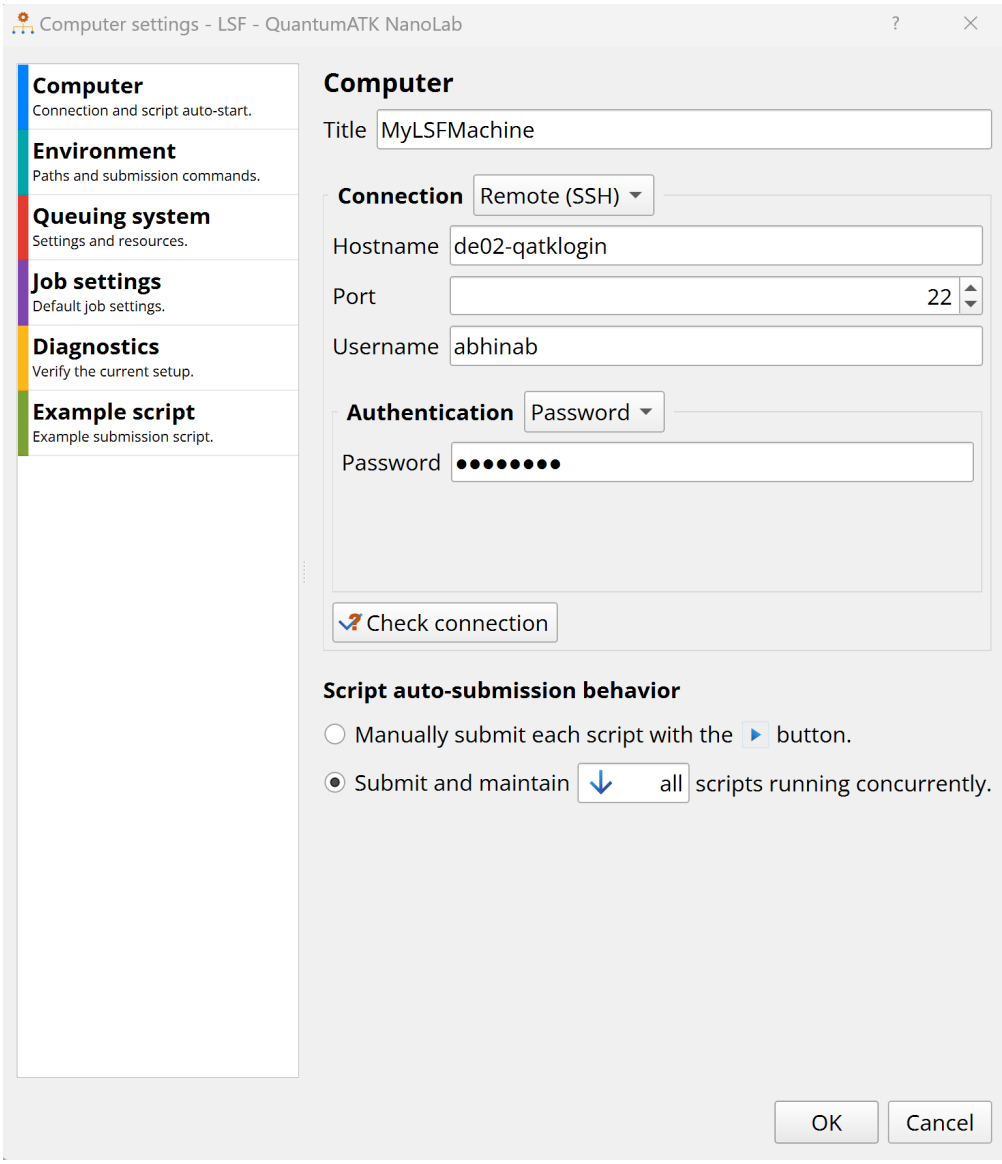
Choose the type of remote machine that matches the job scheduling on your remote Linux cluster, and start setting up the connection. Here, you will choose the *New LSF* machine to set up a machine with a LSF job scheduling system. The option *New DirectRemote* is appropriate for clusters that have no queue system.

The **Computer Settings** widget pops up. It has six main tabs:

- *Computer (Connection and script auto-start)*,
- *Environment (Paths and submission commands)*,
- *Queuing system (Settings and resources)*,
- *Job Settings (Default job settings)*,
- *Diagnostics (Verify the current setup)*,
- *Example script (Example submission script)*.

Computer

Connection to the remote cluster.



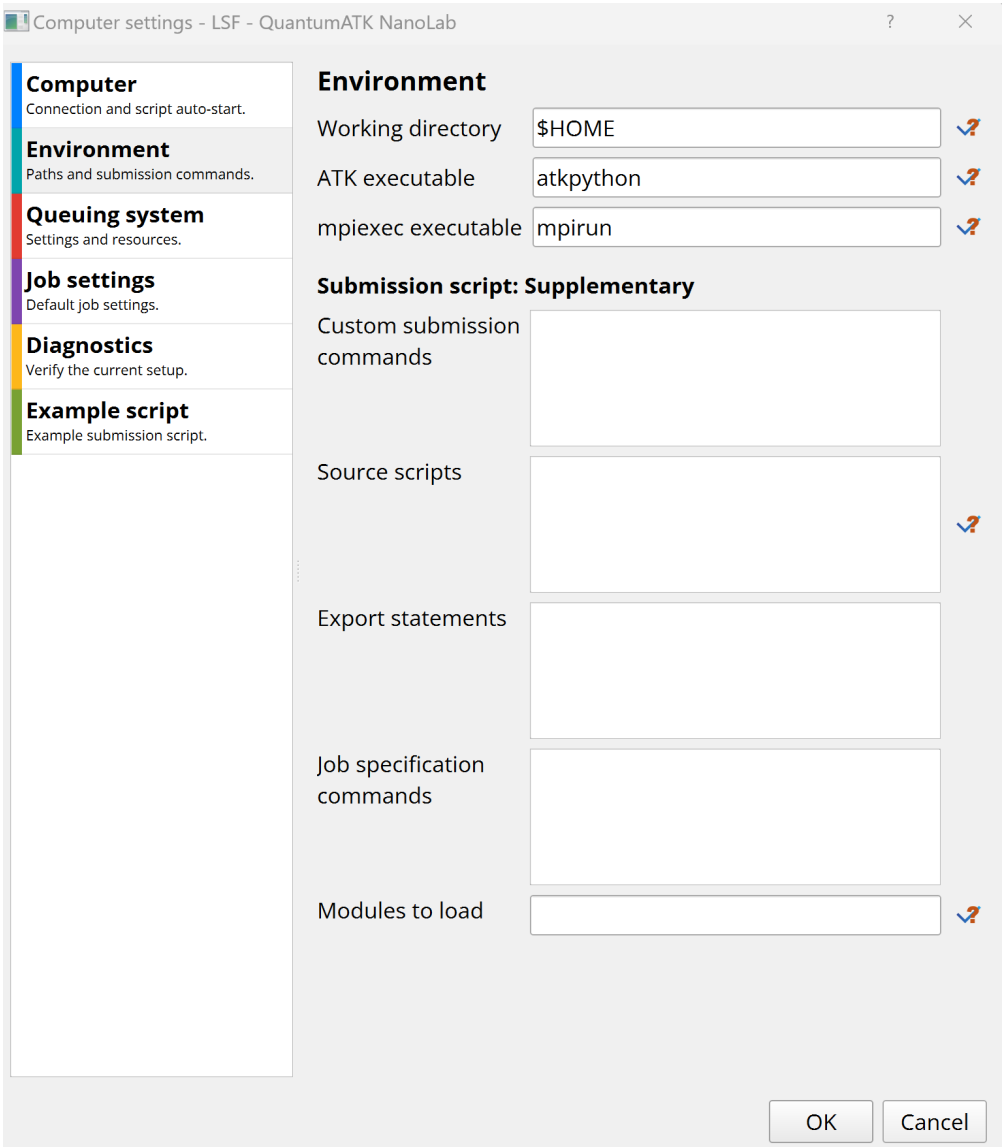
This tab is used to provide the credentials for connecting to the remote cluster of choice. It supports password-based and key-based authentication.

Private key file
The path of your private SSH key, you might already have one or you can generate one following ssh_keys.

Once all settings are added, you can check if they are correct by clicking the **Check connection** button.

Environment

Computing environment on the cluster.



This tab concerns the environment (directories, executables, modules, etc.) on the *remote cluster*, not on your local computer. `$HOME` is therefore the home directory of your user on the cluster, and QuantumATK must of course be installed on the cluster.

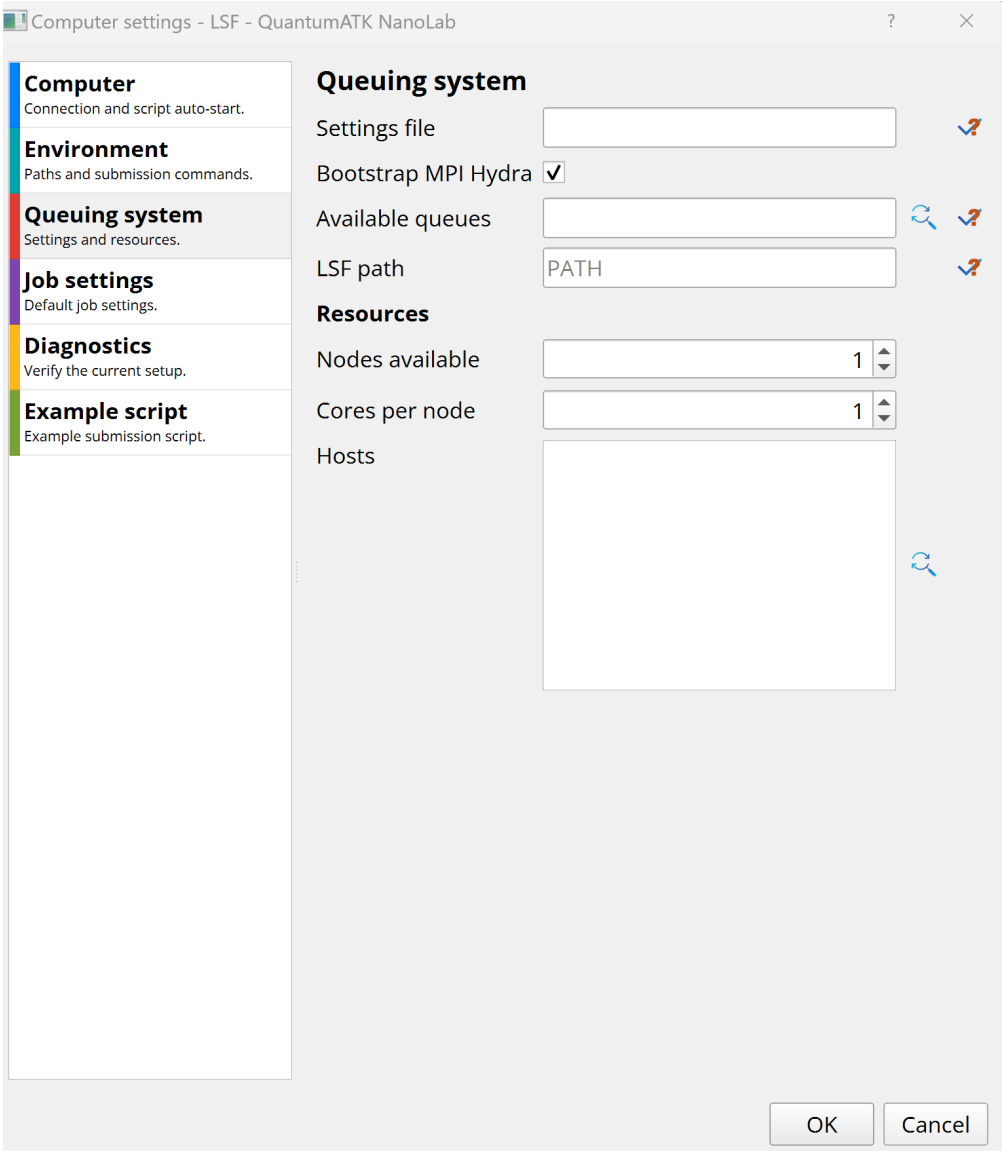
Any scripts that should be sourced in order to get the environment working must be listed. The same goes for required export statements (may be needed to correctly set the `SNPSLMD_LICENSE_FILE`) and cluster modules that should be loaded using the `module load` command (https://modules.readthedocs.io).

You can also provide *custom submission commands*. For example, you can choose a particular queue to submit the job:

```
# Submit job to "myqueue" queue
BSUB -app myqueue
```

Queuing system

Queuing system settings and resources available for job submission.



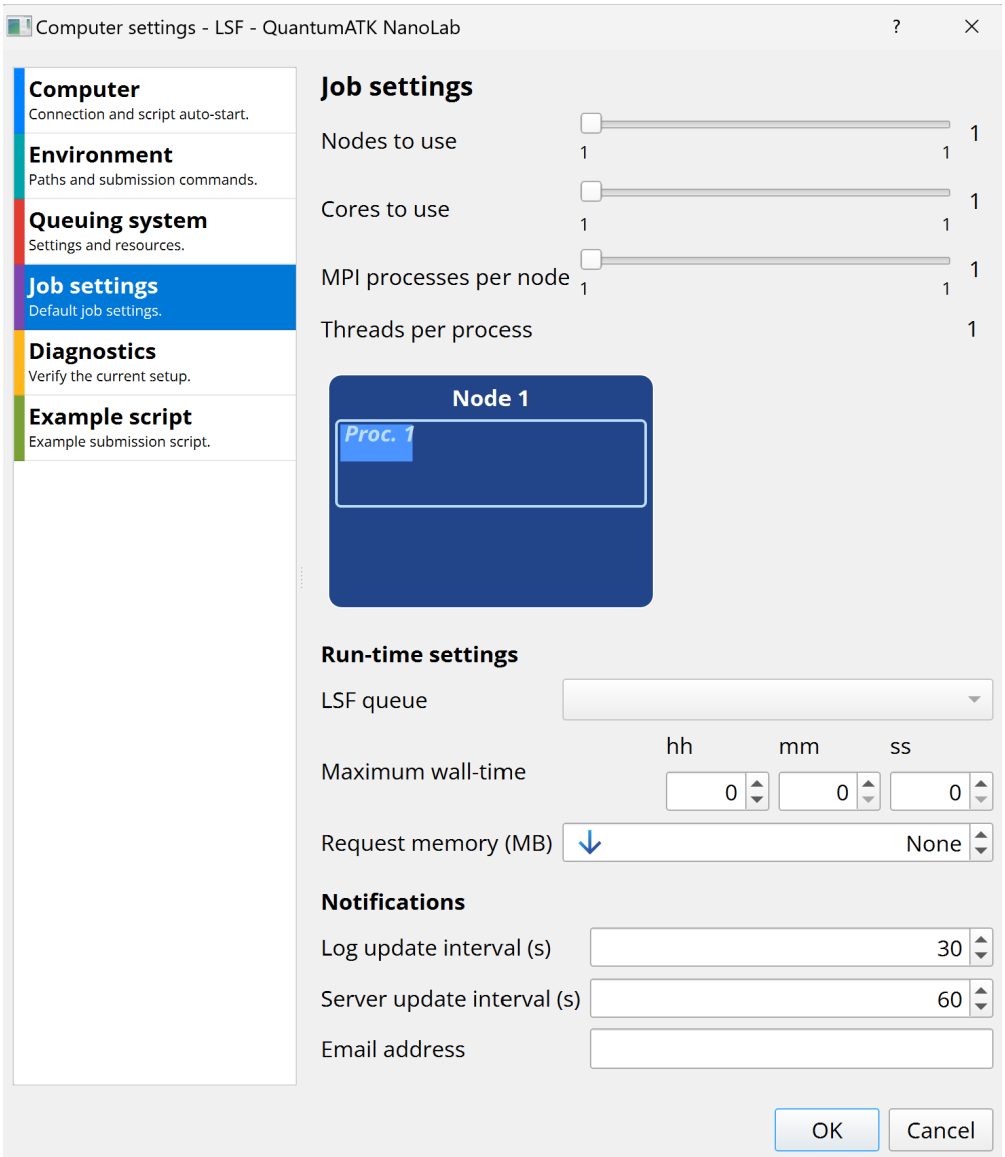
The nodes available and cores per node values are used to determine the default job settings.

LSF path
This is the directory containing `bsub` and other LSF executables. It is not needed if *Settings File* is provided. Log on to the cluster and use the `which` command to locate it:

```
$ which bsub
```

Job Settings

Computing resources requested at job submission.

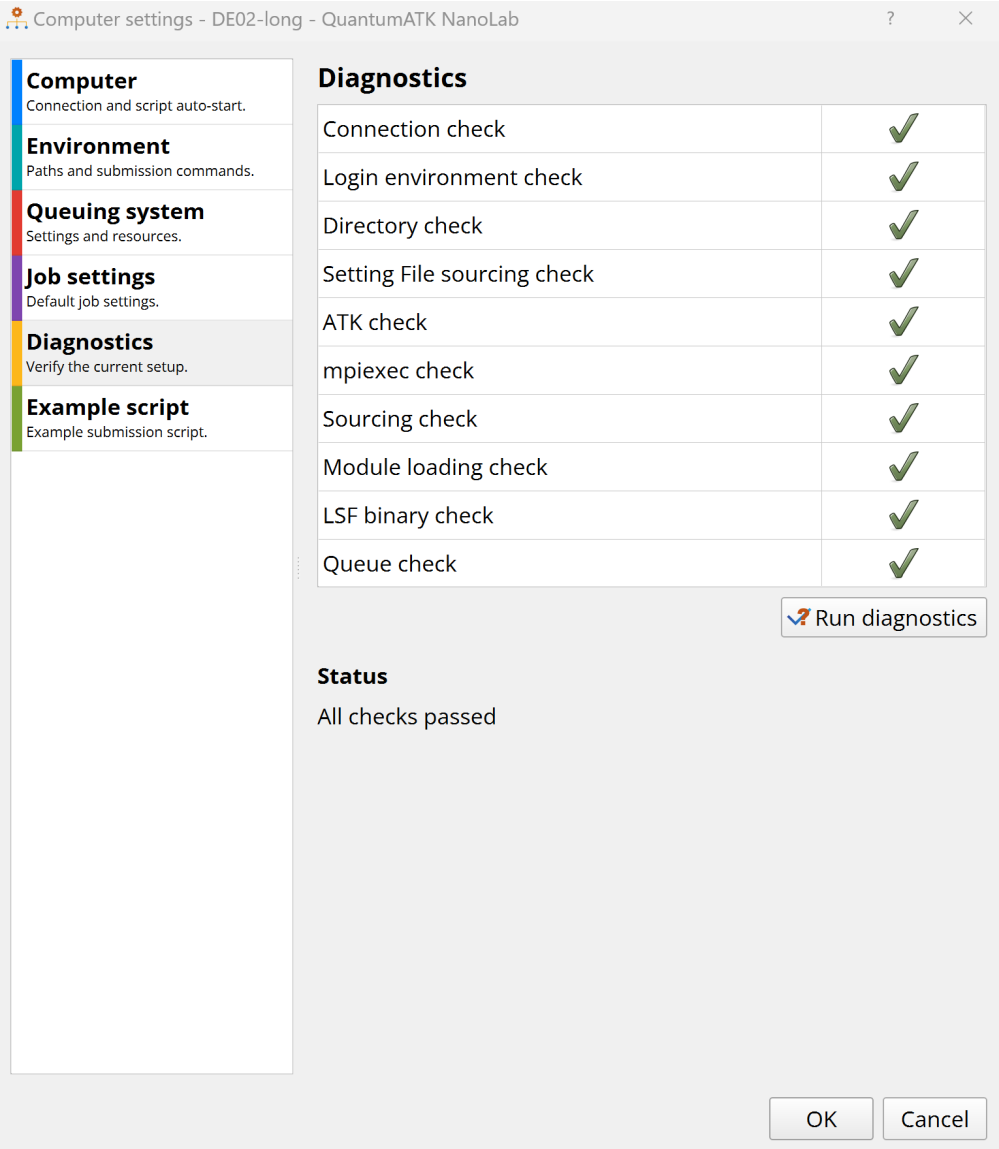


This tab specifies the default computing resources (nodes, cores, queue, time, etc.) requested at job submission. Once the number of cores to use on each node is specified, you can decide how to split them between MPI and threading parallelization. While changing the number of MPI processes, the number of threads per process is automatically adjusted.

The Job Manager will regularly check the job progress on the remote cluster and report it in the log file. You can also receive e-mail notifications from the job scheduler when the job starts, finishes, or terminates. Refer to your cluster administrator to know if email notifications are supported in your cluster.

Diagnostics

Use this tab to test the computer settings. Green check-mark indicates that all settings appear to be OK. Red cross indicates some problem that should be fixed in one of the tabs.



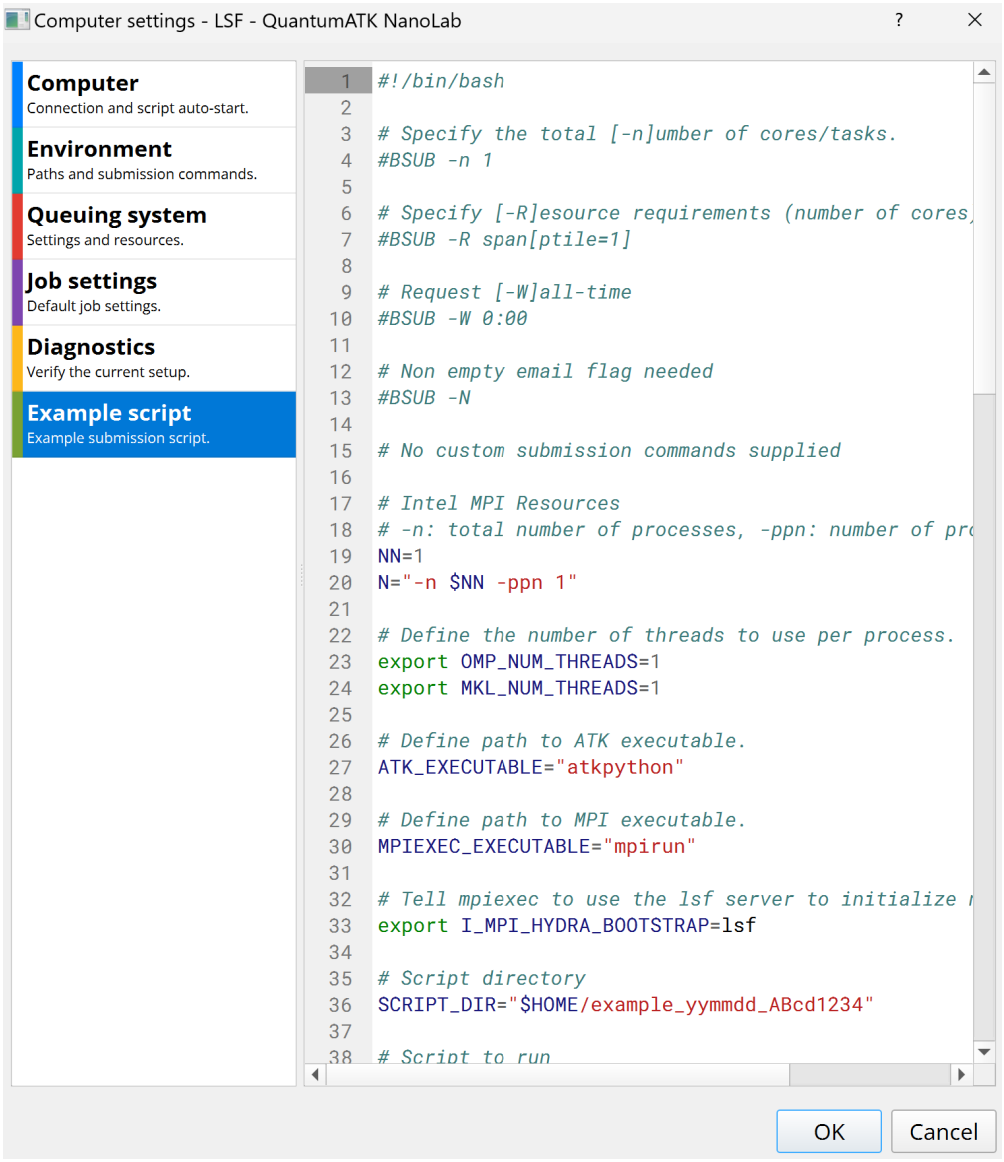
Warning: The Diagnostics tab checks if the options in the **Computer** and **Environment** tabs allow your local computer to connect to the remote cluster and execute the commands needed for job submission and management.

If some field is not specified, the diagnostics will check the default setting. If the connection to the cluster works well with that default, or is at least not disrupted, it will be marked as OK.

You therefore need to run an actual test job to make absolutely sure that all settings are indeed OK.

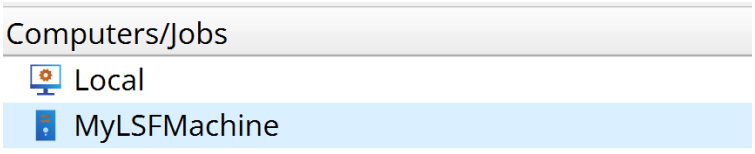
Example script

This shows an example of a job submission script. You can for example check that the options passed to the job scheduler are correct.

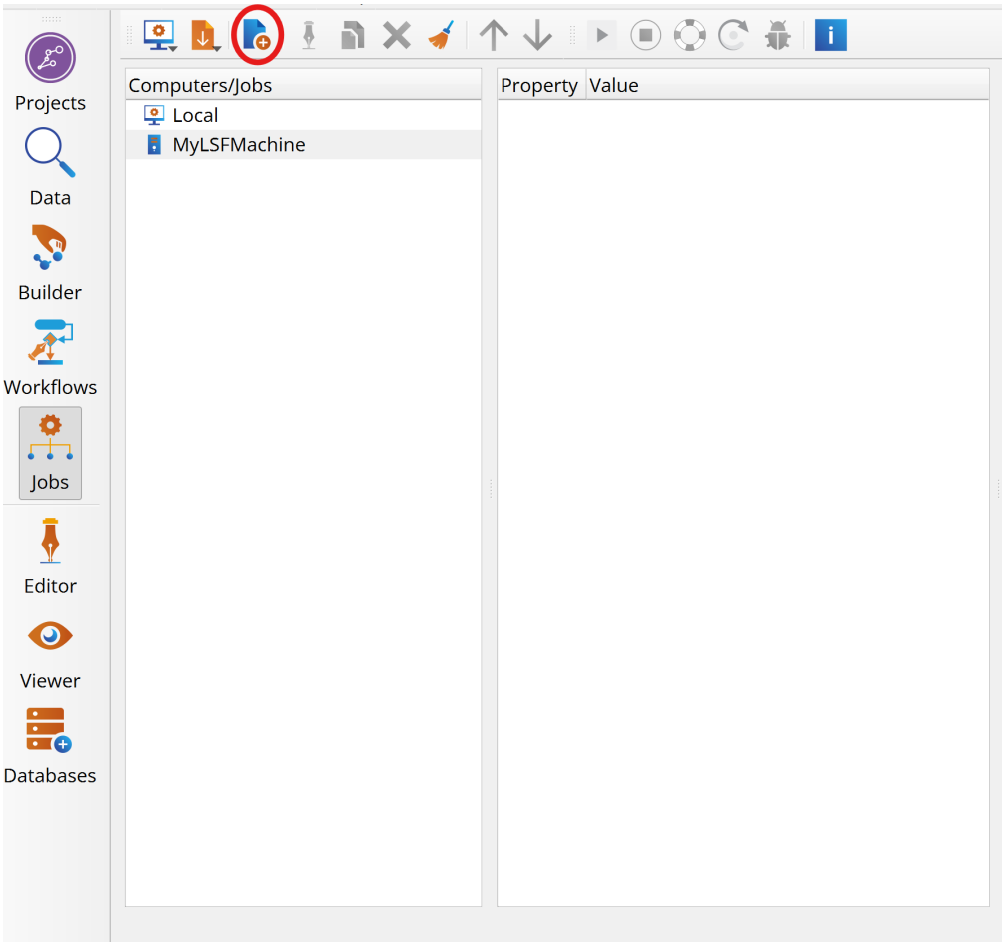


Save and test the new machine

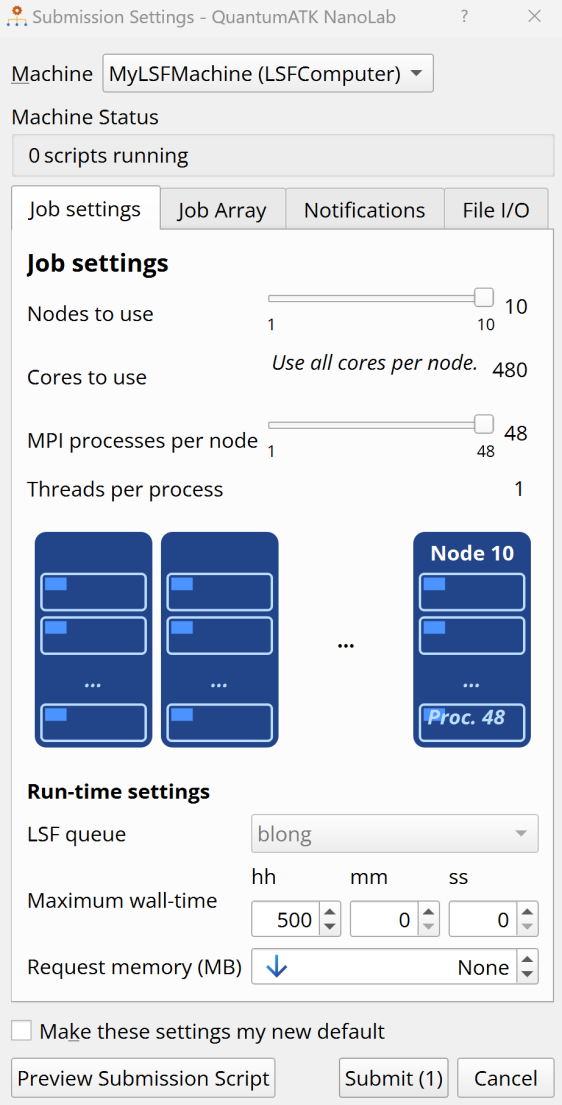
Click *OK* to add the machine to the **Computers/Jobs**.



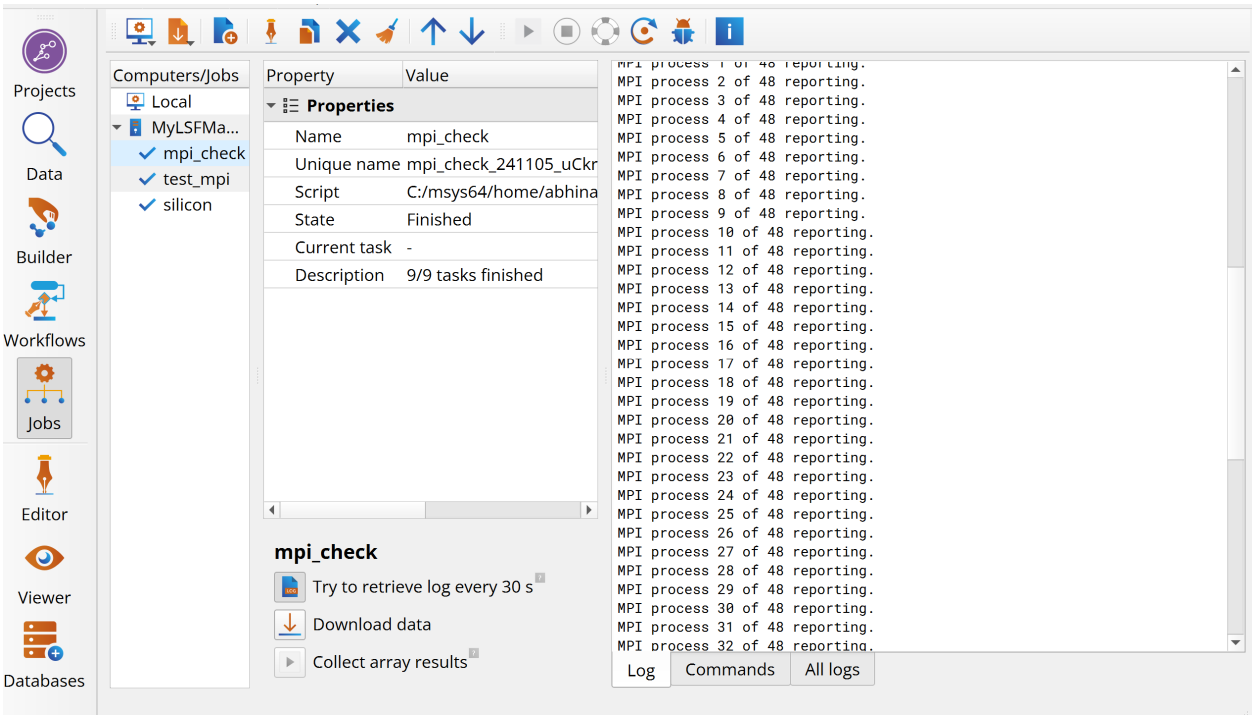
Next, you should run the test scripts `mpi_check.py` and `test_mpi.py` to test the machine settings. In the QuantumATK main window, click on **Add new job**, then choose the script from the file dialog box, and click *Open*.

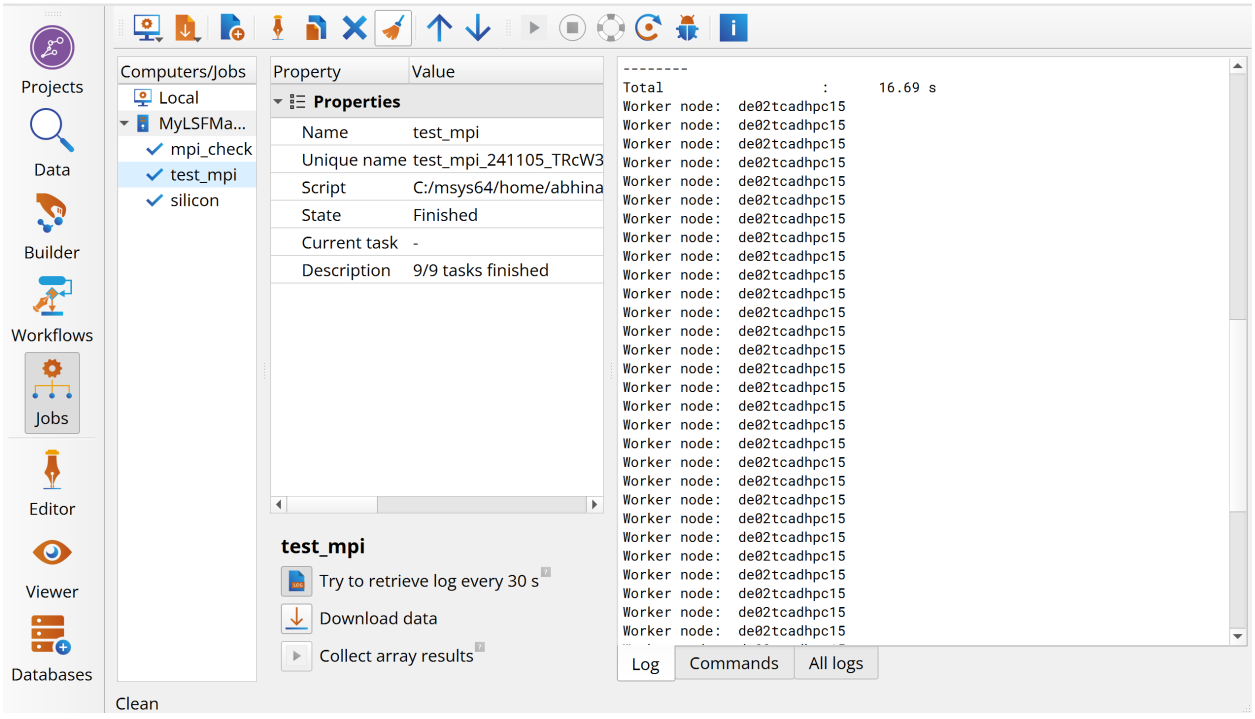


This will open the **Computer Settings** dialog box. Choose the remote machine, in this example *MyLSFMachine*, and click *Submit*.



Once a job has finished you should check the log file to inspect the job output to see if the expected number of processes were used. Click the *Log* tab to open the log. In the two examples below, 48 cores were used in total on a single node.


Projects
Data
Builder
Workflows
Jobs
Editor
Viewer
Databases
Computers/Jobs
Local
MyLSFMa...
mpi_check
test_mpi
silicon
Property
Value
Properties
Name
mpi_check
Unique name
mpi_check_241105_uCkr
Script
C:/msys64/home/abhina
State
Finished
Current task
-
Description
9/9 tasks finished
mpi_check
Try to retrieve log every 30 s
Download data
Collect array results
MPI process 2 of 48 reporting.
MPI process 3 of 48 reporting.
MPI process 4 of 48 reporting.
MPI process 5 of 48 reporting.
MPI process 6 of 48 reporting.
MPI process 7 of 48 reporting.
MPI process 8 of 48 reporting.
MPI process 9 of 48 reporting.
MPI process 10 of 48 reporting.
MPI process 11 of 48 reporting.
MPI process 12 of 48 reporting.
MPI process 13 of 48 reporting.
MPI process 14 of 48 reporting.
MPI process 15 of 48 reporting.
MPI process 16 of 48 reporting.
MPI process 17 of 48 reporting.
MPI process 18 of 48 reporting.
MPI process 19 of 48 reporting.
MPI process 20 of 48 reporting.
MPI process 21 of 48 reporting.
MPI process 22 of 48 reporting.
MPI process 23 of 48 reporting.
MPI process 24 of 48 reporting.
MPI process 25 of 48 reporting.
MPI process 26 of 48 reporting.
MPI process 27 of 48 reporting.
MPI process 28 of 48 reporting.
MPI process 29 of 48 reporting.
MPI process 30 of 48 reporting.
MPI process 31 of 48 reporting.
MPI process 32 of 48 reporting.
Log
Commands
All logs


Projects
Data
Builder
Workflows
Jobs
Editor
Viewer
Databases
Computers/Jobs
Local
MyLSFMa...
mpi_check
test_mpi
silicon
Property
Value
Properties
Name
test_mpi
Unique name
test_mpi_241105_TRcW3
Script
C:/msys64/home/abhina
State
Finished
Current task
-
Description
9/9 tasks finished
test_mpi
Try to retrieve log every 30 s
Download data
Collect array results

Total : 16.69 s
Worker node: de02tcadhpc15
Log
Commands
All logs
Clean

Custom job settings

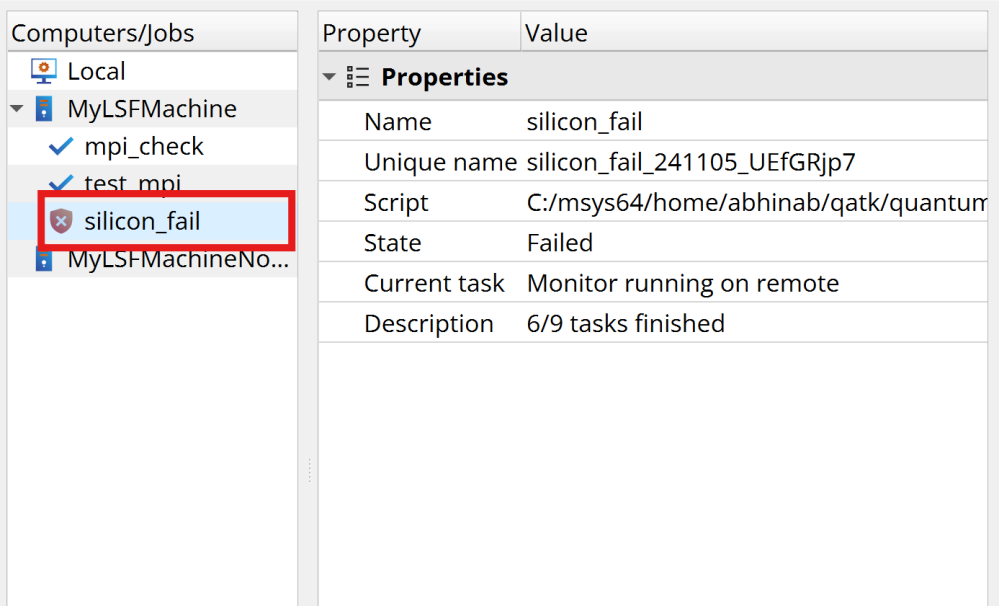
As explained in the tutorial *Job Manager for local execution of QuantumATK scripts*, you can customize many of the job settings before submitting a job. Use the script `silicon.py` as an example QuantumATK script. Download the script, and then *Open* the script from file dialog box by clicking **Add new job**. This will open a **Submission Settings** dialog box.

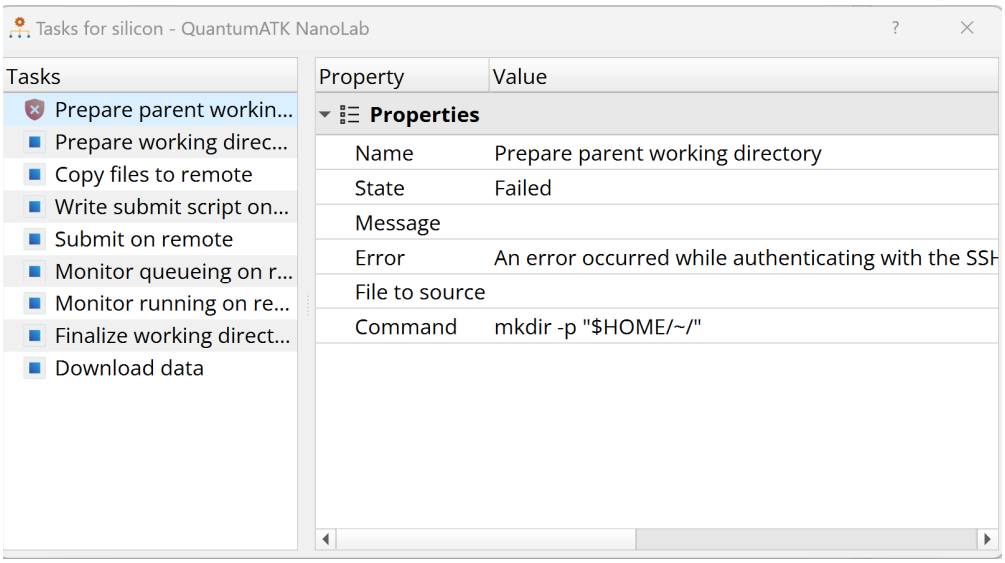
You can now customize the *Job Defaults*, *Job Array*, *Notifications* and *File I/O* tabs, and thereby submit the job with settings different from the default ones you specified above. For example, you can change the number of requested cores and cluster queue and/or maximum wall-clock time.

Set up the job settings as you like, e.g. 4 cores on a single node and 4 MPI processes. Then click *OK* and submit the calculation to the remote cluster.

Debugging

If an error occurs during the execution of the job, this will be indicated by a red icon in the queue, as shown below. You can then click the **Debug logs** icon to open the job debugs information window, which will show you details about the error.





Adding several remote machines

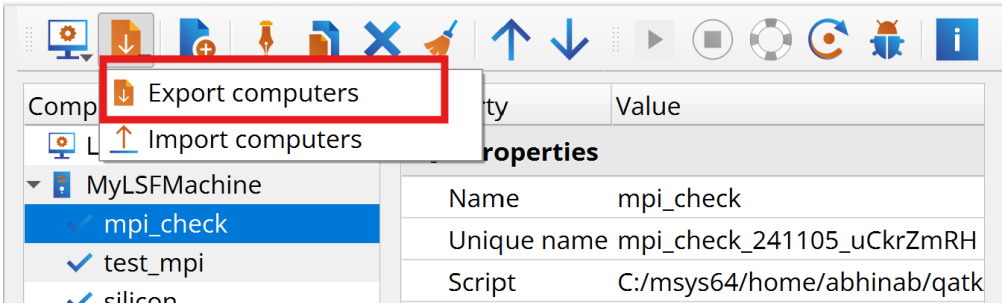
Several remote machines can be added to the **Computers/Jobs**. You can of course add new machines from scratch, but you can also export/import the settings of an existing machine and use those as a template for new machines.

Tip: The import/export functionality is very convenient for sharing machine settings within a group of users.

In the following, you will export the settings of the newly created machine, and then add one more remote machine with different queue and job settings.

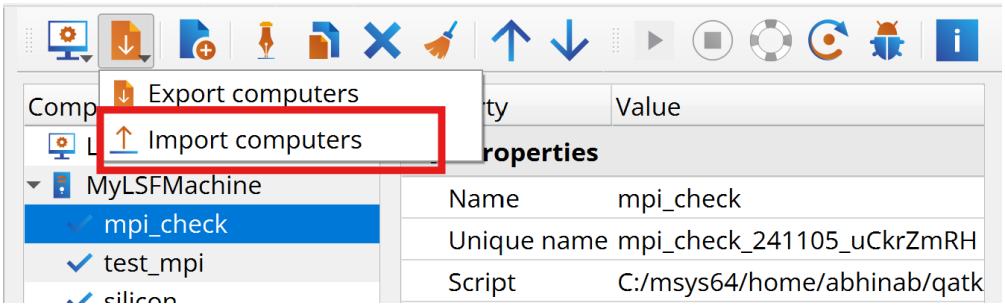
Export

Select the machine and click on *Import or export computers* , then click *Export*, and save the settings in a file.



Import

Next, add a new remote machine with default settings for New LSF. click on *Import or export computers* , then click *Import* and import the settings file you just saved.



You can now modify the settings to create a machine with default settings that are suitable for :

Select the new machine and click *Edit*. This will open the *Computer Settings* dialog box. Give the new machine a reasonable name, e.g. MyNewLSFMachine and set the password, if *Password* is selected for authentication. Then, change the job submission queue to mynewqueue by adding *BSUB -app myqueue* in *Custom submission commands* of the *Environment* tab. Additionally , you can change the *Nodes available*, e.g. to 2, and *Cores per node*, e.g. to 8, in the *Queuing system* tab. Finally, *Run diagnostics* in the *Diagnostics* tab to check if the settings are correct and click *OK*, if it passes all the checks.

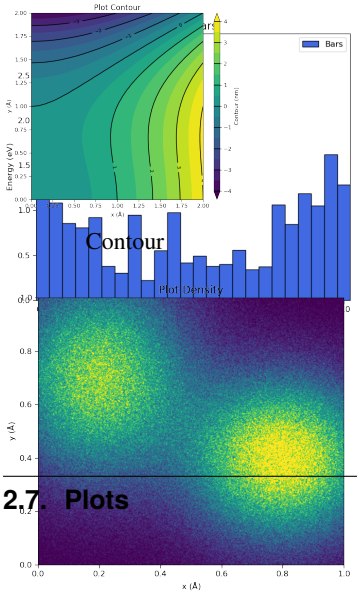
2.7 Plots

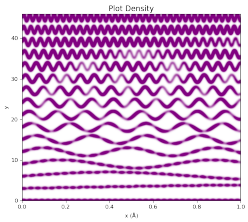
NanoLab offers a plotting framework for creating publication quality plots. A strong emphasis has been put on interactivity, so plots can easily be modified directly in the NanoLab GUI.

You can learn even more about working with plots in the NanoLab GUI by reading the the tutorial: *Working with Plots in NanoLab*.

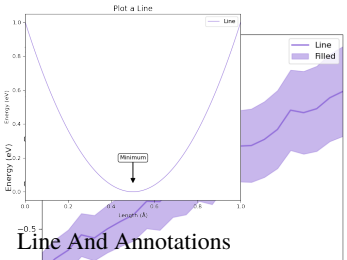
You can also create your own plots through scripting, and you can read much more about that in the *NanoLab Plot Reference Manual*.

2.7.1 Plot Gallery

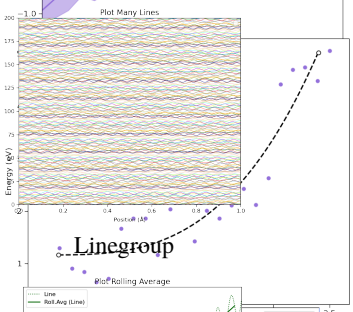




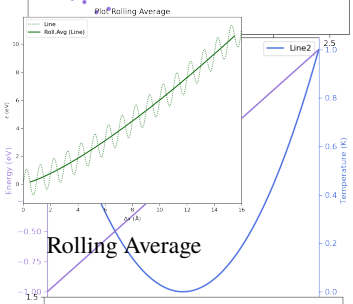
Density For Weighted Lines



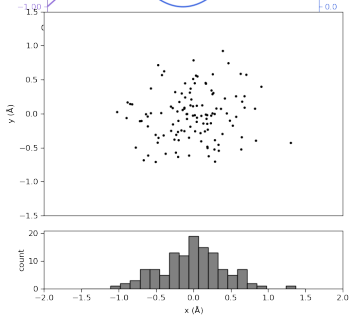
Line And Annotations



Linegroup

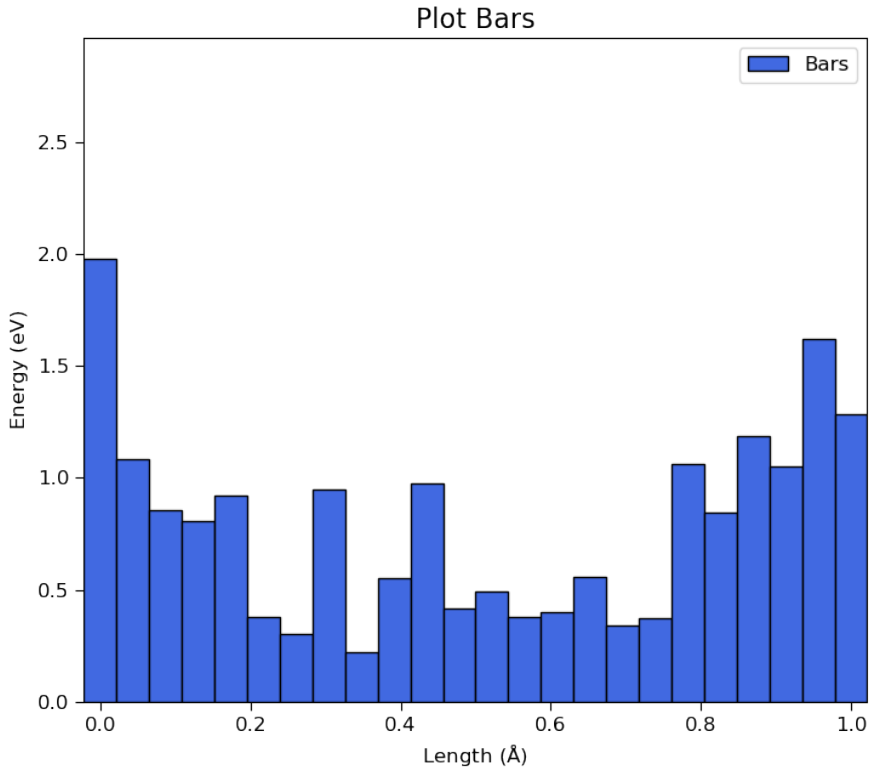


Rolling Average



Scatter And Bars

Bars



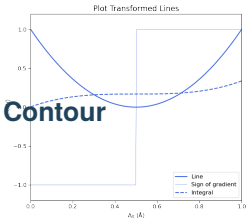

```
# Plot bars
# This example creates a simple bar plot

# Generate some dummy data.
position = numpy.linspace(0, 1, 24) * Angstrom
energy = (numpy.
↪random.rand(24) + numpy.linspace(-1, 1, 24) ** 2) * eV

# Create a model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Plot Bars')
model.legend().setVisible(True)
model.xAxis().setLabel('Length')
model.yAxis().setLabel('Energy')

# Change y-axis autoscale padding.
model.yAxis().setAutoscalePadding(0, 0.5)

# Create and add vertical bar plot.
bars = Plot.VerticalBar(position, energy)
bars.setLabel('Bars')
bars.setFaceColor('royalblue')
model.addItem(bars)

# Autoscale.
model.setLimits()
```

(continues on next page)

(continued from previous page)

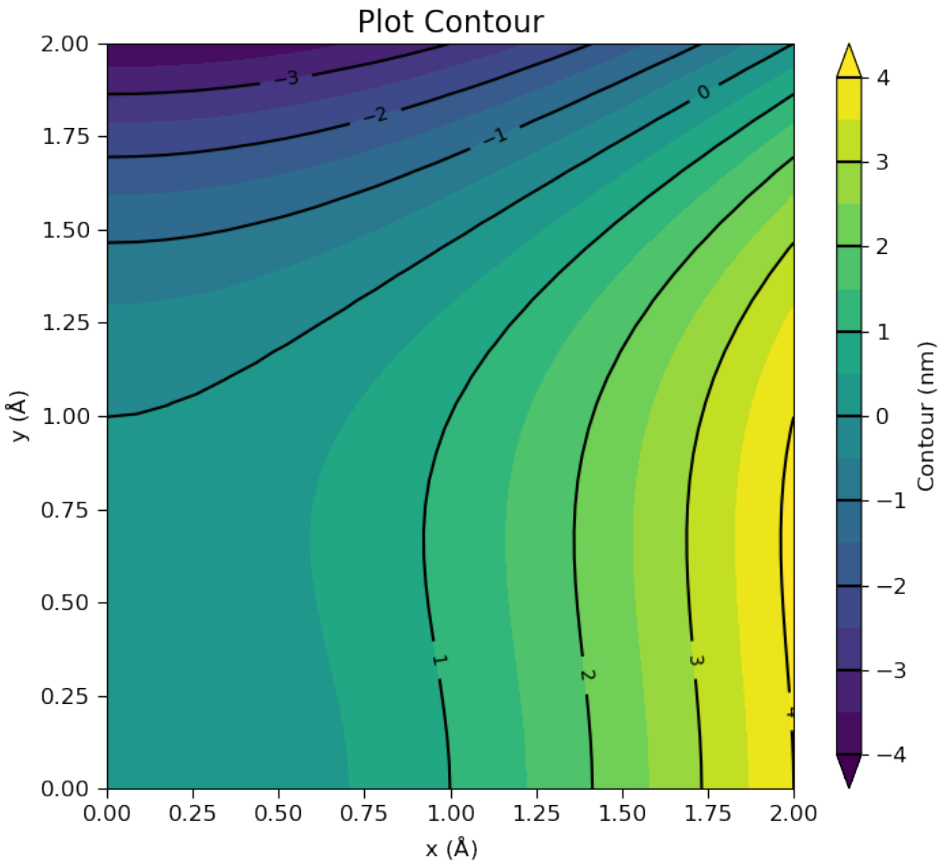
```
26
27 # Show the plot for interactive editing.
28 Plot.show(model)
29
30 # Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
31 Plot.save(model, 'bars.png')
```

References: *Plot.save*, *Plot.show*, *Plot.PlotModel*, *Plot.VerticalBar*.



Transformed Line

Go back to *Plots*




```
1 # Plot contour
2 # This example creates a contour plot.
3
4 # Generate some dummy data.
5 xs = numpy.linspace(0, 2, 24) * Angstrom
6 ys = numpy.linspace(0, 2, 30) * Angstrom
7
8 x_mesh, y_mesh = numpy.meshgrid(xs.inUnitsOf(Angstrom), ys.
  ↪inUnitsOf(Angstrom))
9 heights = (x_mesh ** 2 + y_mesh ** 2 * (1 - y_mesh)) * nm
```

(continues on next page)

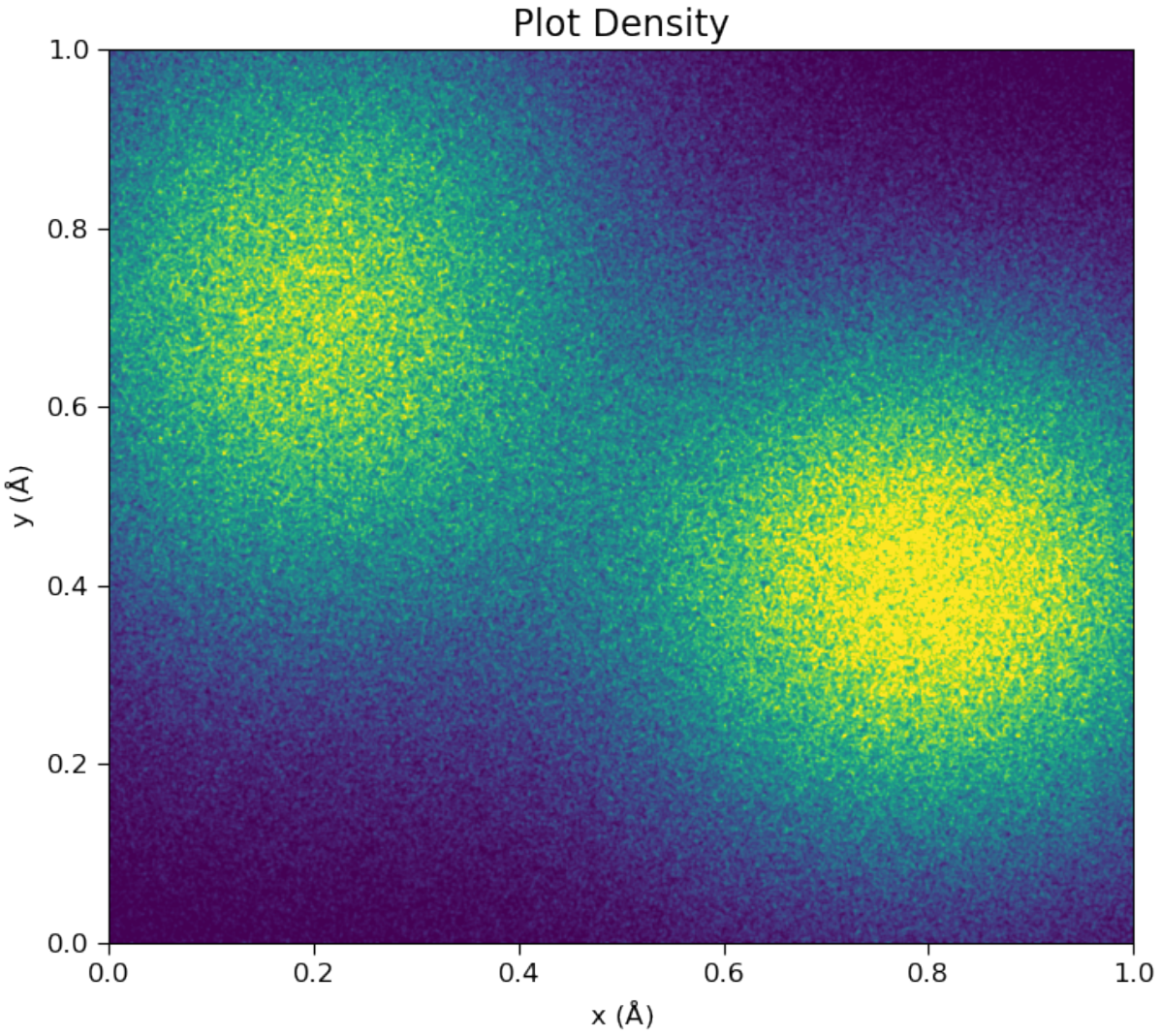
(continued from previous page)

```
10
11 # Create a model and apply some settings.
12 model = Plot.PlotModel(Angstrom, Angstrom)
13 model.framing().setTitle('Plot Contour')
14 model.xAxis().setLabel('x')
15 model.yAxis().setLabel('y')
16
17 # Change x- and y-axis autoscale padding.
18 model.xAxis().setAutoscalePadding(0, 0)
19 model.yAxis().setAutoscalePadding(0, 0)
20
21 # Create and add contour plot.
22 contour = Plot.Contour(xs, ys, heights)
23 contour.setLabel('Contour')
24 contour.setLevels(16)
25 contour.setLimits(-4, 4)
26 contour.setLabelsVisible(True)
27 model.addItem(contour)
28
29 # Autoscale.
30 contour.setLimits()
31 model.setLimits()
32
33 # Show the plot for interactive editing.
34 Plot.show(model)
35
36 # Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
37 Plot.save(model, 'contour.png')
```

References: *Plot.save*, *Plot.Contour*, *Plot.PlotModel*, *Plot.show*.

Go back to *Plots*

Density




```
# Plot density
# This example creates a highly efficient and responsive density plot.

# Generate some dummy data.
N = int(1e6)

xs = numpy.zeros(2 * N)
ys = numpy.zeros(2 * N)

xs[:N] = numpy.random.normal(loc=0.8, scale=0.2, size=N)
xs[N:] = numpy.random.normal(loc=0.2, scale=0.2, size=N)

ys[:N] = numpy.random.normal(loc=0.4, scale=0.2, size=N)
ys[N:] = numpy.random.normal(loc=0.7, scale=0.25, size=N)

xs = xs * Angstrom
```

(continues on next page)

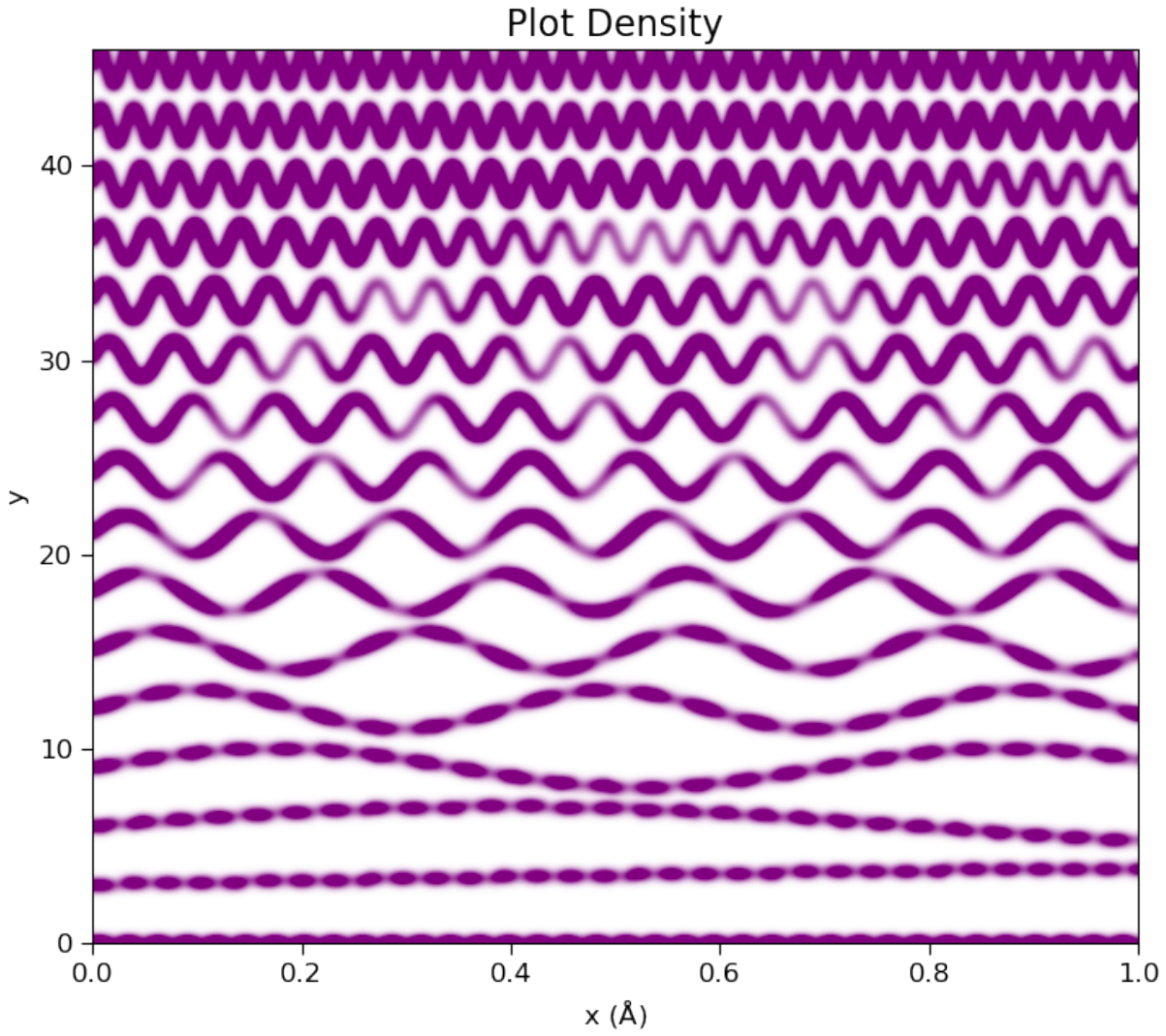
(continued from previous page)

```
ys = ys * Angstrom

# Create a model and apply some settings.
model = Plot.PlotModel(Angstrom, Angstrom)
model.framing().setTitle('Plot Density')
model.xAxis().setLabel('x')
model.yAxis().setLabel('y')

# Change y-axis autoscale padding.
model.xAxis().setAutoscalePadding(0, 0)
model.yAxis().setAutoscalePadding(0, 0)

# Create and add density plot.
density = Plot.Density(xs, ys)
density.setBroadening(1)
density.setColorMode(Plot.COLOR_MODES.COLOR_MAP)
density.setDpi(150)
density.setLimits(0, 5)
model.addItem(density)

# Autoscale.
model.setLimits('x', 0, 1)
model.setLimits('y', 0, 1)

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'density.png')
```

References: *Plot.Density*, *Plot.PlotModel*, *Plot.show*, *Plot.save*.

Go back to *Plots*

Density For Weighted Lines




```
# Plot weighted lines as a density
# This example creates a highly efficient and responsive density plot for plotting weighted lines.

# Generate some dummy data.
xs = numpy.linspace(0, 1, 256) * Angstrom

# Create oscillating lines.
constants = numpy.arange(16)
xs_mesh, constants_mesh = numpy.meshgrid(xs.inUnitsOf(Angstrom), constants)
ys = numpy.sin(constants_mesh**2 * xs_mesh) + 3 * constants_mesh

# Create oscillating weights.
weights = numpy.sin(constants_mesh[::-1, :]**2 * xs_mesh) + 1.5

# Create a model and apply some settings.
```

(continues on next page)

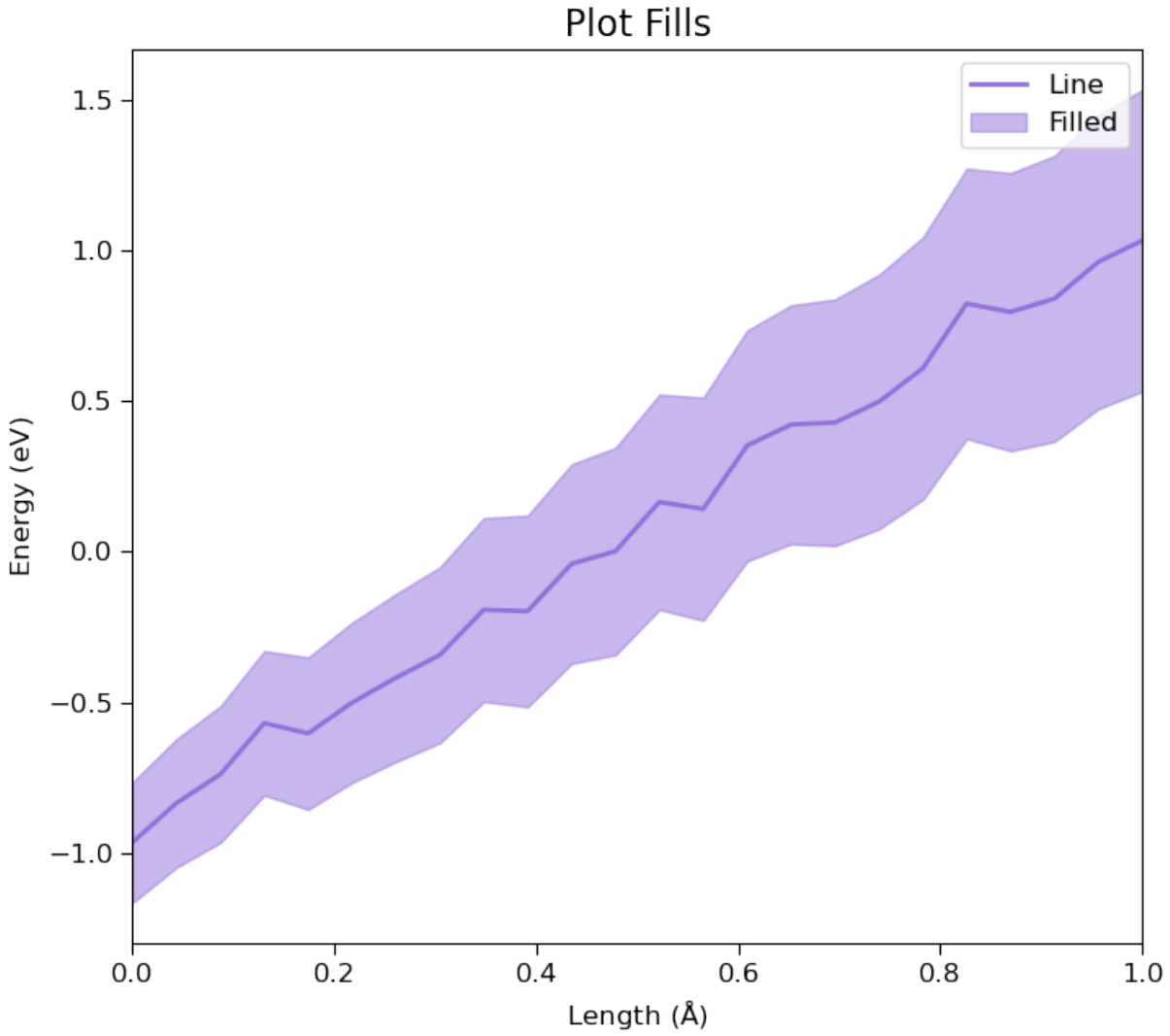
(continued from previous page)

```
model = Plot.PlotModel(x_unit=Angstrom)
model.framing().setTitle('Plot Density')
model.xAxis().setLabel('x')
model.yAxis().setLabel('y')

# Change x- and y-axis autoscale padding.
model.xAxis().setAutoscalePadding(0, 0)
model.yAxis().setAutoscalePadding(0, 0)

# Create and add density with a multiline resampler.
density = Plot.Density(xs, ys)
density.setWeights(weights)
# The resampler upsamples the lines from 256 to 4000 points.
density.setResampler(Plot.MultiLineResampler(sampling=4000))
density.setColor('purple')
density.setBroadening(2)
density.setDpi(72)
density.setLimits(0, 2)

model.addItem(density)

# Autoscale.
model.setLimits()

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'density_for_weighted_lines.png')
```

References: *Plot.Density*, *Plot.PlotModel*, *Plot.show*, *Plot.MultiLineResampler*, *Plot.save*.

Go back to *Plots*

Filled Area




```
# Plot filled area
# This example creates a simple plot with a line and a fill area.

# Generate some dummy data.
position = numpy.linspace(0, 1, 24) * Angstrom
energy = (numpy.linspace(-1, 1, 24) + 0.2 * numpy.random.rand(24)) * eV
energy_error = numpy.linspace(0.2, 0.5, 24) * eV

# Create a model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Plot Fills')
model.legend().setVisible(True)
model.xAxis().setLabel('Length')
model.yAxis().setLabel('Energy')

# Create line.
```

(continues on next page)

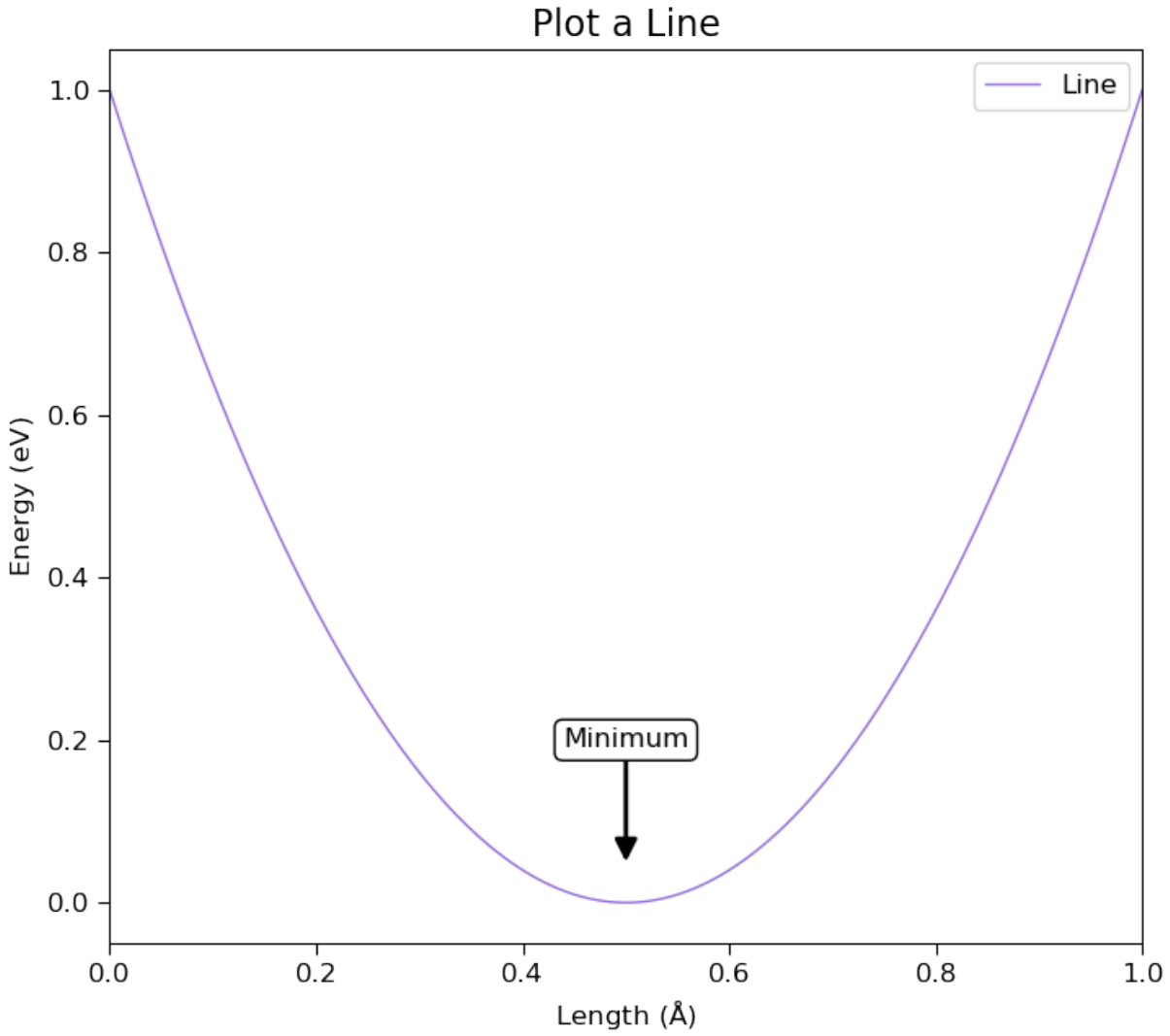
(continued from previous page)

```
line = Plot.Line(position, energy)
line.setColor('mediumpurple')
line.setLabel('Line')
line.setLineWidth(2)
model.addItem(line)

# Create and add fill area.
fill = Plot.HorizontalFill(position, energy - energy_error, energy + energy_error)
fill.setLabel('Filled')
fill.setColor('mediumpurple')
fill.setAlpha(0.5)
model.addItem(fill)

# Autoscale.
model.setLimits()

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'filled.png')
```

References: *Plot.Line*, *Plot.PlotModel*, *Plot.show*, *Plot.save*, *Plot.HorizontalFill*.

Go back to *Plots*

Line And Annotations




```
# Plot line with annotations.
# This example creates a simple plot with a line.

# The line data.
position = numpy.linspace(0, 1, 256) * Angstrom
energy = numpy.linspace(-1, 1, 256) ** 2 * eV

# Create a model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Plot a Line')
model.xAxis().setLabel('Length')
model.yAxis().setLabel('Energy')
model.legend().setVisible(True)

# Create and add line.
line = Plot.Line(position, energy)
```

(continues on next page)

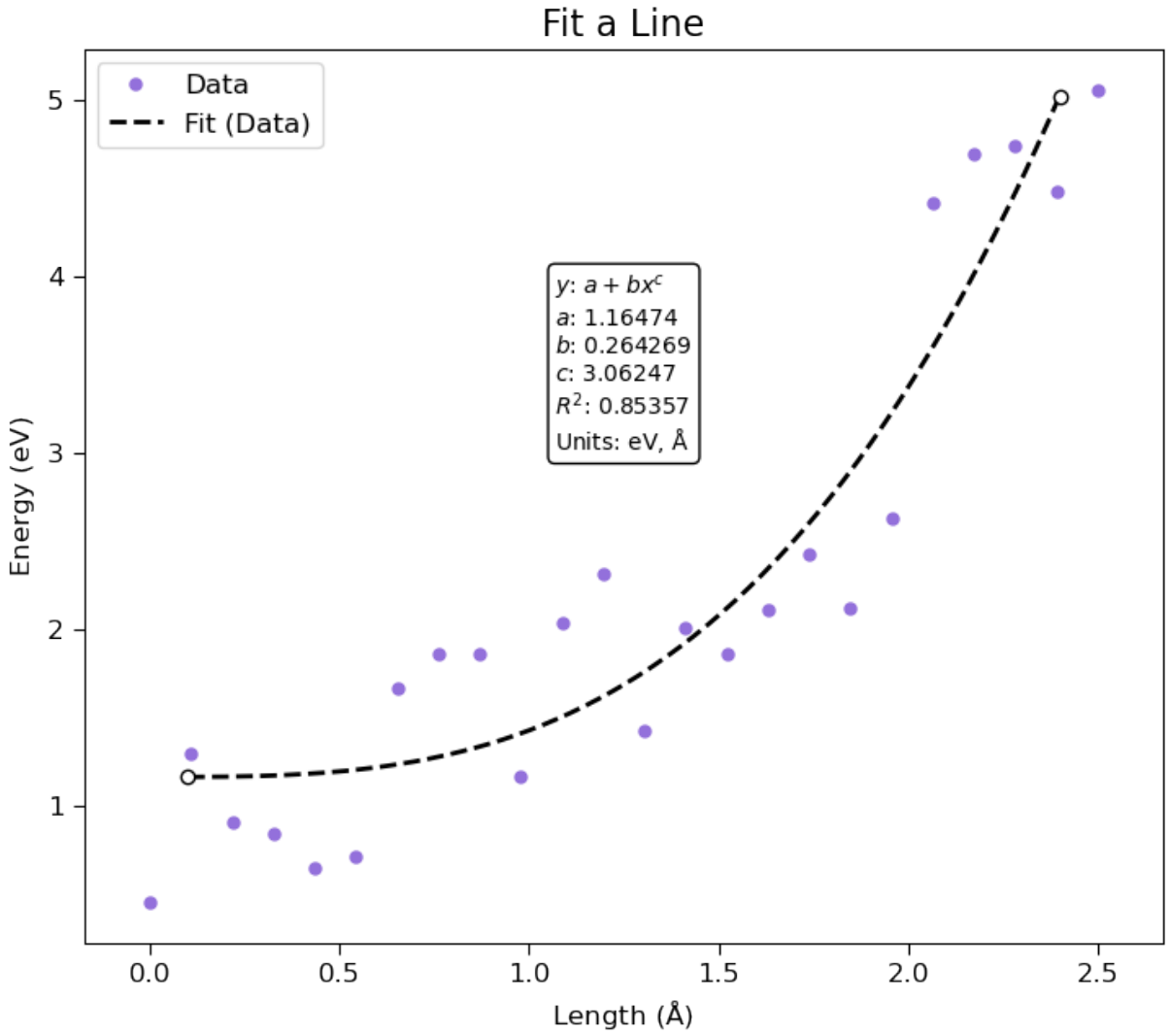
(continued from previous page)

```
line.setLabel('Line')
line.setColor('mediumpurple')
model.addItem(line)

# Add arrow and label annotations.
model.addItem(Plot.Arrow(0.5 * Angstrom, 0.2 * eV, 0.5 * Angstrom, 0.05 * eV))
model.addItem(Plot.Label(0.5 * Angstrom, 0.2 * eV, 'Minimum'))

# Autoscale limits.
model.setLimits()

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'line_and_annotations.png')
```

References: *Plot.Label*, *Plot.Line*, *Plot.Arrow*, *Plot.PlotModel*, *Plot.show*, *Plot.save*.

Go back to *Plots*

Line With Fit




```
Data fitted to a + b * x ** c
OrderedDict([('a', 1.164741495616713), ('b', 0.26426911018527083), ('c', 3.
↪06247474310537)])
```

```
# Plot a line and fit a curve.
# This example creates a plot and a fitted curve with a custom expression.
# The fit expression and fit parameters are then displayed in an
# AttributesLabel in the plot.

# The line data.
position = [
    0.        , 0.10869565, 0.2173913 , 0.32608696, 0.43478261, 0.54347826,
    0.65217391, 0.76086957, 0.86956522, 0.97826087, 1.08695652, 1.19565217,
    1.30434783, 1.41304348, 1.52173913, 1.63043478, 1.73913043, 1.84782609,
    1.95652174, 2.06521739, 2.17391304, 2.2826087 , 2.39130435, 2.5
] * Angstrom
```

(continues on next page)

(continued from previous page)

```

energy = [
    0.45292599, 1.29450797, 0.90624412, 0.84560608, 0.64978286, 0.71261064,
    1.67142063, 1.85748419, 1.85836352, 1.16352569, 2.03563087, 2.31325527,
    1.43093584, 2.01424388, 1.86664553, 2.11120909, 2.42706843, 2.12015567,
    2.62814134, 4.41887562, 4.69864691, 4.73923266, 4.47810888, 5.0582645
] * eV

# Create a model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Fit a Line')
model.xAxis().setLabel('Length')
model.xAxis().setAutoscalePadding(0.07, 0.07)
model.yAxis().setLabel('Energy')

model.legend().setVisible(True)

# Create line.
line = Plot.Line(position, energy)
line.setLabel('Data')
line.setMarkerStyle('o')
line.setMarkerFaceColor('mediumpurple')
line.setMarkerEdgeColor('mediumpurple')
line.setLineStyle(None)

# Create fit and add to line.
fit = Plot.CurveFit('a + b * x ** c')
line.addItem(fit)

# By default the fit range contains the entire line. The range can be modified
# interactively by calling Plot.show(model) and dragging the limits, or it can
# be done in script:
fit.setBounds(0.1 * Angstrom, 2.4 * Angstrom)

# Add an AttributesLabel to display the curve fit attributes in the plot.
attributes = Plot.AttributesLabel(1.25 * Angstrom, 3.5 * eV)
fit.addItem(attributes)

# Get results.
print('Data fitted to ' + fit.expression())
print(fit.parameters())

# Add line to plot.
model.addItem(line)

# Autoscale limits.
model.setLimits()

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
```

(continues on next page)

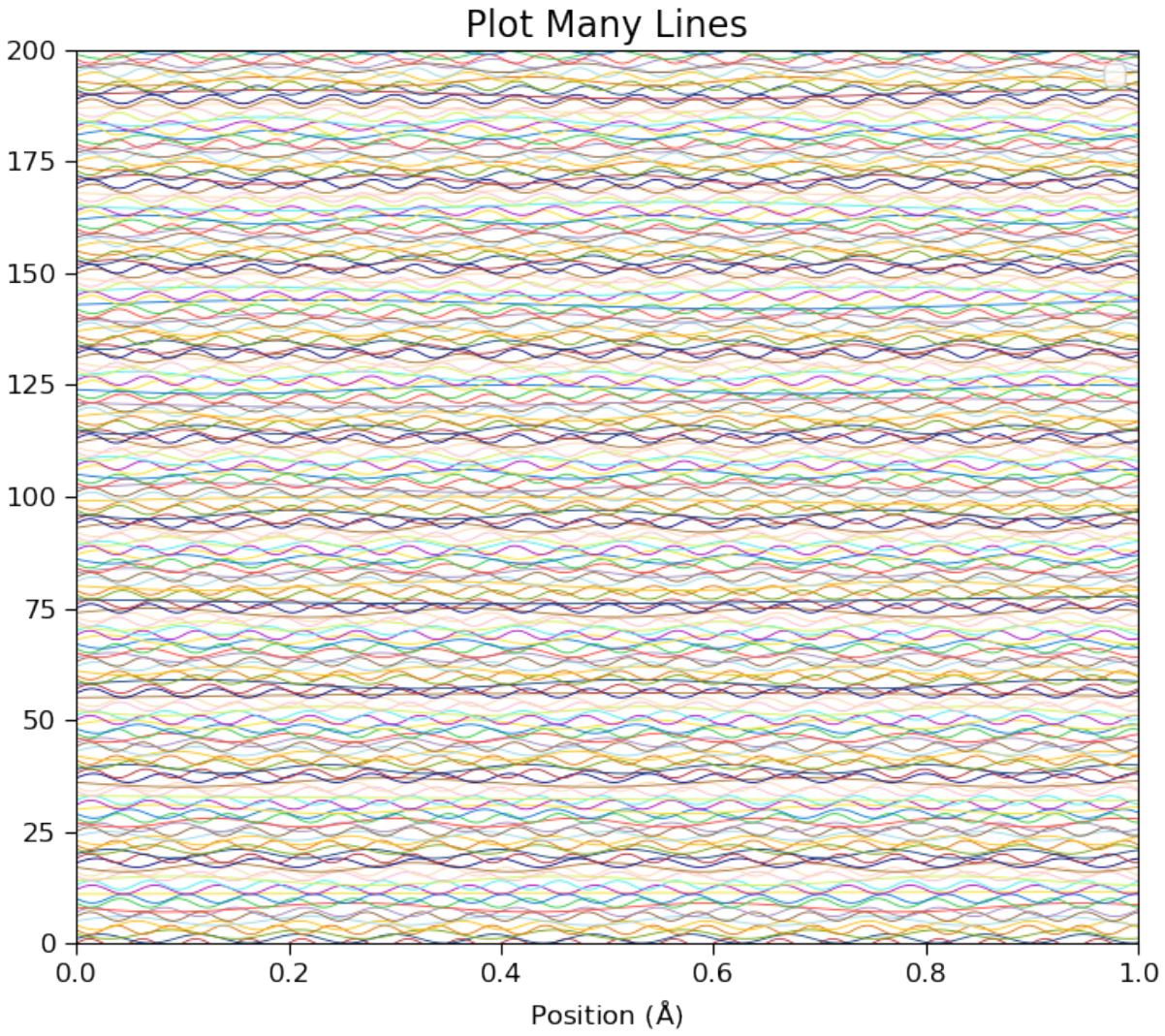
(continued from previous page)

```
Plot.save(model, 'line_with_fit.png')
```

References: *Plot.Line*, *Plot.AttributesLabel*, *Plot.CurveFit*, *Plot.PlotModel*, *Plot.show*, *Plot.save*.

Go back to *Plots*

Linegroup




```
# Plot thousands of lines with a LineGroup.

# Dummy data
xs = numpy.linspace(0, 1, 256) * Angstrom

# Create oscillating lines.
constants = numpy.arange(1024)
xs_mesh, constants_mesh = numpy.meshgrid(xs.inUnitsOf(Angstrom), constants)
ys = numpy.sin(40 * numpy.pi * numpy.cos(constants_mesh) * xs_mesh) + constants_mesh
```

(continues on next page)

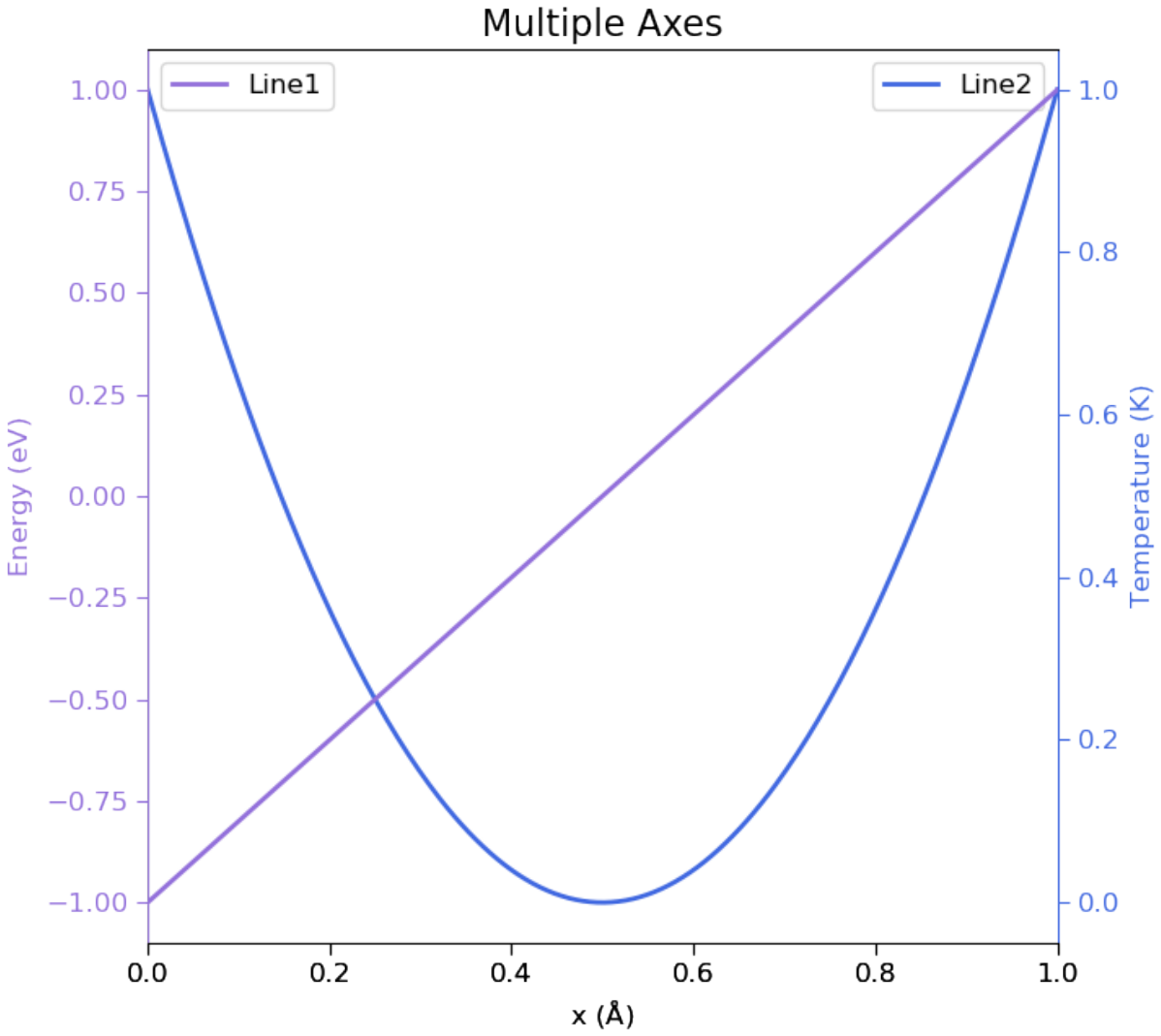
(continued from previous page)

```
10
11 # Create and tweak model.
12 model = Plot.PlotModel(x_unit=Angstrom)
13 model.xAxis().setLabel('Position')
14 model.framing().setTitle('Plot Many Lines')
15 model.legend().setVisible(True)
16
17 # Create line group and lines.
18 group = Plot.LineGroup()
19 model.addItem(group)
20
21 for i in range(1024):
22     line = Plot.Line(xs, ys[i, :])
23     line.setLineWidth(0.6)
24     line.setColor(Plot.Colors[i])
25     group.addItem(line)
26
27 # Autoscale.
28 model.setLimits('y', 0, 200)
29
30 # Show the plot for interactive editing.
31 Plot.show(model)
32
33 # Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
34 Plot.save(model, 'linegroup.png')
```

References: *Plot.LineGroup*, *Plot.Colors*, *Plot.Line*, *Plot.PlotModel*, *Plot.show*, *Plot.save*.

Go back to *Plots*

Multiple Axes




```
# Plot lines in two different plots, with overlaid axis.

# Data
xs = numpy.linspace(0, 1, 256) * Angstrom
ys1 = numpy.linspace(-1, 1, 256) * eV
ys2 = numpy.linspace(1, -1, 256) ** 2 * Kelvin

# Create model1 and add line.
model1 = Plot.PlotModel(Angstrom, eV)
model1.framing().setTitle('Multiple Axes')
model1.framing().setBorders(show_right_border=False)
model1.xAxis().setLabel('x')
model1.yAxis().setLabel('Energy')
model1.yAxis().setColor('mediumpurple')
model1.legend().setVisible(True)

```

(continues on next page)

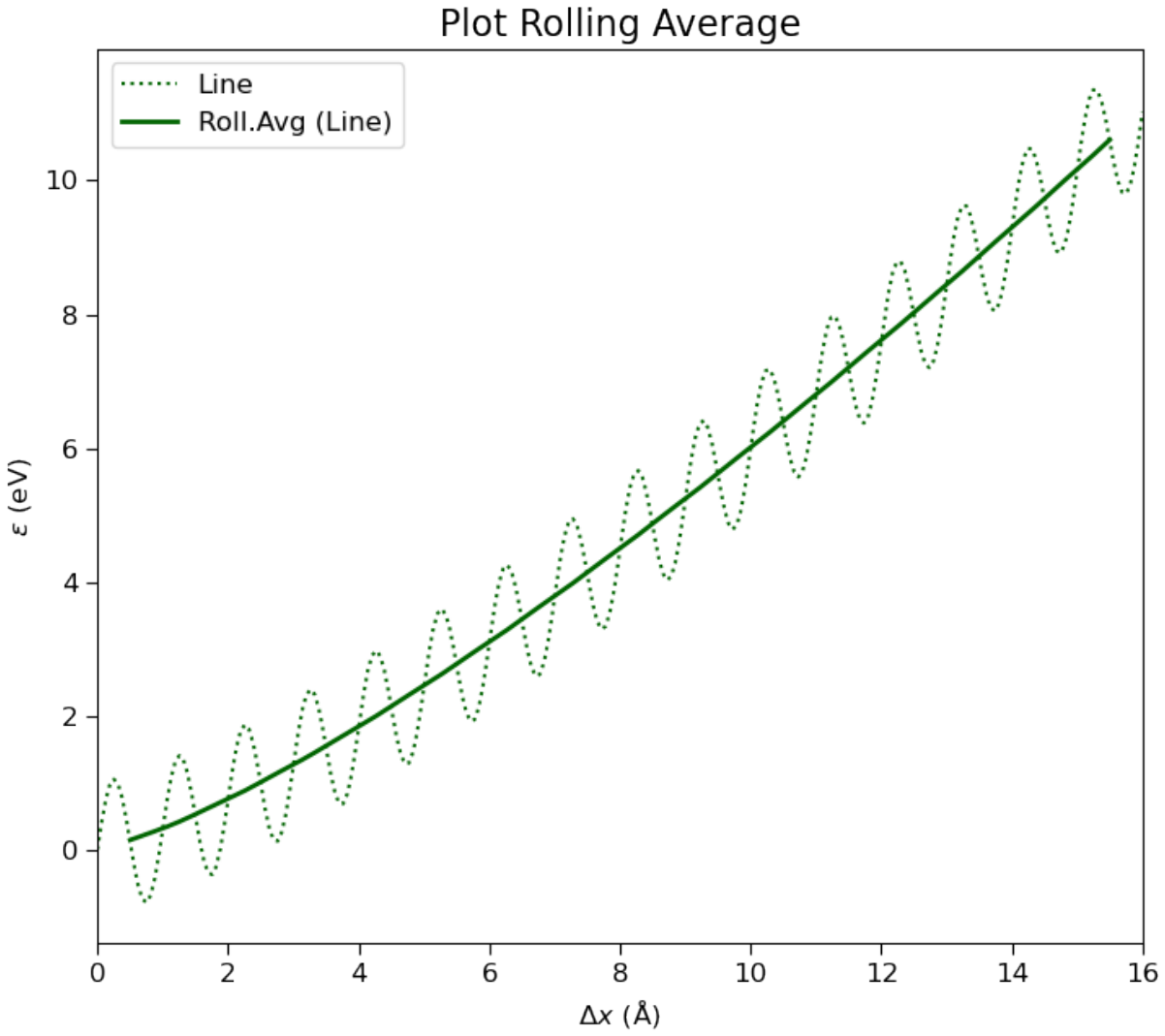
(continued from previous page)

```
line1 = Plot.Line(xs, ys1)
line1.setLabel('Line1')
line1.setColor('mediumpurple')
line1.setLineWidth(2)
model1.addItem(line1)

# Create model2 and add line.
model2 = Plot.PlotModel(Angstrom, Kelvin)
model2.xAxis().setLabel('x')
model2.yAxis().setLabel('Temperature')
model2.yAxis().setMirrored(True)
model2.yAxis().setColor('royalblue')
model2.legend().setVisible(True)

line2 = Plot.Line(xs, ys2)
line2.setLabel('Line2')
line2.setColor('royalblue')
line2.setLineWidth(2)
model2.addItem(line2)

# Autoscale.
model1.setLimits()
model2.setLimits()

# Create layout.
layout = Plot.OverlayLayout()
layout.setMode(Plot.LAYOUT_MODES.SHARE_X)

# Add main model.
frame1 = Plot.PlotFrame(use_frame_model=False)
frame1.addModel(model1)
layout.addItem(frame1)

frame2 = Plot.PlotFrame(use_frame_model=False)
frame2.addModel(model2)
layout.addItem(frame2)

# Show the plot for interactive editing.
Plot.show(layout)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(layout, 'multiple_axes.png')
```

References: *Plot.OverlayLayout*, *Plot.Line*, *Plot.PlotModel*, *Plot.show*, *Plot.PlotFrame*, *Plot.save*.

Go back to *Plots*

Rolling Average




```
# Plot a line and use a Rolling Average to smooth oscillations.

# Dummy data with an overlaid oscillation.
x = numpy.linspace(0, 16, 256) * Angstrom
y = (0.3 * (x / Angstrom) ** 1.3 + numpy.sin(2 * numpy.pi * x / Angstrom)) * eV

# Create and tweak model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Plot Rolling Average')
model.xAxis().setLabel(r'$\Delta x$')
model.yAxis().setLabel(r'$\varepsilon$')
model.legend().setVisible(True)

# Create and add line.
line = Plot.Line(x, y)
line.setLabel('Line')
```

(continues on next page)

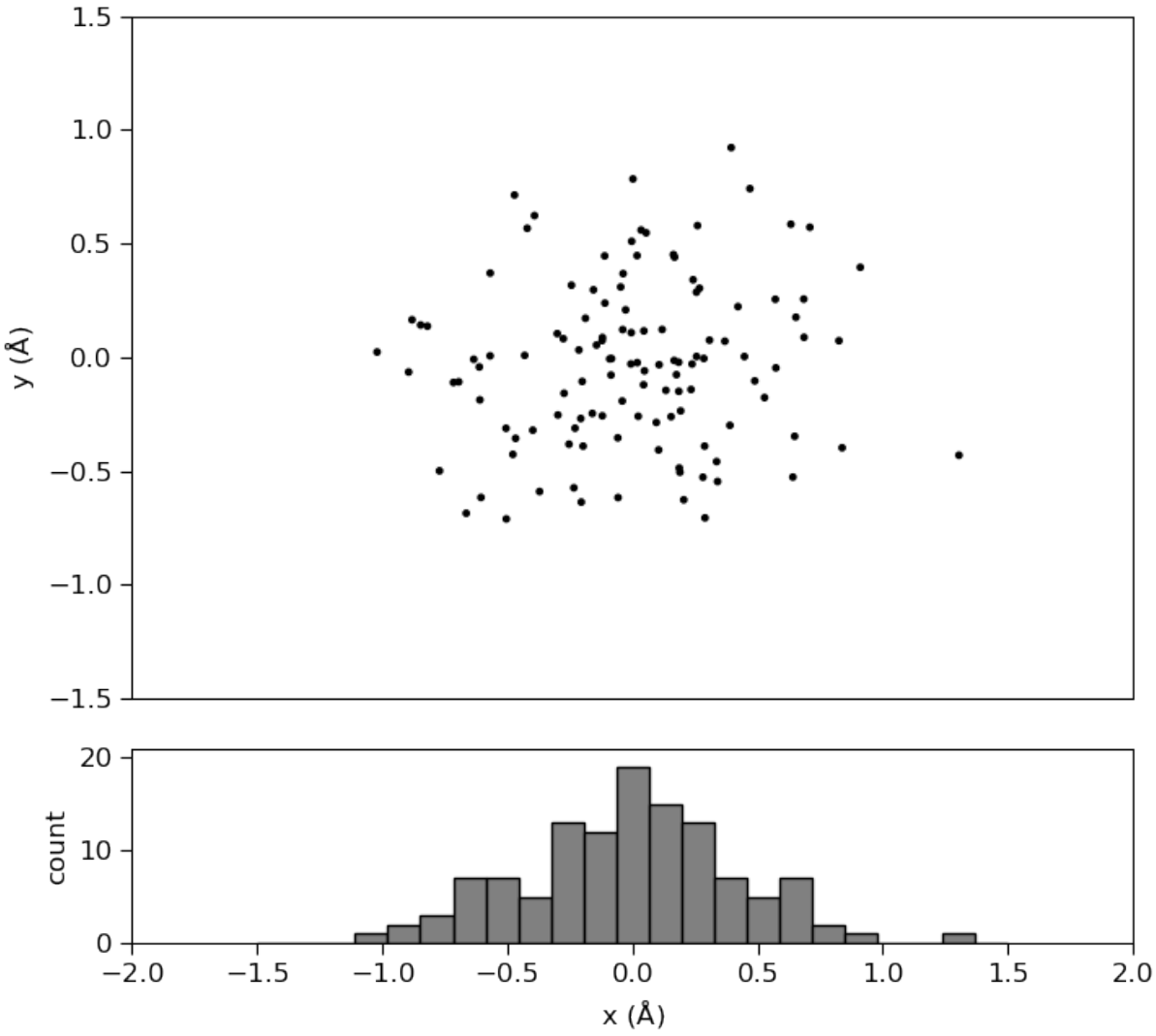
(continued from previous page)

```
line.setColor('darkgreen')
line.setLineWidth(1.5)
model.addItem(line)

# Create and add rolling average with a window of 16 points.
roll = Plot.RollingAverage(window=16)
line.addItem(roll)
line.setLineStyle(':')

# Autoscale.
model.setLimits()

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'rolling_average.png')
```

References: *Plot.Line*, *Plot.PlotModel*, *Plot.show*, *Plot.RollingAverage*, *Plot.save*.

Go back to *Plots*

Scatter And Bars




```
# Plot a scatter plot and histogram of the same data, in two different
# plot frames.

# Data
xs = [
     0.83735136, -0.03760732,  0.64764615, -0.56916293,  0.16771351,  0.36879221,
    -0.60997142,  0.25521883,  0.11830685,  0.25563545, -0.12038023, -0.37151165,
    -0.39852745,  0.04760468,  0.01874401, -0.14411779, -0.47167539,  0.09492484,
    -1.0206757 , -0.20679976,  0.64055442, -0.11149991, -0.8195419 , -0.71561298,
     0.20467069,  0.04404555, -0.12067043, -0.00652415,  0.1842815 ,  1.30424258,
     0.18665118,  0.33963749, -0.50445047,  0.23366367,  0.16632812, -0.18822706,
    -0.03928118,  0.28480758, -0.77132285, -0.46731356, -0.00351658,  0.57264734,
     0.4690202 ,  0.39410025, -0.05785553,  0.38895173, -0.60553757,  0.65265759,
     0.68512054,  0.82472603, -0.42096647, -0.39180296,  0.13309577, -0.08609454,
     0.2812388 ,  0.90992544, -0.0474478 , -0.05890777, -0.63456302, -0.2440444 ,
    -0.88108586, -0.08369874,  0.2674326 , -0.43100448, -0.56871021,  0.52732416,
```

(continues on next page)

(continued from previous page)

```
    -0.69443328, -0.20111549, -0.84669947, -0.16063656,  0.16344137,  0.10416594,
     0.03419954, -0.6121973 ,  0.57048945, -0.23452936, -0.27359465,  0.24200477,
     0.04523835, -0.25387563, -0.66462862, -0.29825646,  0.10609641, -0.02768023,
     0.19217027,  0.70849361,  0.01809639, -0.09136528,  0.05425135, -0.47819754,
    -0.27667665,  0.02260149, -0.12150936, -0.04109076,  0.33587457,  0.6325594 ,
     0.48836006,  0.28791998, -0.30082755,  0.15389378,  0.44696974,  0.42120839,
    -0.1975828 , -0.22986685,  0.18926104, -0.89475401, -0.11048363, -0.21441636,
    -0.50558513,  0.30749674, -0.15624408,  0.00163378, -0.20476512,  0.28942828,
     0.23776839, -0.00519347,  0.18484562,  0.17532173,  0.2595802 ,  0.68402716
] * Angstrom

ys = [
    -0.39715421,  0.36792564, -0.34670964,  0.00663009,  0.44020473,  0.07102315,
    -0.18666689,  0.28657916,  0.12237538,  0.00391008,  0.08724919, -0.5892403,
    -0.31992659, -0.05920487, -0.023152  ,  0.05400336,  0.71322692, -0.28597838,
     0.02357725, -0.2685703 , -0.52583313,  0.44584184,  0.13713652, -0.11021891,
    -0.62578892, -0.12033264, -0.25678569, -0.02885627, -0.02127524, -0.42985407,
    -0.48594983, -0.54523483, -0.70977806, -0.14108246, -0.01369695,  0.17182939,
     0.12188442, -0.00476866, -0.49848981, -0.35582173,  0.50985858, -0.04619683,
     0.74132604,  0.92157524, -0.61588049, -0.29860316, -0.61532293,  0.17643255,
     0.08809562,  0.07311062,  0.56711457,  0.62286086, -0.14466861, -0.07766289,
    -0.52641952,  0.39565981,  0.30934238, -0.35355341, -0.00837116,  0.31705944,
     0.16510211, -0.00542853,  0.30364316,  0.00874412,  0.3702261 , -0.17669301,
    -0.10746404, -0.10582702,  0.14214246, -0.24628123,  0.45151432, -0.4065585 ,
     0.559458  , -0.04214145,  0.25515366, -0.57307117, -0.15778443,  0.34097717,
     0.11634075, -0.38137623, -0.68484207, -0.25304297, -0.03233032,  0.20868762,
    -0.23503984,  0.57160049,  0.44713183, -0.00655961,  0.54723515, -0.42624796,
     0.08212732, -0.25878557,  0.07400322, -0.19167848, -0.4577302 ,  0.58497679,
    -0.10335792, -0.38983413,  0.1043278 , -0.2605955 ,  0.00364975,  0.22308884,
    -0.38997689, -0.31123183, -0.50457249, -0.06438019,  0.23835865,  0.03252147,
    -0.31160181,  0.0756    ,  0.29678598,  0.78384221, -0.63570551, -0.7051451 ,
    -0.02914059,  0.10867163, -0.14926761, -0.07602382,  0.57902892,  0.25606282
] * Angstrom

# Create models.
model = Plot.PlotModel(Angstrom, Angstrom)
model.xAxis().setVisible(False)
model.yAxis().setLabel('y')

model_x = Plot.PlotModel(x_unit=Angstrom)
model_x.xAxis().setLabel('x')
model_x.yAxis().setLabel('count')
model_x.yAxis().setAutoscalePadding(0, 0.1)

# Build and add scatter plot
scatter = Plot.Scatter(xs, ys)
scatter.setLabel('Scatter')

model.addItem(scatter)

# Add histogram as for x-values.
x_bins = numpy.linspace(-1.5, 1.5, 24, endpoint=True) * Angstrom
```

(continues on next page)

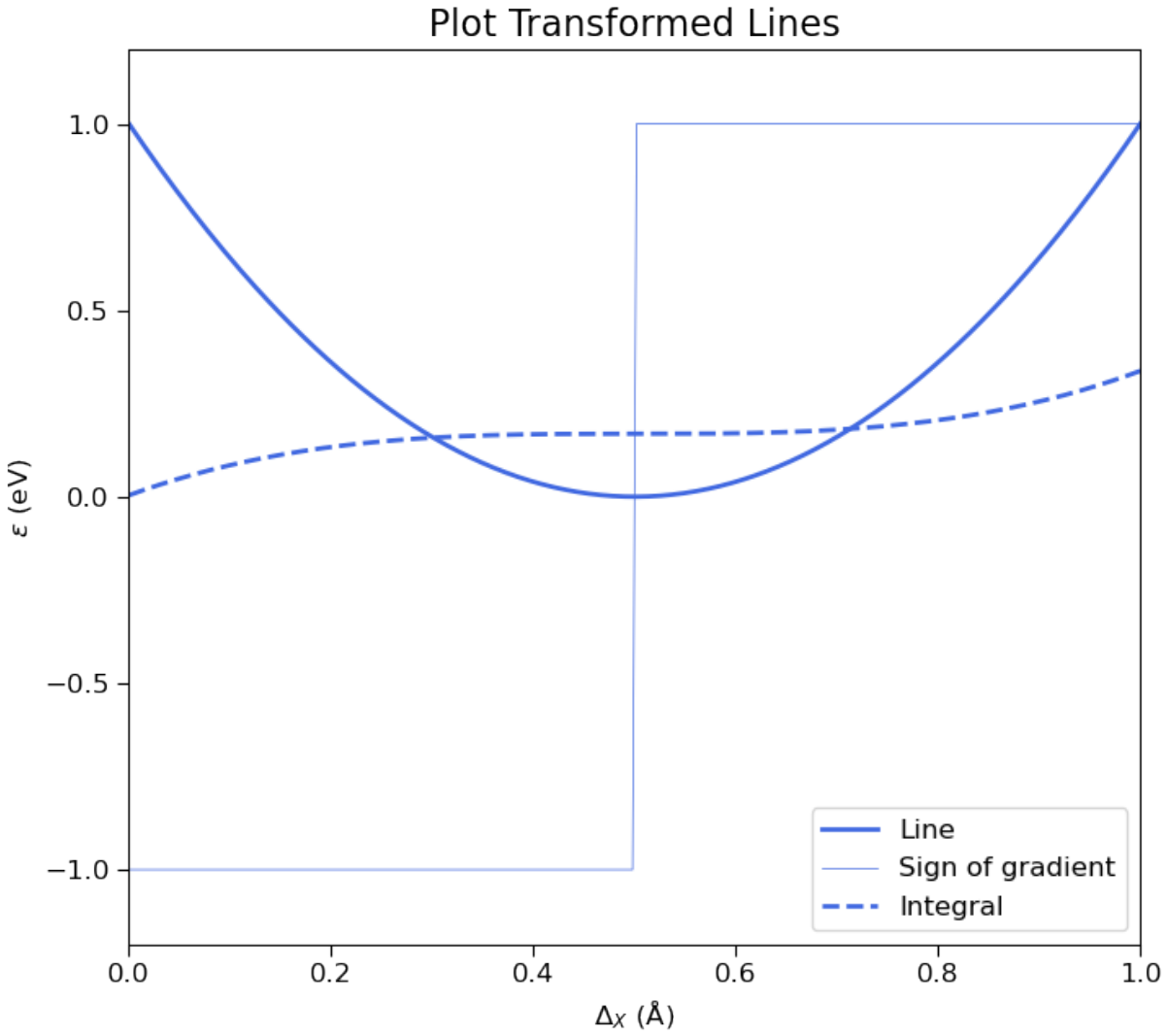
(continued from previous page)

```
x_counts = numpy.histogram(
    xs.inUnitsOf(Angstrom),
    x_bins.inUnitsOf(Angstrom)
)[0]
model_x.addItem(Plot.VerticalBar(x_bins, x_counts))

# Change limits.
model.setLimits('x', -2, 2)
model.setLimits('y', -1.5, 1.5)
model_x.setLimits('y')

# Create layout.
layout = Plot.GridLayout(rows=4, columns=1)
layout.setMode(Plot.LAYOUT_MODES.SHARE_X)

# Add main model.
frame = Plot.PlotFrame(use_frame_model=False)
frame.addModel(model)
layout.addItem(frame, row=0, column=0, height=3, width=1)

# Add x model.
layout.addModel(model_x)

# Show the plot for interactive editing.
Plot.show(layout)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(layout, 'scatter_and_bars.png')
```

References: *Plot.VerticalBar*, *Plot.PlotModel*, *Plot.Scatter*, *Plot.GridLayout*, *Plot.show*, *Plot.PlotFrame*, *Plot.save*.

Go back to *Plots*

Transformed Line




```python
# Plot a line with two transformed lines.

# Dummy data.
x = numpy.linspace(0, 1, 256) * Angstrom
y = numpy.linspace(-1, 1, 256) ** 2 * eV

# Create and tweak model.
model = Plot.PlotModel(Angstrom, eV)
model.framing().setTitle('Plot Transformed Lines')
model.xAxis().setLabel(r'$\Delta_X$')
model.yAxis().setLabel(r'$\varepsilon$')
model.legend().setVisible(True)

# Create line and add to model.
line = Plot.Line(x, y)
line.setLabel('Line')
```

(continues on next page)

(continued from previous page)

```
line.setLineWidth(2)
line.setColor('royalblue')
model.addItem(line)

# Create step.
step = Plot.TransformedLine(y_expression='sign(gradient(y/eV)) * eV')
line.addItem(step)
step.setLabel('Sign of gradient')
step.setLineWidth(0.5)

# Create cumulative sum.
integral = Plot.TransformedLine(y_expression='cumsum(y) * gradient(x/Angstrom)')
line.addItem(integral)
integral.setLabel('Integral')
integral.setLineStyle('--')

# Autoscale.
line.setVisible(True)
model.setLimits('y', -1.2, 1.2)

# Show the plot for interactive editing.
Plot.show(model)

# Save the plot (can also be saved to svg, pdf, jpeg or hdf5).
Plot.save(model, 'transformed_line.png')
```

References: *Plot.TransformedLine*, *Plot.Line*, *Plot.PlotModel*, *Plot.show*, *Plot.save*.

Go back to *Plots*

Working with Plots in NanoLab

Editing, Saving and Merging Plots

Version: Q-2019.12

This guide shows you how to take advantage of NanoLab plotting capabilities to customize, save, merge and re-use plots.

The QuantumATK NanoLab plugins allow you to produce a large variety plots that visualize Bandstructures, Projected Density of States (PDOS), ElectronPhononCouplings, Potentials, etc. Each of these plots can be edited, saved and loaded witin NanoLab. Plots can also be merged together in compound plots that visualize complex relationships between configurations, calculations and analyses. Lastly, those compound plots can be re-used for other calculations and results.

Note: As a simple example this guide uses the Projected DOS and Bandstructure analyses calculated for bulk silicon with a LCAO calculator using the GGA-1/2 PBE exchange correlation.

If you wish to, you may repeat the analysis by downloading the data files here:

- `bulk_si_gga_half_pbe.hdf5`
- `bulk_si_gga_pbe.hdf5`
- `all_plots.hdf5`

Video guide

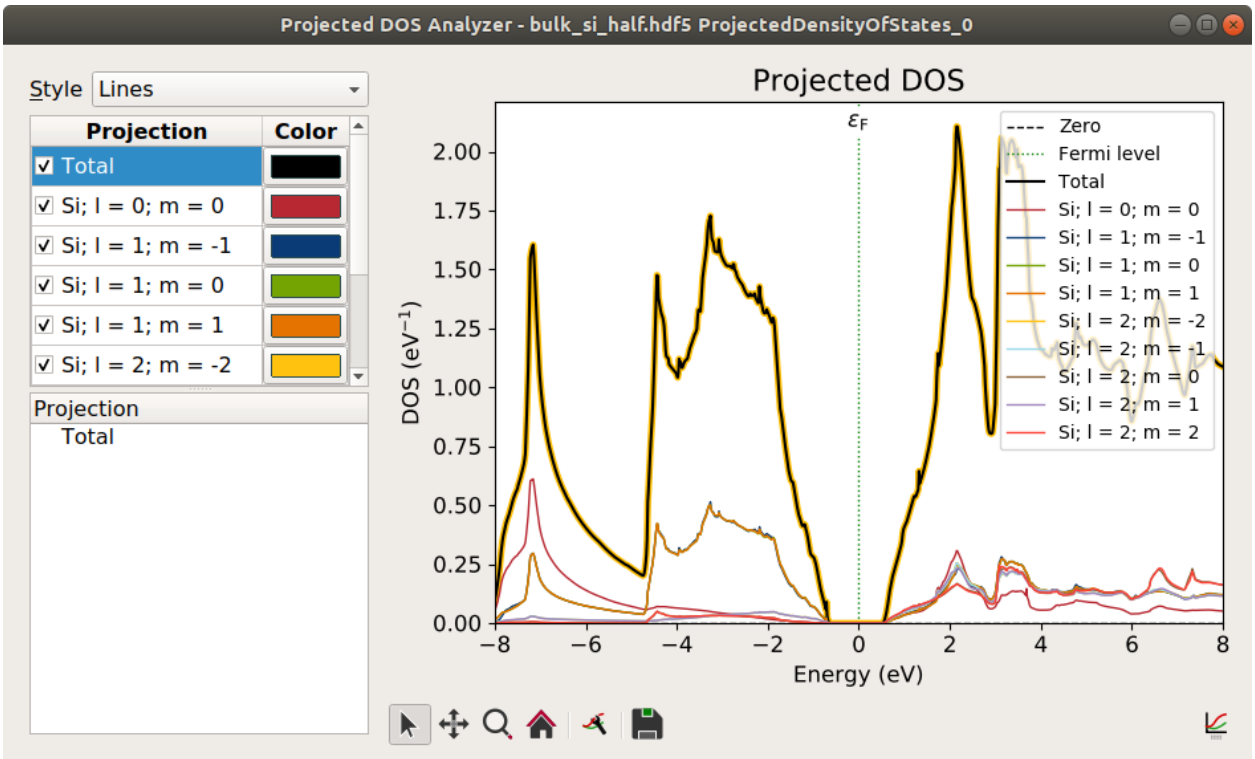
This is a video guide showcasing the NanoLab plotting capabilities. Detailed information is provided in the written guide below.

Plot tools

To facilitate easy editing, all plots are accompanied by a plot tool bar. The toolbar contains these tools:

1. **Interaction tool**, where items can be selected. Right-click on a plot to add arrows, measures and labels to the plot.
2. **Pan tool**. Left-click and drag a plot to change the axis limits. Right-click and drag to zoom.
3. **Zoom tool**. Right-click and drag to draw a zoom-rectangle. Use the mouse wheel to zoom in and out.
4. **Editor tool**. Click this tool bring up the Plot Editor.
5. **Home tool**. Clicking this tool resets the current axes limits.
6. **Save tool**. Click to save the interactive plot in hdf5, export data to txt, or export the plot as an image in e.g. pdf or png formats.
7. **Merge tool**. Drag this icon to merge the plot with itself or other plots.

The PDOS (Projected Density of States) Analyzer plots the density of states projected onto elements, orbitals or shells. For our example bulk silicon calculation the PDOS analyzer looks like this:



Annotate plots

The **Interaction tool** shows helpful tool tips when hovering over different plot items. Once hovered, left clicking selects the item. Right clicking brings up the annotation menu with these options:

- **Autoscale** the plot.
- **Add arrow** annotation.
- **Add label** annotation.
- **Add measure** (general, vertical or horizontal).

When right clicking on a line, the menu also includes these additional options:

- **Add analysis** (average, rolling average, polynomial fit, gaussian fit or a generic curve fit).

Tip: For our example Project DOS plot, one common analysis is to measure the band gap. This can be done by inserting a vertical measure into the plot. The meausure can then be snapped to the lines near the band edges.

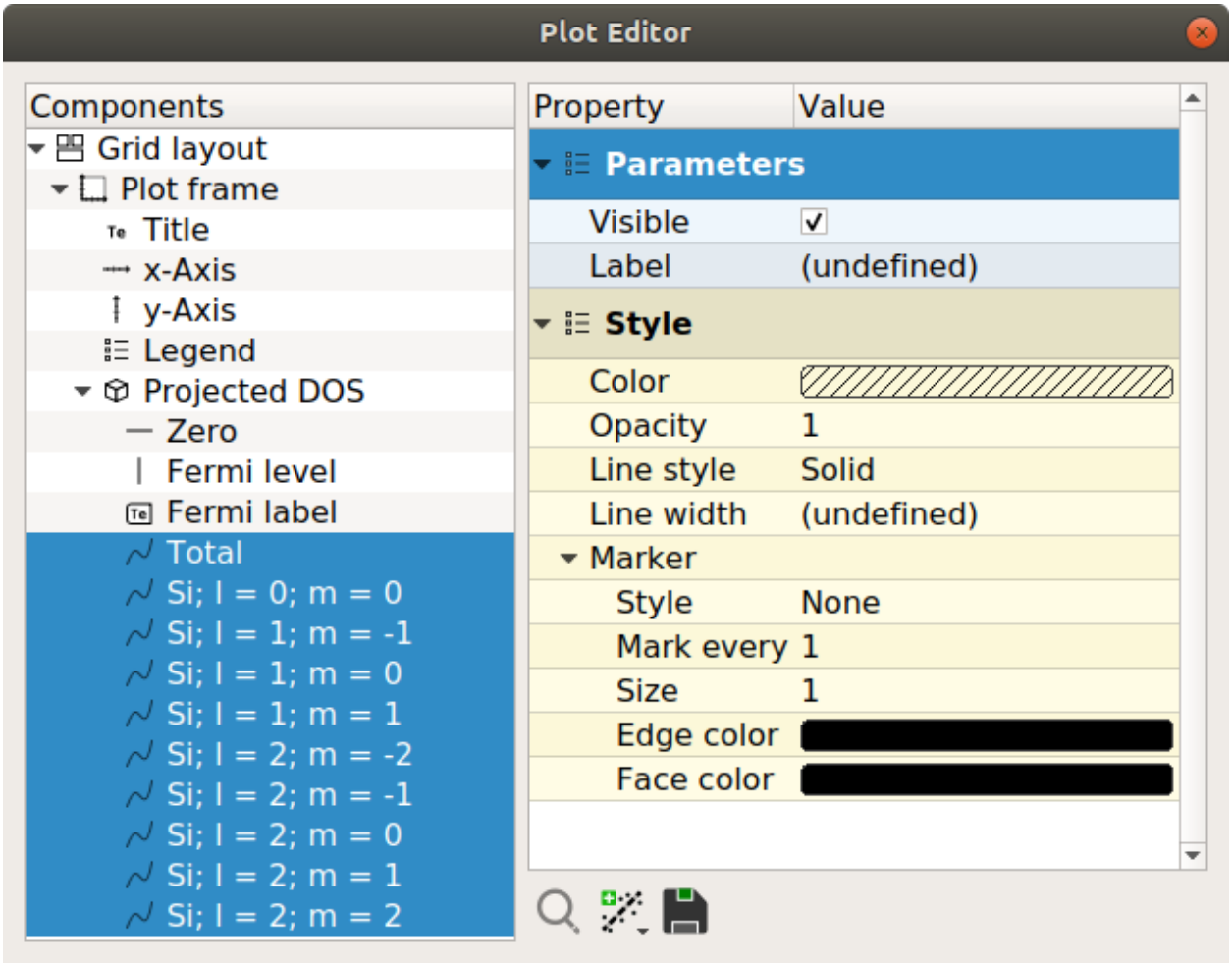
The Plot Editor

Launch the plot editor from the toolbar or by pressing E. The plot editor makes it possible to edit the individual properties of the layout, legend, title, axes, and lines of the plot.

The editor lists all the plot components. At the top level all plots are built from a GridLayout. The layout then contains one (or multiple) Plot Frames. The dimensions of the grid and the position of the individual frames can all be adjusted in the Plot Editor.

Each frame has a title, a legend, an x-axis and a y-axis, and may also contain data from one or more sources. Simply select the different components to edit their color, line-width, text, opacity and so forth.

One such source could be our Projected DOS plot, which consists of some projection lines, a vertical Fermi line and a Fermi label. You can select multiple lines and edit all their properties at once.



The editor also has a few additional tools:

1. **Zoom to item**. Select a single item and zoom to that item.

2. **Add analysis**. Lines support additional analysis that you may add here. You can choose between polynomial fits, gaussian fits, or specify your own fitting function in the curve fit. You can also add an average or a rolling average analysis.

3. Export data. Select the relevant plot items and choose the save icon. This will export data to txt format.

Save and load the plot in the PlotViewer

After editing the plot, you can save it to a file in the hdf5 format. In the file browser MainWindow you can then check the newly created file and the Plot will appear on the Lab floor. Open the Plot using the **PlotViewer** plugin in the Analysis Bar located to the right of the Lab floor.

The opened plot contains all your exiting modifications. You can continue editing, or you can re-open the original analyzer (in our example this would be the PDOS analyzer).

To re-open the analyzer, click the red analyzer icon. This tries to open the original PDOS analyzer. Sometimes the original analysis object is required in order to re-open the analyzer. If the analysis object is missing you can replace the data by dragging and dropping it from the Lab floor. See also the section about Redo analysis.

Note: In our case, we may wish to change from lines to filled stacked lines. This could be done in the PDOS analyzer, so we re-open the analyzer and choose the Stacked style. The PDOS plot is then immediately updated.

Merge plots

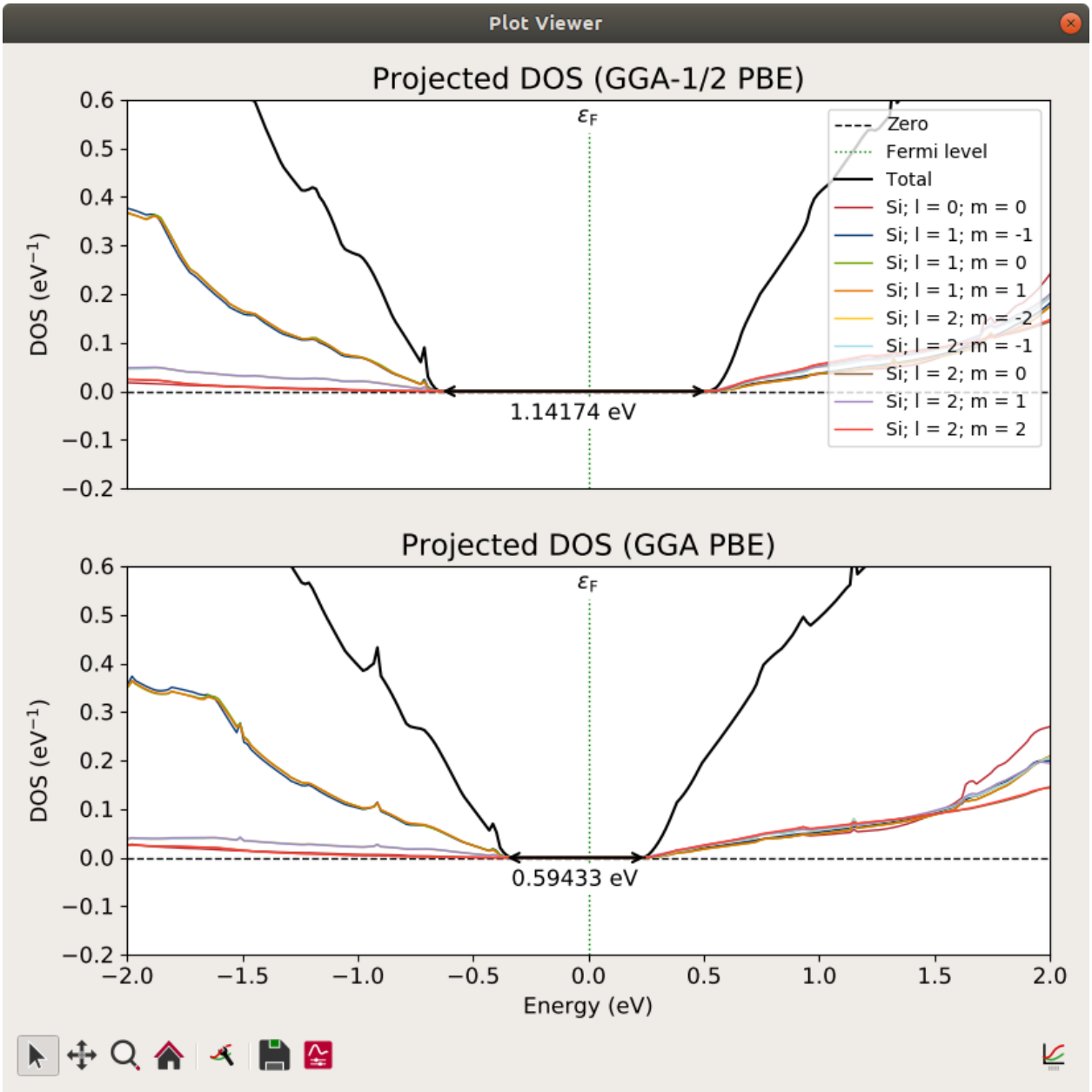
All plots can be merged together by dragging and dropping the plots onto each other. In a plot you can grab the plot by grabbing the merge icon in the plot toolbar, and on the Lab floor you simply grab the Lab floor icon.

In general the plotting framework supports two different ways of merging plots together depending on where you drop the plot.

Drop next to, above or below an existing plot

When dragging a plot, you can drop it on four green areas at the edges of a plot. This will create a new plot containing the original plot along with the dropped plot. Once merged both plots can still be edited and adjusted and their analyzers can still be re-opened.

In our example we also calculated the Projected DOS for bulk silicon with DFT-1/2 turned off. Dropping this PDOS plot next to the GGA-1/2 PDOS plot clearly shows that the two calculations produce a different band gap. Add two horizontal measures and adjust the titles to produce the plot below:



Note: Note that the DFT-1/2 result is very close to the experimentally measured band gap of bulk silicon of 1.14 eV.

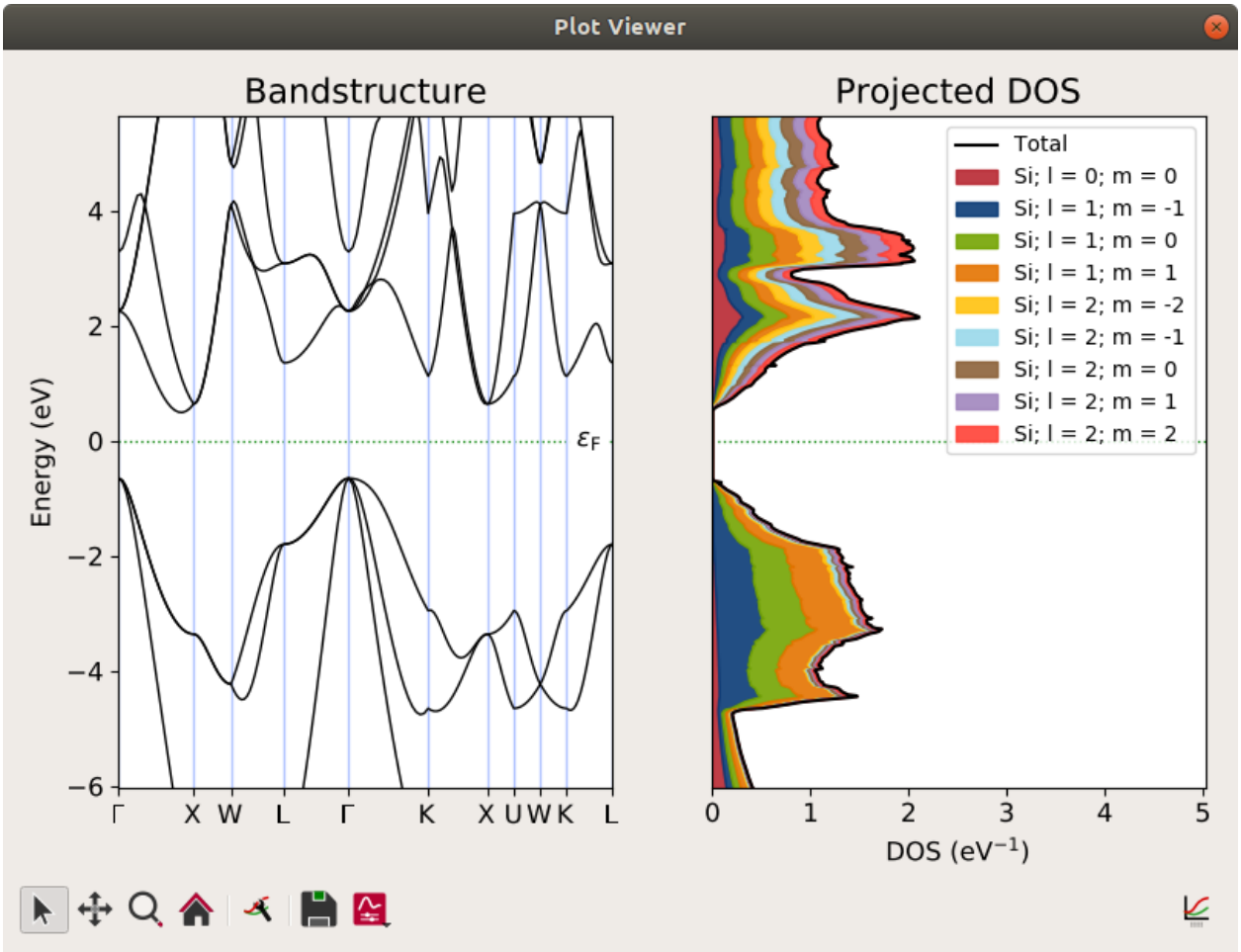
Drop on the existing plot

You can also drop a plot on the middle of another/the same plot. This offers a dialog with some advanced merge options. Depending on the units on the axes of your merged plots, you may have the following options:

1. Overlay with one or two shared axes.
2. Flip the plot and then overlay.
3. Mirror (plot next to each other but with linked axes)
4. Flip the plot and then mirror.

For bulk silicon we also calculated the Bandstructure, and we want to inspect the calculated Projected DOS and Bandstructure next to each other. We grab the merge icon in the Bandstructure plot and drop it on the PDOS plot. In the dialog we choose the option Flip and mirror with shared y-axis.

This creates the following merged plot where the y-axes (the energy axes) on both plots have been linked.



Redo analysis

By merging plots from multiple QuantumATK NanoLab plugins it is possible to create interesting compound plots showing different aspects of your calculated data. In our example the merged plot of the Bandstructure and Projected DOS is one such a plot.

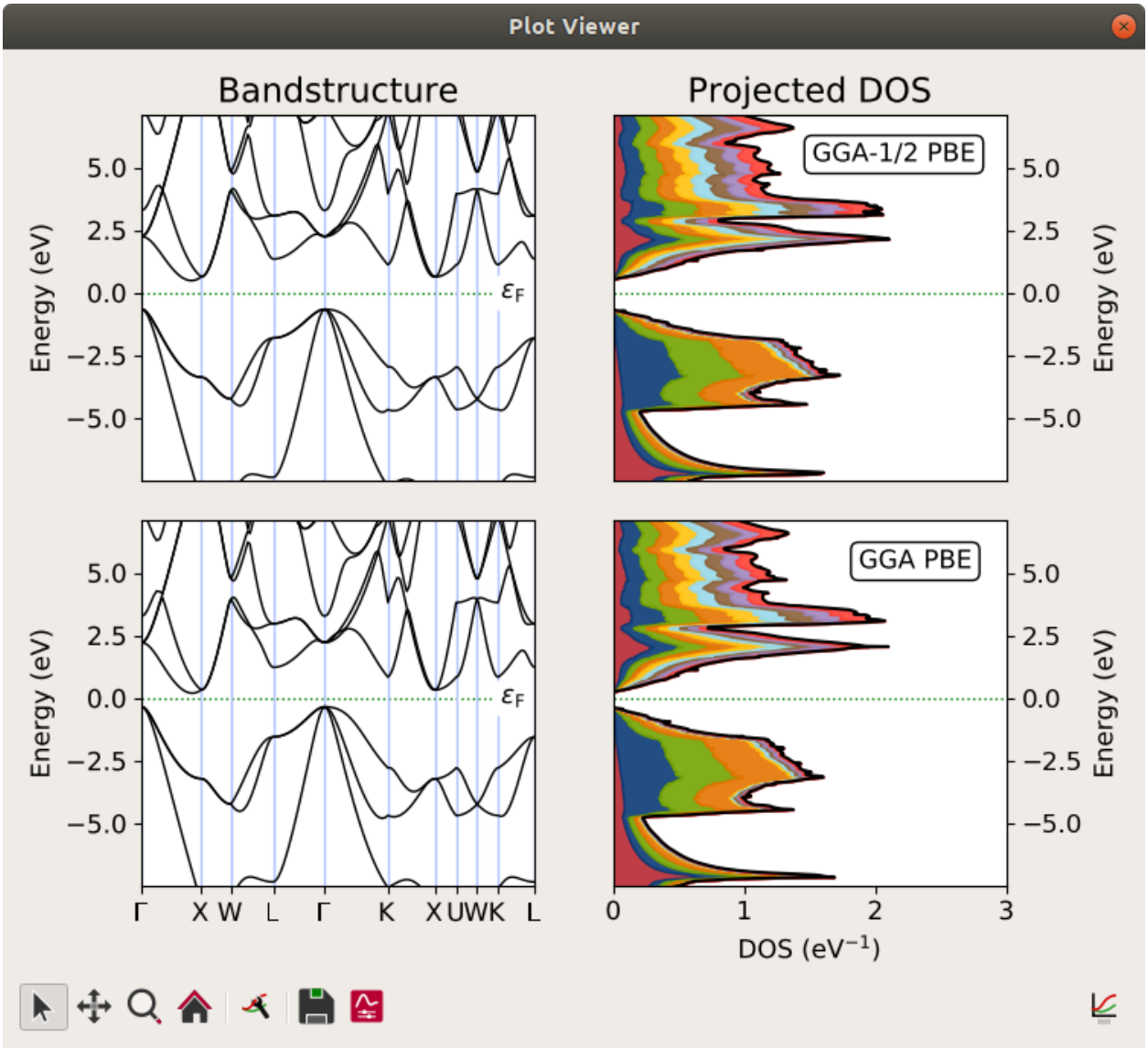
NanoLab allows you to reuse a compound plot. This means that you may easily redo this analysis for another set of calculated results. It is possible to replace data in the following way:

1. Select the data on the Lab floor.
2. Drop the data in the middle of the plot.
3. The plots are then updated with the new data.

You can also create two plots side by side for an easy comparison.

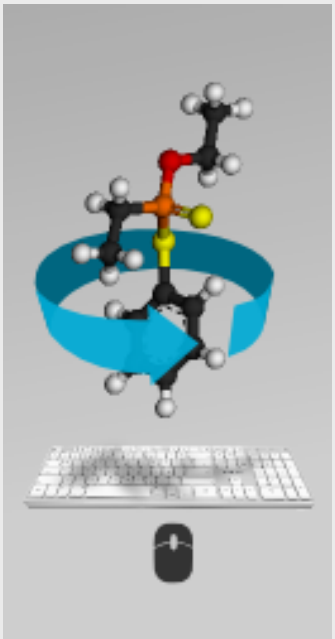
1. Select the data on the Lab floor.
2. Drop the data on the edge of the plot.
3. A new plot is created with the original and the new plot side by side.

In our analysis of bulk silicon we would e.g. select the PDOS and Bandstructure Lab floor items from the GGA PBE calculation (DFT-1/2 disabled) and drop them next to our merged plot. In the compound plot we edited the titles and added two labels to produce the following comparison.


Plot Viewer
Bandstructure
Projected DOS
GGA-1/2 PBE
GGA PBE
Energy (eV)
5.0
2.5
0.0
−2.5
−5.0
ε_F
Γ X W L Γ K X U W K L
0
1
2
3
DOS (eV^{-1})

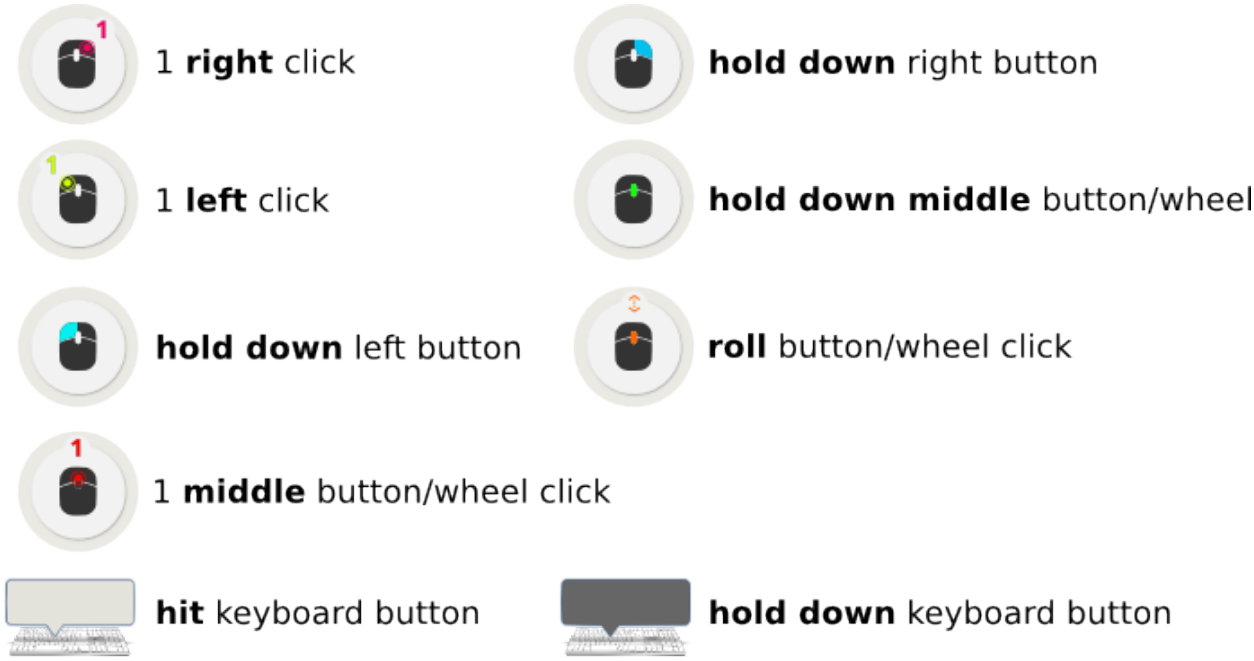
2.8 Manage 3D View and Select Atoms

QuantumATK: 3D functionalities Platforms: Links: BuilderTips2 *In this tutorial you will learn basic functionalities of the Builder and the Viewer. You will learn how to manage the 3D view, and how to select atoms.*



Note: MOUSE AND KEYBOARD HOTKEYS MEMO

The icons below show a list of mouse and keyboard hot keys we will use for the pictures of the tutorial.



The **Builder** and the **Viewer** provide beautiful and easy to use 3D view and selection functionalities. This tutorial explains how to use them.

2.8.1 Rotations

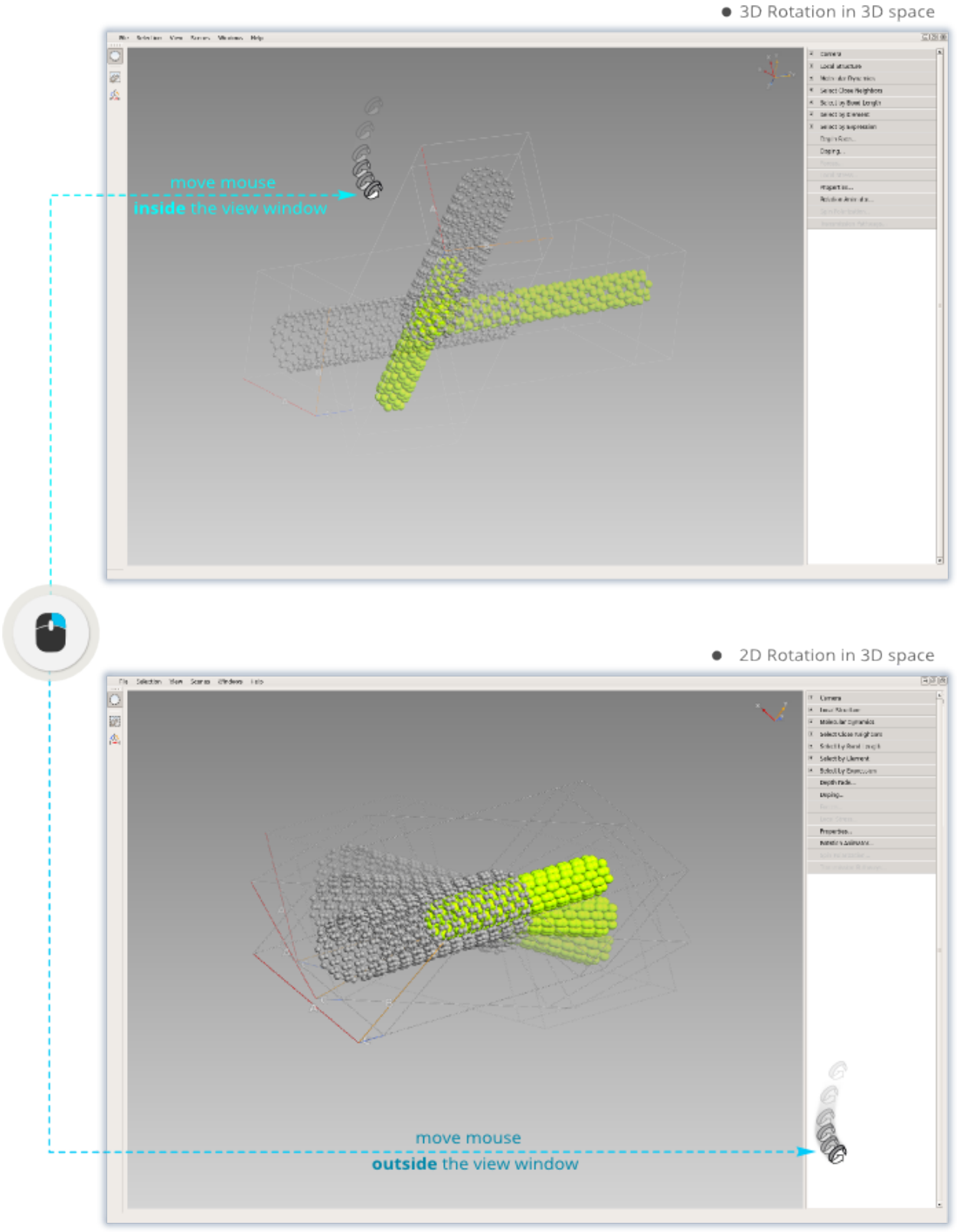
You can do **3D** rotations and **2D** rotations of any configuration: Simply **click and hold down the right mouse-button** and move the cursor using mouse.

Note: 3D and 2D rotations

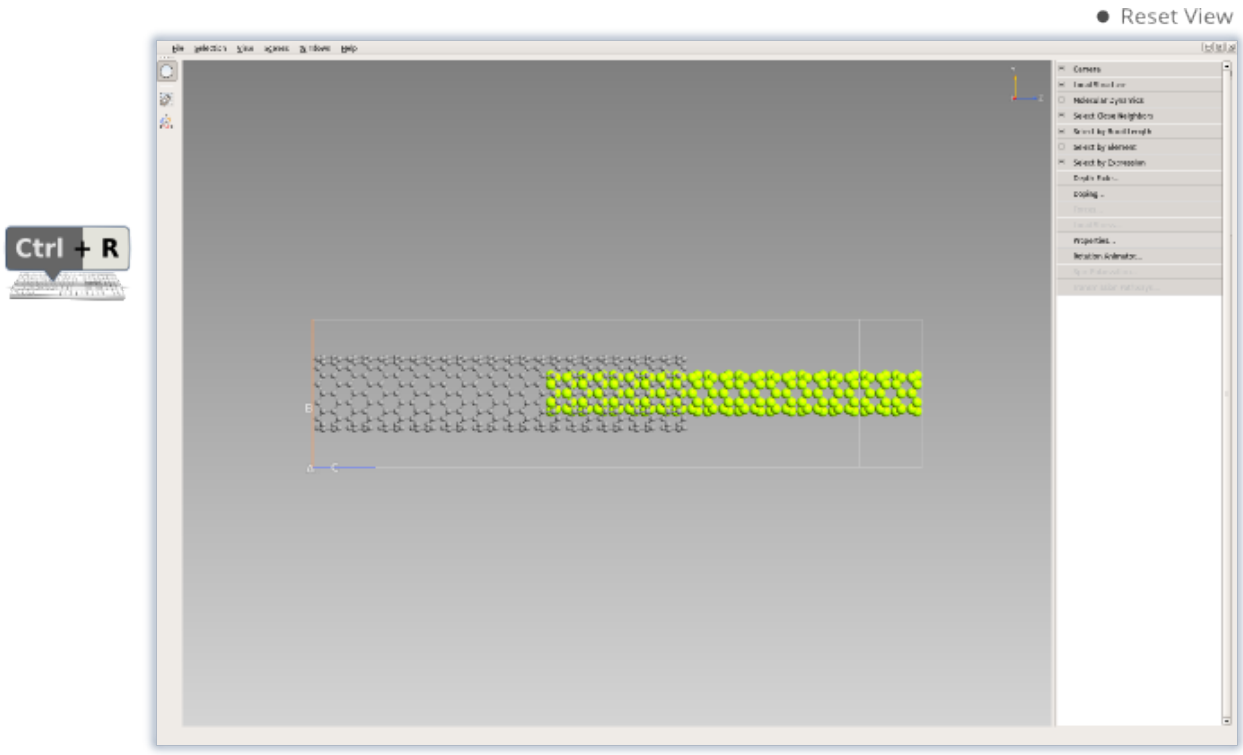
•moving the cursor **inside the Viewer window** gives you **3D** rotations of your 3D object.

•moving the cursor **outside the Viewer window** gives you **2D** rotations of your 3D object.

•Also note that the cursor changes when you are in rotate mode:

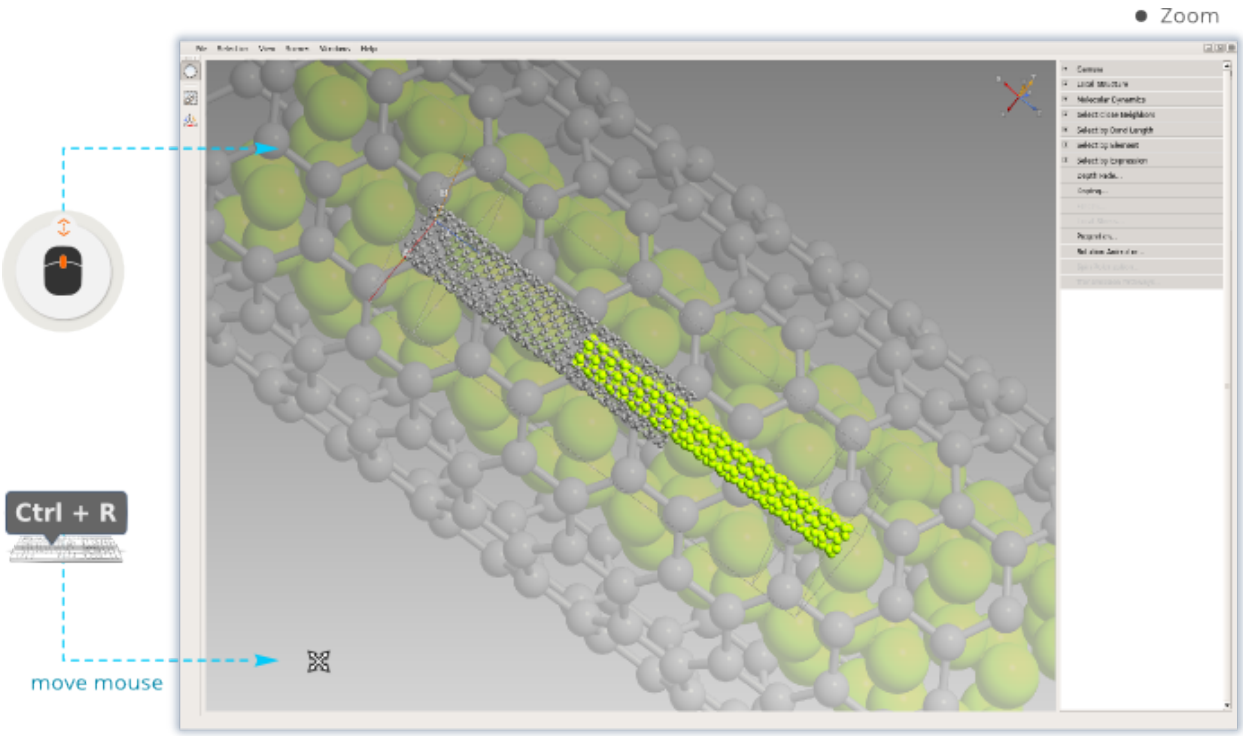

3D Rotation in 3D space
move mouse
inside the view window
2D Rotation in 3D space
move mouse
outside the view window

2.8.2 Reset View



Press **Ctrl+R** on your keyboard to reset the 3D view.

2.8.3 Zoom

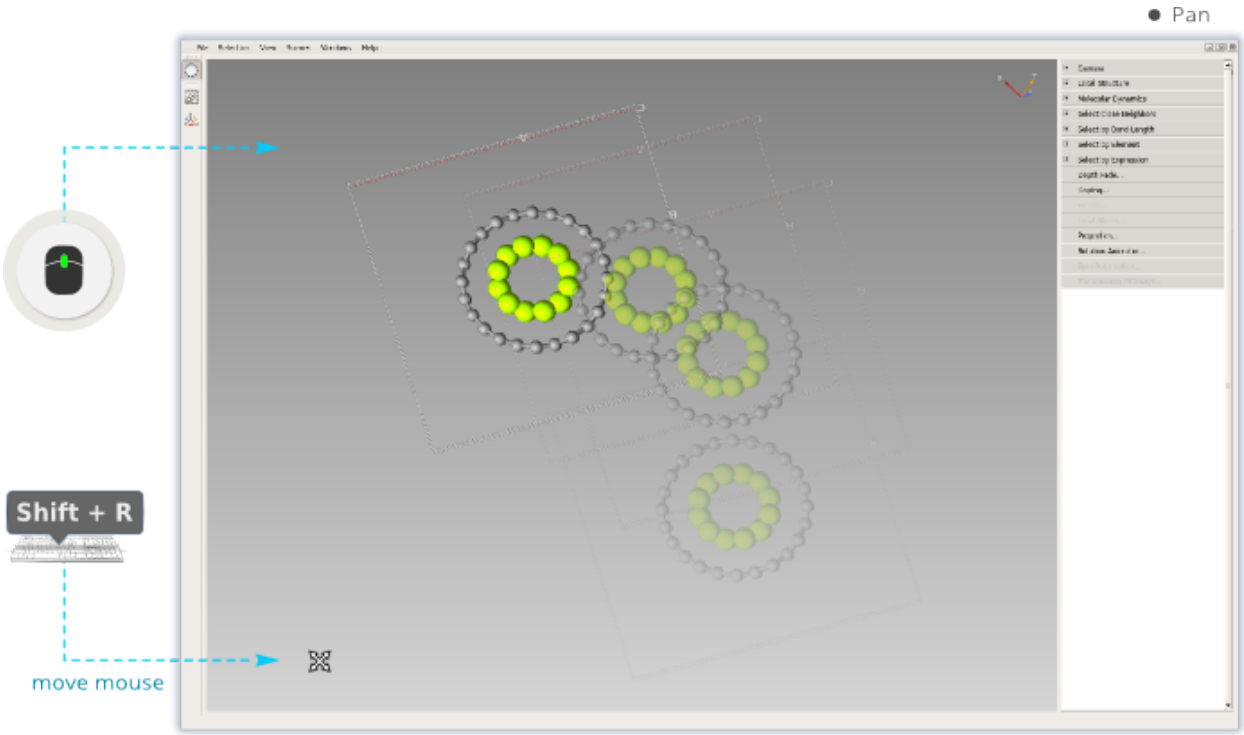


There are two ways to zoom:

1.Use the **mouse wheel**.

2.Hold down **Ctrl** and the **right mouse botton**, and move the mouse cursor up or down ( ).

2.8.4 Drag



There are two ways to toggle the drag function ( ) to move your structure around in the 3D view:

1.Hold down the **mouse middle botton/wheel**.

2.Hold down **Shift** and the **right mouse botton**.

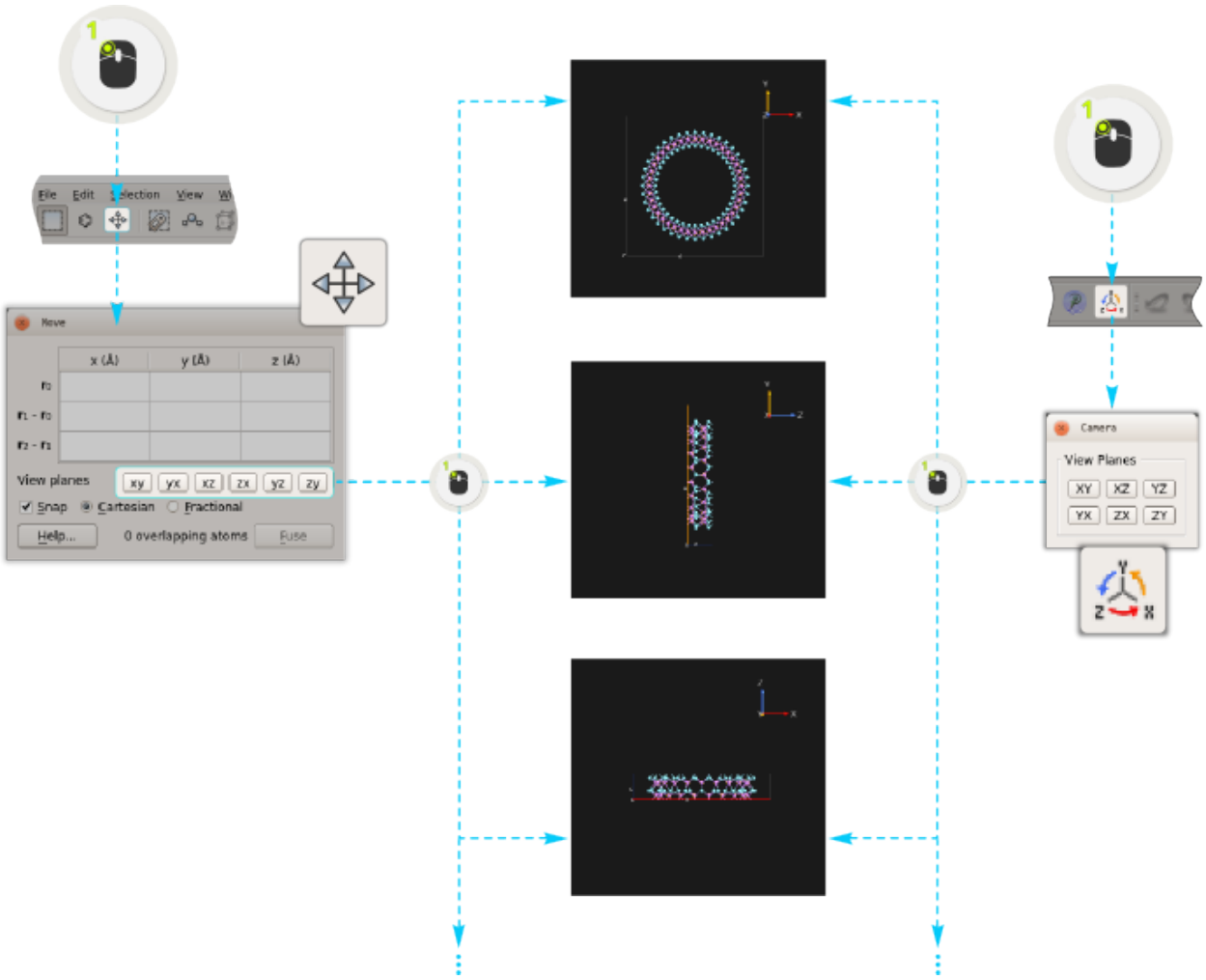
2.8.5 Camera and View Planes

Both the **Builder** and the **Viewer** have functionality that allows you to view a configuration in specific 2D planes: xy, yx, xz, zx, yz, zy.

Note: Camera and Move

•In **Viewer**, you can use the **Camera** plugin to select a 2D view plane.

•In **Builder**, you can use both the **Camera** and the **Move** tool for this.



Note that the **Move** tool puts you in *Move Tool Mode*. If you want to use another tool you must close it first!

2.8.6 Selections

The **QuantumATK** selection and de-selection functionality follows the logic of many other GUIs:

Attention:

•**(hold) Ctrl + [click to object(s)]** : to select object(s)

•**(hold) Shift+ [click to object(s)]** : to de-select object(s) (if they are in selected mode)

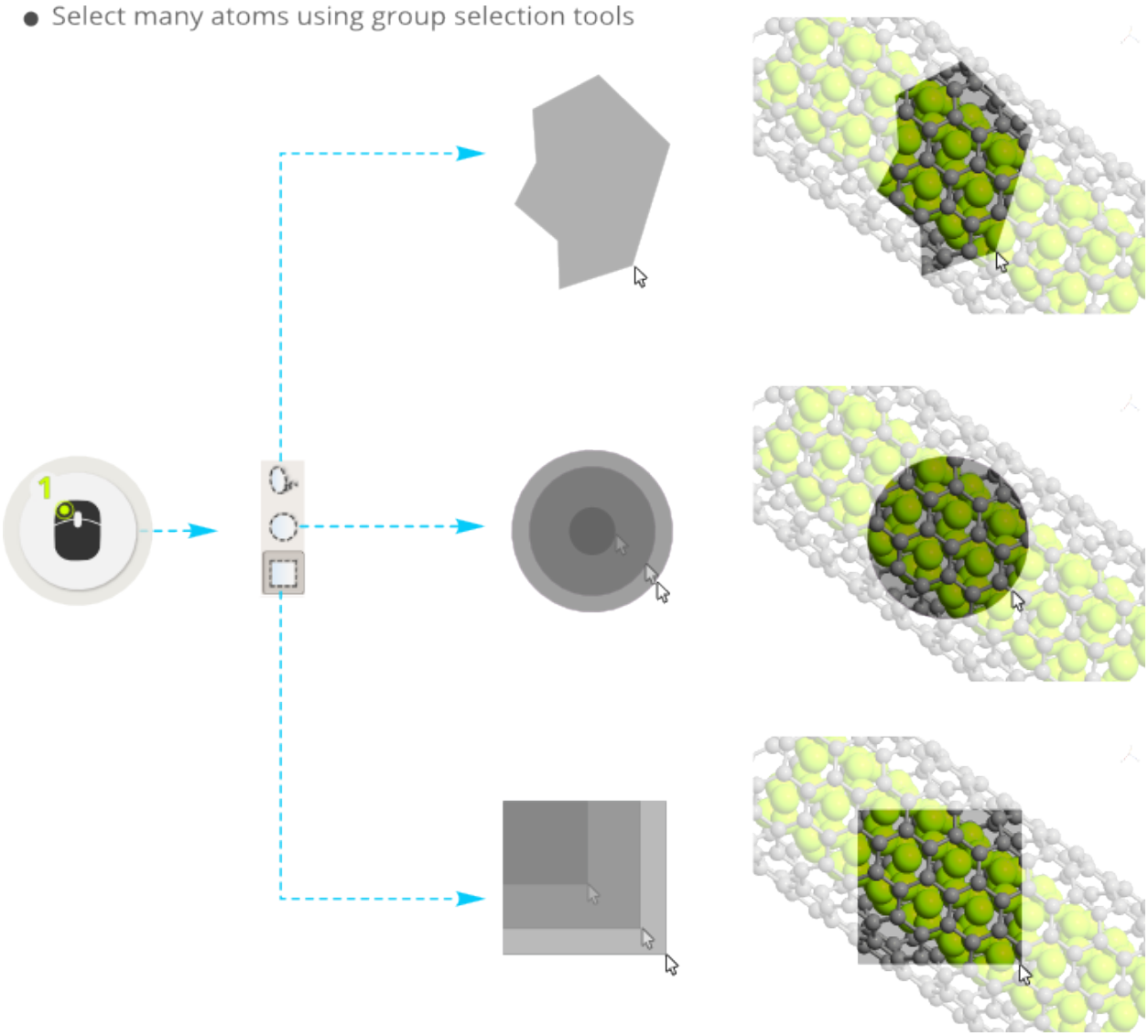
•**(hold) Ctrl + A** : to select all objects

•**Esc** : to de-select everything

•**[click on the background]** : to de-select everything

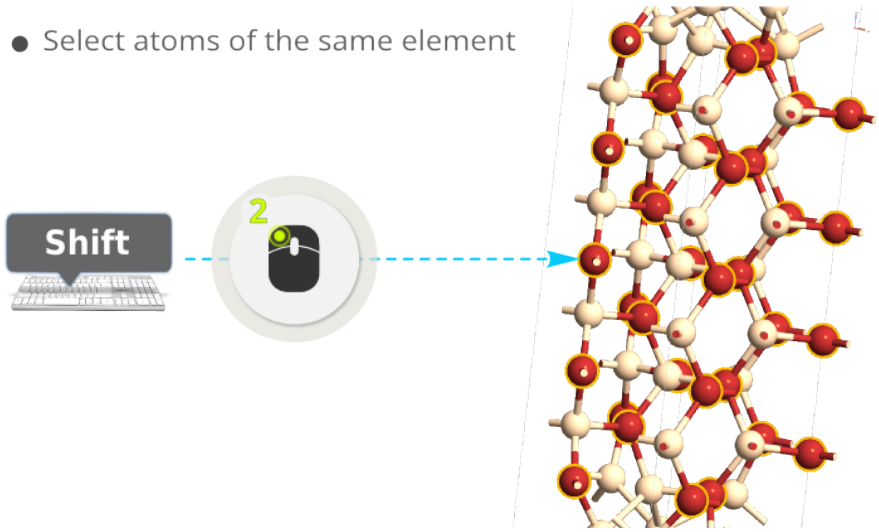
The selected objects/atoms in are indicated by a yellow halo.

•The **Group Selection Tools** allow you to choose a group of atoms using a square, circular or arbitrarily shaped lasso.

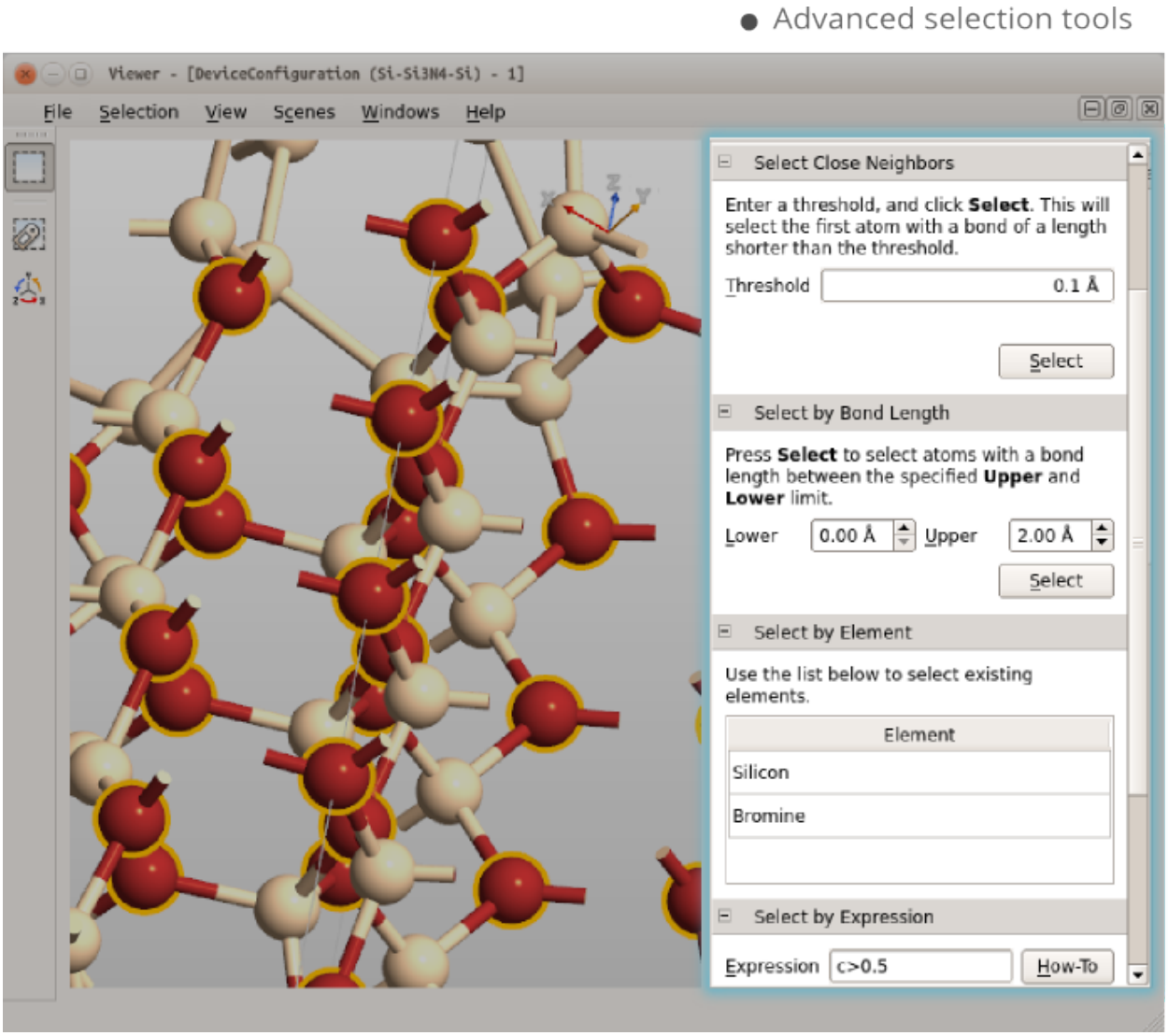


•You can of course use these tools combined with **(hold) Ctrl** to select several groups.

•You can also select all atoms of the same element by double-clicking one of those atoms while holding down the **Shift** key:



More advanced selection tools are also available. In the **Builder** these are located in the **Selection Tools** plugin, while in the **Viewer** they are located as individual **Select** plugins:



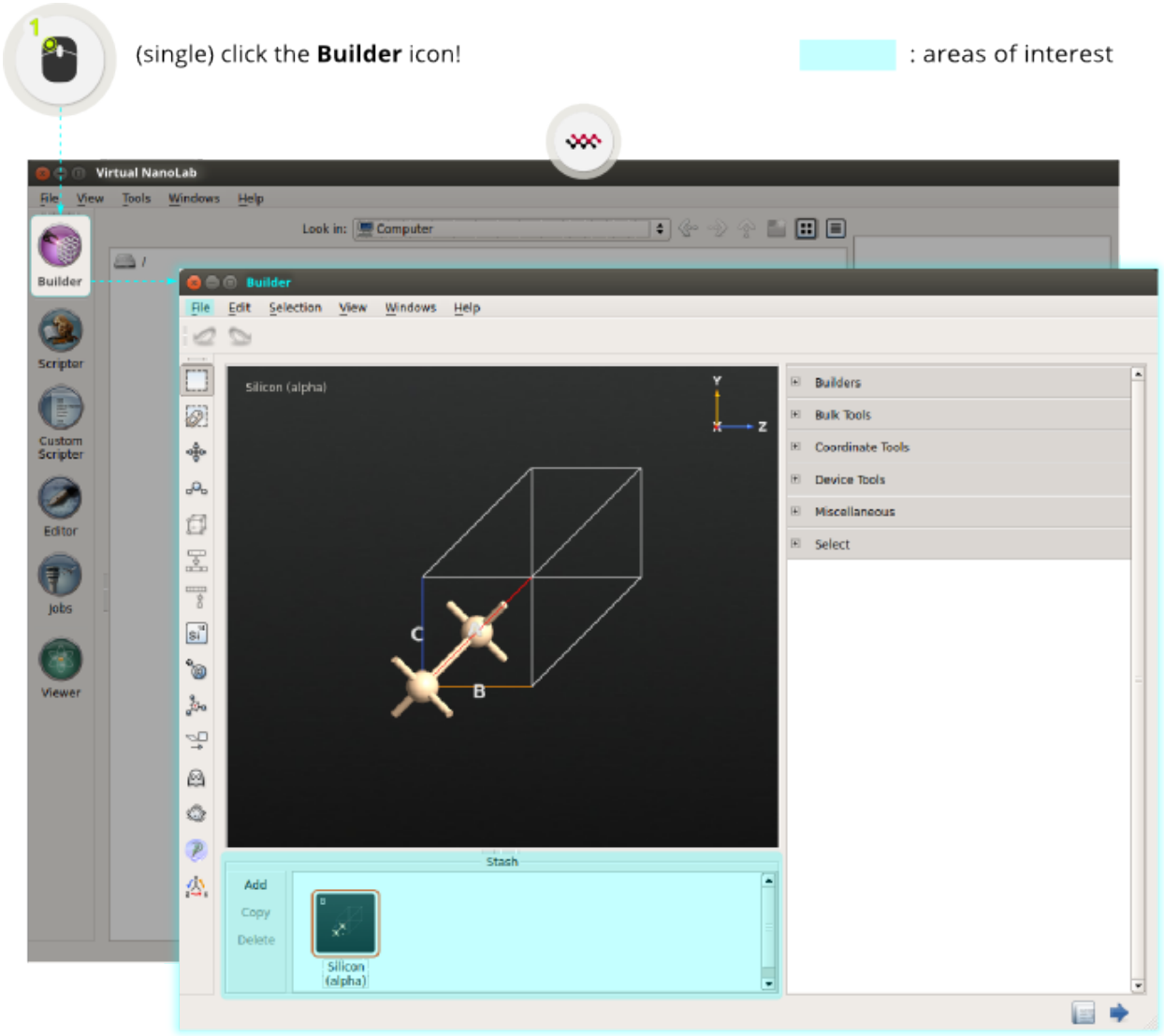
2.9 Importing and exporting files

2.9.1 Import/Export files in QuantumATK

In this tutorial you will learn how to import and/or export files in the QuantumATK **Builder**. The list of supported file types/formats is extensive, including file types `.py`, `.hdf5`, `.nc`, `.xyz`, `.car`, `.in`, `.cell`, `.geon`, `.cif`, `POSCAR`, `CONTCAR`, `.traj`, `.fdf`.

•Open **QuantumATK** and click the **Builder** icon at the top left column of the vertical menu.

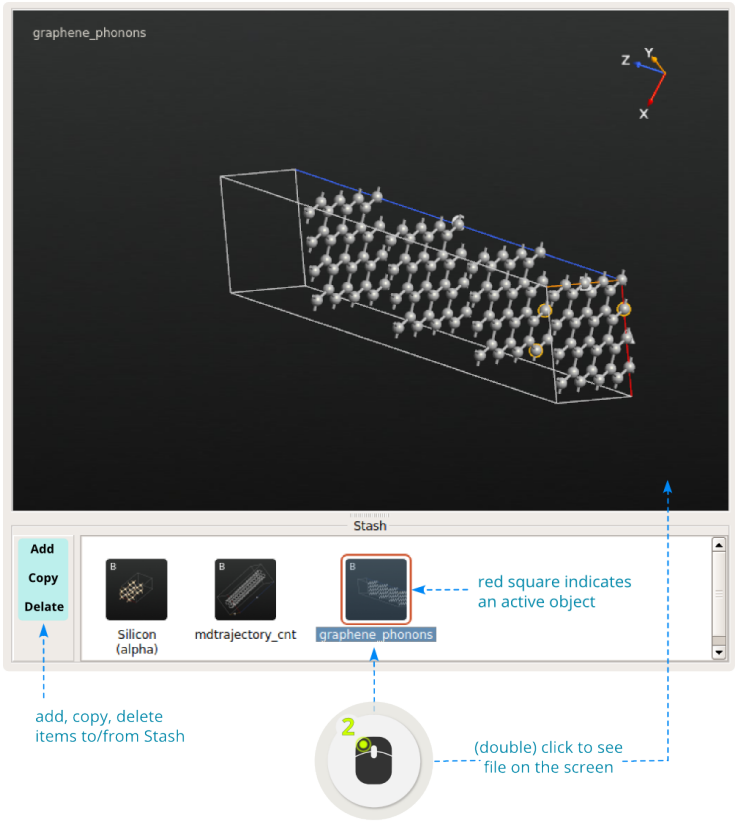
•The **Builder** window appears:

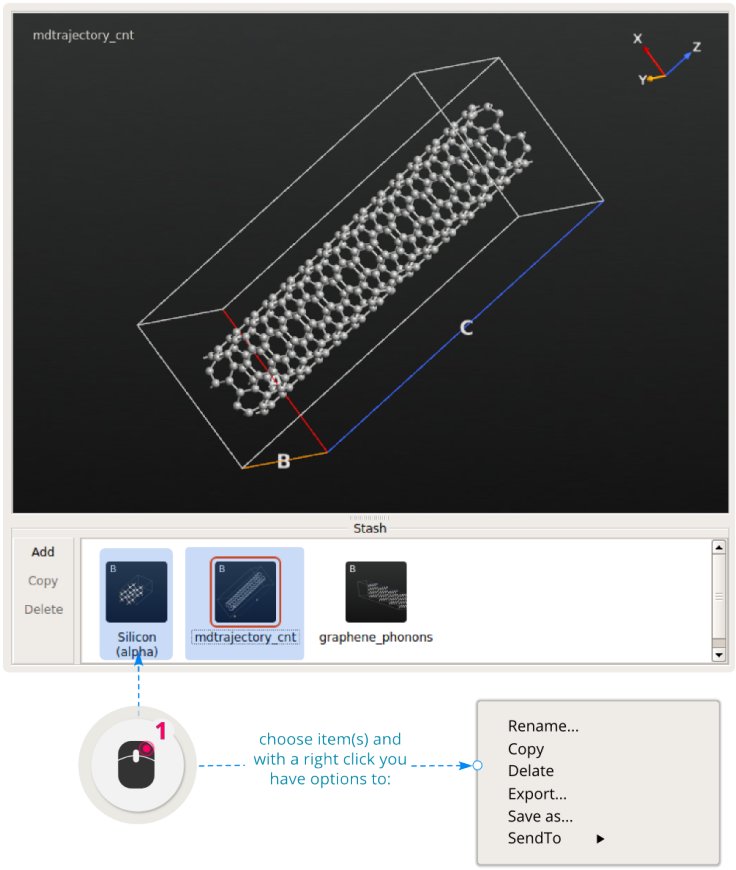


Stash Items

Attention: The **Stash** is like a small desktop for all your structures! You can delete, add, and copy structures. Double-click any of the structures in your **Stash** to visualize it.

The **Stash** functionality is simple and intuitive. The illustration below highlights a few main options and functionlity.

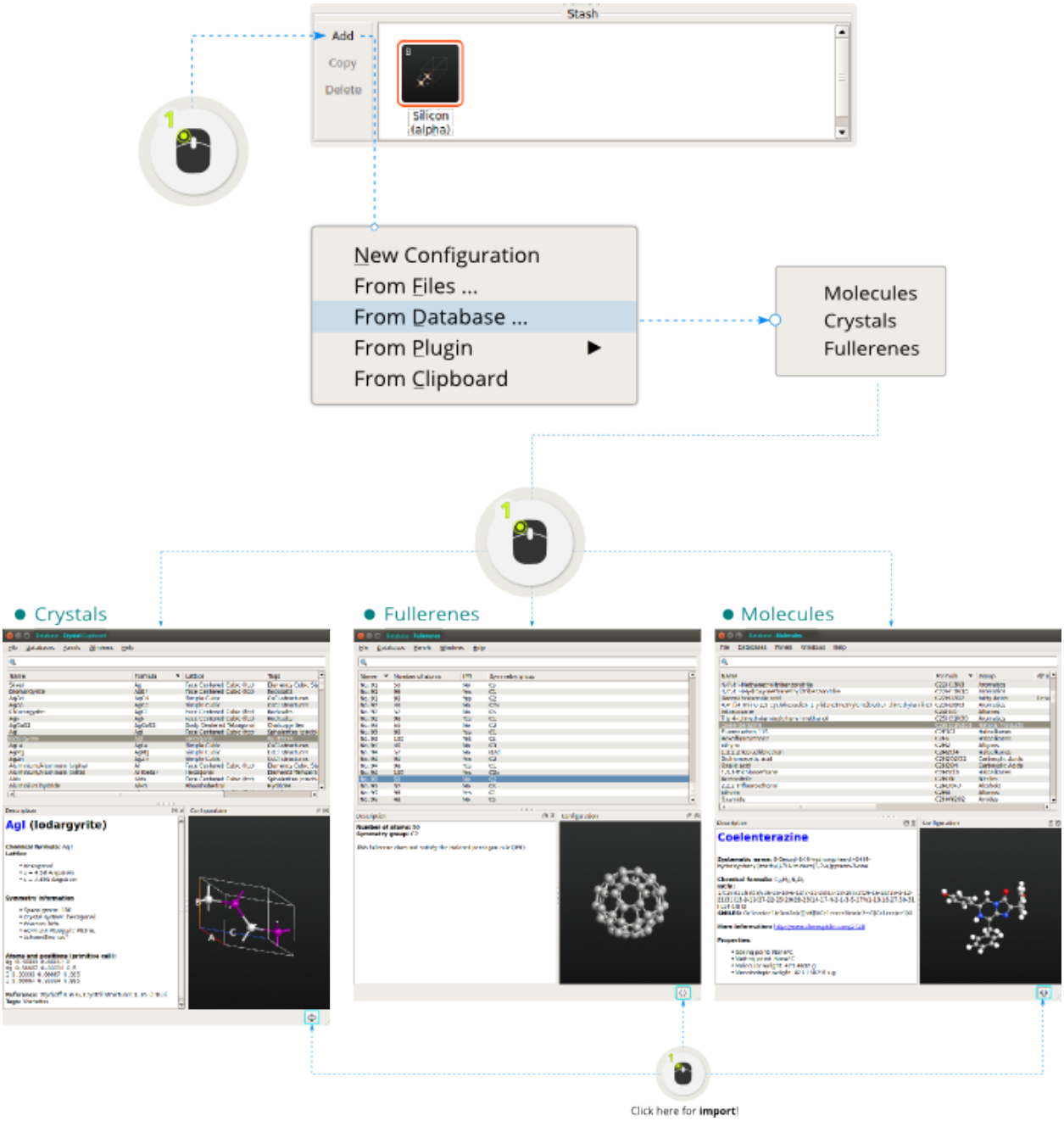

graphene_phonons
Z
Y
X
Stash
Add
Copy
Delate
Silicon (alpha)
mdtrajectory_cnt
graphene_phonons
red square indicates an active object
add, copy, delete items to/from Stash
2
(double) click to see file on the screen


mdtrajectory_cnt
X
Z
Y
C
B
Stash
Add
Copy
Delate
Silicon (alpha)
mdtrajectory_cnt
graphene_phonons
1
choose item(s) and with a right click you have options to:
Rename...
Copy
Delate
Export...
Save as...
SendTo

Add from Database

One of the easiest ways to set up structures is by using the **QuantumATK database**.

•Simply use *Add* → *From Database* to add a known structure to the **Stash**:



The available structures are grouped into three different categories: **Crystals**, **Molecules**, **Fullerenes**.

•You can search for structures and choose them from a list of matches.

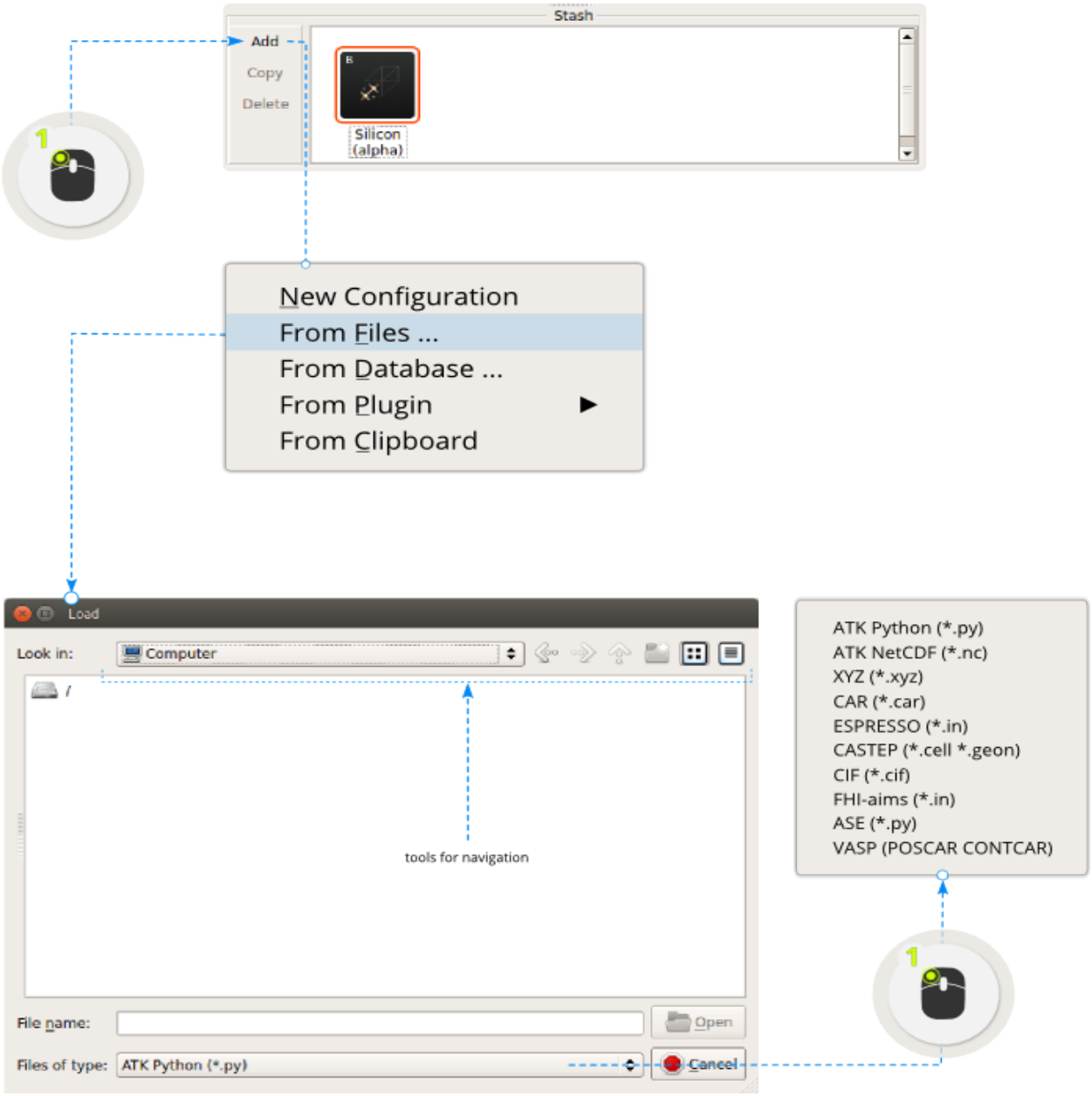
•Use the button or double click the name of the structure to add it to the **Stash**.

Add from Files

If you have files from previous **QuantumATK** projects, or from a database or some other software, it is easy to load them in **QuantumATK**:

- Use *Add* → *From Files* and load structures from the relevant files.

Remember to choose also the file type or extension of your file (see picture below).



Add from Plugins

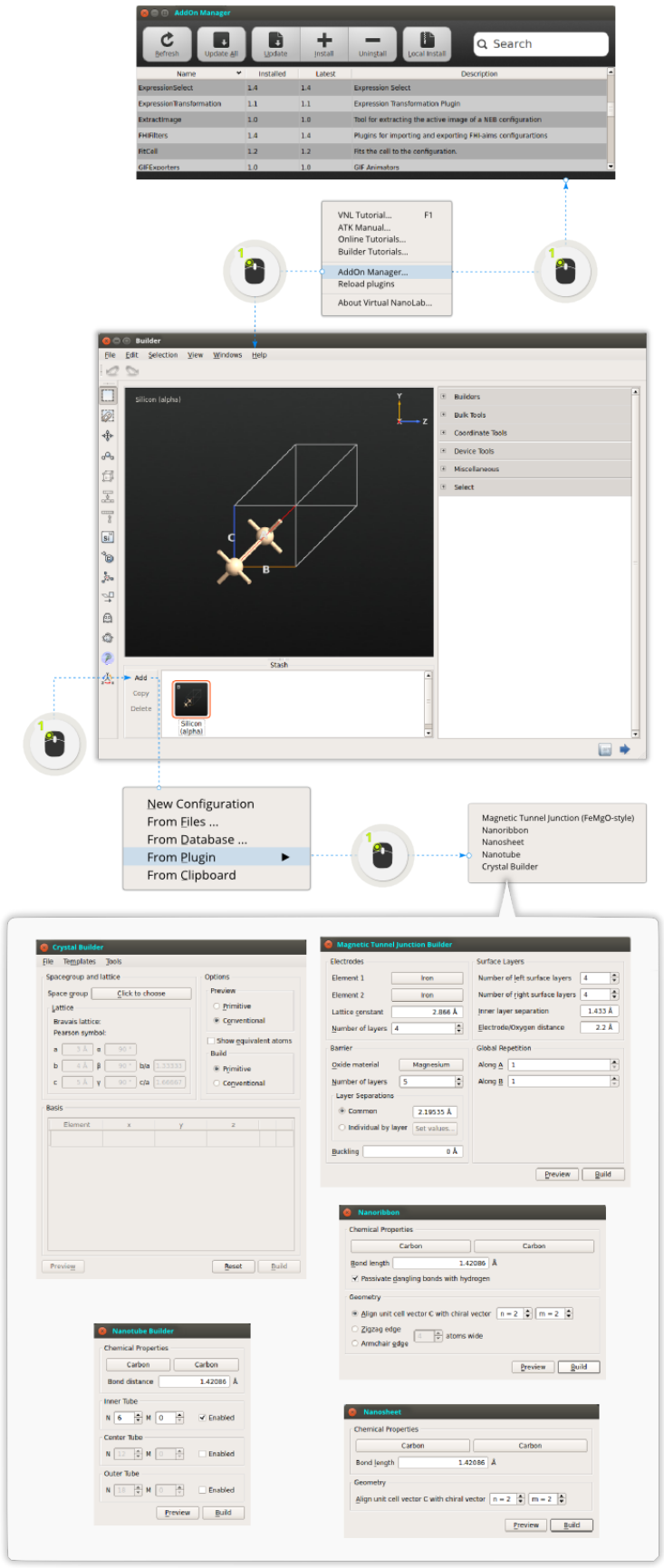
The **QuantumATK Builder** provides a range of convenient plugins for creating structures.

•Use *Add* → *From Plugin* to find a list of helpful plugins.

The list of plugins is extendable:

•Use *Help* → *AddOn Manager* to find new plugins for **QuantumATK**.

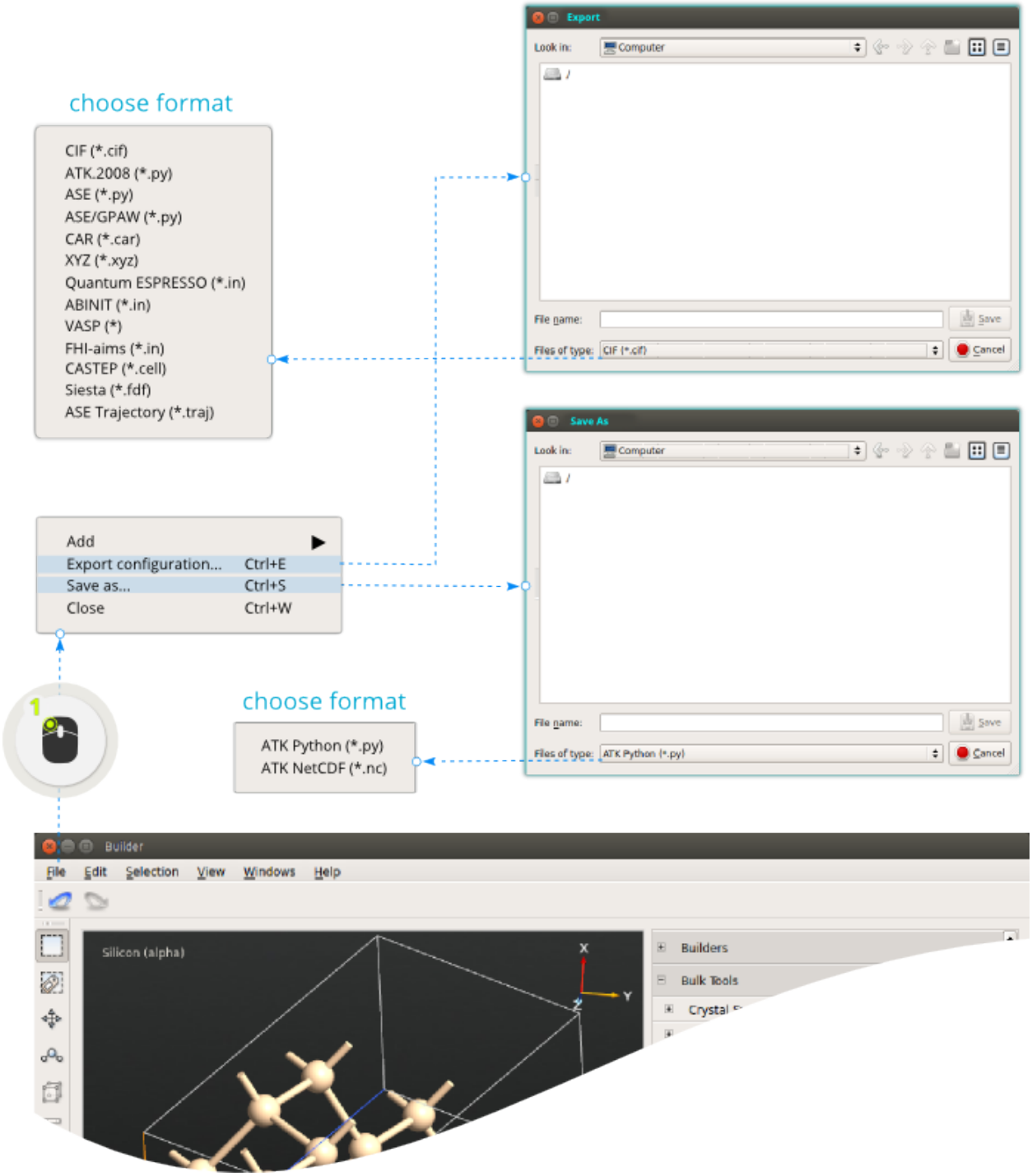
Note: The plugins database keeps expanding, and you can even write your own (Python) plugins, and add them to the existing ones.


AddOn Manager
Refresh
Update All
Update
Install
Uninstall
Local Install
Search
Name
Installed
Latest
Description
ExpressionSelect
ExpressionTransformation
ExtractImage
FHIRIters
FitCell
GIFExporters
Expression Select
Expression Transformation Plugin
Tool for extracting the active image of a NEB configuration
Plugins for importing and exporting FHI-aims configurations
Fits the cell to the configuration.
GIF Animators
VNL Tutorial... F1
ATK Manual...
Online Tutorials...
Builder Tutorials...
AddOn Manager...
Reload plugins
About Virtual NanoLab...
Builder
File Edit Selection View Windows Help
Silicon (alpha)
Builders
Bulk Tools
Coordinate Tools
Device Tools
Miscellaneous
Select
Stash
Add
Copy
Delete
New Configuration
From Files ...
From Database ...
From Plugin
From Clipboard
Magnetic Tunnel Junction (FeMgO-style)
Nanoribbon
Nanosheet
Nanotube
Crystal Builder
Crystal Builder
Magnetic Tunnel Junction Builder
Nanoribbon
Nanotube Builder
Nanosheet
Preview
Build
Reset

Save and Export Structures

•Save a structure by right-clicking on it and choosing *File*.

The two most important options are *Save as* and *Export configuration*. Many export formats are available, apart from the **QuantumATK** native `.nc` format:



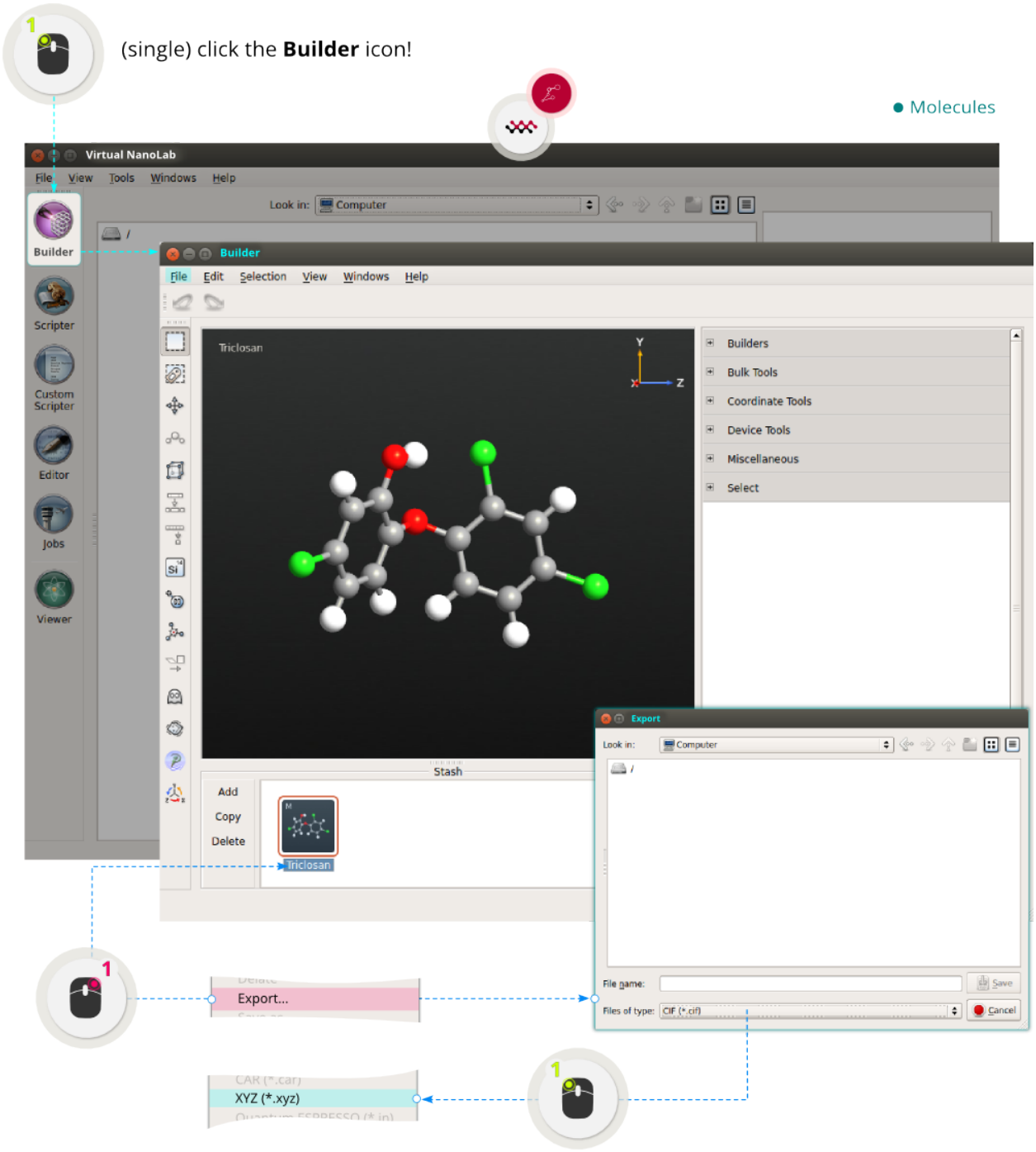
2.9.2 Export XYZ, CIF, CAR, VASP Files in QuantumATK

QuantumATK: File handling Platforms: Links: VASP, VASPfiles, *Import Files* **File types: XYZ, CIF, CAR, VASP** *In this tutorial, you will learn how to build a configuration in QuantumATK and export it for use in another program.*

Built-in export filters

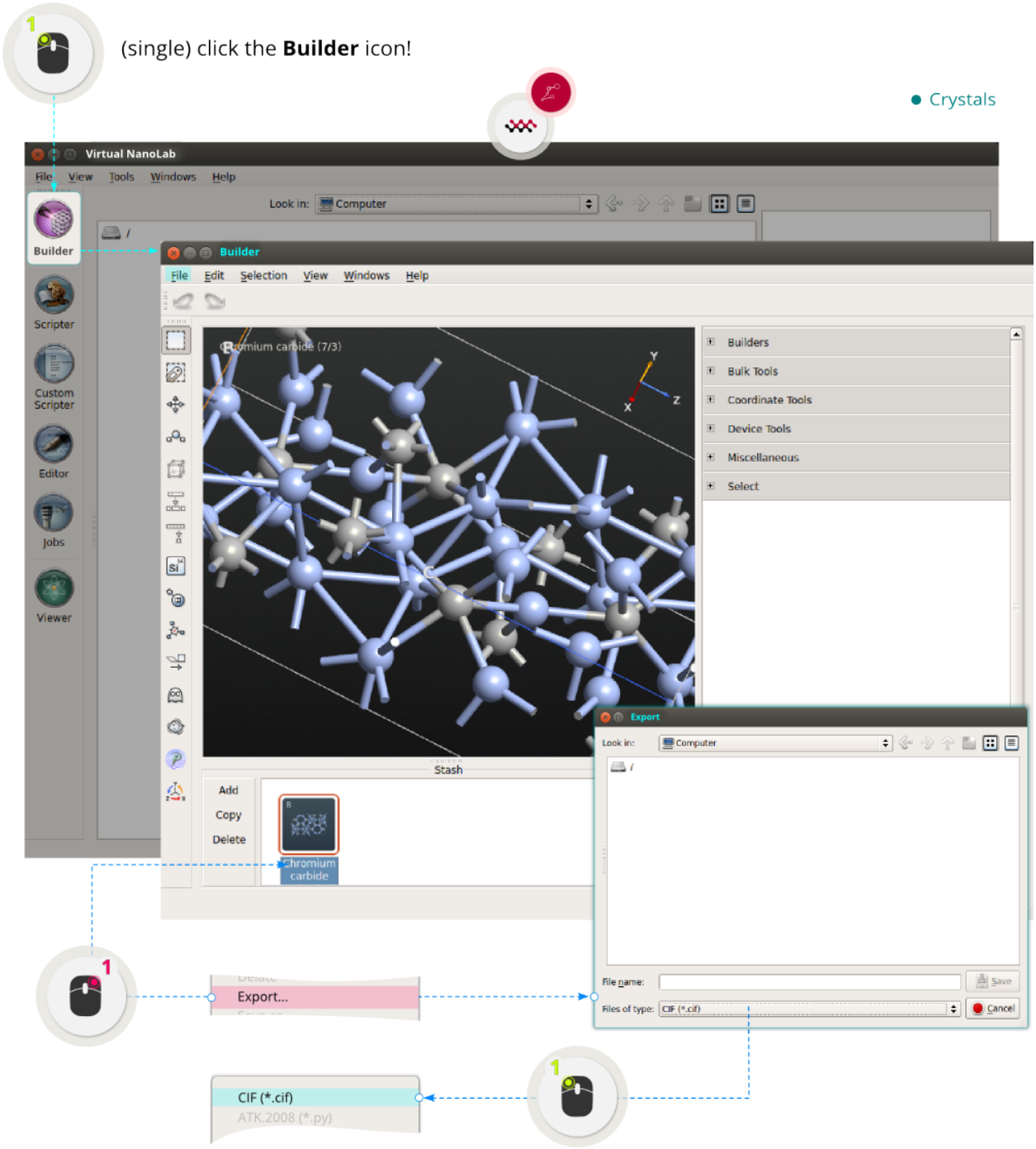
Molecular Structures

For molecular structures, you can directly export the geometry in an **XYZ** file, via the menu *File→Export configuration* in the **Builder**. Choose the XYZ format under *Files of type*, enter the file name, and save the file.


(single) click the **Builder** icon!
Molecules
Virtual NanoLab
File View Tools Windows Help
Look in: Computer
Builder
Scripter
Custom Scripter
Editor
Jobs
Viewer
Builder
File Edit Selection View Windows Help
Triclosan
Builders
Bulk Tools
Coordinate Tools
Device Tools
Miscellaneous
Select
Stash
Add
Copy
Delete
Triclosan
Export
Look in: Computer
File name:
Save
Files of type: CIF (*.cif)
Cancel
Export...
CAR (*.car)
XYZ (*.xyz)

Crystal Structures

A similar approach is possible for exporting periodic structures (crystals, nanotubes, etc) to **CIF** or **CAR** files.



Exporting from scripts

The function **nlprint** is also a convenient tool for exporting structures:

```
nlprint(configuration)
```

Include this line in a script (referring to the correct variable for the structure), and pipe the output to a file when you run it:

```
atkpython script.py > file.xyz
```

There will be a few lines in the file file.xyz that you need to remove, but the format of the coordinate list is proper XYZ. If the structure is a two-probe or bulk configuration, there will be some additional information about the unit cell too.

Note: The **XYZ** file format is a chemical file format. There is no formal standard and several variations exist, but a typical XYZ format specifies the molecule geometry by giving the number of atoms with Cartesian coordinates that will be read on the first line, a comment on the second, and the lines of atomic coordinates in the following lines. The file format is used in computational chemistry programs for importing and exporting geometries. The units are generally in Angstroms. Some variations include using atomic numbers instead of atomic symbols, or skipping the comment line. Files using the XYZ format conventionally have the **.xyz** extension.

Note: **CAR** stands for **Component Application Resource**. The CAR files contain information about the assets that are available for a software application being run on the Brew Mobile Platform. This platform is used to operate applications on a variety of mobile phones. The CAR files are compiled into a binary BAR file. This file is automatically loaded when the application is run.

The **.car** file extension is also used for AtHome Assistant files, Biosym chemical modeler input files, NeoBook Cartoon image format files, CAR Archive compressed archive files, CardMaker card files, Carnivores Ice Age resource files, Design-Your-CD data files and Railroad Tycoon 3 car properties files.

Note: **Crystallographic Information File** (**CIF**) is a standard text file format for representing crystallographic information, promulgated by the International Union of Crystallography (IUCr). CIF was developed by the IUCr Working Party on Crystallographic Information in an effort sponsored by the IUCr Commission on Crystallographic Data and the IUCr Commission on Journals. Full specifications for the format are available at the IUCr website. Many computer programs for molecular viewing are compatible with this format, including Jmol.

Note: **VASP** is a fileformat/extension used by VASP. VASP software uses a relatively large number of input and output files:

INCAR	CONTCAR	DOSCAR	XDATCAR	PCDAT	EIGENVAL
STOPCAR	POSCAR	PROOUT	TMPCAR	OSZICAR	WAVECAR
stout	IBZKPT	ELFCAR	CHG	PROCAR	CHGCAR
POTCAR	KPOINTS	LOCPOT	EXHCAR	DOSCAR	

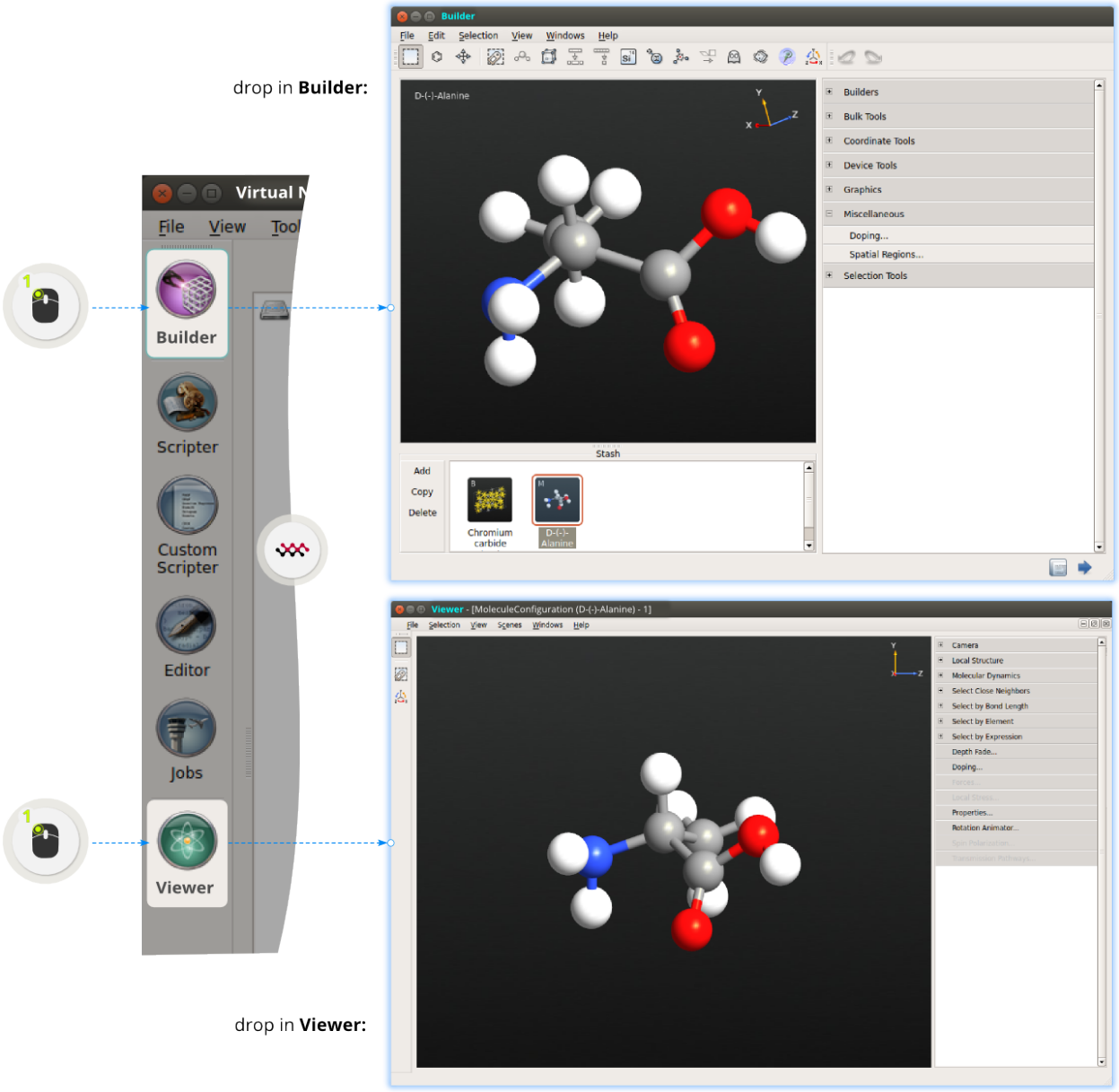
You can find more details about their content on the VASP website.

2.9.3 Import XYZ, CIF, CAR, VASP Files in QuantumATK

QuantumATK: File handling Platforms: Links: *Export Files* **File types: XYZ, CIF, CAR, VASP(POSCAR)** *In this tutorial, you will learn how to import configuration files from other programs.*

Drag and Drop

In the simplest case, all you actually have to do is to drop the file itself on one of the instruments in **QuantumATK**, typically the **Builder** (for editing the structure), or the **Viewer** (for just visualizing the structure).



Importing structures in a script

It is also possible to import and use external files directly in a script:

```
molecule = nlread('myfile.xyz')[0]
```

Note that nlread returns a list, normally only with one element, hence the [0]. The file formats are recognized by the extensions, so the same principle would apply to CIF, CAR, or POSCAR files alike. Here is another way to use the XYZ format, if you prefer to just copy/paste the coordinate list

```
h2o = MoleculeConfiguration(xyz_format=
    """3
    water
    O   0.000 0.000 0.0
    H   0.757 0.586 0.0
    H  -0.757 0.586 0.0""")
```

Note that the format must be that of a real XYZ file, with the two first lines specifying the number of atoms, and the comment/title. For more details, see the manual entry for the `MolecularConfiguration` class.

Note: The **XYZ** file format is a chemical file format. There is no formal standard and several variations exist, but a typical XYZ format specifies the molecule geometry by giving the number of atoms with Cartesian coordinates that will be read on the first line, a comment on the second, and the lines of atomic coordinates in the following lines. The file format is used in computational chemistry programs for importing and exporting geometries. The units are generally in Angstroms. Some variations include using atomic numbers instead of atomic symbols, or skipping the comment line. Files using the XYZ format conventionally have the **.xyz** extension.

Note: **CAR** stands for **Component Application Resource**. The CAR files contain information about the assets that are available for a software application being run on the Brew Mobile Platform. This platform is used to operate applications on a variety of mobile phones. The CAR files are compiled into a binary BAR file. This file is automatically loaded when the application is run.

The **.car** file extension is also used for AtHome Assistant files, Biosym chemical modeler input files, NeoBook Cartoon image format files, CAR Archive compressed archive files, CardMaker card files, Carnivores Ice Age resource files, Design-Your-CD data files and Railroad Tycoon 3 car properties files.

Note: **Crystallographic Information File** (**CIF**) is a standard text file format for representing crystallographic information, promulgated by the International Union of Crystallography (IUCr). CIF was developed by the IUCr Working Party on Crystallographic Information in an effort sponsored by the IUCr Commission on Crystallographic Data and the IUCr Commission on Journals. Full specifications for the format are available at the IUCr website. Many computer programs for molecular viewing are compatible with this format, including Jmol.

Note: **VASP** is a fileformat/extension used by VASP. VASP software uses a relatively large number of input and output files:

INCAR	CONTCAR	DOSCAR	XDATCAR	PCDAT	EIGENVAL
STOPCAR	POSCAR	PROOUT	TMPCAR	OSZICAR	WAVECAR
stout	IBZKPT	ELFCAR	CHG	PROCAR	CHGCAR
POTCAR	KPOINTS	LOCPOT	EXHCAR	DOSCAR	

You can find more details about their content at the VASP website.

2.10 Molecular builder

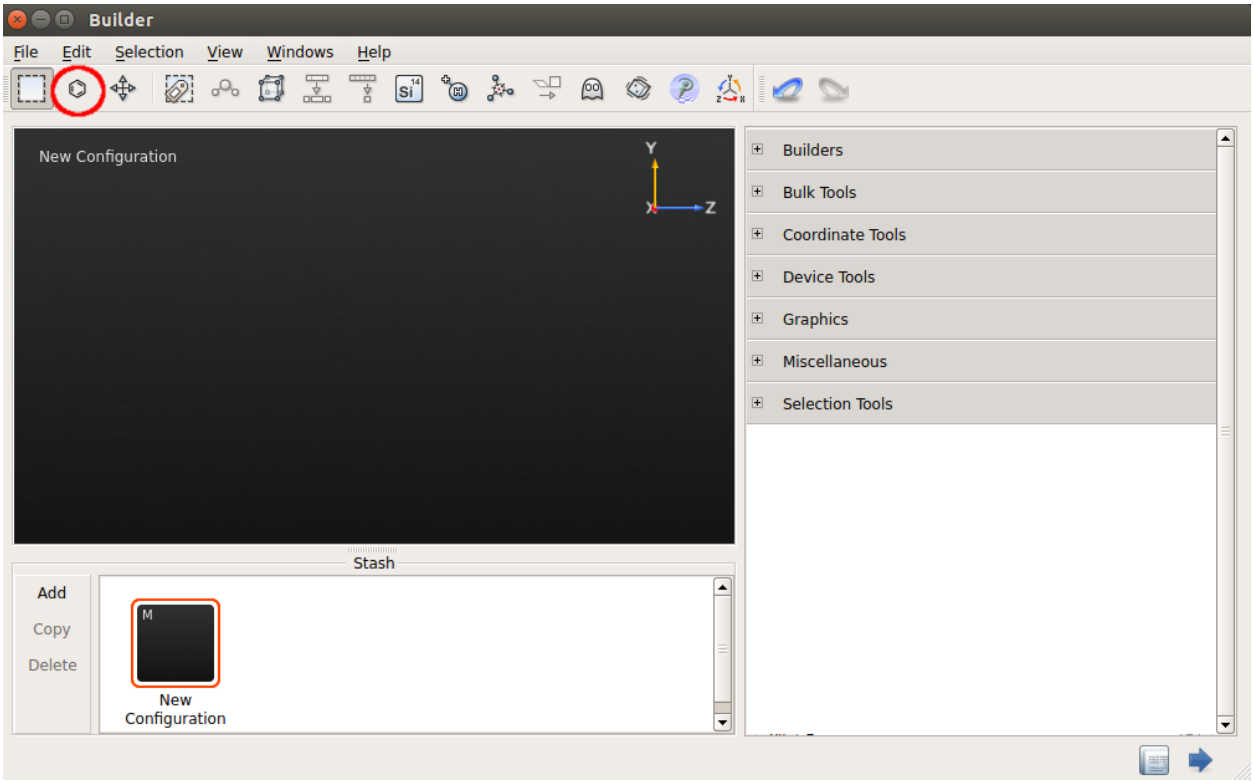
Platforms: *In this tutorial, you will learn how to use the Molecular Builder tool, which makes it easy to build complex molecular structures by combining atoms and molecular fragments. As examples of use, you will build the ethanol and caffeine molecules.*

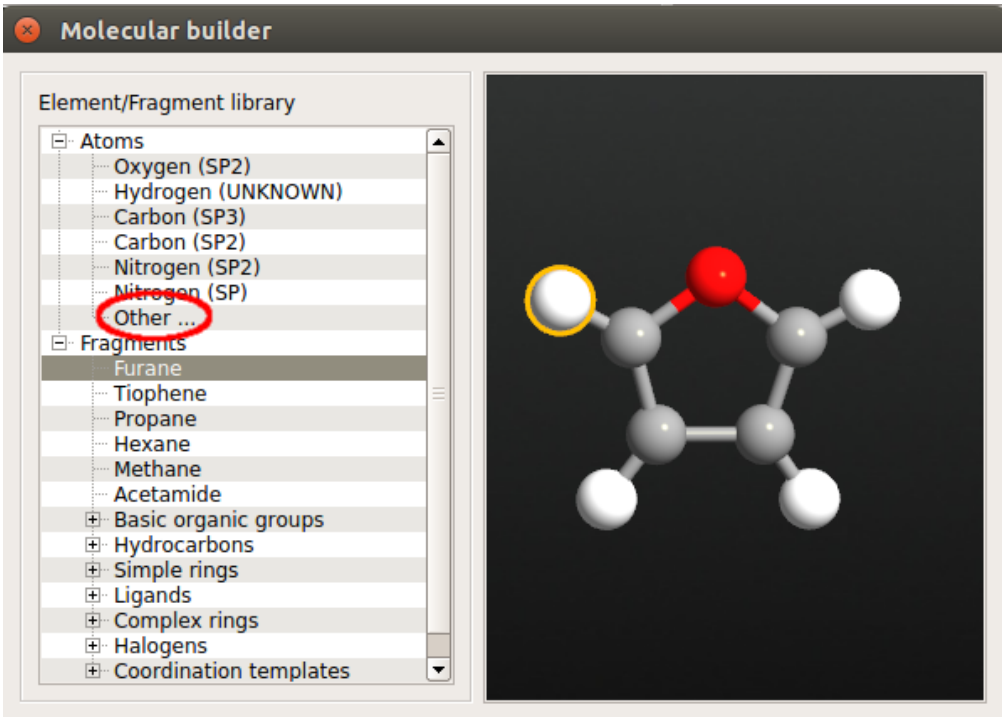
Note: The **Molecular builder** tool is selected using the icon in the top plugins bar. You need to highlight a Stash configuration before activating the tool.

2.10.1 Ethanol molecule

Create a new empty project called molecular_builder and open the Builder to get started:

1. Use *Add* → *New Configuration* to create a Stash item, and remove the hydrogen atom to create an empty configuartion.
2. Activate the **Molecular builder** by clicking . It will look like illustrated below.

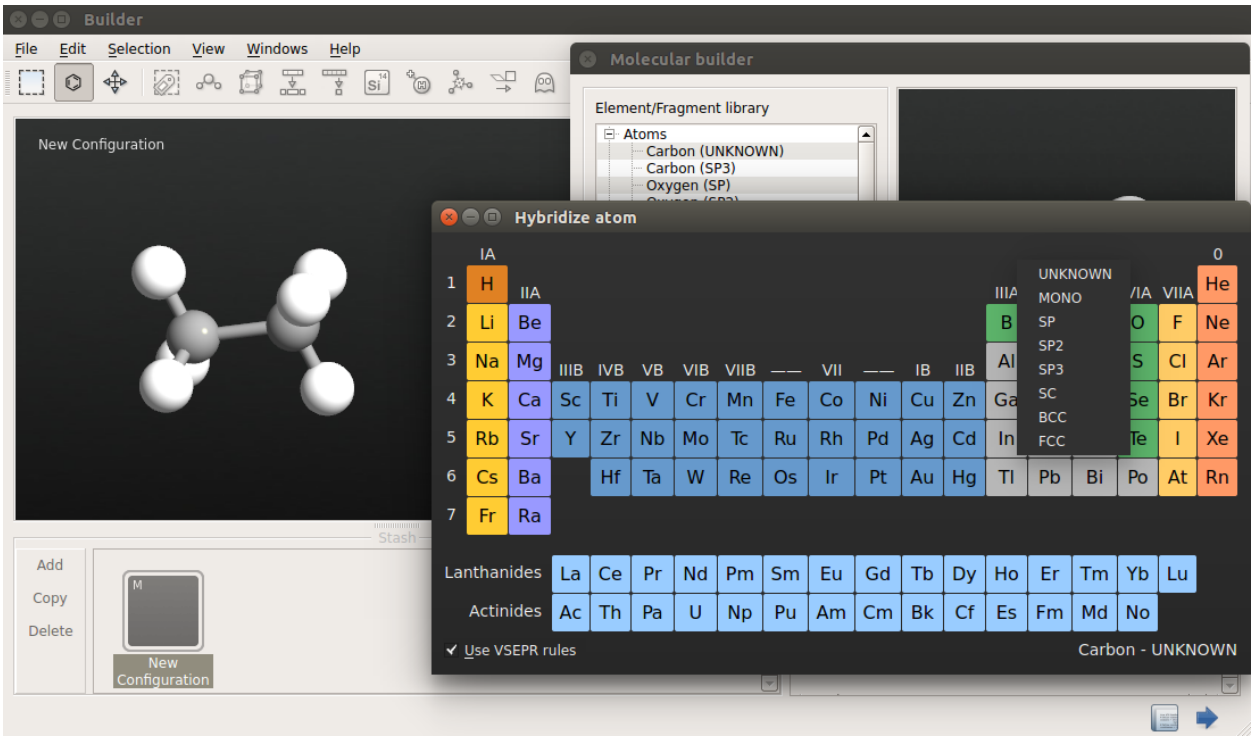




Attention: What you see is a library of elements and fragments that can be added to the current Stash configuration. The most recently used ones appear at the top of the list.

Click **Other** and a periodic table will open up. You can now select the atomic elements you need to build your molecular structure. An easy way to build ethanol is to first build ethane, and then add the hydroxyl group:

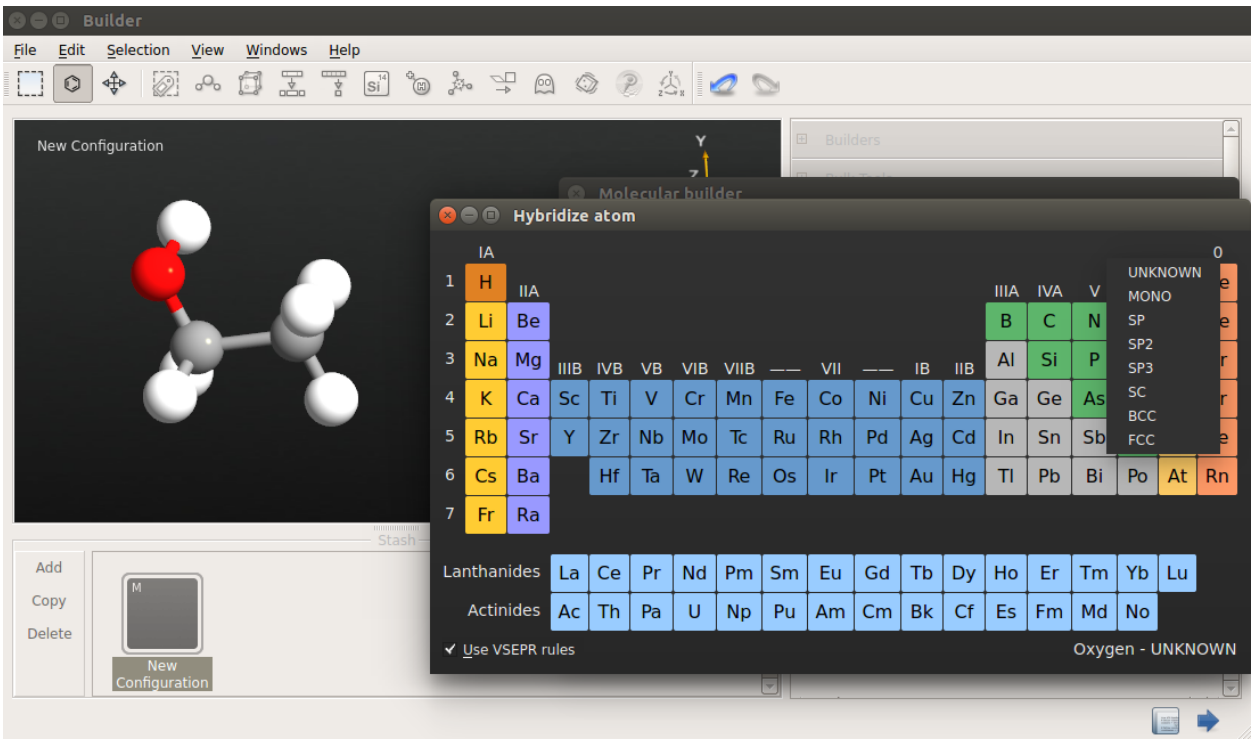
1. Click the C atom and select SP3 to select a C atom that should be sp3 bonded.
2. Click the 3D View to add the sp3 saturated carbon to the configuration. Note that this produces the methane molecule.
3. Then click on one of its hydrogen atoms to substitute that atom for another sp3 bonded carbon. You have then built ethane.



Next, add the hydroxyl group:

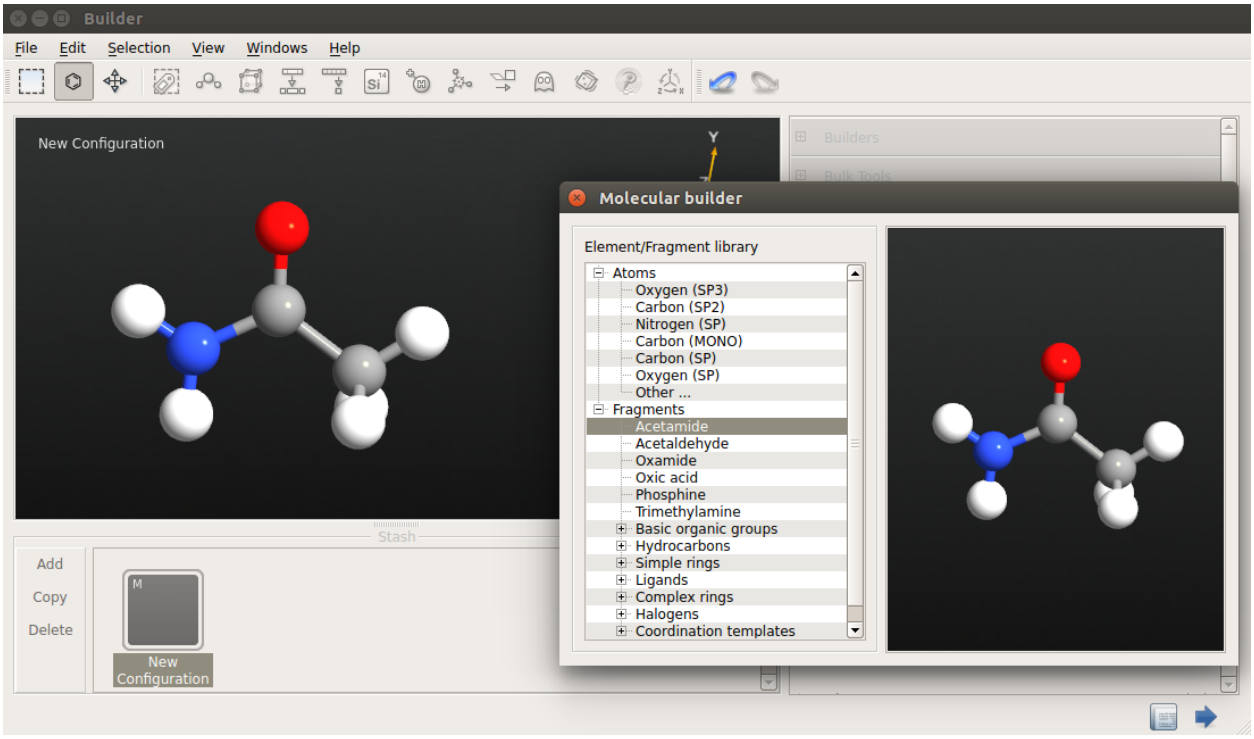
1. Select from the periodic table a sp3 O atom and make sure that the *Use VSEPR rules* option is selected. With this option we make sure that we are adding a OH group to the molecule (see more details about the *Use VSEPR rules* option below).

2. Click one of the hydrogen atoms in the 3D View to substitute it for the OH group.

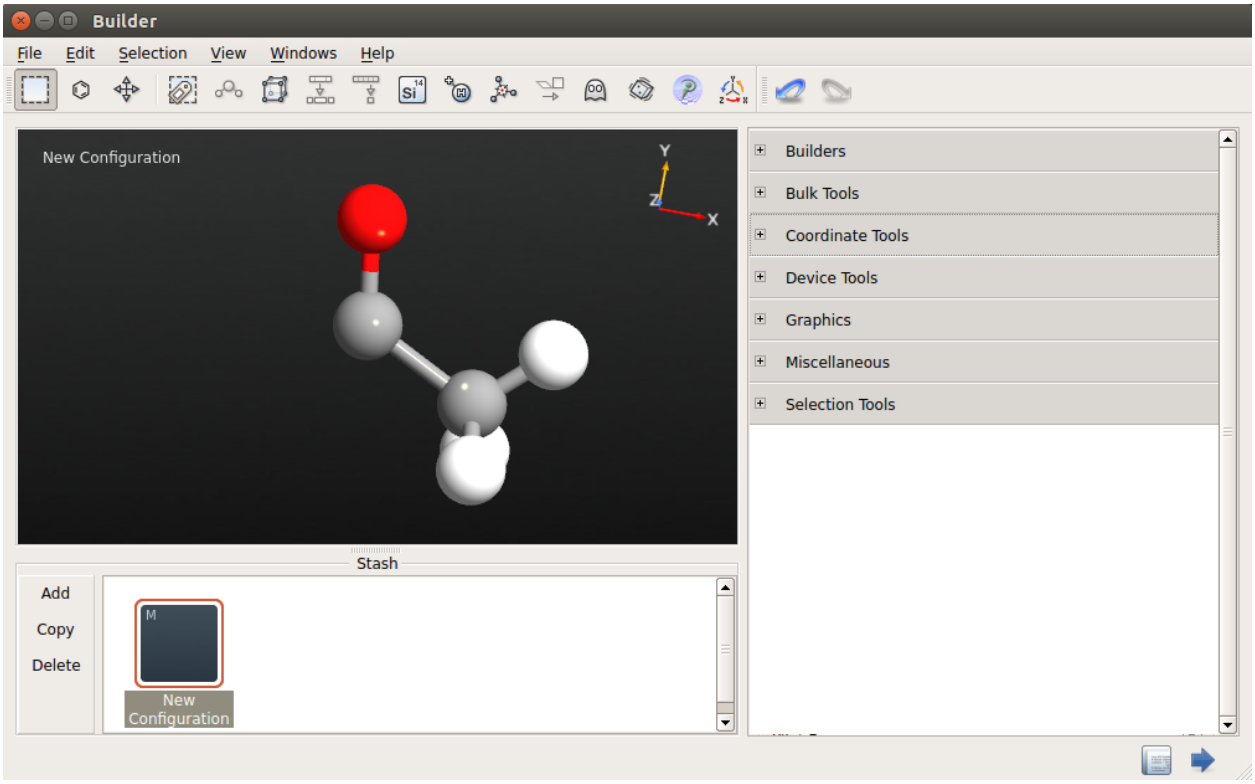


You have just built the ethanol molecule by combining atoms. You could also have built the same molecular structure by combining molecular fragments in a variety of ways. For instance, you could have added Methanol from the fragment library and then replaced one of its hydrogen atoms with the Methane fragment. Another easy alternative could be to select Acetamide fragment, remove its amine group and passivate the molecule to create sp3 O and C atoms:

1.Import Acetamide fragment from the fragment library.

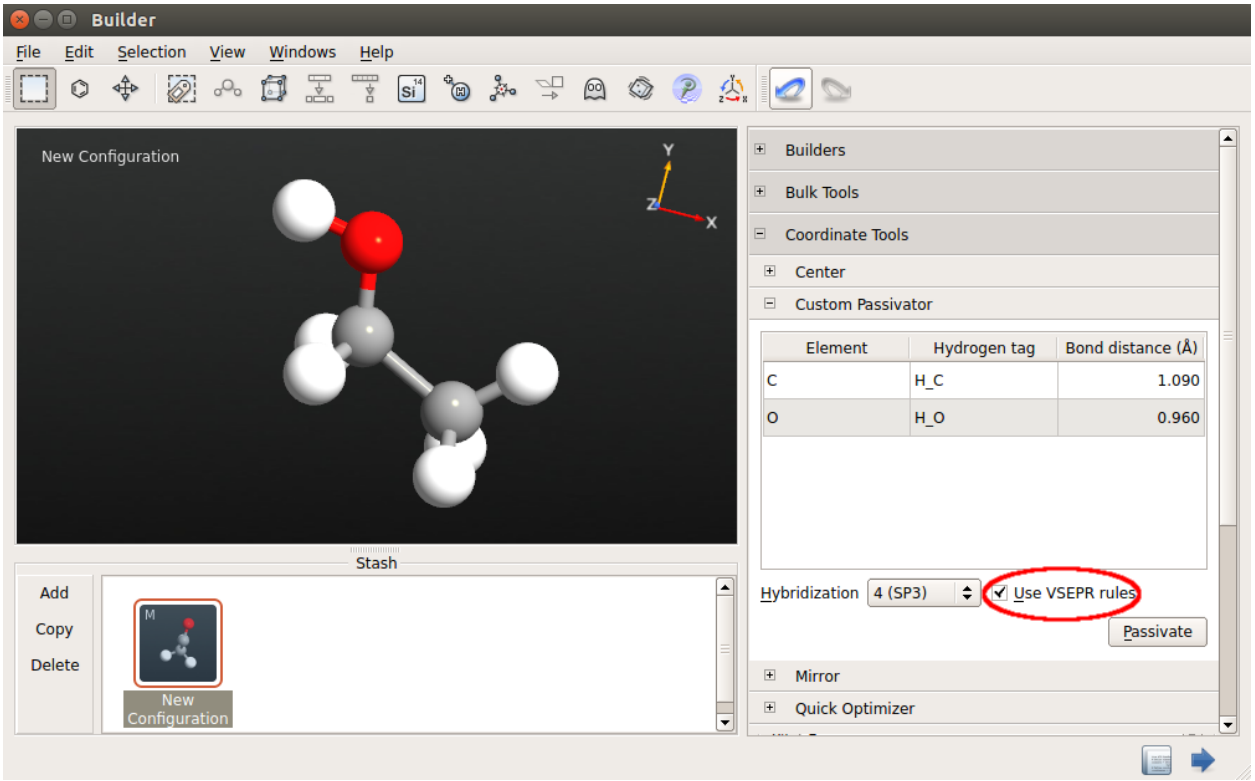


1.Select the amine moiety and remove it.



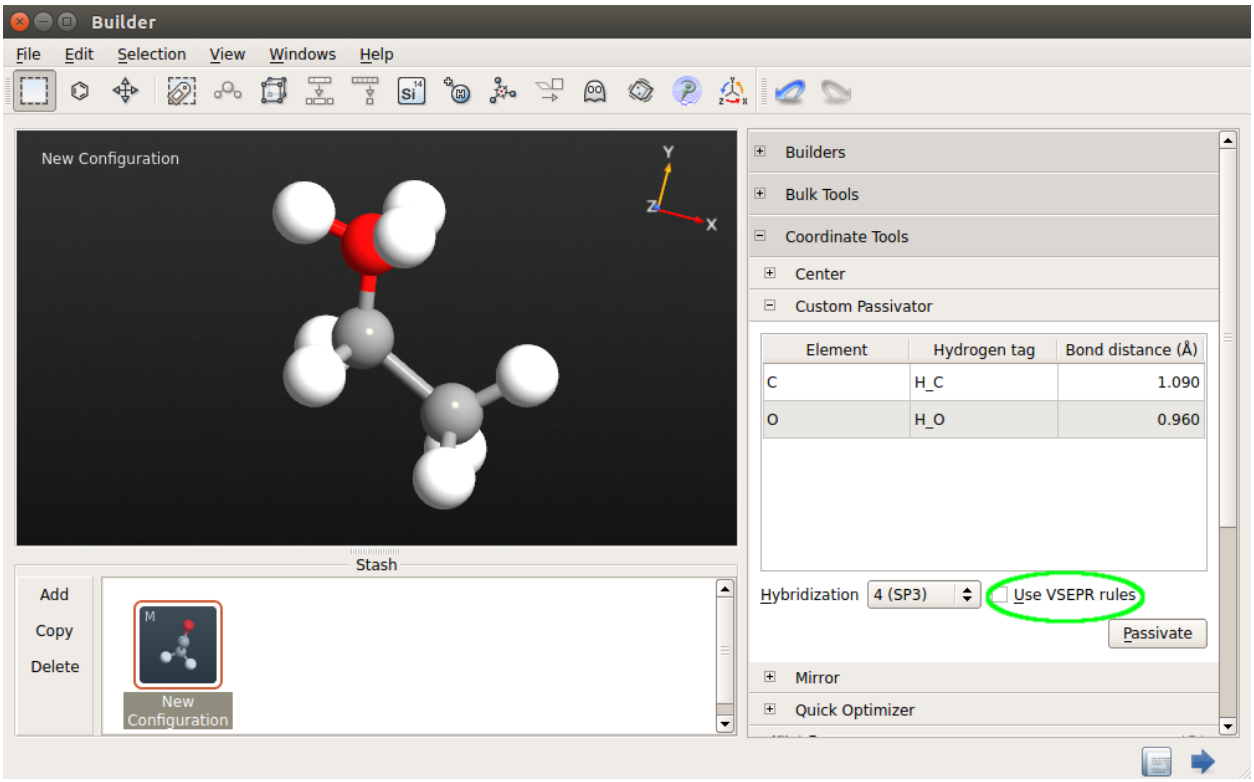
1.Go to *Coordinate Tools* → *Custom Passivator* and choose Hybridization 4(SP3).

2.Tick Use VSEPR rules to ensure that you passivate the structure following the Valence Shell Electron Pair Repulsion (VSEPR) rules, and click Passivate.



The VSEPR rule determines the bonding coordination of an atom considering the minimum repulsion between its valence electron pairs (both shared and unshared). The use of VSEPR rules ensures the passivation of the structure considering the bonding electrons and the lone pairs. Instead, if we passivate the molecule without following the VSEPR rules each atom of the molecule will be passivated with the maximum number of hydrogen atoms for the corresponding hybridisation, as the lone pairs will no be considered.

In the latter case, we get the structure shown below if we passivate the molecule without considering the VSEPR rules.



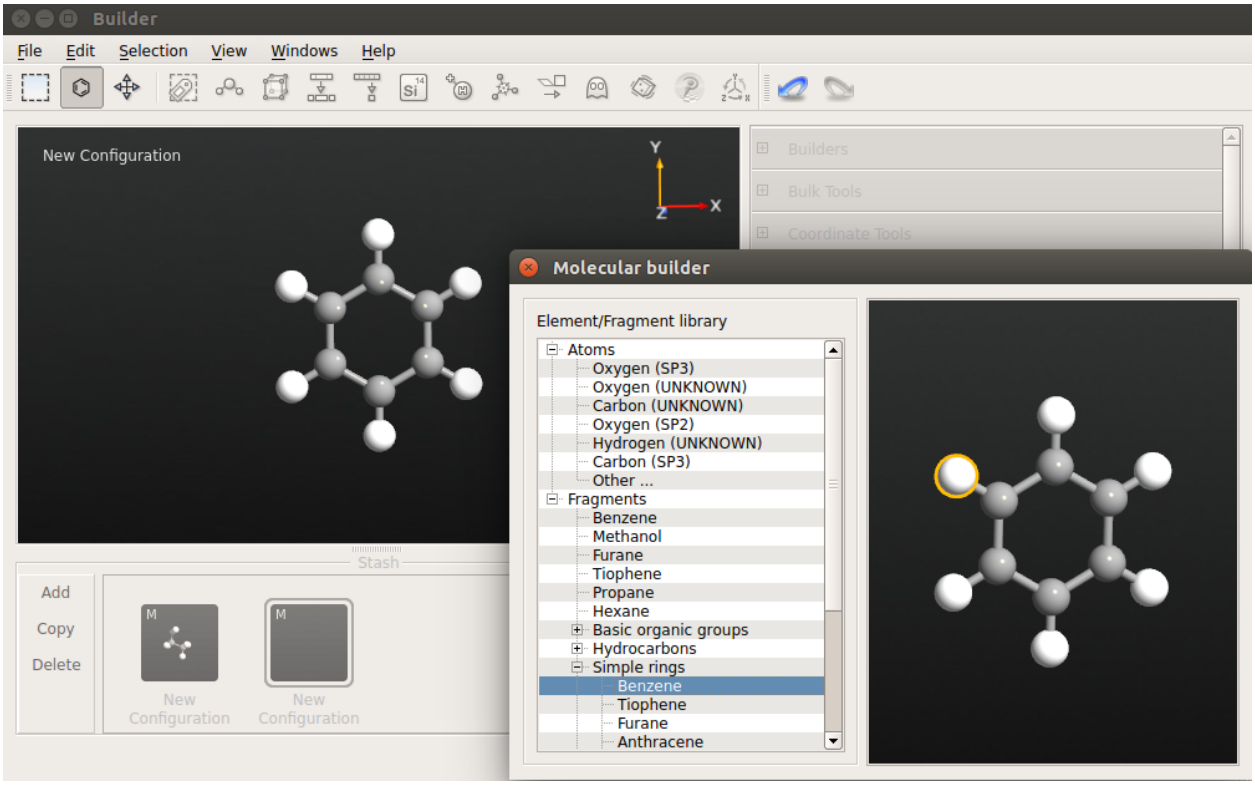
Tip: It is always a safe choice to passivate the structure considering the VSEPR rules when building neutral molecules.

Note: It is not possible to select any of the atoms in the 3D View while the Molecular builder window is active. You need to close this window first before the other Builder functionalities can be used.

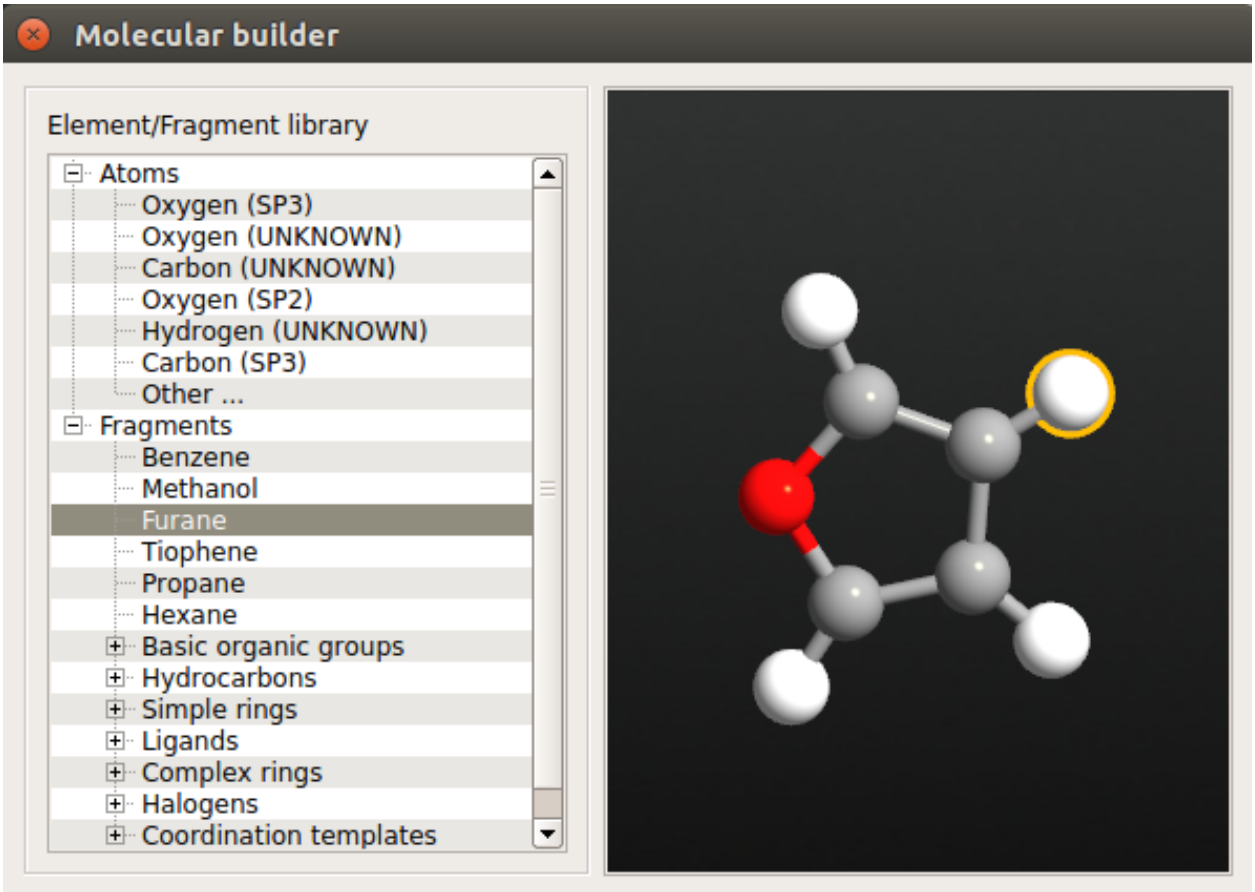
2.10.2 Caffeine molecule

It is quite easy to use the Molecular builder to build complex structures from simple fragments instead of from atoms. You will now build the caffeine molecule, also known as 1,3,7-Trimethylpurine-2,6-dione. The chemical formula is $C_8H_{10}N_4O_2$.

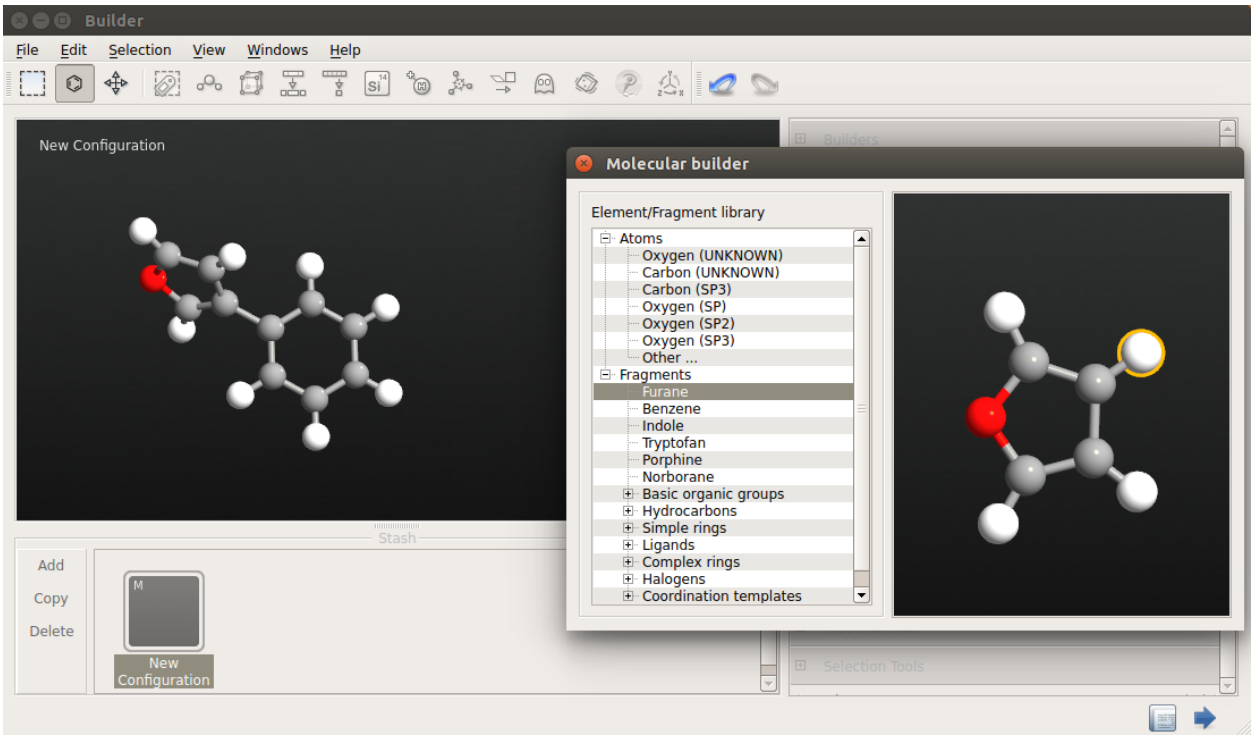
1. Add an empty configuration to the Stash and activate the Molecular builder .
2. Go to *Fragments* → *Simple rings* and select Benzene.
3. Click the 3D View to add benzene to the configuration.



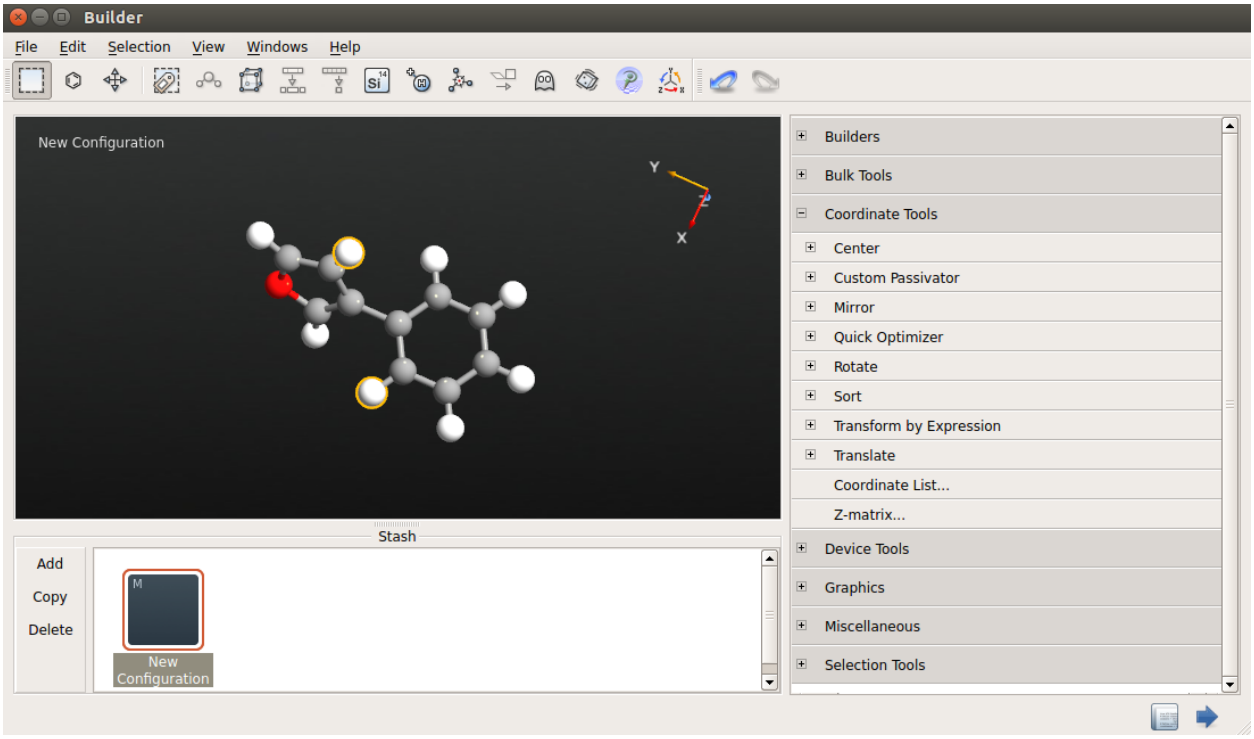
4. Select Furane from the Molecular Builder fragments library, and use mouse-click to **select** the hydrogen atom indicated below.



5. In the 3D View, add the furane fragment to the configuration by clicking one of the hydrogen atoms. Note that the furane is fused to the benzene ring exactly at the hydrogen that was selected in the Molecular builder.

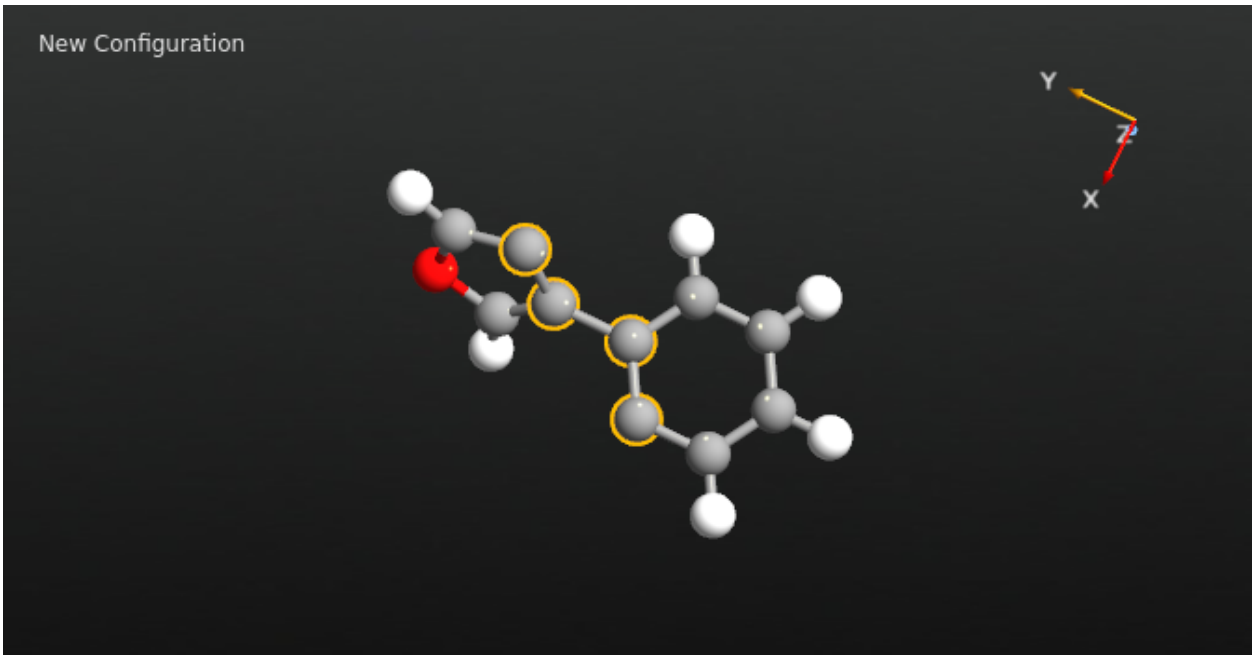


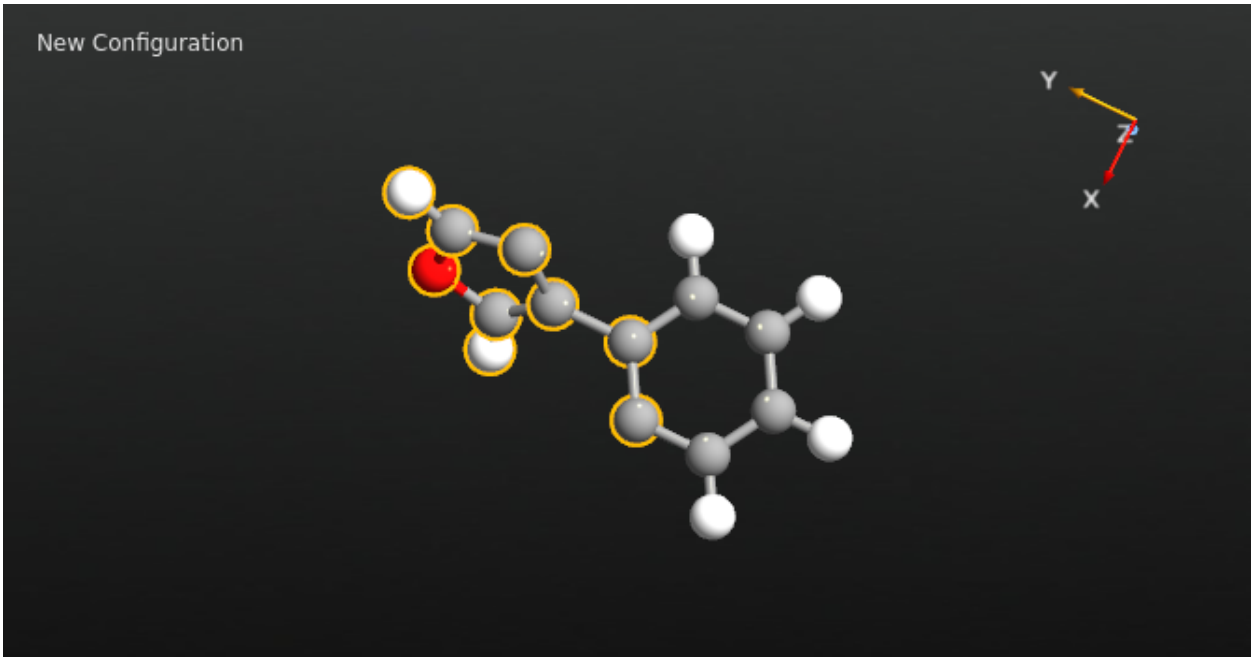
6.Select the hydrogen atoms that are highlighted below and remove them.



In order to make sure that both rings are in the same plane, you need to do a dihedral rotation using the **Z-matrix** plugin:

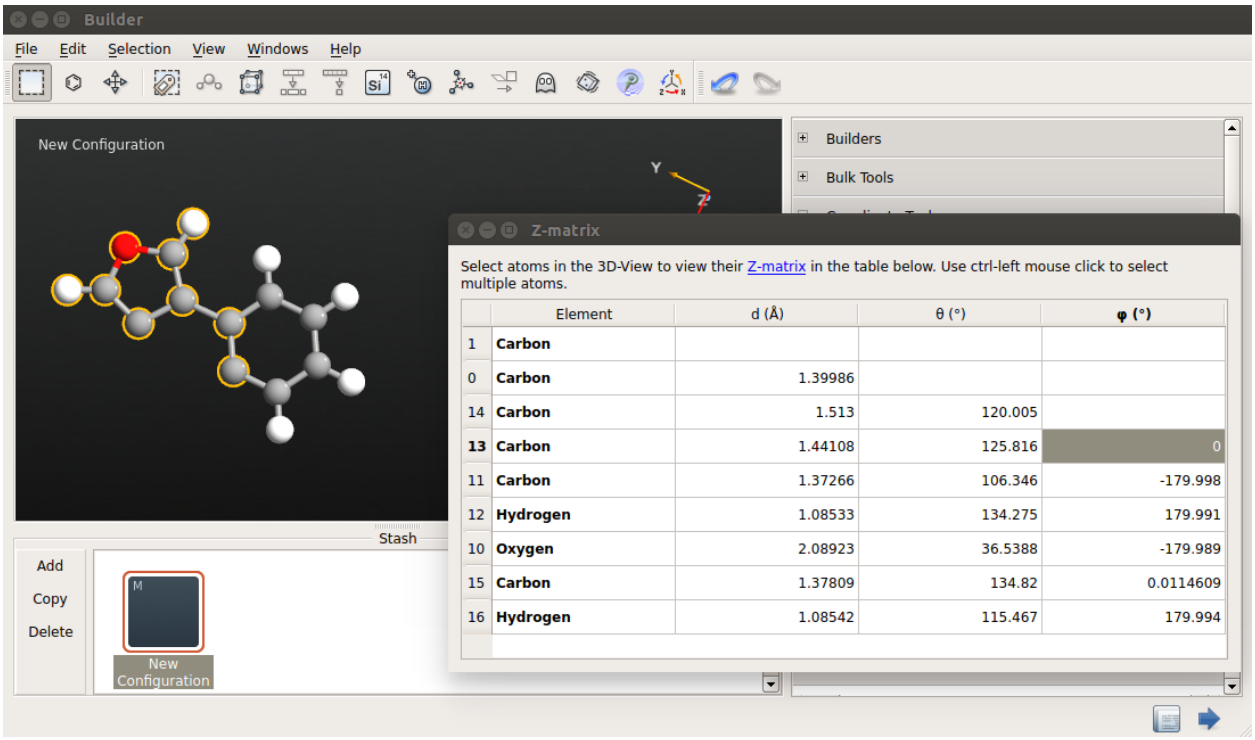
1.First select the atoms that define the dihedral angle. Once the angle is defined, you need to select the atoms that will move with the rotation; select the remaining atoms of furane moiety.





2.Go to *Coordinate Tools* → *Z-matrix* and the **Z-matrix** window will open up.

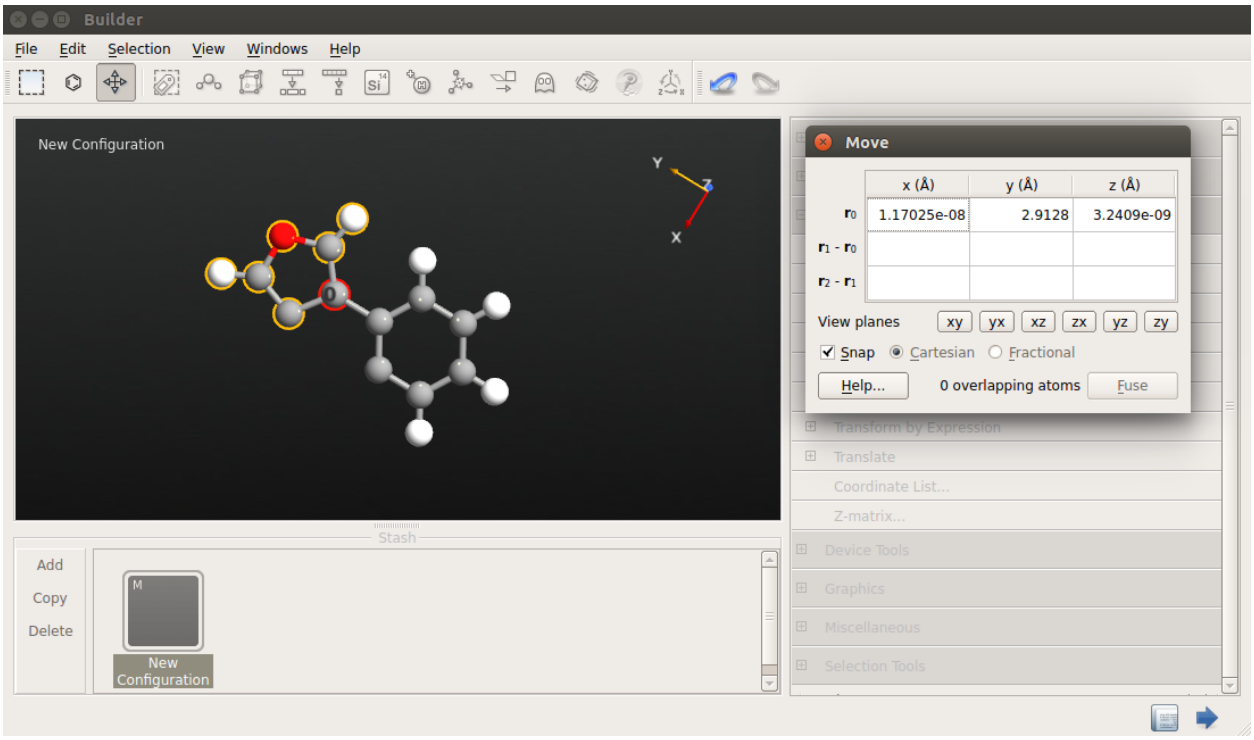
3.Set the value of the angle you want to change as shown in the next figure.



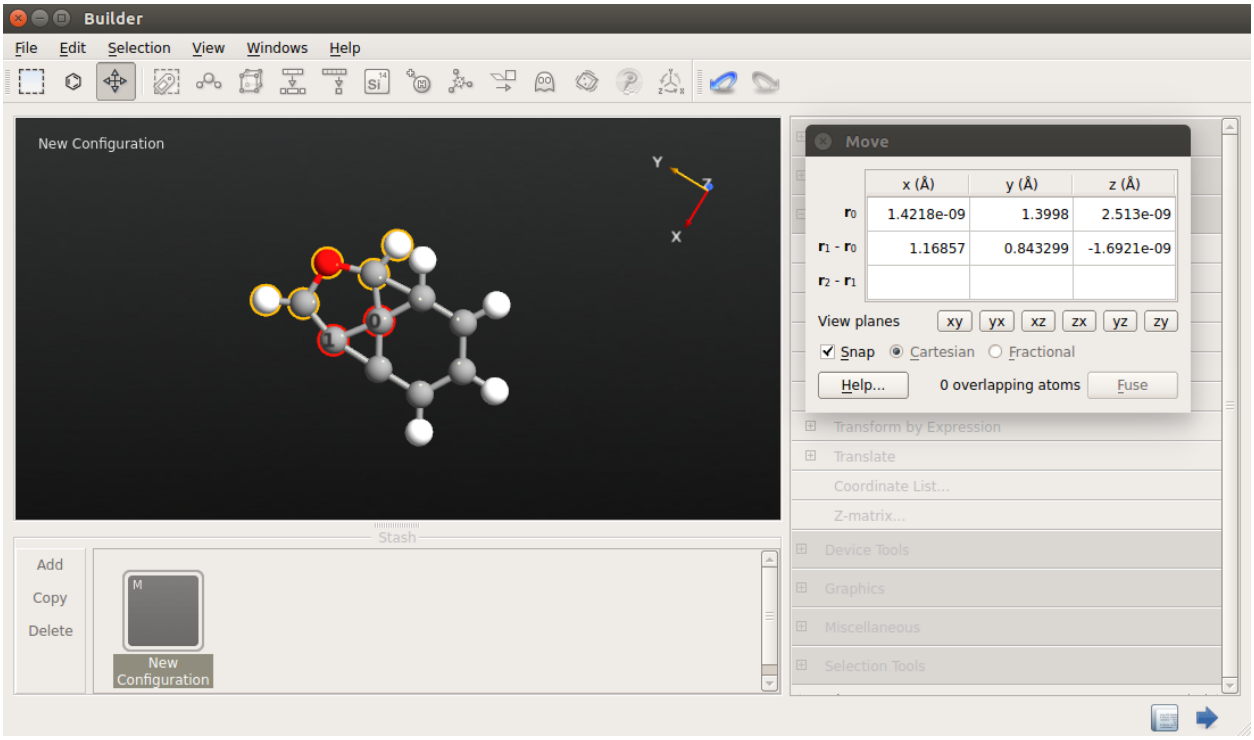
Next, you need to overlap the C atoms that link both rings and remove the overlapping atom by fusing the rings:

1.Select the atoms that will move; the atoms belonging to the furane moiety.

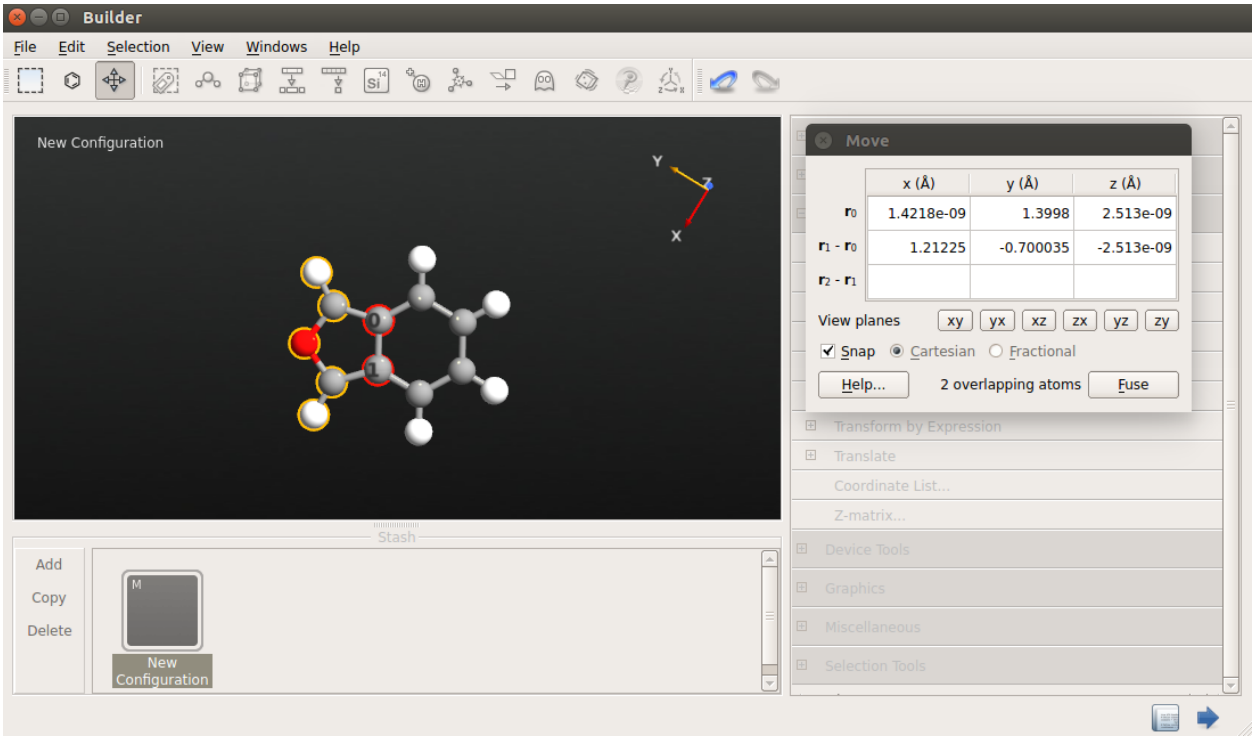
2.Select the **Move** tool and select just the atom of furane linked to benzene moiety.



3.Drag the selected atom so that it ends up on the top of the C atom of bencene linked to it and select the C atom numbered 1 in the below figure.

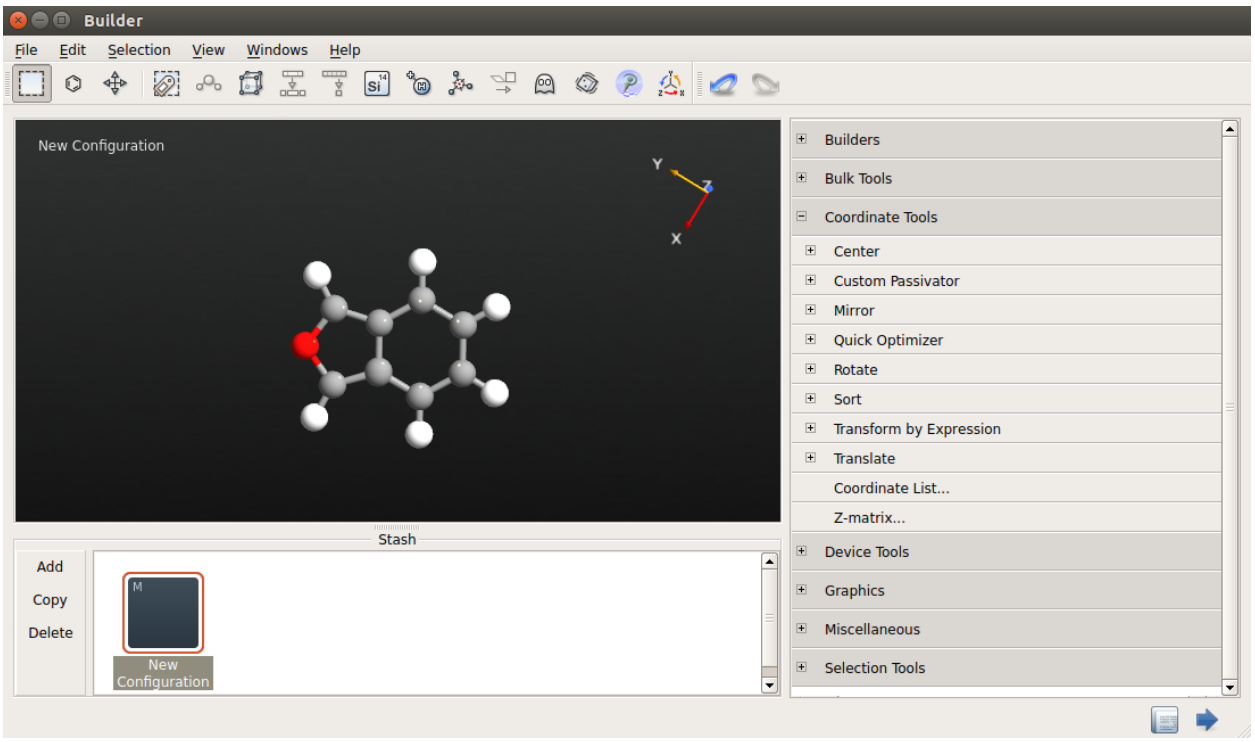


4.Drag the C atom numbered 1 so that it ends up on the top of the nearest C atom of benzene as shown in the figure below.



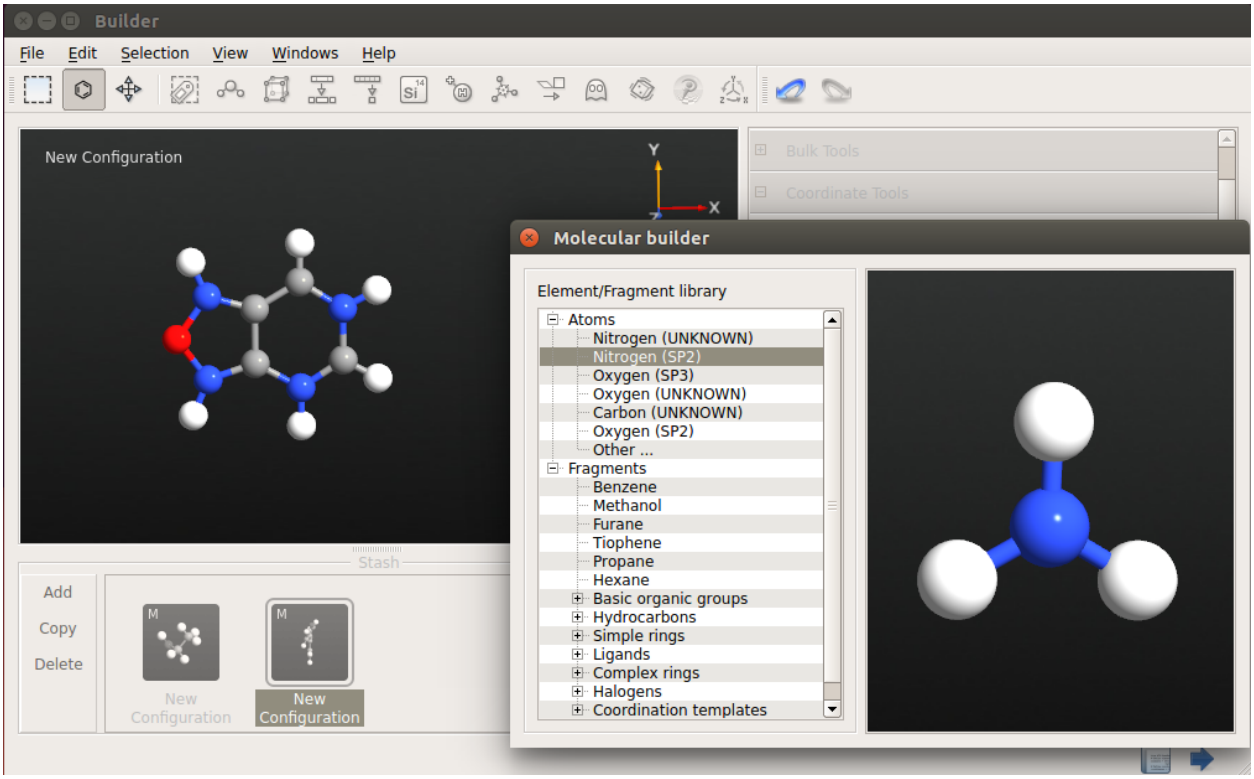
5.Click Fuse to remove the overlapping atoms.

6.Close the **Move** window and click the 3D background to deselect the atoms.

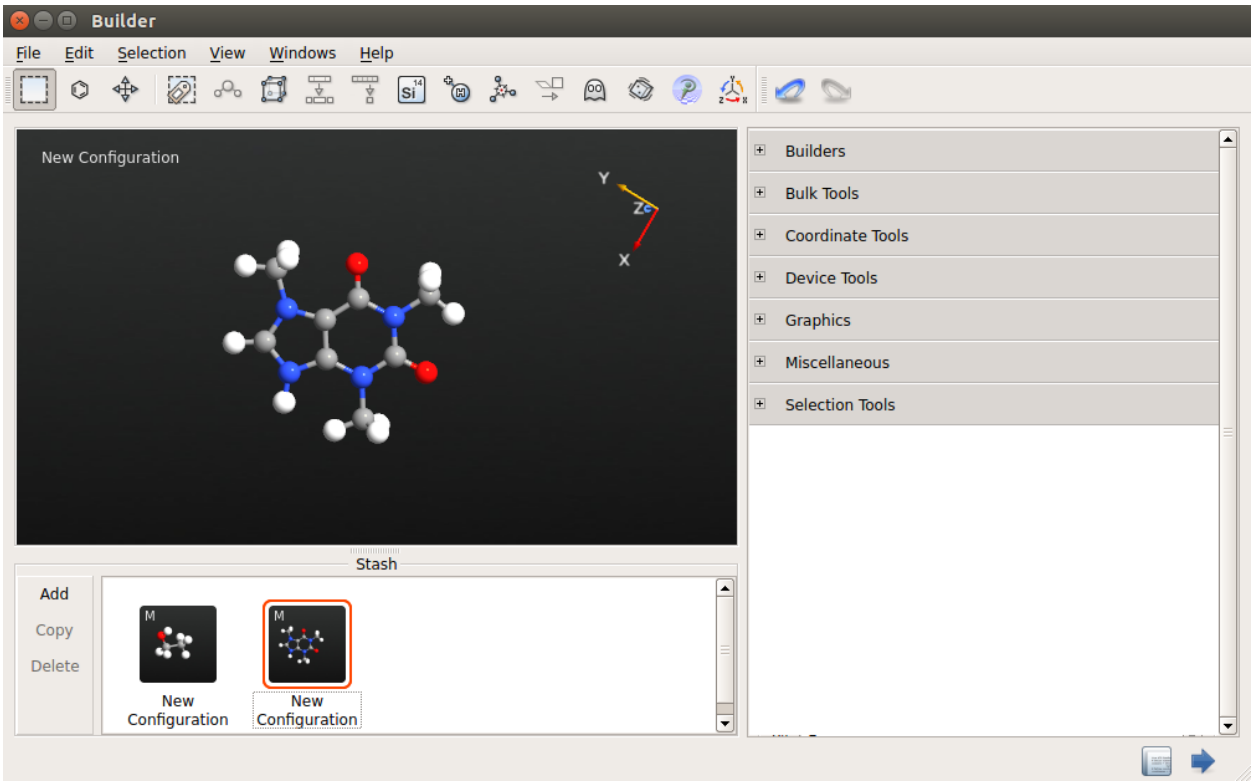


7.Go to **Molecular Builder** and slect sp2 N atom.

8.Replace the C atoms by N as shown in the next figure.



9.Follow the same steps to replace the H atoms by sp2 O atoms and Methane fragments and the oxygen atom belonging to the five-membered ring by a sp2 C atom, as shown in the next figure.



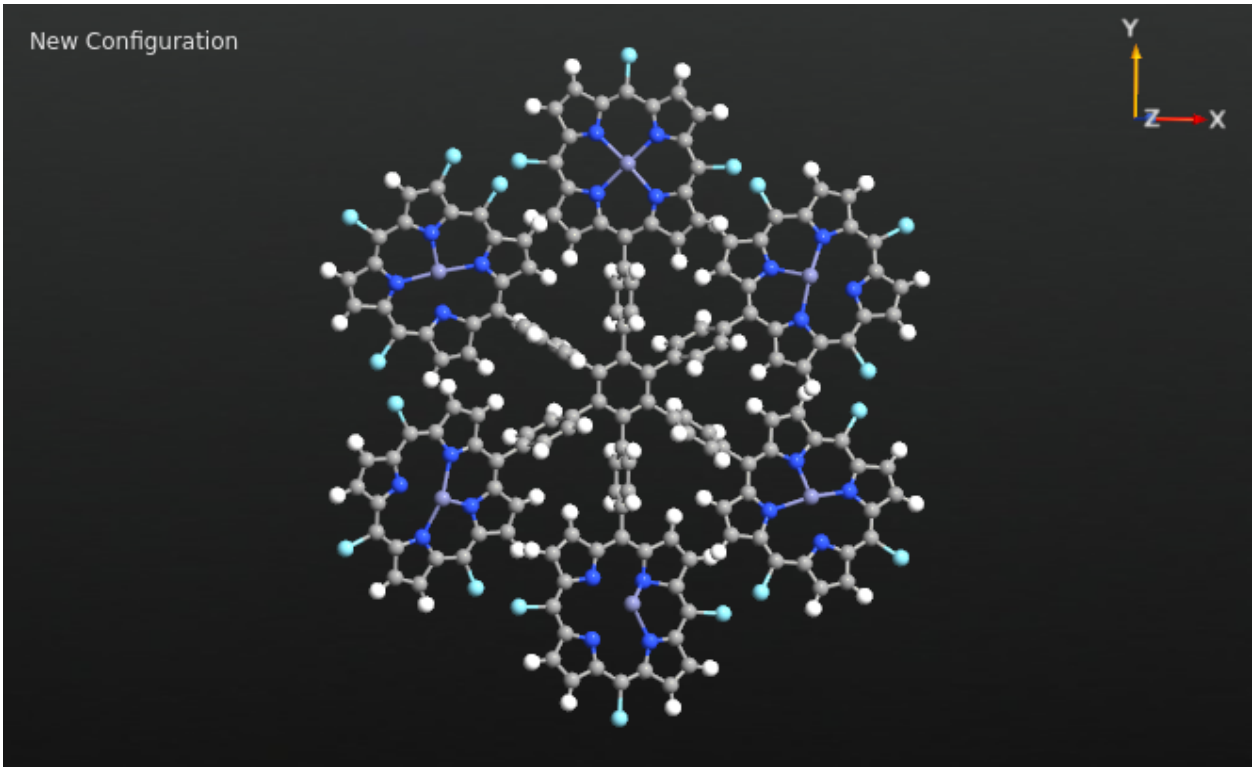
You have built the caffeine molecule! You could also have built it using Indole fragment and making the appropiate

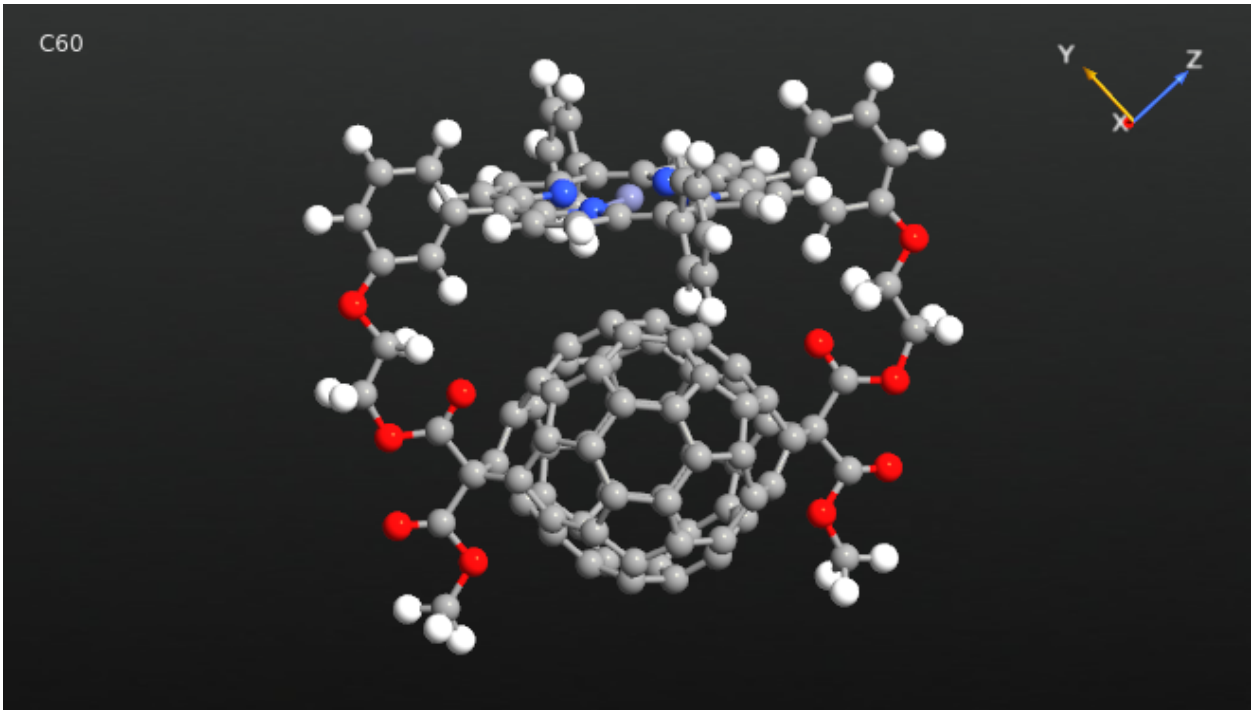
changes in the six- and five-membered rings.

2.10.3 Going further

In this tutorial you have built molecular structures starting from an empty configuration. You can also build a new structure starting from a molecular configuration that you already have or from a configuration imported from the **Database**, using *Add* → *from Database*. You can add as many structures as you want to the Stash. In order to include them in your active configuration, you just need to drag and drop them onto the 3D View.

The next two structures are easily built using the steps outlined above. The first one can be built combining benzene and porphine fragments. For the second, you need to import C60 from the **Database** and then combine it with benzene, porphine, methane and formaldehyde fragments.



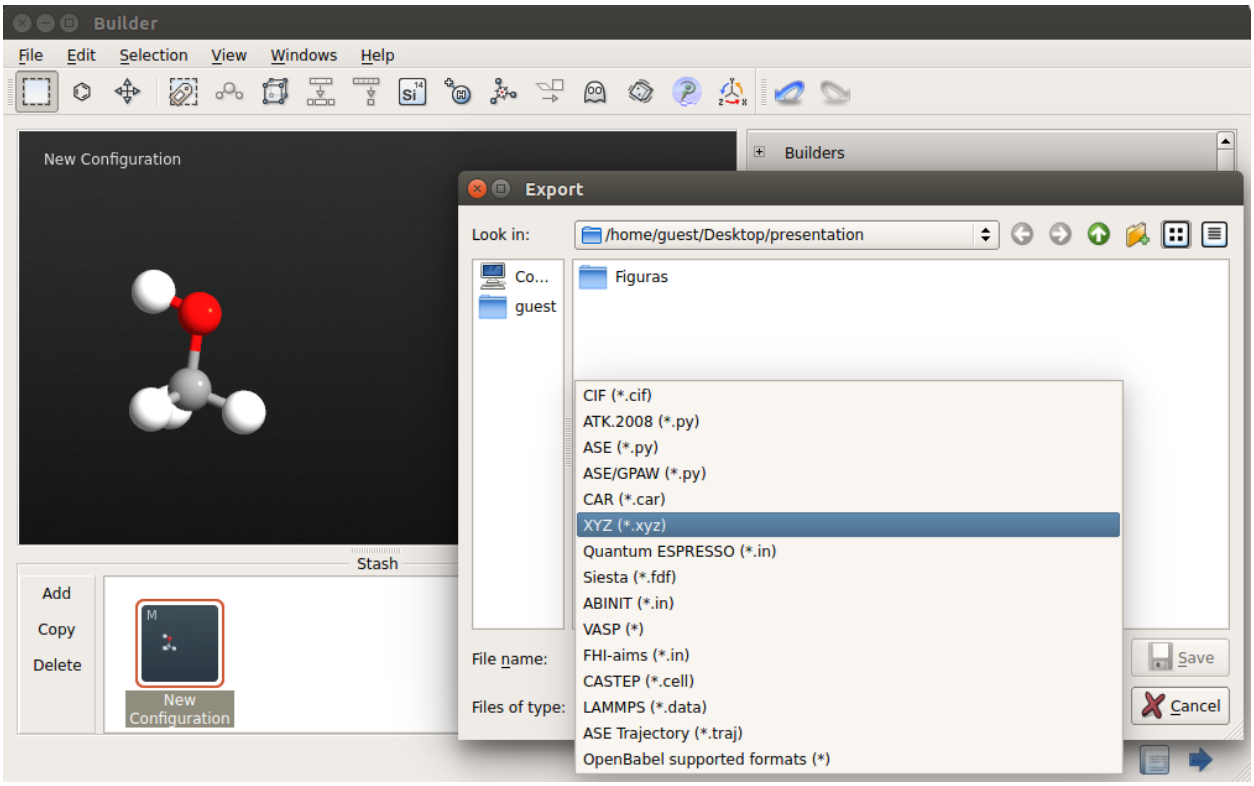


Tip: In order to link a new fragment to a fullerene molecule you will need to change the hybridisation of the carbon atom that will link to the new fragment.

Note: You can click the *Help* button in the Move window to get more details about the Move tool.

2.10.4 Export the Stash Configuration file

Once you have built your molecular structure, you can save it in different formats. You just need to go to *File* → *Export Configuration* as is shown in the next figure.



Please, note that it is also possible to export your configuration to a OpenBabel supported formats.

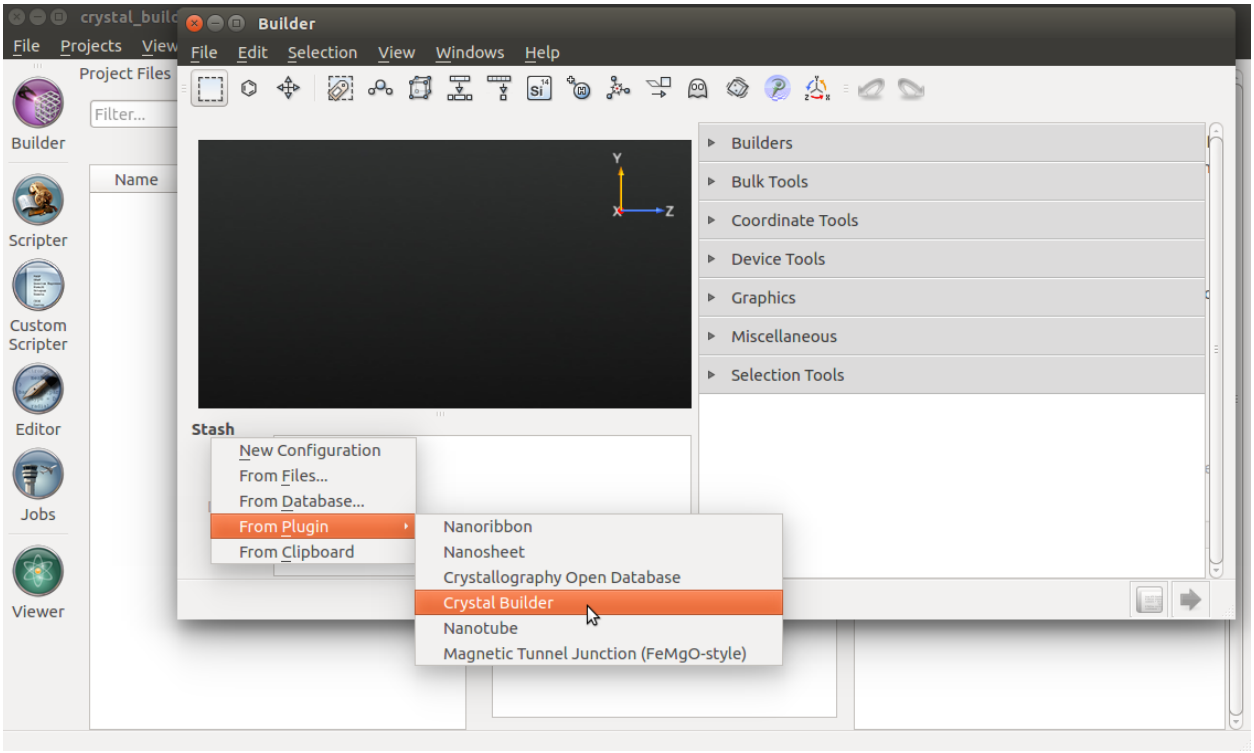
2.11 Using the Crystal Builder

Version: 2016.0

2.11.1 Introduction

QuantumATK has a tool, **Crystal Builder**, which makes it easy to create crystal structures by specifying parameters for the lattice and atom coordinates. The tool is a plugin for the QuantumATK **Builder** and can be found under

Add → *From Plugin* → *Crystal Builder*.



2.11.2 Crystal structure of black phosphorus

As an example, this tutorial focuses on phosphorene (a monolayer of black phosphorus), which is attracting attention in electronics due to its analogy with graphene (see chemistryworld and, e.g., Refs.[1],[2], and[3].)

The table below lists the relevant information about the black phosphorus crystal[4]; space group and lattice constants, and Wyckoff positions for the phosphorus atoms. Based on that information, we first construct the bulk crystal, from which phosphorene is then created.

Space group	*C m c a* (#64)
a	3.3136 Å
b	10.478 Å
c	4.3763 Å
Atom positions	
P	8f: (0, 0.10168, 0.08056)

Hint: Basics of Crystallography

•There are in total 230 space groups, which are enumerated and symbolized in crystallography. In Hermann–Mauguin notation, the 64th group is represented by *C m c a* (a new notation *C m c e* is also used). See, e.g., Ref.[5] for more

[1] H.ăLiu, A.-T. Neal, Z.ăZhu, D.ăTomanek, and P.-D. Ye. Phosphorene: A New 2D Material with High Carrier Mobility. *ArXiv e-prints*, jan 2014. arXiv:1401.4133.

[2] Han Liu, AdamăT. Neal, Zhen Zhu, Zhe Luo, Xianfan Xu, David Tománek, and PeideăD. Ye. Phosphorene: An Unexplored 2D Semiconductor with a High Hole Mobility. *ACS Nano*, 8(4):4033–4041, 2014. doi:10.1021/nn501226z.

[3] Likai Li, Yijun Yu, GuoăJun Ye, Qingqin Ge, Xuedong Ou, Hua Wu, Donglai Feng, XianăHui Chen, and Yuanbo Zhang. Black phosphorus field-effect transistors. *Nat Nano*, 9(5):372–377, may 2014. doi:10.1038/nnano.2014.35.

[4] A.ăBrown and S.ăRundqvist. Refinement of the crystal structure of black phosphorus. *Acta Crystallographica*, 19(4):684–685, Oct 1965. URL: http://onlinelibrary.wiley.com/doi/10.1107/S0365110X65004140/abstract.

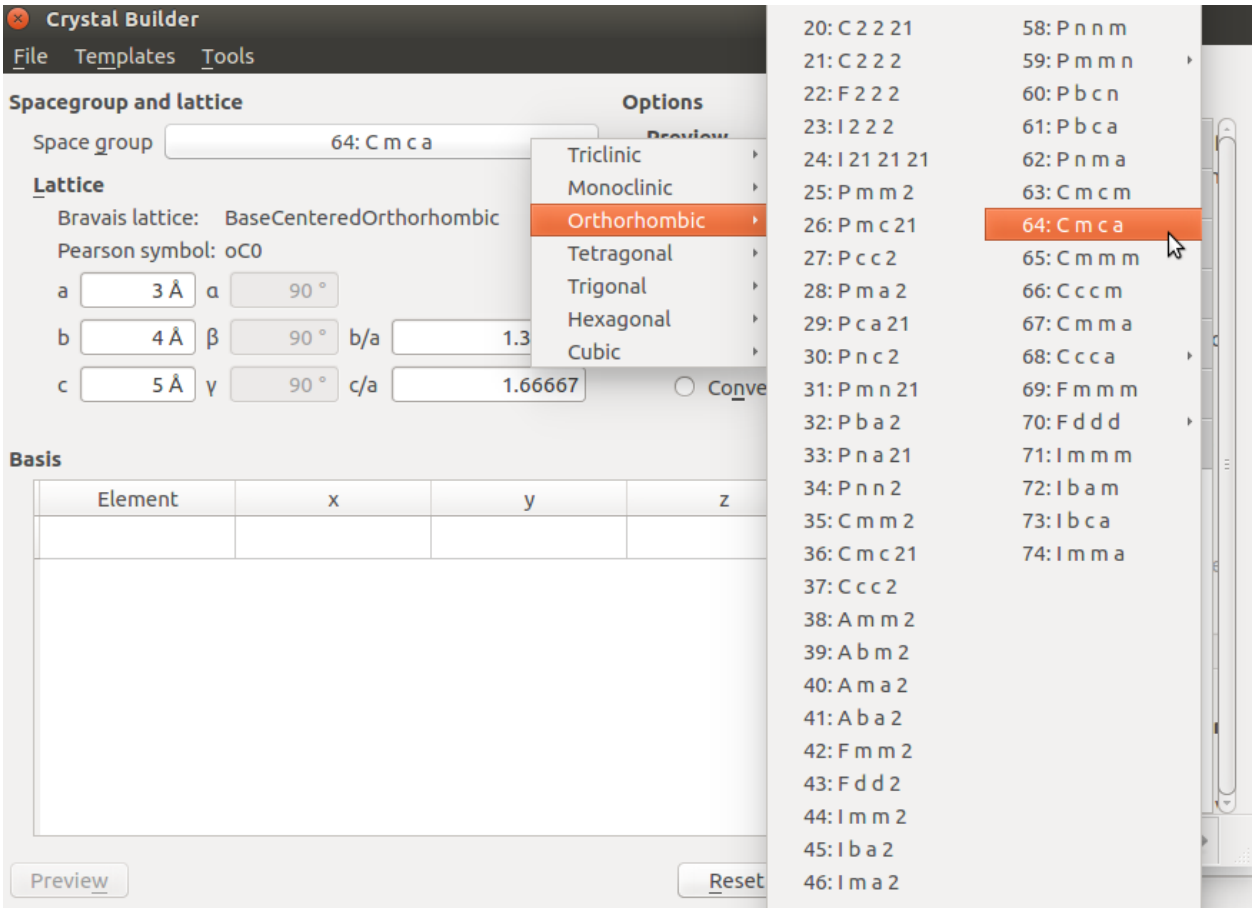
[5] T.ăHahn, editor. *International Tables for Crystallography Volume A: Space-Group Symmetry*. Springer, 2005.

details.

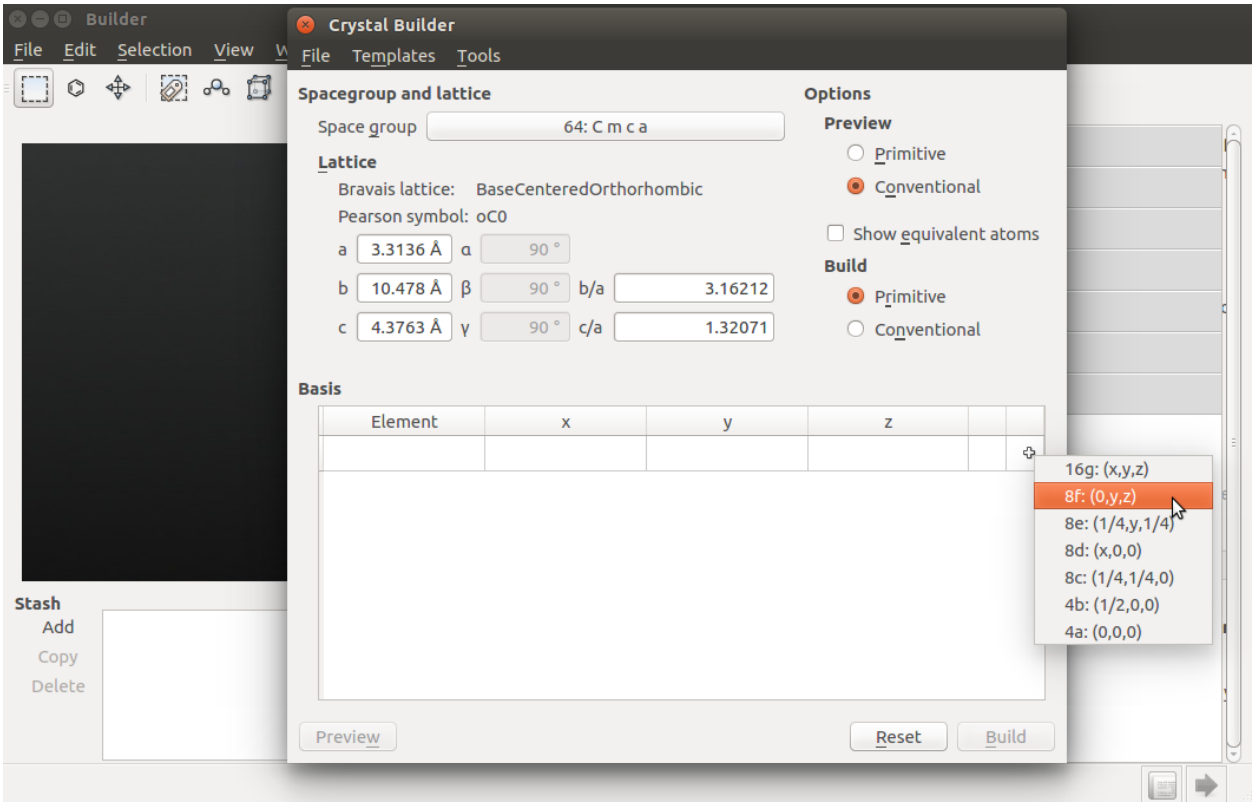
- Wyckoff positions indicate the positions that atoms are allowed to occupy in a lattice belonging to a particular space group. These are specified in fractional coordinates, e.g., (0, 0.10168, 0.08056) in the case of black phosphorus. For each space group, the Wyckoff positions are confined to certain areas in the lattice, e.g., only on the yz-plance in the case of black phosphorus, so as to give all lattice points which are transformed from one another by the symmetry operations belonging to the group.
- Wyckoff positions are denoted according to multiplicity (number of equivalent atoms in a unit cell) and Wyckoff letter (defined for each Bravais lattice). In the case of black phosphorus, the multiplicity is 8 and the Wyckoff letter is f.

Attention: The information given in the table above is enough to fully define the black phosphorus crystal. Inserting P atoms in the Wyckoff position 8f of the lattice with space group #64 yields the black phosphorus crystal.

Let us now build the bulk configuration. First, open the **Crystal Builder** plugin, and select space group 64: C m c a, which is orthorhombic:

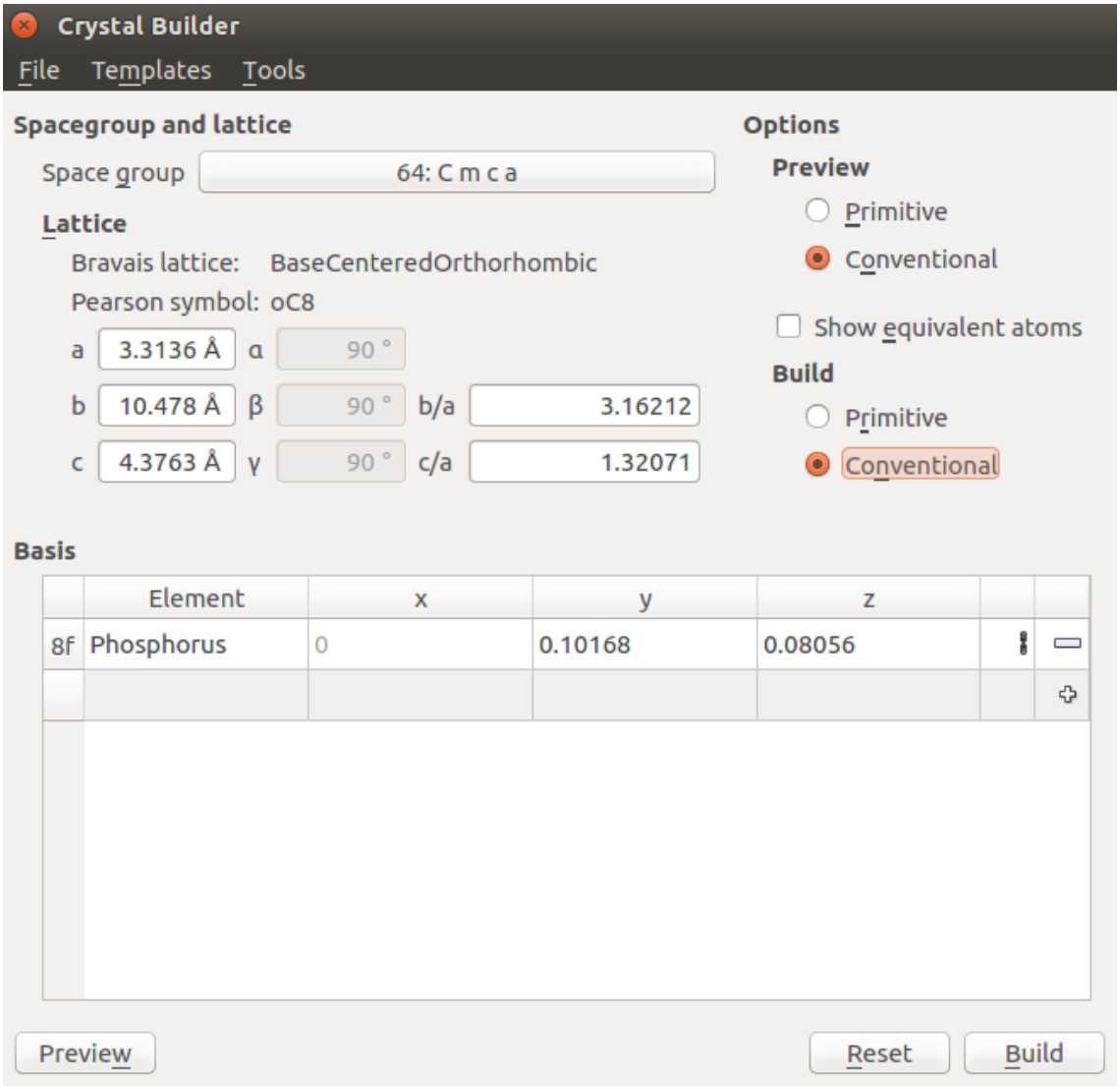


Then insert the a, b and c lattice parameters as listed in the table above, and click the + button to add one atom at the Wyckoff position *8f: (0,y,z)*.

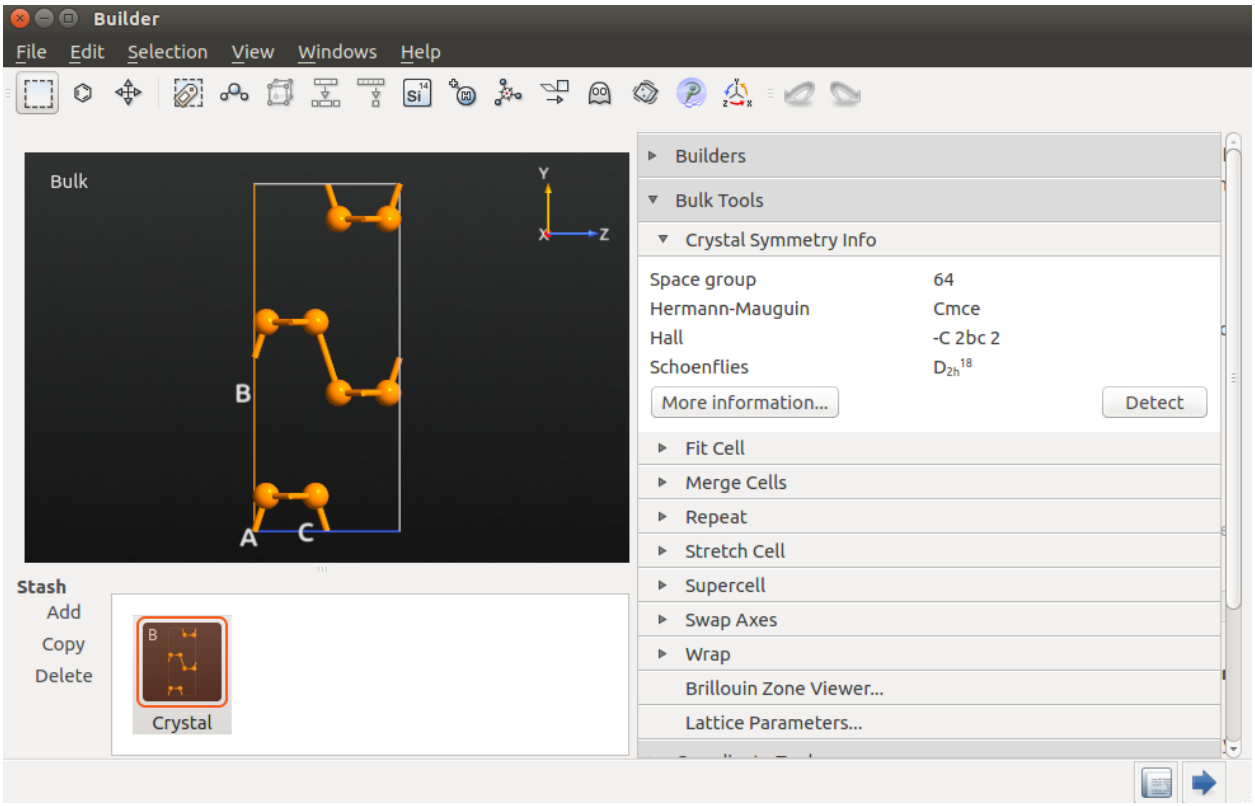


Change the newly added element from hydrogen to phosphorus, and change its y and z positions according to the table above, y=0.10168, z=0.08056 (fractional coordinates). While making these operations, you will see a preview of the structure in the Builder main window.

By default, the Crystal Builder will preview the *conventional* unit cell, but actually build the *primitive* cell. This usually has computational advantages, but in this case the conventional cell is more convenient for the subsequent manipulations needed to prepare the monolayer. Therefore, choose the conventional cell and click **Build** to create your structure.



The black phosphorus crystal is added to the Stash in the Builder, and is ready for further manipulations. Before moving on, open *Bulk Tools* → *Crystal Symmetry Info* and check the symmetry to make sure that the system is truly black phosphorus.



Tip: You can do **3D** rotations and **2D** rotations of any configuration: Simply **click and hold down the right mouse-button** and move the cursor.

•Moving the cursor **inside the Viewer window** gives you **3D** rotations.

•Moving the cursor **outside the Viewer window** gives you **2D** rotations.

•Also note that the cursor changes when you are in rotate mode:

2.11.3 Phosphorene and its bandstructure

One very interesting feature of black phosphorus is the interesting electronic properties when the material is two-dimensional. The name phosphorene was introduced by Liu *et al.* for such a 2D p-type material[Page 277, 1].

Now that you have created black phosphorus, you can easily exfoliate that bulk structure, create phosphorene, and then analyze its electronic properties.

You can find many tutorials on how to use the Builder to create and modify your system and running QuantumATK calculations. Below is a brief example of what you can do with phosphorene – the results can be compared with Ref.[Page 277, 1].

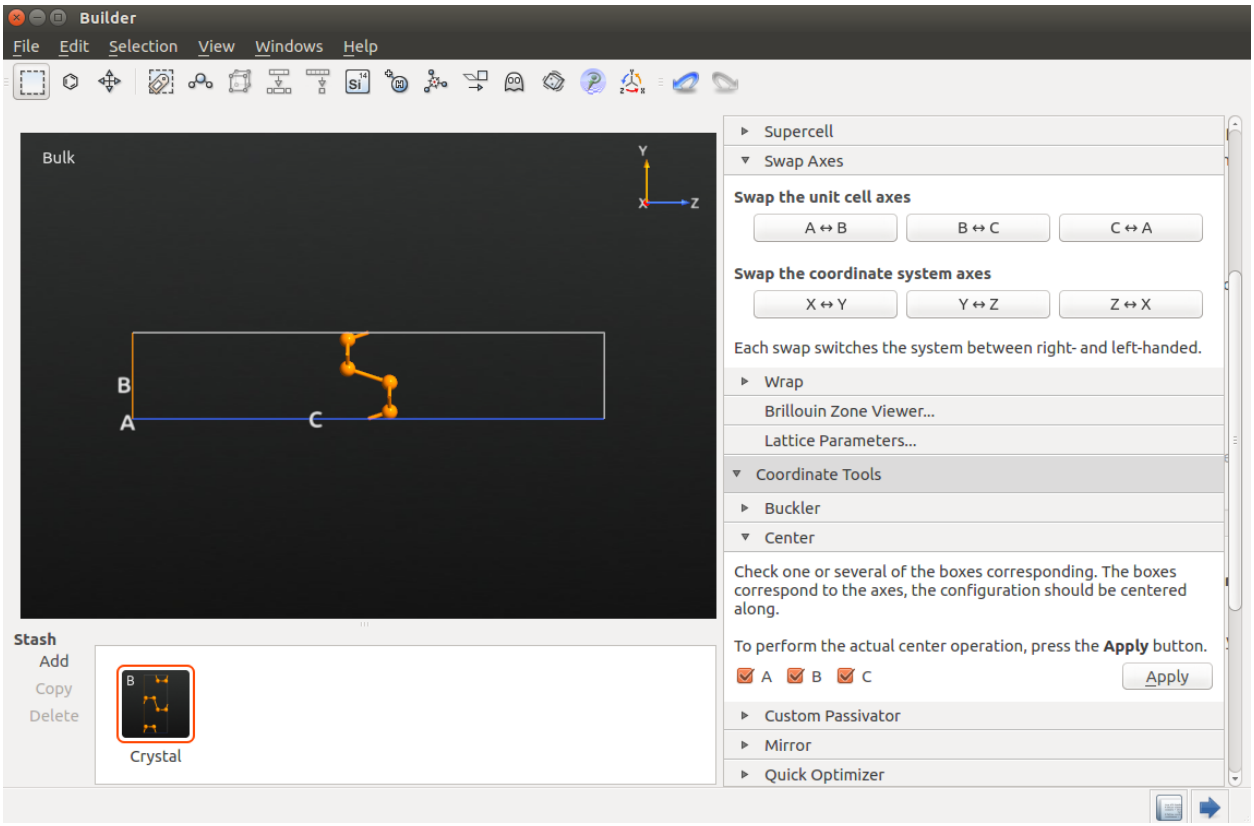
The image below shows phosphorene, created in the QuantumATK Builder by following these steps:

1. Open the *Bulk Tools* → *Swap Axes* tool and click B-C and Y-Z in order to align the phosphorene plane perpendicular to the C axis.
2. The lattice type has now been reset. You therefore need to open *Bulk Tools* → *Lattice Parameters* and select *Simple Orthorhombic* as the lattice type.

3.Delete four P atoms such that you have a monolayer.

4.Open again the *Bulk Tools* → *Lattice Parameters* tool and increase the C vector length to 24 Å. This adds a vacuum region along the non-periodic C-direction.

5.Finally, use the *Coordinate Tools* → *Center* tool to center the configuration in the unit cell.



Hint: Meanings of these manipulations

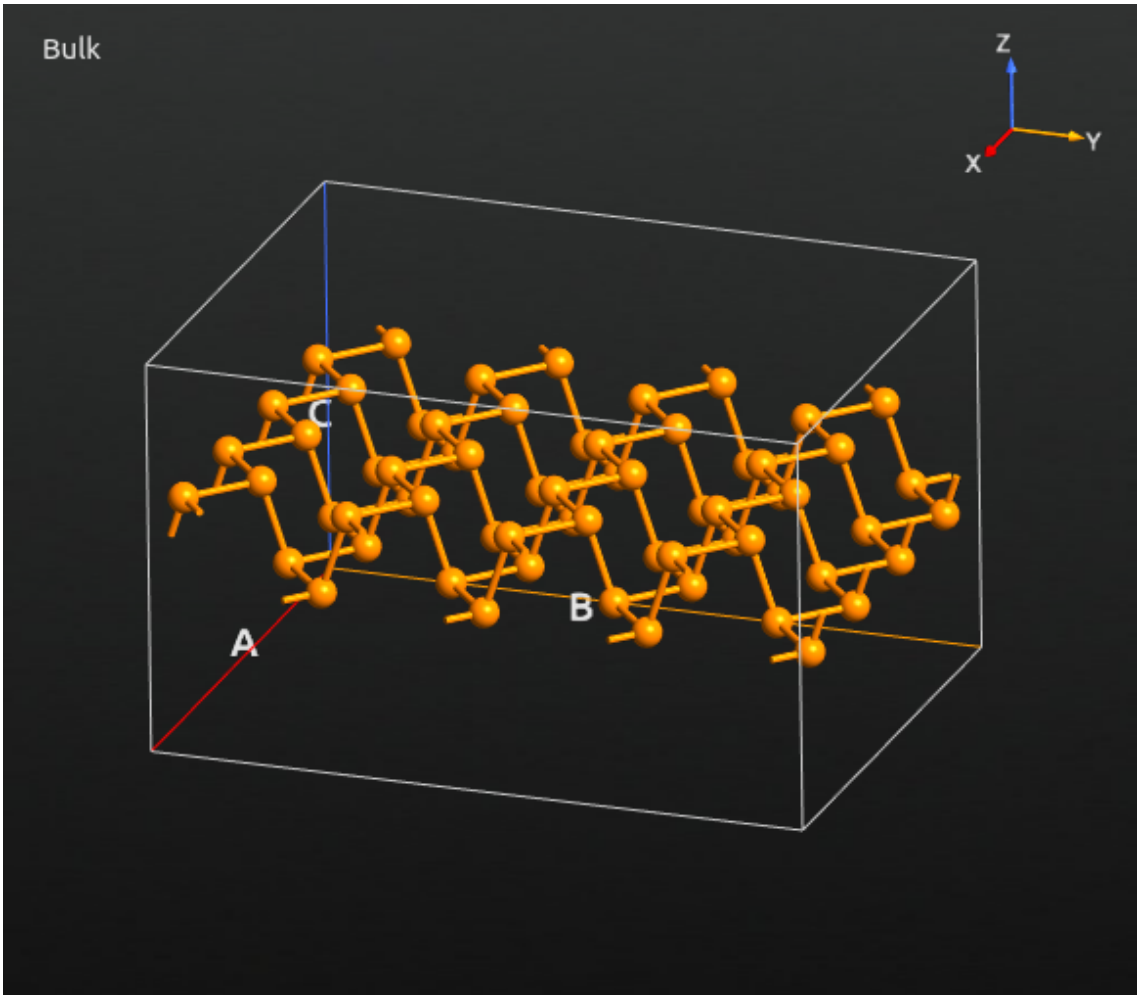
•Manipulation 1 sets each layer to be normal to the z axis, and the zigzag and armchair directions to be parallel to the x and y axes, respectively.

•Manipulation 3 ensures that each conventional unit cell contains a monolayer.

•Manipulation 4 then increases the distance to the repeated images of the monolayer along the C-direction, such that they are electronically decoupled. The configuration thereby represents a true monolayer.

•The snapshot below shows the monolayer obtained after manipulations 1–5 (the structure is repeated 3 times along A and B in order to clearly visualize the monolayer).

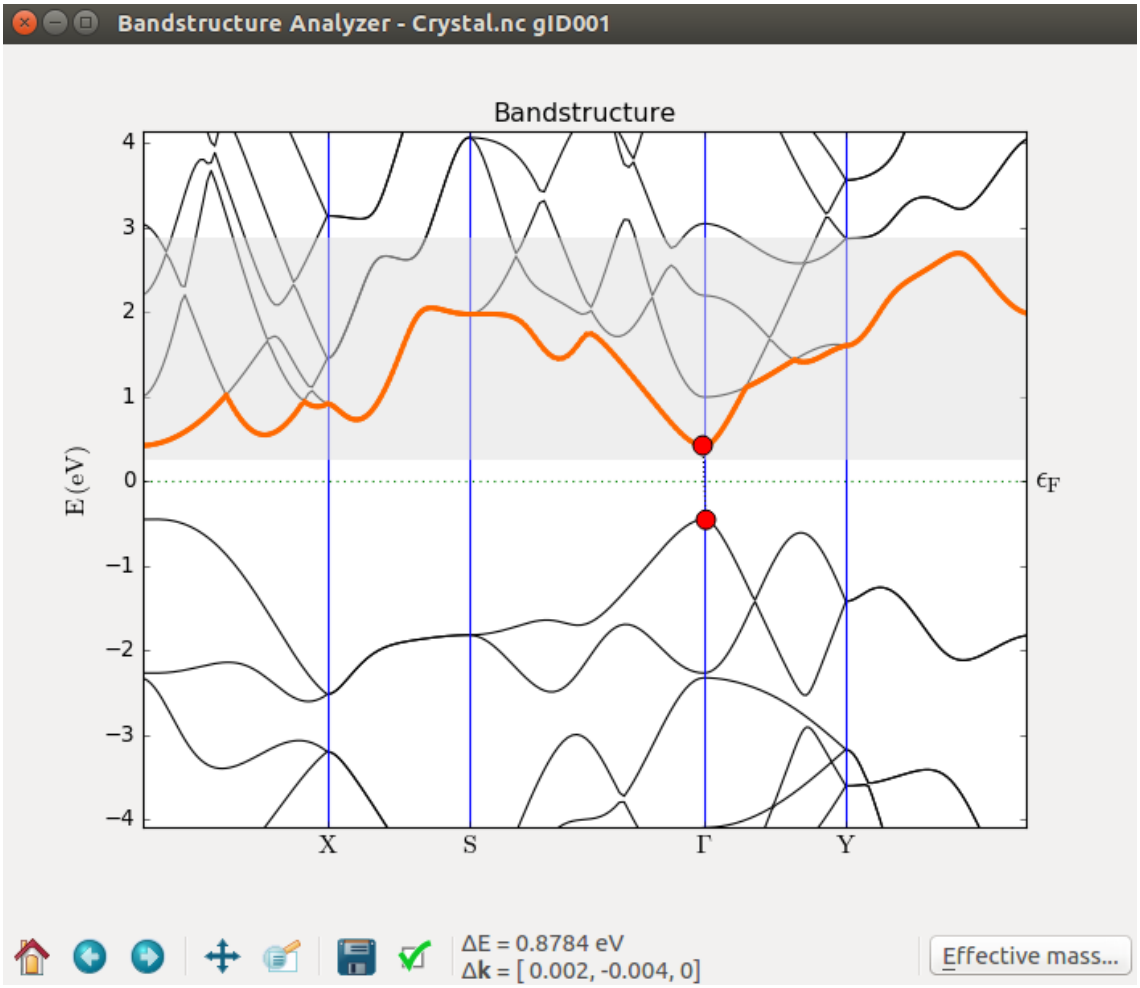
•The definition of the first Brillouin zone of the monolayer depends on the orientation of the unit cell in real space (ABC with respect to XYZ), as well as the orientation of the atoms in the cell (atom positions with respect to ABC). Manipulations 1–5 provide the same orientations as in Refs.[Page 277, 1] and[Page 277, 2], and therefore the same first Brillouin zone as in those references.



Note: You can export this configuration to a CIF file. Once you have the configuration in the **Builder** Stash, you can use the *File* → *Export Configuration* menu or simply right-click the configuration.

Bandstructure

Finally, calculate the bandstructure of phosphorene along the X–S–G–Y Brillouin zone path. The band structure along this path is shown in Refs.[Page 277, 1] and[Page 277, 2]. You should find a band gap of about 0.87 eV, in agreement with the results in these references.



The following ATK-DFT calculator parameters have been used:

•PBE exchange-correlation functional.

•Norm-conserving FHI pseudopotential.

•*DoubleZetaPolarized* basis set.

•90 Hartree density mesh cutoff energy.

•8x8x1 k-point sampling grid.

2.11.4 References

2.12 The Builder Console

The Builder Console is an expert tool in the Nanolab Builder, where you can manipulate the Builder configurations using QuantumATK scripting.

Also, Builder Plugins can expose QuantumATK commands in the console. When plugins write to the console, a visual counter is shown over the console icon.

Unfold the console by clicking the console icon in the bottom left corner of the Builder.

•*Special variables*

–*The active configuration*

–*The selected atoms*

–*The active camera*

–*The Builder Stash*

•*Console Snippets*

•*The console in Builder plugin development*

•*Questions*

2.12.1 Special variables

You can execute any valid QuantumATK command through the console. In addition the console also exposes four magic variables, which allow you to interact with the configurations in the Builder.

The active configuration

The special `configuration` variable always refers to the active configuration in the 3D window. If the current configuration is methane or CH_4, you can use QuantumATK to perform operaitons such as querying the elements list:

```
In [-]: configuration.symbols()
Out[-]: ['C', 'H', 'H', 'H']
```

You can also manipulate the current configuration, using the exposed methods:

```
In [-]: configuration.deleteAtoms([1, 2])
In [-]: configuration.shiftAtoms([0, 0, 1] * Angstrom, indices=[0, 1, 2, 3])
In [-]: configuration.findBonds()
```

Read more about the exposed methods by consulting *MoleculeConfiguration*, *BulkConfiguration*, *SurfaceConfiguration*, *DeviceConfiguration*, *NudgedElasticBand* in the reference manual.

You can also replace the current configuration by assigning to the `configuration` variable:

```
In [-]: configuration = repeat(configuration, 2, 2, 2)
In [-]: configuration = MoleculeConfiguration([Oxygen, Oxygen],
                                              [[0, 0,     0] * Angstrom,
                                               [0, 0, 1.208] * Angstrom])
```

In the Builder Console the configurations also expose their title. This is special for the Builder Console, and cannot be used in other QuantumATK scripts.

```
In [-]: configuration.title()
Out[-]: 'Methane'
```

The selected atoms

The special `selection` variable contains a list of the selected indices of the active configuration.

```
In [-]: selection
Out[-]: array([1, 2])
```

You can change the current selection by setting the selection like this:

```
In [-]: selection = [0, 1, 2, 3]
```

Note that the first element is numbered 0, the second 1 and so on. You can also select All or None of the atoms.

```
In [-]: selection = All
In [-]: selection = None
```

The active camera

The special `camera` variable allows for the manipulation of the current 3D window camera. Center the camera on the configuration, by calling:

```
In [-]: camera.center()
```

The Builder Stash

The list of configurations in the left side of the Builder is called the Builder Stash, and the console has a special `stash` variable that refers to this list of configurations.

Simply retrieve configurations in the stash by referring to their index or title:

```
In [-]: stash[0]
In [-]: stash['GaAs']
```

The latter only works if there is only one configuration titled GaAs. Note that the first element is numbered 0, the second 1 and so on.

Configurations in the stash can be overwritten by assigning different configuration, for example:

```
In [-]: stash['GaAs'] = MoleculeConfiguration([Oxygen, Oxygen],
                                              [[0, 0,   0  ],
                                               [0, 0, 1.208]] * Angstrom)
```

In the stash you can also query both for title and index:

```
In [-]: stash.title(0):
Out[-]: 'GaAs'
In [-]: stash.index('GaAs')
Out[-]: 0
```

The titles of the stash items can also be changed:

```
In [-]: stash.setTitle('GaAs', 'GalliumArsenide')
```

Configurations can be added to and deleted from `stash` by using the methods `insert`, `append`, `duplicate` and `delete`:

```
In [-]: stash.insert(2, new_configuration)  # Or using the title stash.insert('GaAs', new_
↪configuration)
In [-]: stash.append(new_configuration)
In [-]: stash.duplicate(2)                  # Or using the title stash.duplicate('GaAs')
In [-]: stash.delete(2)                     # Or using the title stash.delete('GaAs')
```

You can also switch to another active configuration. This will also update the special `configuration` variable.

```
In [-]: stash.activate(index)
```

The stash also supports iteration,

```
In [-]: for c in stash:
      :     print(c.title())
Out[-]: Methane
      : SiC
      : SiC [111]
```

You can also access the currently selected entries in the stash. Select multiple items by clicking on configurations in the Builder Stash with your mouse. Then call the `stash.selected()` method to retrieve a list of the selected configurations.

```
In [-]: stash.selected()
Out[-]: [Configuration(), ..., Configuration()]
```

An iterating over the selected titles and e.g. centering their camera.

```
In [-]: for item in stash.selected():
      :     stash.activate(item.title())
      :     camera.center()
```

2.12.2 Console Snippets

Sometimes you may wish to execute same set of commands in the Builder Console several times over. The Console snippets help you do just that without having to bother with copy-pasting preformatted commands.

Unfold the console and access the snippets by clicking on the button to the top-right of the console. The first time you do this, the following menu opens:

•configuration_info

•stash_info

•New snippet

•Open snippets folder

The first two entries are example snippets that ship with QuantumATK. Each snippet show a simple example that uses the `configuration` and the `stash` variable respectively.

Click a snippet to insert it into the console. You can also insert and execute the snippet by clicking the running man icon, or open the snippet in the snippet editor by clicking the pencil icon.

New snippets are created by clicking the `New snippet` menu entry. This opens the Snippet editor, where you can rename, edit, delete and execute (run) the snippet in the Builder Console.

Since snippets are just python scripts, they are saved in the *~/.vnl/buillder/console* subfolder. You can open the folder by clicking `Open snippets folder`.

2.12.3 The console in Builder plugin development

Note: This section discusses the development of **BuilderToolBarPlugins** or **BuilderPanelPlugins**, and only relevant if you wish to develop your own plugins.

Builder plugin developers can insert and execute commands in the Builder console.

Several native plugins already expose their QuantumATK commands in the Console. Try using either the **Repeat**, **Center** or **Passivate** Builder plugins to see how it works.

Builder plugins (`BuilderToolBarPlugin` and `BuilderPanelPlugin`) are passed a reference to the Console object in their constructor, and can access it through the `plugin.console()` method.

The Console object exposes the following API:

```
console.execute(script)
```

This writes the script to the console and executes it afterwards.

For advanced users it is also possible to append a script to the console history, and insert additional parameters into the console scope.

```
console.append(script, parameters=None)
```

The `parameters` keyword must contain a dictionary with the defined variables and their values. Note that it is not possible to overwrite the special variables, `configuration`, `selection`, `camera` and `stash`.

It is also possible to simply insert a script into the console without executing it. This can be done by invoking the `write` method on the console object.

```
console.write(script)
```

2.12.4 Questions

Do not hesitate to write questions or comments about the Builder console in the QuantumATK User Forum

2.13 AddOns

2.13.1 Creating AddOns

If you want to create an addon we strongly recommend to read the relevant tutorial: *AddOns*. There is no special restriction for addon creation in general. The Synopsys QuantumATK Team creates addons that ship with the software, but any other QuantumATK user has the option to create addons either for personal use or for wider usage (expanding QuantumATKs capabilities) after Synopsyss consent. For more details read `<installation_path>/atkpython/lib/python3.11/site-packages/AddOns/LICENSE` on Linux installations, or `<installation_path>atkpythonLibsite-packagesAddOnsLICENSE` on Windows. Also you can get more details in the above-mentioned tutorial. The tutorial just gives the general framework through some examples since, obviously, the design of an addon depends on its purpose.

2.13.2 Disabling or Uninstalling AddOns

Its quite simple! Just open the (Help >) **AddOn Manager**, choose the addon(s) you want and click the **Uninstall** button. When restarting QuantumATK these addons will have been removed.

2.13.3 Getting Addons

Some (necessary) addons are installed alongside with the QuantumATK installation on your system. Read more in section *Managing plugins using the AddOn Manager* about getting additional addons.

2.13.4 Installing AddOn

Usually, the addon installation is a straightforward operation to perform. However, you must take into account the kind of user who is using QuantumATK. Namely, there are two different kinds of users:

- **Limited access users**: They can use QuantumATK but cannot modify the installation, which is in control of a system administrator.
- **Administrators**: They have a local installation on their systems where they are local administrators, they can install and uninstall QuantumATK (or other software) themselves.

Install AddOns as limited access users

While local administrators can just run QuantumATK with admin rights and install addons, limited access users have to take an additional step since they cannot write to the main installation folder. In the *Preferences* dialog of QuantumATK, one can find the option to install addons in a separate folder, which is located in the user profile directory. After enabling this option any user can install additional addons.

Installing AddOns on the local system

We distinguish the following cases:

A. Install official addons. In this case just follow these steps (assuming you have internet connection):

- **Step 1:** Open QuantumATK and click the Help button on the toolbar.
- **Step 2:** Choose AddOn Manager - a window pops up (see above).
- **Step 3:** Choose the addon you want (directly or using the search box).
- **Step 4:** Restart QuantumATK to activate the new add-on.

B. Ready to install zip file. In this case we assume that you have already the zip folder of the addon (downloaded from our website or elsewhere) somewhere on your system and would like to install it (no internet connection is required).

- **Step 1:** Open QuantumATK and click the Help button on the toolbar.
- **Step 2:** Choose AddOn Manager - a window pops up.
- **Step 3:** Click the button Install zip/folder.
- **Step 4:** Navigate to the downloaded zip file and install it.
- **Step 5:** Restart QuantumATK to activate the new add-on.

2.13.5 Failing AddOn Installation

There are many possible reasons for this. The most common problems come usually from the following facts:

•Incorrect addon design.

•Problems with your internet connection.

•Some dependencies are missing on your system (a.e. the addon calls external binaries).

•Insufficient permission, see *Install AddOns as limited access users*.

In any case we strongly suggest to open QuantumATK from a command-line terminal in order to see the error messages in detail.

2.13.6 AddOns Location

All the addons which are installed system-wide, are located at `<installation_path>/atkpython/lib/python3.11/site-packages/AddOns` on Linux and `<installation_path>atkpythonLibsite-packagesAddOns` on Windows. If the user has enabled the option to install addons in the user profile directory, the newly installed addons will be located at `<user_profile_dir>/.quantumatk/<quantumatk_version>/AddOns`.

2.13.7 Updating AddOns

In the (Help >) **AddOns Manager** window click the **Refresh** button to make sure the *Installed* and *Latest* versions showed in the table are up to date. Then select an AddOn and click the **Update** button, or the **Update All** button to execute it on all AddOns. Then restart QuantumATK to make sure all changes are applied.

2.13.8 What Is an AddOn

An addon is a code unit that can be added to QuantumATK in order to increases and enhances its capabilities. The addons of QuantumATK are python (.py) files zipped (.zip) into unified bundles. You can find more details at *AddOns*.

2.13.9 Check Installed AddOns

You can see the installed addons either through the (Help >) **AddOns Manager** or by checking the *AddOns Location*.

CHAPTER

THREE

TUTORIALS

3.1 New or Updated

3.1.1 Training and Finetuning of MACE models

Version: X-2025.06

Downloads

```
Train_model_from_scratch.py                          Train_model_with_naive_finetuning.py
Train_model_with_multihead_finetuning.py  Train_mace_models.py  md_example_results.hdf5
c-am-TiSi-TrainingSets.hdf5  run_benchmarks.py  Trained_model_usage_example_with_MD.hdf5
Tutorial_MACE_amorphous_and_crystal_structures_all_data.zip
```

In this introductory tutorial, you will learn how to use QuantumATK to

- Train custom MACE machine-learned force-fields (ML FFs) via Python scripts which can be used for molecular dynamics (MD), geometry optimization, and other force-field-based simulations - in this tutorial for crystal and amorphous TiSi and $TiSi_2$ structures.
- Validate the quality of the trained MACE models.
- Load a trained MACE model into QuantumATK in a workflow via the **MachineLearnedForceFieldCalculator** block.

In recent years, advancements have been made in the world of Machine Learned Force Fields (ML-FFs) with the emergence of universal foundation models. One of the current state-of-the-art potential architectures yielding a good trade-off between performance and accuracy is MACE[1] - a message-passing Neural Network (MPNN) from which prominent universal foundation model versions are preinstalled and available for use in QuantumATK. While such universal models can constitute an excellent calculator in its own right, it is trained to perform reasonably well within a very broad domain. This means that it has trained on a large amount of data, and that the model has a size that limits the inference speed. For problems within a limited application domain, it is also possible to train custom MACE models within QuantumATK. MACE models can be trained either from scratch, which can be used to balance model size and simulation speed against accuracy, or by finetuning foundation models to achieve even better accuracy in a specific application domain while building on top of previous knowledge.

The amount of data needed to train a MACE model depends on the complexity of the system and the kind of training being conducted. In general, more data is usually preferred if training from scratch. However, for fine-tuning from a foundation model on the order of 100 structures can be enough to reach good accuracy in a specific domain. In this

[1] Ilyes Batatia, DavidăPeter Kovacs, Gregor N.ăC. Simm, Christoph Ortner, and Gabor Csanyi. MACE: higher order equivariant message passing neural networks for fast and accurate force fields. In AliceăH. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*. 2022. URL: https://openreview.net/forum?id=YPpSngE-ZU.

tutorial, both approaches will be presented using a small dataset containing a mix of 241 crystal and amorphous structures of TiSi and $TiSi_2$. This data was used to train a Moment Tensor Potential in a different *tutorial*. The dataset is available for download: `c-am-TiSi-TrainingSets.hdf5`. It should be noted that this system is chosen for demonstration purposes. For simple systems, the accuracy advantage of MACE over MTP is not significant. Training an MTP model may be the better choice here, due to its advantages in simulation speed. MACE is in general expected to be capable of reaching higher accuracy in complex systems where more traditional ML FFs face limitations.

Note: It is in general advised to do all training of MACE models on GPUs using 1 process per GPU. It is similarly recommended to run inference with trained MACE models on GPUs to utilize the model efficiently. Instructions on how to run on GPUs can be found in the *technical notes*.

In order to train a MACE model, we utilize the *MACEFittingParameters* and *MachineLearnedForceFieldTrainer* objects. The parameters object is used to set up all training parameters, while the trainer object is used to perform the training. In the reference manual, all available parameters are listed and explained. In this tutorial, only a subset of the parameters with high impact on the model training and accuracy are included. When training from scratch, the most important parameters are the maximum L value, which controls the symmetry (equivariance) of the messages sent within the model architecture, the distance cutoff, which controls the cutoff radius up to which the local environment should be encoded for each node, and the number of channels, which controls the number of features the local environment of each node is described by.

Note: In the **Version:** X-2025.06 release, MACE model training modules are only available in scripting. Workflow Block support in the GUI is not yet included.

Training a MACE model from scratch

We first set up a script to train a small MACE model from scratch. The script is available for download: `Train_model_from_scratch.py`. The script first loads the training data to be used for training. It is important that the training data already includes energy, forces, and (if desired) stress labels, as the MACE training framework does not perform these calculations. The data should be stored in one or more *TrainingSet* objects, which are then appended to a single list. Next, the script sets up the calculator to be used for calculating the isolated atom energies for the atom types featured in the training set data. It is important that the calculator is consistent with the precalculated training data. The calculator can either be loaded from a file if saved as part of the preparation of the training data or it can simply be redefined in the training script directly using the same settings.

Thereafter, the script sets up a *MACEFittingParameters* object that defines the model and hyperparameters for the MACE training.

Finally, the script sets up the *MachineLearnedForceFieldTrainer* object that coordinates the other objects and is used to perform the training.

The script trains a single MACE model on 80% of the 241 provided structures and validates the model on the remaining 20% in each training epoch. Various parameters are set to tune the model size, train and inference speed, as well as the accuracy of the model. The 3 most important parameters introduced in the previous section are set to the following values:

- `max_l_equivariance = 0`
- `distance_cutoff = 5.0`
- `number_of_channels = 8`

Here, the `number_of_channels` parameter is set to 8, which is a very low setting. Higher values will be explored in a small screening study for this parameter later in the tutorial.

Here, the `max_l_equivariance` parameter is set to 0, which corresponds to MACE models named or referred to as small or L0 MACE models. Similarly, `max_l_equivariance = 1` corresponds to medium/L1 models and `max_l_equivariance = 2` corresponds to large/L2 models. This is the parameter that influences computational speed and accuracy of a model the most.

The remaining parameters in the script are accompanied by short explanatory comments in the script. Additional parameters exist and are described in the *MACEFittingParameters* reference manual.

The main role of the *MachineLearnedForceFieldTrainer* object is to coordinate the training process. It takes as input the training data, the calculator, and the fitting parameters. Additionally, it takes care of creating the train-test split if desired. For MACE training it is not required to have a test set. A test set can be used to calculated RMSE values for the energies, forces, and stress on data that has never been seen before at the end of the training. This is therefore different from the validation set, which is created by taking a fraction (editable in the *MACEFittingParameters*) of the training data and using it to validate the model during training. By default, the validation set is 20% of the training data and the validation is run after each training epoch.

In order to run the training, the script is loaded in the QuantumATK Editor from where a Job can be created and submitted via the Job Tool. Instructions how to run on GPUs can be found in the *technical notes*. Using a single node, with a single MPI process, and a single GPU (Nvidia A100) requested in the Job tool the training script took approximately 9 minutes to run when creating this tutorial.

Before looking into the results of the training it is also presented how finetuning of existing models can be performed.

Naive Finetuning of foundation MACE models

Finetuning a foundation model can improve accuracy of the model for a specific domain while benefitting from the general capabilities of the original model. Due to the nature of finetuning, the architecture of the model is fixed and therefore cannot be changed into a smaller or larger model. The size is tied to the foundation model. Finetuning has the advantage that general interactions between types of atoms are already mapped quite well and that for specific domains at specific levels of theory these interactions only have to be finetuned - not learned from scratch which can be a lengthy process for large amounts of data.

In QuantumATK finetuning is done by setting the `foundation_model_path` parameter in the *MACEFittingParameters* object to the path where the foundation model is located. This model can be any MACE `.model` file trained previously. In this tutorial we will use the `mace-mp-0b3-medium` model[2], one of the most recent, high quality MACE models at the time of the **Version:** X-2025.06 release, available at: mace-mp-0b3-medium.model.

Note: Finetunable foundation models are not included in the QuantumATK installation. Therefore, any universal MACE model to be finetuned must first be downloaded and then provided to the `foundation_model_path` parameter. Due to the size of the file, it may be beneficial to store the model in a location on a cluster and have the *MACEFittingParameters* reference the specific cluster location instead of copying the model from the local machine each time a job referencing the model is submitted.

The training script for finetuning the model with the TiSi and $TiSi_2$ dataset is available for download: `Train_model_with_naive_finetuning.py`. The script is similar to the one used for training from scratch. The main difference is the `foundation_model_path` parameter pointing to the location of the foundation model in the *MACEFittingParameters* object. **When running the script, ensure that the path to the foundation model is updated to the location of the model on your system or the cluster where you will run the training.**

During naive finetuning, the foundation model is trained further on new data for a specified number of epochs. While this can significantly improve accuracy within the new, limited domain, it also carries the risk that some of the general knowledge encoded in the original foundation model may be forgotten. This effect can be more pronounced if the new dataset is very different from the original training data. Attention to final model quality and the number of epochs

[2] Ilyes Batatia, Philipp Benner, Yuan Chiang, AlinăM Elena, DávidăP Kovács, Janosh Riebesell, XavierăR Advincula, Mark Asta, Matthew Avaylon, WilliamăJ Baldwin, and others. A foundation model for atomistic materials chemistry. *arXiv preprint arXiv:2401.00096*, 2023.

to train for is therefore important to balance adaptation to new data with retention of the foundation models broader capabilities.

Using a single node, with a single MPI process, and a single GPU (Nvidia A100) requested in the Job tool the training script took approximately 10 minutes to run when creating this tutorial.

Multihead Finetuning of foundation MACE models

Multihead Finetuning, or Multihead Replay Finetuning, is a more refined approach to finetuning. In this approach, a foundation model also forms the basis of the training. Contrary to the Naive Finetuning approach, not only the foundation model is used, but also the training data used to train the foundation model. In practice this results in a model training on several model heads with separate readout layers but otherwise shared model weights. The first model head is the pretrained model head, which will be retrained on a subset of the original training data. How much of the original training data is used can be customized but by default 10000 original structures are used. The other model head will be the model head for the new training data. During training, non-readout weights are trained on both sets of training data, and therefore the model will better retain knowledge about the original training data. This minimizes the risk of the model straying away from key patterns learned from the original data while still allowing the model to learn from the new training data.

Since the Multihead Finetuning approach requires both a foundation model as well as the original training data, more parameters need to be set in the *MACEFittingParameters* object. In order to activate the Multihead version, the `use_multiheads_finetuning` parameter is activated. Furthermore, the `pretrained_head_train_file` parameter is set in one of two ways. If the original model is not one of the MP MACE models, the parameter is given the path to the original training data (in xyz file format). If the model is one of the MP MACE models, the parameter is simply set to `pretrained_head_train_file='mp'`. In this case, the original training data as well as a descriptors file need to be downloaded and local/cluster paths need to be provided to the script. The data used to train the MACE MP foundation models[Page 293, 23] used in this tutorial is available at:

- https://github.com/ACEsuit/mace-foundations/releases/download/mace_mp_0b/mp_traj_combined.xyz
- https://github.com/ACEsuit/mace-foundations/releases/download/mace_mp_0b/descriptors.npy

These files, if Multihead Finetuning with a MP MACE model, have to be supplied via the `mp_data_path` and `mp_descriptors_path` parameters in the *MACEFittingParameters* object. Furthermore, the `foundation_model_path` parameter has to be set in the same way as for the Naive Finetuning.

The training script for Multihead Finetuning is available for download: `Train_model_with_multihead_finetuning.py`. The script is similar to the one used for training from scratch apart from the extra parameters described. In order to run the script, ensure that the paths to the foundation model, MP data, and the descriptors files are correct and updated to where you put the files on your machine/cluster. It is recommended to store them on the cluster where training will happen to reduce the time it takes to transfer files from the Job Tool upon submission.

Using a single node, with a single MPI process, and a single GPU (Nvidia A100) requested in the Job tool the training script took several hours to run when creating this tutorial. It may be advantageous to not run this script but wait for a multi-GPU version in the beginning of the next section also training this as well as other models in less time.

For any MACE training, the job can be monitored via its log output while running. Here, loss scores and error metrics (RMSE by default) for energy, forces, and stress are reported on the validation set after each epoch. After the end of the training, the error metrics are reported for the best trained model on the train and validation set - as determined by the model with lowest loss score. For the Multihead Finetuning, the loss and error metrics are also reported for the model head retraining on the original data. The `default` model head is associated with the new training data and the `pt_head` model head is associated with the original training data. The `train` and `valid` prefixes indicate whether the metrics are reported for the training or the validation set. As an example of the log output, the following error metrics are obtained as the Multihead Finetuning training script finalizes:

[3] Bowen Deng, Peichen Zhong, KyuJung Jun, Janosh Riebesell, Kevin Han, ChristopherăJ Bartel, and Gerbrand Ceder. Chgnet as a pretrained universal neural network potential for charge-informed atomistic modelling. *Nature Machine Intelligence*, 5(9):1031–1041, 2023.

config_type	RMSE E / meV / atom	RMSE F / meV / A	relative F RMSE %	RMSE Stress (Virials) / meV / A (A^3)
train_default	15.9	94.2	7.94	14.8
train_pt_head	71.2	77.8	5.86	9.2
valid_default	17.3	100.8	9.47	13.1
valid_pt_head	46.2	87.3	4.54	8.5

The RMSE values for the validation set are in general slightly higher than for the training set for the new model head, `_default`. In the `_pt_head` model head, the picture is a little more mixed, but the validation error for the forces, which has the largest weight, is also larger than the training error. This behaviour is expected because the model only updates its parameters based on the training data during the training. The validation set is used for monitoring the model performance during training without impacting the model parameters. The combination of train and validation error metrics indicates the overall magnitude of the model accuracy. A true measure of accuracy on completely unseen data is however necessary in order to truly assess the predictive capabilities of the model. When training a model from scratch or with Naive Finetuning, only two rows of error metrics will be reported since only one model head is trained. For this Multihead Finetuning example, four rows are obtained due to training on two model heads. For most practical purposes, mainly the `_default` head is important to obtain good error metrics for since this is the model head that will automatically be extracted to a `.qatkpt` model file for use in QuantumATK when using Multihead Finetuning.

Small study with additional models trained from scratch

In order to give a slightly more nuanced view of how changing MACE training parameters can affect accuracy and performance of a model trained from scratch, before testing the models on unseen data, a few more models will be trained from scratch with `number_of_channels` varying between 8 and 128 while keeping the `distance_cutoff` of 5 Angstrom and the `max_l_equivariance` of 0 (corresponding to small/L0 MACE models) fixed. This is all done in one training script where the various models defined using *MACEFittingParameters* objects that have been conveniently generated by calling the previous *MACEFittingParameters* object in the training series to copy its settings while only setting the parameters to be updated. In this way we can easily generate new model training parameters with increasing numbers of channels. If the models from the previous sections have not yet been trained, they can also conveniently be generated by use of this last training script, available for download here: `Train_mace_models.py`. Instead of running the training on a single process and a single GPU, we this time utilize that the implementation of MACE training in QuantumATK allows distributed training. By setting the parameter `distributed_training=True`, the training can be run on multiple GPUs (with 1 MPI process per GPU). In order to run the script, ensure that the script is updated with the path to the MACE model file to finetune from as well as the paths to the original training data files.

Tip: If using multiple GPUs it is in general recommended to use the same number of MPI processes and GPUs and no threading. The training should be kept to a single node. Running training in a distributed setup can result in potential error messages not always being easy to interpret/available. If coming across errors in distributed training, it can therefore be helpful to attempt running the training on a single process and GPU to retrieve the underlying error message.

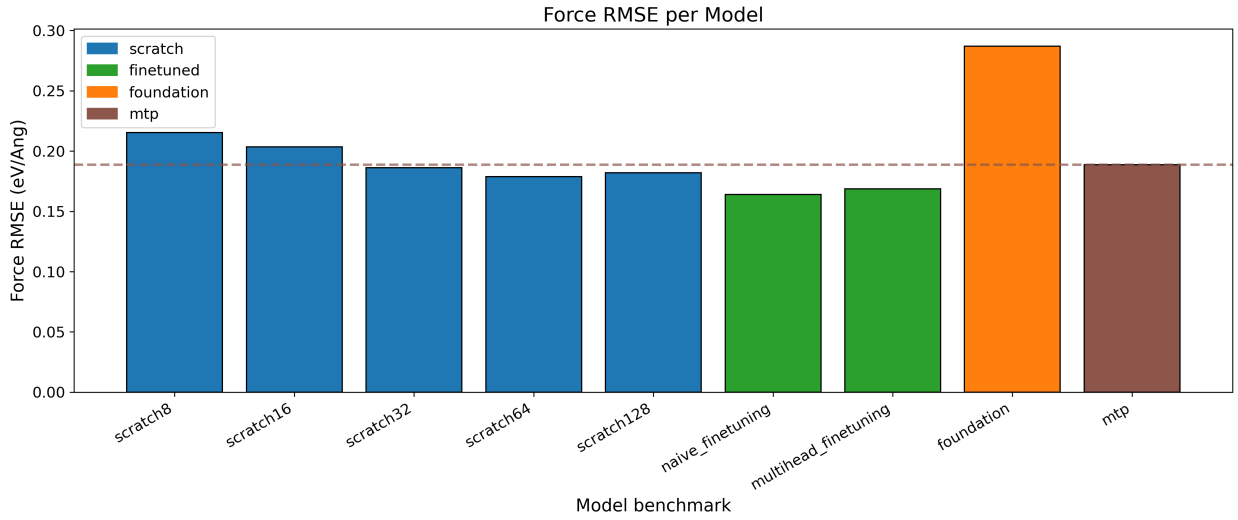
Running the script on 8 MPI processes and 8 GPUs (Nvidia A100), the training of the 7 models took approximately 2 hours in total.

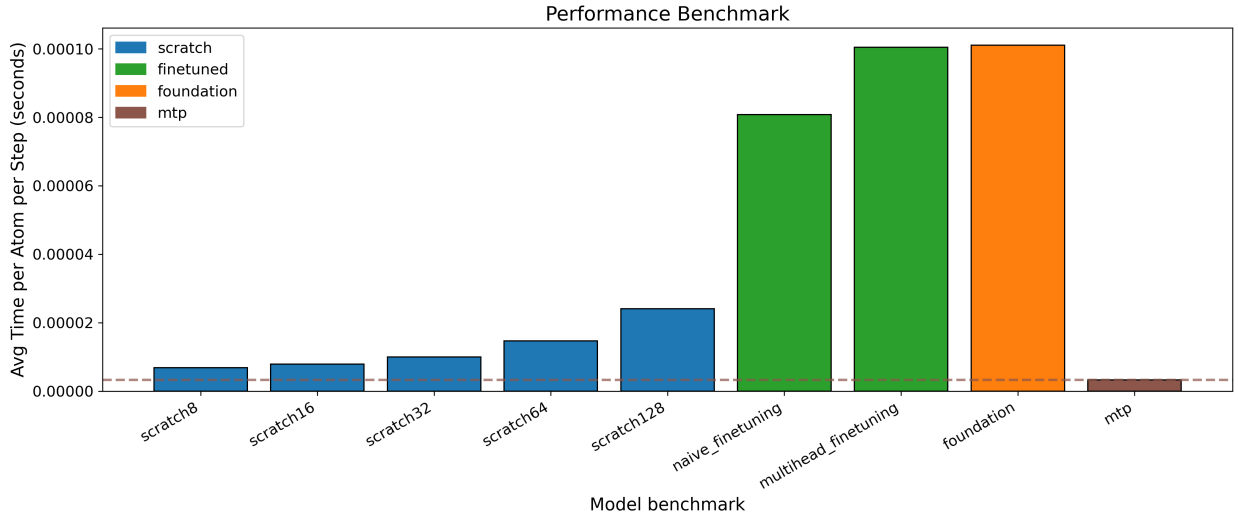
Validation of trained models

With the models trained we can now validate the models by comparing the energies and forces of the MACE models with reference DFT data for unseen data generated with Molecular Dynamics with the reference DFT calculator. This data, containing a mix of crystal and amorphous structures, is stored in the file `md_example_results.hdf5`.

With a different script, `run_benchmarks.py`, we can load the trained models and the reference data to compare the energies and forces of the MACE models with the reference DFT data, as well as run performance benchmarks of an approximately 5000-atom structure for 1000 MD steps. The output of the script is json files containing accuracy RMSE values for the energy and the force predictions as well as the performance benchmark results for the time taken per atom per step in the Molecular Dynamics simulation for each model type. On top of comparing the 7 models trained with scripts in the tutorial, benchmarks for the foundation model - included as the `TorchX_MACE_MP_0b3_medium` potential set in the **Version:** X-2025.06 release - and the MTP - trained in the related MTP *tutorial* using the same training data - are also included in the benchmarks to increase the amount of insight gained. The benchmarking script was run using a single node, with a single MPI process, and a single GPU (Nvidia A100).

The results of the benchmarks in the json files are shown in the figures below.





The two plots above provide a comparison of the different MACE models trained from scratch, the finetuned models, the foundation model, and the MTP model. The forces RMSE plot show that increasing the number of channels in the

MACE models trained from scratch leads to a consistent improvement in accuracy. Based on this plot, it is evident that the finetuned MACE models are the best performing models achieving higher accuracies than the MTP, the foundation model, and the models trained from scratch.

Note that the obtaiend accuracy (for both energy and other predicted quantities) can fluctuate depending on the initial weights. In this tutorial each model was only trained once, with a given random seed, and therefore the accuracy of the models is not guaranteed to be the best possible. In a more systematic study, one might want to run several training runs with different initial weights (by setting another `random_seed` in *MACEFittingParameters*) in order to increase the likelihood of the best possible model being created.

For a brief comparison between the train/validation errors and the test errors, we will swiftly zoom in on the forces error metrics (which have the highest weight - and therefore importance - in the loss function) of the multihead finetuned model metrics. In the error metrics table from the output log in the **Multihead Finetuning of foundation MACE models** section, the train and validation errors for the forces are 94.2 meV/Å and 100.8 meV/Å, respectively. The test error for the multihead finetuned model in the plot above is 168.6 meV/Å. This indicates that the model generalizes reasonably well to unseen data, but with a somewhat lower actual accuracy than reported on the train/validation set. The more training data is included, the more likely it is that the model will generalize well to unseen data. Similarly, the more diverse training data included, the higher the accuracy of the model is likely to become across all error metrics.

Models with higher numbers of channels (and in the case of the finetuned models which have 128 channels and a `max_l_equivariance` setting of 1 corresponding to a medium/L1 level model) achieve lower RMSE values for forces, indicating better agreement with the reference DFT data. However, this improvement comes at the cost of increased model size and slower inference, as also summarized in the performance benchmark plot where the small models trained from scratch make predictions almost as fast as the MTP while the medium sized finetuned and foundation models are orders of magnitude slower.

Focusing specifically on the two finetuned models, it is evident that the naively finetuned model is slightly faster and about as accurate as the multihead finetuned model. The speed difference comes down to the fact that the naive model only contains atom type encodings for the Ti and Si atoms. Due to not being able to find 10000 configurations to replay training with from the original training data, the multihead training included some additional atom types in the training data. Therefore, the model contains more parameters to handle additional atom types which leads to a slightly larger and slower model. As evident from the forces RMSE plot, the finetuned models are likely to perform better than models trained from scratch when trained only on a limited amount of data. The more data included in the training, the higher the likelihood will be to train models from scratch that can learn the underlying physical interactions as well as, or even better than, the finetuned versions. For large data sets, it is often an option worth exploring since, if the accuracy of the models can achieve the same levels as the finetuned models, it might just be possible to train an accurate and small (`max_l_equivariance=0`) model that is faster than models finetuned from foundation models. Due to the lower amount of training data needed and the fact that the nature of the physical interactions, to a large degree, have already been learned by the foundation models, finetuning is however often a more practical path to go to ensure creation of accurate models if performance is not of critical importance.

Since this tutorial is mainly aimed at being an introduction as to how to train MACE models in various ways, no detailed parameter study has been carried out. Similarly, it has not been ensured that models have converged. In order to conduct training with a high probability of convergence to occur, see the **General remarks** section.

Impact of important parameters

The following table summarizes the impact of key parameters on model size, inference time, and accuracy when training from scratch as well as the pros and cons of finetuning.

Parameter / Model setting	Model Size and Memory	Inference Time	Accuracy
Increasing max_l_equivariance	Higher values significantly increase model size.	Increases (can be much slower for higher values).	Generally higher, especially for complex systems. For simple systems, gains may be limited.
Increasing distance_cutoff	Increases the required amount of memory.	Increases exponentially as the local environments include more neighbour atoms.	Improves accuracy for systems with long-range interactions. For short-range systems, smaller cutoffs may suffice.
Increasing number_of_channels	More channels increase model size linearly.	Increases, but less dramatically than max_l_equivariance.	Higher number of channels generally improves accuracy, especially for forces. Diminishing returns for very high values.
Finetuning foundation models	Fixed to model architecture of foundation model used.	Similar to foundation model used.	Typically improves accuracy over training from scratch, especially with limited data.

Based on these considerations, the following general recommendations stand if choosing to train a MACE model from scratch. Note that a requirement for reaching a good model is that sufficient amounts of diverse training data is available.

- `max_l_equivariance`: Typically 0 or 1. Start with 0; optionally increase if accuracy is insufficient and sufficient training data is available.
- `distance_cutoff`: Typically 5–6 Å. Keep within a range of 4-7 Å.
- `number_of_channels`: Typically 64 or 128. Smaller numbers of channels can be tested for creating smaller models where speed-accuracy tradeoffs can be investigated.

There is no guarantee that training from scratch will yield a model that is better than finetuning a foundation model. If the training data is limited or not exhaustive enough, finetuning a foundation model is the better choice. If the training data is large and diverse enough, both approaches can be tested for the given use case. Under the right circumstances, a model trained from scratch might be able to achieve similar accuracy as a finetuned model while being smaller and faster. This is however not guaranteed and depends on the specific training data and application domain.

General remarks

The following remarks are general notes regarding training of MACE models and are not specific to the examples in this tutorial. They are included here for reference.

- Restarting is by default activated in the MACE training. This means that if a training is interrupted, it can be resumed from the last checkpoint. It however has to be resubmitted in the same job folder - this is automatically the case when using the Job Tool and using the resubmit button for a job that has timed out, or when resubmitting the same submission script from the command line. Otherwise the training will start from the beginning again due to the lack of a checkpoints folder.
- For general training of MACE models there is a rough rule of thumb for ensuring the training to be likely to converge. The number of gradient updates - given by `number_of_training_configurations * max_number_of_epochs / batch_size` - should be on the order of 200000 to ensure a good likelihood of the training to converge. For small models, this number can be lower, while for larger models it may be higher but the general rule is good to keep in mind

nonetheless. The exact number differs depending on the model architecture and the specific training but it offers a good starting point when training from scratch. For finetuning, the number of gradient updates required may be lower but as a starting point the rule of thumb can still be followed. For Multihead Finetuning, the replayed training configurations are also included in the number of training configurations for determining the number of gradient updates. Note that in any kind of training, setting the `patience` parameter to a reasonable value (relative to the `max_number_of_epochs`) can cut the training short, if the training seemingly has converged early. This is not required but can be a good practice to avoid unnecessary training time.

- In order to avoid using a lot of disk space and saving a lot of files that will not be used, it may be beneficial to not save all the checkpoints during the training as in the end only the best checkpoint from the training will be used to generate the final model. As has been done in all examples in this tutorial, this is achieved by deactivating the keep checkpoints parameter: `keep_checkpoints=False`.

- As with any kind of machine learning model, during experimentation several train-validation or, if preferred, train-validation-test splits should be screened to ensure that the model is not overfitting to the training data and that the data splitting hyperparameters (`random_seed` - both in *MACEFittingParameters* and in *MachineLearnedForceField-Trainer* if a test set is created) are set meaningfully and optimized. The random seed in the *MACEFittingParameters* also controls the random initialization of the model weights, which can have a significant impact on the final model performance. Therefore, when training from scratch, it is recommended to run the training several times with different random seeds to ensure that different initial weights are attempted as part of the model training process.

- The most important hyperparameters to screen in general have all been introduced during this tutorial. For delicate applications other hyperparameters, as documented in the *MACEFittingParameters*, can also be tuned. However, this is mainly recommended for expert-users, as it requires more profound experience.

Loading custom MACE models into QuantumATK

As part of the `validation scripts`, it has already been shown how to load a custom MACE model into QuantumATK in scripting. Custom MACE models can also be loaded from within the Workflow Builder in Nanolab to generate full workflows. Continuing with the multihead_finetuned model trained above, a TiSi configuration is loaded from a database as a BulkConfiguration.

The Set MachineLearnedForceFieldCalculator block is then selected. In the `Potential set type` dropdown menu, `MACE_potential` is selected. This updates the widget menu specifically to custom MACE models. In the `MACE filename` field, the file with the model can be selected by clicking the `...` symbol. Note that the file to choose is the file with the `.qatkpt` extension. After choosing the model file ensure to click the blue arrow in the left part of the filename field to ensure that the file has been found and is compatible. If suitable for the desired simulation, the `Float precision` can be changed to 64-bit, which is recommended, e.g. for accurate optimization, dynamical matrix calculations, NEB, or transition state optimization. This example will only set up a molecular dynamics simulation and the value is kept at 32-bit. After this, the calculator is ready to use and the widget can be closed again.

A simple MolecularDynamics block can be set up to complete the workflow. Note that the script can run on either CPU alone or also include GPU. This is automatically handled at runtime, as described in the *technical notes*, depending on available resources and licenses. If the license configuration supports it, running with GPU enabled is recommended for MACE models to ensure efficient inference.

The Workflow outlined above is available for download: `Trained_model_usage_example_with_MD.hdf5`. The workflow can be added manually to the Workflows folder within the current project and thereafter opened in the workflow menu in QuantumATK. In order to remove the error shown in the Set MachineLearnedForceFieldCalculator block, the path to the custom MACE model must be updated in the `MACE filename` field. Alternatively, the workflow can be acquired and set up in a separate QuantumATK project by downloading all the mentioned project files in a zip folder, unpacking it, and opening the folder as a project in QuantumATK with all trained models included. The zip folder can be downloaded here: `Tutorial_MACE_amorphous_and_crystal_structures_all_data.zip`

Summary

In this tutorial, it has been shown how to train MACE models from scratch, finetune foundation models, and how validating trained models on unseen data can be approached. It has also shown how to load custom MACE models into QuantumATK and use them in workflows and scripts. The tutorial has provided a brief overview of the most influential MACE training parameters and how they affect the model size, inference time, and accuracy. Furthermore, strategies for how to approach training new models with custom data as well as things that should be taken into consideration when training models for custom applications have been discussed.

Compared to MTP, MACE models can provide an even better accuracy, especially for very complex systems with many atoms, at the expense of slower simulation speed.

References

3.1.2 2D Database and potentials

Version: X-2025.06

In this tutorial, you will learn how to use structures from the 2D materials database integrated in QuantumATK Nanolab and how to couple them with corresponding potential sets made specifically for the structures in the database. Here, it will be exemplified with a SiO 2D-structure and a corresponding Stillinger-Weber potential[1] for which its phonon bandstructure is determined.

Importing a Structure from the 2D Materials Database

The following step-by-step protocol describes how to import a SiO 2D-structure from NanoLabs Internal Database.

1. In the **Builder** *Stash* (1), click on the *Add* button (2) and choose *From Database* (3).

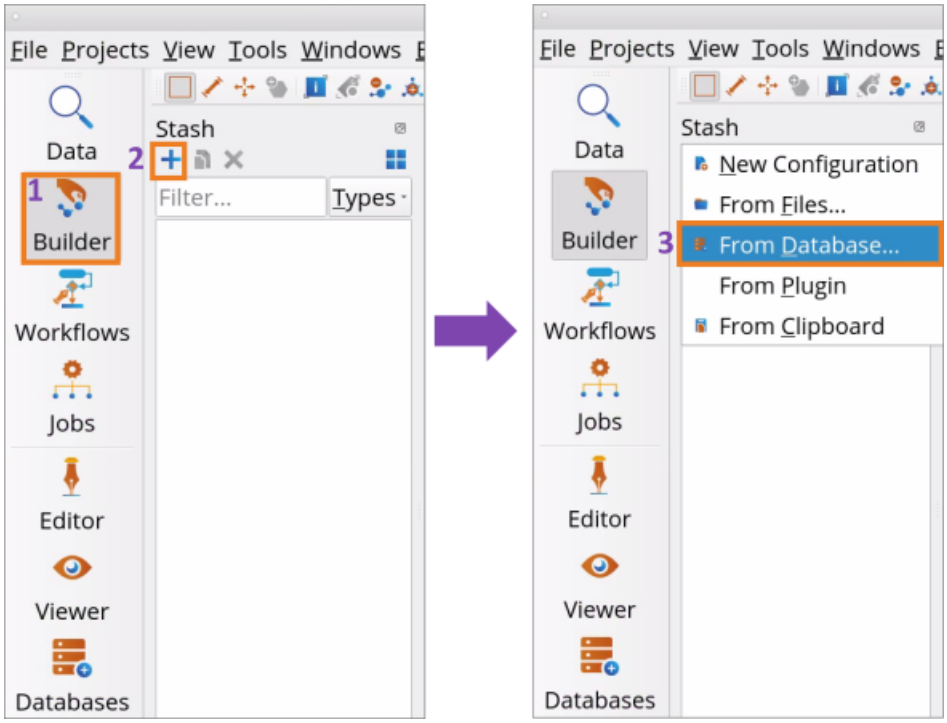
2. Navigate to the *2D materials* Database tab and use the *Search* bar (1) to find the SiO (buckled) structure (2). Then click on the *Add* button (3) to add the structure to the **Builder** *Stash*.

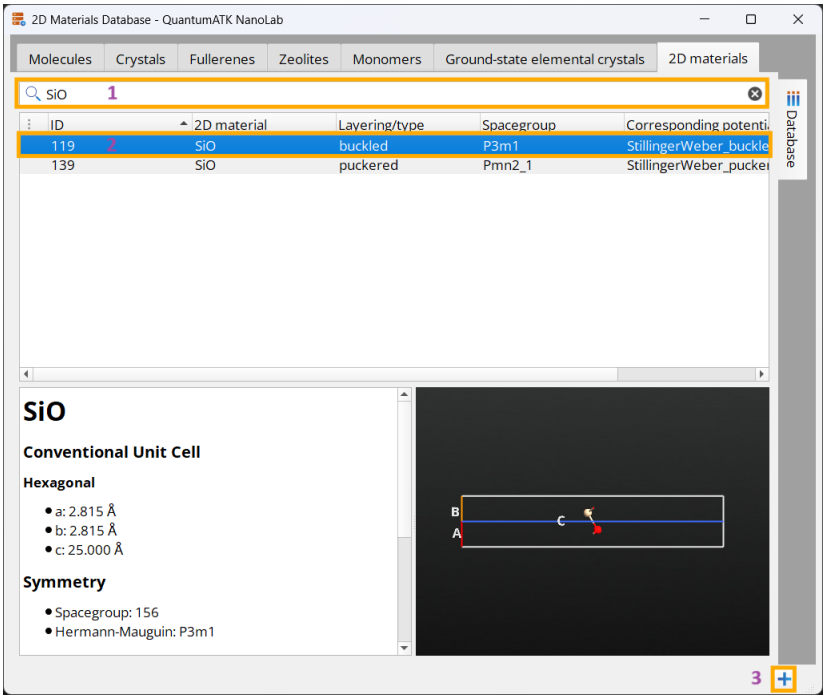
Creating and Setting a Calculator from a 2D Potential Set

In order to set up a forcefield calculator from a potential set corresponding to the structure, the following steps are taken.

1. In the **Builder**, use the *Send content to other tools* button to send the imported buckled SiO structure to the **Workflow Builder**.

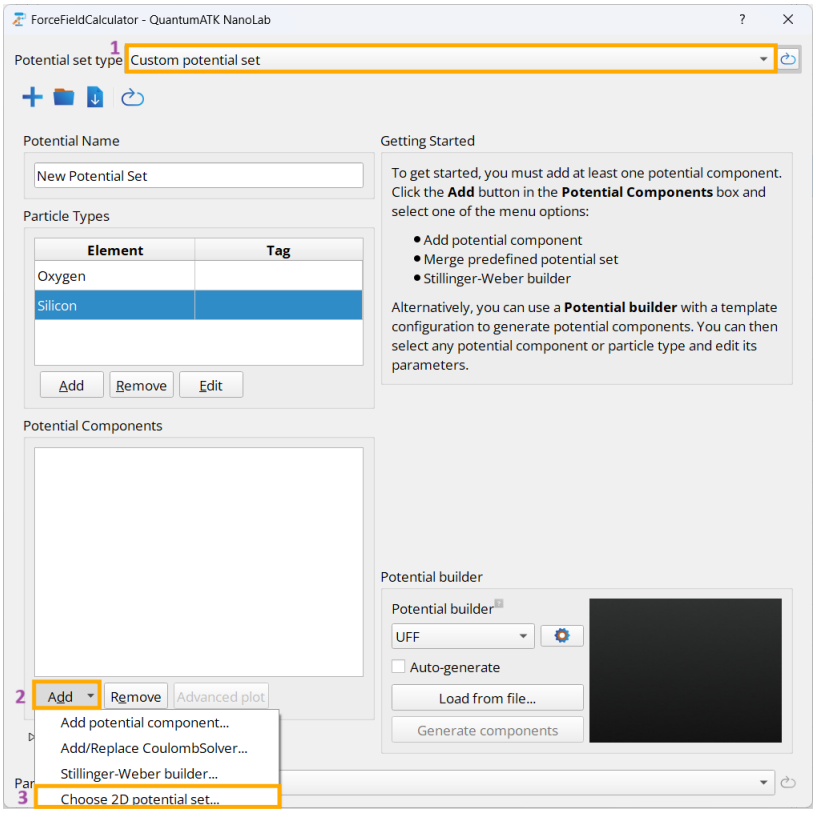
[1] Jin-Wu Jiang and Yu-Ping Zhou. Parameterization of Stillinger-Weber potential for two-dimensional atomic crystals. *arXiv preprint arXiv:1704.03147*, 2017.


File Projects View Tools Windows
Data
Builder
Workflows
Jobs
Editor
Viewer
Databases
1
2
3
Stash
Filter...
Types
New Configuration
From Files...
From Database...
From Plugin
From Clipboard


2D Materials Database - QuantumATK NanoLab
Molecules
Crystals
Fullerenes
Zeolites
Monomers
Ground-state elemental crystals
2D materials
SiO
1
ID
2D material
Layering/type
Spacegroup
Corresponding potenti
119
SiO
buckled
P3m1
StillingerWeber_buckle
139
SiO
puckered
Pmn2_1
StillingerWeber_pucker
Database
SiO
Conventional Unit Cell
Hexagonal
a: 2.815 Å
b: 2.815 Å
c: 25.000 Å
Symmetry
Spacegroup: 156
Hermann-Mauguin: P3m1
B
A
C
3

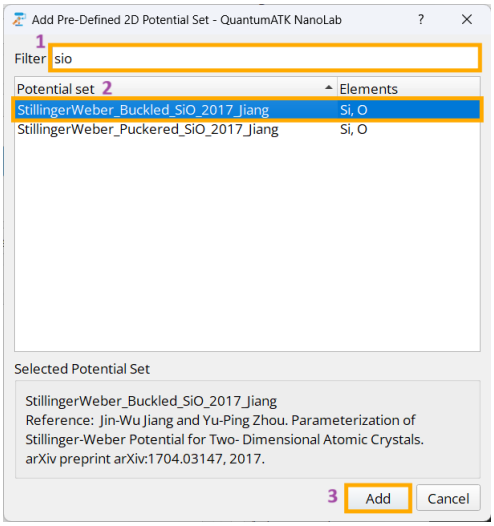
2.Go to the *Block Catalog* → *QuantumATK* tab → *Calculators* category on the right hand panel and drag-and-drop the *SetForceFieldCalculator* block to the middle *Workflow Area* → *Build* tab.

3.Double-click on the *SetForceFieldCalculator* block and change the *Potential set type* from *Pre-defined potential set* to *Custom potential set* (1). In the *Custom potential set* menu, click the *Add* button under *Potential Components* (2) and click the *Choose 2D potential set* option (3) to open a menu for choosing a predefined 2D potential set.



4.Use the *Filter* bar (1) to search for the Stillinger-Weber potential set corresponding to the buckled SiO structure. Marking the potential set (2) and clicking on the *Add* button (3) adds the potential set to the calculator of the SetForceFieldCalculator block.

Note: Most 2D potential sets require specific tags to be present on the structure. These tags are automatically attached on the corresponding structure in the 2D materials database. If a different or a custom structure is used, where one or more tags are missing on the structure, the forcefield calculator block will throw an error highlighting the tag requirement that is not fulfilled. It is possible to circumvent this error by clicking the checkbox field *Ignore 2D tags error* that will appear in the *Custom potential set* menu. This allows the potential set to be scripted for visual inspection. It should be noted that the script will still not be able to complete any calculations with this setting selected, as long as any tag remain missing.



Phonon Bandstructure calculation

Now that a calculator based on the appropriate potential set has been attached to the 2D structure, any desired calculation can follow. In this example workflow, we will calculate the phonon bandstructure. This usually requires a geometry optimization with a tight tolerance for accurate results. The forcefield-structure pair is however already optimized to within an even tighter tolerance than we will use here. Nevertheless, an optimzation step is included for completeness.

1. In the **Workflow Builder**, add an *Optimization and Dynamics* → *OptimizeGeometry* block. Set the *Force tolerance* to 0.001 eV/Å in the menu.

2. Add an *Analysis* → *PhononBandstructure* block. This will automatically add a required *DynamicalMatrix* input block. These blocks will not be discussed in more detail here - see *DynamicalMatrix* and *PhononBandstructure* documentation or related tutorials for more information.

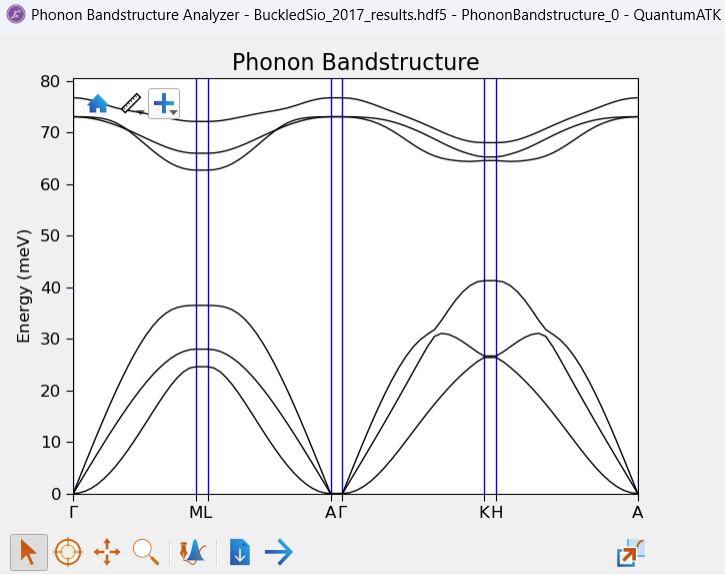
Analyzing Results

Once the calculation has finished, the results can be analyzed.

1. Go to the **Data View** tool.

2. Since the optimization converged after the first step, no additional optimization steps were taken. This can be confirmed by inspecting the original and optimized lattice constants in text representation. In *Data Sources* select the file `BuckledSio_2017_results.hdf5` (or equivalent if a different destination filename has been used) and then in *Data View* window select both *BulkConfiguration_0* (initial geometry) and *optgeom* (optimized geometry), followed by a right-click to *Open All with Text Representation*. As expected we find the that lattice constants of the 2D buckled SiO remained at values of a=2.815 Å and c=25.000 Å.

3. Right-click on the phonon bandstructure *PhononBandstructure_0* in *Data View* and choose to *Open with Phonon Bandstructure Analyzer*. This will display the phonon bandstructure for the simple 2D unit cell.



References

3.1.3 Moment Tensor Potential (MTP) Training for Crystal and Amorphous Structures

Version: W-2024.09-SP1

Downloads

c-am-TiSi-MTP-training.hdf5 MTP_TiSi_basis400_final.mtp c-TiSi-DFT-EOS.hdf5 c-TiSi-MTP-EOS.hdf5 am-TiSi-MTP-validation.hdf5 c-am-TiSi-MTP-training-validation-results.zip

In this introductory tutorial, you will learn how to use QuantumATK to

- Set-up a workflow for training a machine-learned force field (ML FF), MTP, for crystal and amorphous TiSi and $TiSi_2$ structures, which can be used for molecular dynamics (MD), geometry optimization, and other force-field-based simulations.
- Analyze the MTP training results.
- Validate the quality of the trained MTP.

Unlike many empirical potentials with physical functional forms, ML FFs such as MTPs are not highly transferable and must be trained with the target application in mind. For example, an MTP trained for bulk structures will not describe accurately free surfaces or nanoparticles of that material without the addition of such structures to the training set. However, the outlined training protocol can be easily adapted or extended to train an accurate MTP for different geometries, stoichiometries, element composition and other materials by explicitly including relevant training data for such structures.

Prerequisites

•Tutorial *Geometry Optimization*

•Tutorial *Molecular Dynamics Simulations for Generating Amorphous Structures*

Overview of the MTP Training Workflow

ML FFs are trained to predict *ab initio*, i.e., DFT, potential energy surfaces (PES) and vastly improve the accuracy of classical dynamics simulations while keeping the computational cost at the level of empirical force fields. To train a ML FF, one first needs to calculate training data (energy, forces, stress) for generated training configurations (typically, of relatively small sizes) with the reference DFT calculator and then fit an MTP to that DFT training data. Simulations with trained ML FFs can reach the high accuracy, long simulation times and large system sizes that capture the complexity of realistic systems, e.g., amorphous and alloy materials, and processes, e.g., diffusion, crystallization or deposition.

QuantumATK implemented Moment Tensor Potentials (MTPs)[1] as they provide high performance for a given accuracy when compared to other ML FFs models in the literature[2]. MTP advantages include:

•Very fast and supports large scale simulations.

•Allows training to multi-element-materials.

•Systematically improvable.

Tip:

1.Since the MTP is trained to match the DFT reference calculator results, the accuracy of the final MTP is tied to this choice. Thus, it is important to choose the reference calculator that gives a good description of the system and application you want to study.

2.Learn more about ML FFs and their benefits here: Glossary page on What are Machine-Learned Force Fields

---

MTP training for crystal and amorphous structures in QuantumATK is structured into three stages:

1.**Crystal Training Data** - generates crystal training configurations with random displacements and calculates training data (energy, forces, stress) with the reference calculator DFT. Crystal training data forms the bulk of the training set, giving a fundamental description of the material in question. It will also be used to fit an initial MTP in the 2nd MTP training stage.

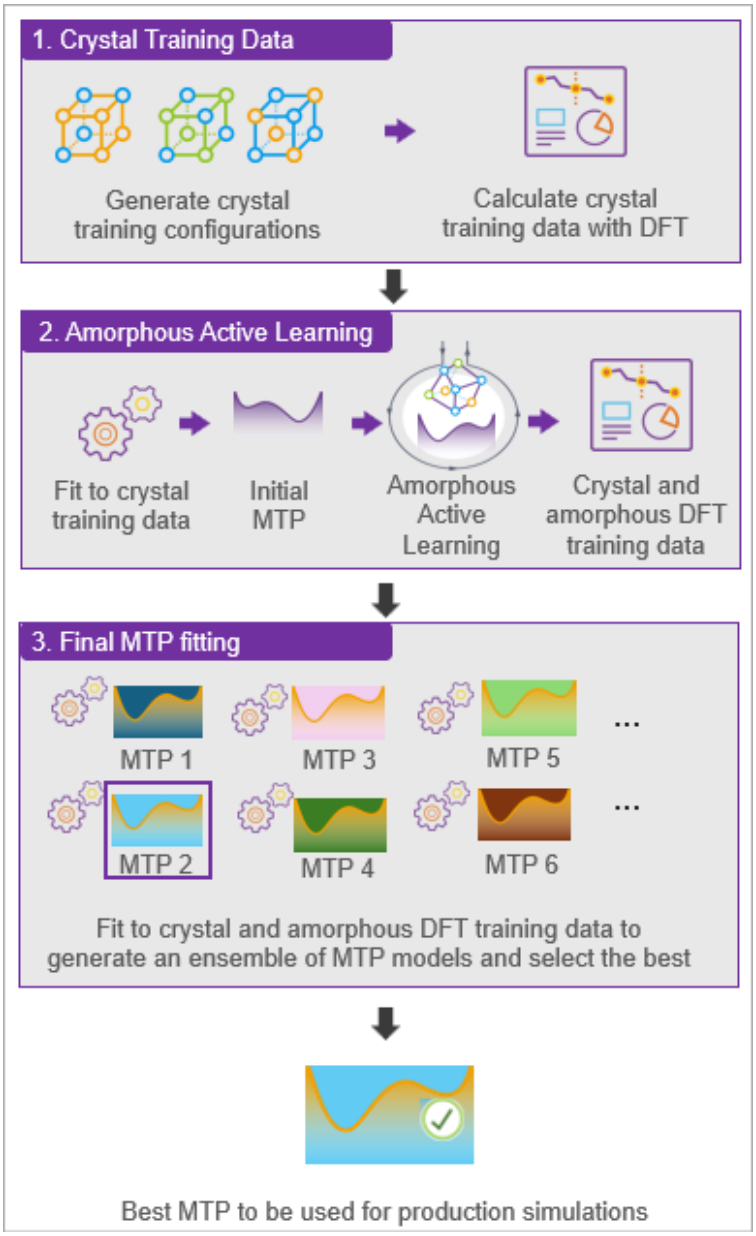
2.**Amorphous Active Learning** - fits an initial MTP to the crystal training data, which is then used to run iterative Active learning geometry optimization and MD simulations to identify relevant configurations which are not represented in the initial training set. This workflow actively adds extrapolating amorphous configurations, incl. DFT data, to the training data set. The combined amorphous + crystal training data is then used in the 3rd stage to fit an ensemble of MTP models.

3.**Final MTP Fitting** - generates an ensemble of MTP models and selects the best fitted MTP that gives lowest and comparable training and testing errors with respect to the reference DFT method.

[1] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

[2] Yunxing Zuo, Chi Chen, Xiangguo Li, Zhi Deng, Yiming Chen, Jörg Behler, Gábor Csányi, AlexanderăV. Shapeev, AidanăP. Thompson, MitchellăA. Wood, and ShyueăPing Ong. Performance and Cost Assessment of Machine Learning Interatomic Potentials. *The Journal of Physical Chemistry A*, 124(4):731–745, 2020. URL: https://doi.org/10.1021/acs.jpca.9b08723, arXiv:https://doi.org/10.1021/acs.jpca.9b08723, doi:10.1021/acs.jpca.9b08723.

The chosen MTP can then be used in production simulations, such as MD, geometry optimization, and other force-field-based simulations.



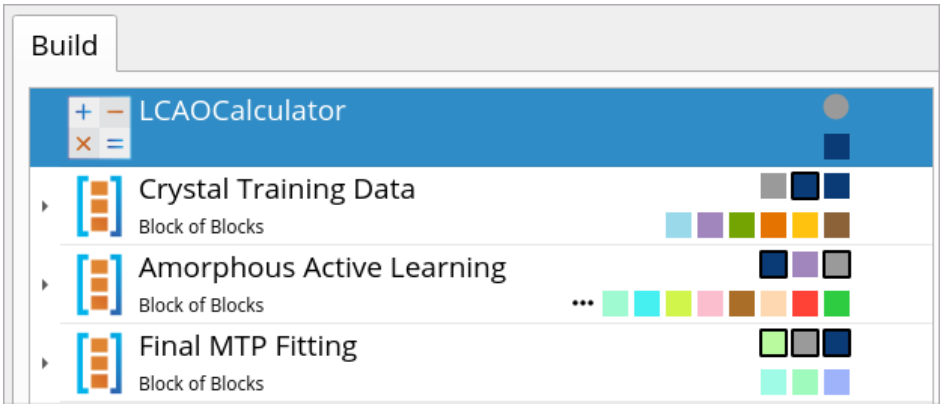
1.Download the ready-to-use MTP training workflow `c-am-TiSi-MTP-training.hdf5` and open it in the **Workflow Builder**.

2.The workflow consists of three aforementioned stages (each contained in a so-called *Blocks of Blocks*) and is applicable to any crystal and amorphous material:

1.**Crystal Training Data**

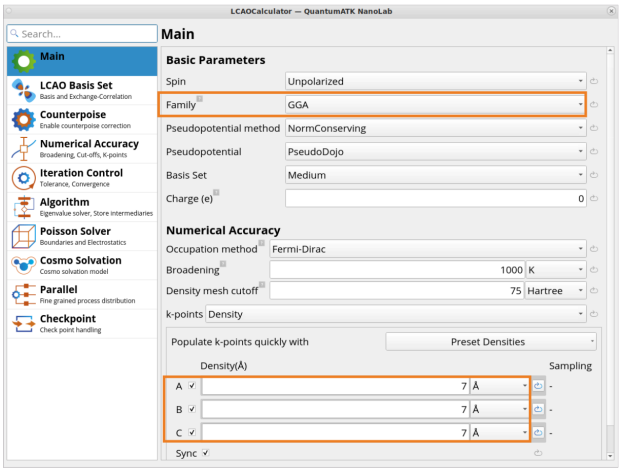
2.**Amorphous Active Learning**

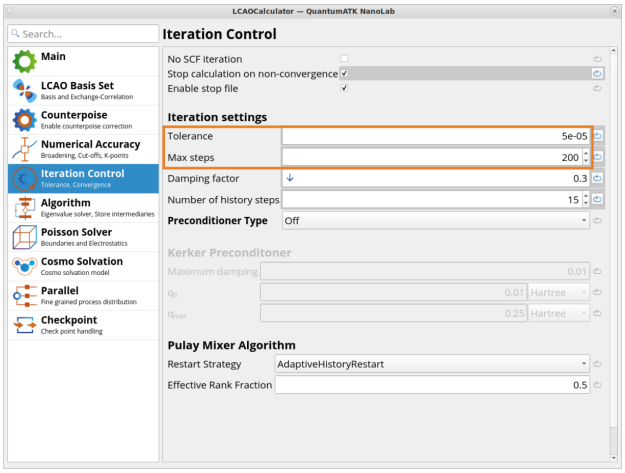
3.**Final MTP Fitting**



3.The **LCAOCalculator** block placed above the three *Blocks of Blocks* is used for defining a reference DFT-LCAO calculator (using the PBE xc-functional) that is used to calculate training data (energy, forces and stress) throughout the MTP training simulations.

- Under the *Main* tab, the *k-point density* is increased from the default *medium* to *high*, i.e., from *4Å x 4Å x 4Å* to *7Å x 7Å x 7Å*, to reduce the impact of sampling discontinuities when slightly changing the cell size during random displacements of the structure.
- Under the *Iteration Control* tab, the SCF *Tolerance* is decreased from the default *1e-04* to *5e-05* to reduce the noise in energy, forces, stress, and the *Max steps* for SCF is increased from default *100* to *200* so that SCF also converges even if the configuration needs more steps due to the tighter tolerance.
- Under the *Algorithm* tab → *Parameters*, the default *Optimize diagonalization for memory* is switched to *Optimize diagonalization for speed*.

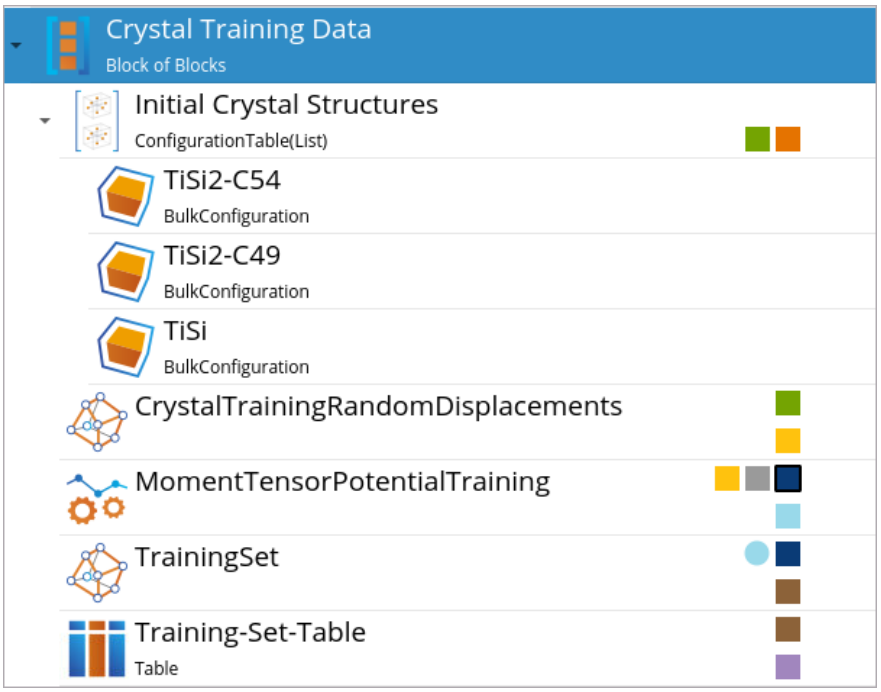




We will now go through the three stages step-by-step, explain the settings, and provide tips and recommendations.

Crystal Training Data

We will begin with the 1st stage of the MTP training workflow - **Crystal Training Data**. This includes generation of crystal training configurations, i.e., repeated and randomly displaced crystal structures of different phases of titanium silicide using the *crystalTrainingRandomDisplacements* protocol, and calculation of training data with DFT for these displaced configurations.

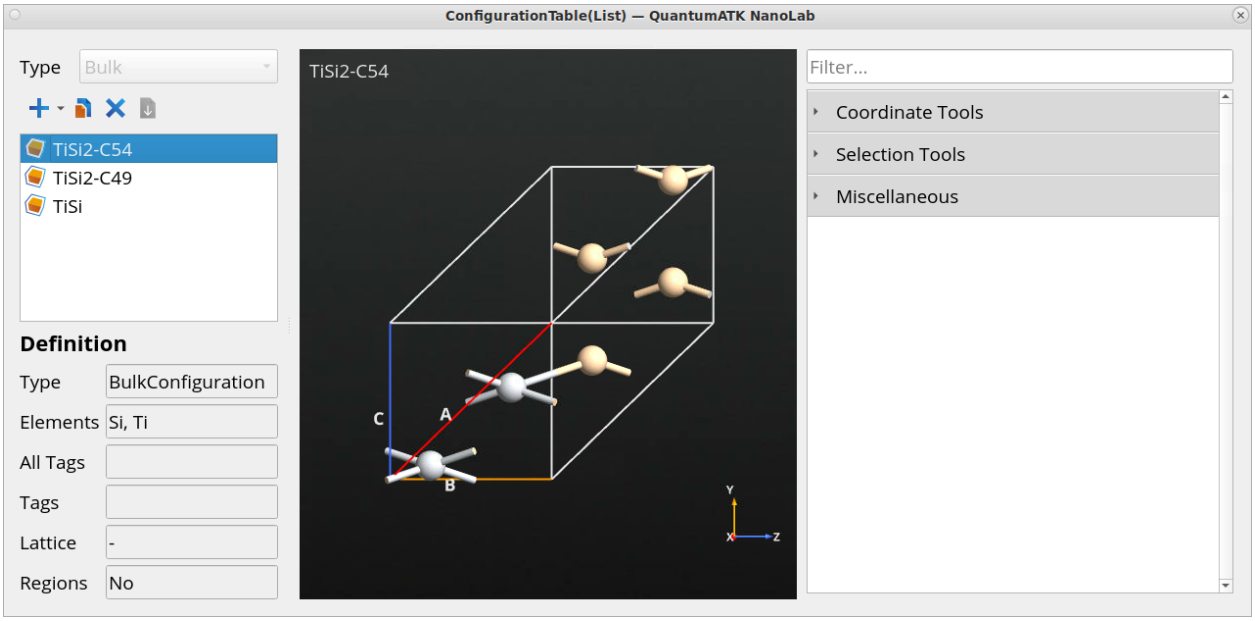


1. The **Initial Crystal Structures** *ConfigurationTable(List)* block contains three crystal structures of titanium silicide, with two different stoichiometries, which we will include in the MTP training: TiSi, $TiSi_2$-C54 and $TiSi_2$-C49. These structures will be used for generating randomly displaced structures - the training configurations.

Note: The required input for the **Initial Crystal Structures** depends on the application of the final trained MTP. If you intend to use the trained MTP for simulating other stoichiometries and crystal phases than TiSi, $TiSi_2$-C54 and $TiSi_2$-C49, then you should include structures with these other stoichiometries and crystal phases as well, to ensure accurate simulations for those materials.

Tip:

1. Crystal structures can be imported from the *NanoLabs Internal Database* in the **Builder** or from the *Materials Project* database in the **Databases**.
2. It is recommended to optimize the geometry of crystal structures with the reference (DFT) calculator before using them for the MTP training.



2. The **CrystalTrainingRandomDisplacements** block sets parameters for randomly displacing configurations around the equilibrium of the provided crystal structures of titanium silicide.

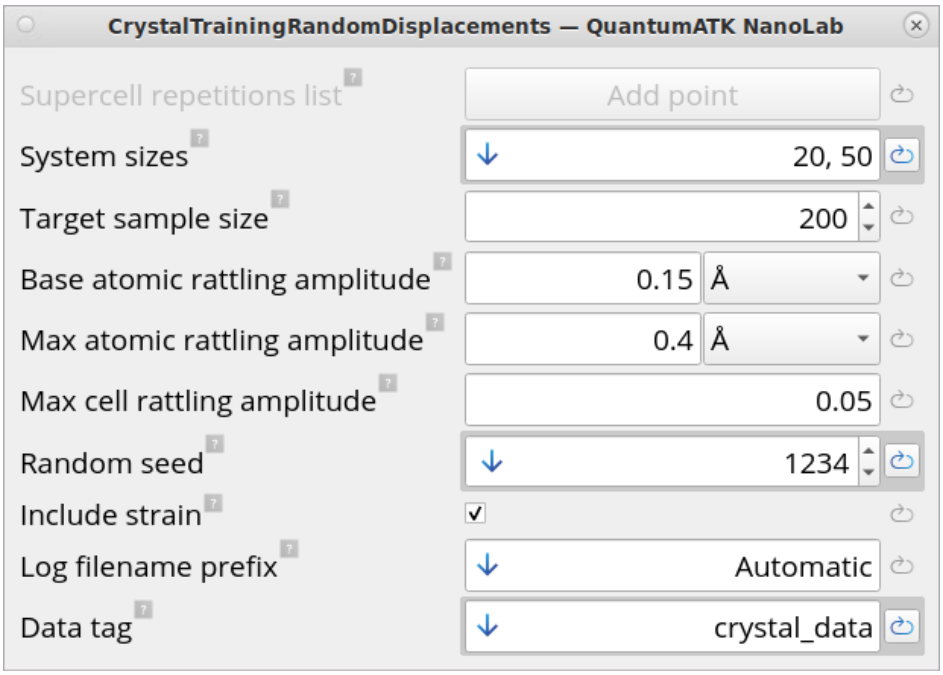
- *System sizes:* Define the target sizes for the supercells in which the atoms will be displaced. It is recommended to use at least 2 different sizes (in this case, *20* and *50*), so that the MTP model can better learn the system-size dependence of the total energy.
- *Target sample size:* Set the minimum for how many displaced configurations should be generated. Note that depending on various input choices, the final number of training configurations can be significantly higher. Here, we choose *200* (default).
- *Base / max atomic rattling amplitudes:* Define by what magnitude the atoms are displaced. Should be chosen so most displaced configurations have max. atomic forces below 5 eV / Ang, and that the max. atomic forces peaks are between 20 – 40 eV / Ang. The default values (*0.15 Å/ 0.4 Å*) work well for most materials, but for pure metals larger values can be chosen (e.g. *0.25 Å / 0.55 Å*).
- *Include strain:* Should be enabled (by default), unless not needed for a specific application.

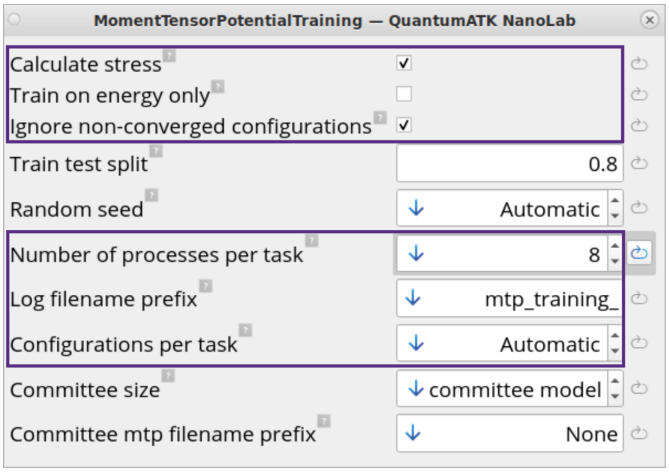
3. The **MomentTensorPotentialTraining** block sets parameters for calculating DFT training data (energy, forces, stress) for the randomly displaced configurations.

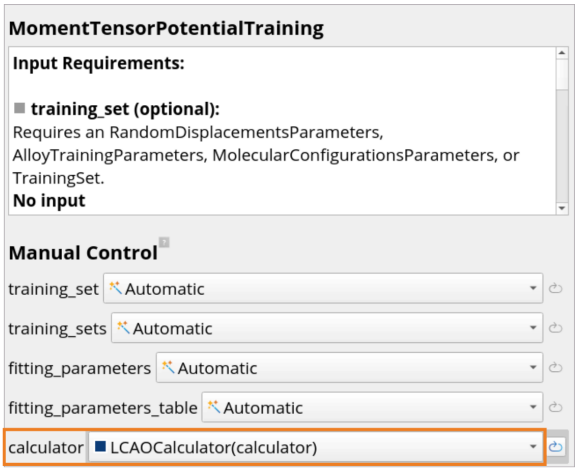
- Only the parameters in purple boxes, i.e., related to the calculation of the DFT training data, are used. The remaining parameters are related to the MTP fitting, which we do not carry out in this stage.
- *Number of processes per task*: Specify how many MPI processes should be used for each DFT calculation (default - *all available processes*). Since the DFT calculations are mostly independent from each other, multiple DFT calculations can be done at the same time. For example, when submitting a calculation on 40 MPI processes, number of processes per task could be *8*, thus resulting in 5 DFT calculations running at the same time. This can speed up the entire calculation, but will use more memory.
- *LCAOCalculator* and *MomentTensorPotentialTraining* blocks must be connected, so that the defined *LCAOCalculator* is used to calculate training data.

4. The **TrainingSet** block exports the generated training data with DFT as a *TrainingSet* object.

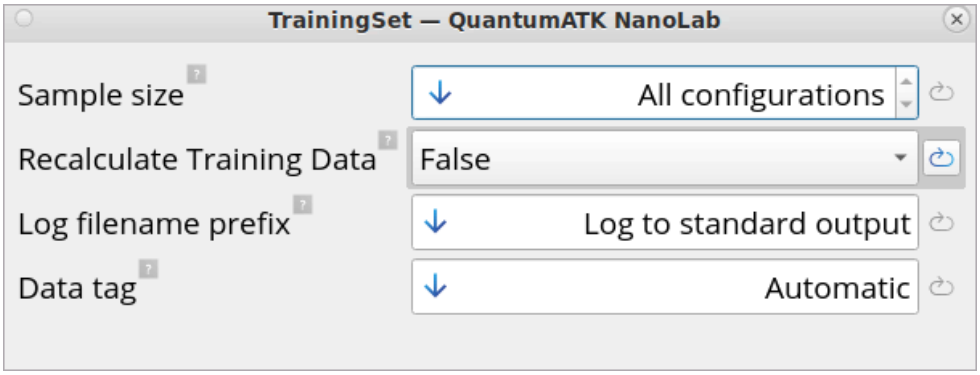
- *Sample size*: Keep this at *All configurations* (default) to include all configurations from the *CrystalTrainingRandomDisplacements* block in the dataset.


CrystalTrainingRandomDisplacements — QuantumATK NanoLab
Supercell repetitions list
Add point
System sizes
20, 50
Target sample size
200
Base atomic rattling amplitude
0.15
Å
Max atomic rattling amplitude
0.4
Å
Max cell rattling amplitude
0.05
Random seed
1234
Include strain
Log filename prefix
Automatic
Data tag
crystal_data


MomentTensorPotentialTraining — QuantumATK NanoLab
Calculate stress
Train on energy only
Ignore non-converged configurations
Train test split
0.8
Random seed
Automatic
Number of processes per task
8
Log filename prefix
mtp_training_
Configurations per task
Automatic
Committee size
committee model
Committee mtp filename prefix
None


MomentTensorPotentialTraining
Input Requirements:
training_set (optional):
Requires an RandomDisplacementsParameters, AlloyTrainingParameters, MolecularConfigurationsParameters, or TrainingSet.
No input
Manual Control
training_set
Automatic
training_sets
Automatic
fitting_parameters
Automatic
fitting_parameters_table
Automatic
calculator
LCAOCalculator(calculator)

•*Recalculate Training Data*: Changed to *False*, because the DFT data has already been calculated. Only if the same configurations are used with a different reference calculator, should this parameter be set to *True*.



Note: In case of training an MTP model for the use of crystal applications only, one could take the *TrainingSet* generated above and directly move to the 3rd stage - **Final MTP Fitting**. The resulting MTP would be able to describe the trained crystal phases up to medium temperatures (i.e., well below the melting point where the crystal is still stable). Crystal properties, such as lattice constants, elastic moduli, or phonons, are typically described very well using this training data.

Amorphous Active Learning

In the 1st stage of the MTP training we generated crystal training data for the initial MTP fitting. In the 2nd stage, **Amorphous Active Learning**, we will include training data for amorphous configurations. Here, it is not so straightforward to identify all relevant amorphous configurations beforehand, thus we will employ the *ActiveLearningSimulation* approach for on-the-fly improvement of training dataset by actively adding extrapolating amorphous training configurations from dynamic simulations, such as geometry optimization and melt-quench MD.

Note:

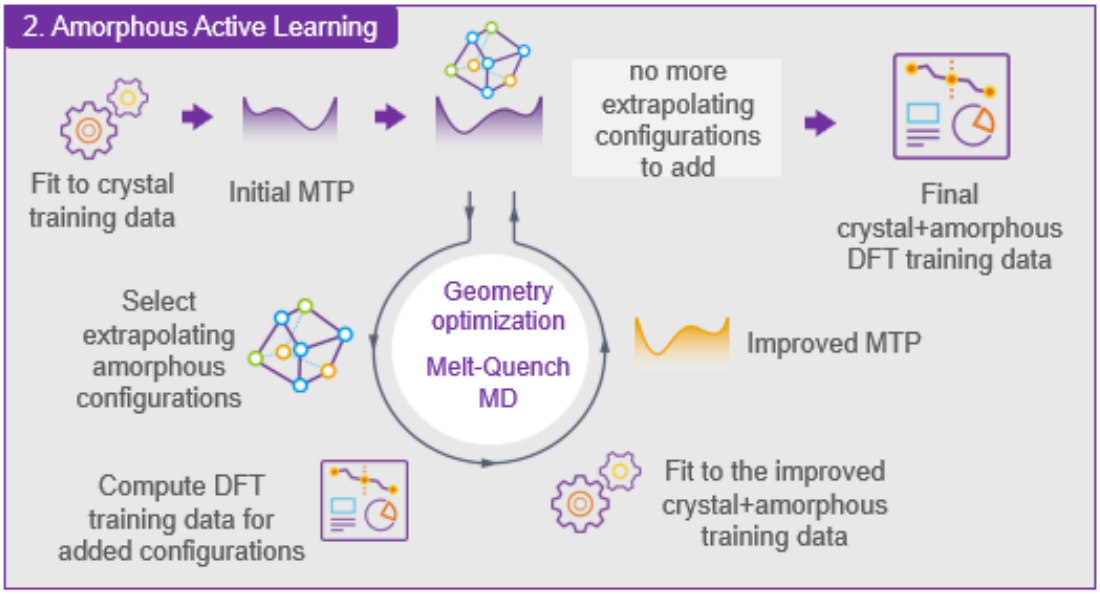
1. Active learning is a strategy for iteratively finding and selecting new atomic configurations for which the MTP has the highest uncertainty, i.e., highest prediction errors (so-called *extrapolating configurations*). These configurations are most likely to improve the MTPs accuracy when added to the training data set. Since Active learning targets specific configuration spaces instead of randomly sampling configurations, it reduces the need for expensive DFT training data calculations, which leads to efficient and fast training.
2. It is recommended to include Active learning when training an MTP for amorphous systems, interfaces, surface processes, and systems at high temperatures.
3. In addition to geometry optimization and MD, methods such as NEB, AKMC and TFMC can also detect extrapolating configurations and add them to the training data set.

During **Amorphous Active learning**, first, one fits an initial MTP to the crystal training data and then iteratively repeats the flow below until there are no more extrapolating configurations to add, as also shown in the figure below:

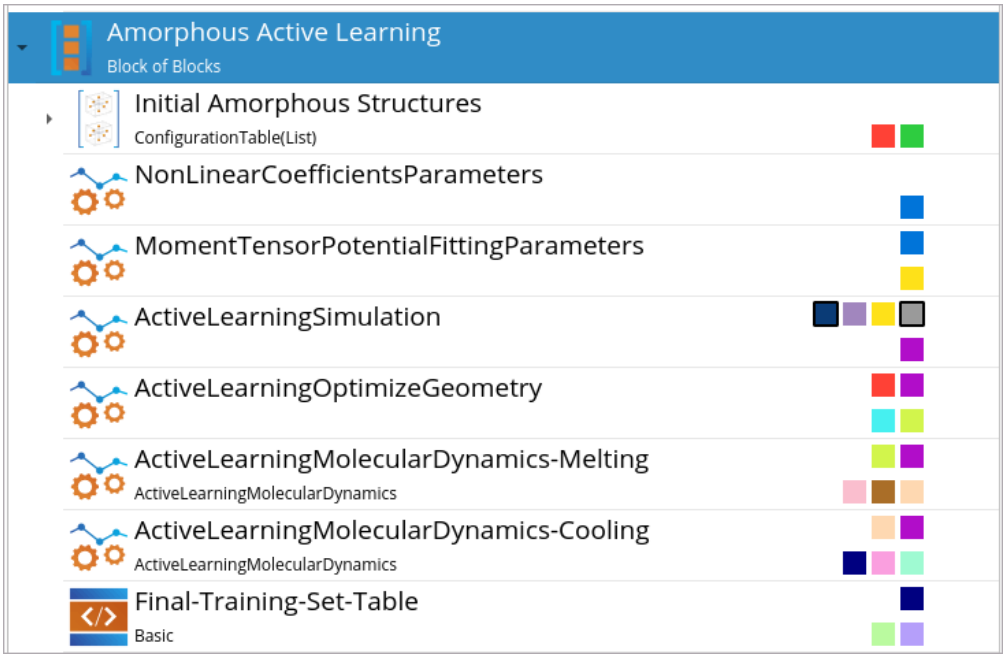
•Perform geometry optimization and MD melt-quench (melting/cooling) simulations with the current MTP to find additional extrapolating amorphous training configurations.

•Calculate DFT training data for these added training configurations.

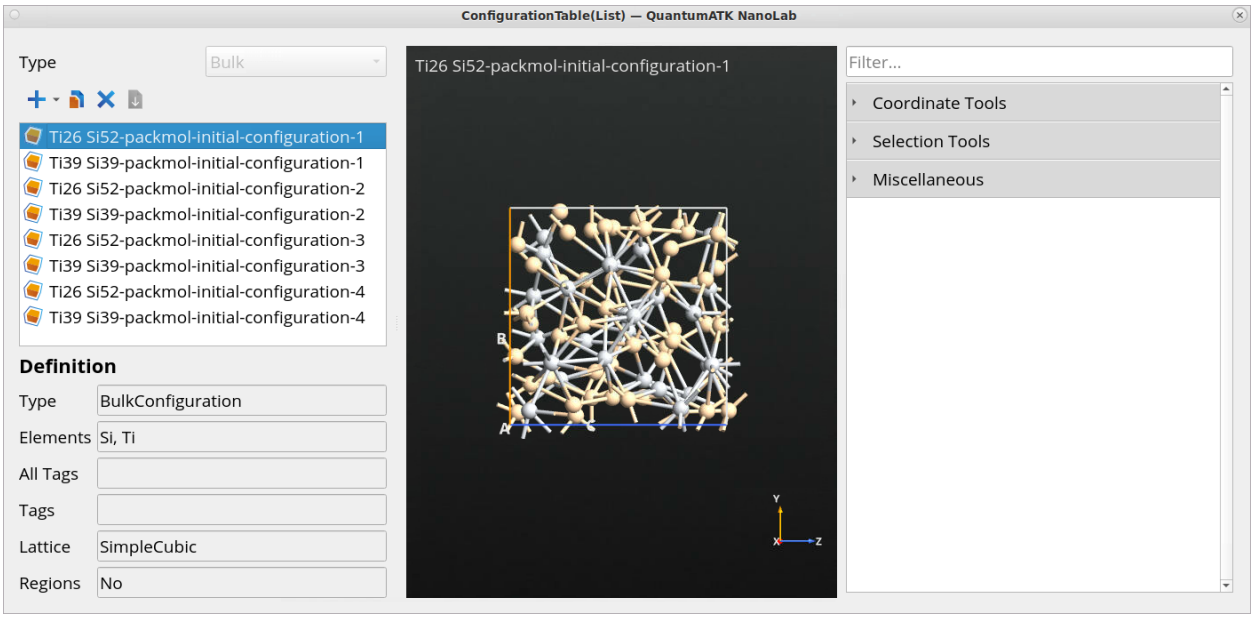
•Fit a preliminary MTP to the improved training data.



The actual workflow for the **Amorphous Active Learning** stage in the **Workflow Builder** looks like this:



1. The **Initial Amorphous Structures** *ConfigurationTable(List)* block contains initial amorphous structures of titanium silicide, where each of them will be used as a starting point for *Active learningGeometryOptimization* and *Active-LearningMolecularDynamics* simulations. Since we have eight initial amorphous $Ti_{39}Si_{39}$ and $Ti_{26}Si_{52}$ structures, QuantumATK will perform eight Active learning geometry optimization and eight subsequent Active learning MD melt-quench simulations.



Tip: Initial amorphous configurations can be generated using *Packmol* in the **Builder** as showcased in the tutorial *Molecular Dynamics Simulations for Generating Amorphous Structures* or can be generated on-the-fly using the *AmorphousPrebuilder* block in the **Workflow Builder**.

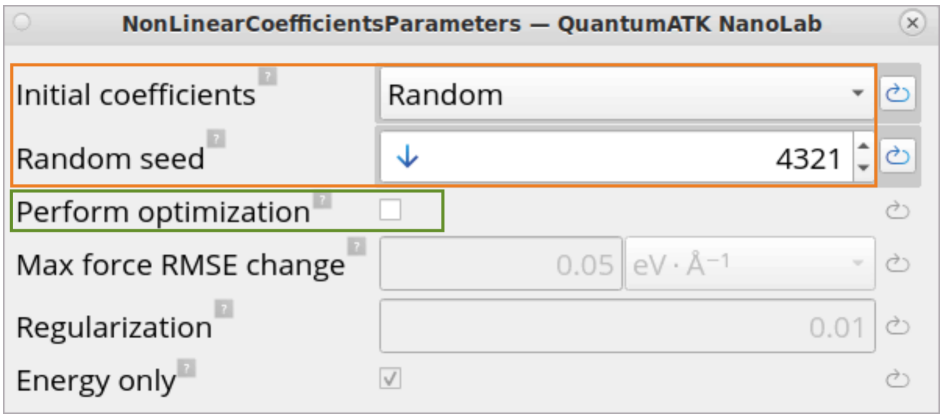
Now lets set up the generation of MTP fitting parameters that will be fitted to the training set data by minimizing the error (energy, forces and stresses) with respect to the reference DFT calculator.

2. The **NonLinearCoefficientsParameters** block contains settings for randomly generating non-linear coefficients for the radial functions for each element pair (see *NonLinearCoefficientsParameters* for more info), that will be used in the next step - calculation of MTP fitting parameters (see *MomentTensorPotentialFittingParameters* for more info).

- *Initial coefficients*: Set to *Random* to use random values for initial non-linear coefficients.
- *Random seed*: Set to an arbitrary number, but use the same number for different Active learning runs. This way consistent results are achieved between different Active learning runs.
- *Perform optimization*: recommend to keep it unchecked in the *Amorphous Active Learning* stage as non-linear MTP training (optimizing non-linear coefficients) usually does not give much improvement in accuracy and it will increase the calculation time.

3. The **MomentTensorPotentialFittingParameters** block contains settings for MTP fitting using the non-linear coefficients generated in the previous step.

- *Basis size*: Use *400* basis function in this example. Generally, it is recommended to use either *Predefined Basis Small* (default), or between *300 – 800* basis functions. More basis functions usually provide a better accuracy (lower training error), but the simulation can become less stable, especially in out-of-equilibrium situations, e.g., at high temperatures or energies.
- *Outer cutoff radii*: Set to *4.5 Ang* in this example. This value specifies the range of the atomic environments. Typically values between *4-5 Ang* are recommended. Similar to basis size, larger values can increase accuracy, whereas smaller values can provide more stability.
- *MTP filename*: One should always specify a filename with `.mtp` extension. This is where the trained MTP parameters are stored.



•*Forces cap*: Specify this value in order to exclude configurations with very large forces, i.e., above the *Force cap*, from the training. For Active learning it is always recommended to set this to a relatively large value, e.g. *500 eV / Ang*, to include closer atom distances with larger forces in order to make sure the repulsive behavior is predicted correctly in such situations. Increasing the *Forces cap* (within some limit) makes the simulation more stable, but less accurate. Thus, while setting the *Forces cap*, one needs to balance between simulation stability and accuracy.

Note: Within the MTP framework, the total energy of a configuration is computed as the sum of energy contributions from atomic environments. Atomic environments are defined as the representation of the immediate chemical environment of each atom in the system within a cutoff radius. Moment tensors include the radial, angular, and many-body distribution of atoms within a cutoff sphere. Their contractions into scalar basis functions provide an invariant representation of atomic environments (structure) as input for MTP training[Page 305, 1].

Now lets set up Active learning simulation parameters.

4. The **ActiveLearningSimulation** block contains settings for running Active learning simulations. MTP fitting parameters from the previous step and the initial training data from the crystal training are used as an input.

•*Candidate/ Retrain Threshold*: These are numerical measures governing how much configurations can differ from the training set before they are considered to be extrapolating. If the current configuration has an extrapolation grade higher than the candidate threshold (default *1*), it will be kept as a possible candidate to be included in the next training. If the value is larger than the retrain threshold (default *3*), the current simulation will be stopped, extrapolating configurations will be added to the training data set, and then Active learning will be re-started now using an improved MTP fitted to the extended training data set.

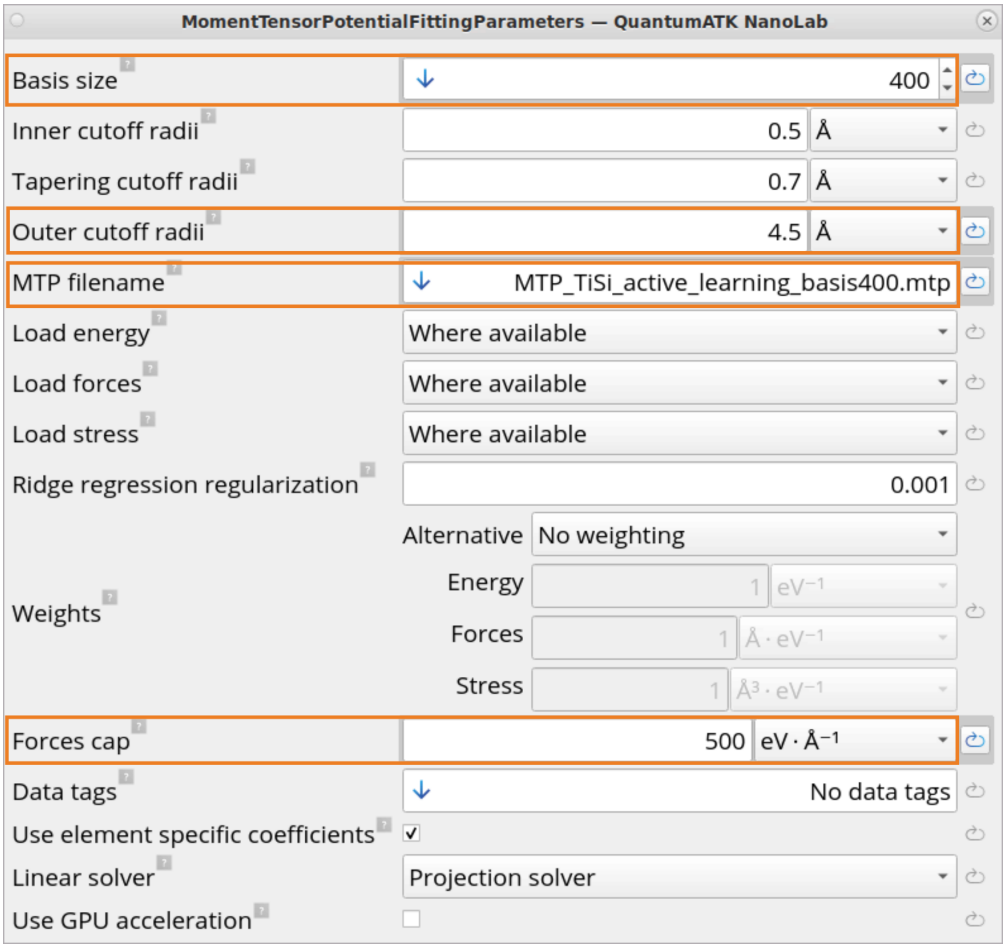
•*Limit candidates*: Set to *100*. Limit the number of recorded candidates for each simulation. This can be beneficial in case the simulation moves between candidate and retrain threshold for a long period and many candidates are registered. In that case the candidates are typically very correlated and not all of them are needed. The limited candidates are then taken from the raw list of candidates at a regular interval, before a structural filter extracts the most different configurations and adds them to the training set.

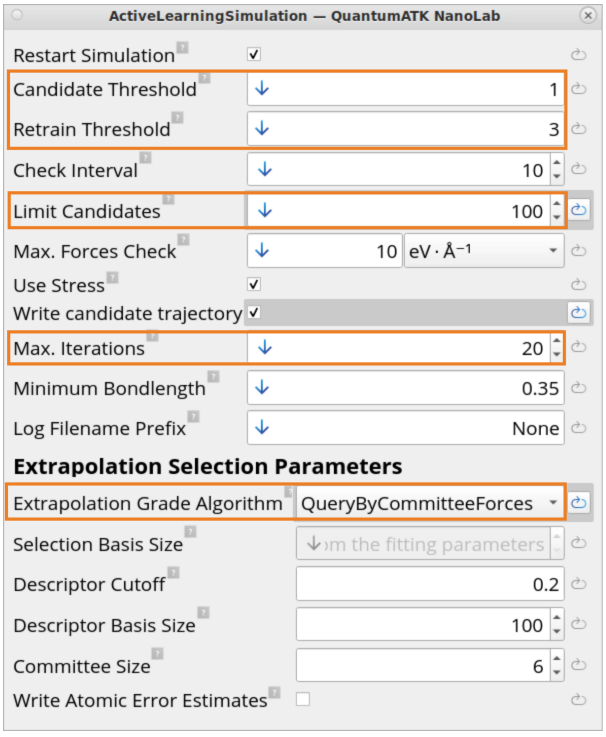
•*Max. Iterations*: The maximum number of re-training iterations, i.e., how many times the simulation will start again with a new MTP. Stick to the default value. It can be increased if more iterations are expected.

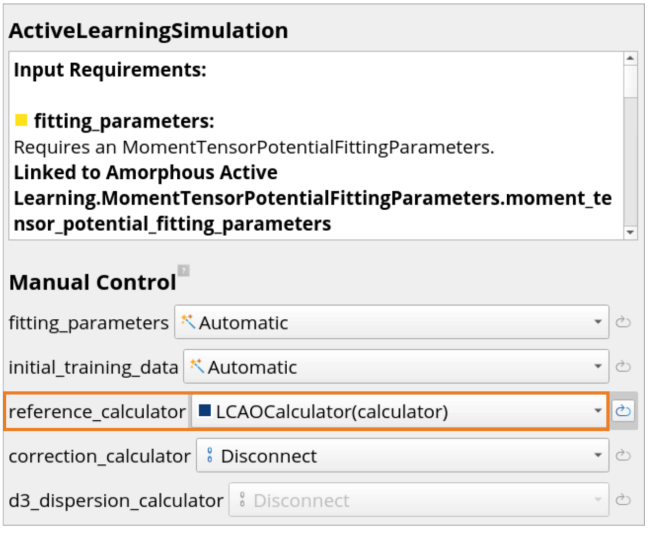
•*Extrapolation grade algorithm*: Select which algorithm is used to calculate the extrapolation grade. Unless there is only a single element in the simulation, we always recommend to set this to *QueryByCommitteeForces*.

•*LCAOCalculator* and *ActiveLearningSimulation* blocks must be connected, so that the defined *LCAOCalculator* is used to calculate training data during Active learning.

5. The **ActiveLearningOptimizeGeometry** block contains settings for performing Active learning geometry optimization. All initial amorphous configurations from the *Initial Amorphous Structures ConfigurationTable(List)* block will be used as a starting point for the *ActiveLearningOptimizeGeometry*. The optimizations will be run independently and

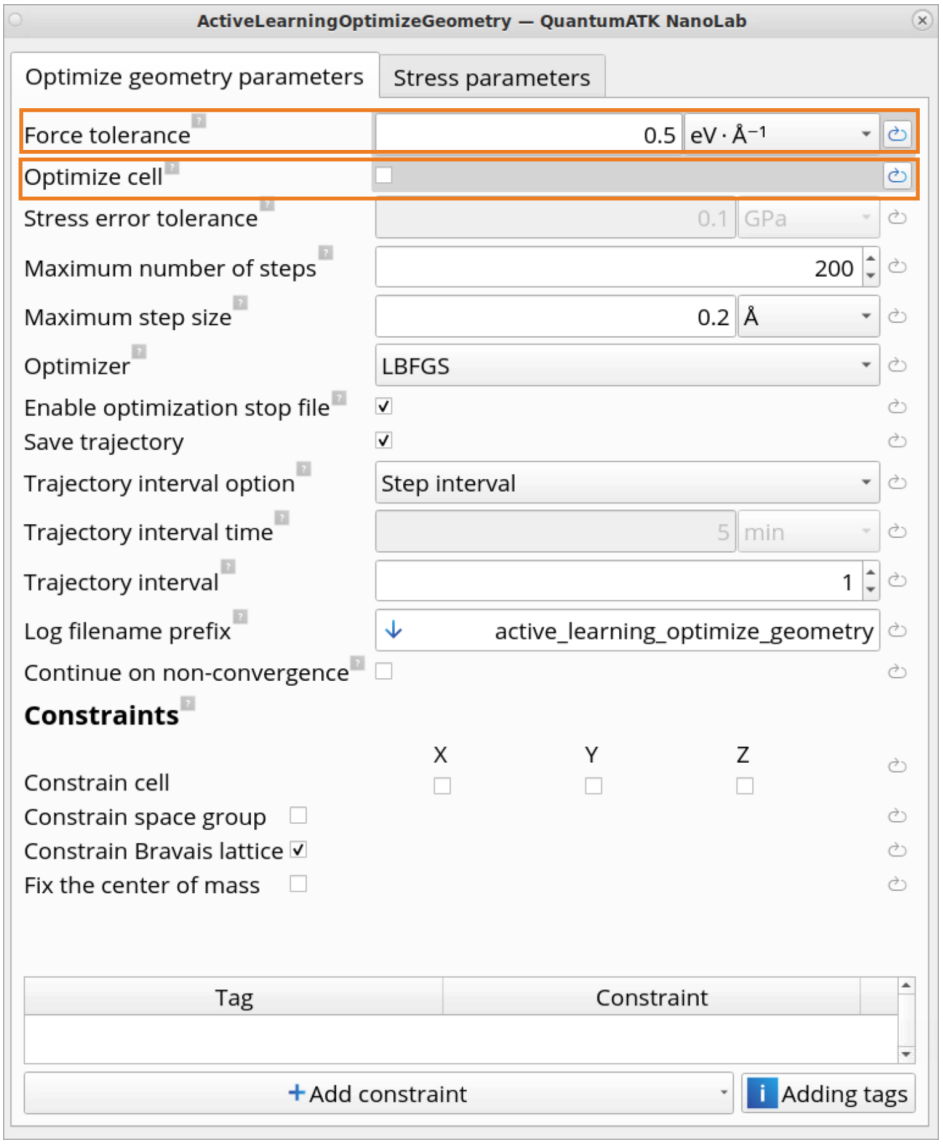

MomentTensorPotentialFittingParameters — QuantumATK NanoLab
Basis size
400
Inner cutoff radii
0.5
Å
Tapering cutoff radii
0.7
Å
Outer cutoff radii
4.5
Å
MTP filename
MTP_TiSi_active_learning_basis400.mtp
Load energy
Where available
Load forces
Where available
Load stress
Where available
Ridge regression regularization
0.001
Alternative
No weighting
Weights
Energy
1
eV⁻¹
Forces
1
Å · eV⁻¹
Stress
1
Å³ · eV⁻¹
Forces cap
500
eV · Å⁻¹
Data tags
No data tags
Use element specific coefficients
Linear solver
Projection solver
Use GPU acceleration


ActiveLearningSimulation — QuantumATK NanoLab
Restart Simulation
Candidate Threshold
1
Retrain Threshold
3
Check Interval
10
Limit Candidates
100
Max. Forces Check
10
eV · Å⁻¹
Use Stress
Write candidate trajectory
Max. Iterations
20
Minimum Bondlength
0.35
Log Filename Prefix
None
Extrapolation Selection Parameters
Extrapolation Grade Algorithm
QueryByCommitteeForces
Selection Basis Size
Descriptor Cutoff
0.2
Descriptor Basis Size
100
Committee Size
6
Write Atomic Error Estimates


ActiveLearningSimulation
Input Requirements:
fitting_parameters:
Requires an MomentTensorPotentialFittingParameters.
Linked to Amorphous Active Learning.MomentTensorPotentialFittingParameters.moment_tensor_potential_fitting_parameters
Manual Control
fitting_parameters
Automatic
initial_training_data
Automatic
reference_calculator
LCAOCalculator(calculator)
correction_calculator
Disconnect
d3_dispersion_calculator
Disconnect

in parallel (as much as the parallel settings allow that, otherwise sequentially). The DFT calculations and training are done collectively, using the combined extrapolating candidate configurations selected from all optimizations.

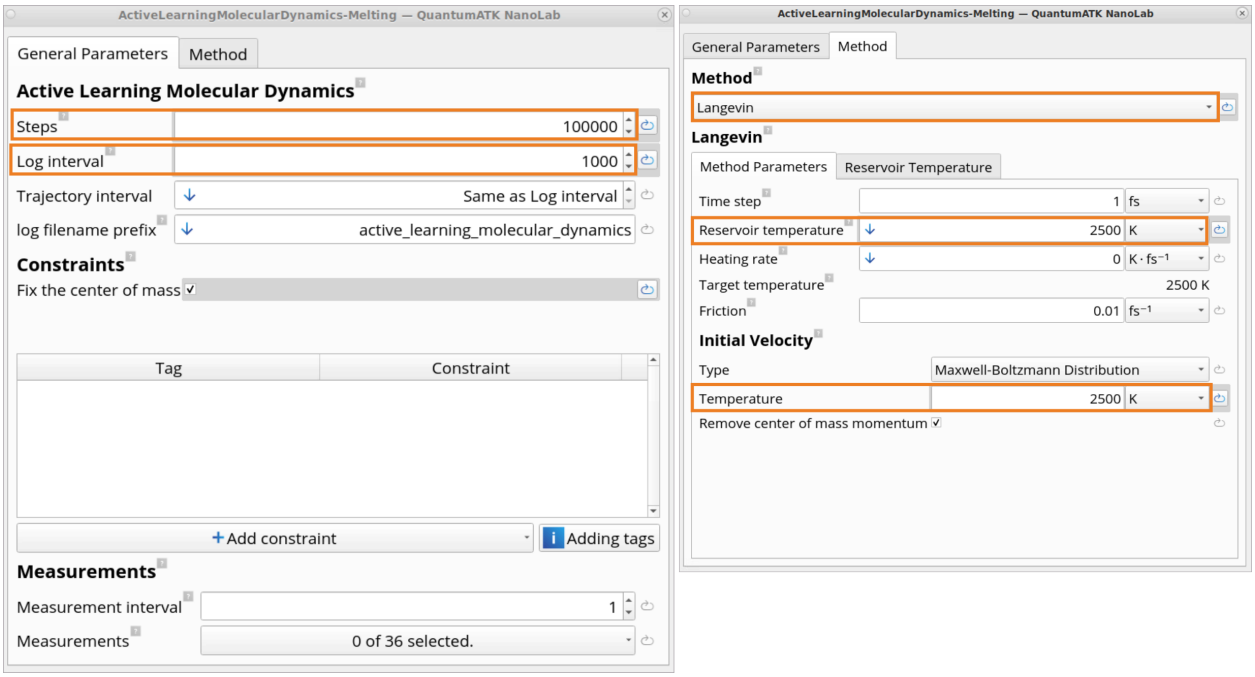
- *Force tolerance*: set a bit higher than the default, i.e., to *0.5 eV / Ang*, as this part should primarily do a quick pre-optimization to avoid large forces in the active learning MD part.
- *Optimize cell*: this is disabled to make the optimization more robust.



The combined crystal training + amorphous geometry optimization training data is used to train an improved MTP, which is then used for subsequent Active learning melt-quench MD simulations.

6. The **ActiveLearningMolecularDynamics-Melting** block contains settings for performing Active learning MD melting simulations. Here, we want to run MD at relatively high temperatures to melt the disordered configurations and sample many different relevant atomic environments.

- *Steps*: set to *100 000* MD steps and set *Log interval* to *1000* MD steps.
- *Method*: set to *Langevin*.
- *Reservoir temperature* and *Temperature of initial velocities* is set to *2500 K* to melt the disordered configurations.



Tip: One can also use a different thermostat type than *Langevin*, however, to ensure the simulation is relatively robust one should pick a robust thermostat, like Berendsen or Bussi-Donadio-Parrinello. At such high temperatures, it is recommended to run NVT simulations, whereas NPT simulations should primarily be run at lower temperatures.

Note:

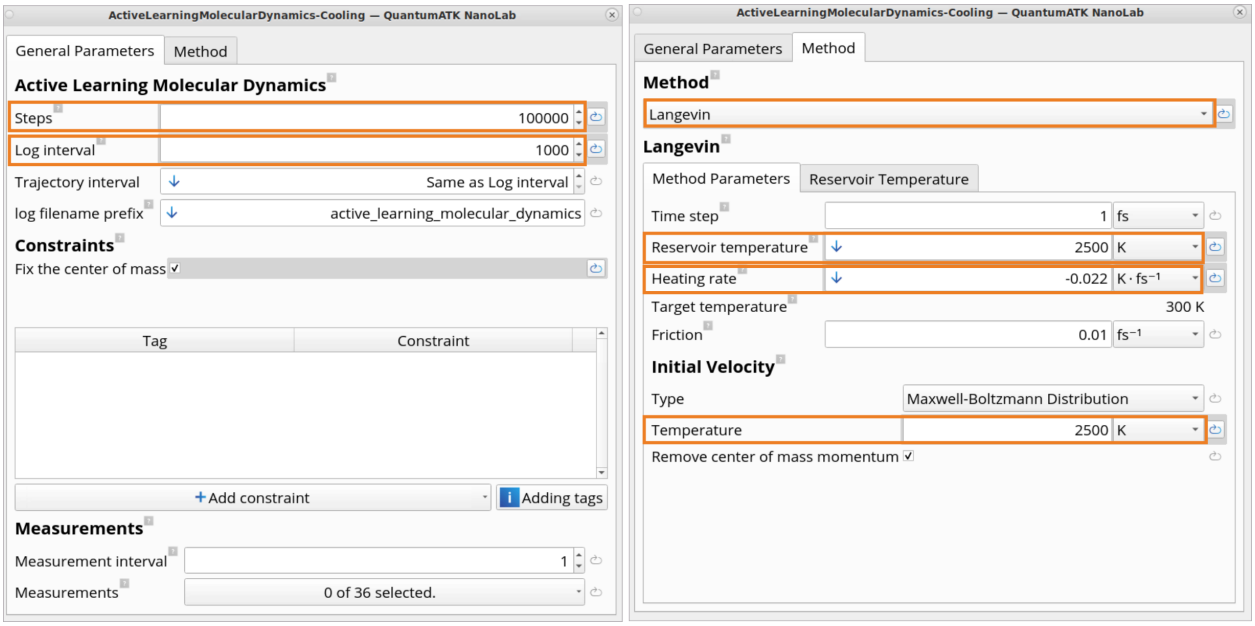
- The chosen **Active learning MD temperature** should reflect a physical process (e.g., chemical reactions, phase transitions, or material properties) that occur within a specific temperature range and that you want to simulate in production simulations with a trained MTP.
- The temperature should not be too low as it would miss configurations relevant at high temperatures (e.g., disordered amorphous structures).
- The temperature should also not be too unphysically high, as in that case the system may visit configurations that are physically unrealistic, or not representative of the systems true equilibrium state. In addition, due to too high temperature, the system may also fail to properly sample the configuration space by jumping over energy barriers rather than exploring the free energy surface in a smooth, gradual manner.
- To sum it up, too low or too high temperature could result in the reduced accuracy of Active learning and thus the trained MTP.

The combined crystal training + amorphous geometry optimization + amorphous MD melting training data is used to train a further improved MTP, which is then used in subsequent Active learning MD cooling simulations.

7. The **ActiveLearningMolecularDynamics-Cooling** block contains settings for performing Active learning MD cooling simulations - to simulate cooling the liquid configuration to room temperature.

- *Steps*: set to *100 000* MD steps and set *Log interval* to *1000* MD steps.
- *Method*: set to *Langevin*.

•*Reservoir temperature* and *Temperature of initial velocities* is set to *2500 K*, but now we specify a negative heating rate of *-2200/100 000 K/fs* to decrease the temperature from *2500 K* to *300 K* during the 100 ps of MD.

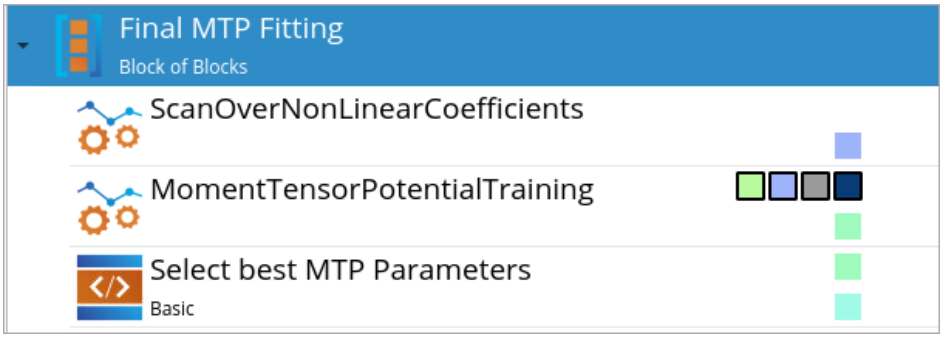


8. The **Final-Training-Set-Table** block picks up the *Training-Set-Table* containing the combined data from crystal displacements and Active learning and exposes it under a more obvious name, so that it can easily be used in the following **Final training** stage.

Final MTP Fitting

In the 2nd *Amorphous Active Learning* stage we did preliminary MTP fittings to generate amorphous training data for titanium silicide on-the-fly. For this, we used only one set of initial non-linear MTP coefficients. In the 3rd stage, **Final MTP Fitting**, we will fit an ensemble of MTP models to the crystal+amorphous training data and select the best fitted MTP. We will scan over 30 different initial guesses of non-linear MTP coefficients to generate different fitted MTPs based on these initial guesses. We then select the one which gives the lowest, and comparable, training and testing errors with respect to DFT. This can then be used in production simulations.

The actual workflow for the **Final MTP Fitting** stage in the **Workflow Builder** looks like this:



1. The **ScanOverNonLinearCoefficients** block contains settings for scanning over different initial guesses of non-linear MTP coefficients and obtaining MTP fitting parameters for these different initial guesses. *scanOverNonLinearCoeffi-*

cients is essentially an outer loop over *NonLinearCoefficientsParameters* + *MomentTensorPotentialFittingParameters* that loops over different random seeds.

•*Number of initial guesses*: Defines how many different initial values should be considered. The default value of *30* is usually sufficient. Increasing the number means a better chance of finding the best possible MTP at a cost of an approximately a linear increase in the calculation time.

•*Basis size*: Use *400* basis functions for this example (same as in active learning). Generally, it is recommended to use either *Predefined-Small*, or between *300 – 800* basis functions. More basis functions usually provide a better accuracy (lower training error), but the simulation can become less stable, especially in out-of-equilibrium situations, e.g., at high temperatures or energies. For calculations which require a high accuracy, but not stable MD, it can be meaningful to increase the number of basis functions beyond this range. In rare cases, mainly for calculations which require a very high accuracy, such as precise phonon calculations or defect energies, it can be meaningful to increase the number of basis functions to as much as 5000.

•*Cutoff radii*: Inner and tapering cutoff radii rarely need to be changed from their defaults. The outer cutoff radius should be set to the same as in Active learning (*4.5 Å*), as the extrapolation behavior will depend on this choice.

•*MTP filename suffix*: Specify the filename, where the MTP parameters are written to. The different fits will create files such as `0_mtp_filename_suffix.mtp`, `1_mtp_filename_suffix.mtp`, etc.

•*Forces cap*: Specify this value in order to exclude configurations with large forces. In this stage it does not have to be as large as for Active learning, we recommend *100 -200 eV / Ang* for most cases, unless the application explicitly requires to be able to deal with very large forces (e.g. impact of high-energy ions). We choose *200* here, so that some large forces are included to train the MTP to be able to deal with close atoms, that might occur during the melting MD simulations.

•*Perform optimization*: Run a separate, local optimization of the non-linear coefficients. This takes much longer, and in most cases it gives very little improvement in accuracy, therefore it is recommended to have it disabled (default).

Tip:

•*ScanningOverNonLinearCoefficients* is kind of a non-linear MTP optimization, but a crude one in that one just tests many initial guesses and then picks the best one (without the explicit local optimization of non-linear coefficients). For systems that do not have too many elements it works well.

•If the number of elements becomes larger (5+ different elements), it can be beneficial to use the particle-swarm-optimization (PSO) method *ParticleSwarmOptimizationParameters* to optimize the non-linear parameters instead of scanning over different initial guesses. This will take longer, but provides a more systematic optimization and often results in a better accuracy for complex systems. Since we only have 2 elements in the system we use here, we will not use PSO in this tutorial.

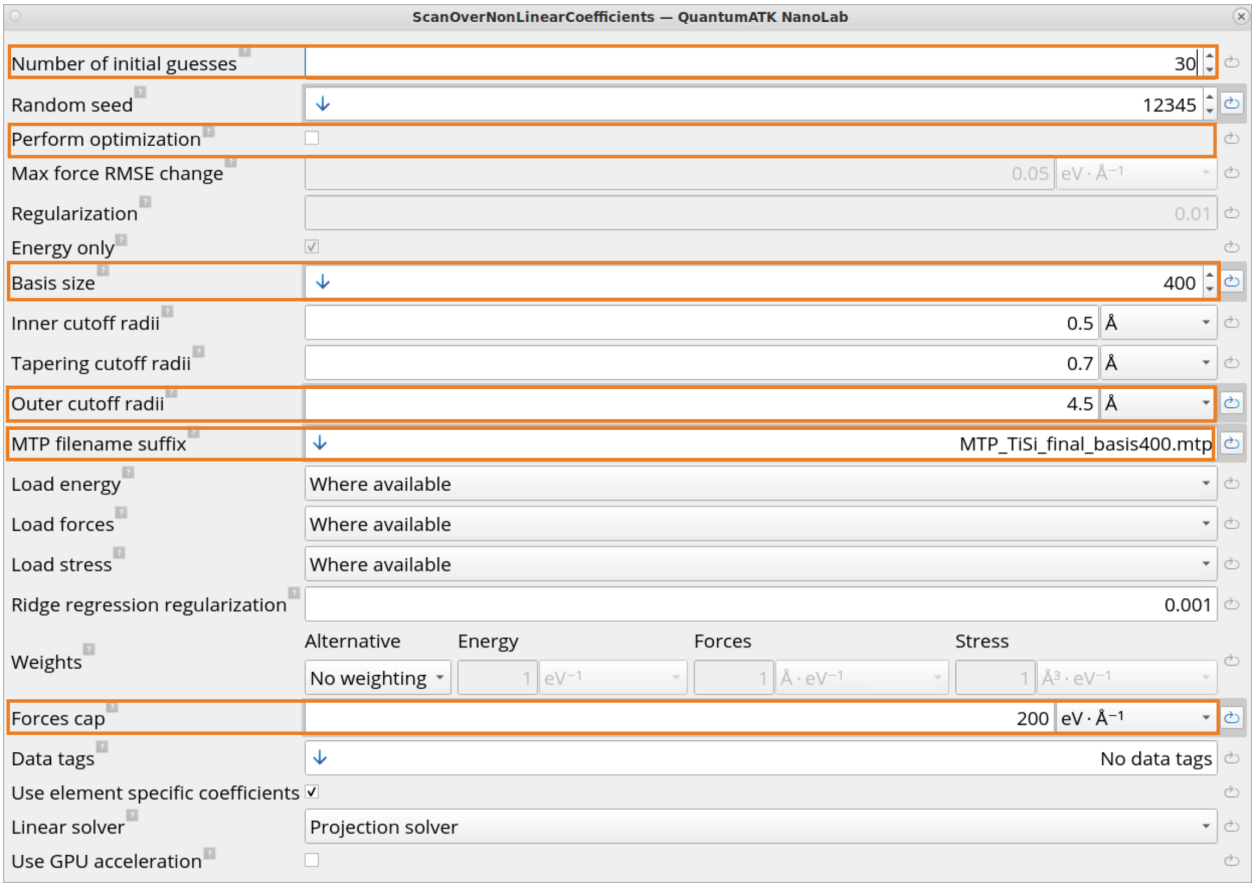
2.The **MomentTensorPotentialTraining** block contains settings for taking care of running and analyzing the MTP fitting runs based on 30 different sets of MTP fitting parameters. See *MomentTensorPotentialTraining* to learn more.

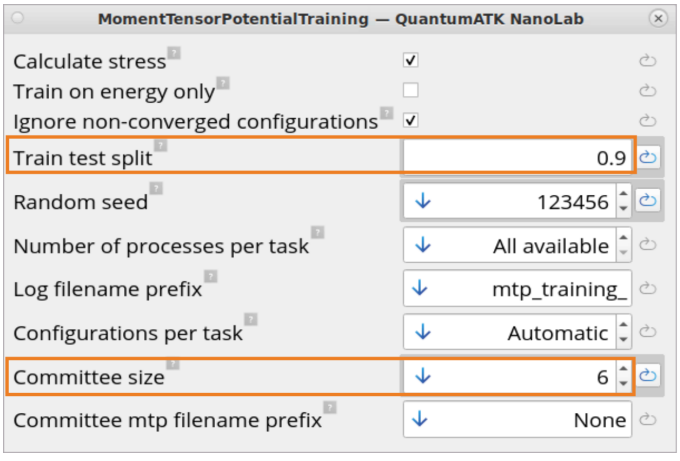
•*Train test split*: Set it to *0.9* to use 90 % of the calculated reference data for training and 10 % for testing. The split is done randomly. Specify an arbitrary number for the random seed to make it deterministic and reproducible, e.g., if one wants to change other hyperparameters like a cutoff radii or a basis size later and compare the results.

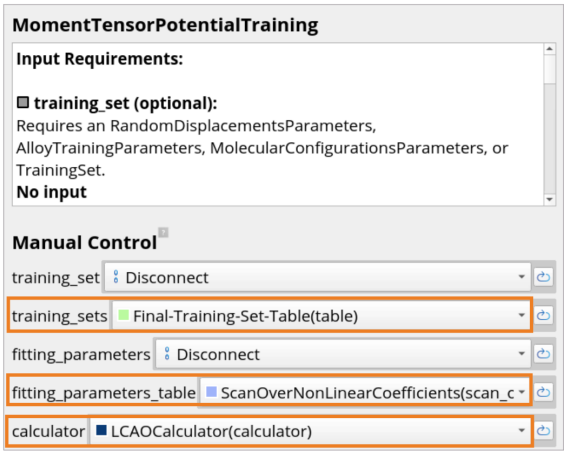
•*Committee size*: Number of committee models to train and store using the same data and hyperparameters for the MTP MD uncertainty predictions. This can be set to a small number (we recommend 6). The committee model can then be used in MD simulations to obtain an estimate of the energy and forces error during the production simulations with a trained MTP, using the *MolecularDynamicsErrorPredictionHook*.

•Ensure that the *MomentTensorPotentialTraining* block is linked to the *Training-Set-Table*, *ScanOverNonLinearCoefficients* and *LCAOCalculator* blocks.

Note:


ScanOverNonLinearCoefficients — QuantumATK NanoLab
Number of initial guesses
30
Random seed
12345
Perform optimization
Max force RMSE change
0.05
eV · Å⁻¹
Regularization
0.01
Energy only
Basis size
400
Inner cutoff radii
0.5
Å
Tapering cutoff radii
0.7
Å
Outer cutoff radii
4.5
Å
MTP filename suffix
MTP_TiSi_final_basis400.mtp
Load energy
Where available
Load forces
Where available
Load stress
Where available
Ridge regression regularization
0.001
Weights
Alternative
Energy
Forces
Stress
No weighting
1
eV⁻¹
1
Å · eV⁻¹
1
Å³ · eV⁻¹
Forces cap
200
eV · Å⁻¹
Data tags
No data tags
Use element specific coefficients
Linear solver
Projection solver
Use GPU acceleration


MomentTensorPotentialTraining — QuantumATK NanoLab
Calculate stress
Train on energy only
Ignore non-converged configurations
Train test split
0.9
Random seed
123456
Number of processes per task
All available
Log filename prefix
mtp_training_
Configurations per task
Automatic
Committee size
6
Committee mtp filename prefix
None

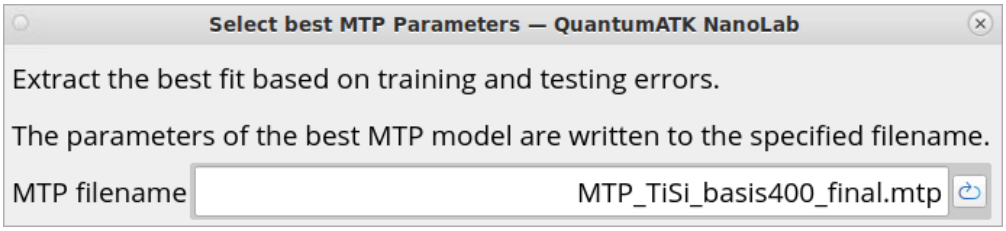

MomentTensorPotentialTraining
Input Requirements:
training_set (optional):
Requires an RandomDisplacementsParameters, AlloyTrainingParameters, MolecularConfigurationsParameters, or TrainingSet.
No input
Manual Control
training_set
Disconnect
training_sets
Final-Training-Set-Table(table)
fitting_parameters
Disconnect
fitting_parameters_table
ScanOverNonLinearCoefficients(scan_c
calculator
LCAOCalculator(calculator)

•We use random numbers to initialize the coefficients for the MTP training, randomly displacing crystal configurations and training-test set splitting. Therefore, reproducibility of the results to the numerical accuracy is only ensured if you use the same **Random seed** while re-running the MTP training simulation.

•Many different sets of MTP parameters could result in similar results as there could be many degenerate minima in the hyper-parameter surface. Thus it is not a cause for concern if you get different sets of MTP fitting parameters with similar accuracy.

3.The **Select best MTP Parameters** block is for selecting the best fitted MTP from 30 generated MTPs based on training and testing errors.

•*MTP filename*: The parameters for the best fit are written to this file. It should have the extension `.mtp`. This mtp file is what you will use for setting up production simulations with the trained MTP.



Submitting the MTP Training Calculation

The MTP training set-up is now finished and you are ready to submit this job.

1.Change the *Workflow* name and *Filename* as needed (e.g., `c-am-TiSi-MTP-training-results.hdf5`).

2.Click on the *Send content to other tools* button and choose *Jobs as a script* to send the workflow to the **Jobs** tool to submit this job on a local machine or a remote cluster. It is recommended to use MPI parallelization for MTP training simulations, as both training data calculations with DFT and MTP fitting benefit from MPI parallelization.

3.This also generates a Python script `c-am-TiSi-MTP-training-results.py`, based on this workflow. It can be opened and edited with the **Editor** tool and then sent to the **Jobs** tool, if you need to make manual changes.

4.Estimate of the MTP training run-time: in our test it took 5 hours on the 40-core machine.

Analyzing MTP Training Results

Once the MTP training calculation has finished, we are ready to analyze the results.

Download the following output files if needed: `c-am-TiSi-MTP-training-results.log`, `c-am-TiSi-MTP-training-results.hdf5`, `c-am-TiSi-MTP-training-results_4.hdf5`, `active_learning_molecular_dynamics.4.log` and open them in the **Data View**.

1.`c-am-TiSi-MTP-training-results.log`: The main `.log` file contains a lot of information about the progress of the overall simulation, while the DFT training data, Active learning geometry optimization and MD log outputs are written to separate files. The most important parts of the `c-am-TiSi-MTP-training-results.log` file are:

•**Completion of Active learning** geometry optimization and melt-quench MD simulations, reporting the number of added extrapolating amorphous training configurations. For titanium silicide, 14, 8 and 0 extrapolating amorphous configurations were added during Active learning geometry optimization, MD melting and cooling phases, respectively.

```
+-------------------------------------------------------------------------------+
| Active learning geometry optimization completed successfully.                 |
| Number of iterations for MTP training 2                                       |
| Number of additional training configurations: 14                              |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Active learning MD completed successfully.                                    |
| Number of iterations for MTP training: 2                                      |
| Number of additional training configurations: 8                               |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Active learning MD completed successfully.                                    |
| Number of iterations for MTP training: 1                                      |
| Number of additional training configurations: 0                               |
+-------------------------------------------------------------------------------+
```

- **MTP training report** at the end of the `.log` file. This displays the size of the training set (216 configurations 90% of all training configurations) and testing set (25 configurations 10% of all training configurations) for each fit. It also shows a list with all the training and testing RSME errors and r^2 values for energy, forces, and stress for all 30 fits with respect to the reference DFT calculator. In this case, each fit corresponds to a different initial guess for the non-linear coefficients. One can inspect the errors manually and select the best fitted MTP with lowest RMSE errors.

- **Best MTP fit**, reported by QuantumATK based on the automatic analysis of all the errors. In this case, QuantumATK reports the MTP fit number 19 to be the best, corresponding to the lowest overall training and testing errors. The best fitted MTP is written to `MTP_TiSi_basis400_final.mtp`, which can then be used for validation and production simulations. All the 30 fitted MTPs are written to files `0,...,29_MTP_TiSi_final_basis400.mtp`.

```
+-------------------------------------------------------------------------------+
| Moment Tensor Potential Training Report                                       |
+-------------------------------------------------------------------------------+
| Index                 Training set size(s)            Testing set size(s)     |
|    19                                 216                                25   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| RMSE (training):                                                              |
|                                                                               |
| Index             Filename     Energy/atom           Force          Stress    |
|                                       (eV)         (eV/Å)        (eV/Å^3)     |
|     0 0_MTP_TiSi_[...].mtp    0.0025043219    0.1012940348    0.0389318624   |
|     1 1_MTP_TiSi_[...].mtp    0.0029281002    0.1158755465    0.0361733476   |
|     2 2_MTP_TiSi_[...].mtp    0.0037681036    0.1250996880    0.0376190870   |
|     3 3_MTP_TiSi_[...].mtp    0.0028928380    0.1201118210    0.0429722560   |
|     4 4_MTP_TiSi_[...].mtp    0.0022172316    0.1082486877    0.0278265693   |
|     5 5_MTP_TiSi_[...].mtp    0.0028463811    0.1041497742    0.0395561149   |
|     6 6_MTP_TiSi_[...].mtp    0.0022442684    0.1029489858    0.0357410797   |
|     7 7_MTP_TiSi_[...].mtp    0.0040683096    0.1241173502    0.0543206816   |
|     8 8_MTP_TiSi_[...].mtp    0.0044881187    0.1169965922    0.0731741130   |
|     9 9_MTP_TiSi_[...].mtp    0.0019220696    0.0954770279    0.0309075969   |
|    10 10_MTP_TiSi[...].mtp    0.0022222135    0.1040334579    0.0271600170   |
|    11 11_MTP_TiSi[...].mtp    0.0020293338    0.0981887142    0.0296488370   |
|    12 12_MTP_TiSi[...].mtp    0.0037672343    0.1055226664    0.0613020433   |
|    13 13_MTP_TiSi[...].mtp    0.0027541694    0.0999899064    0.0464718686   |
|    14 14_MTP_TiSi[...].mtp    0.0028242243    0.1101901201    0.0451052006   |
```

(continues on next page)

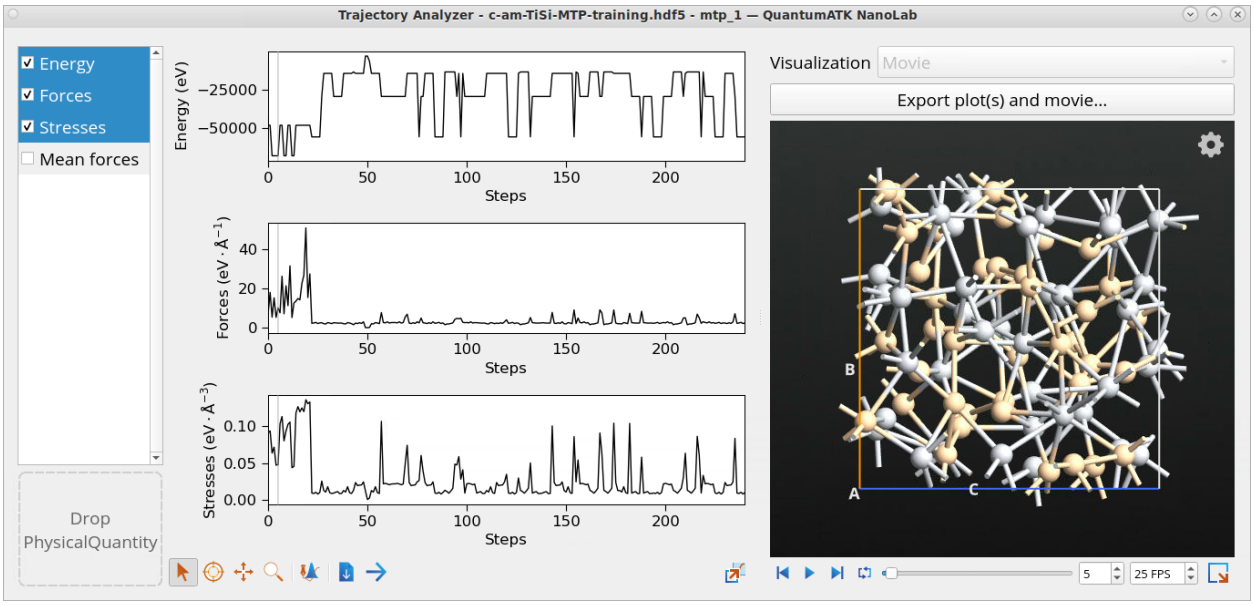
(continued from previous page)

```
|    15 15_MTP_TiSi[...].mtp    0.0033376688    0.1123514802    0.0511654854   |
|    16 16_MTP_TiSi[...].mtp    0.0035935329    0.1315763184    0.0431630558   |
|    17 17_MTP_TiSi[...].mtp    0.0028141484    0.1199110297    0.0192196160   |
|    18 18_MTP_TiSi[...].mtp    0.0027824647    0.1003360462    0.0409380488   |
|    19 19_MTP_TiSi[...].mtp    0.0015130631    0.0938986276    0.0186246025   |
|    20 20_MTP_TiSi[...].mtp    0.0037646145    0.1260588897    0.0243364675   |
|    21 21_MTP_TiSi[...].mtp    0.0035362934    0.1128509267    0.0456808120   |
|    22 22_MTP_TiSi[...].mtp    0.0024321919    0.1017323905    0.0413082318   |
|    23 23_MTP_TiSi[...].mtp    0.0025639095    0.1113184768    0.0416981450   |
|    24 24_MTP_TiSi[...].mtp    0.0017477251    0.1023449077    0.0162820468   |
|    25 25_MTP_TiSi[...].mtp    0.0035681856    0.1294370831    0.0291967493   |
|    26 26_MTP_TiSi[...].mtp    0.0032966773    0.1102070614    0.0526647940   |
|    27 27_MTP_TiSi[...].mtp    0.0021051690    0.0994845053    0.0240463666   |
|    28 28_MTP_TiSi[...].mtp    0.0032323621    0.1044183263    0.0573661541   |
|    29 29_MTP_TiSi[...].mtp    0.0021269500    0.1094758067    0.0216593875   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| RMSE (testing):                                                               |
|                                                                               |
| Index             Filename     Energy/atom           Force          Stress   |
|                                        (eV)          (eV/Å)        (eV/Å^3)   |
|    19 19_MTP_TiSi[...].mtp    0.0010749092    0.0838614510    0.0193958613   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| r^2 (training):                                                               |
|                                                                               |
| Index             Filename     Energy/atom           Force          Stress   |
|    19 19_MTP_TiSi[...].mtp    0.9999999998    0.9937161787    0.0391601952   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| r^2 (testing):                                                                |
|                                                                               |
| Index             Filename     Energy/atom           Force          Stress   |
|    19 19_MTP_TiSi[...].mtp    0.9999999999    0.9924226840   -1.1109615599   |
+-------------------------------------------------------------------------------+

Best fit at index 19.
Copying the best parameters to file MTP_TiSi_basis400_final.mtp.
```

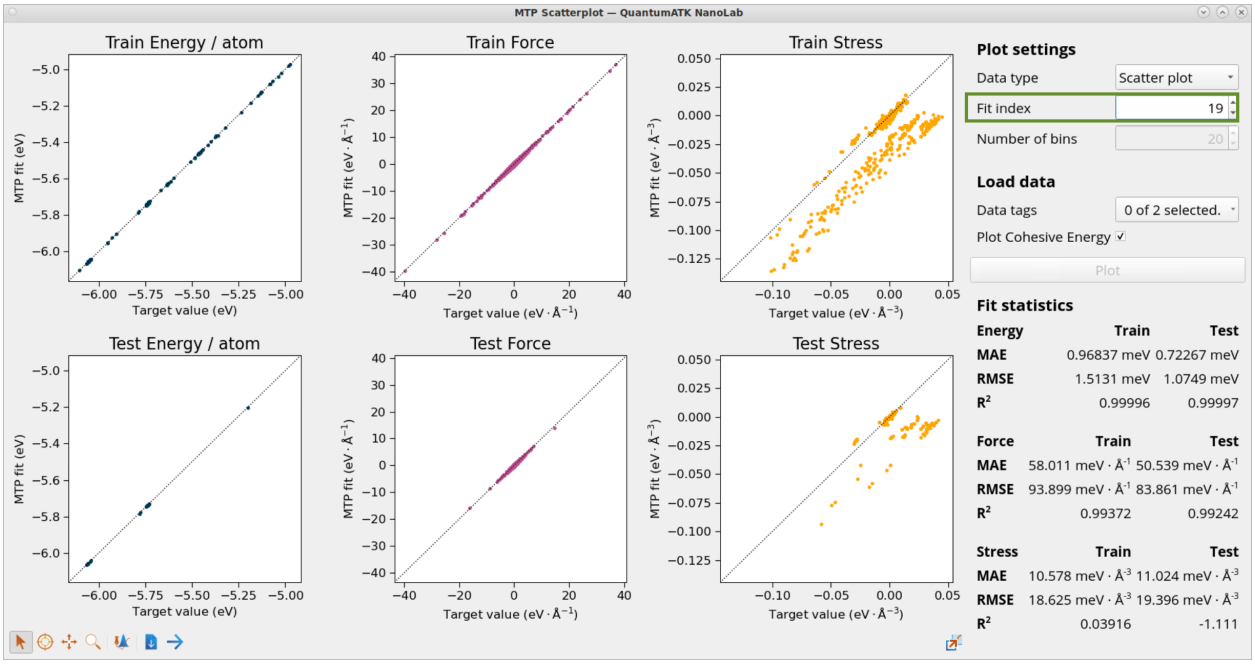
Note: r^2 values, coefficient of determination, indicate how good the MTP is at predicting the trained properties in the dataset, with values closer to 1 being better. Looking at the log-file, we can see that all of the trained MTPs have an r^2 value of over 0.99999999 indicating that they are all accurate within the configurational space sampled in the training data set, thus not helping much in choosing the most accurate fitted MTP. Therefore, to find the best MTP fit, one has to look for an MTP fit with lowest training and testing RMSE errors.

2. `c-am-TiSi-MTP-training-results.hdf5`: The main `.hdf5` file can be used to inspect crystal and amorphous training data, as well as training and testing errors for all 30 fits.

- **Amorphous and crystal training configurations**, along with DFT calculated energy, forces and stresses. Right-click on the *mtp_1* object to open it with the *Movie tool*. The total number of training configurations for titanium silicide is 241 from which 22 are amorphous configurations from the **Amorphous Active Learning** stage (in the beginning of the set) and 219 are crystal configurations from the **Crystal Training Data** stage.



•**MTP fitting analysis** to graphically inspect MAE, RMSE and r^2 values for all 30 MTP fits. Right-click on the *mtp_1* object to open it with the *MTP Analyzer*. Click on *Fit index* to choose a fit, e.g., 19 (19 is the best fit in this case), and then click *Plot*. The *MTP Analyzer* produces a scatterplot of MTP predicted values vs. the reference DFT values for energy, forces, and stress, for each of the training and testing datasets.



The above scatter plot compares the MTP predicted data (y-axis) and reference DFT data (x-axis). The energy, forces and stress values are compared for both training and test sets and indicates high accuracy of the trained MTP: the data points for energy and forces lie along the 45 degree line and for stress there are minor deviations, but overall good correlation.

Note: We recommend to look at the training configurations to aid potential debugging. In case your production simulations with the trained MTP give unexpected results, check if the training data set includes all the important

configurations that needs to be included.

3.Look at `c-am-TiSi-MTP-training-results_x.hdf5`, `active_learning_molecular_dynamics.x.log` and `active_learning_geometry_optimization.x.log` to inspect Active learning geometry optimization and melt-quench MD trajectories. Since we started from 8 different initial amorphous configurations, we have 8 different `.hdf5/.log` files, where x=0-7.

•Each `.hdf5` file contains an *alopt*, *md* and *md_2* objects, which can opened with the *Movie tool*

•The *alopt* object is for inspecting the Active learning geometry optimization trajectory along with the calculated energy and forces.

•The *md* object is for the Active learning MD melting trajectory along with the calculated temperature, energy, pressure, extrapolation grade, etc. every 10 MD steps.

•The *md_2* object is for the Active learning MD cooling trajectory along with the calculated temperature, energy, pressure, extrapolation grade, etc. every 10 MD steps.

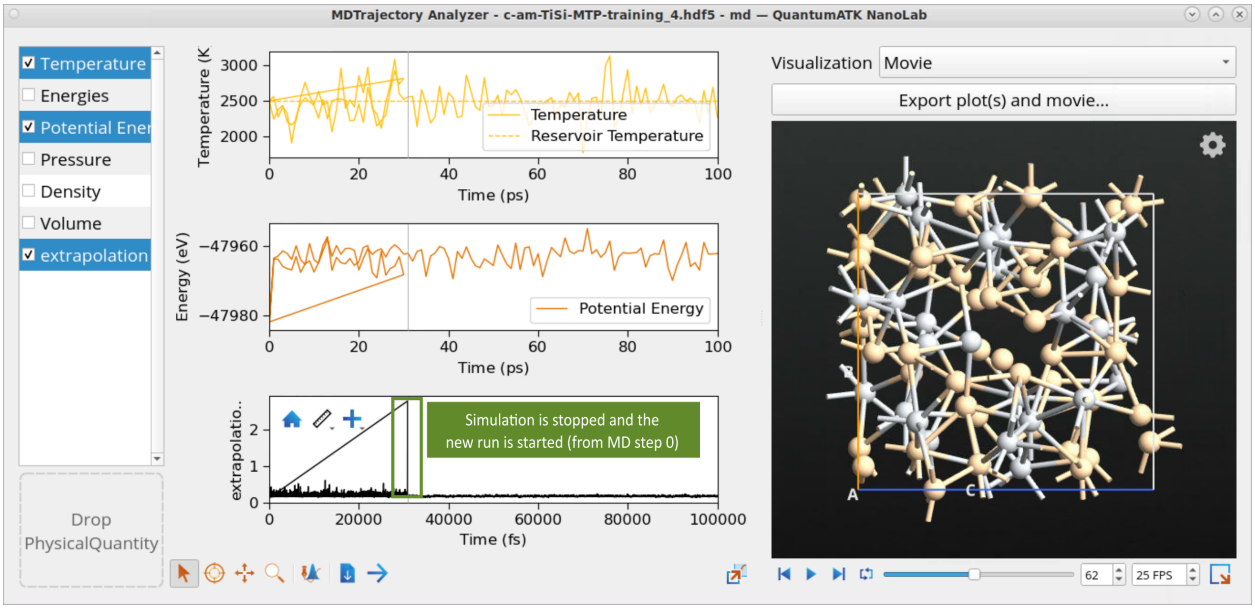
•Open, e.g., the *md* object from the `c-am-TiSi-MTP-training-results_4.hdf5` with the *Movie Tool* and `active_learning_molecular_dynamics.4.log` with *Text Representation* to inspect MD trajectory during the Active learning MD melting phase.

```
| MD      23000       23000.0    -4.79677e+04          2418.3        2500.0    |
| MD      24000       24000.0    -4.79693e+04          2148.1        2500.0    |
| MD      25000       25000.0    -4.79616e+04          2224.4        2500.0    |
| MD      26000       26000.0    -4.79685e+04          2301.4        2500.0    |
| MD      27000       27000.0    -4.79666e+04          2644.5        2500.0    |
| MD      28000       28000.0    -4.79659e+04          2918.8        2500.0    |
| MD      29000       29000.0    -4.79632e+04          2733.6        2500.0    |
| MD      30000       30000.0    -4.79684e+04          2812.7        2500.0    |
| Step 30850: Extrapolation grade 1.7276932788031036. Adding to candidates     |
| Step 30860: Extrapolation grade 2.797831465597506. Adding to candidates      |
| Step 30870: Extrapolation grade 18.301103354747035.                          |
| Stopping simulation and starting new training.                               |
+------------------------------------------------------------------------------+
| Selecting the most different candidate configurations to add to the training |
| data. The selection is started from 3 candidate configurations.              |
| 3 configurations have been selected to be added to the training data.        |
+------------------------------------------------------------------------------+
+------------------------------------------------------------------------------+
| TremoloX Domain Decomposition Report                                         |
+------------------------------------------------------------------------------+
| Enabled: False                                                               |
| Number of Domains: 1                                                         |
| Decomposition Pattern: 1x1x1                                                 |
| CPU Information:                                                             |
|  Process ID 0 at de02tcadhpc16 (1 thread)                                    |
+------------------------------------------------------------------------------+
+------------------------------------------------------------------------------+
| MD       step          time       potential   instantaneous     reservoir    |
| MD                                    energy     temperature   temperature    |
| MD                      (fs)            (eV)             (K)           (K)    |
+------------------------------------------------------------------------------+
| MD          0           0.0    -4.79820e+04          2500.0        2500.0    |
| MD       1000        1000.0    -4.79636e+04          2243.0        2500.0    |
```

(continues on next page)

(continued from previous page)

```
| MD        2000        2000.0    -4.79629e+04          2377.8        2500.0    |
| MD        3000        3000.0    -4.79615e+04          2435.5        2500.0    |
| MD        4000        4000.0    -4.79622e+04          2207.2        2500.0    |
```



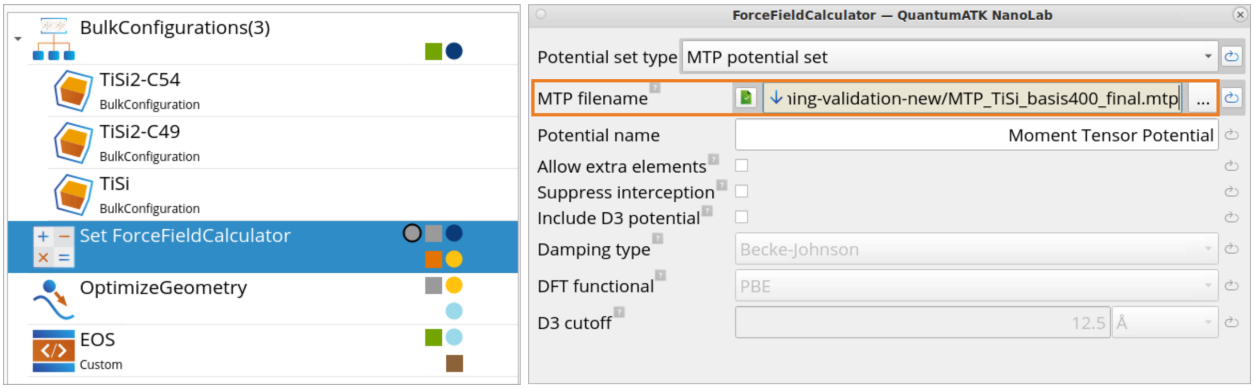
- The extrapolation grade becomes 1.73 at step 30850 and 2.80 at step 30860. As these are larger than the candidate threshold of 1, but smaller than the retrain threshold of 3, these extrapolating configurations are kept as candidates to be included in the next training and the simulation proceeds.
- When the extrapolation grade becomes 18.3 at step 30870, thus larger than the retrain threshold, the current simulation will be stopped, extrapolating configurations will be added to the training data set, and then Active learning MD will be re-started now using an improved MTP fitted to the extended training data set.

Note: As shown in this tutorial, Active learning targets PES areas where the model is uncertain and selects the extrapolating amorphous configurations (22 in this case), for which to calculate DFT training data, thus minimizing the number of expensive DFT calculations needed. Randomly sampling PES using MD alone would require much longer simulation time scales, which would then become computationally unfeasible if DFT is used for each MD step.

MTP Validation for Crystals

To validate the trained MTP for crystals, one can use the trained MTP to calculate lattice constants and equation-of-state (EOS) for the relevant crystal phases of titanium silicide, TiSi, $TiSi_2$-C54 and $TiSi_2$-C49 and compare with the corresponding DFT results.

1. Download `c-TiSi-DFT-EOS.hdf5` and `c-TiSi-MTP-EOS.hdf5` workflow files for DFT and MTP, respectively, and open them in the **Workflow Builder**.
2. In the *Set ForceFieldCalculator* block, specify the location and name of the final trained MTP, `MTP_TiSi_basis400_final.mtp`, which will be used for the validation.



Run the workflows, and once the calculations of optimizing lattice constants and obtaining EOS are finished, we are ready to analyze the results.

3. Download `c-TiSi-DFT-EOS-results.hdf5` / `c-TiSi-MTP-EOS-results.hdf5` and then in the *Data View* window select the *optgeom*, *optgeom_1* and *optgeom_2*, which contain optimized geometries of the three different crystal phases.

4. Right-click on *optgeom*, *optgeom_1* and *optgeom_2* objects to *Open All with Text Representation*. Inspect and compare lattice constants of optimized geometries obtained with the trained MTP and DFT. MTP gives good agreement with DFT results (within 1-2%) for all crystal phases, as shown in the table below.

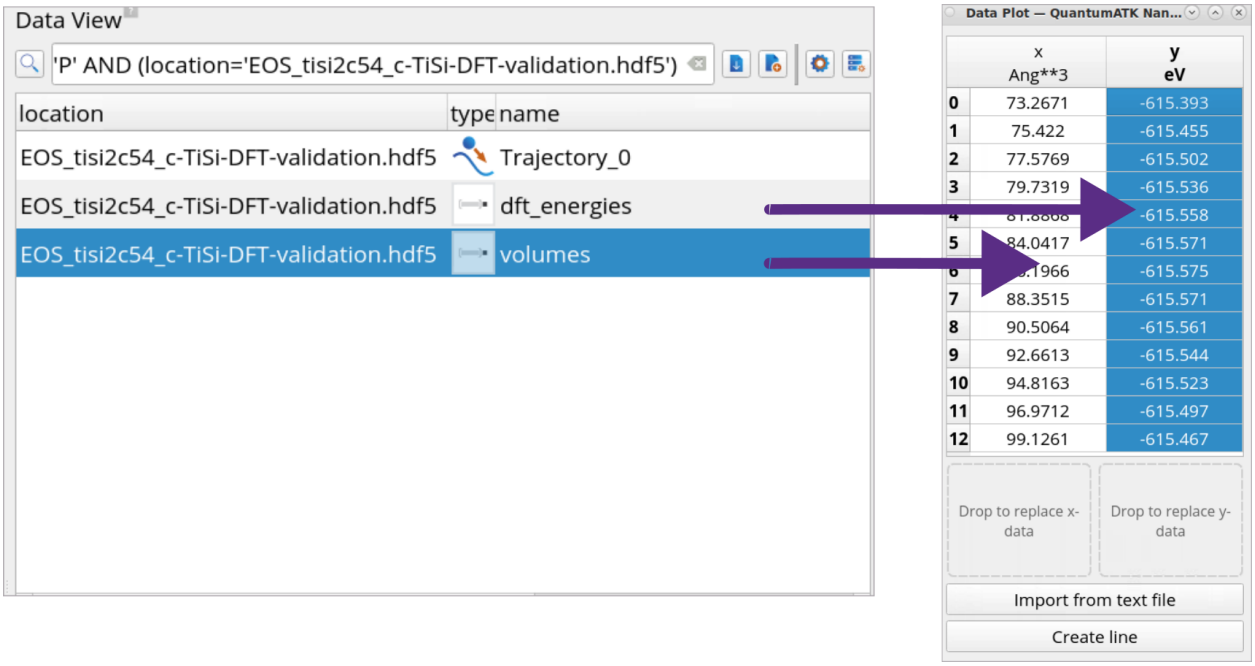
```
# Item:     1
# Location: c-TiSi-MTP-EOS.hdf5
# Name:     optgeom
# Type:     BulkConfiguration
+-------------------------------------------------------------+
| Bulk Bravais lattice                                        |
+-------------------------------------------------------------+
Type:
FaceCenteredOrthorhombic

Lattice constants:
a =      4.779122 Ang
b =      8.162575 Ang
c =      8.694788 Ang
```

Lattice constants with MTP vs. DFT

	TiSi		$TiSi_2$-C54		$TiSi_2$-C49	
	MTP	DFT	MTP	DFT	MTP	DFT
a	4.779 Å	4.829 Å	3.524 Å	3.558 Å	3.657 Å	3.647 Å
b	8.163 Å	8.306 Å	13.298 Å	13.644 Å	4.955 Å	5.025 Å
c	8.695 Å	8.595 Å	3.604 Å	3.591 Å	6.419 Å	6.562 Å

5. Calculated EOS (energy as a function of volume) are saved in these files `EOS_tisi2c49_c-TiSi-MTP-validation.hdf5`, `EOS_tisi2c54_c-TiSi-MTP-validation.hdf5`, `EOS_tisi_c-TiSi-MTP-validation.hdf5` for MTP and in these files `EOS_tisi2c49_c-TiSi-DFT-validation.hdf5`, `EOS_tisi2c54_c-TiSi-DFT-validation.hdf5`, `EOS_tisi_c-TiSi-DFT-validation.hdf5` for DFT.

6. In **Data View**, find the *Data Plot* tool under *Supplementary tools*, and then drag-and-drop *volumes* and *dft_energies* from the *Data View* panel to the *Data Plot* tool *x* and *y* drop areas, respectively.



7. Click *Create line* to plot EOS for each crystal phase with DFT and MTP.
8. Repeat the steps 6 and 7 for each crystal phase with DFT and MTP for a total of 6 plots.
9. Drag-and-drop the handle from one plot to another to combine the plots into one.
10. Combined plots for $TiSi_2$-C54 and $TiSi_2$-C49 show that the trained MTP reproduces the DFT trend, so that the $TiSi_2$-C49 crystal phase is lower in energy than the $TiSi_2$-C54 crystal phase at the minimum.

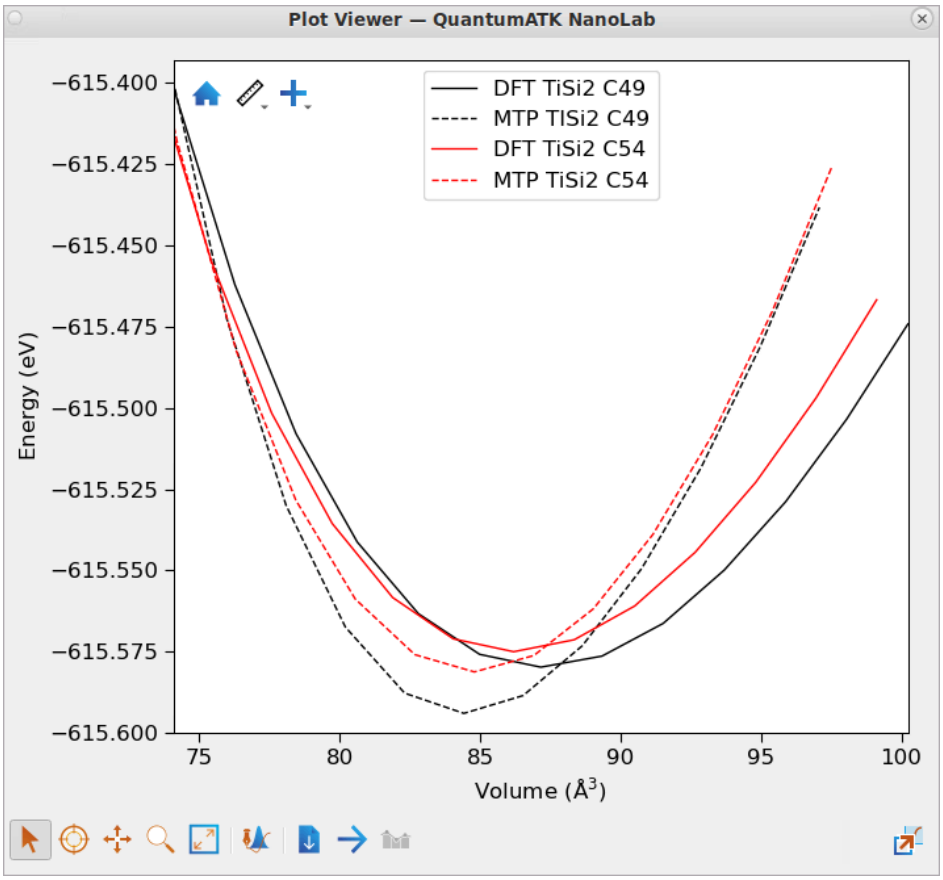
MTP Validation for Amorphous

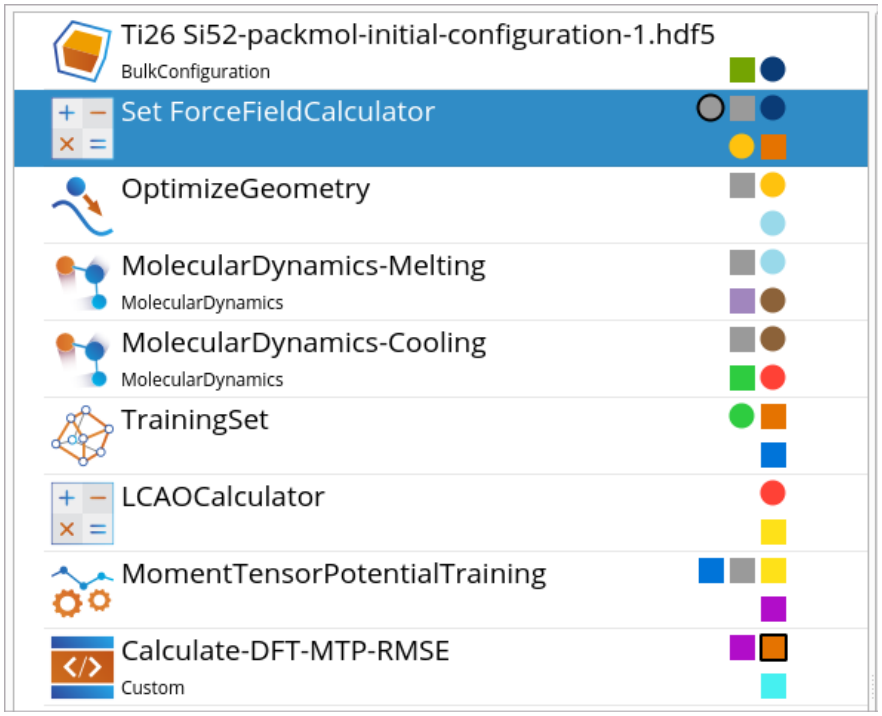
To validate the trained MTP for amorphous structures, one can use the trained MTP to run some short melt-quench MD simulations (50 ps melting and 50 ps cooling) on, e.g., $Ti_{26}Si_{52}$, take some ~10 snapshots from each and recalculate these with DFT and calculate the energy, forces and stress RMSE between MTP and DFT.

1. Download `am-TiSi-MTP-validation.hdf5` workflow and open it in the **Workflow Builder**.
2. In the *Set ForceFieldCalculator* block, specify the location and name of the final trained MTP, `MTP_TiSi_basis400_final.mtp`, which will be used for the validation.

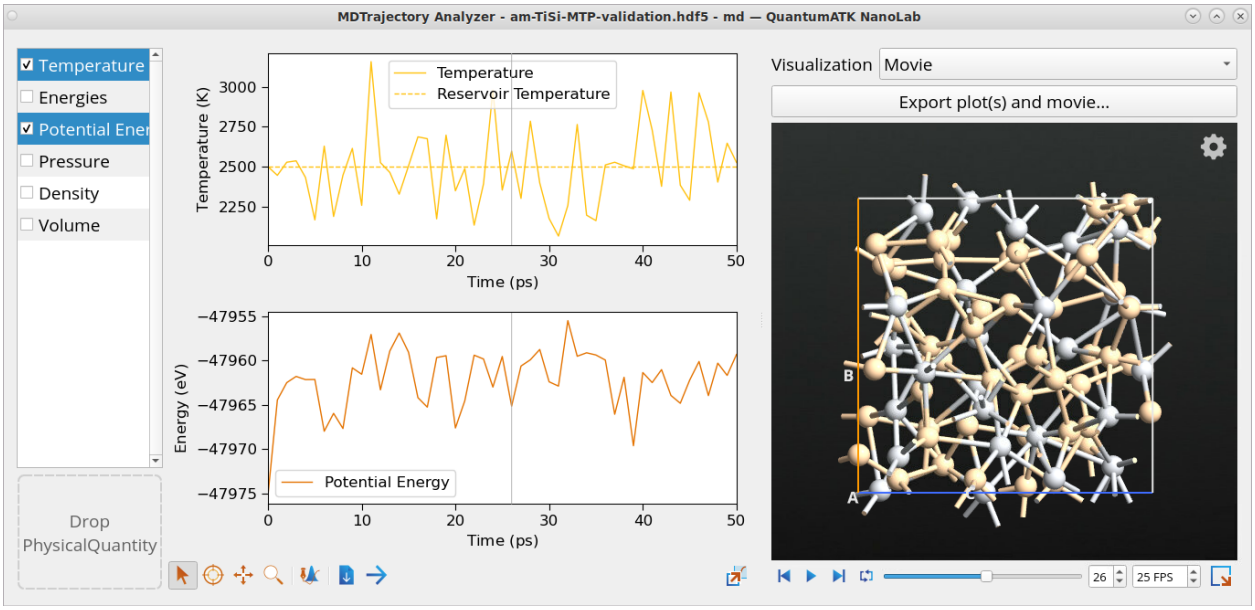
   Run the workflow and once the calculation is finished, we are ready to analyze the results.
3. Open the `am-TiSi-MTP-validation-results.log` and at the end of the file inspect the calculated energy, forces and stress RMSE between MTP and DFT. Small RMSE values shown in the figure below indicate a good accuracy of the trained MTP.

```
+-------------------------------------------------+
| Energy RMSE: 0.0172973844972 eV                 |
| Forces RMSE: 0.231592088227 eV/Ang              |
| Stress RMSE: 0.0238206661876 eV/Ang**3          |
+-------------------------------------------------+
```


Plot Viewer — QuantumATK NanoLab
DFT TiSi2 C49
MTP TISi2 C49
DFT TiSi2 C54
MTP TiSi2 C54
Energy (eV)
−615.400
−615.425
−615.450
−615.475
−615.500
−615.525
−615.550
−615.575
−615.600
75
80
85
90
95
100
Volume (Å^3)


Ti26 Si52-packmol-initial-configuration-1.hdf5
BulkConfiguration
Set ForceFieldCalculator
OptimizeGeometry
MolecularDynamics-Melting
MolecularDynamics
MolecularDynamics-Cooling
MolecularDynamics
TrainingSet
LCAOCalculator
MomentTensorPotentialTraining
Calculate-DFT-MTP-RMSE
Custom

4.Open the *md* and *md_2* objects from `am-TiSi-MTP-validation-results.hdf5` using the *Movie Tool* to inspect MD trajectories from the 50 ps MD melting and from 50 ps cooling simulations, respectively.



Note: It is recommended to check the MD trajectories with the trained MTP to ensure that the trajectories look as expected and appear physical. In this case, check that the amorphous configuration does not crystallize during the cooling part of the simulation. In case of unexpected crystallization, one can consider increasing the size (number of atoms) of the amorphous training configurations and/or add additional amorphous configurations from Active learning to the training data set.

Summary and Outlook

In this introductory tutorial, you have learnt how to set up and analyze MTP training and MTP quality validation calculations for crystal and amorphous materials. In order to adapt the introduced MTP training protocol for different geometries (surfaces, interfaces, nanoparticles), stoichiometries, element composition and other materials, you need to explicitly include relevant training data for such structures.

Depending on the intended MTP application, consider also validating these properties against the reference DFT values:

- Elastic constants, bulk modulus, phonon bandstructure.
- RDF and ADF for amorphous structures (it requires some long DFT-MD simulations, which is expensive).
- Experimental properties that are relevant for the application, if available, but note that since the MTP is trained to reproduce DFT results it can only be as accurate as the underlying DFT model.

References

3.1.4 MRAM workflow in QuantumATK: Study of STT-MRAM free layer stability

Version: U-2022.12

Downloads & Links

MRAM builder Basic QuantumATK Tutorial ATK Reference Manual Vampire

This video tutorial will introduce the QuantumATK interface to the Vampire code and use it to calculate the free layer stability factor in a STT-MRAM magnetic tunnel junction structure. The stability factor is obtained from atomistic spin dynamics simulations using Vampire with input parameters obtained from density functional theory (DFT).

Contents

•*MRAM workflow in QuantumATK: Study of STT-MRAM free layer stability*
–*Video*
–*Introduction*
–*Workflow for calculating the free layer stability in a STT-MRAM MTJ structure*
–*Vampire*
–*References*

Video

Introduction

Spin transfer torque magnetic random access memory (STT-MRAM) is currently being used as standalone and embedded MRAM in modern technology nodes. Further enhancement of the application of STT-MRAM as last level cache requires optimization of the performance and scalability of the magnetic tunnel junction (MTJ) stacks comprising the reference and free magnetic layers as well as additional tunnel barrier and capping layers. The main parameters to optimize in a STT-MRAM are

•**Reading**. This is determined by the tunneling magneto resistance (TMR), which can be calculated using DFT-NEGF in QuantumATK.

•**Writing**. The writing operation depends on multiple parameters such as spin transfer torque, magnetic moment of the free layer, and Gilbert damping. These quantities can also be studied with QuantumATK.

•**Storage**. The storage is characterized by the stability factor $\Delta = \frac{\Delta E}{k_B T}$, where ΔE is the anisotropy energy at the operating temperature T. This is the topic for this tutorial.

Engineering the MTJ to co-optimize the three main requirements of reading, writing, and storage is an active area of research and development.

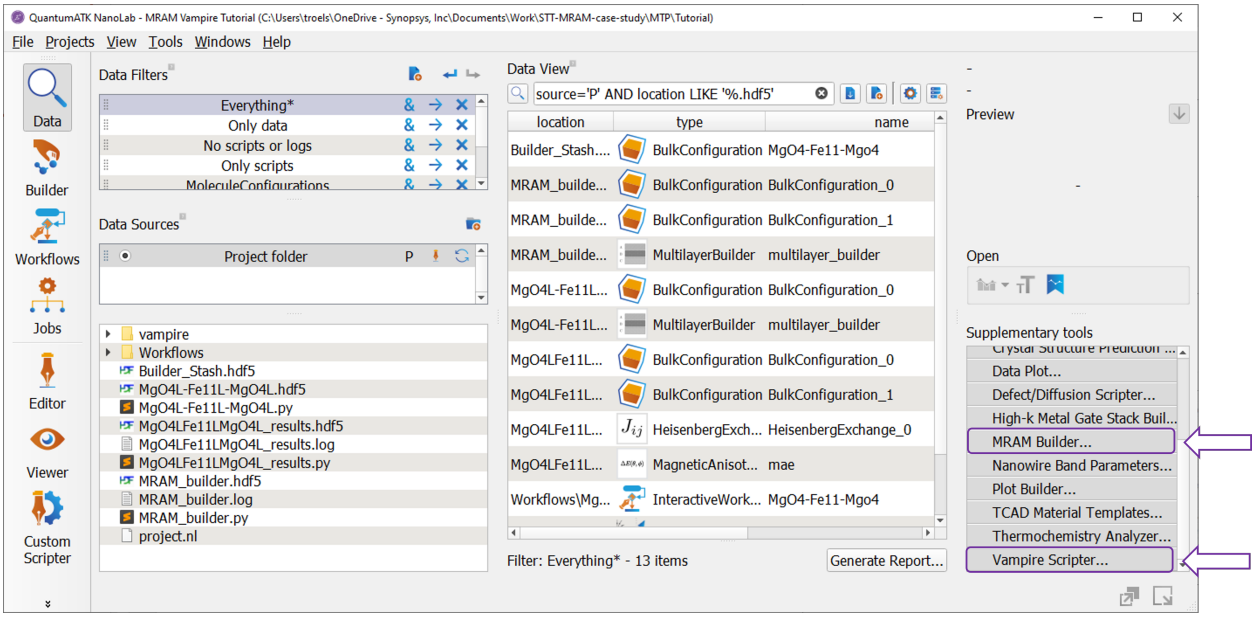
In this video tutorial we demonstrate a work-flow for calculating the stability factor based on atomistic calculations. Below we summarize the main steps in the work flow, but leave the details to the video.

Workflow for calculating the free layer stability in a STT-MRAM MTJ structure

In the tutorial we adopt the following workflow

- Setup the structure using the **MRAM Builder** and perform structural relaxation with machine-learned forcefield of the moment tensor potential (MTP) type.
- Calculate Heisenberg exchange coupling parameters and uniaxial anisotropy energies with the *HeisenbergExchange* and *MagneticAnisotropyEnergy* objects using DFT.
- Setup and run Vampire simulations using the **Vampire Scripter** and analyze the results.

The figure below shows where to find the **MRAM builder** used for building the MTJ structure, and the **Vampire Scripter** used for setting up the Vampire simulations.



Vampire

The Vampire simulation tool enables you to perform a range of different atomistic spin dynamics simulations. Common to them all is the basic description of the spin Hamiltonian of the system. While parameterizations of the spin Hamiltonian exist for bulk materials, it is important to note that several of these parameters change significantly in nano-structured systems and in particular at interfaces between different materials. It it thus advantageous to couple DFT calculated parameters with the Vampire simulations.

Spin Hamiltonian

The starting point for atomistic spin dynamics simulations performed in Vampire is the spin Hamiltonian

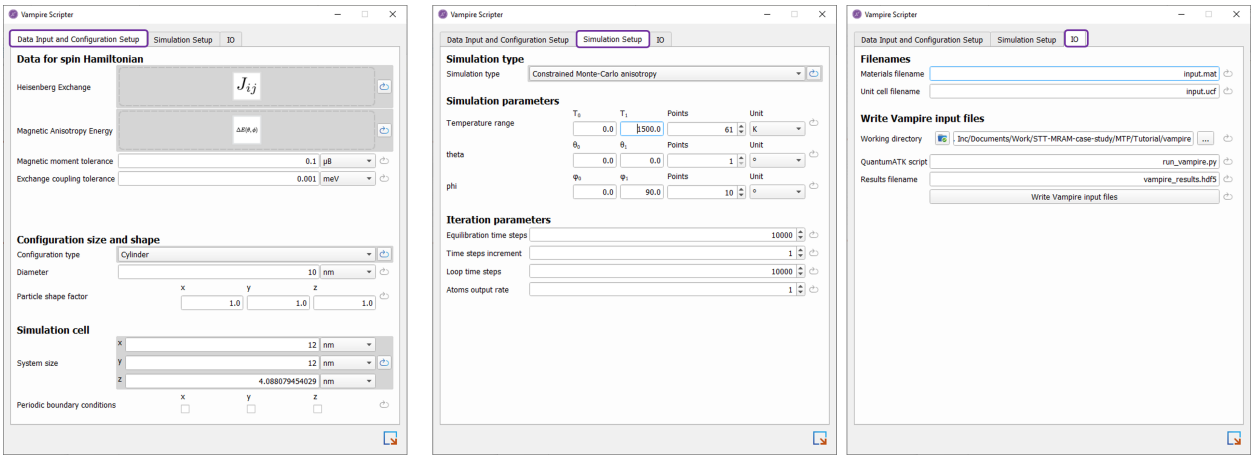
$$H = -\sum_{i<j} J_{ij}\,\mathbf{S}_i \cdot \mathbf{S}_j - \sum_i k_i (\mathbf{S}_i \cdot \mathbf{e}_z)^2 - \sum_i \mu_i\,\mathbf{B}_{app} \cdot \mathbf{S}_i$$

where J_{ij} are the Heisenberg exchange coupling parameters that can be calculated in QuantumATK using the *HeisenbergExchange* analysis object. The spin vector $\mathbf{S}_i$ is a normalized vector with the local spin moment direction on atom i, and k_i is the uniaxial anisotropy energy for atom i, which can be calculated in QuantumATK using the *MagneticAnisotropyEnergy* study object, and $\mathbf{e}_z$ is a unit vector along the z-direction. B_{app} is an externally applied field and μ_i is the local spin moment of atom i.

More information can be found on the Vampire home page and in refs.[1] and[2]. Please cite these papers appropriately if using Vampire in scientific publications.

Vampire scripter

When opening the **Vampire Scripter**, you need to drag-and-drop a *HeisenbergExchange* onto the first tab in the scripter. This is always required. For magnetic anisotropy energy calculation, you should also provide a *MagneticAnisotropyEnergy* object since this is responsible for the uniaxial anisotropy constant giving rise to to the anisotropy energy. The first tab is also where you specify the configuration size and shape as well as the simulation cell.



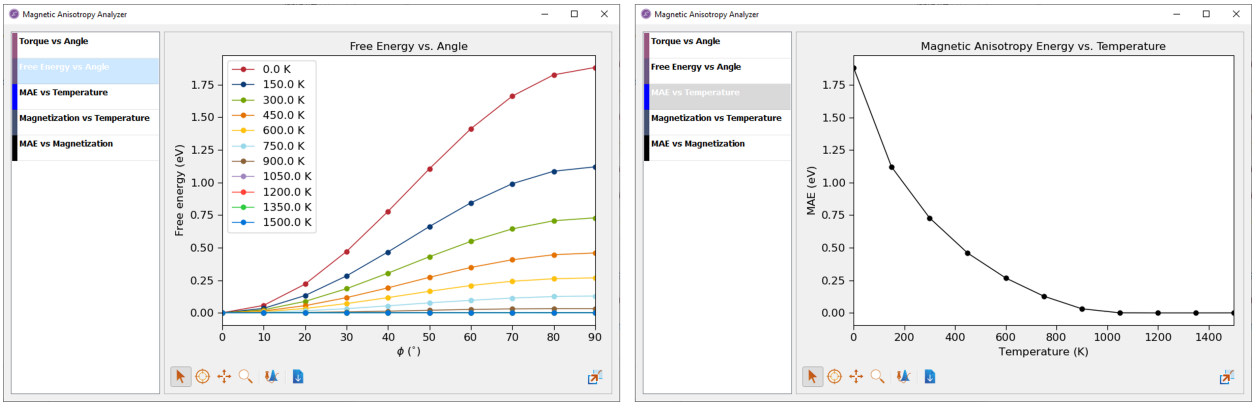
In the second tab you specify the simulation type and various parameters, while in the third tab you specify filenames and where to run the simulation.

[1] RăFăL Evans, WăJ Fan, PăChureemart, TăA Ostler, MăOăA Ellis, and RăW Chantrell. Atomistic spin model simulations of magnetic nanomaterials. *Journal of Physics: Condensed Matter*, 26(10):103202, feb 2014. URL: https://dx.doi.org/10.1088/0953-8984/26/10/103202, doi:10.1088/0953-8984/26/10/103202.

[2] P.ăAsselin, R.ăF.ăL. Evans, J.ăBarker, R.ăW. Chantrell, R.ăYanes, O.ăChubykalo-Fesenko, D.ăHinzke, and U.ăNowak. Constrained Monte Carlo method and calculation of the temperature dependence of magnetic anisotropy. *Phys. Rev. B*, 82:054415, Aug 2010. URL: https://link.aps.org/doi/10.1103/PhysRevB.82.054415, doi:10.1103/PhysRevB.82.054415.

Anisotropy energy

Using the constrained monte-carlo (CMC) anisotropy method in Vampire[Page 333, 2], it is possible to calculate the anisotropy energy $\Delta E(T)$, as shown in the tutorial. This differs from the uniaxial anisotropy energy, ΔE_0, obtained from the *MagneticAnisotropyEnergy* study object by being evaluated at a finite temperature, whereas ΔE_0 is a zero-temperature result. Also, the $\Delta E(T)$ obtained from the Vampire simulation is obtained from the large cylindrical structure, whereas the ΔE_0 from the *MagneticAnisotropyEnergy* study object is obtained from the unit cell. There is thus an area factor difference such that at 0 K we thus expect $\Delta E(0K)/\Delta E_0 \approx \pi R^2/A_{uc}$, where A_{uc} is the unit cell area.



The figure shows two of the analysis tabs in the the **Magnetic Anisotropy Analyzer** used for visualizing and analyzing the result of the Vampire CMC anisotropy simulation. From the MAE value at 300 K $\Delta E(300K) = 0.729\,\mathrm{eV}$ we manually obtain the magnetic stability factor for the 10 nm diameter cylinder

$$\Delta_{10nm} = \frac{\Delta E(T)}{k_B T} = \frac{0.729eV}{0.0258} = 28$$

For 10 years of retention, one typically requires $\Delta = 60$. Assuming that ΔE is proportional to the area, this would require a cylinder with a diameter of

$$\frac{\Delta = 60}{\Delta_{10nm}} = \frac{\pi D}{\pi(10\,\mathrm{nm})^2} \Rightarrow D = 10\,\mathrm{nm}\sqrt{\frac{60}{28}} \approx 15\,\mathrm{nm}$$

References

3.1.5 Generating A Magnetoresistive RAM (MRAM) Stack using the MRAM-Builder

Version: U-2022.12

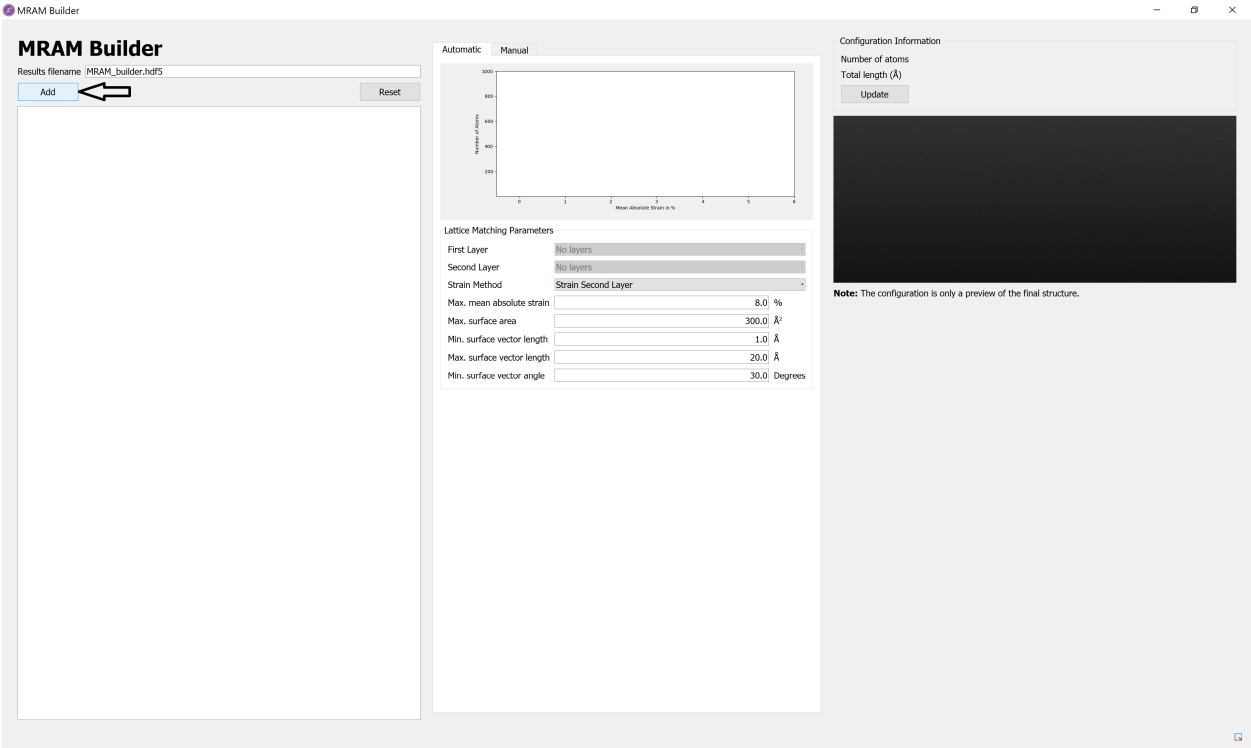
Introduction

Magnetoresistive RAM has become of increased interest for memory technology in the recent years. This device uses magnetic states to store information. It writes and reads electronically, hence enables to integrate in semiconductor chips. Understanding the configuration at the atomic level will help to engineer different properties i.e. *MagneticAnisotropyEnergy* (MAE). To understand the correlation between changes in the atomic structure and stack composition, it is often needed to create multiple configurations. However constructing such configurations at DFT level is very time consuming. Here we use moment tensor potential (MTP) which is a machine learned interatomic potential, to construct such configurations. Machine learned interatomic potentials (MLIPs) have emerged popular in recent years since they can provide ab initio level of accuracy within a realistic timescale, even for relatively complex materials. More information about the MTP can be found here .

Workflow to generate the MgO-FeCo-MgO MRAM structure

We begin by setting up the atomic structure of a MgO-FeCo-MgO structure. Start by opening the **MRAM Builder**. The MRAM Builder can be found in the **Nanolab**. In the MRAM Builder do the following (see image below):

•Press the **Add** icon. This will add new layers.

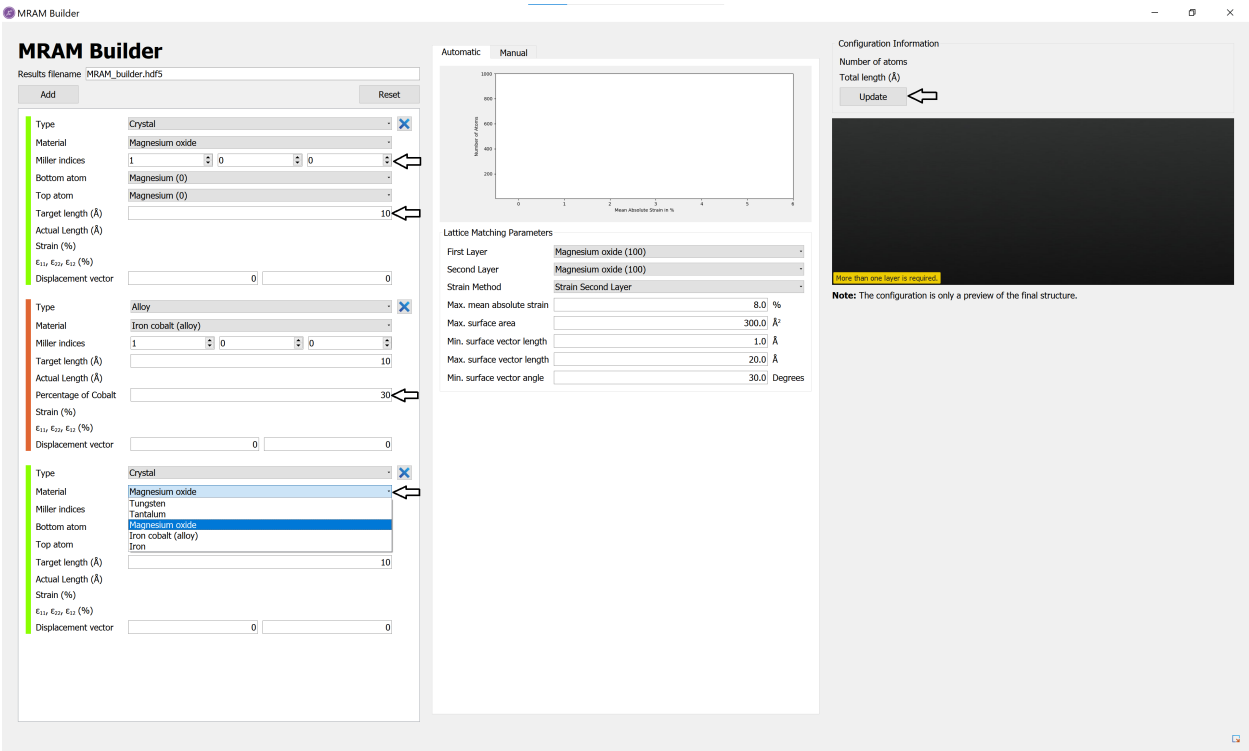


In the layers you can do the following changes (see image below):

•You can change the **material**, **miller indices** and the **thickness** of the layer.

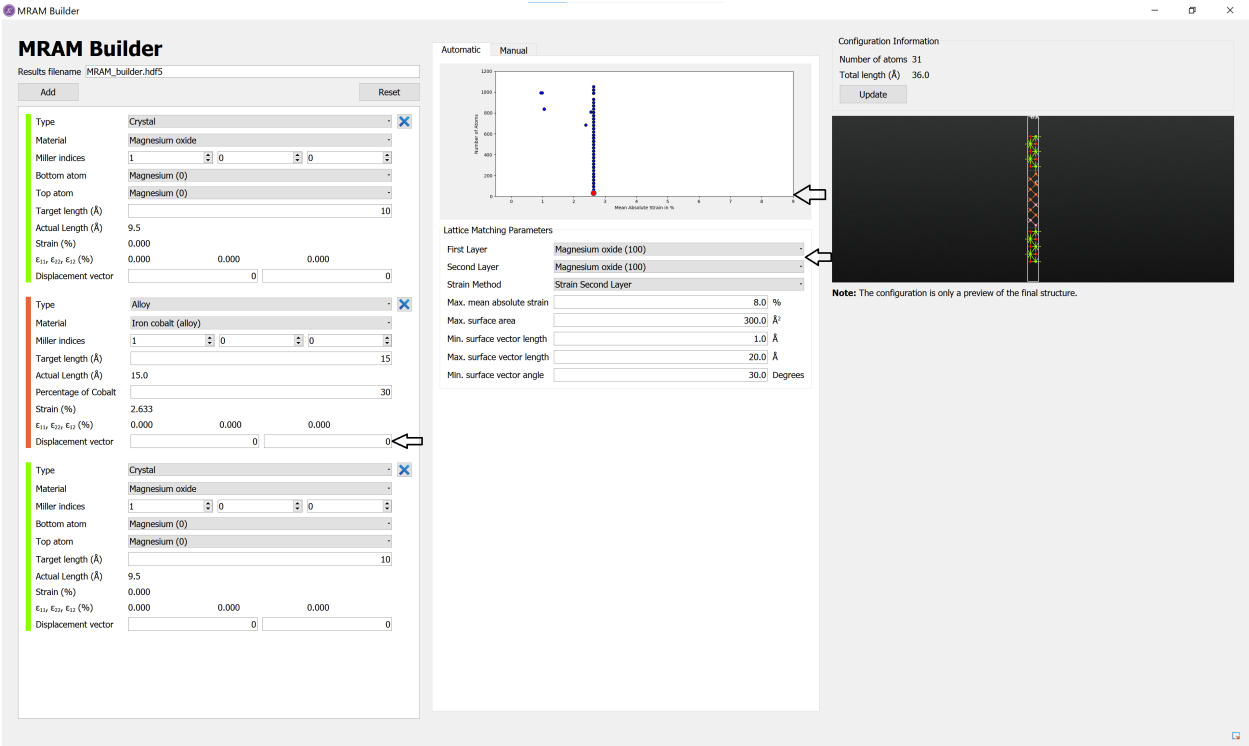
•If the material type is alloy then you can select the **percentage of Cobalt**.

•After the changes you need to click **Update**. This will create multiple stack candidates with different ratio of strain vs. number of atoms.



Next you can do the following steps:

- you can select the configuration based on mean average strain and number of atoms from the **mean average strain vs number of atoms** plot.
- You can also change the **Displacement vector** with the mouse wheel or arrow keys to move a certain layer laterally in the AB-plane. This can be used to control the interface alignment, e.g. which atoms above each other. This will help to create a better initial configuration to obtain configurations with less defects.
- You can modify the strain method as well as the layer that you want to strain from **Lattice Matching Parameters**.



Send the calculation to the **Job Manager** using the icon and run the calculation. If needed some modification in the script, you can generate a script using the **Send-to** button and run the calculation.

Note:

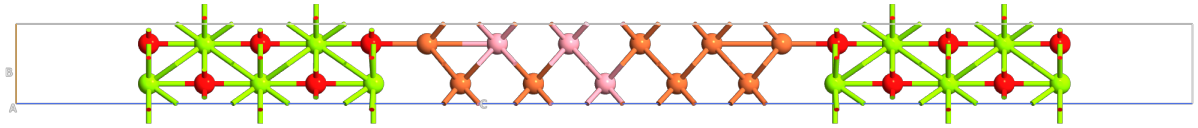
- The builder for this type of stack is primarily designed to work with *CrystalLayer* and *AlloyLayer* types. Pre-trained MTP models for Fe-MgO, FeCo-MgO, Ta-Fe, Ta-FeCo, W-Fe, W-FeCo interfaces and FeCo alloy are provided in the MRAM builder. Other interfaces can be used in scripting, if a suitable interface calculator is specified. This could be a DFT calculator, a force-field calculator, or a user-trained MTP model.
- The `relaxation_length` and `interaction_length` can be tuned according to the need of the interface. If the layer thickness is less than the `relaxation_length` then the optimization will not be done for the interface. By default, interfaces are not annealed, but this can be activated if needed.

Tip:

- To run the MultilayerBuilder efficiently in parallel we recommend using the OpenMP parallelization.
- If you have multiple alloy layers in the stack, you should provide different values for the `random_seed` parameter in *AlloyLayer*. This will ensure that all the alloy layers in the stack are different.
- If you increase the thickness of the layer from the default values, increasing the `relaxation_length` and `interaction_length` parameters in the script can help to achieve better configuration. For more details on these parameters please check the *MultilayerBuilder* study object.

The final MRAM stack configuration can be accessed using the `configuration()` method in *MultilayerBuilder*. A MgO-FeCo-MgO MRAM stack obtained from such calculation is shown below. Download the provided script (`MRAM_builder_example.py`) to generate such a configuration. This configuration is now ready for further DFT

calculation for different electronic and magnetic properties. One can also use it as a starting configuration to manually add defects or dopant atoms in the Builder to investigate their effect on the magnetic properties.



3.1.6 Generating A High-k Metal Gate Stack Using the HKMG-Builder

Version: U-2022.12

Introduction

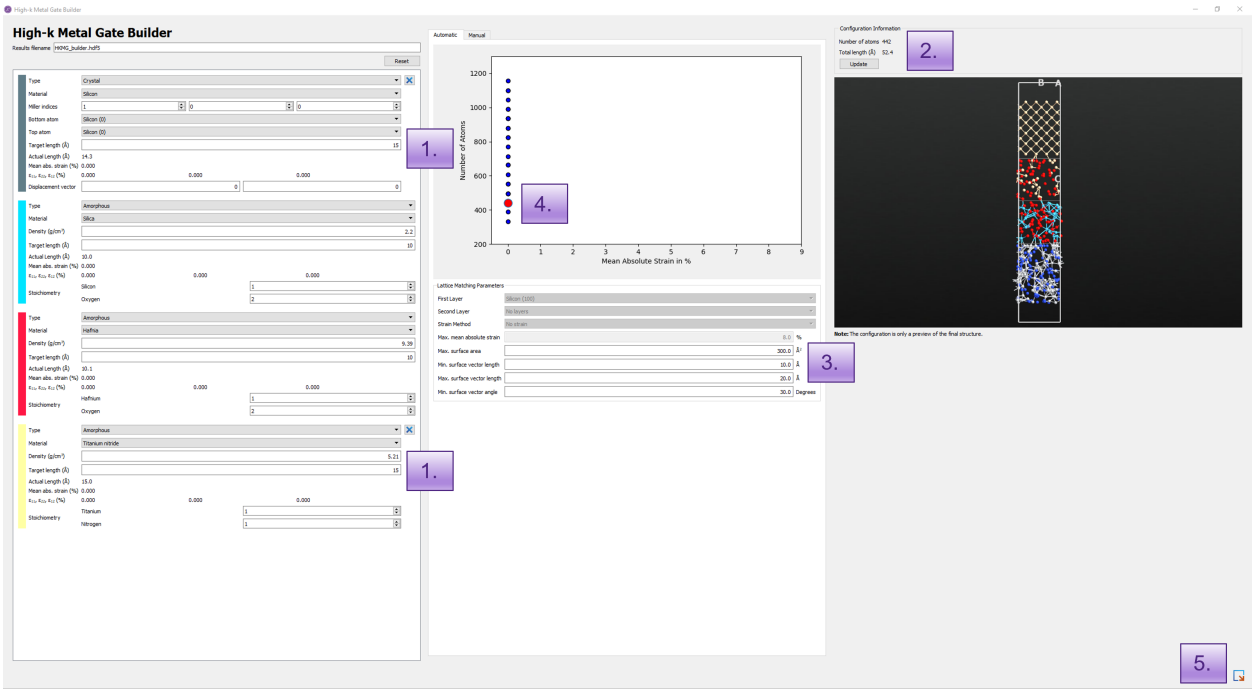
HKMG stacks are complex device building blocks used in transistor gates in modern logic applications. They consist of layers of different materials in different states (e.g. crystalline, poly-crystalline, amorphous). Understanding these stacks at the atomic level will help to tune electronic and structural properties. However, constructing and structurally optimizing such stacks at the DFT level requires considerable computational resources. Instead, we use moment tensor potentials (MTP) which are machine learned force fields (ML-FFs). ML-FFs have become popular in recent years since they can provide *ab-initio* level of accuracy within a realistic timescale. More information about MTP can be found here.

Workflow

Step 1: Prepare the script using the builder

Open the **High-k Metal Gate Stack Builder** found in the *Supplementary tools* section of **NanoLab**.

The builder is pre-populated with a four-layer stack consisting of the semiconductor substrate, an insulating oxide, the high-k material and finally a metallic layer. Different choices of materials are available and all layers can be selected either as crystalline or amorphous. In the case of amorphous layers, the stoichiometry may be specified. For this tutorial, the default settings will not be changed except for the lengths of the outer layers which are considered to be bulk-like. The builder produces a script which will generate the actual HKMG-stack when run. To obtain the script, follow the steps below.



1.Change the target length of silicon and titanium nitride to 15 Å.

2.Click *Update* to show the stack preview.

3.Increase the min. surface vector length to 10 Å to filter out small unit cells.

4.Select the third stack from the bottom of the size-strain plot. This stack has a square base.

5.Generate the script by clicking on and sending to the **Editor** or download it here: `HKMG_builder_original.py`.

Tip:

•The displacement of the layers in the AB plane can be adjusted by clicking into the **Displacement vector** field and using the mouse wheel or arrow keys. The updated interface will be shown in the preview after pressing enter. This is particularly useful for aligning atoms in crystal-crystal interfaces and reducing defects.

Step 2: Get candidate structures by running the script

The generated script has to be run in order to build the final stack structure. As an alternative, the preview structure can be sent to the **Builder** to be inspected and manually modified. However, especially for amorphous layers, it is strongly recommended to refine the initial structure using the *MultilayerBuilder*, which will take care of optimizing interfaces and properly annealing amorphous layers.

Tip:

•To run the MultilayerBuilder efficiently we recommend using OpenMP parallelization.

•For layers of a certain thickness it might be advisable to change the *relaxation_length* and *interaction_length* arguments for better results.

Since generating the amorphous layers is a stochastic procedure, it is not guaranteed that the result of a single script run will yield a stack with satisfactory properties. In the following we will therefore build several candidate stacks by repeating the layer set-up and the *MultilayerBuilder* run in a loop using different random seeds. Take a look at the modified script (`HKMG_builder_loop.py`) and optionally change the sample size or the random seed. Then run the script. This step might take several hours on a multi-core workstation.

Note: Validated, ready-to-use MTPs are provided for all material and interface combinations available in the builder. However, user generated potentials can be substituted in the script.

Step 3: Select a high-quality stack from the set of candidates

Now that we have a set of initial candidate structure from step 2, we can proceed to select the best stack configuration. The most important quality criteria to consider are unphysical bonding patterns and various forms of (coordination-) defects formed in particular at the interfaces.

Download the provided script (`selectHKMGconfigurations.py`) which first checks for too close atoms. Then, it passivates dangling Si-bonds at the Si-SiO_2 interface by inserting additional hydrogen and oxygen atoms. Finally it filters out stacks with undesired structural patterns, such as O-O bonds, as well as clearly over- and under-coordinated atoms. Run the script in the same folder containing the builder output to receive a list of low-defect stacks ranked by the number of added saturating hydrogen atoms.

3.1.7 How to select the right calculator

A calculator is the engine that performs the atomic simulations, defining how we describe the interactions of the electronic system. It both represents a level of theory (full quantum mechanical approach, semi empirical approach, or classical approach) and algorithmic settings.

In this tutorial, you will receive a brief overview of the different calculators and learn to select the calculator that best fits your needs in *QuantumATK NanoLab*.

Table of Contents

- *The Calculator types*
- *The DFT Calculators (LCAO and Plane Wave)*
- *The Semi Empirical Calculator*
- *The Force Field Calculator*

The Calculator types

When setting up a calculation in the *Workflows* tool ( ) you must select a calculator to perform the atomic scale modelling.

	Quantum Mechanical (DFT) Calculators
DFT: LCAO *DFT: Plane Wave*	The two quantum mechanical calculators both model the electronic properties within the framework of density functional theory (DFT). These DFT calculators provide the most accurate results, but require the largest computational resources. It is recommended to use these when the accuracy of the results is of foremost concern.
	Semi Empirical Calculator
Semi Empirical	The semi empirical calculator is also based on DFT, but it approximates the electron-electron interactions, using experimental data. Due to these approximations the semi empirical calculator is orders of magnitudes faster than the quantum mechanical calculators, but the accuracy is correspondingly reduced. The semi empirical calculator is often used when qualitative results will suffice, or when a DFT calculator cannot be used.
	The Classical Calculator
Force Field	The force field calculator, is based on classical mechanics and estimate the forces between atoms using a range of element-pair specific potentials. The force field calculator is the fastest calculator, orders of magnitudes faster than the semi empirical calculator, and is often the calculator of choice when dealing with huge systems (many thousands of atoms) or when modelling the full dynamic evolution of the system on longer time frames (molecular dynamics).

The DFT Calculators (LCAO and Plane Wave)

Both the plane wave calculator and the LCAO calculator solve the time independent Kohn–Sham equations. The main difference is that the LCAO calculator employs numerical LCAO (**L**inear **C**ombination of **A**tomic **O**rbitals) basis sets, while the plane wave calculator employs a PW (**P**lane **W**ave) basis set.

Neither of the two DFT calculators require any system specific input, and can in general be expected to produce results of similar quality, independent of the system. A basis set is required for each element species in the system under investigation (for plane wave, the basis set is only used for initialization and projections). Nearly all basis set provide support for all elements, with only a few exceptions.

The Plane Wave Calculator

The plane wave bases incorporate the periodicity of the bulk system into the basis set, which allows a higher accuracy to be obtained. The plane wave calculator is the state of the art calculator, though it is often slow and can only be applied to bulk systems. *Introducing the QuantumATK plane-wave DFT calculator*

See *DFT: Plane Wave* for more details.

The plane wave calculator is suitable for treating system sizes on the order of hundreds of atoms.

The LCAO Calculator

In general, it is recommended to use the LCAO calculator, which can work with both molecules and bulk systems. In most cases, an accuracy close to the plane wave calculator can be achieved at significantly reduced computational effort. The accuracy of the LCAO calculator can be improved even more by using a larger basis set, at the cost of longer computation times. See *DFT: LCAO* for more details.

The LCAO calculator is suitable for treating system sizes on the order of thousands of atoms.

Note: Computational effort for the DFT calculators depends heavily on the exchange correlation functional and calculator settings.

The Semi Empirical Calculator

The semi empirical calculator requires a parameter set.

QuantumATK NanoLab includes a selection of parameter sets designed for a range of purposes. However, each parameter set is only constructed for a subset of elements on the periodic table. Thus, it may be that there is no parameter set which provides support for all elements in the system under investigation.

The semi empirical calculator is suitable for treating system sizes on the order of thousands of atoms.

The is the calculator of choice when performing large scale NEGF calculations (*NEGF: Device Calculators*, see *Transport calculations with QuantumATK*)

See *Semi Empirical* for more details.

The Force Field Calculator

The force field calculator requires a potential set, comprised of potential components, which defines the types of atomic interactions to be modeled between different atomic element pairs. Consequentially, the potential set must include support for all elements in the system under investigation.

QuantumATK NanoLab includes a selection of potential sets designed for a range of purposes (See *Force Field* for more details.). However, each potential set is only constructed for a subset of elements on the periodic table. Thus, it may be that there is no potential set which provides support for all elements in the system under investigation. In such a case, there is an advance feature to create a custom potential set, which must be manually configured to handle all relevant atomic interactions between all relevant atomic element pairs.

The force field calculator is suitable for treating system sizes on the order of tens of thousands of atoms.

See *Force Field* for more details.

Bonding and non-bonding potential sets

Roughly speaking, potential sets come in two flavors: rigid and flexible.

Rigid potential sets do not handle molecular bonding, and are thus geared towards modelling non-reactive interactions such as angle-strain (represented by harmonic potentials) or dispersion (represented by van der Waals potentials).

To model chemical reactions, a flexible potential set should be used instead. Flexible potential sets can model the formation and breaking of bonds, allowing the calculator to reproduce reaction chemistry without explicit quantum mechanical considerations.

Machine Learned Potentials

MTPs (**M**oment **T**ensor **P**otentials) are machine-learned potentials that allow dynamics modelling of arbitrary materials systems. The accuracy of the results, as well as what scenarios can be modeled, are dependent on the training data used to produce the MTP. MTPs are suitable to use when large systems (e.g., amorphous materials or multi-layer stacks) need to modeled with near-*ab initio* accuracy.

3.1.8 Using Thermochemistry Analyzer to Compare Chemical Reactions

Version: T-2022.03

Downloads & Links

IdealGasThermoChemistry CrystalThermoChemistry

In this tutorial, we will learn how to use the *Thermochemistry Analyzer* GUI for computing and comparing free energies of chemical reactions. This tool can be used to quickly set up thermochemical calculations of molecules and bulk/surface configurations, import computed thermochemistry data into a database and plot free energy change of reactions.

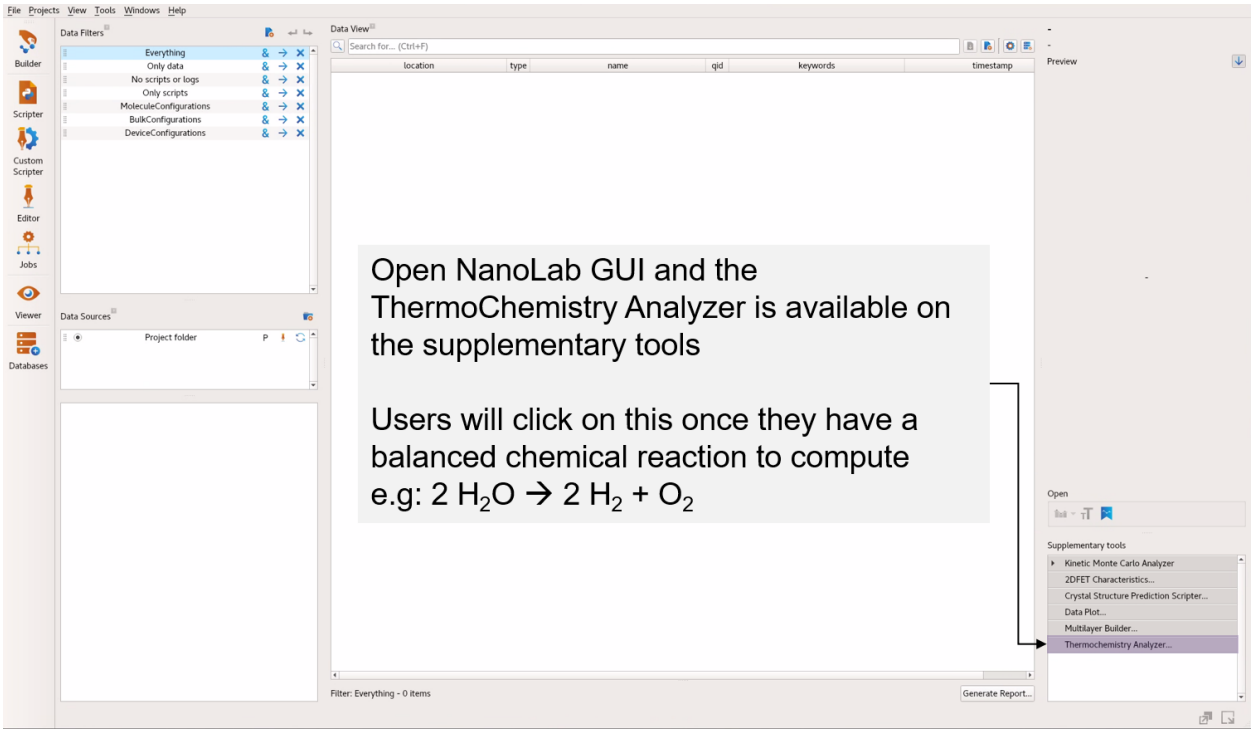
Background

In any process under study, several chemical reactions take place at the same time and some of them may compete with each other. It is interesting to find out which reaction is more dominant under a given process condition. One way to find the dominant reaction from a group of reactions is to compare their reaction free energies. Free energy (G) of a chemical species indicates the total energy available to it to diffuse, vibrate and react with other species in a medium. Free energies of molecules can be computed under the ideal gas approximation while the free energy of bulk/surface systems can be computed under the quasi-harmonic approximation. A reaction constitutes reactants and products, each of which includes 1 or more species. A reaction free energy (G) is then the difference between the combined free energies of the product species and that of the reactant species. The more negative G is, the more favorable a reaction becomes at that condition and vice-versa. It is to be noted that the G changes with process variables like temperature, pressure and concentration.

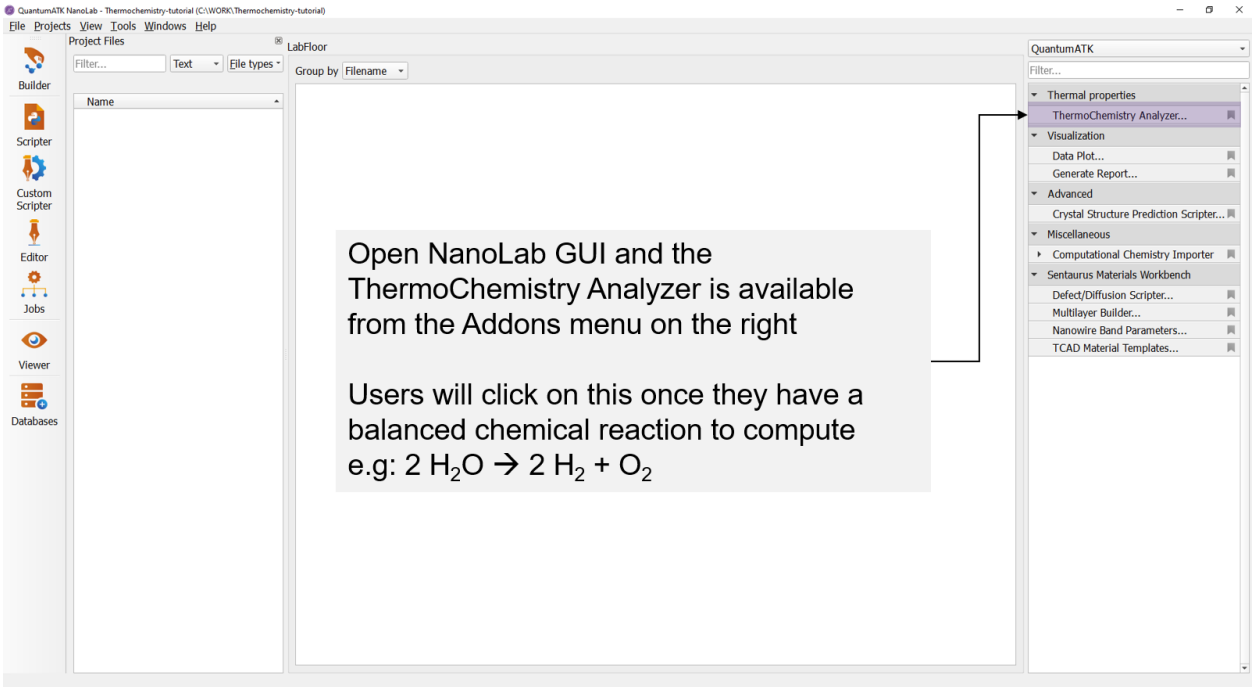
In **QuantumATK**, the *Thermochemistry Analyzer* GUI provides a one-stop-solution to compute the free energies of the participating species in a reaction using Density Functional Theory (DFT) and, ultimately, compare Gs of several reactions. The *Thermochemistry analyzer* comes with a *Thermochemistry Database* to store and extend user-generated thermochemical data of different species so the user can compare reactions whenever they want without having to compute/import the thermochemical data every single time.

Getting started

Open the **QuantumATK** GUI, NanoLab, and click *Thermochemistry Analyzer* from the supplementary tools menu on the right.

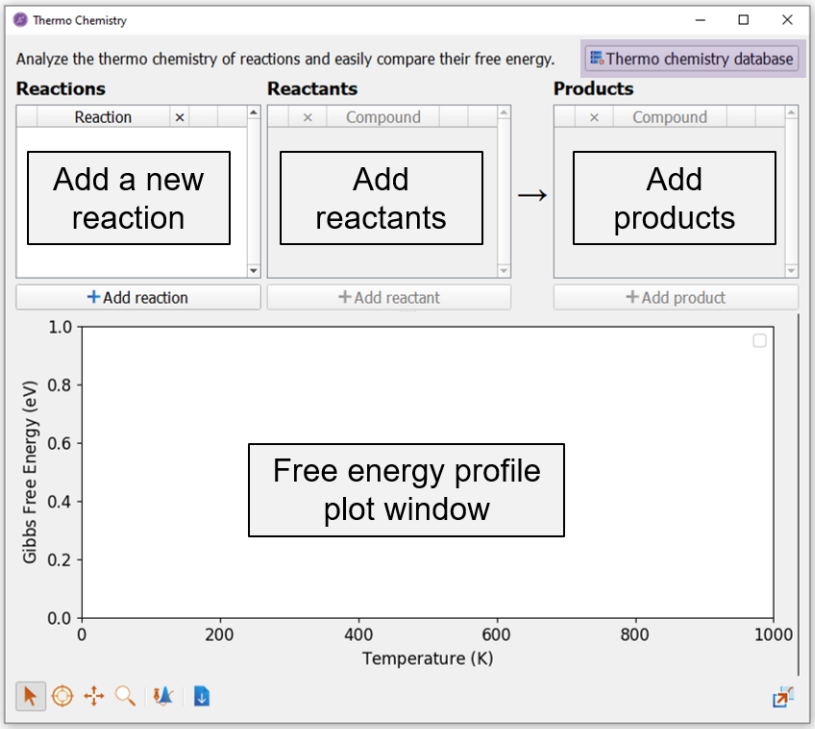


In the S release, it is found in the Addons section to the right.



Understanding the Thermochemistry Analyzer GUI

The GUI of *Thermochemistry Analyzer* is illustrated in the below image.

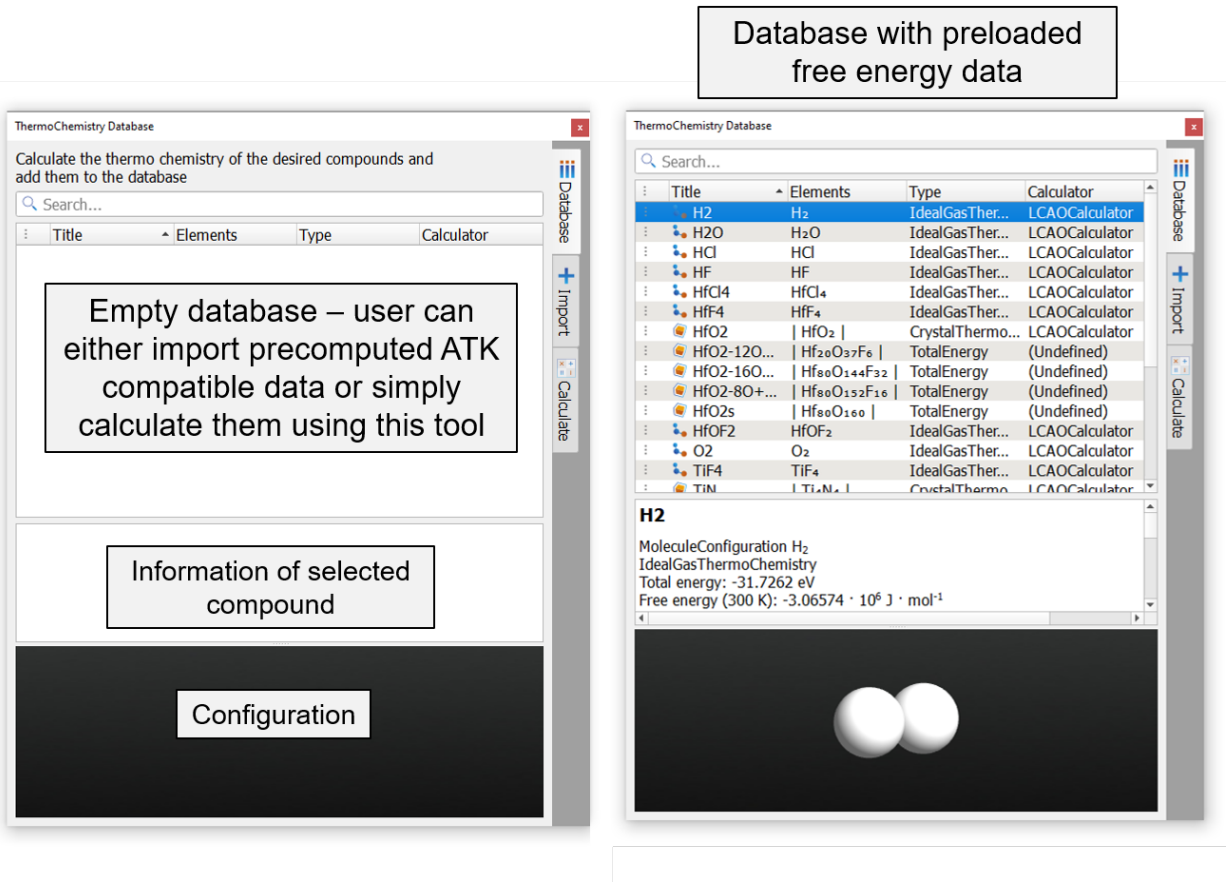


The GUI window has two main regions. The top region is for reaction input, in which the user can add a new reaction by clicking the *Add reaction* button. The next step is to add the reactant and product species of that reaction one-by-one by clicking the *Add reactant* and the *Add product* buttons.

The bottom region is a plot window, which will be updated with curves plotting the free energy of the reaction as a function of temperature. The free energy curves will appear as soon as the user adds the first reactant/product species from the *Thermochemistry Database* and updates automatically as the other reactant and product species are added.

Access to the *Thermochemistry database* is allowed by pressing the button at the top right corner of this GUI window highlighted in purple in the above image.

The below image shows an illustration of the *Thermochemistry Database*.



Thermochemistry Database GUI contains three sub-windows titled Database, Import and Calculate.

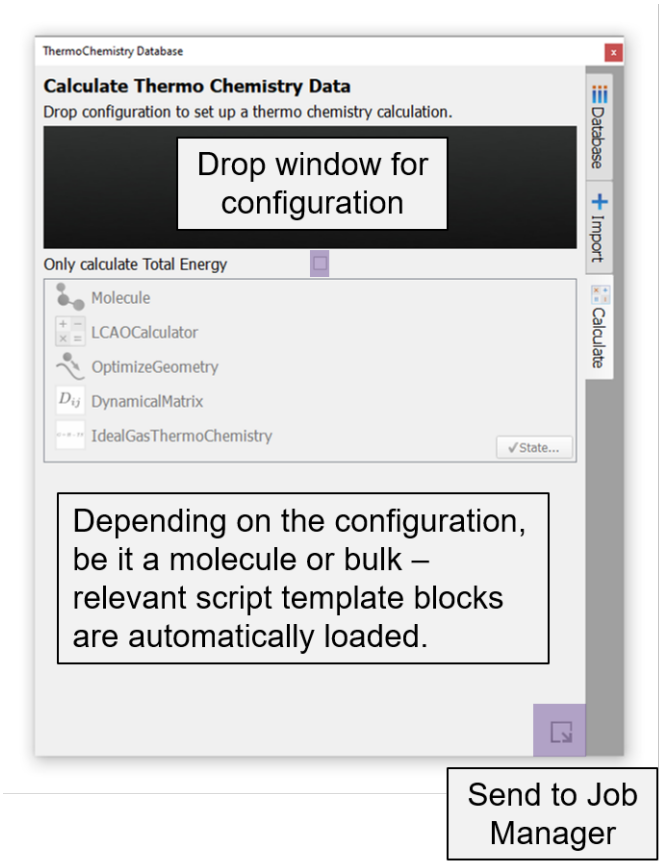
Database: A place to store thermochemistry data

The thermochemistry data of molecules and bulk/surface models are listed in this sub-window. The data is locally stored in .vnl/databases/thermochemistry. User can copy precomputed ATK compatible database entries to the above folder directly. However, we provide precomputed data for a few molecules, bulk material and surface geometries that will appear as shown in the above image.

The database gives the following information of the species: its user defined name, chemical formula, type of data (ideal gas thermochemistry, crystal thermochemistry or total energy) and the type of calculator used. On selecting any of the entry in the database, its geometry will be shown in the configuration window at the bottom and the settings used to compute its electronic structure will be printed in the text box above it.

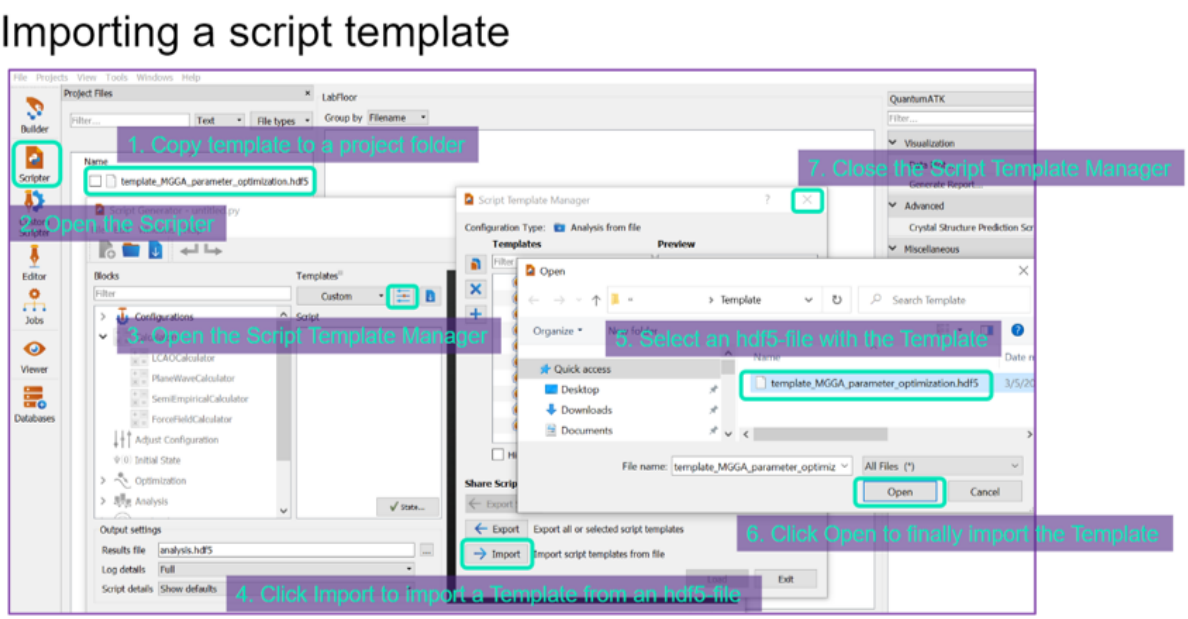
Calculate: Setup free energy calculation of a molecule or bulk configuration

Free energy calculation of a species can be set up using this sub-window.



Here, the user will drag and drop a configuration on to the drop window. Depending on the configuration, molecule or bulk, a corresponding script template to compute free energy will be loaded. This template typically contains by default the *Configuration*, *LCAOCalculator* with PBE XC functional and Pseudo-Dojo high basis set, *OptimizeGeometry* block, *DynamicalMatrix* and *IdealGasThermoChemistry* for molecule configuration (read *Notes*) or *CrystalThermoChemistry* for bulk configuration (read *Notes*). The user can edit the DFT calculator settings, constraints and tolerance for the geometry optimization and dynamical matrix. The default settings are verified to reproduce free energies reported in JANAF tables for several molecules. The user can select the *Only calculate Total Energy* option to replace the DynamicalMatrix and ThermoChemistry blocks in the scripter with a TotalEnergy object. The job can be submitted to a cluster or local computer via JobManager directly from this sub-window.

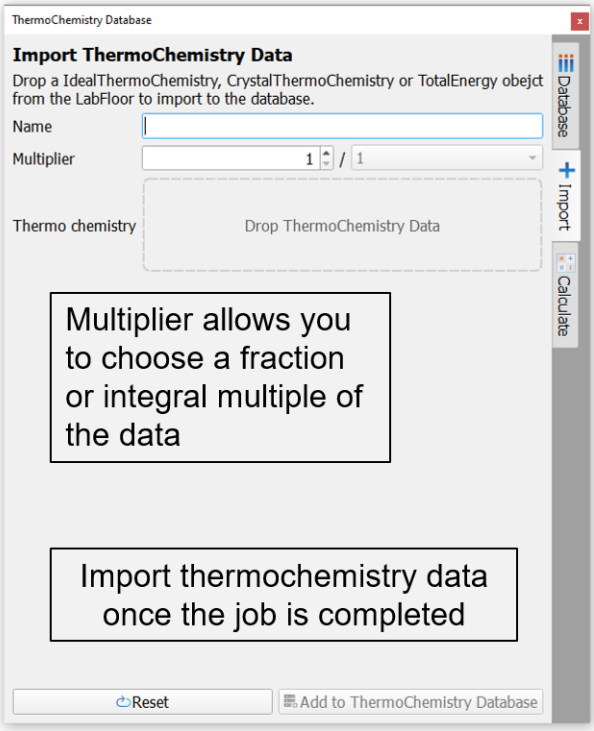
Alternatively, these calculations can also be set up in script generator using the following templates `bulk_LCAO_thermochemistry_template.hdf5`, `molecule_LCAO_thermochemistry_template.hdf5` and `surface_LCAO_thermochemistry_template.hdf5` for bulk, molecule and surface configurations, respectively. When templates are used, the calculator can also be changed. But be careful not to compute free energy using data computed using different calculators. These templates can be loaded to the script generator following the below graphic.



The resulting ThermoChemistry objects from these calculations can be imported to the *Thermochemistry Database* using the next sub-window.

Import

Once the calculations are finished and loaded on the LabFloor, the free energy/total energy data objects can be imported to the thermochemistry database in this tab in 3 simple steps.



(I)Type a suitable name for the configuration (this is the name used in the reaction equation).

(II)Drop the thermochemistry object on to the drop field and choose a multiplier (fraction or integer). For example, if a bulk configuration has 4 stoichiometric units of the compound, then the user can choose 1/4th of the thermochemistry data to correspond to one stoichiometric unit of the material.

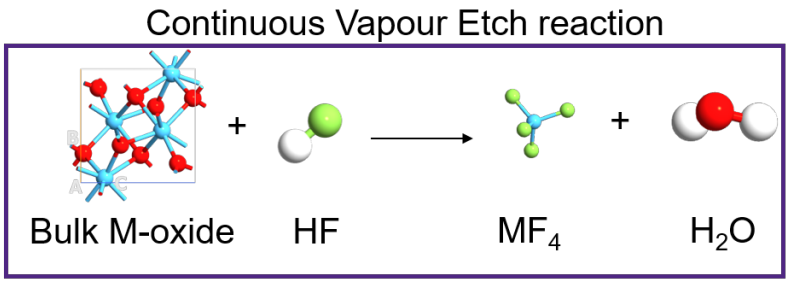
(III)Finally, click *Add To ThermoChemistry Database* button. The entry will appear in the *Thermochemistry Database*.

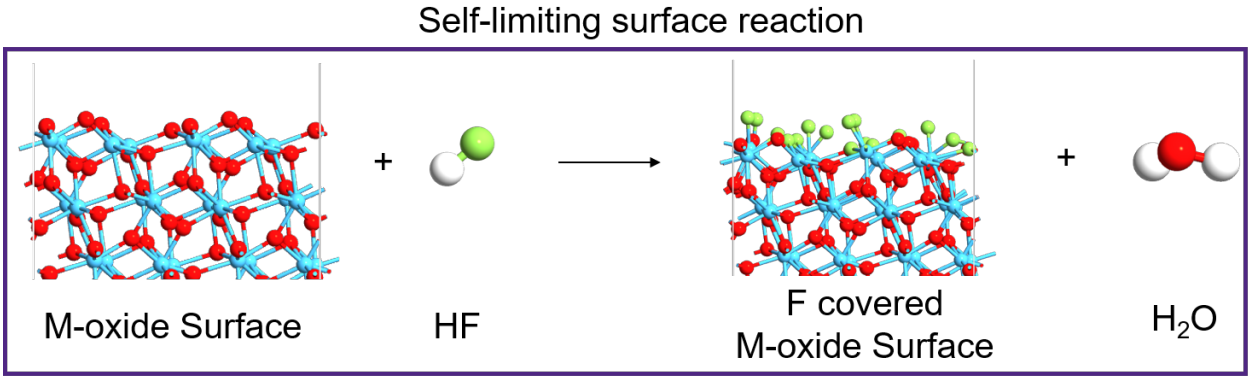
Example: Temperature Window for Thermal Atomic Layer Etching of HfO_2 and ZrO_2

To demonstrate the *Thermochemistry Analyzer*, we will try to reproduce the free energy curves published by Mullins *et al.*[1] in which hydrogen fluoride (HF) gas interacts with either the bulk or surface of HfO_2 and ZrO_2 . We use the same geometries as the ones used in the publication and they are preloaded in the *thermochemistry database* along with their free energy data computed using **QuantumATK** as explained above. For surface geometries, we only use the total energy with the assumption that the free energy contributions from the reactant and product surfaces effectively cancel each other.

For the first reaction, let us consider etching of HfO_2 bulk using HF. The reaction is HfO_2 + HF -> HfF_4 + H_2 O . The below video shows how this can be done using the *Thermochemistry Analyzer* GUI in **Quantum ATK**.

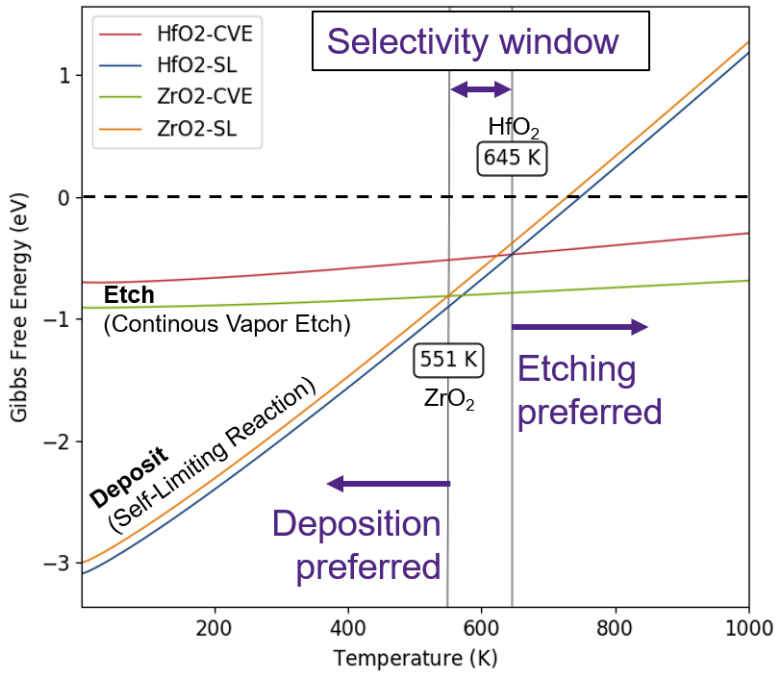
We follow the same procedure and add corresponding reactions for HF interaction with HfO_2 surface, ZrO_2 bulk and ZrO_2 surface and the resulting free energy plot is shown below. We used the data entries from the thermochemistry database titled HfO2s and ZrO2s for the bare HfO_2 and ZrO_2 surface models, respectively and used HfO2-8O+16F and ZrO2-8O+16F for the fluorinated HfO_2 and ZrO_2 surface models, respectively. The reactions studied are described in the image below.





The bulk to gas reaction is termed as continuous vapour etching (CVE) reaction and the surface to surface reaction is termed as self-limiting (SL) reaction.

[1] Rita Mullins, Suresh KondatiăNatarajan, SimonăD Elliott, and Michael Nolan. Self-Limiting Temperature Window for Thermal Atomic Layer Etching of HfO2 and ZrO2 Based on the Atomic-Scale Mechanism. *Chemistry of Materials*, 32(8):3414–3426, 2020.



The free energy profiles (G vs T) of CVE and SL reactions of a material will intersect at some temperature at which both reactions are equally probable. The reaction that has the lowest G value is more favourable at other temperatures. The crossover temperature for HfO_2 and ZrO_2 are found to be 645 K and 551 K, respectively. The corresponding values in the reference paper are 657 K and 534 K, obtained using PAW based plane wave code.

Below 551 K, HF passivates both HfO_2 and ZrO_2 surfaces and starts etching both materials beyond 645 K, of course subject to kinetic barriers. Between 551 K and 645 K, there is a selectivity window where HF passivates HfO_2 but etches ZrO_2 . Please keep in mind that this is only a thermodynamic study and kinetics could still make the reactions difficult to proceed at these temperatures. Nevertheless, they help immensely to screen reactant chemicals and specific reactions for indepth kinetic analysis.

General Uses

Apart from applications to process development, *thermochemistry analyzer* can be used for any general *ab initio* thermodynamic studies such as finding stable phases, reaction paths, gas-sensing of materials and so on. Since total energy objects can be stored in the database, one can also use it to compute and compare binding/adsorption/desorption energies.

3.1.9 Electronic Properties of Phase Change Material $Ge_2Sb_2Te_5$

Version: T-2022.03

In this tutorial, we will investigate electronic properties of $Ge_2Sb_2Te_5$, a phase change material used in non-volatile memory technology, using Density Functional Theory (DFT) calculations. Please follow the tutorial introduction_quantumatk1 to familiarize yourself with the use of scripter to set up calculations. Only the final scripts will be provided in this tutorial.

Geometry

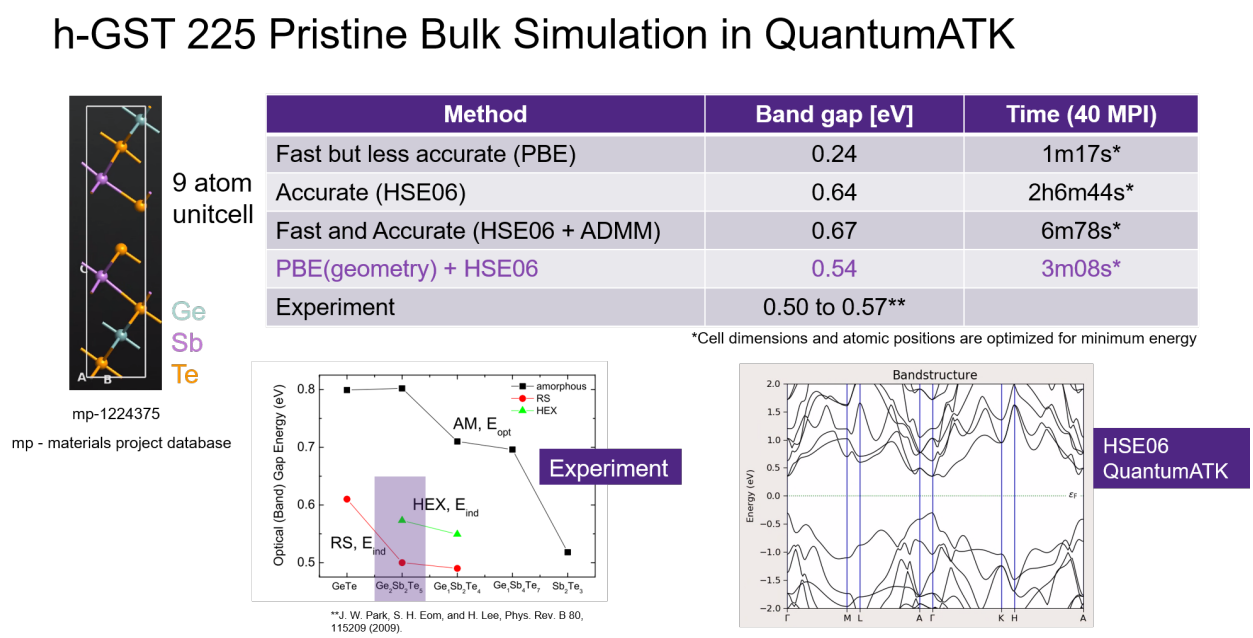
For this tutorial, let us first download the unitcell configuration of $Ge_2Sb_2Te_5$, in a trigonal crystal system with hexagonal lattice symmetry, from the materials project database (mp-1224375) using the Database Tool available in the NanoLab. For reference, the geometry can also be downloaded from here `GST-geometry.py`.

Bandgap Calculation

As the first property, let us compute the bandgap of $Ge_2Sb_2Te_5$ and compare to literature data. To do that, the unit cell geometry will be optimized with space group constraint for minimum energy first and then its BandStructure will be computed. Additionally, we will compare the default GGA (PBE XC functional) results to more accurate HSE06 calculations using the PseudoDojo LCAO basis sets. To be exact, 4 different calculation settings will be used as follows:

- PBE functional based geometry relaxation and BandStructure calculation(`PBE.py`)
- HSE06 hybrid functional based geometry relaxation and BandStructure calculation(`HSE.py`)
- HSE06 hybrid functional with ADMM basis set for both geometry relaxation and BandStructure calculation(`HSE-ADMM.py`)
- PBE functional based geometry relaxation followed by HSE06 hybrid XC functional based BandStructure calculation(`PBE-HSE.py`)

The bandgap can be quickly viewed from the BandStructure object preview window or from the BandStructure Analyzer object. The corresponding results of the above calculations (using 40 MPI processes in 1 node) are given in the below image.



Experimental bandgap of hexagonal $Ge_2Sb_2Te_5$ is known to be between 0.50 and 0.57 eV[1]. PBE only calculation is the fastest but the bandgap is underestimated to 0.24 eV only. Standard HSE06 calculation took over 2 hours and gave a bandgap of 0.64 eV. Inclusion of ADMM basis sets accelerated the HSE06 calculation to completion within 7 minutes

[1] Jun-Woo Park, SeungăHwan Eom, Hosun Lee, JuarezăLF DaăSilva, Youn-Seon Kang, Tae-Yon Lee, and YoonăHo Khang. Optical properties of pseudobinary GeTe, Ge2Sb2Te5, GeSb2Te4, GeSb4Te7, and Sb2Te3 from ellipsometry and density functional theory. *Physical Review B*, 80(11):115209, 2009.

and produced a similar bandgap. However, HSE06 based bandstructure calculation on PBE optimized geometry gave a better bandgap value of 0.54 eV while taking only about 3 minutes.

Lattice Parameters

Lattice parameters of the $Ge_2Sb_2Te_5$ unitcell after geometry optimization using the PBE and HSE06 functionals are given in the below image.

	a / c [Å]	
Method	**QuantumATK**	**Literature**
PBE	4.31 / 17.17	
HSE06	4.27 / 17.28	
Experiment		4.25 / 17.27*

*J. Appl. Phys., 92, 7, 1 (2002)

Both PBE and HSE06 optimized geometries have lattice parameters within 0.1 Å of the experimental values[2].

Cohesive Energies

Cohesive energies of the $Ge_2Sb_2Te_5$ is computed using PBE and HSE06 methods using E_{bulk} - ($2*E_{Ge} + 2*E_{Sb} + 5*E_{Te}$) and the results are shown below. Here, E_{bulk} is the total energy of bulk system and E_{Ge}, E_{Sb} and E_{Te} are atomic reference energies of Ge, Sb and Te, respectively.

	Cohesive Energy (eV/atom)	
Method	**QuantumATK**	**Literature**
PBE	-3.71	-3.75*
HSE06	-4.37	-

*Phys. Rev. B 84, 085130 (2011)

Ref. Ge, Se, Te energies are for isolated atoms

	Cohesive Energy (eV/atom)	
Method	**QuantumATK**	**Literature**
PBE	-0.124	-0.103*
HSE06	-0.119	-

*https://materialsproject.org/materials/mp-1224375/

Ref. Ge, Se, Te energies are for respective monocrystalline

Te : mp-19
Sb : mp-104
Ge : mp-32

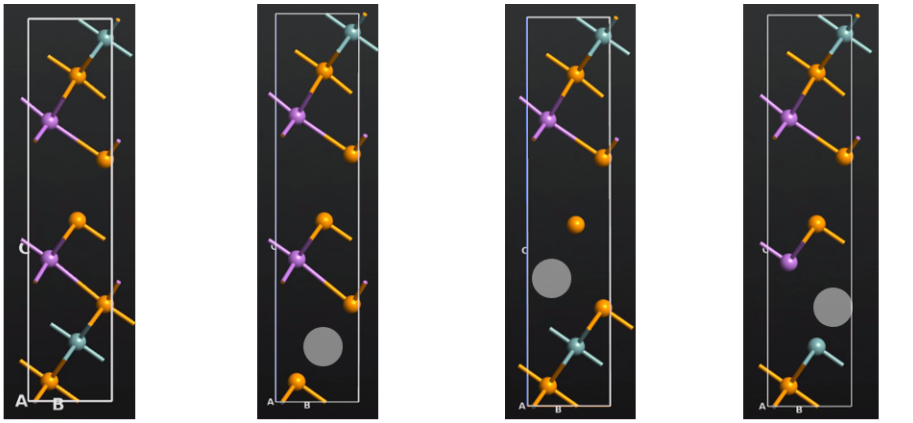
In the above results, two different sets of reference energy values are used for the elemental contributions. First set of cohesive energies (top of the picture) are computed using total energies of isolated Ge, Sb and Te atoms as reference

[2] BJăKooi and JăThăM DeăHosson. Electron diffraction and high-resolution transmission electron microscopy of the high temperature crystal structures of GexSb2Te3+x (x= 1,2,3) phase change material. *Journal of applied physics*, 92(7):3584–3590, 2002.

and match well with literature[3]. The following scripts are used for the above reference calculations (Ge (`HSE`, `PBE`), Sb (`HSE`, `PBE`), Te (`HSE`, `PBE`)). For the second set, total energies per atom in homogeneous bulk systems are used. The corresponding scripts are attached here (`Ge`, `Sb`, `Te`). Reference cohesive energy is calculated using the data from materials project database using the following IDs (GST:mp-1224375, Ge:mp-32, Sb:mp-104, Te:mp-19). From the above image, we find that our calculations agree reasonably with the published calculated reference values within 4 meV/atom for isolated atom reference and within 20 meV/atom for bulk reference. Note that the literature reference uses a different basis set and calculator settings.

Neutral Vacancy Formation Energies

In this section, we compute the formation energies of neutral vacancies within a unitcell of $Ge_2Sb_2Te_5$ using PBE XC functional and compare to literature results[4]. The formation energy of a single neutral vacancy is obtained as $E_{\text{pristine GST}}$ - ($E_{\text{GST with vacancy}}$ + $E_{\text{bulk reference}}$), where $E_{\text{bulk reference}}$ is the energy of the removed atom in its bulk state.



QuantumATK [eV]	0.59	1.29	1.22
Reference [eV]	0.96	1.60	1.20

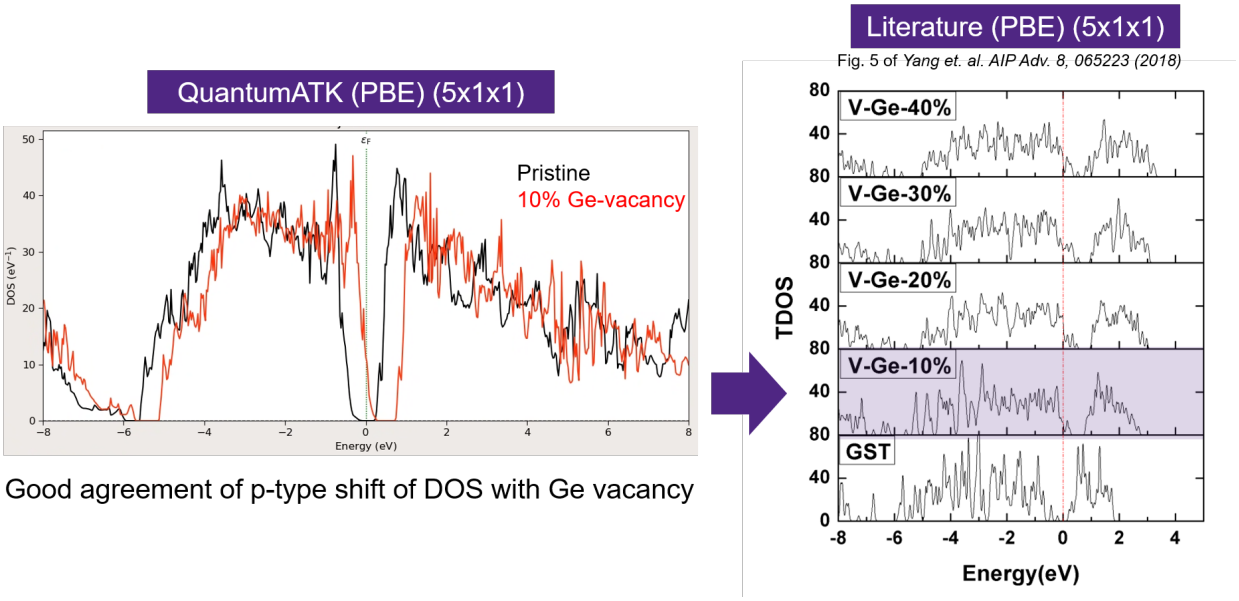
We perform three vacancy calculations by removing either one Ge, one Sb or one Te atom from the unitcell geometry as highlighted in the above image and replacing them with a ghost atom. A ghost atom is used to include the basis functions for that removed atom in the calculation. The corresponding scripts are given here (`Ge-vacancy`, `Sb-vacancy`, `Te-vacancy`). As can be seen from the above image, the formation energy of Ge vacancy is the lowest followed by Te and Sb. This ordering is similar to the literature results with some discrepancies in the absolute values. This could be attributed to the different lattice parameters and basis sets (reference: plane wave basis and PAW, QuantumATK: LCAO and PseudoDojo medium) used in the literature reference and in our calculations.

[3] Baisheng Sa, Jian Zhou, Zhitang Song, Zhimei Sun, and Rajeev Ahuja. Pressure-induced topological insulating behavior in the ternary chalcogenide Ge2Sb2Te5. *Physical Review B*, 84(8):085130, 2011.

[4] Fei Yang, Tian Chen, Minglei Wang, Beibei Yan, Luxu Wan, Daoming Ke, and Yuehua Dai. First-principles study of the properties for crystal Ge2Sb2Te5 with Ge vacancy. *AIP Advances*, 8(6):065223, 2018.

Total DOS With and Without Ge Vacancy

In this section we compare the total density of states from the GST (5x1x1) supercell with and without a Ge vacancy and compare the results to literature[Page 353, 4]. The scripts used for this calculation can be downloaded from these links (`Pristine`, `VGe`).



Good agreement of p-type shift of DOS with Ge vacancy

The above plot shows the DOS comparison and a clear p-type shift is observed on including 10% Ge vacancy. This observation is in good qualitative agreement with the reference.

3.1.10 STM simulations of tunneling anisotropic magneto resistance (TAMR)

Version: S-2021.06

In this tutorial you will learn how to use the *LocalDensityOfStates* analysis object for simulating STM measurements of surfaces. In particular we will be studying the tunneling anisotropic magneto resistance (TAMR) which is a measure of the difference in STM current and conductance depending on the magnetization direction of the substrate.

Contents

•*STM simulations of tunneling anisotropic magneto resistance (TAMR)*

–*Introduction*

–*Setting up the 2LFe/W(110) structure*

–*Local Density of States calculations*

–*Analyzing the results*

–*Co adatom on 2LFe/W(110)*

–*COSMICS project*

References

Introduction

Scanning tunneling spectroscopy is an experimental technique which uses a scanning tunneling microscope (STM) to probe the local density of electronic states (LDOS) of surfaces and materials on surfaces at the atomic scale. A scanning tunneling spectrum is obtained by placing the STM tip above a particular place on the sample. With the height of the tip fixed, the electron tunneling current is measured as a function of electron energy by varying the voltage between the tip and the sample. The conductance, dI/dV, obtained from such a measurement is proportional to the local electron density of states (LDOS) at the position of the tip.

TAMR

The resistance (inverse of conductance) of a tunneling junction involving just a single ferromagnetic layer may depend on the magnetization direction relative to the crystallographic axes. This effect is called tunneling anisotropic magnetoresistance (TAMR) and has been observed in several STM experiments[1,2]. The TAMR is caused by spin-orbit coupling, which results in an anisotropy of the density of states in and above the surface layer. The TAMR is potentially interesting for spintronics applications as it requires only one magnetic layer as opposed to traditional magnetic tunneling junctions involving both a reference and free magnetic layer.

This tutorial is inspired by the experimental and theoretical works in[Page 355, 1,2] which studies the TAMR of an iron bilayer on a W(110) surface, 2LFe/W(110), as schematically illustrated Fig. 3.1 below.

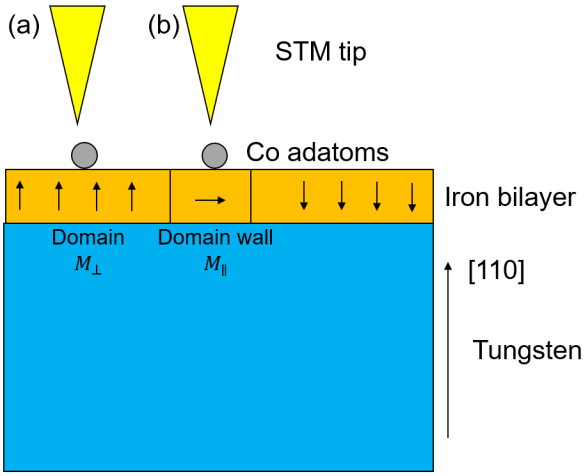


Fig. 3.1: Schematic illustration of the experiments. An iron bilayer is deposited on W(110). The Fe bilayer forms magnetic domains with the magnetization perpendicular to the interface. The magnetic domains are separated by domain walls in which the magnetization is parallel with the interface plane. By positioning the STM tip either over the domain (a) or over the domain wall (b) it is possible to measure the TAMR effect.

This surface can be prepared in such a way that finite domains of out of plane magnetizations are formed. Within the domain wall the magnetization will be in-plane. By measuring with STM the conductance over a domain you get the

[1] N.ăNeel, S.ăSchroder, N.ăRuppelt, P.ăFerriani, J.ăKroger, R.ăBerndt, and S.ăHeinze. Tunneling Anisotropic Magnetoresistance at the Single-Atom Limit. *Phys. Rev. Lett.*, 110:037202, Jan 2013. URL: https://link.aps.org/doi/10.1103/PhysRevLett.110.037202, doi:10.1103/PhysRevLett.110.037202.

[2] Johannes Schöneberg, Paolo Ferriani, Stefan Heinze, Alexander Weismann, and Richard Berndt. Tunneling anisotropic magnetoresistance via molecular $\ensuremath \pi $ orbitals of Pb dimers. *Phys. Rev. B*, 97:041114, Jan 2018. URL: https://link.aps.org/doi/10.1103/PhysRevB.97.041114, doi:10.1103/PhysRevB.97.041114.

out of plane component $(dI/dV)_\perp$ while the same measurement over a domain wall you get the in-plane component $(dI/dV)_\parallel$. The TAMR is then found as

$$TAMR = \frac{(dI/dV)_\perp - (dI/dV)_\parallel}{(dI/dV)_\perp}$$

The main experimental results from[1] are shown in the figure below. Panel (a) shows the measured dI/dV over a Co adatom on a 2LFe/W(110) surface for both in- and out of plane magnetization and panel (b) shows the TAMR as function of sample voltage.

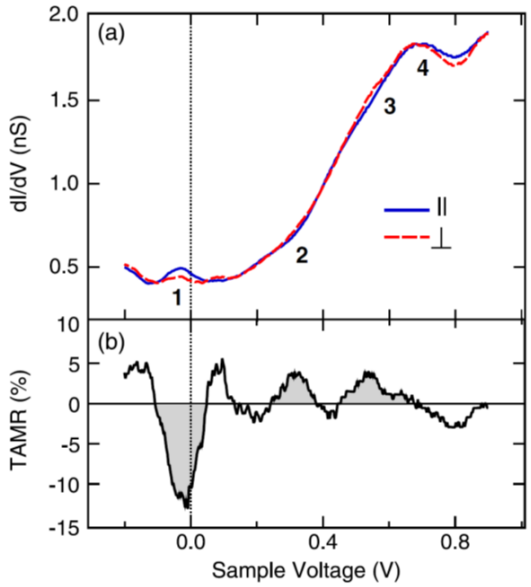


Fig. 3.2: Experimental results. (a) dI/dV spectra obtained on Co adatoms on domains (red dashed line) and on domain walls (blue solid line) of an Fe double-layer film on W(110). (b) TAMR determined from the spectra in (a). From[Page 355, 1].

Tersoff-Hamann approximation of simulated STM

The *LocalDensityOfStates* can be used to simulate scanning tunneling microscopy (STM) images within the Tersoff-Hamann approximation[3] assuming that the tip wave function has s-orbital character and is non-magnetic. When simulating an STM image, the `energy` can be interpreted as the bias difference applied between the tip and the surface and the *LocalDensityOfStates* represents the STM dI/dV measurement.

Since the local density of states $n(E,r) = A\frac{dI}{dV}|_{V=V_s}$ with $E = e \cdot V_s$ where V_s is the sample voltage, we can simulate the TAMR as

$$TAMR = \frac{n_\perp(E,r) - n_\parallel(E,r)}{n_\perp(E,r)} \tag{3.1}$$

Likewise, the current measured in an STM with an applied bias of V_s and with the tip at zero bias can be approximated as the partial electron density originating from eigenstates in the energy interval $[0; e \cdot V_s]$.

[3] J.ăTersoff and D.ăR. Hamann. Theory of the scanning tunneling microscopy. *Phys. Rev. B*, 31:805, 1985. doi:10.1103/PhysRevB.31.805.

Setting up the 2LFe/W(110) structure

We will start by setting up the structure for the calculations.

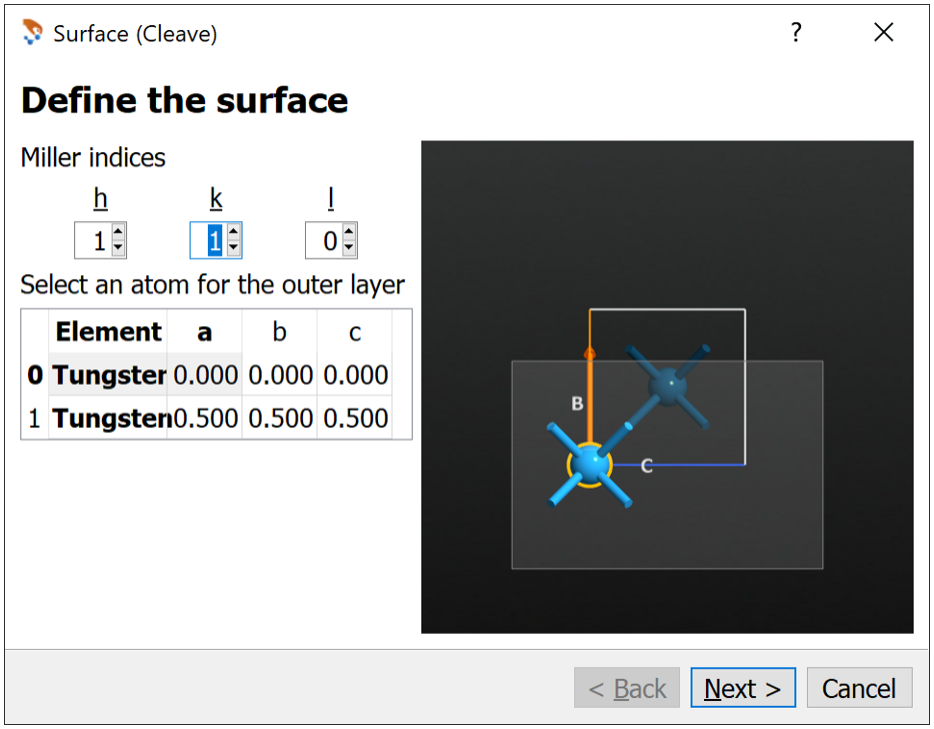
Structure

Open the **Builder** and

•Add a Wolfram/Tungsten bulk configuration from the database (search for *Wolfram*).

•Make a (110) surface using the **Surface (Cleave)** under **Builders** in the right column.

•On the first page set *h=1*, *k=1*, *l=0* . Press *Next*.



•On the second page simply press *Next* using all default values.

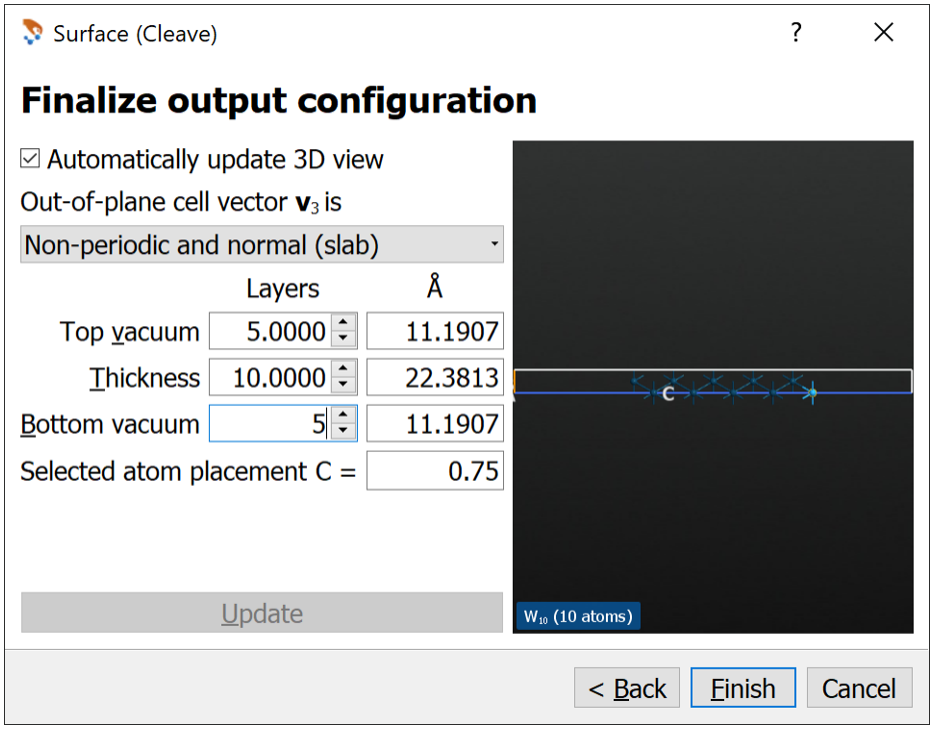
•On the third page apply the changes (see image below)

–From the combo box choose *Non-periodic and normal (slab)*

–Set the **Top** and **Bottom** vacuum to *5 layers*

–Set the **Thickness** to *10 layers*

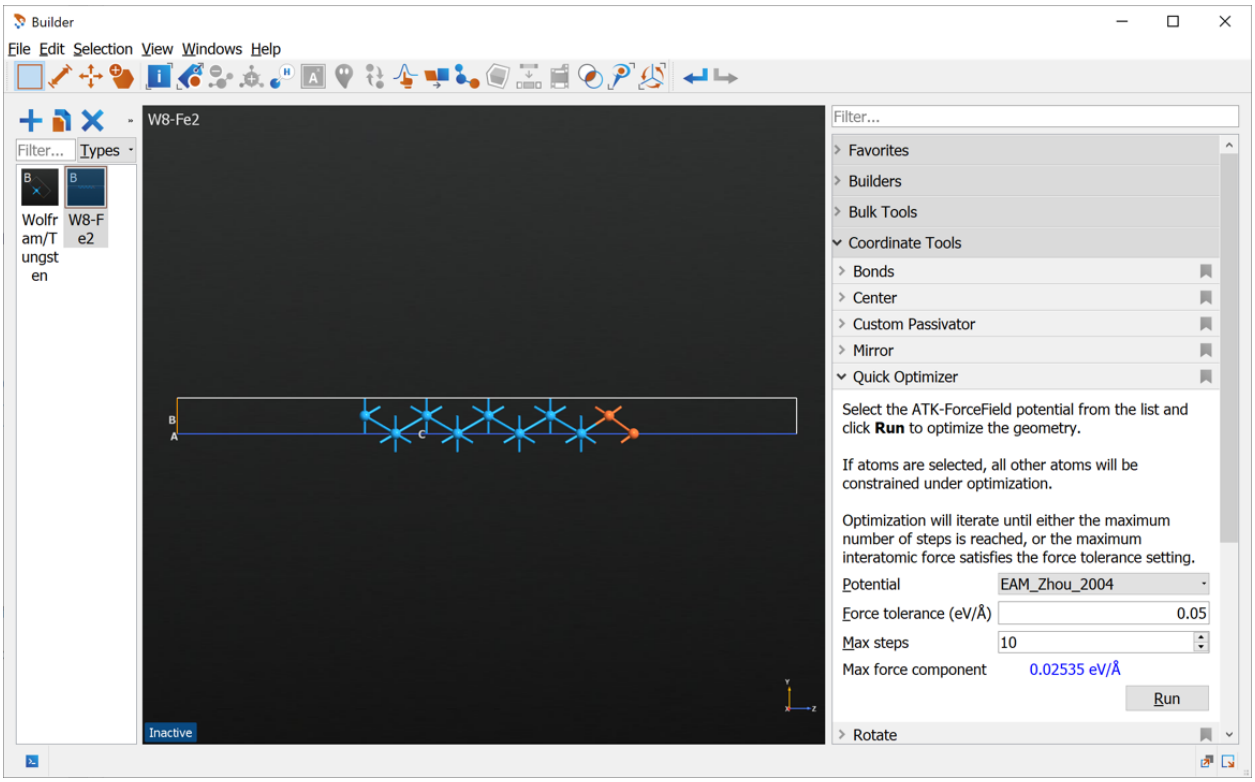
–Press *Finish*.



•Change the two rightmost **W** atoms to **Fe**.

•Rename the structure to `W8-Fe2` (press `F2`).

We can make an initial optimization of the structure using the **Quick Optimizer** (see image below). Run the optimization, which only takes a few seconds.



Send the structure to the **Script Generator** using the icon.

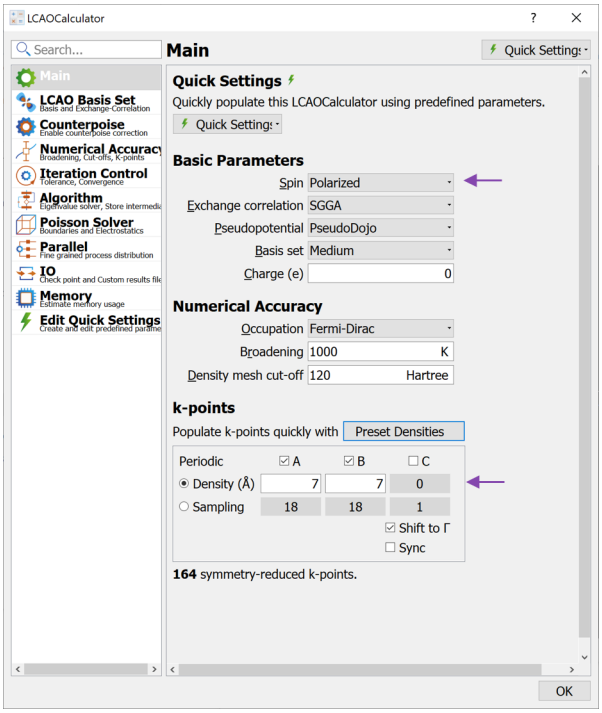
Tip: The **Quick Optimizer** is an easy way to do an initial optimization of the structure to make it more reasonable before proceeding with the main calculations. It is fast because it uses a ForceField, and will automatically suggest the possibilities from the internal QuantumATK database. However, not all structures will be supported by an appropriate ForceField.

Structural relaxation

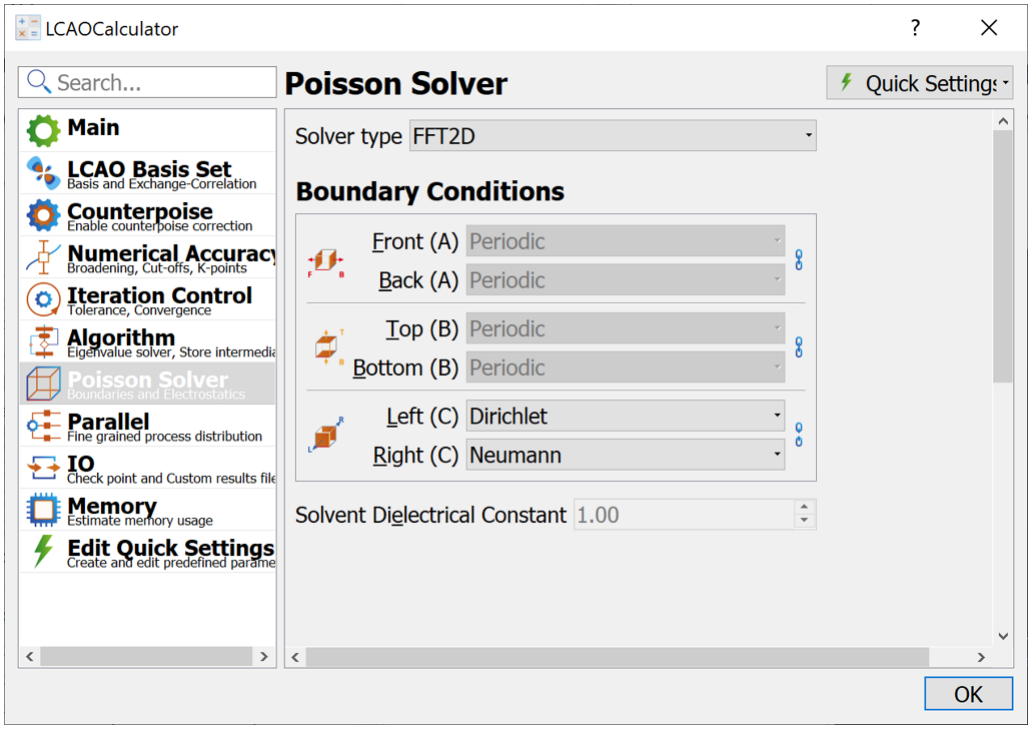
Although the structure has been relaxed using the **Quick Optimizer**, we will still perform a geometry optimization using a DFT calculator. The geometry optimization will be performed using a *spin-polarized* calculator, while the subsequent analysis will be performed with *spin-orbit coupling* included.

Now do the following:

- Add an **LCAOCalculator** to the script, open the **LCAOCalculator** widget and apply the changes on the main page
  - Change **Spin** to *Polarized*
  - Increase the **k-point density** to *7* (select from the *Preset Densities*).



- Go to the **Poisson Solver** page and apply the changes (see image below)
  - Change the **solver type** to *FFT2D*
  - Change the **boundary conditions** in the *C-direction* to *Dirichlet* and *Neumann*. (Click the chain-symbol to the right of the combo boxes to allow for different boundary conditions.)



•Close the **LCAOCalculator** widget.

Note: The reason for modifying the Poisson solver boundary conditions is to allow for a dipole shift across the W/Fe interface. At any interface between two materials a certain charge transfer will occur, which will give rise to a surface dipole. Using periodic boundary conditions in the C-direction will result in a electrostatic potential with a certain slope in the vacuum region, effectively acting as an external electric field. By using the combination of *Dirichlet* and *Neumann* boundary conditions, the electrostatic potential will be flat in the vacuum regions, but have different values on each side of the slab. Try to add an `ElectrostaticPotential` or `ElectrostaticDifferencePotential` analysis to the script and inspect the results after the calculation using the **Grid 1D Projector**.

•Now add an **OptimizeGeometry** to the script and use the default settings.

Finally change the name of the **Results file** to `W8-Fe2-polarized.hdf5` and run the calculation. The relaxed structure can also be downloaded here `W8_Fe2_relaxed_configuration.py`.

Local Density of States calculations

Having relaxed the atomic coordinates we are now ready to study the TAMR effect by calculating the local density of states (LDOS). The LDOS contains the DOS at a given energy projected onto a real-space position $n(E, r)$.

Send the relaxed configuration to the **Script Generator** and go through the following steps:

•Add an **LCAOCalculator** and open the widget.

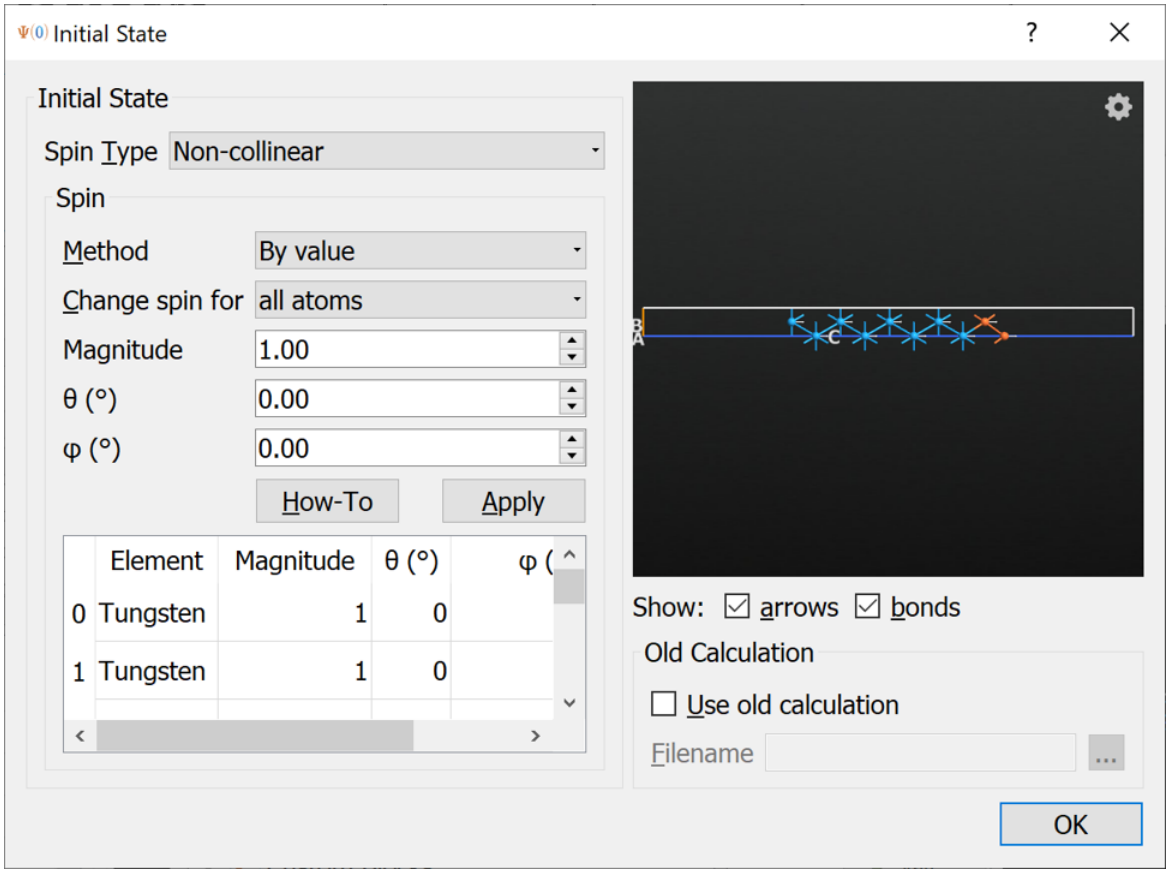
–Change **Spin** to *Noncollinear Spin-Orbit*

–Increase the **k-point density** to *7* (select from the *Preset Densities*).

–Modify the **Poisson Solver** as described above.

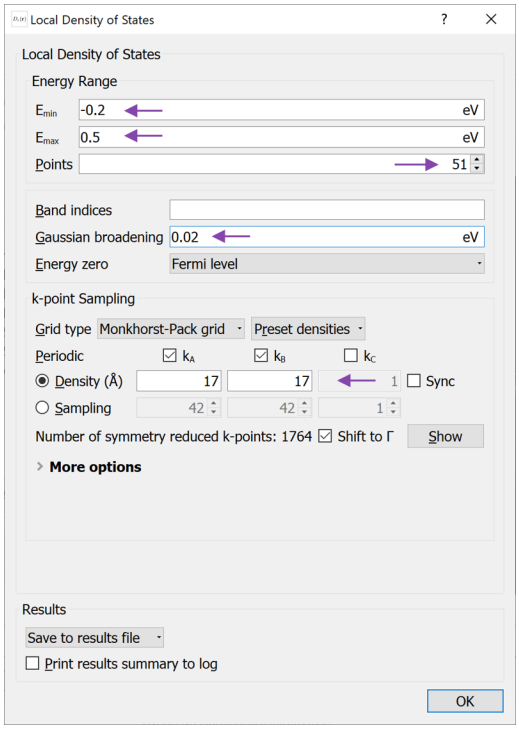
–Close the **LCAOCalculator**

•Add an **Initial State** block to the script and open it. In the **InitialState** widget you can change the initial spin orientation, when changing the **Spin Type** to *Non-collinear* (see below). For global spin rotations applied to all atoms the initial spin orientation will usually be kept during the self-consistent calculation. For now you can leave the angles at the default values of $\theta = 0°$ and $\phi = 0°$, which corresponds to the out-of-plane magnetization.



Tip: The spin vectors are shown in the 3D window, allowing easy inspection of the initial spin state.

•Add a **LocalDensityOfStates** object and open it to apply the changes indicated in the figure below



•Add a $\mathrm{Tr}(\rho S)$ **MullikenPopulation** object to the script.

•Finally change the **Results file** to `W8-Fe2-theta-0-phi-0.hdf5`. Send the job to **Job Manager** and run the calculation. The calculation takes around 30 minutes when run on a single node with 16 processes.

Note: `LocalDensityOfStates` objects can take up a lot of memory, since a real-space grid object is stored for every energy point. In order to reduce the memory of the calculation as well as of the stored `.hdf5` file, the number of energy points can be lowered. It is also possible to modify the density of real-space grid points. In order to do the latter, you must send the script to the **Editor** and add the line `density_mesh_cutoff=20*Hartree` to the **LocalDensityOfStates** object as shown below.

```
# ------------------------------------------------------------
# Local Density Of States
# ------------------------------------------------------------
kpoints = KpointDensity(
    density_a=17.0*Angstrom,
    density_c=0.0*Angstrom,
    force_timereversal=False,
    )

local_density_of_states = LocalDensityOfStates(
    configuration=bulk_configuration,
    kpoints=kpoints,
    energies=numpy.linspace(-0.2, 0.5, 51)*eV,
    band_indices=All,
    energy_zero_parameter=FermiLevel,
    spectrum_method=GaussianBroadening(0.02*eV),
    density_mesh_cutoff=20*Hartree,              # <--- Added line
```

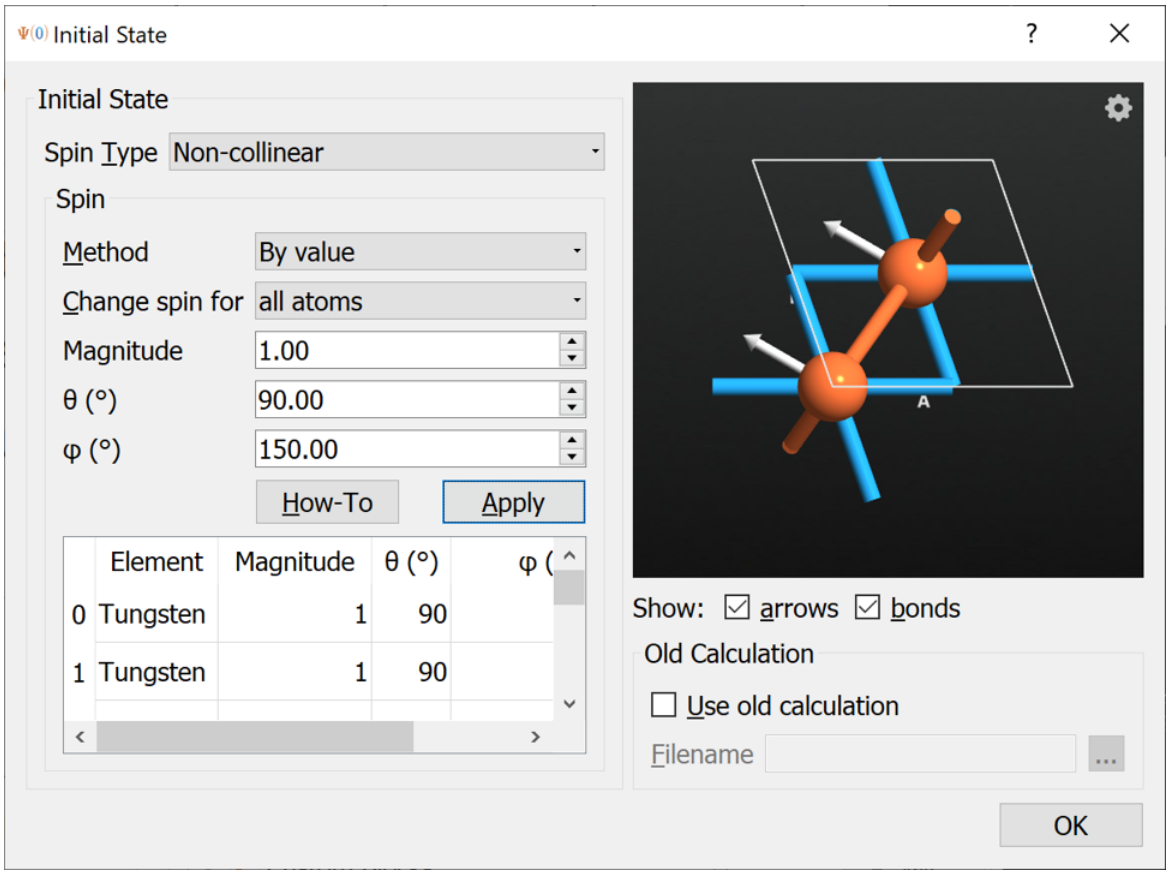
(continues on next page)

(continued from previous page)

```
    )
```

Now we will change the initial spin to simulate the in-plane magnetization.

•Open the **Initial State** widget again and change the angles to $\theta = 90°$ and $\phi = 150°$, which is the orientation with the lowest energy (try for yourself to scan over ϕ angles and calculate the total energy at each.) Press **Apply** to rotate the angles. The spin orientation should now look as below:



Close the **InitialState** widget and change the **Results file** to `W8-Fe2-theta-90-phi-150.hdf5` and run the calculation.

When both calculations have finished, we are ready to analyze the results.

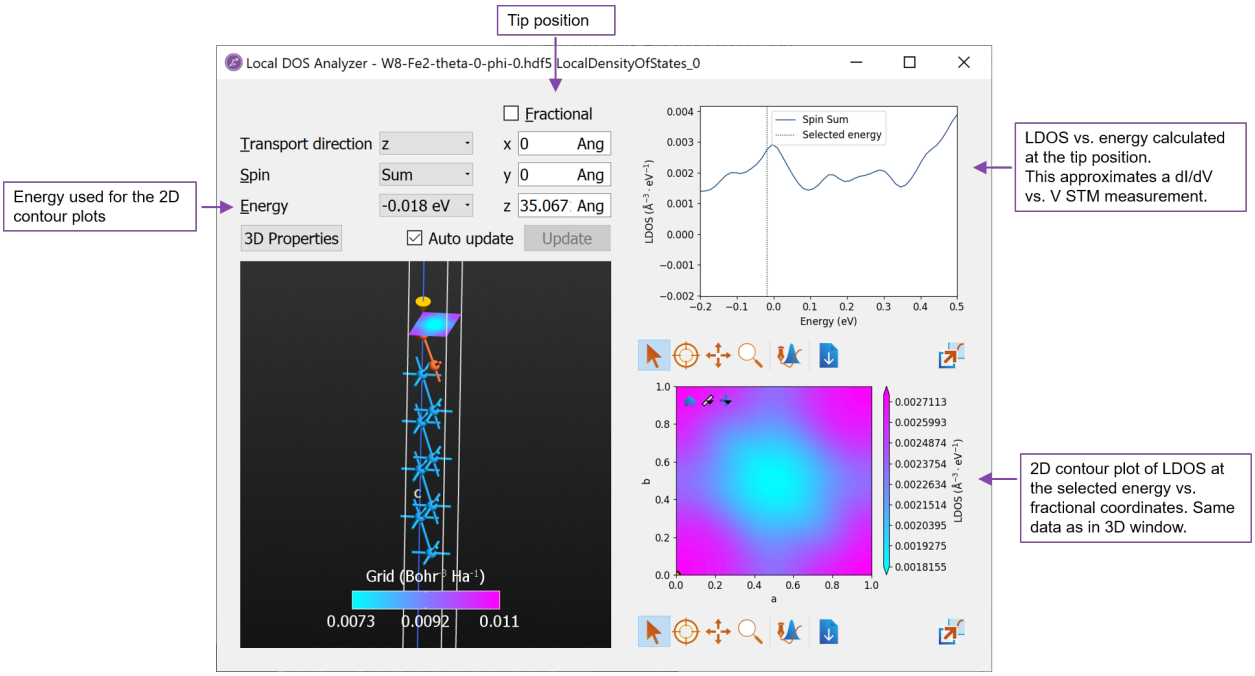
Analyzing the results

Once the calculation has finished we are ready to analyze the results. Before analyzing the **LDOS** calculations, take a look at the calculated **MullikenPopulations** using the **Viewer**. Verify that the spins on the iron atoms are pointing in the expected directions - out of plane (in the z-direction) for the `W8-Fe2-theta-0-phi-0.hdf5` file and in-plane for the `W8-Fe2-theta-90-phi-150.hdf5` file.

LDOS analysis

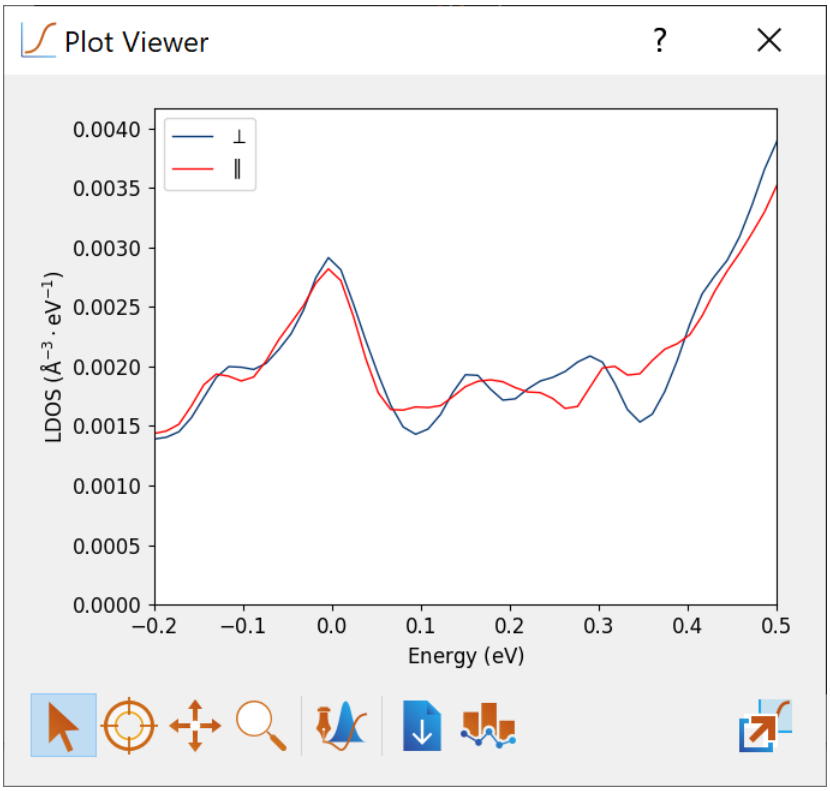
Mark the $D_\epsilon(\mathbf{r})$ **LocalDensityOfStates** object in the file `W8-Fe2-theta-0-phi-0.hdf5` and open it with the **Local DOS Analyzer**. The analyzer should appear as shown below. In the STM-like analyzer, you can adjust the tip position using both fractional- and Cartesian coordinates. When the tip position is changed, the LDOS vs. energy plot (upper right) will be updated. Change the tip position to 2 Å above the surface Fe atom at *(0, 0, 35.067)* Å. Try also to change the energy and see how the contour plot changes. The contour plot in the 3D window with the atomic structure contains the same data as the contour plot in the lower right plot.

Right click on the line in the upper right plot and choose **Edit item** and change the label *Spin Sum* to `$\perp$`, which QuantumATK will automatically parse and show as the symbol.

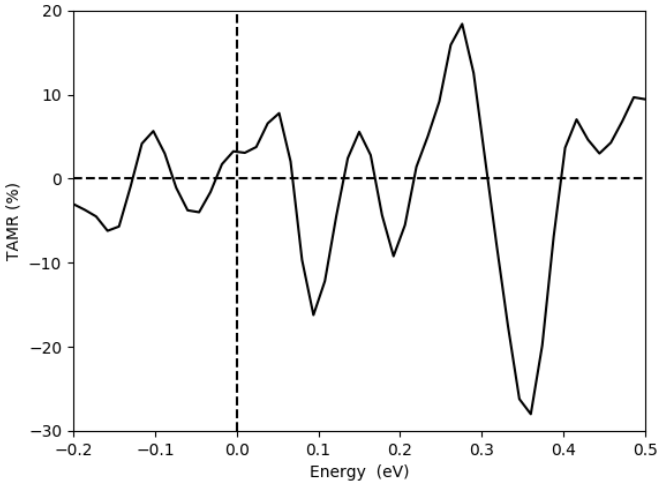


Now open the **LocalDensityOfStates** object from the file `W8-Fe2-theta-90-phi-150.hdf5` containing the in-plane magnetization. Apply the same setting for the tip position as above and edit the line in the upper right plot and change the label to `$\parallel$`. Also change the color and/or line style, such that it is different from the default setting.

Now drag one of the LDOS vs. energy plots onto the other and select *Overlay with shared axes* to obtain a plot like below. It is evident that the LDOS is different for the two magnetization directions and that there is a TAMR effect.



In order to quantify the TAMR effect, download and run the script `calculateTAMR.py`. The script calculates the LDOS vs. energy 2 Å above the surface Fe atom as we just did in the analyzer above. Then it calculates the TAMR using equation (3.1) above and produces the figure shown below. In qualitative agreement with the experimental findings[Page 355, 1] the TAMR is oscillating as a function of energy (bias voltage in an STM experiment). The magnitude of the calculated TAMR of $\sim \pm 20$



Co adatom on 2LFe/W(110)

In order to get a more direct comparison to the experiments from[Page 355, 1] shown in Fig. 3.2 at the top, we here present results for a Co adatom positioned on the 2LFe/W(110) slab. The calculations have been performed in the same way as described above. Try for yourself to add a Co atom and relax the structure using the **Quick Optimizer** in the **Builder** by going through the following steps:

- In the **Builder**, repeat the 2LFe/W(110) structure two times in the A- and B directions.
- Add a Co atom in a bridge position between two surface Fe atoms.
- Relax the configuration using the **Quick Optimizer**.
- Send the configuration to the **Script Generator** and setup the spin-orbit and **LocalDensityOfStates** calculations as described above.

A final script is also given here: `W8_Fe2_Co.py`. The script loops over the (θ, ϕ) angles and performs the self-consistent spin-orbit calculation and LDOS calculation for both angle pairs $(0, 0)$ and $(90, 150)$. Run either the scripts you have made yourself or the one given here.

When the calculations are finished (the calculation takes several hours when running on a single node with 16 cores) the results can be analyzed and plotted with the script `calculateTAMR-Co-adatom.py` to give a figure like the one shown below, on the right. The calculated LDOS and TAMR share many similarities with the experimental results in Fig. 3.3 (left part):

- The overall shape of the dI/dV and LDOS plots agree with a small peak close to the Fermi energy and a larger peak around 0.6 eV.
- Both the calculated and measured TAMR shows a clear dip around the Fermi energy and an oscillating behavior of the TAMR. The oscillation in the calculated TAMR are, however, more rabid than in the experiments.

A potential cause of the differences between experimental and calculated data is the tip distance to the Co adatom. Due to the localized basis set used in the QuantumATK calculations, the LDOS needs to be calculated relatively close to the surface - here 2 Å above the Co atom. It is very likely that the experimental distance is somewhat larger. Also, the calculated results are obtained with a supercell which is a 2x2 repetition of the unit cell. It is likely that this is too small a supercell to match the experimental conditions.

COSMICS project

TAMR effects are studied in the COSMICS project founded by the European Unions Horizon 2020 research and innovation program under Grant Agreement No. 766726. More information about the COSMICS project can be found here: http://cosmics-h2020.eu/

References

3.1.11 Bulk Magnetic Anisotropy Energy

Version: S-2021.06

In this tutorial you will learn how to use the *MagneticAnisotropyEnergy* study object to calculate the magneto-crystalline anisotropy (MCA) energy of bulk FePt in the $L1_0$ phase.

The result of the *MagneticAnisotropyEnergy* will be compared with self-consistent total energy calculations including spin-orbit coupling, and you will learn about the most important numerical parameters to converge, when performing MCA calculations.

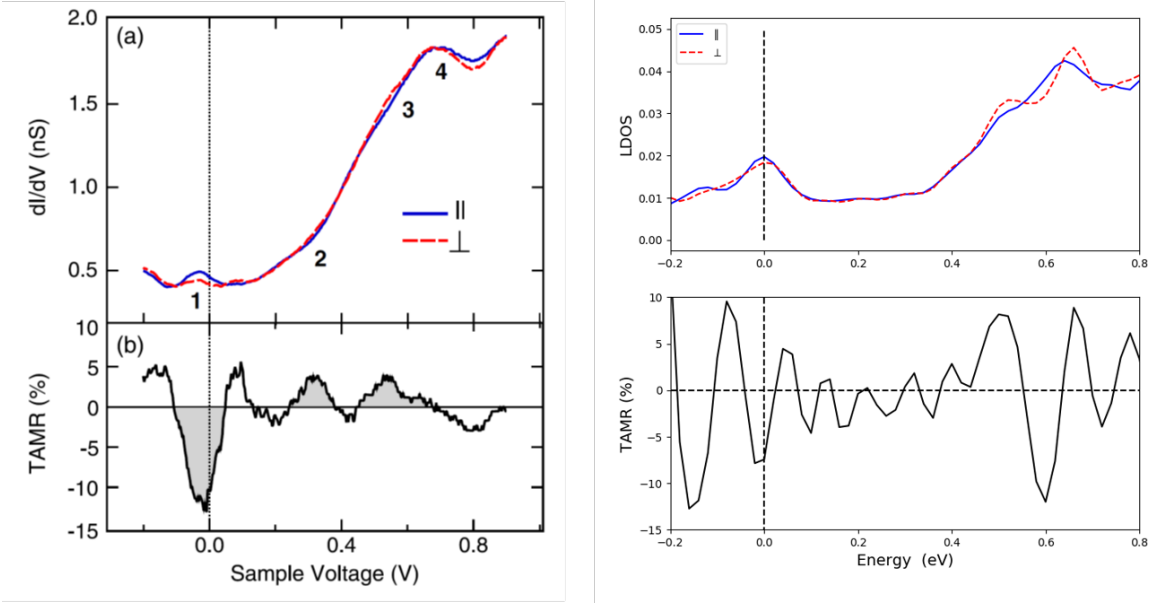


Fig. 3.3: Experimental results (left). (a) dI/dV spectra obtained on Co adatoms on domains (red dashed line) and on domain walls (blue solid line) of an Fe double-layer film on W(110). (b) TAMR determined from the spectra in (a). From[Page 355, 1]. The right part of the figure shows the data calculated with QuantumATK.

Contents

•*Bulk Magnetic Anisotropy Energy*
–*Introduction*
–*Theory*
–*MAE of FePt*
–*TotalEnergy calculations*
–*Convergence of results*
–*COSMICS project*

Introduction

The magnetic anisotropy energy (MAE) is the energy difference between two magnetization directions. Different physical mechanisms can contribute to the MAE, but here we are concerned with the magnetocrystalline anisotropy (MCA). Another important contribution is the shape-anisotropy, which occurs if the magnetic material has a non-spherical shape.

The origin of the MCA is the spin-orbit coupling and it is characterized by the dependence of the energy of a magnetic system on the orientation of the magnetization with respect to the crystallographic structure of the material. The axis (or plane) corresponding to the minimum of energy is the so-called easy axis (plane).

The MAE is of central interest for both fundamental and practical reasons, with STT-MRAM being one of the most important technological examples.

STT-MRAM

Spin-transfer torque magnetic random access memory (STT-MRAM) enables higher densities, low power consumption and reduced cost compared to regular MRAM devices. The main advantage of STT-MRAM over regular MRAM is the ability to scale the STT-MRAM chips to achieve higher densities at a lower cost.

One of the key requirements of an STT-MRAM device is of course to be able to store information. This requires that the energy-barrier between the 1 and 0 states is large enough to not allow switching due to thermal fluctuations. The MCA is a key element in determining the energy barrier and thus the thermal stability of a STT-MRAM memory unit.

Theory

The magnetic anisotropy energy is defined as the energy difference between two spin orientations:

$$MAE = E(\theta_1, \phi_1) - E(\theta_0, \phi_0),$$

where the spin orientation is described by the two spherical angles θ and ϕ. The MAE can be calculated directly from self-consistent calculations including spin-orbit coupling. This can, however, be computationally quite demanding since spin-orbit calculations can some-times be difficult to converge at the desired accuracy.

Force theorem

In the *MagneticAnisotropyEnergy* study object we calculate the MAE using the force theorem, which allows you to evaluate the energy difference using non-self consistent band energies[1]:

$$MAE = \sum_i f_i(\theta_1, \phi_1)\epsilon_i(\theta_1, \phi_1) - \sum_i f_i(\theta_0, \phi_0)\epsilon_i(\theta_0, \phi_0),$$

where $f_i(\theta, \phi)$ is the occupation factor for band i (including both band and k-point index) with the spin orientation (θ, ϕ) and $\epsilon_i(\theta, \phi)$ is the corresponding band energy.

The use of the force theorem (FT) has been validated in several papers, e.g.[2],[3] against self-consistent calculations of total energies. Using the FT has several advantages over the self-consistent approach:

[1] G.ăH.ăO. Daalderop, P.ăJ. Kelly, and M.ăF.ăH. Schuurmans. First-principles calculation of the magnetocrystalline anisotropy energy of iron, cobalt, and nickel. *Phys. Rev. B*, 41:11919, 1990. doi:10.1103/PhysRevB.41.11919.

[2] Piotr Blonski and Jurgen Hafner. Density-functional theory of the magnetic anisotropy of nanostructures: an assessment of different approximations. *Journal of Physics: Condensed Matter*, 21(42):426001, 2009. URL: http://stacks.iop.org/0953-8984/21/i=42/a=426001.

[3] M.ăBlanco-Rey, J.I. Cerda, and A.ăArnau. Validity of perturbative methods to treat the spin–orbit interaction: application to magnetocrystalline anisotropy. *New J. Phys.*, 21:073054, 2019. URL: https://iopscience.iop.org/article/10.1088/1367-2630/ab3060.

•It is usually faster since the self-consistent solution of the spin-orbit calculation is computationally significantly more demanding than the polarized one.

•Often it is harder to achieve good convergence for a spin-orbit calculation than for a polarized one.

•Using the FT approach allows for projection analysis since each band energy has an associated eigenvector that can be projected onto atomic sites or individual orbitals (see the :ref:ťmae_bulk_analyze_resultsť section). The projection analysis is often helpful in understanding the physics behind the calculated MAE. An example could be the difference between surface/interface atoms and bulk atoms in the contribution to the MAE in a metallic slab[4].

Note: About the naming: In this tutorial, the name magnetic anisotropy energy (MAE) will be used somewhat interchangeably with the name magnetocrystalline anisotropy (MCA). A third name, which is often used, is perpendicular magnetic anisotropy (PMA) and refers to an interface plane, often with a magnetic tunnel junction in mind - see also *Magnetic Anisotropy Energy of Fe-MgO-Fe MTJ structure*. In the literature, the MAE will sometimes include the so-called shape anisotropy as well, which accounts for the (classical) dipole-dipole interaction between magnetic moments[Page 368, 1].

MAE of FePt

$L1_0$ FePt has been studied in a number of scientific papers and is one of the materials with the largest known MCA. In this tutorial we will use the structure from[Page 368, 3]. It is well-known that even small structural difference can have a big impact on the MCA. In order to have a direct comparison to the calculated values in[Page 368, 3] we will not perform any structural relaxation.

Structure

The structure of the FePt cell is shown below.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleTetragonal(2.722*Angstrom, 3.71281*Angstrom)

# Define elements
elements = [Platinum, Iron]

# Define coordinates
fractional_coordinates = [[ 0.5,  0.5,  0.5],
                          [ 0. ,  0. ,  0. ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

[4] K.ăBairagi, A.ăBellec, V.ăRepain, C.ăChacon, Y.ăGirard, Y.ăGarreau, J.ăLagoute, S.ăRousset, R.ăBreitwieser, Yu-Cheng Hu, YenăCheng Chao, WoeiăWu Pai, D.ăLi, A.ăSmogunov, and C.ăBarreteau. Tuning the Magnetic Anisotropy at a Molecule-Metal Interface. *Phys. Rev. Lett.*, 114:247203, 2015. URL: https://link.aps.org/doi/10.1103/PhysRevLett.114.247203.

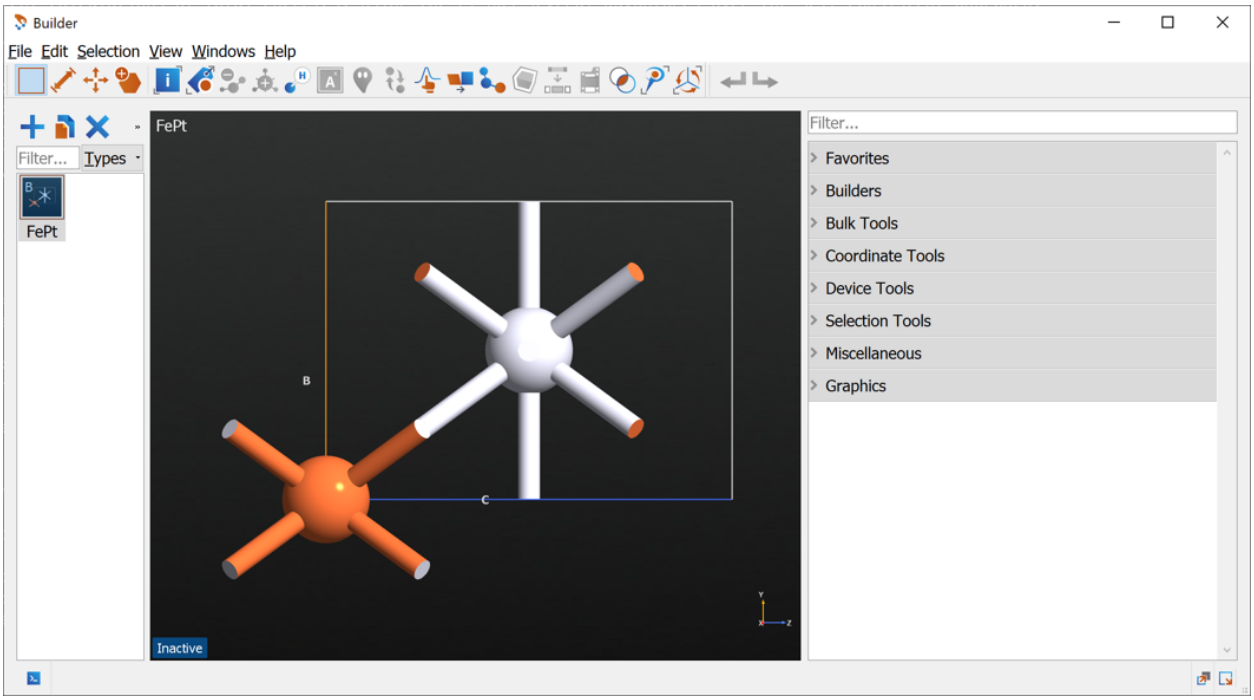
Copy the python script with the bulk configuration and paste it into the **Builder**:

1.Press the **Add** icon

2.Select **From Clipboard**

3.Press `F2` and rename the structure to FePt.

The structure should look like this:



Send the structure to the **Script Generator** using the icon.

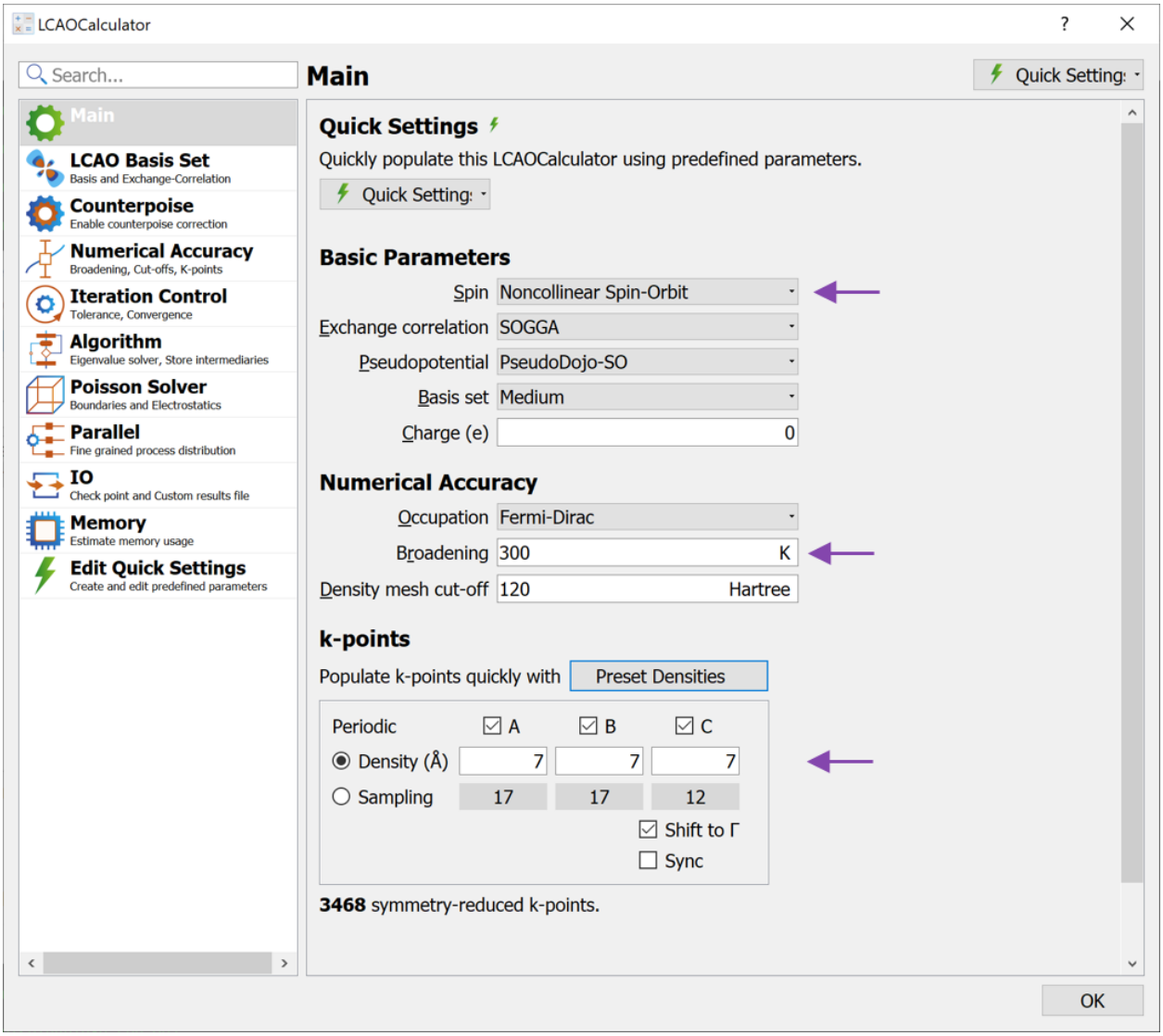
Setup MagneticAnisotropyEnergy calculation

MCA values are small quantities, most often below 1 meV/atom. This puts strong requirements on the numerical settings and a careful convergence study is recommended for any use of *MagneticAnisotropyEnergy*.

In the *Convergence of results* section we will go into more details about convergence studies. For now we will setup a calculator with reasonably well-converged settings resulting in calculated MCA values converged within ~ 0.2 meV.

In the **Script Generator** add an **LCAOCalculator**. Open the calculator block and apply the following changes (see image below):

- Change the **Spin** to *Noncollinear Spin-Orbit*
- Change the **Broadening** to *300 K*
- Finally increase the **k-point density** to *7 Å* (select from the **Preset Densities** combo box)



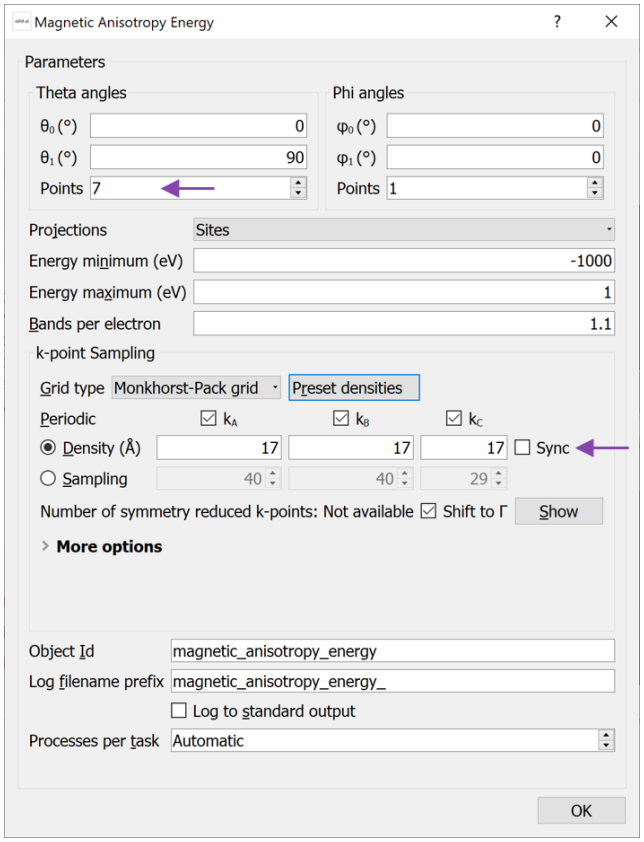
Close the calculator widget and add a **MagneticAnisotropyEnergy** block, located under the **Study object** category. Open the **MagneticAnisotropyEnergy** block to change the settings (see image below):

- Increase the number of **Theta angle points** to *7*.

  If you are only interested in the energy difference between the out-of plane, $\theta = 0$, and in-plane, $\theta = 90$, configurations it suffice to just have two Theta angle points. For tutorial purposes we add more angles to inspect the behavior of $MAE(\theta)$.

- Increase the **k-point density** to *17 Å* (select from the **Preset densities** combo box)

- Leave the **Projections** to *Sites*, but notice that you can also project on *Sites and Shells* or *Sites and Orbitals*. These options can be useful for analysis of the MAE at e.g. interfaces, and will be addressed in a different tutorial.



Close the **MagneticAnisotropyEnergy** widget. You are now ready to submit the calculations. Note that due to the rather high k-point sampling in both the **LCAOCalculator** and in the **MagneticAnisotropyEnergy** object, the calculation is somewhat computationally demanding and should preferably be run with several MPI processes.

Send the calculation to the **Job Manager** using the icon and run the calculation. The calculation can take up to several hours, depending on the available computational resources. On a single node with 24 cores, it takes around 10 minutes with QuantumATK R-2020.09-SP1.

Note: A **MagneticAnisotropyEnergy** object can only be calculated when the calculator has **Spin** set to *Noncollinear Spin-Orbit*.

Analyze results

Once the calculation is done, we can inspect the result by either using **Text Representation** or the dedicated **MAE Analyzer**:

- Find the **Text Representation** button in the right-hand panel. This will show a summary of the calculations as shown here:

```
# Title: FePt.hdf5 - magnetic_anisotropy_energy
# Type:  MagneticAnisotropyEnergy
+-------------------------------------------------------------------------------+
| MagneticAnisotropyEnergy Report                                               |
+-------------------------------------------------------------------------------+
```

(continues on next page)

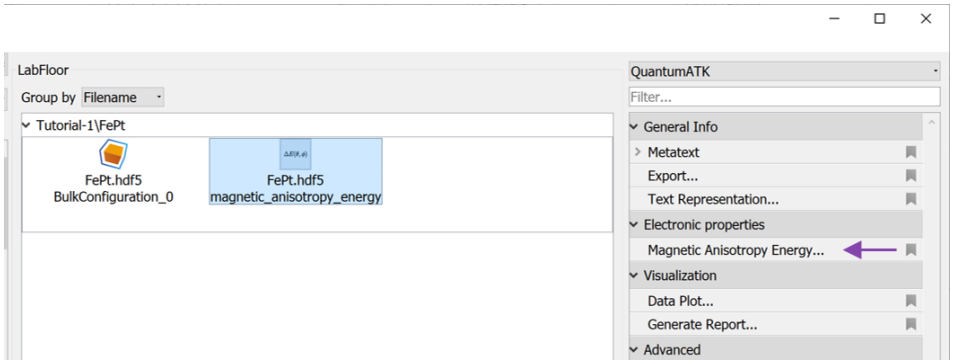
(continued from previous page)

```
+-------------------------------------------------------------------------------+
| Calculated angles (Degrees)                                                   |
+-------------------------------------------------------------------------------+
| (theta, phi) = (   0.00,    0.00)                                             |
| (theta, phi) = (  15.00,    0.00)                                             |
| (theta, phi) = (  30.00,    0.00)                                             |
| (theta, phi) = (  45.00,    0.00)                                             |
| (theta, phi) = (  60.00,    0.00)                                             |
| (theta, phi) = (  75.00,    0.00)                                             |
| (theta, phi) = (  90.00,    0.00)                                             |
|                                                                               |
+-------------------------------------------------------------------------------+
| Total magnetic anisotropy energy (MAE)                                        |
+-------------------------------------------------------------------------------+
| MAE = E(theta_1,   phi_1) - E(theta_0,   phi_0)                               |
| MAE = E(  90.00,    0.00) - E(   0.00,    0.00)                               |
| MAE =  2.3823 meV                                                             |
+-------------------------------------------------------------------------------+
```

The bottom of the output shows the final MAE calculated as the energy difference between the first and last angles. The positive MAE value means that the $\theta = 0°$ spin direction has a a lower energy than the $\theta = 90°$ direction.

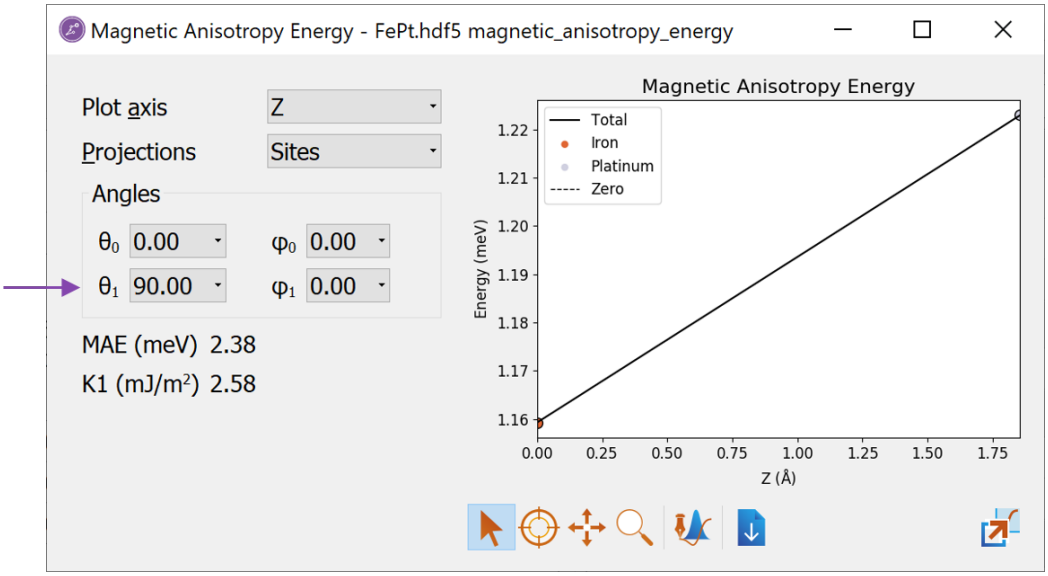
The calculated MCA values of 2.3 meV is slightly smaller than the values reported in[Page 368, 3] ranging from 2.59 meV for self-consistent total energies calculated with VASP to 2.93 meV calculated with Siesta. As will be shown in *Convergence of results*, the results ontained so far are not completely converged with respect to k-point sampling. Using higher k-point sampling, the QuantumATK values are 2.5 meV and are thus in very close agreement with the self-consistent results of VASP.

•The result of a MagneticAnisotropyEnergy calculation can also be analyzed with the Magnetic Anisotropy Energy analyzer. Mark the calculated MagneticAnisotropyEnergy on the **LabFloor** and open the analyzer.



The analyzer shows the contribution to the MAE from different projections, in this case for each atom. It shows the Fe and Pt atoms contribute more or less equally to the total MAE.

The calculated angles can be selected in the **Angles** combo boxes. Selecting $\theta_0 = 0$ and $\theta_1 = 90$ will result in the largest values.



In the lower left, the **total MAE** is shown in units of meV. Just below is shown the **K1** value, which here is defined as $K1 = MAE/(2 \cdot A)$ where A is the cross section area. This quantity is mainly relevant when studying interfaces, where **K1** is the surface anisotropy energy density. More details about this can be found in the tutorial *Magnetic Anisotropy Energy of Fe-MgO-Fe MTJ structure*.

MCA vs. angle

The dependence of the MCA value on the θ angle can be shown with a small python script.

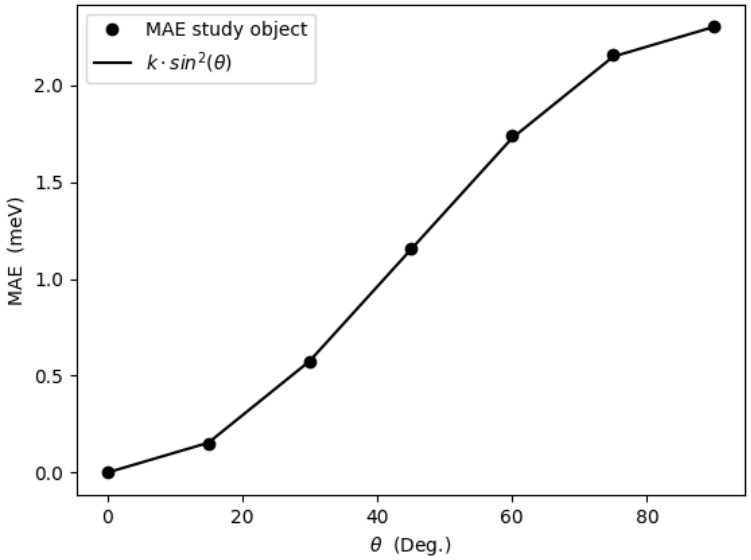
```
import pylab

filename = 'FePt.hdf5'

# Load the MAE object
magnetic_anisotropy_energy = nlread(filename, MagneticAnisotropyEnergy)[0]

# Get the calculated angles
theta_angles = magnetic_anisotropy_energy.thetaAngles()

# Get the calculated MAE values
mae_values = [magnetic_anisotropy_energy.magneticAnisotropyEnergy(
    theta_0=0*Degrees, theta_1=theta).inUnitsOf(meV)
    for theta in theta_angles]

# Get the total MAE
k = mae_values[-1]

# Plot the results
pylab.figure()
pylab.plot(theta_angles, mae_values, 'ko', label='MAE study object')
pylab.plot(theta_angles, k*numpy.sin(theta_angles)**2, 'k-', label='$k\\cdot sin^2(\\
↪theta)$')
pylab.xlabel('$\\theta$  (Deg.)')
pylab.ylabel('MAE  (meV)')
```

(continues on next page)

(continued from previous page)

```
pylab.legend(loc=0)
pylab.show()
```

Run the script to produce the figure below. In the script, the MAE values are extracted for different theta angles and plotted. The plot shows that the MAE follows a $k \cdot \sin^2(\theta)$ dependence in agreement with several previous studies, where k is the maximum MCA (MAE) value.



TotalEnergy calculations

As mentioned above, the MAE calculated with the *MagneticAnisotropyEnergy* study object is calculated using the force theorem (FT), where the spin-orbit calculation is performed non-self-consistently from a polarized calculation.

It is also possible to directly calculate the MAE from total energy differences of self-consistent spin-orbit calculation as will be shown below. It can often speedup the calculation by first performing a relatively inexpensive spin-polarized calculation and use this as an initial state for the more expensive Noncollinear Spin-Orbit calculation.

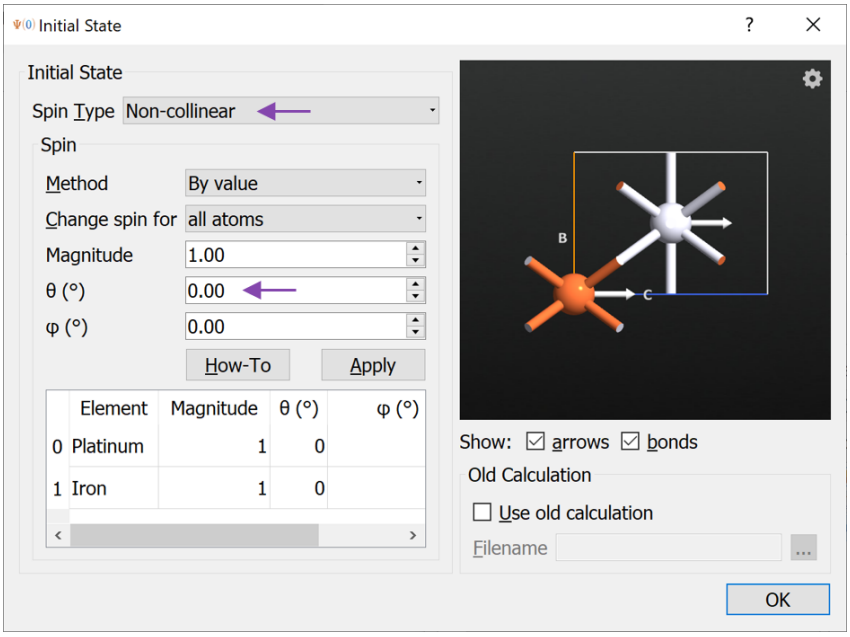
We will now show how to setup a Noncollinear Spin-Orbit with the spins rotated in a particular direction. After that we will use a slightly modified python script to loop over different spin-directions.

•Go back to the **Script Generator** and delete the **MagneticAnisotropyEnergy** block.

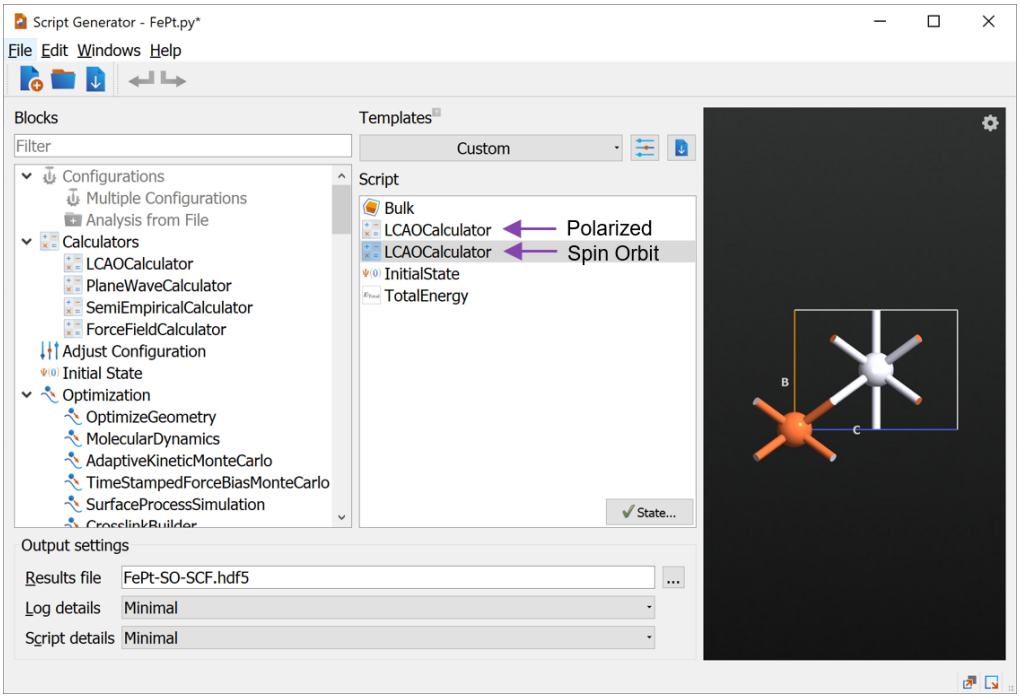
•Add another **LCAOCalculator** block to the script and apply the same settings as above.

•Open the first **LCAOCalculator** block and change the **Spin** to *Polarized*.

•Add an **InitialState** block and open it. Change the **Spin Type** to *Non-collinear*. Now you can change the spin direction of some or all atoms by modifying the θ and ϕ angles and press **Apply**. More instructions on how to setup advanced spin structures and use the widget are available if you press the **How-To** button.



•Close the InitialState widget and add a E_{Total} **TotalEnergy** analysis to the script. The script should now look as in the figure below.



Below we provide a script that loops over initial state angles. In order to understand this script, we first take a look at the current script and learn how to modify it. Send the script to the **Editor** and apply the following changes (see image below):

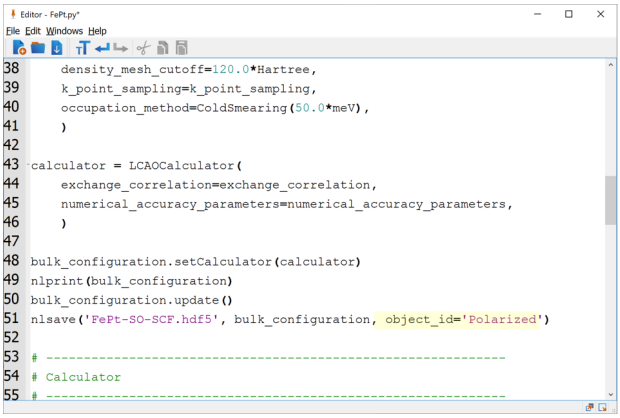
•Add `object_id='Polarized'` to the `nlsave` command in line 51.

•Add the line

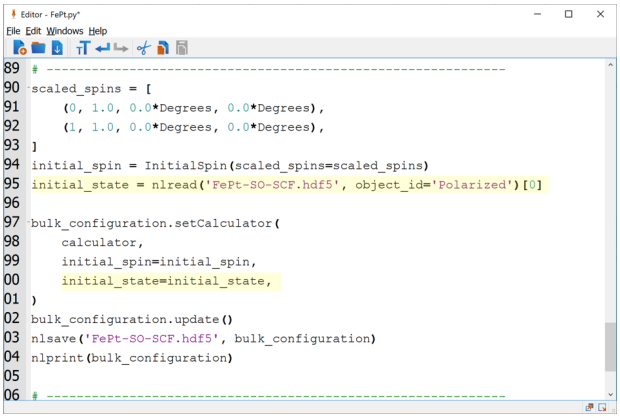
```
initial_state = nlread('FePt-SO-SCF.hdf5', object_id='Polarized')[0]
```

after the definition of `initial_spin` on line 95.

- When setting the calculator, add `initial_state=initial_state` to the setCalculator function.



```
38      density_mesh_cutoff=120.0*Hartree,
39      k_point_sampling=k_point_sampling,
40      occupation_method=ColdSmearing(50.0*meV),
41      )
42
43  calculator = LCAOCalculator(
44      exchange_correlation=exchange_correlation,
45      numerical_accuracy_parameters=numerical_accuracy_parameters,
46      )
47
48  bulk_configuration.setCalculator(calculator)
49  nlprint(bulk_configuration)
50  bulk_configuration.update()
51  nlsave('FePt-SO-SCF.hdf5', bulk_configuration, object_id='Polarized')
52
53  # ------------------------------------------------------------
54  # Calculator
55  # ------------------------------------------------------------
```



```
89  # ------------------------------------------------------------
90  scaled_spins = [
91      (0, 1.0, 0.0*Degrees, 0.0*Degrees),
92      (1, 1.0, 0.0*Degrees, 0.0*Degrees),
93  ]
94  initial_spin = InitialSpin(scaled_spins=scaled_spins)
95  initial_state = nlread('FePt-SO-SCF.hdf5', object_id='Polarized')[0]
96
97  bulk_configuration.setCalculator(
98      calculator,
99      initial_spin=initial_spin,
00      initial_state=initial_state,
01  )
02  bulk_configuration.update()
03  nlsave('FePt-SO-SCF.hdf5', bulk_configuration)
04  nlprint(bulk_configuration)
05
06  # ------------------------------------------------------------
```

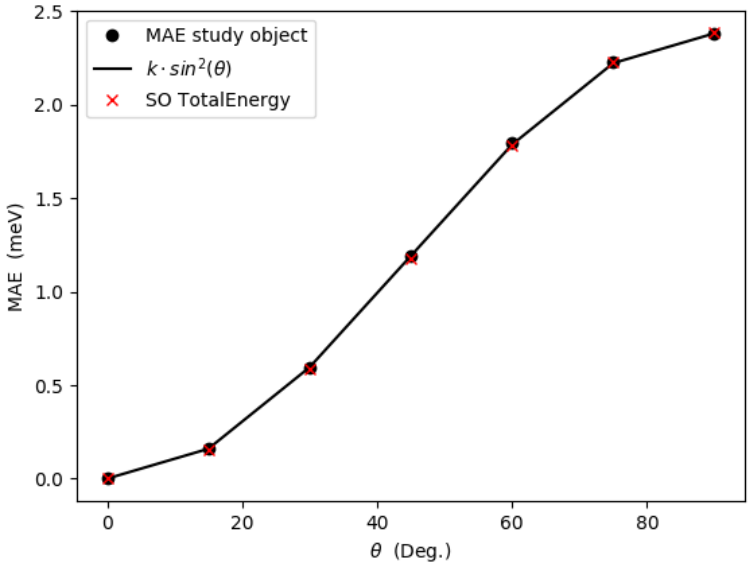
The script you now have, will first perform a spin-polarized calculation and save the result. After that, a spin-orbit calculation is performed using the polarized calculation as the initial state and with the spins rotated by the specified angles. This procedure can be done for several angles and is most easily done by inserting a *for loop* over angles. This is done in the script `FePt.py`, where the angles are setup to be the same as used in the **MagneticAnisotropyEnergy** object above. Download the script and save it to your working directory and submit the calculation with the **Job Manager**.

On a single node with 24 cores the calculation takes around 10 minutes, i.e. approximately the same time of the **MagneticAnisotropyEnergy** object. However, for larger structures with more atoms, it can be expected that the self-consistent calculations are significantly more time consuming than the *MagneticAnisotropyEnergy* study object.

Compare FT and TotalEnergy results

When the calculation is done we are ready to compare the FT results from the **MagneticAnisotropyEnergy** object with the self-consistent total energies just calculated.

Download and run the script `FePt-SO-SCF-loop-over-angles.py` to obtain the plot shown below.



It is obvious from the plot that the FT results matches very well with the self-consistent total energies. This finding is in agreement with already published results[Page 368, 3],[5].

You have now learned how to perform MAE calculation using (i) the *MagneticAnisotropyEnergy* study object which uses the force theorem and non-selfconsistent spin-orbit calculations and (ii) from total energy differences of self-consistent spin-orbit calculations with different spin orientations. The two approaches are seen to give very similar results, but using the *MagneticAnisotropyEnergy* study object allows for more analysis, such as atom- or orbital contributions to the MAE. More examples of this are provided in the tutorial *Magnetic Anisotropy Energy of Fe-MgO-Fe MTJ structure*.

Convergence of results

As mentioned above, the MAE values are very small quantities on the order of 1 meV/atom. As has been noticed in several papers, e.g.[Page 368, 3],[5], it often requires a very high k-point sampling to converge the MAE values. In this section we investigate the convergence behavior of the most important numerical parameters. These are parameters on the **LCAOCalculator**:

•K-point sampling

•Density mesh cutoff

•Basis set

•Occupation function broadening

Tip: For more information on these parameters, you can see the relevant technical notes and manual pages:

•*NumericalAccuracyParameters*

•*Pseudopotentials and basis sets available in QuantumATK*

•*Occupation Methods*

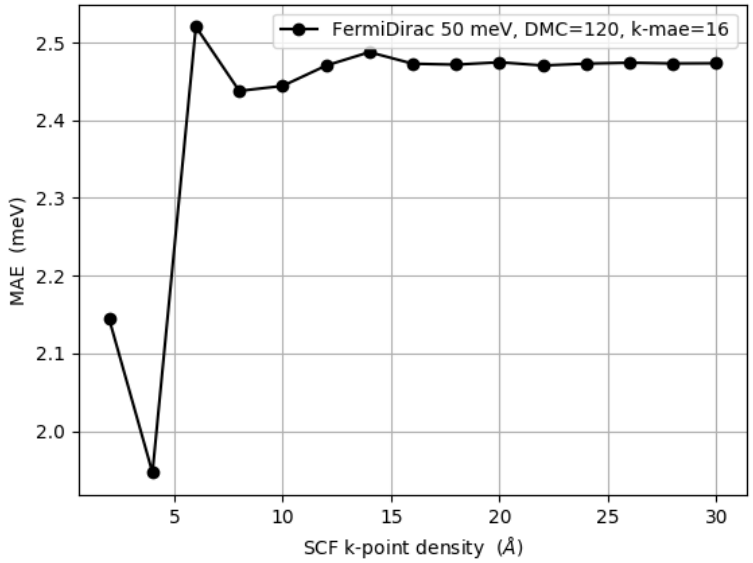
For the **MagneticAnisotropyEnergy** object the main parameter to converge is

•K-point sampling in the **MagneticAnisotropyEnergy** used for the force theorem (band energies) calculation.

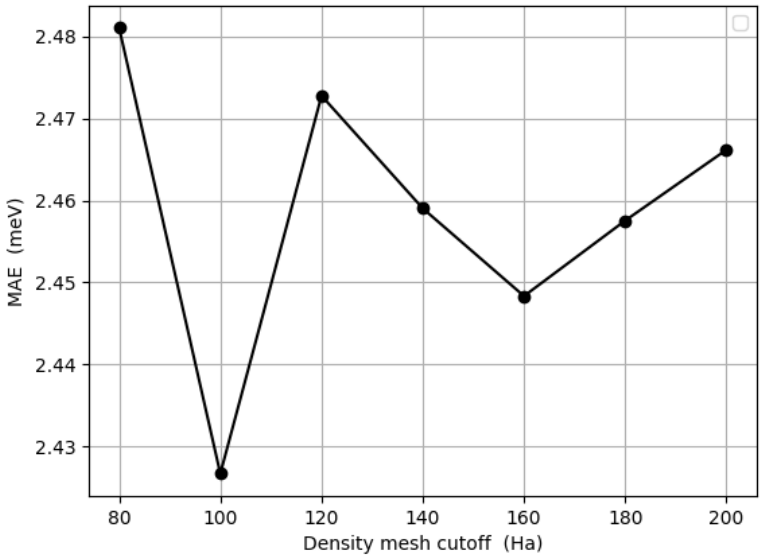
Convergence studies are easily performed in **QuantumATK** by setting up a calculation with the **Script Generator** and sending the script to the **Editor** where loops over the convergence parameters can be setup. In this section we will focus on the result of such calculations with the aim of bringing awareness to the importance of checking for convergence. All calculations in this section have been performed with the occupation function in the `FermiDirac` (default) with a broadening of 50 meV (corresponding to 580 Kelvin).

We begin with the parameters on the **LCAOCalculator**. In the figure below we show the calculated MAE as function of *k-point density*. We emphasize that the default of 4 Å in the calculator is not enough in this case. The value of 7 Å used in the above calculation is reasonably well converged, although for fully converged results a k-point density > 15 Å is required. The data have been obtained with a density mesh cutoff of 120 Hartree and a k-point density in the **MagneticAnisotropyEnergy** object of 16 Å.

[5] L.ăLeăLaurent, C.ăBarreteau, and T.ăMarkussen. Magnetocrystalline anisotropy of Fe, Co, and Ni slabs from density functional theory and tight-binding models. *Phys. Rev. B*, 100:174426, Nov 2019. URL: https://link.aps.org/doi/10.1103/PhysRevB.100.174426, doi:10.1103/PhysRevB.100.174426.



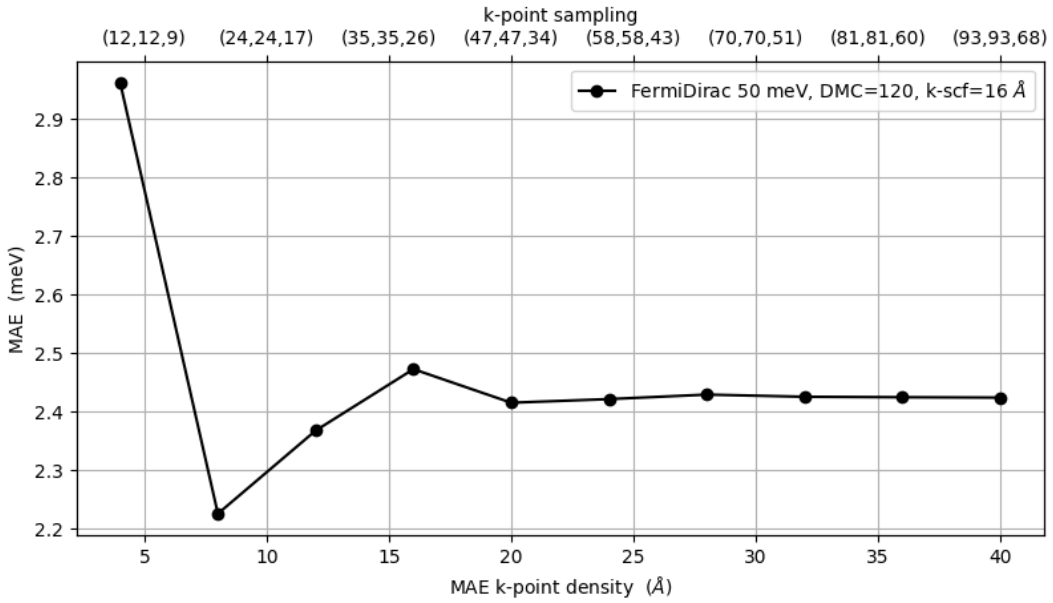
The figure below show dependence of the MAE on the *density mesh cutoff*. Although the data might not look fully converged, we note that the absolute variation is much smaller than in the figure above. The default value of 120 Hartree seems to be converged to no more than an 0.05 meV error. The data for this figure have been obtained with a k-point density of 16 Å in both the **LCAOCalculator** and in the **MagneticAnisotropyEnergy** object.



Another numerical parameter to check for convergence is the LCAO *basis set size*. The table below show the MAE results obtained with *Medium, High, and Ultra basis sets*. We observe that the calculated MAE value are almost the same for all LCAO basis set sizes with variations less than 0.1 meV indicating that the Medium basis used above is sufficient.

LCAO Basis set		
Medium	High	Ultra
2.55 meV	2.49 meV	2.57 meV

We next address the convergence of the MAE values with respect to the *k-point sampling* in the **MagneticAnisotropyEnergy** object. The figure below show results obtained with a density mesh cutoff of 120 Hartree and a k-point density of 16 Å in the **LCAOCalculator**. We see that the k-point density of 17 Å used above is reasonably well converged and above 20 Å, the results are essentially constant.

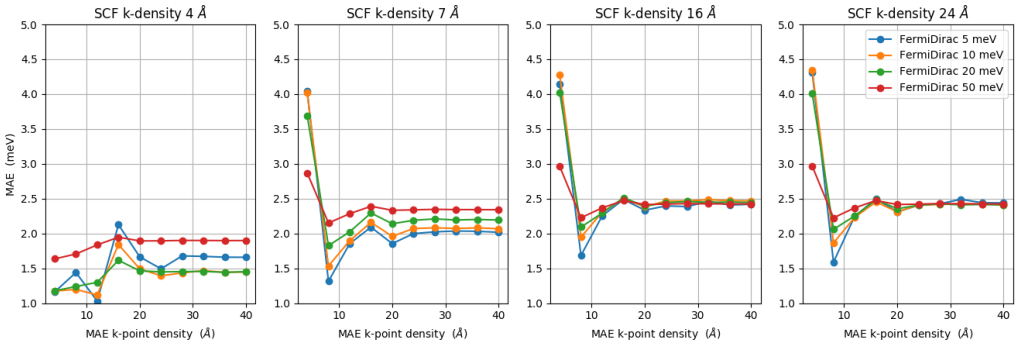


We finally consider the convergence of the broadening used in the occupation method, in this case the *Fermi-Dirac* occupation function. The main reasoning for using a finite broadening in the occupation function is to be able to use a lower k-point sampling. This is illustrated in the figure below. The figure shows four subplots. The data in each subplot has been calculated with a constant k-point density in the self-consistency (SCF) loop and the non-self-consistent spin-orbit calculation. This k-point density is shown in the title of each sub-plot. In a particular sub-plot, we show the MAE vs. k-point density used in the **MagneticAnisotropyEnergy** (MAE) object for four different occupation function broadenings: 5, 10, 20, and 50 meV.

We first observe that for high enough SCF k-point densities (16 Å or 24 Å), and sufficiently high MAE k-point density (> 10 Å), all curves are essentially on top of each other, showing that the value of the occupation function broadening is less important. However, for an intermediate SCF k-point density of 7 Å, we see that the curve for 50 meV broadening is closer to the converged MAE value of 2.5 meV than the curves with lower broadening.

At the lowest SCF k-point density of 4 Å (left-most plot) we see that none of the curves reaches the converged value of 2.5 meV.

Important: Convergence of SCF k-point density and occupation function broadening goes hand in hand. When changing the occupation function broadening, the required k-point sampling also changes.



COSMICS project

The **MagneticAnisotropyEnergy** study object has been developed within the project COSMICS founded by the European Unions Horizon 2020 research and innovation program under Grant Agreement No. 766726. Details about the object can be found here *MagneticAnisotropyEnergy*. Application of the *MagneticAnisotropyEnergy* object to metallic slabs and comparison with QuantumEspresson and tight-binding calculations are presented in ref[5].

More information about the COSMICS project can be found here: http://cosmics-h2020.eu/

References

3.1.12 Magnetic Anisotropy Energy of Fe-MgO-Fe MTJ structure

Version: S-2021.06

In this tutorial you will learn how to use the *MagneticAnisotropyEnergy* study object to calculate the perpendicular magnetic anisotropy (PMA) energy of a Fe-MgO-Fe magnetic tunnel junction (MTJ) structure.

You will learn how to analyze site- and orbital projected magnetic anisotropy energy (MAE) values and compare the results to `OrbitalMoment` based calculations.

The MAE is of central interest for both fundamental and practical reasons, with STT-MRAM being one of the most important technological examples.

Note: This tutorial builds on the introductory MAE tutorial treating a bulk system: *Bulk Magnetic Anisotropy Energy*, and it is assumed that the reader is familiar with its contents. If you are not already familiar with MAE calculations in QuantumATK, we suggest you go through that tutorial before continuing here.

Contents

•*Magnetic Anisotropy Energy of Fe-MgO-Fe MTJ structure*

–*Introduction*

–*Fe-MgO-Fe MTJ structure*

–*MagneticAnisotropyEnergy calculation*
–*What causes the PMA?*
–*COSMICS project*

Introduction

One of the key requirements of an STT-MRAM device is to be able to store information. This requires a sufficiently high energy-barrier between the 1 and 0 states of the system, such that a switch does not happen due to random thermal fluctuations. The MAE is a key element in determining the energy barrier and thus the thermal stability of a STT-MRAM memory unit. The stability factor is given by[1]:

$$\Delta = \frac{H_K M_S V}{2 k_B T}$$

where H_K is the anisotropy field, M_S is the saturation magnetization, V is the cell volume, k_B is the Boltzmann constant and T is the temperature.

When the magnetization direction is perpendicular to the interfaces of different materials, the cell design is called PP (perpendicular to plane) and is the design in focus for this tutorial[Page 382, 1].

For PP cells, H_K is determined by the PMA which has three main contributions: (i) contribution from the bulk of the magnetic material, bulk PMA, (ii) contribution from the interfaces and surfaces, interfacial PMA (iPMA), and (iii) from the shape anisotropy. Bulk Fe in the BCC phase has vanishing bulk PMA, while the iPMA can be substantial and is the focus of this tutorial.

In this tutorial we will calculate the PMA using the *MagneticAnisotropyEnergy* study object, which uses the force theorem to evaluate the energy difference between two magnetization directions. More information about the force theorem and validation of its use can be found in the reference manual: *MagneticAnisotropyEnergy* and in the tutorial: *Bulk Magnetic Anisotropy Energy*.

Fe-MgO-Fe MTJ structure

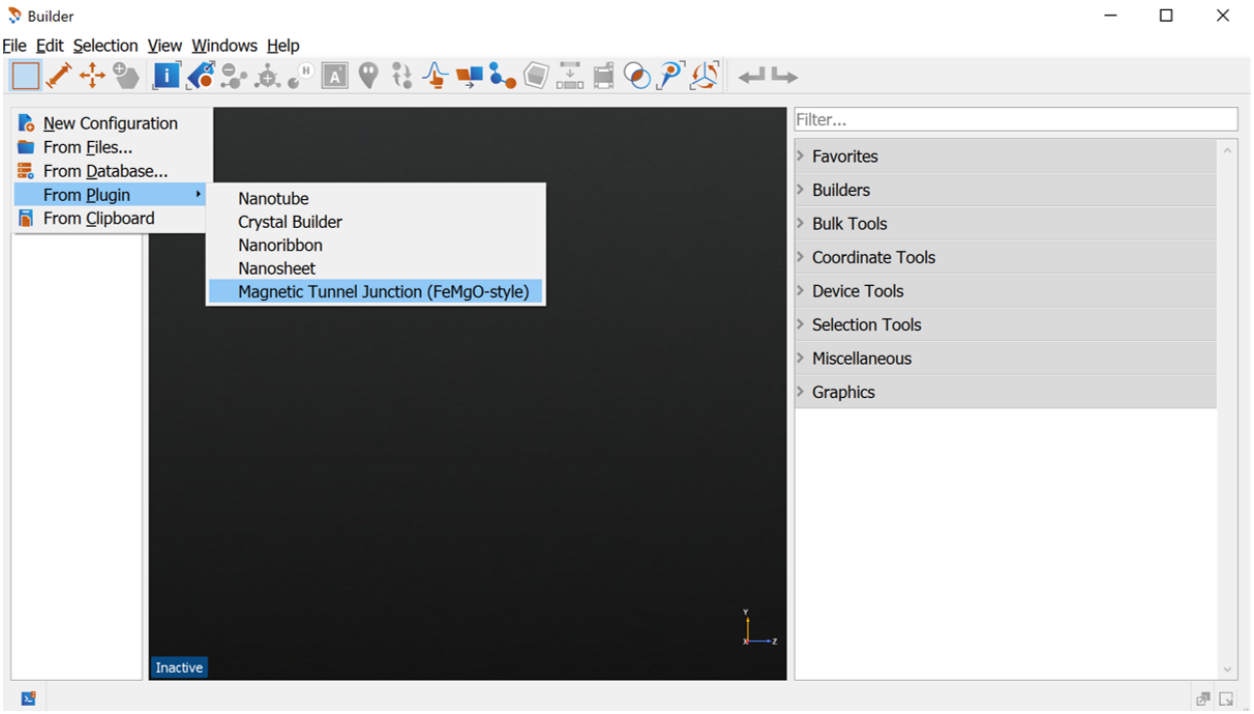
We begin by setting up the atomic structure of a Fe-MgO-Fe structure. Start by opening the **Builder**. In the Builder do the following (see image below):

•Press the **Add** icon

•Select **From Plugin**

•Select **Magnetic Tunnel Junction (FeMgO-style)**

[1] AăV Khvalkovskiy, DăApalkov, SăWatts, RăChepulskii, RăS Beach, AăOng, XăTang, AăDriskill-Smith, WăH Butler, PăB Visscher, DăLottis, EăChen, VăNikitin, and MăKrounbi. Basic principles of STT-MRAM cell operation in memory arrays. *Journal of Physics D: Applied Physics*, 46(7):074001, feb 2013. URL: https://doi.org/10.1088/0022-3727/46/7/074001, doi:10.1088/0022-3727/46/7/074001.



In the **Magnetic Tunnel Junction Builder** apply the following changes (see image below)

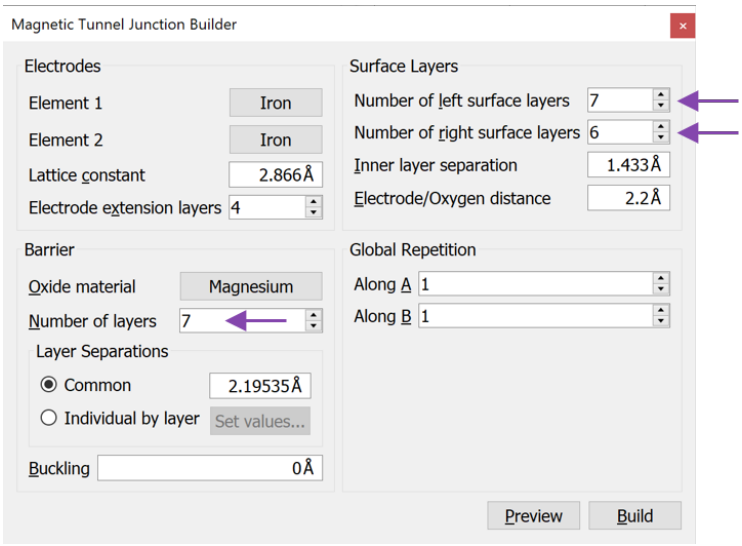
•For the **Barrier** set the **Number of layers** to *7*

•**For the Surface Layers:**

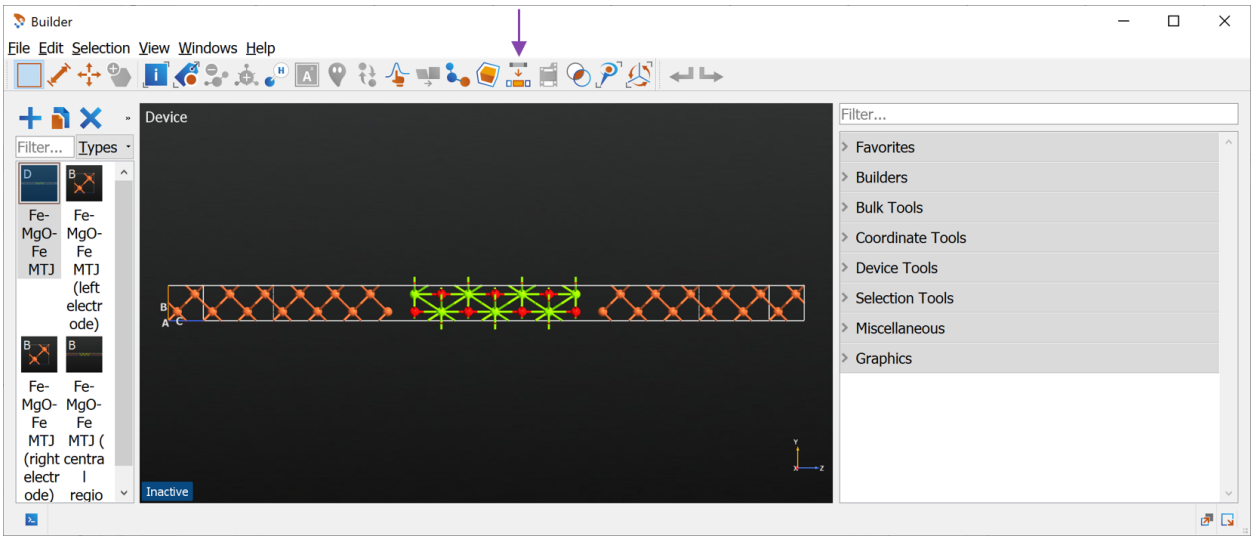
–Set **Number of left surface layers** to *7*

–Set **Number of right surface layers** to *6*

•Press **Build**

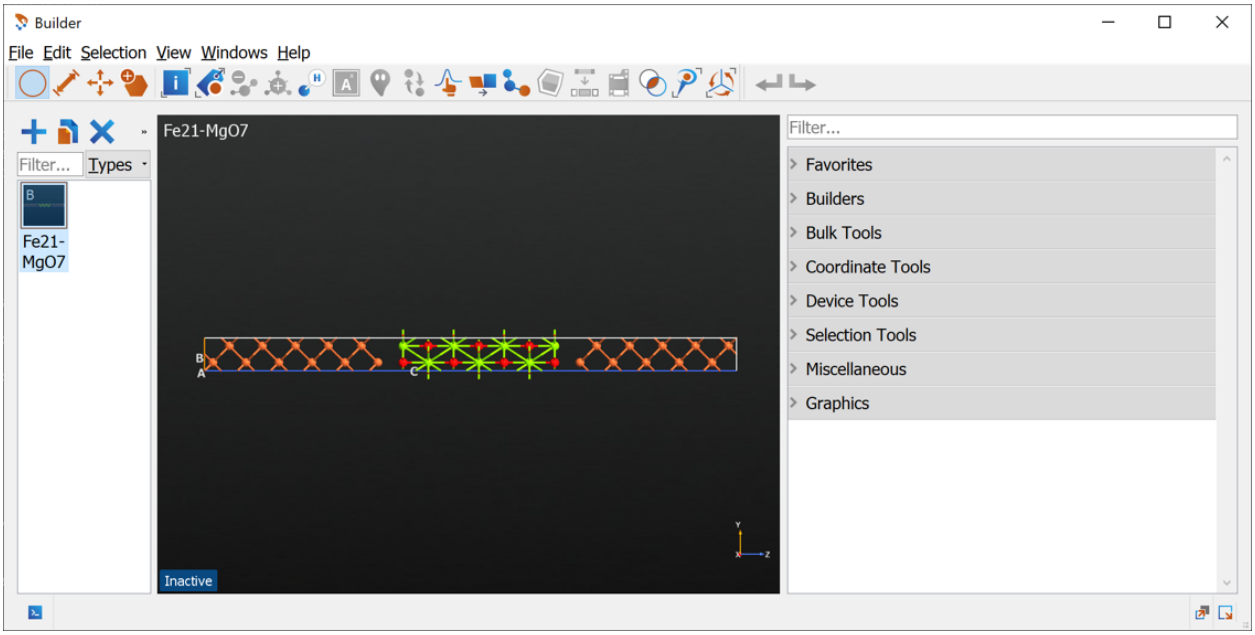


You now have a **DeviceConfiguration** in the Builder. Split the device into electrodes and central region using the button indicated in the figure below.



Once the device has been split, you can delete the original device structure as well as the electrodes. Finally rename the remaining central region to *Fe21-MgO7* (press `F2`), indicating that the structure has 21 Fe layers and 7 MgO layers. Your builder should now look as below.

Note: The although we setup the MTJ structure with 6 and 7 surface layers, we ended up with a bulk configuration of 21 layers. The reason is that we have also included the 4 electrode extension layers on both sides. The total number of layers is thus the sum of left and right surface layers and the electrode extension layers, $6 + 7 + 4 + 4 = 21$.



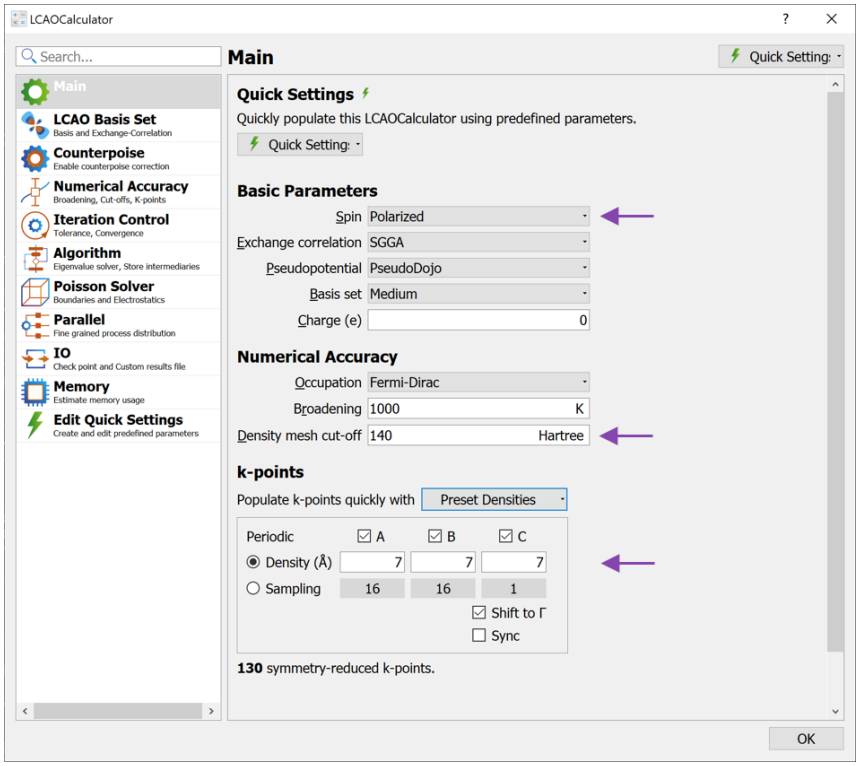
Send the structure to the **Script Generator** using the **Send-to** button.

Structural relaxation

The next step will be to relax the structure. In the **Script Generator** do the following:

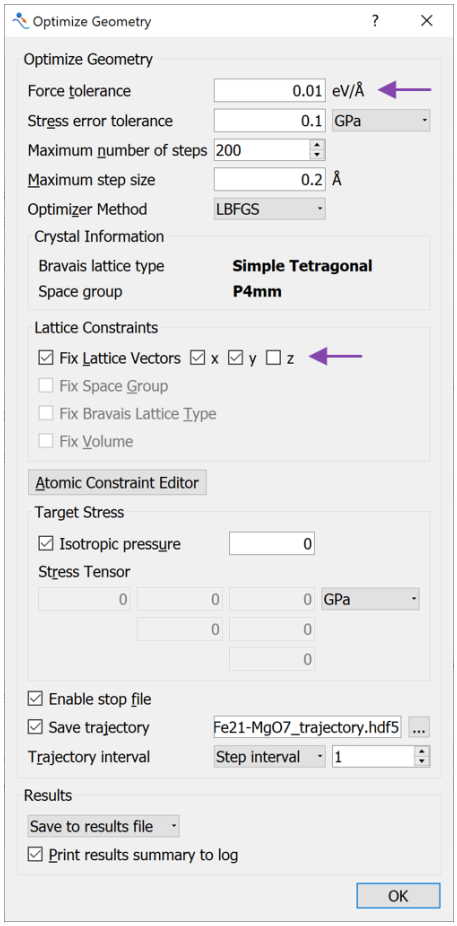
- **Add an** **LCAOCalculator block and open it (see image below):**

  - Change the **Spin** to *Polarized*. This is necessary since iron is magnetic and cannot be correctly modeled with an unpolarized calculation.
  - Increase the **Density mesh cut-off** to *140 Hartree*. For this configuration, the geometry optimization converges faster with this increase.
  - Increase the **k-point density** to *7 Å* (select from the *Preset Densities* combo box).
  - Leave the remaining settings as defaults and close the widget again.



- **Add an** **OptimizeGeometry block to the script and open it to apply the following changes (see image below):**

  - Change the **Force tolerance** to *0.01 eV/Å*
  - Uncheck the **Fix Lattice Vectors** for the z direction. This allows the C-lattice vector in the unit cell to be optimized. The A- and B- lattice vectors are kept fixed, although the atoms are allowed to move also in the x- and y-directions.



You are now ready to submit the calculations. Send the calculation to the **Job Manager** using the **Send-to** button and run the calculation. The calculation can take several hours to complete even when run with multiple MPI processes. For this reason we provide here the relaxed structure: `Fe21-MgO7-relaxed.py`. Either wait for the Geometry Optimization job to finish or download the relaxed configuration to obtain the relaxed configuration. Transfer the relaxed configuration to the **Script Generator**.

MagneticAnisotropyEnergy calculation

In the **Script Generator** do the following:

- **Add an LCAOCalculator and open it (see image below):**

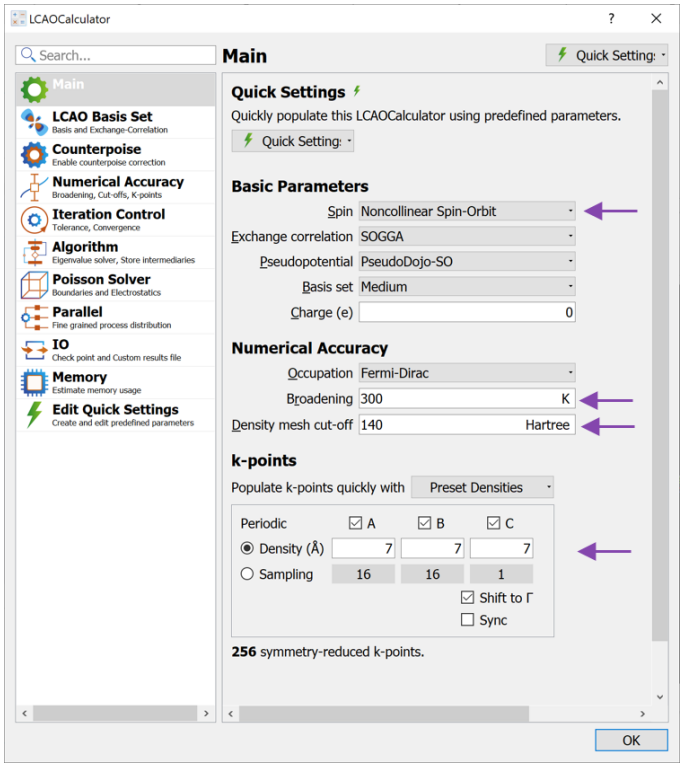
  – Change the **Spin** to *Noncollinear Spin-Orbit*.

  – Set the **Broadening** to *300 K*

  – Increase the **Density mesh cut-off** to *140 Hartree*.

  – Increase the **k-point density** to *7 Å* (select from the **Preset Densities** combo box).

  – Leave the remaining settings as defaults and close the widget again.

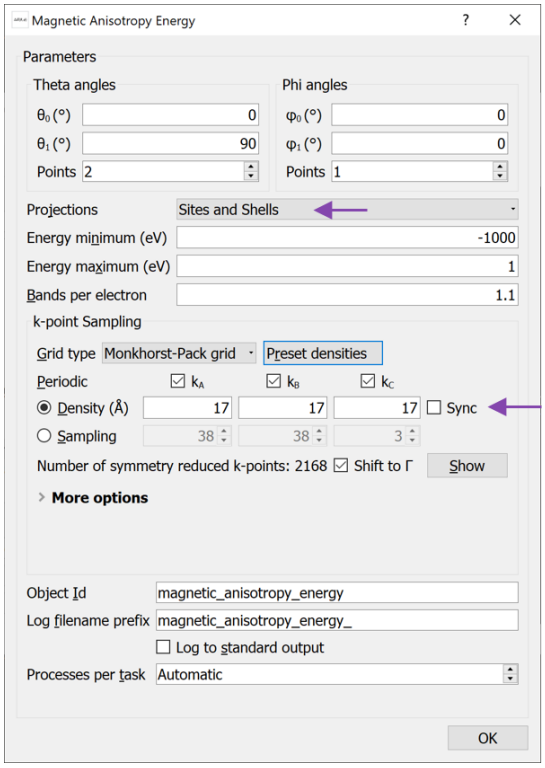


•Add a MagneticAnisotropyEnergy study object and open it (see image below):

–Change **Projections** to *Sites and Shells*. This will calculate the contribution to the MAE from the s, p, d and f shells on each atom.

–Increase the **k-point density** to *17 Å* (choose this value from the **Preset densities**).

–Leave the remaining settings as defaults and close the widget again.



Note: The total MAE will always be calculated irrespective of which **Projections** have been selected. The choice of **Projections** only determines which kind of analysis can be performed. The default choice of `Sites` allows a decomposition of the MAE onto atomic sites, but no information about which orbitals are involved. Choosing `Sites and Shells` gives additional information about which shells (s, p, d, f) contribute, while choosing `Sites and Orbitals` give detailed information about individual orbitals, e.g. p_x, p_y, p_z etc.

If more detailed projections are chosen, more memory will be required and the calculation time will also increase.

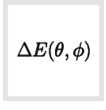
If you are only interested in the total MAE value, it can be beneficial to choose `No Projection` as this will lower the required memory and reduce the computation time.

Send the calculation to the **Job Manager** using the **Send-to** button and run the calculation. The calculation can take several hours. On a single node with 24 cores, the calculation takes about two hours.

Analyzing the results

Once the calculation has finished we are ready to analyze the results:

- Check the `hdf5` file on the **LabFloor**.
- Mark the **MagneticAnisotropyEnergy** object and open the analyzer **Magnetic Anisotropy Energy**



The **Magnetic Anisotropy Energy** analyzer opens with a window showing the site-resolved contributions to the MAE (if `No Projection` was chosen when setting up the MAE object, this plot will be empty). The type of projections can be selected to the left of the plot.

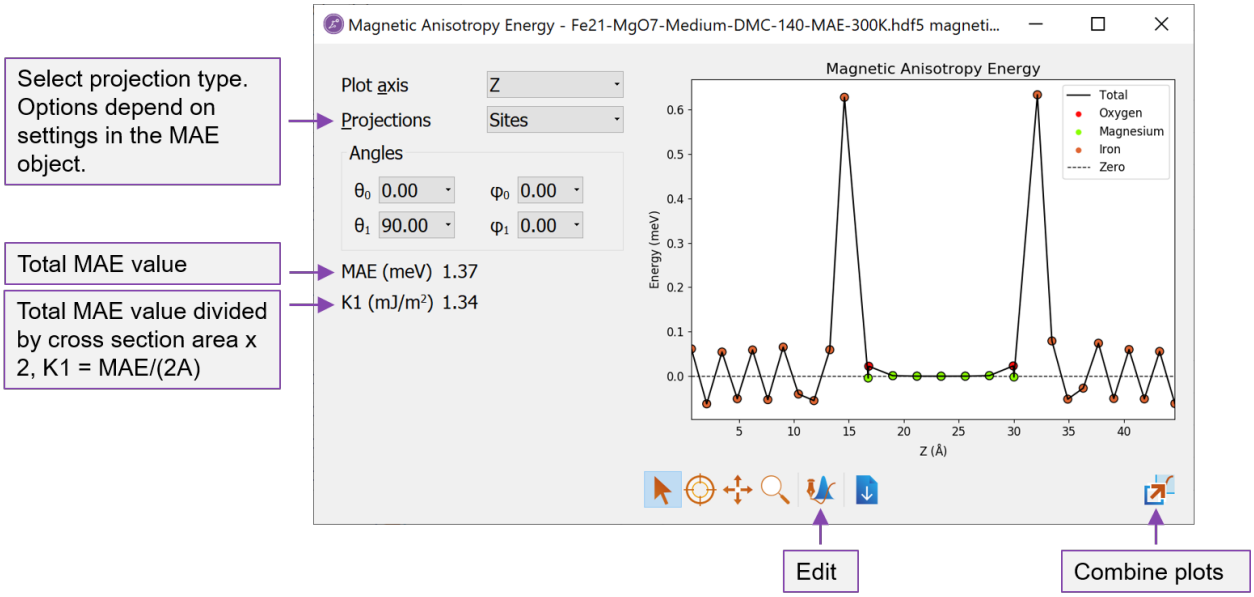


Fig. 3.4: The **Magnetic Anisotropy Energy** analyzer shows both the total MAE and the projected values.

In the lower left, the **total MAE** is shown in units of meV. Just below is shown the **K1** value, which here is defined as $K1 = MAE/(2 \cdot A)$ where A is the cross section area. The factor $1/2$ accounts for the fact that most systems will have two equivalent interfaces. If this is not the case, the **K1** value needs to be determined manually.

From the analyzer, we see that:

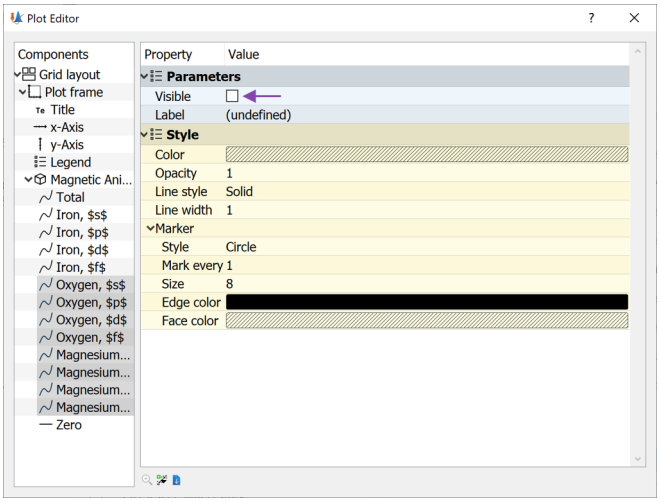
- The MAE value is positive showing that the perpendicular (out-of-plane) magnetization is the energetically most favorable and the Fe-MgO-Fe structure is thus suitable for a PP cell design (note however, that the shape anisotropy, which favors in-plane magnetization, is not taken into account).
- The contribution to the positive MAE mainly comes from the interface Fe layers.
- The bulk Fe layers furthest away from the Fe-MgO interface have a smaller contribution to the MAE and shows an oscillating behaviour, summing up to a very small contribution. This means that K1 equals the surface anisotropy energy density, σ.
- The Mg and O atoms essentially do not contribute to the MAE.

Since the MagneticAnisotropyEnergy object was calculated with **Projections** set to `Sites and Shells` we also have the possibility to analyze which orbital shells are responsible for the MAE:

- Change **Projections** to *Shells* in the MAE analyzer window.

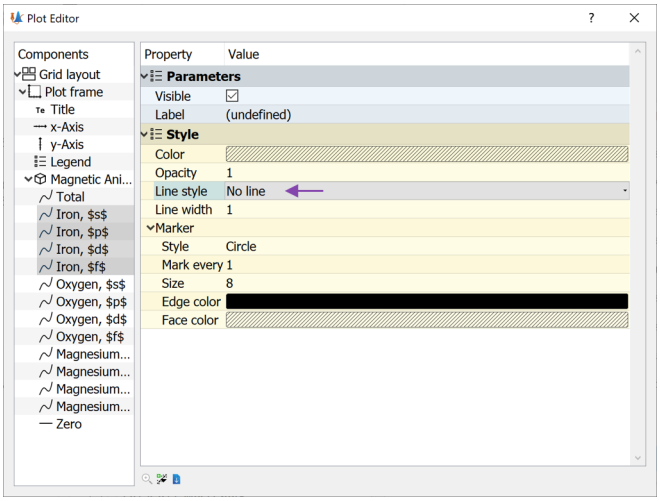
Several new lines are now added to the plot and at first sight it appears difficult to analyze. However, in **QuantumATK** it is very easy to customize plots. Since the Mg and O atoms are unimportant for the total MAE, we do not want to show results for those. In order to edit the plot do the following:

- Press the **Edit** button (see image above) or simply press **e**.
- Mark all the *Oxygen* and *Magnesium* lines
- Un-check the **Visible** box (see image below)

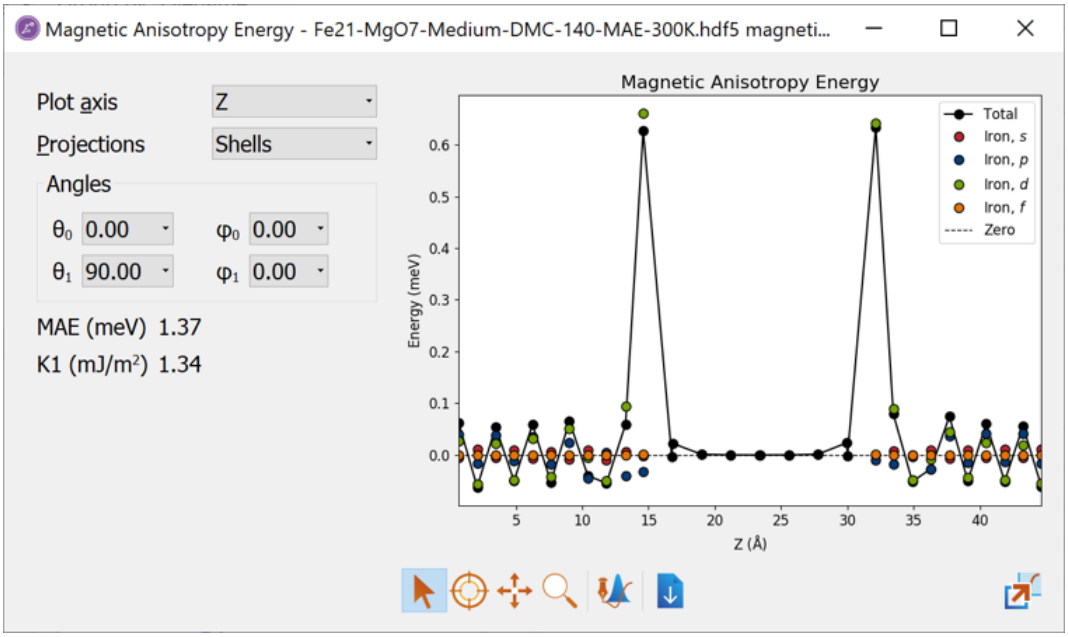


•Now mark all the *Iron* lines

•Change the **Line style** to `No line` (see image below)



We have now removed all the oxygen and magnesium data from the plot, as well as the lines between the points for the iron atoms. Now the analyzer should look like the image below.



It is now evident that the finite MAE value essentially only comes from the Fe d-orbitals. Such information can indeed be very useful since it provides insight into how the MAE can be optimized by modifying the Fe d-orbitals.

Tip: Such analysis and experiments have been the focus of the Horizon 2020 *COSMICS project*.

Orbital moment analysis

We learned from the analysis above that the MAE in the Fe-MgO-Fe MTJ structure essentially comes from the Fe d-orbitals. Due to this simple contribution and since the spin-orbit coupling in iron is rather weak, it is possible to estimate the MAE with a more perturbative approach using the Bruno formula[2]:

$$MAE = -\frac{\xi_{Fe-d}}{4}(L_x - L_z)$$

Here L_x is the x-component of the orbital moment, when the spin orientation is along the x-axis ($\theta = 90°$) and L_z is the z-component of the orbital moment, when the spin orientation is along the z-axis ($\theta = 0°$). ξ_{Fe-d} is the spin-orbit coupling strength from the Fe d-orbitals.

The orbital moment can be calculated in QuantumATK, for systems with **Spin** set to *Noncollinear Spin-Orbit*. Usually a non-selfconsistent spin-orbit calculation restarting from the polarized one is sufficient.

Since we already have non-selfconsistently updated spin-orbit calculations from the MagneticAnisotropyEnergy we can use these for an **OrbitalMoment** analysis. In the script, where the MagneticAnisotropyEnergy was calculated, we can insert the following lines at the end of the script:

```
# Get the non-selfconsistently updated spin-orbit configurations for the specified
# (theta, phi) angles.
configuration_so_0 = magnetic_anisotropy_energy.updatedSpinOrbitConfiguration(0*Degrees,␣
↪0*Degrees)
```

(continues on next page)

[2] Patrick Bruno. Tight-binding approach to the orbital magnetic moment and magnetocrystalline anisotropy of transition-metal monolayers. *Phys. Rev. B*, 39:865–868, Jan 1989. URL: https://link.aps.org/doi/10.1103/PhysRevB.39.865, doi:10.1103/PhysRevB.39.865.

(continued from previous page)

```
configuration_so_90 = magnetic_anisotropy_energy.
↪updatedSpinOrbitConfiguration(90*Degrees, 0*Degrees)

# ------------------------------------------------------------
# Orbital Moment
# ------------------------------------------------------------
kpoints = KpointDensity(
    density_a=7.0*Angstrom,
    force_timereversal=False,
    )

orbital_moment = OrbitalMoment(
    configuration=configuration_so_0,
    kpoints=kpoints,
    temperature=300.0*Kelvin,
)
nlsave('Fe21-MgO7.hdf5', orbital_moment, object_id='Orbital moment 0')

orbital_moment = OrbitalMoment(
    configuration=configuration_so_90,
    kpoints=kpoints,
    temperature=300.0*Kelvin,
)
nlsave('Fe21-MgO7.hdf5', orbital_moment, object_id='Orbital moment 90')
```

Note: The MagneticAnisotropyEnergy is a study object, which has a number of task to complete, when called with `magnetic_anisotropy_energy.update()`. The study upject keeps track on which tasks have already been finished and will not do them again. Therefore, when the lines above to your script with the MagneticAnisotropyEnergy calculation, the MAE calculation will not do any work (except for detecting that it has already finished all the tasks), and the script will quickly proceed to the last part.

The orbital moment can also be setup using the **Script Generator**. In this case, it is important to remember that the analysis of orbital moment using the Bruno formula requires two calculations for different spin orientations. Such a script is provided here, where a self-consistent polarized calculation is followed by two non-selfconsistent spin orbit calculation for two different initial spin orientations: `orbital-moment-script.py`.

Once the orbital moments have been calculated, the results can be analyzed with the following script.

```
import pylab

# File name.
filename = 'Fe21-MgO7.hdf5'

orbital_moment_z = nlread(filename, object_id='Orbital moment 0')[0]
orbital_moment_x = nlread(filename, object_id='Orbital moment 90')[0]

# Spin-orbit coupling strength
e_so = 60 * meV

# Get the total moment
L_z = orbital_moment_z.orbitalMoment()[2] / bohr_magneton
```

(continues on next page)

(continued from previous page)

```
L_x = orbital_moment_x.orbitalMoment()[0] / bohr_magneton

# Calculate the MAE using Bruno's formula
mae = -e_so / 4 * (L_x - L_z)

nlprint('Total MAE (Bruno formula) = {: .2f} meV'.format(mae.inUnitsOf(meV)))


# Get the atom resolved moments
L_z_atom = orbital_moment_z.atomResolvedOrbitalMoment()[:, 2] / bohr_magneton
L_x_atom = orbital_moment_x.atomResolvedOrbitalMoment()[:, 0] / bohr_magneton

mae_atom = -e_so / 4 * (L_x_atom - L_z_atom)

# Get the z-coordinates
z = orbital_moment_z._configuration().cartesianCoordinates()[:, 2].inUnitsOf(Ang)

pylab.figure()
pylab.plot(z, mae_atom, 'ko-')
pylab.xlabel('z ($\\AA$)')
pylab.ylabel('MAE (meV)')
pylab.show()
```

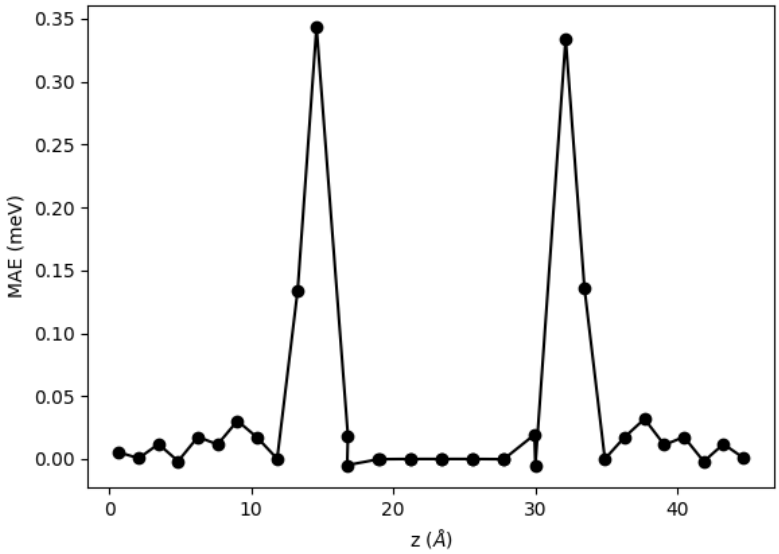
Note: In the analysis we use that the spin-orbit coupling energy of the Fe d-orbitals is 60 meV[3].

The outcome of the script is that the total PMA obtained from the Bruno formula is

```
Total MAE (Bruno formula) = 1.16 meV
```

The total MAE (PMA) is thus in fairly good agreement with the Force theorem (FT) result obtained from the *MagneticAnisotropyEnergy* object of 1.37 meV. It is also possible to get an atom-resolved plot as shown below, which is also produced by the same script:

[3] G.ăAutes, C.ăBarreteau, D.ăSpanjaard, and M.ăDesjonqueres. Magnetism of iron: from the bulk to the monatomic wire. *Journal of Physics: Condensed Matter*, 18(29):6785–6813, jul 2006. doi:10.1088/0953-8984/18/29/018.



The atom-resolved MAE obtained from the Bruno formula clearly shows a similar behavior as the site-resolved MAE plot shown above: The main contribution comes from the interface Fe layers, while the bulk Fe has a vanishing contribution. However, there are some quantitative differences. The results obtained from the orbital moment and the Bruno formula show a rather large contribution from the second interface layer, which is not evident in the FT results. The oscillations in the bulk part of the Fe also comes out slightly smaller from the Bruno formula than from the FT.

The fairly good agreement between the Bruno formula and the FT method serves to validate both approaches. It should, however, be noticed that such a good agreement is not always the case. In calculations including heavier elements with stronger spin orbit coupling, it has been shown that the Bruno formula does not agree with the FT method[4] or self-consistent total energy calculations. The Bruno formula should therefore be used with care, while the FT method used in the MagneticAnosotropyEnergy object is in good agreement with total energy calculations, also for heavy elements[Page 394, 4]. For more details about comparing the FT method with total energies see the tutorial *Bulk Magnetic Anisotropy Energy*.

What causes the PMA?

In the above sections we have seen that the contribution to the interfacial PMA comes mainly from the interface Fe layers, while bulk Fe has much smaller contribution. But what causes the positive interfacial PMA? There can be (at least) three sources:

1. Fe-O hybridization. The Fe d-orbitals hybridizes with the O orbitals. In previous studies it has e.g. been shown theoretically and experimentally that hybridization with graphene or C60 increases the PMA of Co surfaces.
2. Strain and relaxation effects. The atomic positions are slightly distorted close to the surface compared to the bulk coordinates.
3. Interface asymmetry. This asymmetry exist at any interface, also at a Fe surface, i.e. a Fe - vacuum interface.

In this section we will try to analyze and quantify the contribution from each of the three sources. Since the calculations are performed in the same way as above, we will focus on the results and analysis. We will do that by comparing the total and projected MAE for different configurations as detailed below.

We will start by analyzing the relative importance of the Fe-O hybridization vs. the interface asymmetry. For doing this, we remove the MgO layers from the relaxed Fe-MgO-Fe configuration, and calculate the MagneticAnisotropyEnergy

[4] M.ăBlanco-Rey, J.I. Cerda, and A.ăArnau. Validity of perturbative methods to treat the spin–orbit interaction: application to magnetocrystalline anisotropy. *New J. Phys.*, 21:073054, 2019. URL: https://iopscience.iop.org/article/10.1088/1367-2630/ab3060.

for this 21 layer Fe slab. Since we do not want to mix in contributions from strain effects, we keep the Fe atoms in the same positions and do not perform any further relaxations. The calculation can be run with this script `Fe21-vacuum.py`.

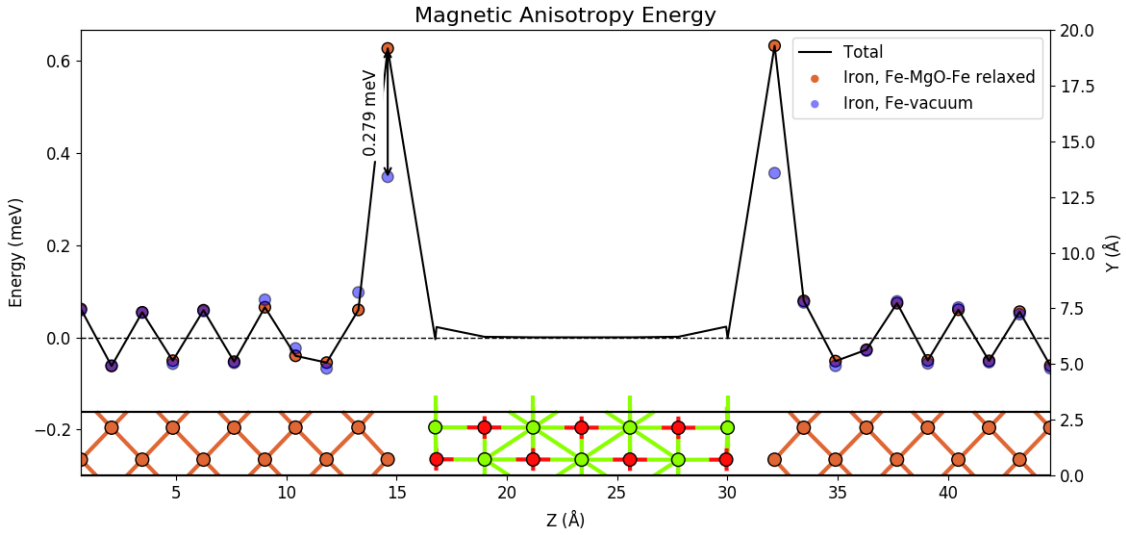
In order to address the effect of strain effects, we also repeat the MagneticAnisotropyEnergy calculation for the unrelaxed configuration produced directly by the MTJ builder plugin. The calculation can be run with this script `Fe21-MgO7-unrelaxed.py`.

The total PMA calculated for the three systems is shown in the table below. The total PMA is largest for the relaxed Fe-MgO-Fe structure.

Structure Fe-MgO-Fe Relaxed	Fe-MgO-Fe Unrelaxed	Fe-vacuum
1.37 meV	0.68 meV	0.79 meV

The figure below compare the relaxed Fe-MgO-Fe structure with the Fe-vacuum structure, where the Fe atoms are in the same positions. We observe that the interface Fe layes have a higher PMA value when hybridizing with the oxygen atoms in the Fe-MgO-Fe structure than when interfacing vacuum in the Fe-vacuum system. For each interface Fe atom the PMA is increased by 0.28 meV. The Fe-O hybridization thus have a significant influence increasing the total PMA from 0.79 meV for the Fe-vacuum structure to 1.37 meV for the Fe-MgO-Fe structure. It is interesting to see that the Fe-O hybridization is a very local effect essentially only affecting the PMA value of the interface Fe layers, while all the other Fe atoms essentially have the same PMA value in the Fe-MgO-Fe and Fe-vacuum structures.

The interface asymmetry, which is present in both the Fe-MgO-Fe structure and the Fe-vacuum structure (although a different asymmetry) is also an important effect since the Fe-vacuum structure also has a reasonably high PMA value of 0.79 meV, whereas bulk Fe has close to zero PMA.



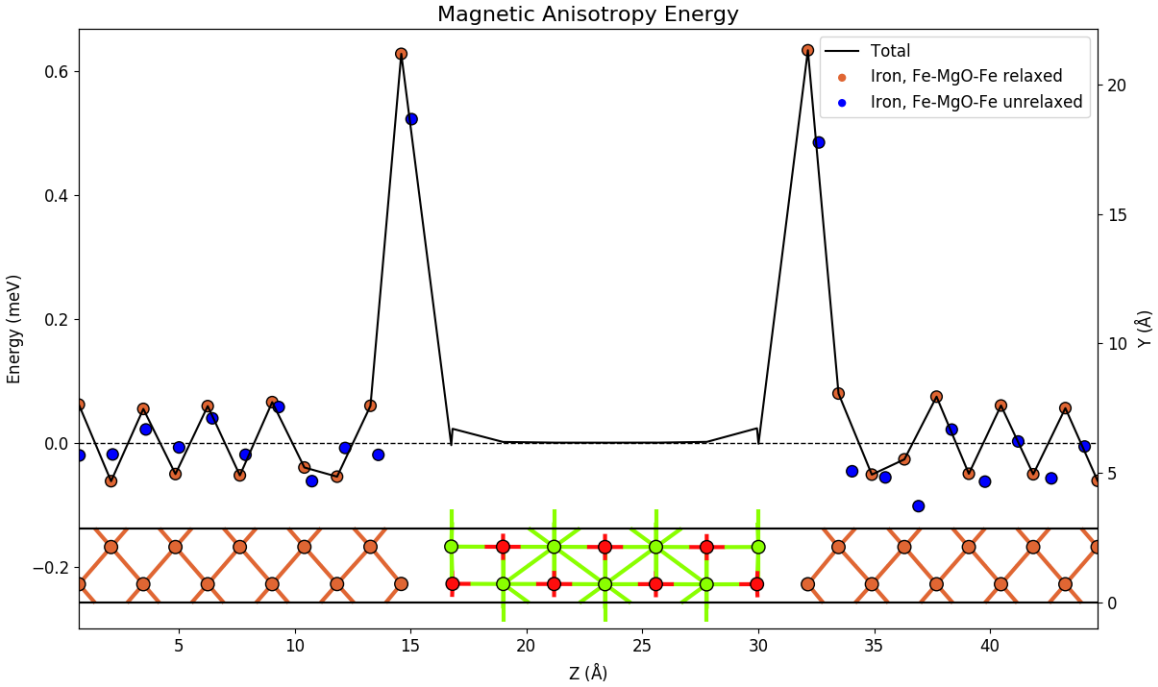
Tip: The figure above shows the structure of the configuration together with the MAE plot. This figure has been made with the following procedure:

- Open the Magnetic Anisotropy Energy analyzer for both the Fe-MgO-Fe and the Fe-vacuum structures and edit the plots individually.
- Combine the two plots by dragging one onto the other using the **Combine plots** button (see Fig. 3.4 above).
- Mark the Fe-MgO-Fe structure on the **LabFloor** and select the **Configuration Plot** from the right column of analyzers.

•Combine the Configuration plot with the MAE plots.

•Adjust y-axis limits to make the plot look as you want (use the edit button or press e - see Fig. 3.4 above)..

Next we address the effect of strain and relaxation effects. From the table above we see that the unrelaxed Fe-MgO-Fe structure has a total PMA value of 0.68 meV, i.e. only about half the value of the relaxed structure. This difference illustrates the importance of including the strain and relaxation effects and it also shows that PMA values are very sensitive to small details in the atomic structure.



The above figure compares the atomic contributions to the PMA for relaxed and unrelaxed Fe-MgO-structure. The unrelaxed structure has smaller PMA values at the interface layers and shows smaller oscillations in the bulk part of the Fe. However, the qualitative picture of the PMA being determined by the interface layers, remain the same for the two configurations.

COSMICS project

The **MagneticAnisotropyEnergy** study object has been developed within the project COSMICS founded by the European Unions Horizon 2020 research and innovation program under Grant Agreement No. 766726. Details about the object can be found on the manual page here *`MagneticAnisotropyEnergy`*. Application of the *MagneticAnisotropyEnergy* object to metallic slabs and comparison with QuantumEspresso and tight-binding calculations are presented in ref[5].

More information about the COSMICS project can be found here: http://cosmics-h2020.eu/

[5] L.ăLeăLaurent, C.ăBarreteau, and T.ăMarkussen. Magnetocrystalline anisotropy of Fe, Co, and Ni slabs from density functional theory and tight-binding models. *Phys. Rev. B*, 100:174426, Nov 2019. URL: https://link.aps.org/doi/10.1103/PhysRevB.100.174426, doi:10.1103/PhysRevB.100.174426.

References

3.1.13 Heisenberg exchange coupling of iron and cobalt

Version: S-2021.06

In this tutorial you will learn how to use the *HeisenbergExchange* analysis object to calculate the Heisenberg exchange coupling parameters of bulk Fe and Co. You will also learn how to calculate the Curie temperature and exchange stiffness from the coupling parameters.

Introduction

When studying magnetic properties of materials, density functional theory (DFT) often provides a good description. From the spin-density it is possible to calculate the zero-temperature magnetization and magnetic ordering and one can calculate the exchange splitting from the bandstructure or density of states. It is, however not a standard analysis to calculate excited state properties such as the exchange stiffness, Curie temperature, spin wave dispersions, or magnetization dynamics.

When studying such quantities the starting point is often the Heisenberg model. The Heisenberg exchange Hamiltonian is given by[1],[2]

$$H = -\sum_{i \neq j} J_{ij} \hat{e}_i \hat{e}_j \tag{3.2}$$

where $\hat{e}_i$ is the normalized local spin vector on atom i, and J_{ij} is the Heisenberg exchange coupling constant. In the following short *Theory* section we will detail how the exchange coupling matrix elements are calculated from DFT calculations and we will also show how to obtain the Curie temperature and exchange stiffness.

Theory

We calculate the exchange coupling matrix elements $J_{i\mathbf{0},j\mathbf{R}}$ between an atom i in the central unit cell labeled $\mathbf{0}$ and atom j in a different unit cell displaced from the central one with a lattice vector $\mathbf{R}$. We use a real-space Greens function implementation[3],[4] of the original Liechtenstein-Katsnelson-Antropov-Gubanov (LKAG) formula[Page 397, 1]:

$$J_{0i,\mathbf{R}j} = -\frac{1}{4\pi} \int_{-\infty}^{E_F} d\epsilon \mathrm{Im}\mathrm{Tr}\left[\Delta_i G_{i,j}(\epsilon, \mathbf{R}) \Delta_j G_{i,j}(\epsilon, -\mathbf{R})\right] \tag{3.3}$$

where i and j label atomic indices within a unit cell, $\mathbf{R}$ is a lattice vector, and where $\Delta_i = H_i^{\uparrow} - H_i^{\downarrow}$ is the on-site difference between the up and down part of the Hamiltonian matrix. More details about the implementation can be found in the reference manual *Notes*.

[1] A.I. Liechtenstein, M.I. Katsnelson, V.P. Antropov, and V.A. Gubanov. Local spin density functional approach to the theory of exchange interactions in ferromagnetic metals and alloys. *Journal of Magnetism and Magnetic Materials*, 67(1):65–74, 1987. URL: http://www.sciencedirect.com/science/article/pii/0304885387907219, doi:https://doi.org/10.1016/0304-8853(87)90721-9.

[2] M.ăPajda, J.ăKudrnovský, I.ăTurek, V.ăDrchal, and P.ăBruno. Ab-initio calculations of exchange interactions, spin-wave stiffness constants, and Curie temperatures of Fe, Co, and Ni. *Phys. Rev. B*, 64:174402, Oct 2001. URL: https://link.aps.org/doi/10.1103/PhysRevB.64.174402, doi:10.1103/PhysRevB.64.174402.

[3] Asako Terasawa, Munehisa Matsumoto, Taisuke Ozaki, and Yoshihiro Gohda. Efficient Algorithm Based on Liechtenstein Method for Computing Exchange Coupling Constants Using Localized Basis Set. *Journal of the Physical Society of Japan*, 88(11):114706, 2019. URL: https://doi.org/10.7566/JPSJ.88.114706, doi:10.7566/JPSJ.88.114706.

[4] XuăHe, Nicole Helbig, MatthieuăJ. Verstraete, and Eric Bousquet. TB2J: A python package for computing magnetic interaction parameters. *Computer Physics Communications*, 264:107938, 2021. URL: https://link.aps.org/doi/10.1103/PhysRevB.64.174402, doi:10.1103/PhysRevB.64.174402.

Curie temperature

The Curie temperature, T_C, of a ferromagnet is the temperature at which the average magnetic moment becomes zero due to random fluctuations of the local magnetic moment directions. T_C can be estimated from the Heisenberg model in different ways. Within the mean field approximation (MFA) one have[2]

$$T_C^{MFA} = \frac{2}{3k_B} \sum_{j\mathbf{R}\neq i\mathbf{0}} J_{i\mathbf{0}j\mathbf{R}} \tag{3.4}$$

The MFA formula typically overestimates the Curie temperature when compared to experiments as will also be evident below. A different approach is based on the random phase approximation (RPA)[Page 397, 2]:

$$\frac{1}{k_B T_C^{RPA}} = \frac{6\mu_B}{M} \frac{1}{N_q} \sum_{\mathbf{q}} \frac{1}{E(\mathbf{q})} \tag{3.5}$$

where $E(\mathbf{q})$ is the spin wave energy at q-point $\mathbf{q}$ and M is the magnetic moment per atom and μ_B is the Bohr magneton. Since an average over the Brillouin zone is performed, the RPA calculation needs to be converged with respect to the number of q-points. This will be illustrated below.

Exchange stiffness

The exchange stiffness describes the energy of a long wavelength spin wave excitation. In particular, if the energy as function of wave number q is written as

$$E(q) = E_0 + A_{ex} q^2 + \ldots$$

the exchange stiffness constant, A_{ex} is seen to be given by the curvature of the $E(q)$ vs. q curve. A_{ex} can be calculated from the Heisenberg exchange parameters as[Page 397, 2]:

$$A_{ex} = \frac{2\mu_B}{3M} \sum_{\mathbf{R}} \sum_{ij} J_{i\mathbf{0},j\mathbf{R}} r_{i\mathbf{0},j\mathbf{R}}^2 \tag{3.6}$$

where $r_{i\mathbf{0},j\mathbf{R}}$ is the distance between atom i in cell $\mathbf{0}$ and atom j in cell $\mathbf{R}$. In practice, (3.6) can be difficult to converge with respect to the sum over lattice vectors $\mathbf{R}$. This is particularly pronounced in the case of iron where the exchange parameters are very long ranged. In this case it is possible to dampen the oscillations and extrapolate the result to zero damping as:

$$\begin{aligned} A_{ex} &= \lim_{\eta\to 0} A_{ex}(\eta) \\ A_{ex}(\eta) &= \frac{2\mu_B}{3M} \sum_{\mathbf{R}} \sum_{ij} J_{i\mathbf{0},j\mathbf{R}} r_{i\mathbf{0},j\mathbf{R}}^2 e^{-\eta r_{i\mathbf{0},j\mathbf{R}}} \end{aligned} \tag{3.7}$$

Below we will see how to perform the extrapolation to $\eta \to 0$ in practice.

Setting up calculations

We are now ready to start calculating the Heisenberg exchange parameters for Fe and Co. In order to enable a direct comparison to ref.[Page 397, 2] we will be studying Co in the FCC phase. We will start by performing calculations and analysis for Co(FCC). The calculations for Fe (BCC) are performed in the same way, but the analysis, and in particular the calculation of the exchange stiffness, will require a special treatment in the case of Fe.

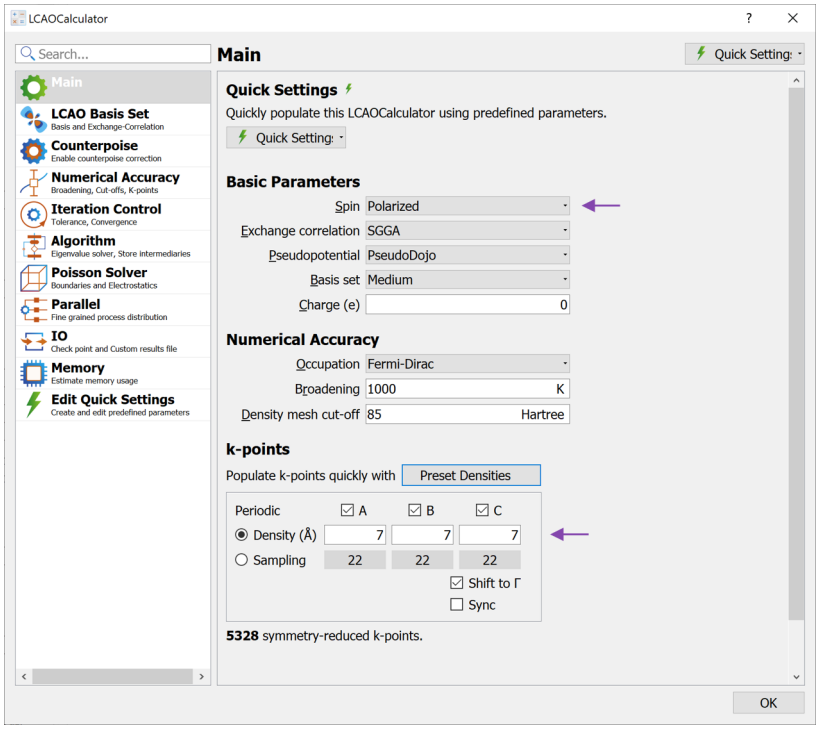
Open the **Builder** and add **Cobalt (alpha)** from the crystals database. Send the configuration to the **Script Generator**.

Now do the following:

- Add an **LCAOCalculator** to the script, open the **LCAOCalculator** widget and apply the changes on the main page (see image below)

– Change **Spin** to *Polarized*

– Increase the **k-point density** to *7* (select from the *Preset Densities*).



– Close the **LCAOCalculator** widget.

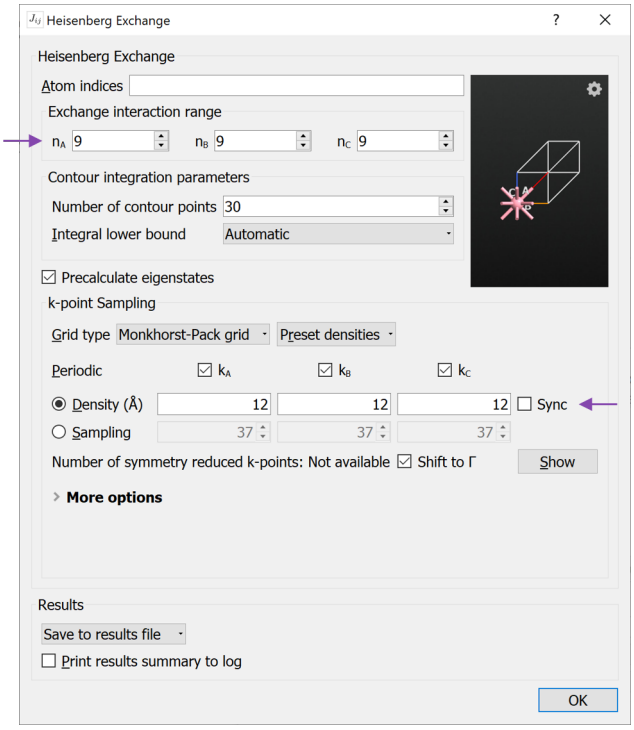
- Add a **HeisenbergExchange** analysis object to the script and open the **HeisenbergExchange** widget. Apply the changes (see image below)

– Change exchange interaction range to $n_A = 9$, $n_B = 9$, $n_C = 9$.

Note: The exchange interaction range determines how many times the input configuration is repeated when performing summations over $\mathbf{R}$ in the formula above. An exchange interaction range of $n_A = 9$ means that translations corresponding to $-5R_A, -4R_A, \ldots 4R_A, 5R_A$ are included, where R_A is the lattice vector in the A-direction. Therefore, the exchange interaction range needs to be an odd integer.

– Increase the **k-point density** to *12* (select from the *Preset Densities*). The **k-point density** determines how many k-points are used when calculating the real-space Greens function (See (4.3) in the reference manual *Notes*).

– You may also check the box **Precalculate eigenstates**. By checking this box, the calculation will run faster. This should however only be used for configurations with relatively few atoms as the memory requirements will be increased.



–Close the **HeisenbergExchange** widget.

•Finally change the name of the **Results file** to `Co-FCC.hdf5` and run the calculation.

Note: The calculations are here performed using the experimental lattice constants. For more complicated structures like interfaces, a structural relaxation should be performed before the **HeisenbergExchange** calculation.

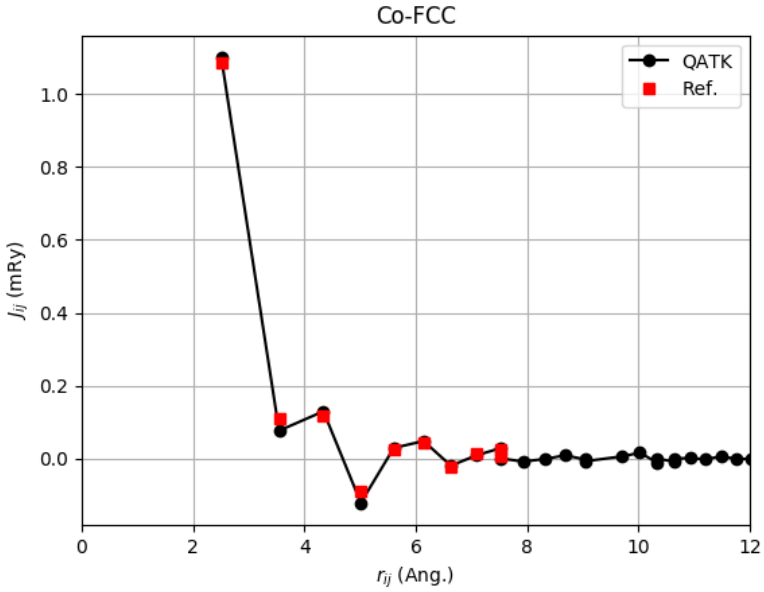
Analyzing the results

When the calculation has finished we are ready to analyze the results. Since no dedicated HeisenbergExchange analyzer has been developed yet, the analysis will be based on python scripts.

Exchange coupling element vs. distance

First, we will look at the exchange interactions elements themselves and make a plot of $J_{i\mathbf{0},j\mathbf{R}}$ vs. $r_{i\mathbf{0},j\mathbf{R}}$. The script `plot_Jij_vs_r.py` loads the just calculated **HeisenbergExchange** object and extracts the relevant data and plots the result. Together with the QuantumATK data we plot the data from ref.[Page 397, 2] for Co-FCC. Download and run the script `plot_Jij_vs_r.py`.

Now run the script. The script will produce a figure like the image shown below. Note that in order to make a direct comparison to the numbers from ref.[Page 397, 2] (Table I) we plot the exchange coupling values in units of mRy.



The positive sign of the nearest neighbor coupling elements shows that Co is a ferro-magnet. Due to the minus-sign in (3.2) the energy is lowered for aligned spins, when the coupling elements are positive leading to ferromagnetic coupling.

Note: There is no clear consensus in the literature about the sign-convention in (3.2) and sometimes the minus-sign is omitted.

Curie temperature

We now turn to the calculation of the Curie temperature. As mentioned above, there are two approximations included in QuantumATK. The mean-field approximation (MFA) formula is the most simple and the result is simply obtained:

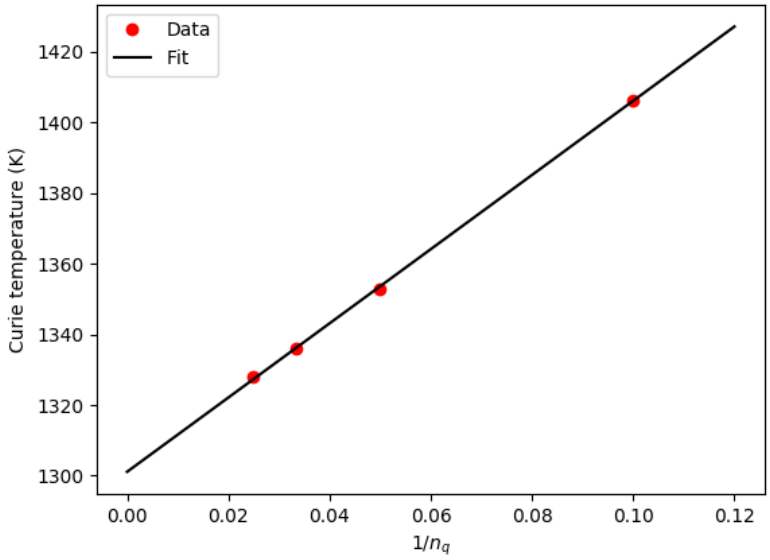
```
T_C_mfa-tutorial = heisenberg_exchange.curieTemperature()
```

The random phase approximation includes a sum over q-points, and convergence with respect to the number of q-points should be checked. The default value might not be high enough for all cases. In the case of Co, it turns out the RPA calculated Curie temperature scales linearly with the inverse number of q-points in each direction. By performing a linear fit, it is therefor possible to extrapolate the result to the limit of infinitely many q-points by evaluating the fit $1/n_q = 0$. This is done in the script `curie_temperature.py`, which both calculates the MFA result and the RPA result extrapolated to infinite q-point sampling. Download and run `curie_temperature.py`. The output of the script is

```
-------------------------------------------------
Curie temperature:
-------------------------------------------------
MFA                                  :  1648.9 K
RPA (default q-grid)                 :  1435.0 K
RPA (extrapolate to infinite q-grid) :  1301.1 K
-------------------------------------------------
```

The figure below is made by the script and illustrates the linear scaling of the RPA Curie temperature as a function of

the inverse number of q-points.



The experimentally determined Curie temperature for Co is 1388-1398 K[Page 397, 2]. It is evident that the RPA result is in reasonable agreement with the experimental value, while the MFA clearly overestimates the Curie temperature.

Exchange stiffness

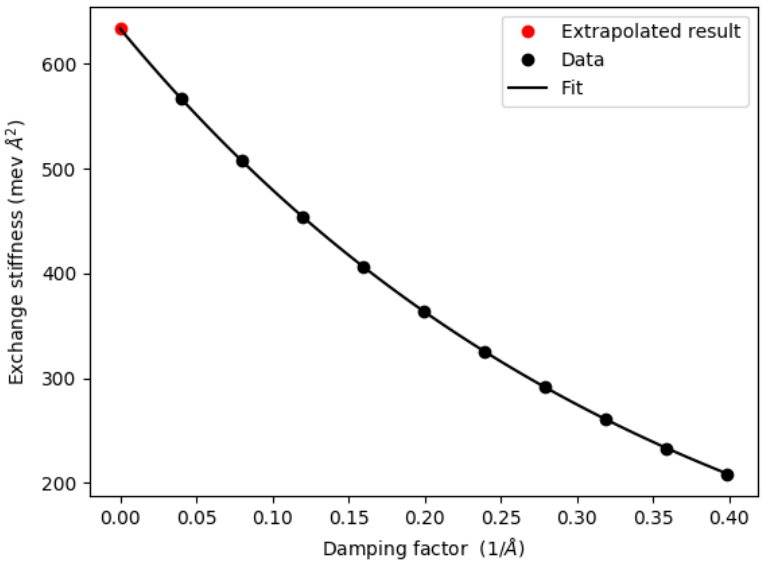
We will now see how to calculate the exchange stiffness for Co. The exchange stiffness given by (3.6) is obtained as

```
exchange_stiffness = heisenberg_exchange.exchangeStiffness()
```

As discussed above and in Ref.[Page 397, 2] it is sometimes necessary to dampen the long-range interaction and subsequently extrapolate to zero dampening following (3.7). The procedure for doing so is

```
extrapolated_exchange_stiffness, exchange_stiffness_vec, damping_factor_range, p_fit \
    = heisenberg_exchange.extrapolatedExchangeStiffness()
```

where the final extrapolated result is given by *extrapolated_exchange_stiffness*. The other returned quantities are a vector of calculated exchange stiffness values obtained for the damping factors given by *damping_factor_range*. The *damping_factor_range* is by default determined automatically as ranging from $0.1/d_0$ to $1/d_0$, where d_0 is the nearest neighbour distance. Finally the polynomial fit is also returned. Together, these allow for a visual inspection of the fitting and extrapolation as done in the script `exchange_stiffness.py` and shown in the figure below.



Download and run the `exchange_stiffness.py`. The output of the script is

```
-------------------------------------------------
Exchange stiffness:
-------------------------------------------------
A_ex                          :  633.8 meV*Ang**2
A_ex (extrapolated)           :  633.7 meV*Ang**2
-------------------------------------------------
```

It is evident that both the direct evaluation using (3.6) and the extrapolated results are in close agreement. While this is often the case, the situation for Fe is a bit more complicated.

Exchange stiffness for Fe

We will now study the Heisenberg exchange of Fe in the BCC phase. Find **Iron (alpha)** in the crystal database in the **Builder** and repeat the steps above for calculating the J_{ij} **HeisenbergExchange**. When modifying the **HeisenbergExchange** object we need to include a longer interaction range:

- Open the **HeisenbergExchange** widget and change exchange the interaction range to $n_A = 13$, $n_B = 13$, $n_C = 13$.
- The other settings used for Co can be applied for Fe as well.
- Finally change the name of the **Results file** to `Fe-BCC.hdf5` and run the calculation.

When the calculation is done the exchange stiffness can be analyzed with the script `exchange_stiffness_Fe.py`. Download and run the script to get the output

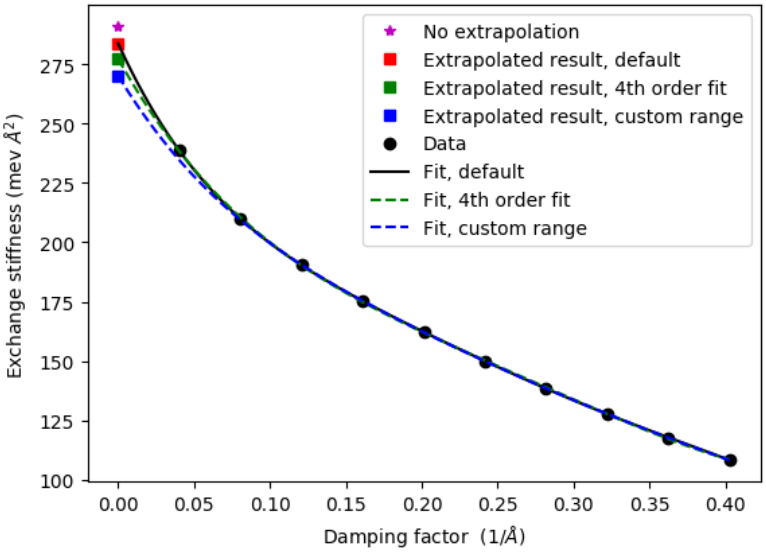
```
-------------------------------------------------------
Exchange stiffness:
-------------------------------------------------------
A_ex                                :  290.7 meV*Ang**2
A_ex (extrapolated, default)        :  283.7 meV*Ang**2
A_ex (extrapolated, 4th order fit   :  277.2 meV*Ang**2
```

(continues on next page)

(continued from previous page)

```
A_ex (extrapolated, custom range)  :  270.2 meV*Ang**2
-------------------------------------------------------
```

and the figure



The script shows how to customize the extrapolation of the exchange stiffness by either (i) changing the polynomial order from the default value of 5 to 4, or (ii) by customizing the damping factor range. All fits look perfectly reasonable, but the extrapolated exchange stiffness constants differ somewhat. This illustrates that in the case of Fe it is difficult to define an exactly calculated exchange stiffness. More discussion on this topic can be found in Ref.[Page 397, 2].

Summary of results

The tables below summarizes the Curie temperature and exchange stiffness results and compares with calculated and experimental values from Ref. [1] =[Page 397, 2]. The values of extrapolated exchange stiffness are obtained using the default settings.

Curie temperatures (K)	QATK	Calc. [1]	Exp. [1]
Co, MFA	1648	1645	1388-1398
Co, RPA	1301	1311	1388-1398
Fe, MFA	1353	1414	1044
Fe, RPA	894	950	1044

Exchange stiffness (meV Å^2)	QATK	Calc. [1]	Exp. [1]
Co	634	663	580
Fe	284	250	280

It is evident that the Curie temperature results from QuantumATK matches well with the calculated results

from[Page 397, 2]. It is also evident that the MFA overestimates the Curie temperature quite significantly compared to experiments whereas the RPA results (extrapolated to infinite q-point sampling) are significantly closer to the experimental values.

The exchange stiffness values are also in good agree with both reference calculations and experimental values.

In summary, this tutorial shows how to calculate the Heisenberg exchange coupling parameters of iron and cobalt. It is demonstrated how to calculate the Curie temperature and exchange stiffness and the obtained values are compare favorably with theoretical- and experimental results from the literature.

COSMICS project

The **HeisenbergExchange** analysis object has been developed within the project COSMICS founded by the European Unions Horizon 2020 research and innovation program under Grant Agreement No. 766726. Details about the object can be found here *HeisenbergExchange*.

More information about the COSMICS project can be found here: http://cosmics-h2020.eu/

References

3.1.14 Building an model of an epoxy thermoset material

Version: S-2021.06

In this tutorial you will learn how to build a model of a thermoset material created by cross-linking different molecules using the example of an epoxy-amine system. You will then learn how to analyze the cross-linking reaction to estimate the gel point of the material. Understanding the structure and properties of different thermoset materials is critical in choosing the correct material for a wide range of applications. It can also help in designing new materials with certain desired properties. This tutorial demonstrates how you can use the QuantumATK tools to simulate cross-linking reactions between different molecules. This methodology can be applied to a wide range of materials and chemical reactions.

To generate a model of the thermoset material, a combination of classical molecular dynamics and a cross-linking algorithm is used to create a trajectory of how the material may be formed through reaction. This results in giving a series of final structures at high degrees of cross-linking. The properties of these structures, such as glass transition temperature or Youngs modulus can then be further investigated using molecular dynamics. This tutorial focuses on building the initial model of the material. A further tutorial, *Analyzing the thermo-mechanical properties of a polymer material* demonstrates how to estimate the thermo-mechanical properties of the thermoset material using the model built in this tutorial.

Contents

•*Building an model of an epoxy thermoset material*

–*Theory*

∗*Epoxy-Amine Materials*

∗*Thermoset Building Algorithm*

–*Building the thermoset model*

∗*Building the epoxy-amine configuration*

∗*Building the Cross-linked Thermoset Material*

–*Analyzing the Thermoset Reaction*

Conclusions

Theory

Epoxy-Amine Materials

Polymer materials can be generally categorized as either thermoset or thermoplast materials. Thermoplast materials typically melt on heating, and are usually constructed of individual polymer chains that are bound together through non-bonding forces. Plastic materials such as PVC (polyvinyl chloride), PET (polyethylene terepthalate) and polyethylene are all types of thermoplast polymers. In constrast, thermoset materials typically do not melt on heating, and are constructed of a 3D covalently bonded network. This network can be formed through post-synthetic cross-linking of polymer chains or by using reactant monomers that are able to link more than two other monomers. Common thermoset materials include Bakelite, vulcanized rubbers and polyester resins.

Epoxy resins are one industrially important class of thermoset materials. These materials are commonly made of two components. The first is the resin containing epoxy functional groups. The second is a molecule containing functional groups that react with the epoxy to form a covalent network. This second molecule is often termed a hardener, and commonly contains amine functional groups. Primary amines and epoxy functional groups can react in a two-stage process requiring one amine and two epoxy groups. In the first step the amine nitrogen attaches to an epoxy carbon, opening the epoxy ring and transferring a hydrogen to the epoxy oxygen. This forms a secondary amine and alcohol functional groups. This secondary amine is able to react again with another epoxy group using the same mechanism, this time producing a tertiary amine. This reaction scheme is illustrated in the figure below.

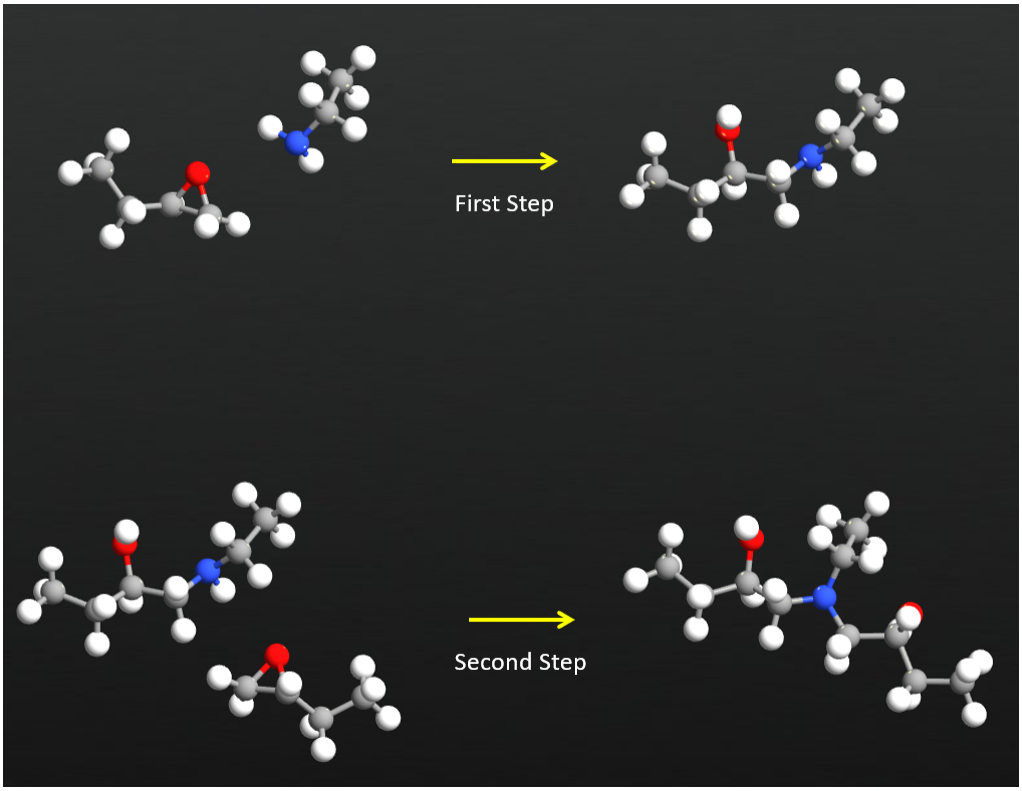


Fig. 3.5: The reaction mechanism of epoxy-amine cross-linking.

Thermoset Building Algorithm

In order to build a representative model of a thermoset material, it is necessary to arrange the molecules in such a way so that the internal bonding energy is minimized and the resulting structure is in a relaxed configuration. Given the various constraints on each molecule this is very difficult to construct directly. A more productive approach is to start with a system of necessary reactants, and then progressively react and relax the structure so as to mimic the actual chemical process used to form the final structure. Variations of this type of building approach have been published and have shown to produce realistic models[1][2][3].

An overview of the building algorithm implemented in QuantumATK is shown in the figure below. The building process starts with an initial configuration of the system before cross-linking. The iterative building process starts with saving a snapshot of the current configuration. Available cross-linking reactions are then performed according to a reaction scheme specified as input to the calculation. As this chemically transforms the structure, new atom types and potential terms are automatically calculated for the new configuration. The system is then relaxed using NVT molecular dynamics. In this step reaction coordinates from the previous cross-linking reaction are gradually relaxed over the whole duration of the dynamics step. This ensures that additional strain is not added to the configuration by the rapid minimization of internal coordinates. Finally, an NPT molecular dynamics step is carried out to further relax the configuration and also correct the density of the overall system. This loop continues until the reaction is completed, producing a trajectory of structures along the path between initial and final cross-linked structures.

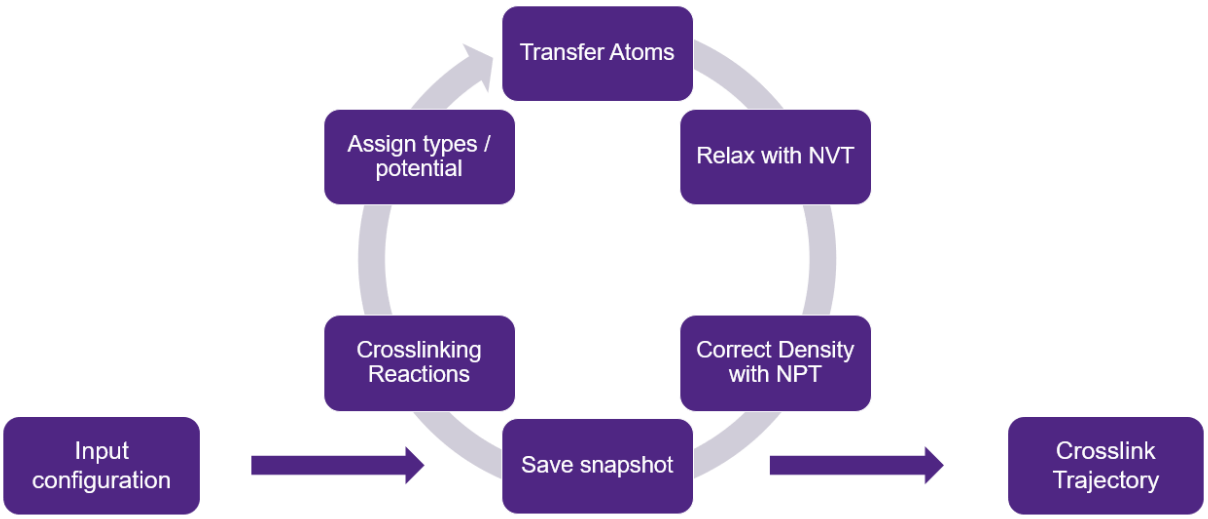


Fig. 3.6: The general algorithm used to build cross-linked structures.

In this simulation the reaction scheme used to transform the configuration can be specified in one of two ways. The simplest way is to use a *ReactionTemplate* class. Here molecular configurations are given that represent the reacting functional groups at different stages of the reaction. These templates are then used to identify reacting groups in the input configuration by scanning the bond graph. Some templates for common cross-linking reactions are also defined in the **CrosslinkBuilder** script block. The second way reacting atoms can be specified is using a *CrosslinkReaction* class. Here the reaction scheme, including bonds formed and broken, are specified using tags which correspond to tags in the configuration. Using this method it is possible to have complete control over which atoms react in each step. In this tutorial we will use one of the pre-defined templates.

[1] V. Varshney, S. S. Patnaik, A. K. Roy, B. L. Farmer: A molecular dynamics study of epoxy-based networks: Cross-linking procedure and prediction of molecular and material properties. Macromolecules 41, 6837 (2008)

[2] C. Li, A. Strachan: Molecular simulations of crosslinking process of thermosetting polymers Polymer 51, 6058 (2010)

[3] S. V. Kallivokas, A. P. Sgouros, D. N. Theodorou: Molecular dynamics simulations of EPON-862/DETDA epoxy networks: structure, topology, elastic constants, and local dynamics Soft Matter 15, 721 (2019)

With the reactions defined, further rules are used to determine which atoms react at each step. Bond and angle limits are employed that specify that no reaction can take place if it creates bonds lengths longer than or bond angles more narrow that particular cutoffs. This ensures that distortions to the overall chemical structure are limited when reacting the configuration. It is also possible to limit the total number of reactions performed at each step. During the NVT relaxation step two different approaches to relaxing bonds are taken. For atoms that are only connected to the structure through a single extended bond, the position of these atoms is directly optimized before the NVT step begins. This is most commonly applied to hydrogens migrating from one atom to another. For newly formed bonds that connect two different substructures, these are relaxed gradually over NVT simulation. This is done by shifting the minimum in the bond potential for those atoms from the initial extended bond length to the final equilibrium bond length.

The final completion point of the reaction is determined by three criteria which, if any one is satisfied the reaction is determined to be complete and the building process is complete. The first criterion is that a specified extent of reaction is achieved. This allows cross-linking calculations to be stopped at any point along the building process. A second criteria is that no reactions are found during a set number of steps. As the building process is carried out it is possible to not find any possible reactions that conform to the bond and angle limits. If this happens after a set number of consecutive cycles, it can be assumed that there are no more viable reactions to be found and the building process is complete. The final criteria is when a specific bond cutoff is reached. During the reaction cycles it is possible to increase the reaction bond cutoff if no reactions are found during the cycle. This allows the building process to prioritize bonding molecules that are relatively close and only later include longer bond formation when necessary.

Building the thermoset model

Building the epoxy-amine configuration

To start building the epoxy-amine system, we first have to build epoxy and amine molecules. In this tutorial the molecules diglycidyl ether of bisphenol F (EPON-862) and diethylenetoluenediamine (DETDA) are used. The structures of each of these molecules are shown in the figure below. Open the **Builder** and select *Add* → *New Configuration*. Start by building a molecule of DETDA. This can be done by switching to the molecular builder. Here atoms in the existing configuration can be replaced by selecting atoms or chemical groups and clicking on hydrogen atoms. New atoms can also be added by clicking and dragging from an existing atom. This will add an atom bonded to the selected atom. Bonds can also be added to the configuration by clicking and dragging between two atoms. For bonds to be created the option *Static bonds* needs to selected. When *Static bonds* is enabled the displayed bonds are saved with the configuration. Once a molecule of DETDA is created, add another configuration to the builder and create a molecule of EPON-862. At this stage it is also convenient to set the appropriate forcefield tags for the molecules. In this simulation the OPLS-AA forcefield is used. To calculate the forcefield types go to the **Forcefield Types** plugin in the **Selection Tools** section. Select the `OPLS-AA` forcefield and then click the *Calculate Types* button. This automatically assigns types based on the elements and the bond graph of the configuration.

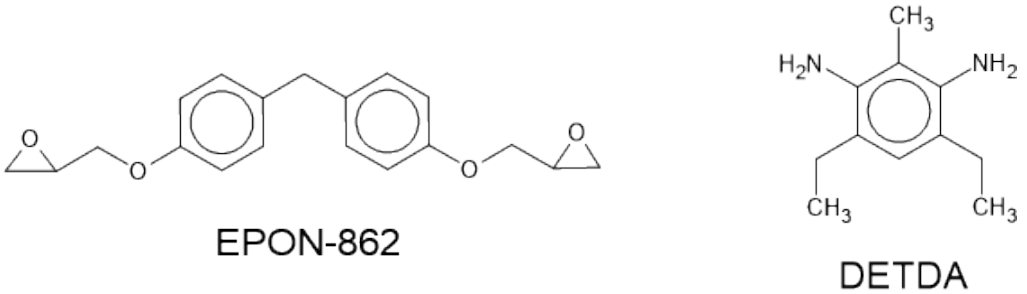


Fig. 3.7: The chemical structures of the initial reactants EPON-862 and DETDA.

Note: The molecular builder is a powerful tool that enables you to easily build a wide range of molecules. For a more

detailed introduction to the many features, see the molecular builder tutorial video here.

To build the initial starting structure for the cross-linking reaction, these molecules need to be packed into a simulation cell. This can be done using the **Packmol** builder plugin. Open the plugin and add both molecules from the **Stash** into the molecules box. As both molecules have two functional groups, and two epoxies are able to react with each amine group, a stoichiometric mixture contains two epoxy molecules for every amine. In the **Packmol** plugin, set the amine count to `128` and the epoxy count to `256`. Also set the *Cell density* to `0.6`gram cm^{-3}. The **Packmol** input dialog is shown below. Clicking *Create* will now add the starting configuration to the **Stash**.

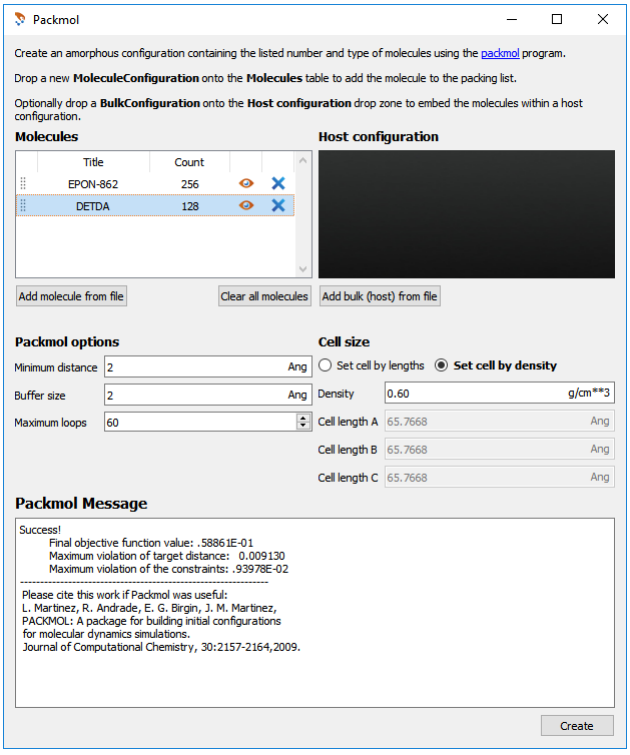


Fig. 3.8: The input options used in the Packmol plugin.

As **Packmol** provides only an approximate molecular packing, it is recommended to improve the initial packing of the molecules by running an additional molecular dynamics calculation. To set up this simulation, send the structure generated by **Packmol** to the **Scripter** using the **Send To** icon. First add a **Forcefield** calculator block to the script. To select the forcefield, open the calculator block and in the buttons on the right select *New –> OPLS-AA*. This adds an OPLS-AA potential builder to the script that will construct an OPLS-AA forcefield for the starting configuration. All of the default options for the potential are suitable, so just click *OK* on the resulting dialogs and return to the script generator.

Next, the molecular dynamics simulation can be defined. Since both the velocities and density of the structure need to be equilibrated, the *Stable NPT* template can be used. This defines three different molecular dynamics methods that equilibrate first the velocities and then the densities in a way that allows for high-energy or approximate starting structures. To add the template, in the *Template* drop-down menu select *Molecular Dynamics –> Forcefield –> Stable NPT Equilibration*. The default for this template equilibrates the structure to a temperature of 300K. As the cross-linking is being done at 480K, the configuration should be equilibrated to this higher temperature. Open the last molecular dynamics block and set the *Reservoir Temperature* and *Final Temperature* to `480`K.

The final MolecularDynamics dialog is shown above. With the OPLS-AA forcefield and the molecular dynamics defined the script is now ready to run. Send the script to the **Job Manager** and run by using the icon. This will

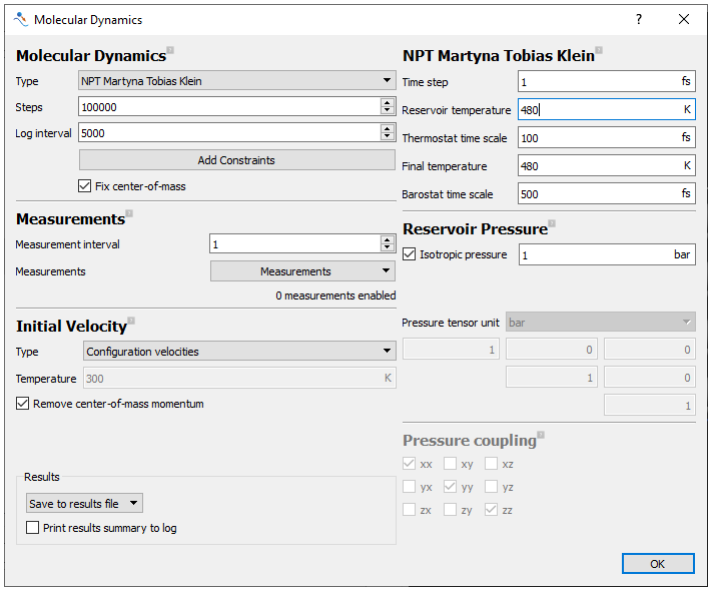


Fig. 3.9: The input options used in the initial molecular dynamics equilibration.

produce three molecular dynamics trajectories equilibrating the initial structure. The final image in the last trajectory will next be used as a starting configuration for the cross-linking simulation.

Building the Cross-linked Thermoset Material

Start by dragging the final image of the previous trajectory onto the **Scripter** . This opens the configuration in the **Scripter** where the script for the cross-linking simulation can be created. Set the output file to `Epoxy_Amine_Thermoset.hdf5`. Next, add a **CrosslinkBuilder** block from the **Optimization** group. This is the block that carries out the cross-linking simulation. The forcefield to be used is defined as part of this block, so a calculator does not need to be added first.

Open the **CrosslinkBuilder** block to set the parameters for the reaction. Start by choosing a reaction scheme. This is done by pushing the *Select start group* button. This brings up a database of the available reaction templates. The reaction templates define both the atoms in the configuration that are able to react and the way in which the bonding changes during each reaction step. An animation of each reaction scheme is shown in the window below the list of reactions. In this system an epoxy is reacting with an aromatic amine, so select the *Epoxy Aromatic Amine* reaction and select *Add*. This adds this reaction template to the script.

Next the parameters for the reaction need to be set. In this dialog:

- Select the *Maximum number of reactions* and set the value to `20`. This limits the number of reactions per step to 20. This can prevent high-energy configurations freezing in during the initial steps of the reaction.
- Set *Temperature* to `480`K. This is the temperature used in the MD relaxation steps.
- Set *Bond search radii increment* to `0.25`Å.
- Set *Maximum bond search radii* to `10`Å. These two options set how reactions are searched for. After each cycle, if no possible reactions are found, the bond search radii is increased by the set increment. When the bond search radii reaches the maximum value, the simulation is completed. In this way the bond search radii is used as a termination criteria. The options *Finish after cycles without reaction* and *Reaction percentage* can also be used as termination criteria for the simulation.

The final dialog box is shown in the figure below. The calculation is now ready to be run. Send the script to the **Job Manager** and run by using the **Send To** icon . The script produces a Trajectory containing a snapshot of the configuration after each NPT equilibration following the reaction. This calculation may take several hours, as it typically goes through around 100 reaction cycles before completion.

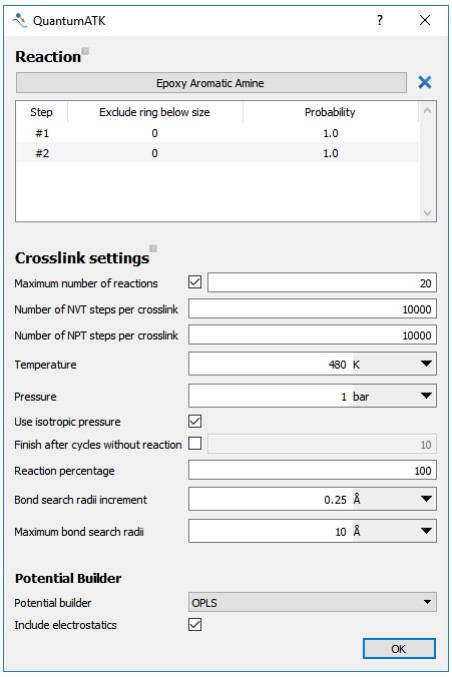


Fig. 3.10: The input options used in the CrosslinkBuilder.

Analyzing the Thermoset Reaction

Once the calculation has completed the cross-linking reaction can be examined. In this section the precise configuration produced by the simulation depends on the initial starting configuration and assigned molecular dynamics velocities, which are randomized. Each calculation will therefore produce similar, but non-identical final configurations. During the calculation information about the number of reactions performed, the density of the material and the stage of the simulation is written to the calculation log. An example of the expected output is shown below.:

```
+-------------------------------------------------------------------------------+
| CROSSLINKING CYCLE 2                                                          |
+-------------------------------------------------------------------------------+
| Performing cross-linking                                                      |
|    Max search radii is 5.0 Ang                                                |
|    Found 161 Reactions                                                        |
|    Performed 20 Reactions                                                     |
|    Cross-linking 7.8% Complete                                                |
| Relaxing Transfer Atoms                                                       |
| Running NVT Step                                                              |
| Running NPT Step                                                              |
| Energy of structure is 32496.5361 kcal_mol                                    |
| Density of structure is 0.9541 g/cm**3                                        |
+-------------------------------------------------------------------------------+
| CROSSLINKING CYCLE 3                                                          |
```

(continues on next page)

(continued from previous page)

```
+-------------------------------------------------------------------------------+
| Performing cross-linking                                                      |
|    Max search radii is 5.0 Ang                                                |
|    Found 139 Reactions                                                        |
|    Performed 20 Reactions                                                     |
|    Cross-linking 11.7% Complete                                               |
| Relaxing Transfer Atoms                                                       |
| Running NVT Step                                                              |
| Running NPT Step                                                              |
| Energy of structure is 30673.3158 kcal_mol                                    |
| Density of structure is 0.9614 g/cm**3                                        |
+-------------------------------------------------------------------------------+
```

Information about the progress of the reaction is also written to the Trajectory. Select the trajectory on the Lab Floor and open it using the **Movie Tool**. Right-click on the configuration to bring up the context menu to display bonds and to show the structure unwrapped. By playing the movie you can see the progress of the reaction. In the graphs on the left you can also see the density, relative energy, bond search radius and percentage of reaction completed for each frame. In particular, the change in density enables the estimation of shrinkage during the cross-linking process. The example calculation is shown in the movie tool below.

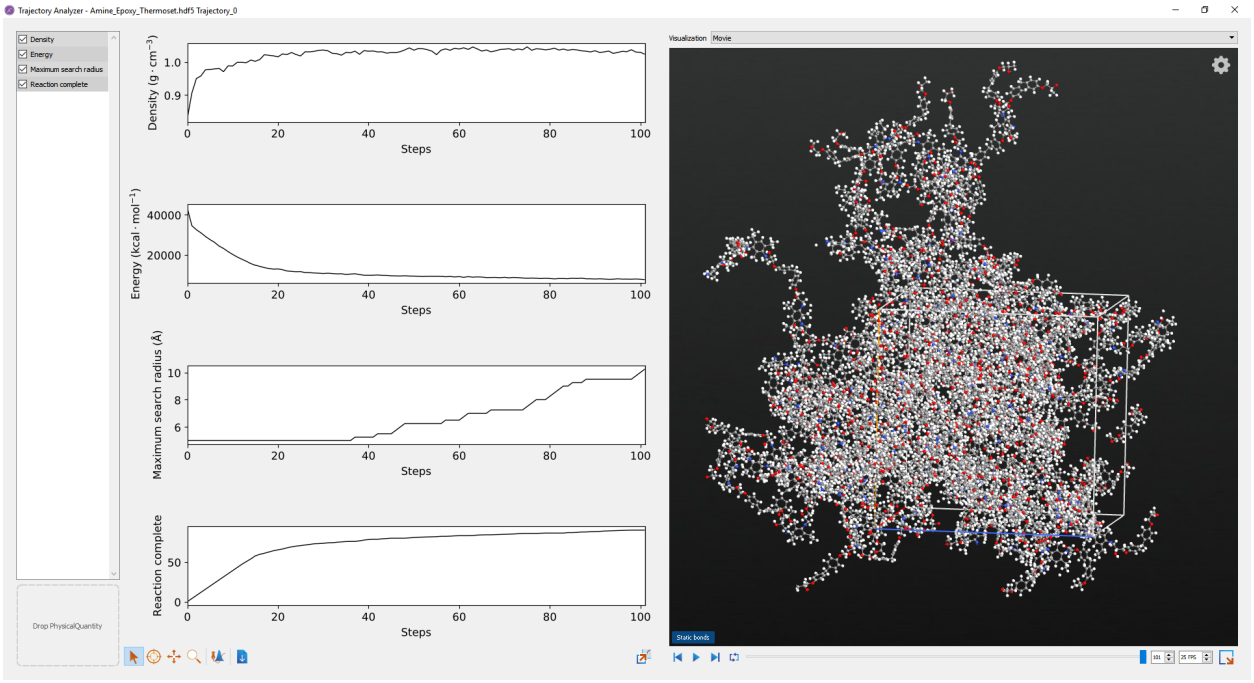


Fig. 3.11: An example epoxy-amine cross-linking calculation.

It is also possible to see the formation of larger molecular structures when cross-linking by viewing the trajectory in the **Viewer** . Select the trajectory in the Lab Floor and drag it over to the **Viewer**. Here you can color atoms in the structure based on the molecule that they belong to. Molecules are determined by the bond graph of the configuration. In the viewer plugins open the **Color Atoms** plugin and set *Color by* to `Molecules`. Also select *Maximize color difference*. This calculates the color difference independently for each image so that it is easier to see each molecule in the image. Select apply and then run the trajectory. As the trajectory is animated the atoms change color depending on the molecule that they are part of.

Note: In the trajectory there may be some unreacted molecules remaining at the end of the simulation. This occurs due to the relatively short time used between cross-linking steps. To increase the amount of cross-linking, a larger relaxation time between cross-linking steps can be used. This increases the diffusion of reactive groups between cross-linking steps brings more groups within bonding range. In this system a relaxation time of 50000 steps in both the NVT and NPT cycles leads to the formation of one network structure with no remaining independent molecules.

Using the cross-linking trajectory it is also possible to estimate the gel point of the material. One method of characterizing this is by estimating the point at which the central sub-structure grows through the periodic simulation boundaries in each direction. The change in periodicity in the sub-structure typically leads to a change in its rheology. To calculate the periodicity of the substructure in each frame of the reaction trajectory, download the provided script `Find_Gel_Point.py`. This takes the file name of the trajectory as an input argument, and plots the resulting periodicity and number of molecules in each frame. A typical plot from the example system is shown in the figure below. This shows that the system starts to become periodic at around 64% conversion. This is close to the theoretical gel point at 58%[4].

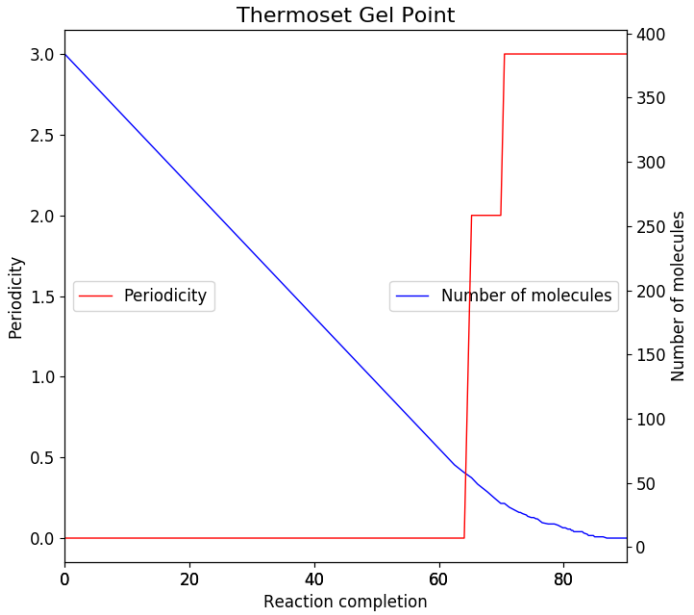


Fig. 3.12: Estimating the gel point by calculating the periodicity of the molecular sub-structure.

Another means of estimating the gel point is to examine the changing molecular mass of the system during the reaction. As the reaction starts, individual molecules react together forming larger molecules. As the gel point is reached these larger structures join together to form one dominant sub-structures. This can be observed by calculating the average reduced molecular mass of the system. The reduced molecular mass is simply the molecular mass ignoring the largest molecule. During the reaction the average reduced molecular mass typically reaches a maximum near the gel point, as the large individual molecules join the dominant sub-structure[Page 407, 1]. To calculate the reduced molecular mass, download the attached script `Find_Molecular_Mass.py`. As before, this script takes the file name of the trajectory as an argument, and plots the average total and reduced molecular mass during the reaction. A typical plot for the example calculation is shown in the figure below. From this plot it can be seen that the maximum in the reduced mass occurs at 61%, close to the previous estimates of the gel point.

[4] E. Hädicke, H. Stutz: Comparison of the structure of step-growth networks obtained by Monte Carlo simulation and branching theory J. Appl. Polym. Sci. 85, 929 (2002)

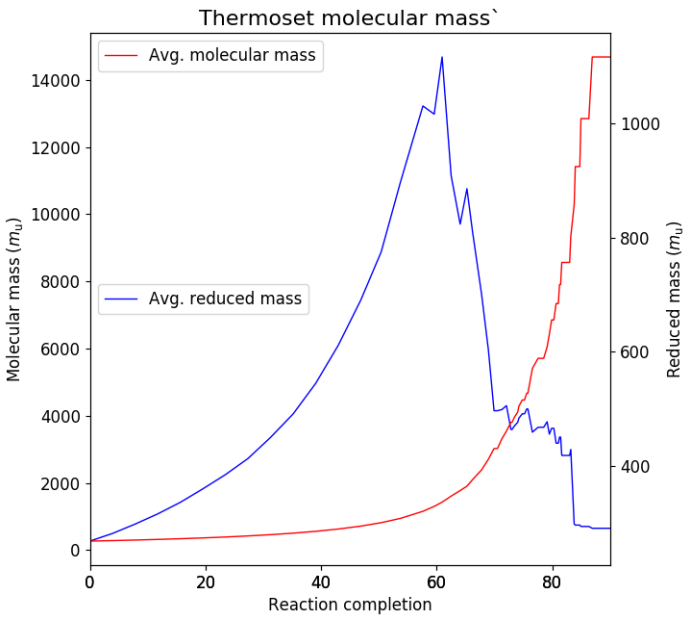


Fig. 3.13: Estimating the gel point by calculating the average reduced molecular mass.

Conclusions

In this tutorial the process for building a typical epoxy-amine thermoset material was outlined. Using the **CrosslinkBuilder** tool, a high degree of cross-linking was able to be achieved, although this can be increased by using a longer time step between cross-linking events. Analyzing the reaction process showed that the building process also exhibits a realistic gel point, even at shorter cross-linking intervals. Overall this tutorial shows how reliable models of complex thermoset materials can be easily created in QuantumATK.

References

3.1.15 Analyzing the thermo-mechanical properties of a polymer material

Version: S-2021.06

In this tutorial you will learn how to estimate the thermo-mechanical properties of a polymer system, most notably the glass transition temperature, Youngs modulus and Poisson ratio. These properties are critical performance parameters in many applications. They can determine, for instance, the operating temperature or the maximum load of a given material. As these properties are determined by the atomic structure of the polymer system they can be estimated using molecular simulation.

The material used as an example will be a epoxy-amine thermoset material consisting of diglycidyl ether of bisphenol F (EPON-862) and diethylenetoluenediamine (DETDA). The tutorial *Building an model of an epoxy thermoset material* demonstrates the steps required to build a model of this material. This tutorial continues from that tutorial by demonstrating how the properties of the model can be simulated. The procedure outlined here is quite general, and can be applied to different types of polymeric and non-polymeric materials. Most notably, the same simulation procedure can be used on models of linear thermoplast polymers, such as poly(methyl metacrylate) and polyvinyl chloride, built using the **Polymer Builder** tools. It is not necessary that a polymer system be cross-linked for these methodology to give good results.

Contents

•*Analyzing the thermo-mechanical properties of a polymer material*

–*Glass Transition Temperature*

–*Youngs Modulus and Poisson Ratio*

–*Conclusions*

Glass Transition Temperature

One important property of many polymer materials is the glass transition temperature. This is the temperature at which the material changes from a rigid, glassy state to a more flexible rubbery state. This transition from glassy to rubbery states in linear polymers usually corresponds to more torsional movement of the polymer chains as well as an increase in free volume. The glass transition temperature is typically estimated using a simulated annealing process, where the material is cooled during an NPT molecular dynamics simulation. Crossing the glass transition temperature also involves a change in the rate of thermal expansion, which can be identified by a change in the gradient of the density vs. temperature plot.

To calculate the glass transition temperature, an annealing simulation must first be performed. To set up this simulation, start by downloading the thermoset polymer model. Send this model to the **Scripter** by first opening it in the **Builder** and then using the icon to send the configuration to the **Scripter**. Alternatively you can save the HDF5 file in your project directory, open the file on the Lab Floor, and then drag the *Bulk Configuration* directly onto the **Scripter**. Start by adding a **ForceField** calculator. Open the calculator block and using the buttons on the right select *New* –> *OPLS-AA* to add an OPLS-AA forcefield. The next step is to equilibrate the system at an appropriate starting temperature[1]. This can be achieved by adding a **PolymerEquilibration** block from **Study Objects**. The most important parameter to set here is the final temperature for equilibration. This temperature should be high enough above the glass transition temperature that it enables getting an estimate of the thermal expansion rate above the glass transition temperature. Open the **PolymerEquilibration** block and set the *Final Temperature* to `600`K. The remaining default parameters are all appropriate. Once the system is properly equilibrated the annealing calculation can be performed. To do this, add a **MolecularDynamics** block from the **Optimization** group. Open this block and make the following changes:

•Select the `NPT Martya Tobias Klein` ensemble.

•Set the *Steps* to `1750000` and *Log interval* to `1000`.

•In the *Measurements* section, add measurements for temperature, volume and density.

•Set the initial velocity to `Configuration velocities`.

•Set *Reservoir temperature* to `600`K and *Final temperature* to `250`K. Note that this sets a temperature sweep of 350K over 1.75ns at a rate of 0.2K/ps.

The remaining default parameters are suitable. The final molecular dynamics dialog is shown in the figure below.

The script is now ready to be run. Send it to the **Job Manager** and run the script by using the **Send To** icon . This will produce the annealing trajectory that can be used to estimate the glass transition temperature. Note that as the script is performing over 3 million molecular dynamics steps, running it will take several hours.

Once the calculation is completed, open the trajectory from the Lab Floor using the **Glass Transition Temperature Analyzer**. This tool provides a plot of the density of the system with respect to temperature. It also estimates the glass

[1] L. J. Abbott, K. E. Hart, C. M. Colina: Polymatic: a generalized simulated polymerization algorithm for amorphous polymers Theor. Chem. Acc. 132, 1334 (2013)

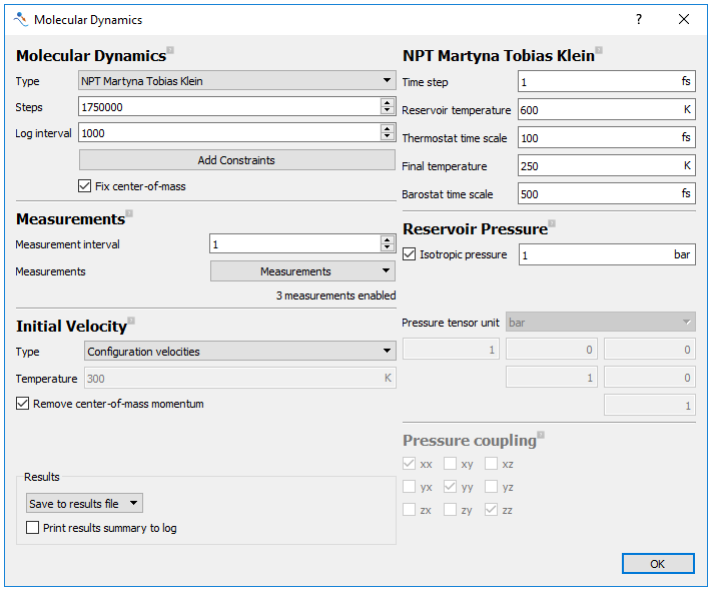


Fig. 3.14: Settings for the annealing calculation to estimate Tg.

transition temperature through fitting lines to the low and high temperature data, and then calculating the temperature at which the lines intersect. Click the *Estimate fit limits* button to automatically estimate the glass transition temperature. This adds limits indicating the high and low temperature fitting regions, as well as an estimate of the glass transition temperature. The limits can be manually adjusted to improve the quality of the fit. The glass transition temperature analyzer using the example calculation is shown in the figure below. This shows a glass transition temperature of 417K. This is in agreement with the range of experimental estimates of the glass transition temperature of this material[23].

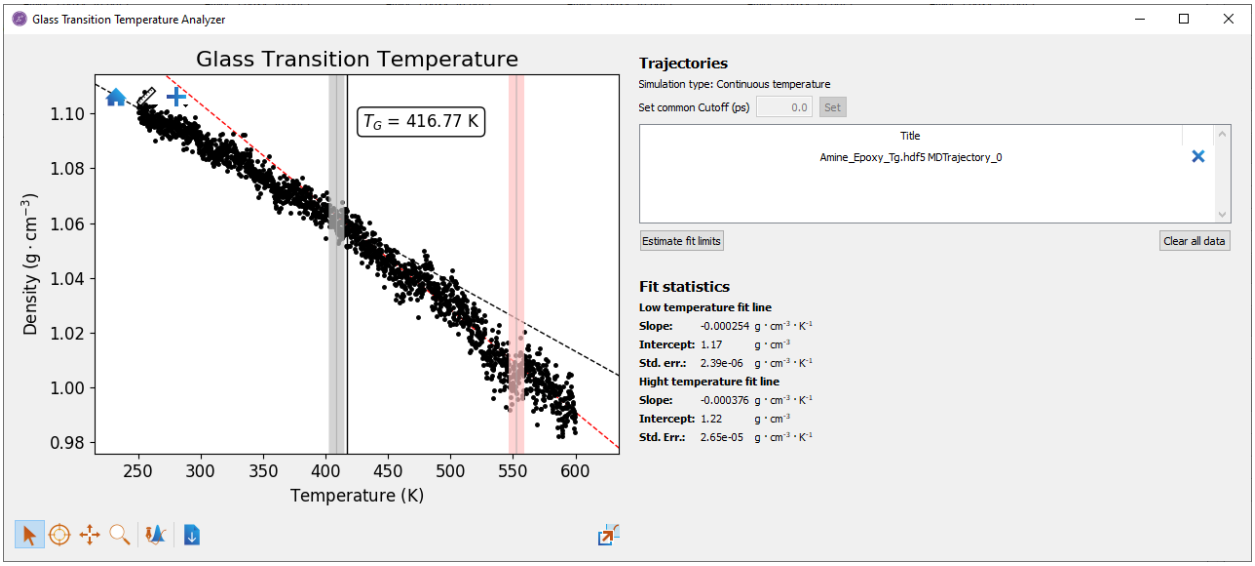


Fig. 3.15: Glass transition temperature analyzer showing predicted Tg.

[2] K. Tao, S. Yang, J. C. Grunlan, Y. S. Kim, B. Dang, Y. Deng, R. L. Thomas, B. L. Wilson, X. Wei: Effects of carbon nanotube fillers on the curing processes of epoxy resin-based composites J. Appl. Polym. Sci. 102, 5248 (2006)

[3] L. Sun, G. L. Warren, J. Y. OReilly, W. N. Everett, S. M. Lee, D. Davis, D. Lagoudas, H. J. Sue: Mechanical properties of surface-functionalized SWCNT/epoxy composites Carbon 46, 320 (2008)

Youngs Modulus and Poisson Ratio

Another important property of many epoxy systems is the Youngs Modulus. This is a measure of the amount of stress required to produce a given strain in the elastic deformation region. The Youngs modulus can be directly simulated using molecular dynamics. Here the strain is added by by progressively extending one of the simulation cell lengths. The resulting stress in this direction is then recorded. The gradient of the stress vs. strain curve then gives the Youngs modulus.

To carry out a simulation of the elastic properties of the material, a starting structure first must be selected. In the previous step a simulated annealing calculation was carried out, resulting in structures close to equilibrium at a range of temperatures. One option for selecting a starting structure is to take configurations from this annealing trajectory at the temperatures of interest. As these configurations are close to equilibrium, they may only need a relatively short equilibration calculation of around 100ps at the temperature of interest before starting the stress-strain simulation. This is possible because in the annealing calculation the starting configuration was first well-equilibrated at the starting temperature. As the Youngs modulus can change significantly, especially when crossing the glass transition temperature, this can be an efficient way to calculate the Youngs modulus at a range of temperatures. If the glass transition temperature has not been previously calculated, it is then still advisable to ensure that the starting configuration is well equilibrated at the temperature of interest using a reliable equilibration protocol. This is the approach that will be used in the following calculations, where the Youngs modulus of the example material will be estimated at room temperature.

To start setting up the calculation of Youngs modulus, open the final structure from the Crosslink Builder in the **Scripter** . As with the glass transition temperature, start by adding a **Forcefield** Calculator and then a **PolymerEquilibration** block from **Study Objects**. Open the **PolymerEquilibration** block and check the default parameters. As the system is being equilibrated to room temperature, all of the default parameters are appropriate. The required molecular dynamics calculations for simulating the stress-strain relationship can now be added. As the epoxy-main material is isotropic, the Youngs modulus should be the same in each dimension. The Youngs modulus can therefore be calculated in each direction and the average taken to provide a final value. Add 3 **MolecularDynamics** blocks from **Optimization** to the script. Each block will be used to simulate the Youngs modulus in a specific direction. Open the first **MolecularDynamics** block and set the following parameters.

•Set the *Type* to `Stress Strain`.

•Set *Steps* to 100000 and *Log Interval* to 1000.

•In the *Measurements* section, add all of the diagonal (xx, yy and zz) strains.

•In the Initial Velocity section, set *Type* to `Configuration velocities`.

•In the Stress Strain MD section, select the `xx` as the *Strain Direction*. Also set the the final strain to 20%.

Note: In this simulation the cell is being strained by 20%. This is much larger than strictly required for a Youngs modulus calculation. Typically a total strain of around 2% is required is required for an estimate of the Youngs modulus. Here a larger strain is being used to illustrate the yield point on the stress-strain plot. As the calculated Youngs modulus can depend on the strain rate, it is generally better to use a lower strain rate.

Repeat the same steps for the remaining two molecular dynamics blocks, except in these set the *Strain direction* to `yy` and `zz` respectively. For setting the strain in the X direction, the molecular dynamics dialog should look like the one below.

Before running the script, a few simple manual edits need to be performed. Open the script by sending it to the **Editor** using the **Send To** icon . By default calculations are chained together so that the resulting configuration from one calculation is used as the input for the next. In this case however, that would result in the material strained in one direction would be used as the input for straining in the next direction. The script needs to be changed so each

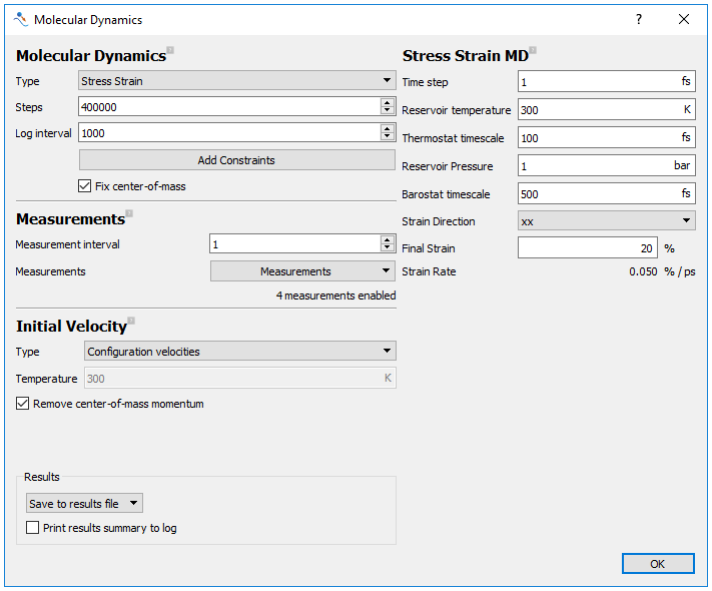


Fig. 3.16: Settings for the annealing calculation to estimate Tg.

stress-strain simulation starts from the same equilibrated structure. The links connecting each molecular dynamics calculation should be removed. These links occur in the line:

```
bulk_configuration = md_trajectory.lastImage()
```

Here the starting bulk configuration is being over-written with the final bulk configuration from the molecular dynamics simulation. Delete this after each stress-strain molecular dynamics calculation. This will mean that the bulk configuration is not updated, resulting in the same bulk configuration being used as input for each dynamics simulation.

With these changes the script is now ready to run. Send it to the **Job Manager** and run the script by using the **Send To** icon . The final output of the calculation will be three molecular dynamics trajectories with the longitudinal stress and the three diagonal strains recorded in it. As before, since this script performs 2.8ns of molecular dynamics it can take several hours to run.

Once the calculation is finished, open the resulting HDF5 file on the Lab Floor. Open one of the MDTrajectories in the **Movie Tool** and observe the trajectory. You should see the material being extended along the longitudinal direction, while the material shrinks in the transverse directions. In the stress-strain simulation an NPT ensemble is used, allowing the transverse directions to respond according to the applied strain. To calculate the Youngs modulus a plot of the stress with respect to the strain should be created. In the Lab Floor, open the **Data Plot** tool. From the first trajectory drag the strain in the xx direction onto the X axis, and the xx stress only the Y axis. Click *Create line* to generate the plot.

In general, stress can fluctuate widely during an NPT molecular dynamics calculation compared to the stress resulting from the applied strain. This can obscure some of the important details of the stress-strain plot. To make it clearer, open the **Plot Editor** by clicking on the *Edit* icon at the bottom-left of the plot. This window shows the different elements in the plot and enables editing their properties. Select the curve and set the *Opacity* in the **Plot Editor** to `0.1`. Right-click on the curve and select *Add analysis* → *Rolling average*. This will help to average out some of the fluctuation in the stress. Select the rolling average, move the ranges to the edges of the plot and in the **Plot Editor** set *Window* to `1000`. The rolling average should be linear at low strain, and then deviate from linearity at higher strain. To find the gradient at low strain right-click on the original plot and select *Add analysis→Polynomial fit*. This adds a line of best fit onto the plot. Use the fitting ranges to move the line over the initial linear region of the stress-strain plot. The gradient and intercept of the line is shown in the **Plot Editor** with the *Fit line* selected. It is also shown by simply hovering the mouse over the line. Below is an example plot from the model system strained in the X direction.

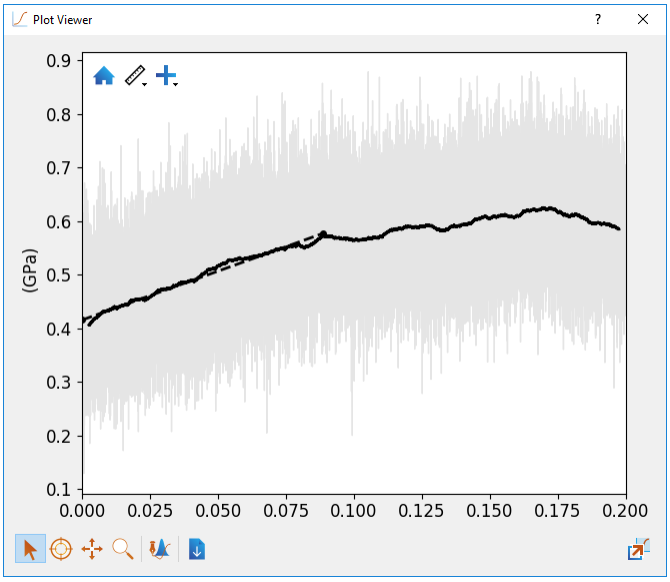


Fig. 3.17: Plot of stress with respect to strain in the X direction in the epoxy-amine thermoset material.

A similar procedure can be used to estimate the Poisson ratio. This is the ratio of the strain in transverse direction to the applied strain. In each simulation the strain in each direction was saved in the MDTrajectoy. These can be plotted in the same way as the stress and strain to produce plots that can be fitted with lines of best fit. The gradient of these lines is then the Poisson ratio.

In the example calculation, averaging over the 3 strain directions and 6 Poisson ratios gives an average Youngs modulus of 2.0GPa and Poisson ratio of 0.35. The Youngs modulus in this case is lower than measured experimental values[Page 416, 2Page 416, 3]. This is likely due to the incomplete cross-linking reaction in the initial building of the epoxy-amine material. The starting configuration used in this tutorial was obtained from the *Building an model of an epoxy thermoset material* tutorial, which used a relatively low number of steps between cross-linking reactions. Increasing the number of NVT and NPT steps per cross-linking step increases the time between cross-linking events and also the distance that reactive atoms can move towards each other. In other simulations where the number of steps was set to 50000 for both NVT and NPT it was possible to link all reactant molecules into one large periodic structure. Increasing the degree of cross-linking results in a corresponding increase in the Youngs modulus. In calculations using a model built with more steps between cross-linking reactions, the same protocol for calculating the Youngs modulus resulted in values between 2.45GPa and 3.11GPa in the range of 90-95% completed cross-linking. This is in line with the experimentally measured Youngs modulus. The Poisson ratio however appears less sensitive to the degree of cross-linking. Here the calculated value of 0.35 is in agreement with the experimental value[4].

Conclusions

In this tutorial the glass transition temperature, Youngs modulus and Poisson ratio of a thermoset polymer material were estimated using molecular dynamics simulation. Here these showed good agreement with the experimental glass transition temperature and Poisson ratio, while the Youngs modulus was somewhat less than the experimental value. This was due to the incomplete cross-linking in the original configuration due to the short time between cross-linking steps. Overall this tutorial shows how reliable models of complex thermoset materials can be easily created and their properties accurately simulated in QuantumATK. These methods can be successfully applied across a range of materials, including thermoset and thermoplast polymers.

[4] J. L. Tack: Thermodynamic and mechanical properties of EPON 862 with curing agent DETDA by molecular simulation Masters thesis, Texas A&M University (2006)

References

3.1.16 Generating A Moment Tensor Potential for HfO_2 Using Active Learning

Version: T-2022.03

Background

Machine learned interatomic potentials (MLIPs) have become popular in recent years since they can provide ab initio level of accuracy in energies, forces and stresses needed for MD simulations, geometry optimizations and single point calculations at modest computational effort. Moment tensor potentials (MTPs) are a class of MLIPs that have been identified to provide high performance for a given accuracy when compared to other MLIP models in the literature[1]. On training an MTP for a system of interest, our goal is to establish a structure-property relationship - here, structure is the atomistic description of the system (3N Cartesian coordinates) and property is the systems potential energy surface (PES - 3N dimensional function).

Within MTP framework, the total energy of a configuration is computed as the sum of energy contributions from atomic neighborhoods. Atomic neighborhoods are defined as the representation of the immediate chemical environment of each atom in the system within a cutoff radius. Since Cartesian coordinates are not invariant to translations, rotations and permutations of atoms in the configuration, they can not be used as such as inputs in MTP training. Moment tensors include the radial, angular, and many-body distribution of atoms within a cutoff sphere. Their contractions into scalar basis functions provide an invariant representation of atomic environments (structure) needed as input for MTP training[2]. An electronic structure method, such as density functional theory (DFT), that describes the system of interest accurately is chosen as the reference method to compute the properties of interest. A reference data set includes the reference configurations representing the system of interest along with their energies (E), atomic forces (F) and stress tensors (S) computed using the chosen reference method.

A MTP can be trained by fitting a linear regression model to the reference data set. MTP training on a reference data set can be performed in two ways: batch learning and active learning. In batch learning, an MTP is trained on a precomputed reference data set. In active learning, a preliminary reference data set is improved on-the-fly by actively adding missing configurations to it. This is done by first training a preliminary MTP on the initial reference data and starting an MD simulation using this MTP potential. When geometries that are very different from the initial reference geometries are encountered in the MD, they are added to the reference data set after computing their E, F and S using the reference method as described in[3] and[4].

It is important to understand that MTPs or any other MLIPs are trained to describe a system accurately (very close to the reference method) within the configurational space of the system sampled in the training data set. Unlike many empirical potentials with physical functional forms, MLIPs are not highly transferable. Therefore, an MLIP trained for a bulk material might not describe free surfaces or nanoparticles of that material accurately unless such geometries are explicitly included in the training data set. Moreover, an MLIP trained for room temperature behavior of a system might not work reliably to describe the system at high temperatures unless disordered structures relevant at high temperatures are included in the training dataset. To improve transferability, active learning of the system at its desired state (high temperatures or surface geometries) should be carried out in order to include the missing reference data.

[1] Yunxing Zuo, Chi Chen, Xiangguo Li, Zhi Deng, Yiming Chen, Jörg Behler, Gábor Csányi, AlexanderăV. Shapeev, AidanăP. Thompson, MitchellăA. Wood, and ShyueăPing Ong. Performance and Cost Assessment of Machine Learning Interatomic Potentials. *The Journal of Physical Chemistry A*, 124(4):731–745, 2020. URL: https://doi.org/10.1021/acs.jpca.9b08723, arXiv:https://doi.org/10.1021/acs.jpca.9b08723, doi:10.1021/acs.jpca.9b08723.

[2] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

[3] EvgenyăV. Podryabinkin and AlexanderăV. Shapeev. Active learning of linearly parametrized interatomic potentials. *Comput. Mater. Sci.*, 140:171–180, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0927025617304536, doi:https://doi.org/10.1016/j.commatsci.2017.08.031.

[4] Konstantin Gubaev, EvgenyăV. Podryabinkin, GusăL.W. Hart, and AlexanderăV. Shapeev. Accelerating high-throughput searches for new alloys with active learning of interatomic potentials. *Comput. Mater. Sci.*, 156:148–156, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0927025618306372, doi:https://doi.org/10.1016/j.commatsci.2018.09.031.

In this tutorial we will generate an MTP for bulk HfO_2 that can predict DFT quality E, F and S of both crystalline and amorphous configurations using **NanoLab** GUI. Since DFT is used as reference method, this will take 4 to 6 days depending on the computational resources available.

Tip: If you are only interested in training an MTP to simulate crystal structures, you can skip the active learning part and go from the initial reference geometries directly to the final training in Step 3.

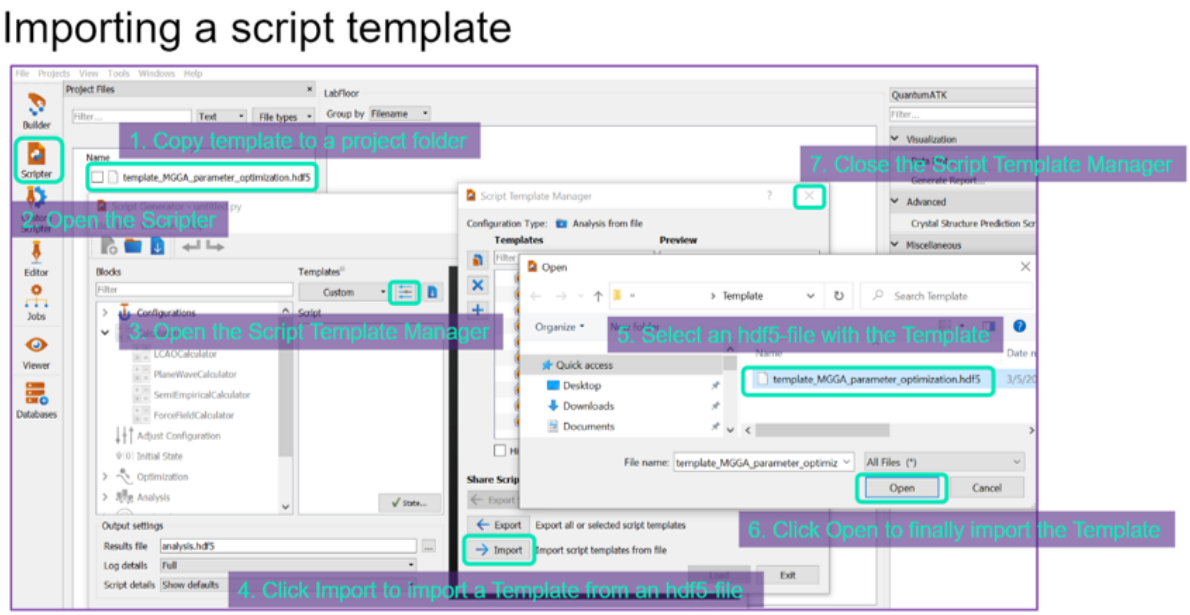
Getting Started

Create a new project folder in **NanoLab** and copy the following files containing different bulk phase configurations of crystalline HfO_2 (`cubic.hdf5`, `monoclinic.hdf5`, `orthorhombic.hdf5`).

Tip: Crystal reference geometries can be conveniently obtained using the open databases such as Materials Project that is accessible via NanoLab

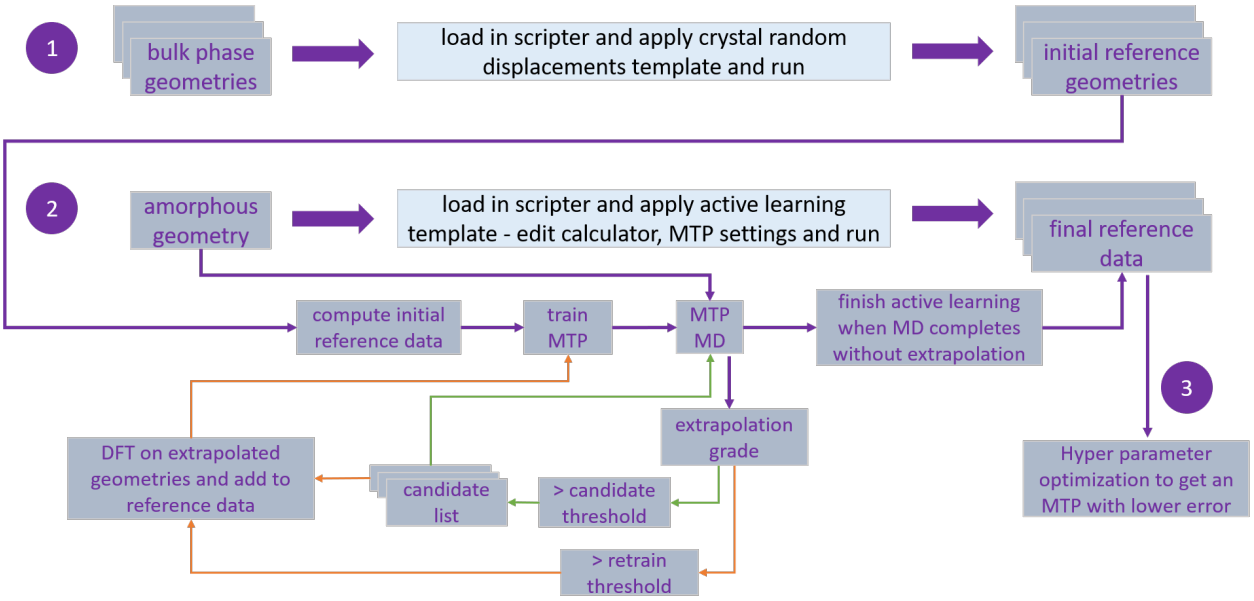
---

As the next step, import the following two templates `random_displacement_template.hdf5` and `active_learning_template.hdf5` to the script generator following the procedure given in the below graphic.

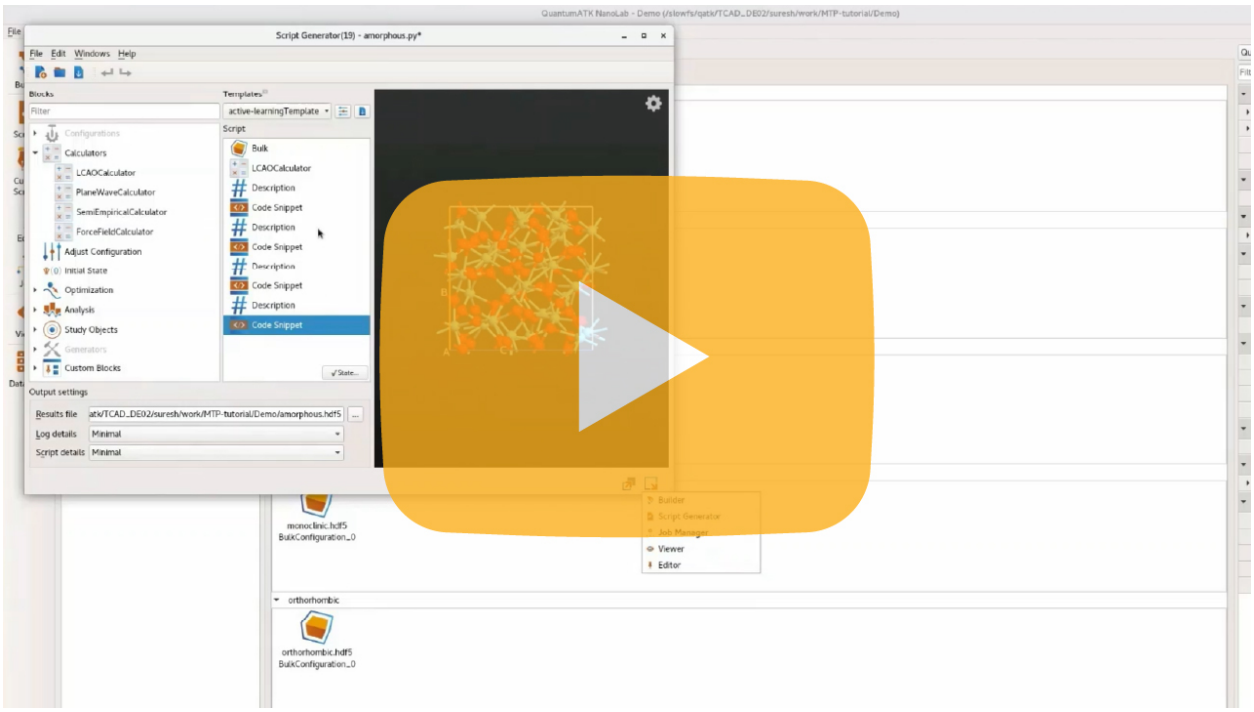


Workflow

An schematic illustration of the workflow is shown below. There are three steps involved which are detailed in the text further below.



A video demonstrating the workflow can be viewed below.



Step 1: Prepare Initial Reference Configurations

Load one of the bulk phase configurations of HfO_2 on to the script generator and apply the *random_displacement_template*. This will add a code snippet for *crystalTrainingRandomDisplacements* in which the user can edit the supercell repetition list, ionic and cell rattle amplitudes to create random configurations based on the geometry. These randomly generated geometries will be stored as a list in the file named *initial_training_sets.hdf5*. Choose large repetitions for small unitcells and *vice-versa*. Save the script and run it locally as it takes only a few seconds.

Repeat the above procedure for the other two bulk phase configurations of HfO_2 . The corresponding python scripts used for this tutorial can be downloaded here (`cubic.py`, `orthorhombic.py` and `monoclinic.py`). Notice that, in the above scripts, the supercell repetition list is different for the different crystal phases depending on the size of the unitcell. The template uses (1x1x1) and (2x2x1) supercells by default. However, for the cubic phase, include (2x2x2) and (3x3x3) instead of the default as in the scripts above. The resulting *initial_training_sets.hdf5* file will contain three lists one for each bulk phase.

Note: The above scripts must be run in sequence and can be run locally (via atkpython in terminal or via JobManager) as it takes only a few seconds.

Warning: If you alter the scripts from above, different repetition list or rattling amplitudes, then the training geometries will be different and ultimately result in a different MTP fitting parameter set. Thus, the users might find that the exact training and testing errors differ a bit from what is given further below.

Step 2: Compute Reference Data and Setup Active Learning

Now that we have a set of initial reference geometries from step 1, we can proceed to compute reference data using DFT and improve the initial training data using active learning molecular dynamics on bulk amorphous configuration of HfO_2 . For this, use the amorphous configuration from this file `amorphous.hdf5`. This initial amorphous configuration was generated using the Packmol Builder plugin and preoptimized with Trinastic_HfOSiTaTi_2013 forcefield. Send this configuration to the script generator and apply active learning template. This loads a calculator and 4 code snippet blocks to the script generator that can be edited.

Calculator: The reference calculator to use for MTP training

For this example, the DFT settings we have chosen include LCAO PseudoDojo medium basis set and PBE exchange-correlation functional. Detailed calculator settings can be found in the script `active-learning.py`.

Note: For accurate MTP training, higher accuracy settings than the ones used in this tutorial may be needed.

Snippet 1: Import reference geometries

In the first snippet, there is code to extract the training data parameters for the reference configurations automatically from *initial_training_sets.hdf5*. The snippet also includes optional code to extract geometries from a *Trajectory* or *MDTrajectory* objects, which can be uncommented if one wants to use configurations from MD trajectories as initial training data.

Snippet 2: Compute reference data

In the second snippet, we use *MomentTensorPotentialTraining* object to compute reference energies, forces and stress using the DFT reference calculator loaded earlier. The user can modify the values for *number_of_processes_per_task*, to set the number of MPI processes used for each reference calculation and for the *log_filename_prefix*. The other settings can be left as such. This snippet will create a file named *reference-data.hdf5* which includes the initial reference data set corresponding to the geometries from step 1 to be used in active learning.

Note: We suggest running this script on a node with several cores as this could take 1 to 2 days using 16 cores.

Snippet 3: Customize MTP training

In the third snippet, the user can set values of key parameters needed for the MTP training. This includes the following:

NonLinearCoefficientsParameters : User can specify if they like to perform optimization of the non linear coefficients that determine the shape of the basis functions for each element pair. If set to *True*, then there are options to perform optimization using energy only, maximum optimization iterations and a random seed to initialize the coefficients. Please note that this is a local optimization in the MTP parameter space, and different initial values may result in different solutions. How to find initial values for a good fit will be discussed below.

constant_terms: No need to edit this part of the snippet as it automatically computes the isolated atomic energies of the elements in the reference data set using the reference calculator, which is not yet automatically computed for active learning.

MomentTensorPotentialFittingParameters: In this part of the snippet, the size of the basis set to be used, cutoff radii for atomic neighborhood description and force cap on configurations can be provided. Note that geometries with a maximum atomic force above the *forces_cap* value will be ignored during training.

Snippet 4: Set up active learning

In this final snippet all settings relevant for active learning will be provided.

ActiveLearningSimulation: In this part, user can provide the threshold values for the extrapolation grades and an interval at which they will be computed in MD[Page 420, 4]. The configuration will be added to a candidate list when the extrapolation grade goes above *candidate_threshold* value and the MD will stop if the extrapolation grade goes over *retrain_threshold*. The *max_forces_check* value sets the force value, which, when encountered during the simulation, will trigger computation of extrapolation grade for the corresponding MD configuration.

MaxwellBoltzmannDistribution: is included to initialize the velocities at a target temperature for MD. In this tutorial example, we chose to run active learning at 3000 K.

NPTMartynaTobiasKlein: is used as the default MD ensemble method which relaxes both ions and volume. This can be changed to other ensemble methods if needed.

The method *runMolecularDynamics* starts an active learning MD simulation of the amorphous HfO_2 geometry. Every time the *retrain_threshold* is reached, this method will be restarted until the MD simulation completes the set number

of time steps. Before each restart, unique geometries from the candidate list will be selected and reference calculations on those geometries will be performed using the reference calculator. The additional reference data is then added to the initial dataset and a new MTP will be generated using the settings in snippet 3. This updated MTP will be used for the restarted MD simulation.

The script that is used for this tutorial is available here `active-learning.py`.

At the end of active learning simulation, a file named *active_learning_candidates.hdf5* is created, that contains the additional reference data.

Note: A common source of problems in MTP active learning is the following error message:

No new candidates found in active learning MD.

This is often accompanied by large values for the extrapolation grade. The reason for this problem is in most cases that the condition number of the training matrix is too large, which causes numerical inaccuracies when inverting this matrix to calculate the extrapolation grade and selecting the candidates via the maxvol algorithm. In this case it usually helps to reduce the number of MTP basis functions to improve the condition number and the numerical stability of the maxvol calculation. Alternatively, one can also try to include more different training configurations in the initial training set.

To repeat active learning simulation with a different MD setting, the training data computed so far should be put-together as the initial training data. In this case, *reference-data.hdf5* consists of the crystalline random displacement data as a *MomentTensorPotentialTraining* object and *active_learning_candidates.hdf5* contains data for unique amorphous geometries found in active learning as a *Trajectory* object. The *initial_training_data* parameter in the *ActiveLearningSimulation* block in the script should be a list of either *Trajectory* objects or *MomentTensorPotentialTraining* objects. We can convert a *MomentTensorPotentialTraining* object in file *reference-data.hdf5* into a *Trajectory* object and prepare the updated initial training data for consecutive active learning simulations as

```
moment_tensor_potential_training=nlread('reference-data.hdf5',
↪MomentTensorPotentialTraining)[-1]
traj_1=TrainingSet(moment_tensor_potential_training, recalculate_training_data=False)
traj_1=traj_1.configurations()

traj_2=nlread('active_learning_candidates.hdf5',Trajectory)[-1]

initial_training_data = [traj_1, traj_2]
```

In a similar way, the *initial_training_data* can be augmented from several active learning simulations.

Step 3: Find an MTP with Lowest Error

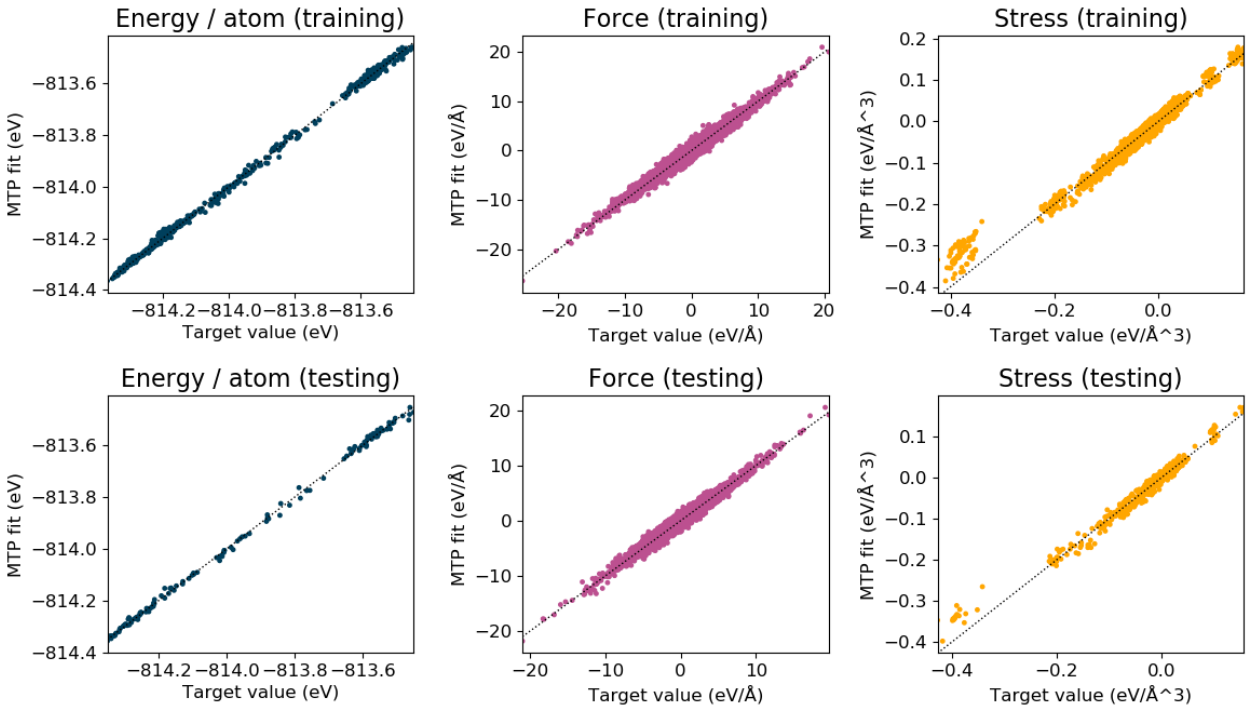
In active learning, the non linear coefficients are randomly initialized. So, in order to find an MTP with the lowest error, we have to train several MTPs with different initial non linear coefficients. This is synonymous with randomly sampling several points in the MTP hyper parameter space and performing a local optimization of the non linear coefficients.

This can be done using a script like this `MTP_training.py` which uses a for loop over different *random_seed* value used to initialize the coefficients. Training data from the files *reference-data.hdf5* and *active_learning_candidates.hdf5* are used as input in this batch training. This script also uses 80% of the reference data for training and 20% for testing.

Note: This step involves training about 200 MTPs in sequence and might take several hours. For better performance choose MPI over openMP, as the MTP training implementation primarily supports MPI parallelization. Users could reduce the number of MTPs trained in line 32 of the script. By default, the script runs a non-linear coefficients

optimization for each fit. To accelerate the training, users can turn the optimization off by specifying *perform_optimization=False* and *initial_coefficients=Random*. Then the fits run without local optimization of the non-linear coefficients and the user can choose to separately optimize the coefficients for the best fit from the 200.

The log output lists training errors for all of the MTPs generated from running the script and the MTP with lowest error for the training/test set (Energy RMSE: 10.7/10.7 meV/atom; Force RMSE: 0.39/0.40 eV/Å; stress RMSE:10.5/9.5 meV/Å^3) is chosen for the validation MD test. Scatter plots comparing the energy, force and stress computed from MTP and reference method are shown below.



Note: Please note that training errors and the above plot will look different when any change to the scripts in steps 1, 2 and 3 are made. As long as the errors are comparable to the above given values, the MTP can be used in validation MD.

The MTP stated above can be generated using this script `MTP_training_best.py`, provided the same reference data obtained from executing `active-learning.py` is used.

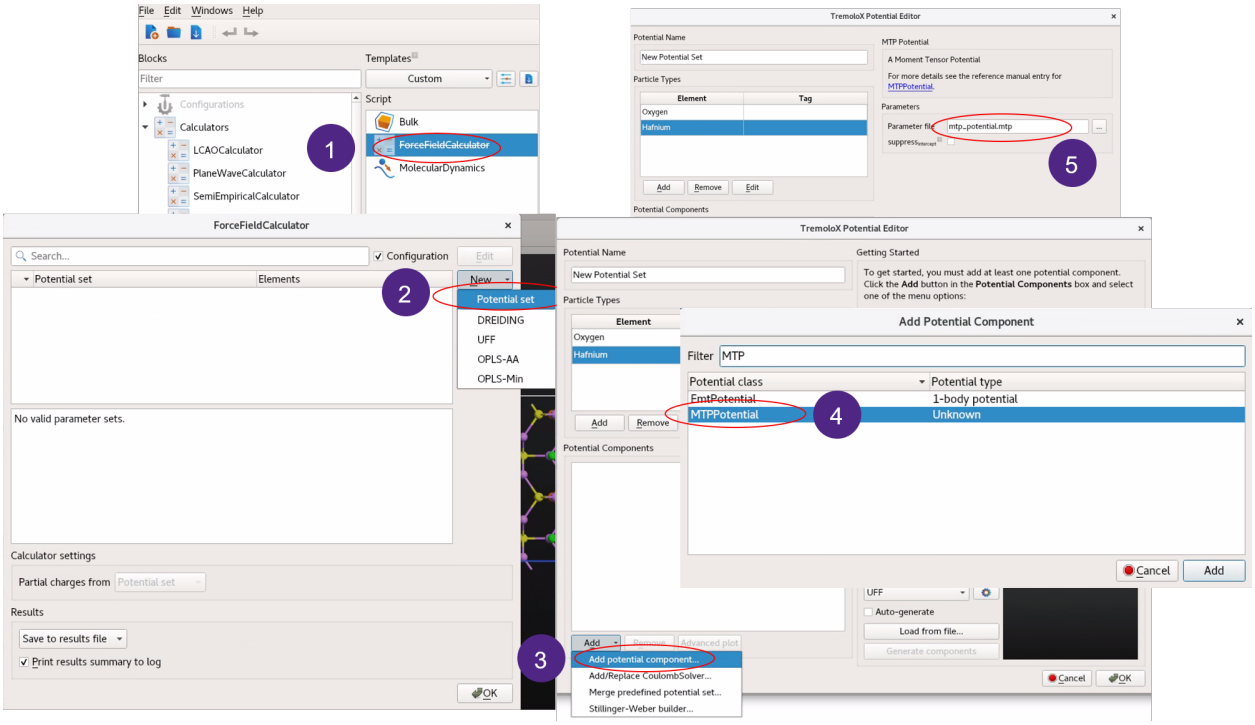
Note: In the above script, the random seed for initializing the non linear coefficients is choosen as that which gave the lowest force RMSE when running MTP_training.py script. It is an optional step since the final mtp file is available from the output of MTP_training.py anyways.

The scatter plot shown above can be created by running atkpython in a terminal and calling the below lines of code while inside the folder with the results for MTP_training_best.py.

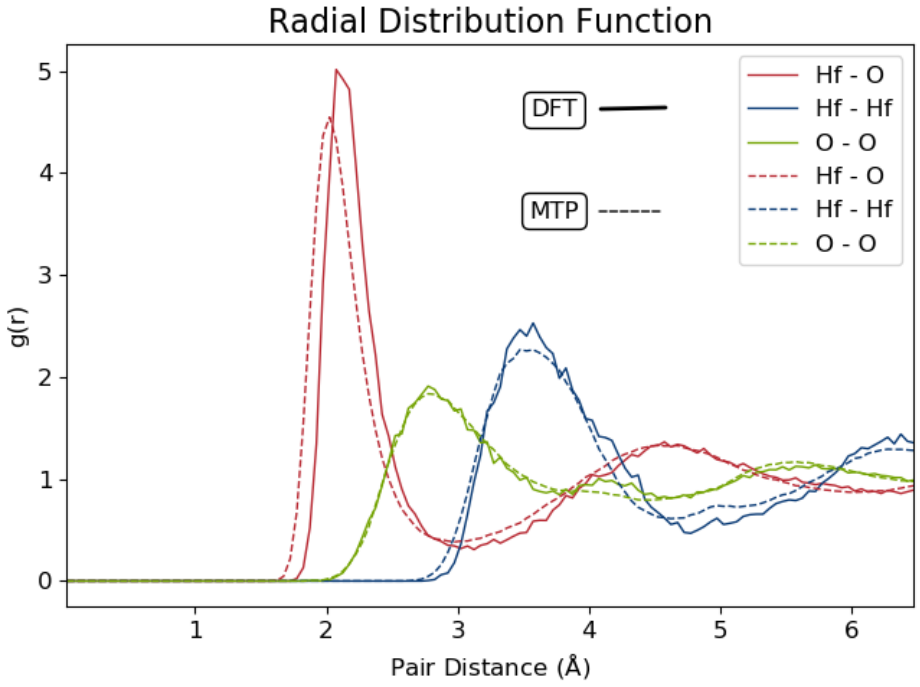
```
moment_tensor_potential_training=nlread('MTP_training.hdf5',
↪MomentTensorPotentialTraining)[-1]
moment_tensor_potential_training._nlplotScatter(fit_index=0)
```

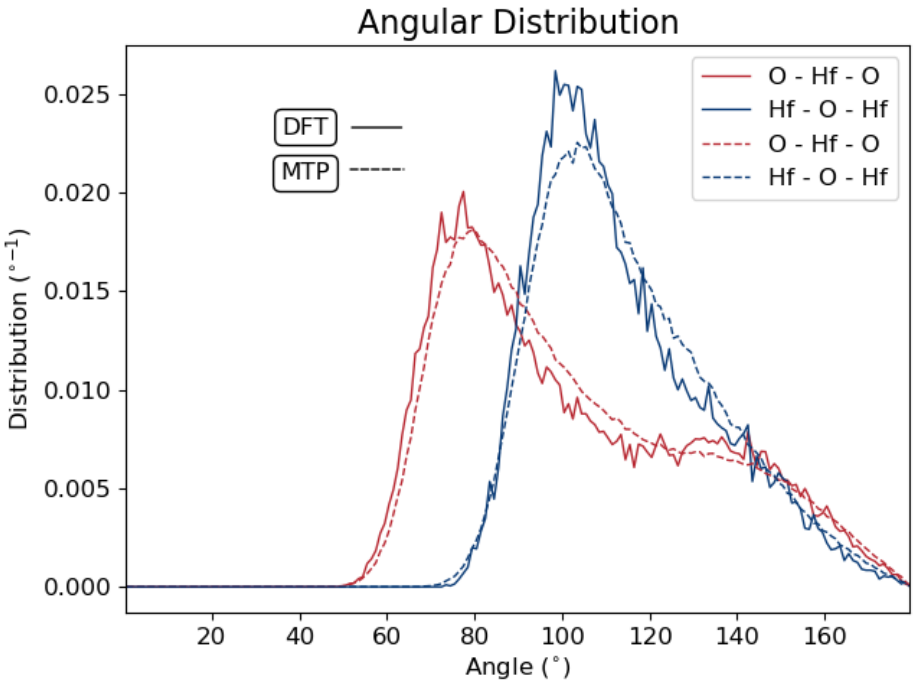
Validation MD Simulation

Now, the MTP obtained in Step 3 will be employed in an MD simulation of amorphous HfO_2 for 0.1 ns using this script `MD.py`. The mtp file used in this script can be found here `mtp_potential.mtp`. This mtp parameter file can be loaded to the script generator via GUI as shown below (click the graphic to see enlarged).



The radial and angular distribution functions from the resulting trajectory is compared with those from a DFT-MD simulation using the same XC functional but reduced basis set as shown below.





As can be seen from the images above, there is a good agreement between the MTP generated trajectory and the DFT trajectory.

The above procedure can be directly applied to train accurate MTPs describing any bulk system of interest with minimal effort using **QuantumATK**.

References

FAQ Section

Q1. In step 1, one prepares the initial configurations for their system. How important is it to define all of the different phases of the material since in the next step (during active learning), the missing configurations are added anyways?

A1. Active learning is used to find and add missing configurations in the initial dataset with the objective to map out the configurational space of interest. But for active learning to work well, the initial training data should have at least coarsely sampled all relevant regions of the configurational space of interest to some extent. So, for example, if your interest is to get the properties of one particular crystalline phase of HfO_2 only – then it is enough to include the randomly displaced crystalline geometries of that crystalline phase only in step 1. Then the active learning, using the chosen crystalline phase as the initial geometry for molecular dynamics, ensures finding missing geometries to represent that crystalline phase accurately. Now, if describing amorphous HfO_2 is your goal, then it helps to include crystal displacement data for all available phases in step 1 and do the active learning as described in the tutorial.

Q2. Is step 1 always necessary or can one start from active learning directly?

A2. If one looks at the workflow figure in the tutorial, it mentions training a preliminary MTP from the initial training data and use it to run the active learning MD (not DFT MD). DFT is employed only when missing geometries are found and needed to be added to the training data. So, we want to make sure that this preliminary MTP is already good enough to sample the equilibrium around the crystal geometries. This is where step 1 helps us immensely and becomes necessary.

Q3. This question is bout the fitting parameter options in step 2, snippet3. Is there some way to guess these options for particular system - for example cutoff radii (check if they have any sense) without actually finishing step 2.

A3. In snippet 3, there are three blocks. The constant term block is parameter free. Keywords in non-linear optimization block are also mostly universal. System specific keywords are in the MTP training block. Outer radii can be chosen based on the radial distribution function of the system and how many coordination shells you would like to include in the atomic description. Basis set size should be chosen based on the complexity of the atomic environment. Our suggestion is to start active learning with a smaller basis set (around 300 functions) and repeat it with increased basis. Step 2 includes computing the initial training geometry and active learning in which the former is the computationally expensive part. Once the initial training data is computed, re-running the script in the same folder will use the already computed initial reference data and only perform active learning part again. The costly step in active learning is the DFT calculation of the missing geometries. So every time you repeat active learning, the active learning candidates from previous run can be added to the initial training data set.

Q4. Under non-linear parameters optimization, what else could be optimized if energy_only option is False and when this is recommended?

A4. When this keyword is set to False, then the non-linear parameters are optimized with respect to E, F and stresses. We suggest using energy only optimization in active learning. If needed, you can optimize on forces and stresses in batch learning. Note that this is only a local optimization in the hyper parameter space.

Q5. In the user example for ActiveLearningSimulations in the manual, Hf and O isolated energies are given in the example script but here, in the tutorial, we dont have them defined. Why is that?

A5. In snippet 3, there is a code block to compute the constant terms, which are the atomic energies of the elements in the system. They are computed automatically before training the preliminary MTP for active learning.

Q6. In step 3, it is said that this may take several hours to get done. Where could one define the parallel options? Would it be possible to do in parallel the steps from the loop?

A6. Yes, one can run the script in step 3 multiple times with different random state while keeping the index in the loop smaller. Then consolidate the results from all the individual log files to find the one with the lowest error.

Q7. Regarding the MTP_training.py and MTP_training_best.py, the former contains a loop over random seeds values, while in the second one the value is given. Is the best (lowest RMSE) seed chosen automatically (given somewhere in main output of MTP_training.py) or user need to actually screen all these mtp_fit_mtp_i.log files?

A7. In this example, the best random seed is not found automatically. The output of MTP_training.py (MTP_training.log) contains a list of RMSEs and the corresponding MTP file names at the end of the file. One can choose the .mtp file with the lowest force error and use it for validation MD. So, it is optional to run MTP_training_best.py. However, one can write a print statement (print(i,rand)) in line 34 of the script MTP_training.py to get the values of rand printed to the log file.

Q8. This question is about validation MD simulation step. MD.py script contains MTP obtained in the step 3. But on the last two pictures there displayed also results of DFT-MD calculations. So what basis sets were used for the DFT-MD run?

A8. For validation, a short DFT-MD with PBE XC functional and FHI Single Zeta Polarized basis set is performed to get the DFT results.

3.1.17 Simulating Si Deposition using Silane

Version: R-2020.09-SP1

In this tutorial, we will look at a QuantumATK workflow to simulate the reaction mechanisms involved in the deposition of Si using silane. We will perform surface-science style slab calculations using plane-wave DFT to study the adsorption and dissociation of silane along with the formation and desorption of H_2 from the surface.

Background

Using thermal chemical vapor deposition (TCVD) technique, Si is deposited by gas phase pyrolysis of SiH_4 . SiH_4 molecules impinge on the Si substrate and donates H atoms to the Si surface followed by the deposition of SiH_x . The H atoms on the surface form H_2 molecules and desorb at elevated temperatures. This is the process we will simulate in this tutorial following the reference article[1]. This can be done in 3 steps. Step 1 involves computing the reference bare Si(100) surface in a supercell and an SiH_4 molecule in the gas phase. In step 2, we study the adsorption and dissociation of SiH_4 on the relaxed Si(100) surface. Step 3 concerns the formation and desorption of an H_2 molecule from the Si surface.

[1] ThongăN.-M. Le, P.ăRaghunath, LamăK. Huynh, and M.ăC. Lin. A computational study on the adsorption configurations and reactions of SiHx (x=1-4) on clean and H-covered Si(100) surfaces. *Appl. Surf. Sci.*, 387:546–556, 2016. URL: http://www.sciencedirect.com/science/article/pii/S0169433216313289, doi:https://doi.org/10.1016/j.apsusc.2016.06.099.

Getting started

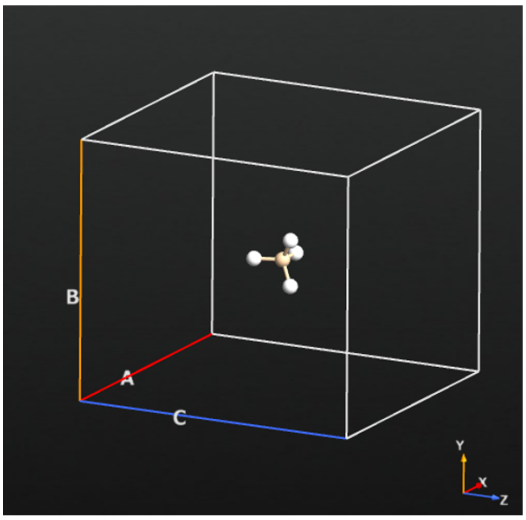
Open the QuantumATK GUI, NanoLab, and select Create New in the Project Dialog. Create a new project in an empty folder with an appropriate name.

Step 1: Reference Calculations

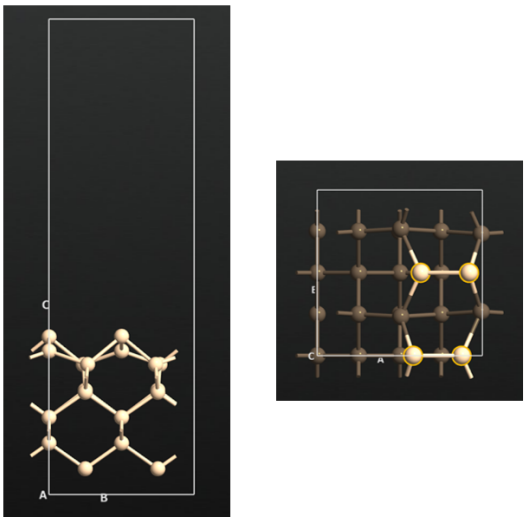
We will perform calculations for the reference SiH_4 molecule and the bare Si(100) surface using a similar setup as the reference article[Page 430, 1]. The relevant python scripts are found here `SiH4.py`, `Si-alpha-100.py`. Panel a in the figure below shows the SiH_4 molecule in a simulation box. Panel b in the figure shows the side and top views of the reconstructed Si(100) surface. From the top view one can see the Si-dimer formation on the surface and these are the reactive sites for SiH_4 adsorption and dissociation. Total electronic energies of these two calculations are summed up to give the reference energy value of the noninteracting system for adsorption energy calculations. The adsorption energy is the difference between the total energy after the process step (interacting) and before the process step (noninteracting) energies:

$$E_{\text{adsorption/binding}} = E_{\text{interacting}} - E_{\text{noninteracting}}$$

a) SiH_4 molecule in a box

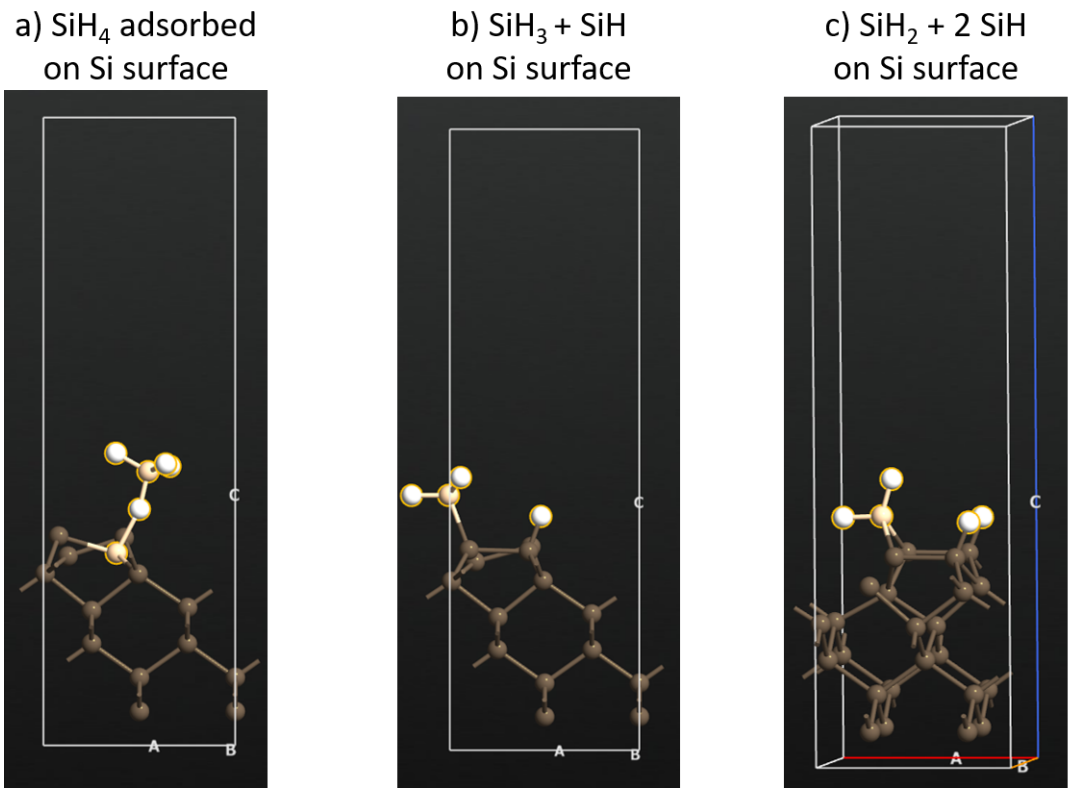


b) Reconstructed α-Si(100) surface in a box

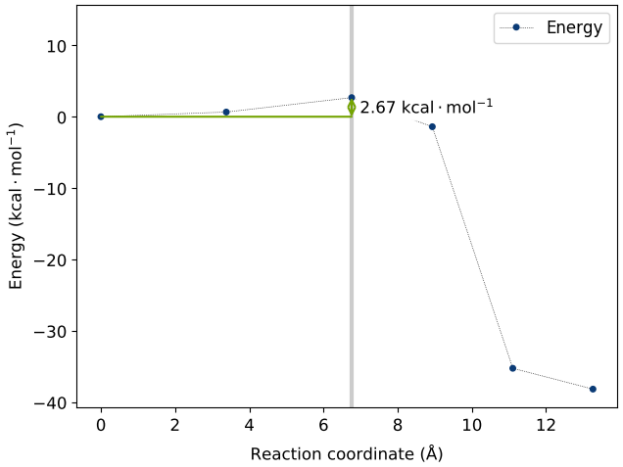


Step 2: Adsorption and Dissociation of SiH_4

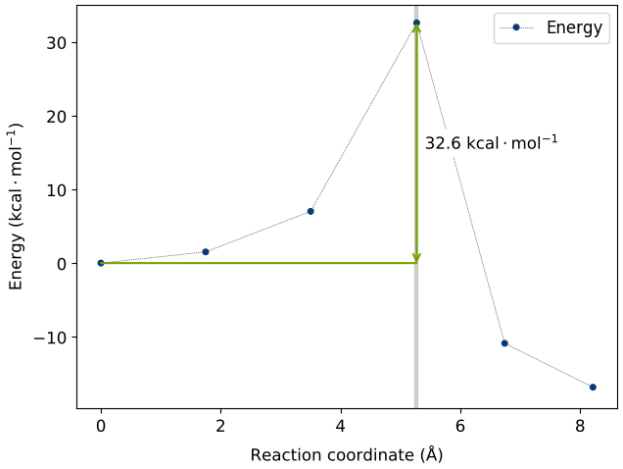
Next step is to adsorb an SiH_4 molecule on the reconstructed Si surface and study the interaction between the two. Panel a in the figure below shows the adsorbed state of SiH_4 on the Si surface in which the molecule is intact and only a weak binding energy of -2.99 kcal/mol (reference article: -2.6 kcal/mol). Panel b in the figure shows a dissociated state of SiH_4 in which a Si-H bond from the molecule dissociates and forms a surface bound Si-H and SiH_3 species. The dissociated state has a binding energy of -41.27 kcal/mol (reference article: -41.3 kcal/mol) which is exoergic and thus favorable. Panel c shows another dissociated Si-H bond from the SiH_3 fragment and formation of 2 Si-H species on the surface in total with a binding energy of -58.11 kcal/mol (reference article: -58.2 kcal/mol). On the reconstructed surface, there are two Si dimers. The SiH_2 fragment takes the bridge site along one of the two surface Si dimers and the H atoms bind to the Si atoms of the other dimer. The scripts to run these geometry optimizations can be downloaded here: `SiH4-Si.py`, `SiH3-SiH.py`, `SiH2-2SiH.py`. A python script to compute the binding energies of these geometries can be downloaded here `adsorption_energies.py`.



Now we can compute the dissociation pathways from the intact adsorbed state (a) towards the two dissociated states (b and c) shown in the picture above. For this we will use the climbing image nudged elastic band approach implemented in QuantumATK. We have used 4 interpolated images between (a and b) and between (b and c). The relevant scripts to run these calculations can be downloaded here: `NEB-a-b.py`, `NEB-b-c.py`. The converged pathways with a maximum force on atoms within 0.1 eV/Å are shown in the animated gifs below. The activation barrier for the first H dissociation connecting geometries a and b is a very low value of 2.67 kcal/mol (reference aricle: 5.2 kcal/mol). This process should not require high temperatures to proceed spontaneously.



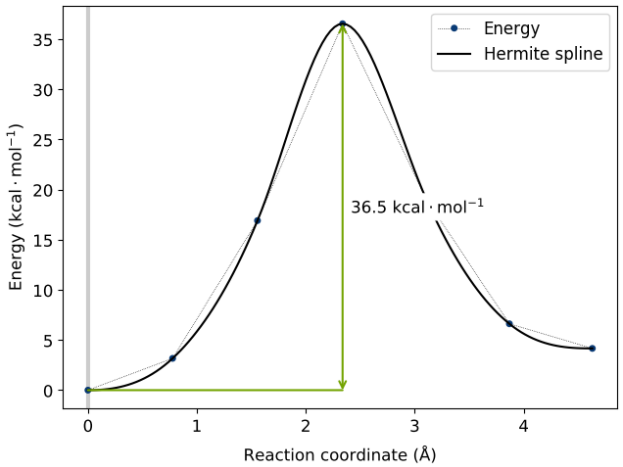
The second H dissociation pathways connecting b and c has an activation barrier of 32.6 kcal/mol (reference article: 31.1 kcal/mol). This pathway requires elevated temperatures to breach the energy barrier.



Step 3: Formation and Desorption of H_2

The H atoms on the surface can diffuse on the Si surface and form H_2 molecule and desorb from the surface. This would then result in the deposition of Si and the removal of H ensures further adsorption of silane molecules on the surface. To study this process, we modified geometry c to have a H_2 molecule adsorbed on the Si surface instead of two separated H atoms. A python script for this calculation can be downloaded here `SiH2-H2.py`. This calculation resulted in the newly created H_2 molecule desorbing into the vacuum spontaneously upon optimization as shown in the below animation.

This means that there is no activation barrier for the desorption of an H_2 molecule once it is formed on the surface. However, there will be an activation barrier for the diffusion of H on the surface which we will compute next. For this, we choose geometry b from before and compute the activation barrier for the diffusion of H from the atop Si site of one dimer to the other as shown in the animation below. The calculated energy barrier is computed to be 36.5 kcal/mol (reference article - Fig.4 TS2: 34.8 kcal/mol). This is a large barrier, however, it is known to be breached at a temperature of 570 K[2].



[2] J.ăH.ăG. Owen, D.ăR. Bowler, C.ăM. Goringe, K.ăMiki, and G.ăA.ăD. Briggs. Hydrogen diffusion on Si(001). *Phys. Rev. B*, 54:14153–14157, Nov 1996. URL: https://link.aps.org/doi/10.1103/PhysRevB.54.14153, doi:10.1103/PhysRevB.54.14153.

Conclusions

In this tutorial, we computed the energetics related to the mechanisms involved in the Si deposition process and compared the results to a previously published article that used a different PAW based plane-wave DFT code. We showed an excellent agreement with the published research article. This tutorial demonstrates the capabilities of QuantumATk in modeling the complex mechanisms involved in thin film processing.

References

3.2 Semiconductors

3.2.1 Phonon-limited mobility in graphene using the Boltzmann transport equation

Version: Q-2019.12-SP1

Downloads & Links

DynamicalMatrix *HamiltonianDerivatives* *ElectronPhononCoupling* *Mobility*

In this tutorial you will learn how to calculate the phonon-limited mobility in graphene. The mobility will be calculated using the Boltzmann transport equation (BTE) with the electronic structure, phonons and electron-phonon coupling calculated using density functional theory (DFT).

The mobility μ will be calculated using two different methods to calculate the relaxation times τ entering the BTE:

1. **Full angular (k,q)-dependence**: in this method, the full dependency of τ on the electron and phonon wave vectors $\mathbf{k}$ and $\mathbf{q}$ is taken into account, so that $\tau = \tau(\mathbf{k}, \mathbf{q})$. In the following, we will refer to this as the **(k,q)-dependent method**.
2. **Isotropic scattering rate**: in this method, only the energy dependence of τ is considered, so that $\tau = \tau(E)$, and $\tau(E)$ is assumed to vary isotropically in momentum-space. In the following, we will refer to this as the **E-dependent method**.

As it will be shown, the two methods give essentially the same results, but the second method is considerably faster than the first one.

In the *theory section*, you can read about the theoretical background. A more comprehensive description can also be found in the paper[1].

[1] T.ăGunst, T.ăMarkussen, K.ăStokbro, and M.ăBrandbyge. First-principles method for electron-phonon coupling and electron mobility: Applications to two-dimensional materials. *Phys. Rev. B*, 93:035414, Jan 2016. doi:10.1103/PhysRevB.93.035414.

Geometry and electronic structure of graphene

In the **Builder**, add a graphene configuration by clicking *Add* → *From Database* and searching for *Graphene* in the structure database.

Next, you have to increase the vacuum gap above and below the graphene. Click on *Bulk Tools* → *Lattice parameters* and set the lattice parameter along the C-direction to C = 20 Å.

Center the configuration by clicking on *Coordinate Tools* → *Center*, and press *Apply*. Now click on the button to send the structure to the **Script Generator**. In the main panel of the **Script Generator**, set the **Results file** to `Graphene_relax.hdf5`,

To perform an accurate calculation of the relaxation times τ and the mobility μ, the first important step is to optimize the lattice parameters A and B and calculate the electronic band structure. In order to do this, add an **LCAOCalculator** block and in the *Main* section set the following parameters:

- *Calculators* → *LCAOCalculator*

–Set the **Exchange correlation** to *LDA*

–Set the **Pseudopotential** to *FHI*

–Set the **Occupation method** to *Methfessel-Paxton*

–Set the **Density mesh cutoff** to *90 Ha*

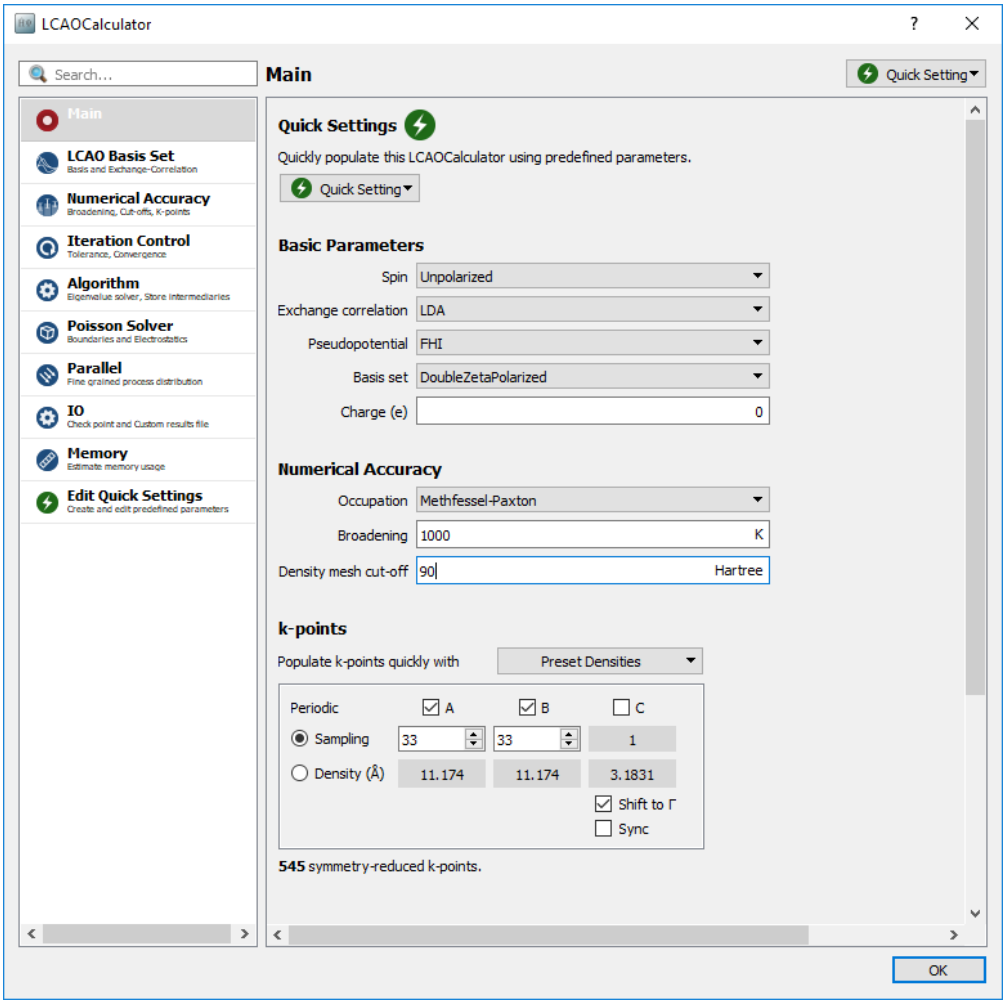
–Set the **k-point sampling** to:

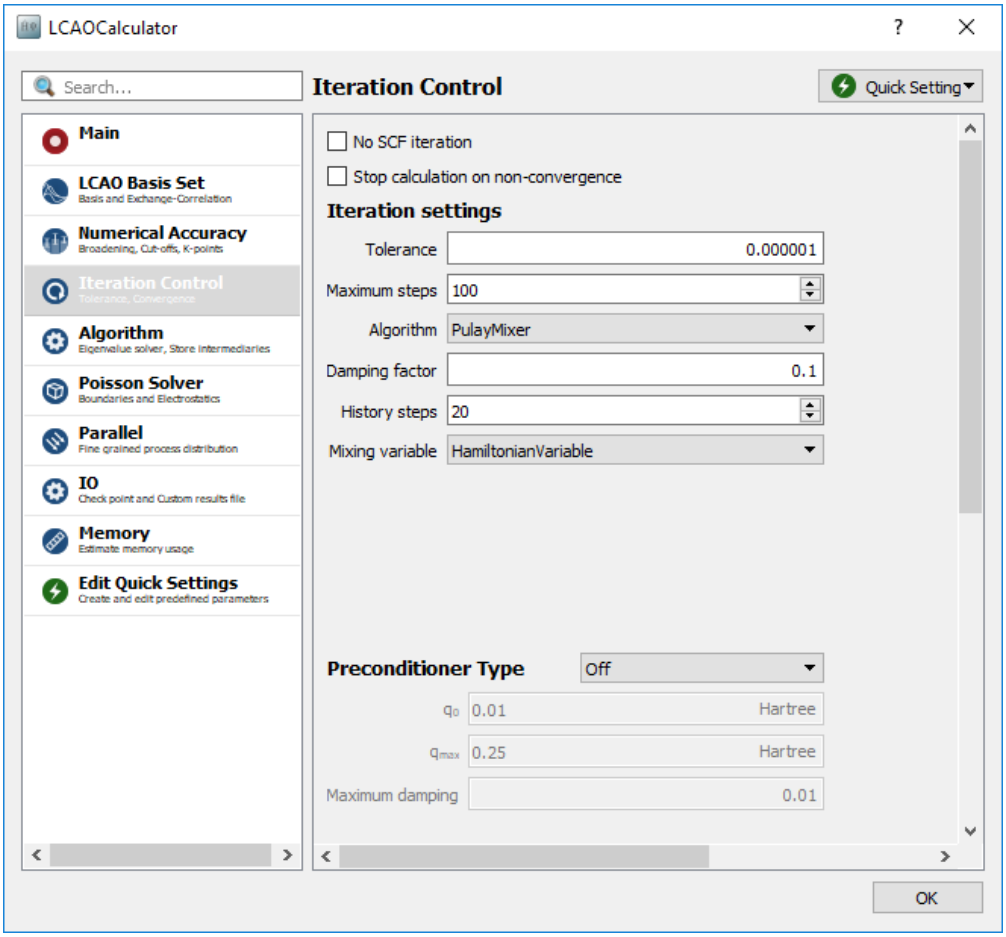
∗k_A = 33

∗k_B = 33

∗k_C = 1

–On the **Iteration Control** tab, Set the **Tolerance** to *0.000001*, i.e. 10^{-6}

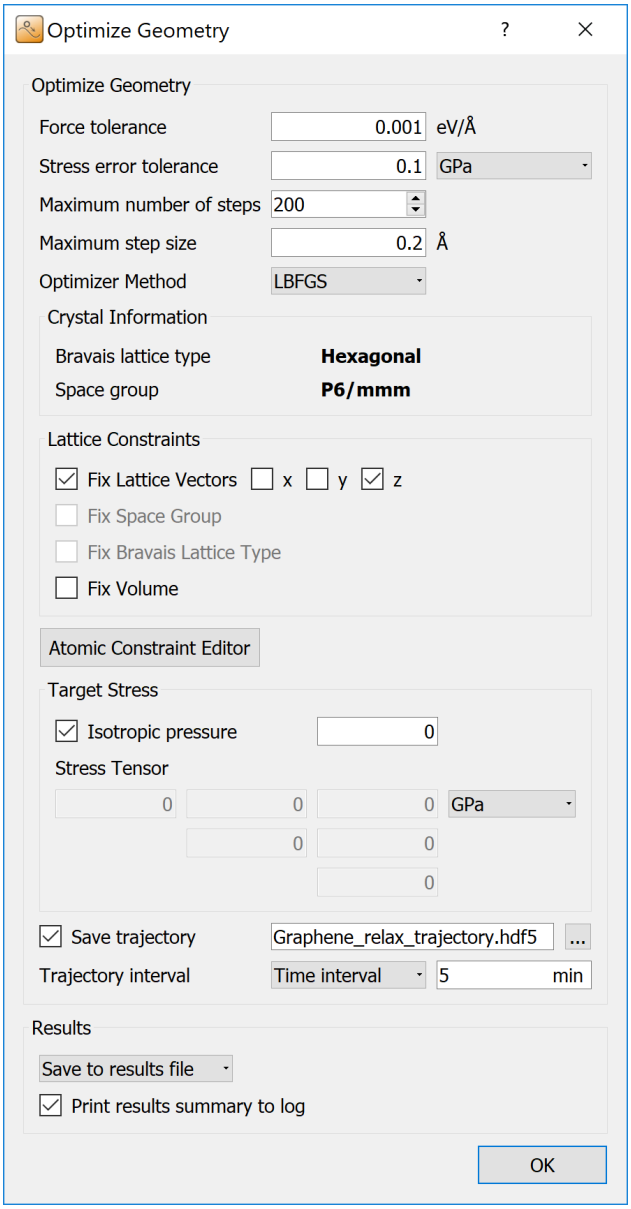

LCAOCalculator
Search...
Main
Quick Setting
Main
LCAO Basis Set
Basis and Exchange-Correlation
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Algorithm
Eigenvalue solver, Store intermediaries
Poisson Solver
Boundaries and Electrostatics
Parallel
Fine grained process distribution
IO
Check point and Custom results file
Memory
Estimate memory usage
Edit Quick Settings
Create and edit predefined parameters
Quick Settings
Quickly populate this LCAOCalculator using predefined parameters.
Quick Setting
Basic Parameters
Spin
Unpolarized
Exchange correlation
LDA
Pseudopotential
FHI
Basis set
DoubleZetaPolarized
Charge (e)
0
Numerical Accuracy
Occupation
Methfessel-Paxton
Broadening
1000
K
Density mesh cut-off
90
Hartree
k-points
Populate k-points quickly with
Preset Densities
Periodic
A
B
C
Sampling
33
33
1
Density (Å)
11.174
11.174
3.1831
Shift to Γ
Sync
545 symmetry-reduced k-points.
OK



- *Optimization* → *OptimizeGeometry*

–Set the **Force tolerance** to *0.001 eV/Å*

–Untick **Fix lattice vectors** in the x and y directions.



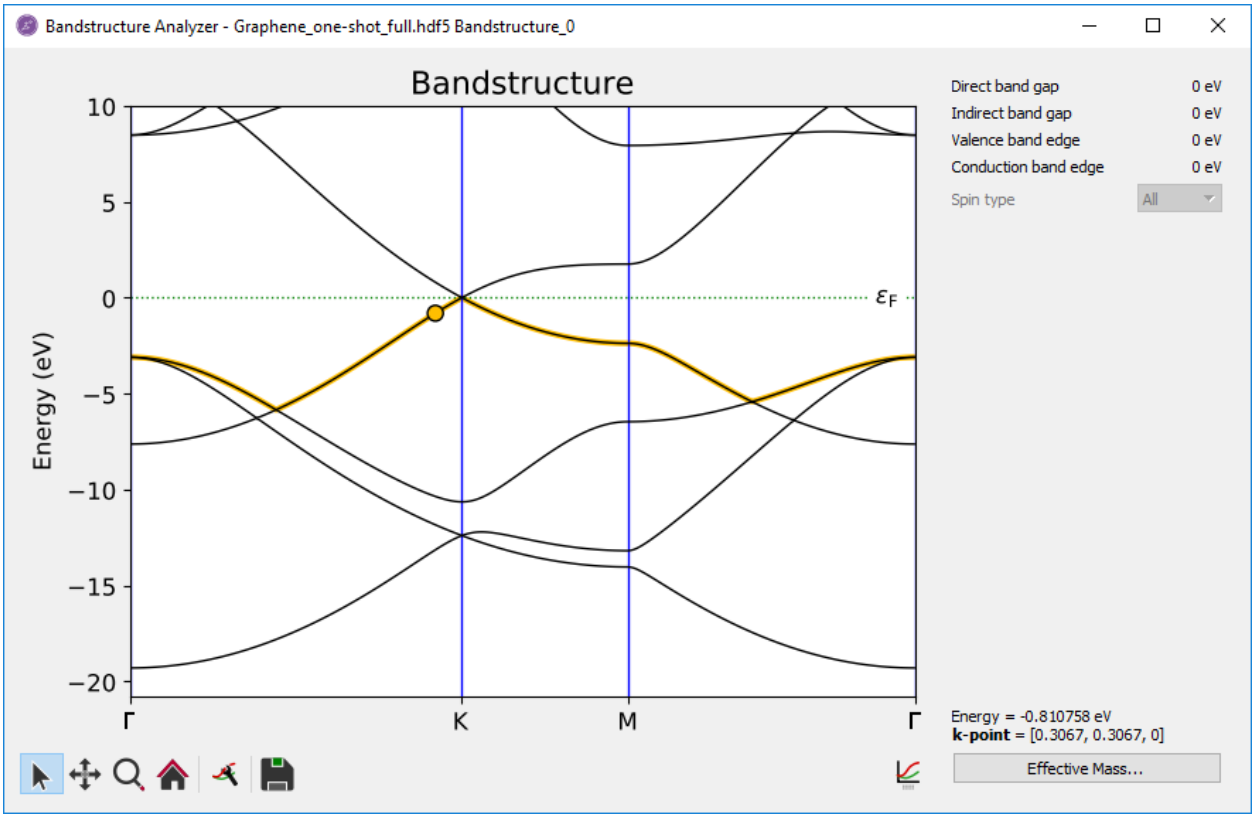
- *Analysis* → *Bandstructure*

–Set the number of **Points per segment** to *100*

–Set the **Brillouin zone route** to *G, K, M, G*

Now, send the script to the **Job manager**, save it as `Graphene_relax.py`, and click on the button to run the calculation.

When the calculation is done, click on the **Bandstructure** object contained in the file `Graphene_relax.hdf5` on the **LabFloor** and use the **Bandstructure Analyzer** to visualize the band structure.



By placing the mouse cursor on top of a band, information about the band is shown. You see that the valence band (highlighted in yellow in the figure above) is number 3 and the conduction band is number 4. These are the two electronic bands relevant for the calculation of the mobility, and in the following we will concentrate on these two bands.

Phonons in Graphene

The next step is to calculate the dynamical matrix of graphene. In order to test the quality of the result you will also calculate the phonon band structure, which is based on the calculated dynamical matrix.

Open the **Script generator** , and create a new script:

- Add an *Analysis from File* block and select *BulkConfiguration_1* in `Graphene_relax.hdf5`.
- Add a *Study Objects → DynamicalMatrix* block and modify the following settings:

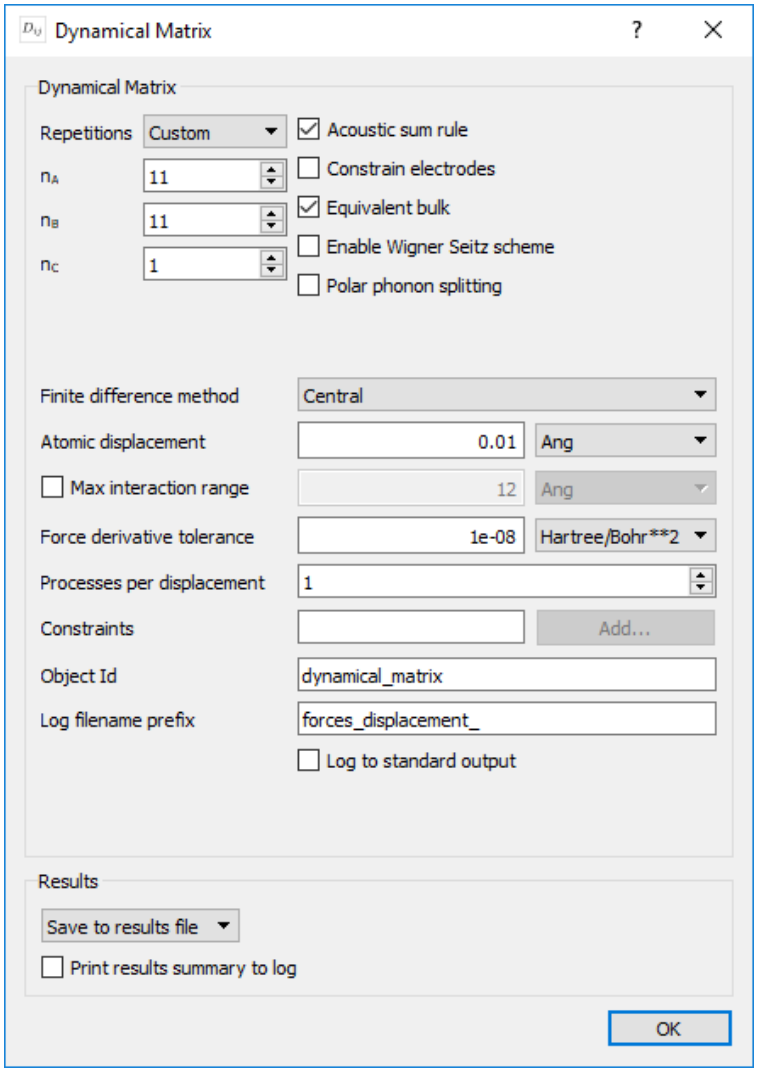
  – Set **Repetitions** to *Custom*

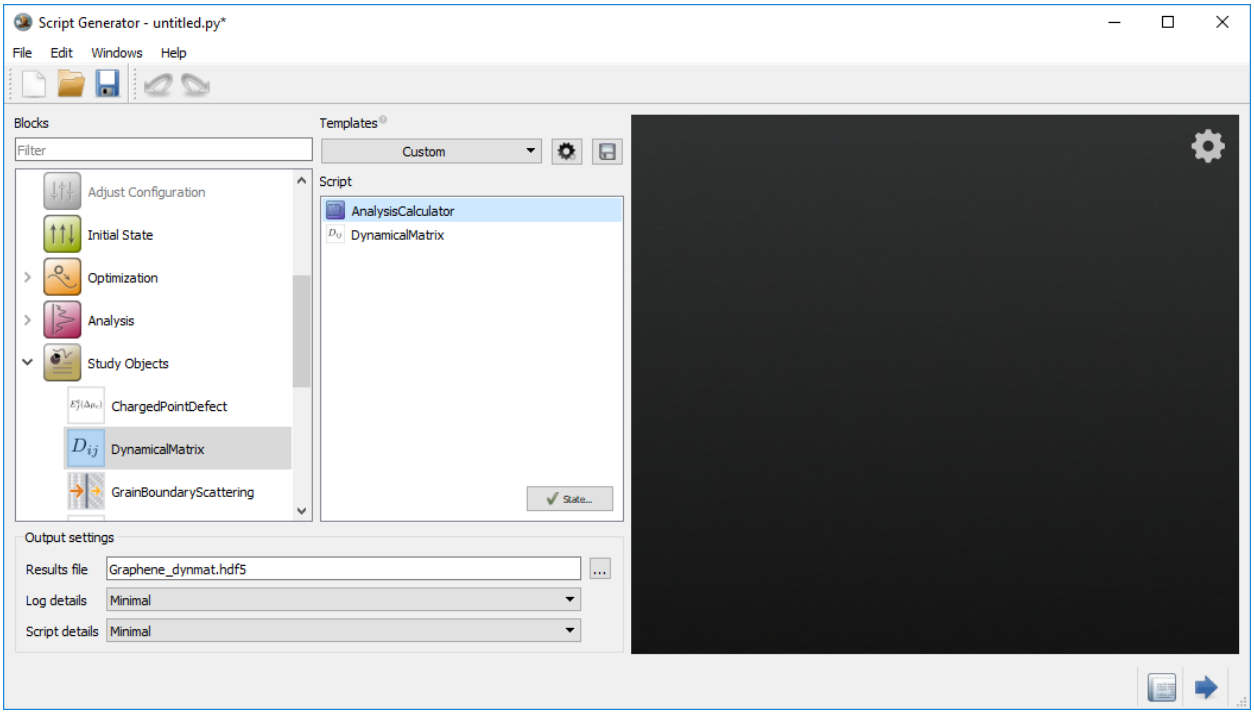
  – Set the **Number of repetitions** to:

  $*n_A = 11$

  $*n_B = 11$

  $*n_C = 1$


Dynamical Matrix
?
Dynamical Matrix
Repetitions
Custom
Acoustic sum rule
n_A
11
Constrain electrodes
n_B
11
Equivalent bulk
n_C
1
Enable Wigner Seitz scheme
Polar phonon splitting
Finite difference method
Central
Atomic displacement
0.01
Ang
Max interaction range
12
Ang
Force derivative tolerance
1e-08
Hartree/Bohr**2
Processes per displacement
1
Constraints
Add...
Object Id
dynamical_matrix
Log filename prefix
forces_displacement_
Log to standard output
Results
Save to results file
Print results summary to log
OK



Note: In previous versions of **QuantumATK** it was necessary to manually scale down the number of k-points when doing a **DynamicalMatrix** or **HamiltonianDerivatives** calculation to take into account the repeated cell. From **QuantumATK-2019.03** and later versions the k-points are automatically scaled to maintain the same accuracy as the unit cell, and we can simply re-use the calculator settings.

Tip: Due to the large dimensions of the 11×11 graphene super cell (242 atoms), the calculation of the *DynamicalMatrix* object is rather time consuming. However, the calculation can be parallelized over the atomic displacements. Since there are two atoms in the graphene unit cell, and each is displaced in the x, y and z directions, there are in total 6 calculations to be performed. Maximum efficiency is obtained when the number of calculations times the value of the parameter *Processes per displacement* matches the total number of cores used for the calculations. In the present case, the calculation takes about 30 minutes if *Processes per displacement* = *4* and the calculation is run on 24 cores.

- Add an *Analysis* → *PhononBandstructure* block and set the following parameters:
  - Set the number of **Points per segment** to *100*
  - Set the **Brillouin zone route** to *G, M, K, G*

Finally, in the main panel of the **Script Generator** set the **Default output file** to `Graphene_dynmat.hdf5`, send the script to the **Job manager**, save it as `Graphene_dynmat.py` and click on the button to run the calculation.

Warning: Remember to make the file `Graphene_relax.hdf5` available for the script if you are running on a cluster. In the **Job Manager** you can use the **I/O** tab to transfer additional files together with a script. In this case, you may also need to modify the path in the script before submitting to the cluster, to make sure it points at the correct location of the file.

When the calculation is done, go back to the in the **LabFloor**, and inspect the $\{\omega_i(\mathbf{q})\}$ **PhononBandstructure** object contained in the file `Graphene_dynmat.hdf5` using **Compare data** or **PhononBandstructure Analyzer**. The calculated phonon band structure should match with that shown in the figure below.

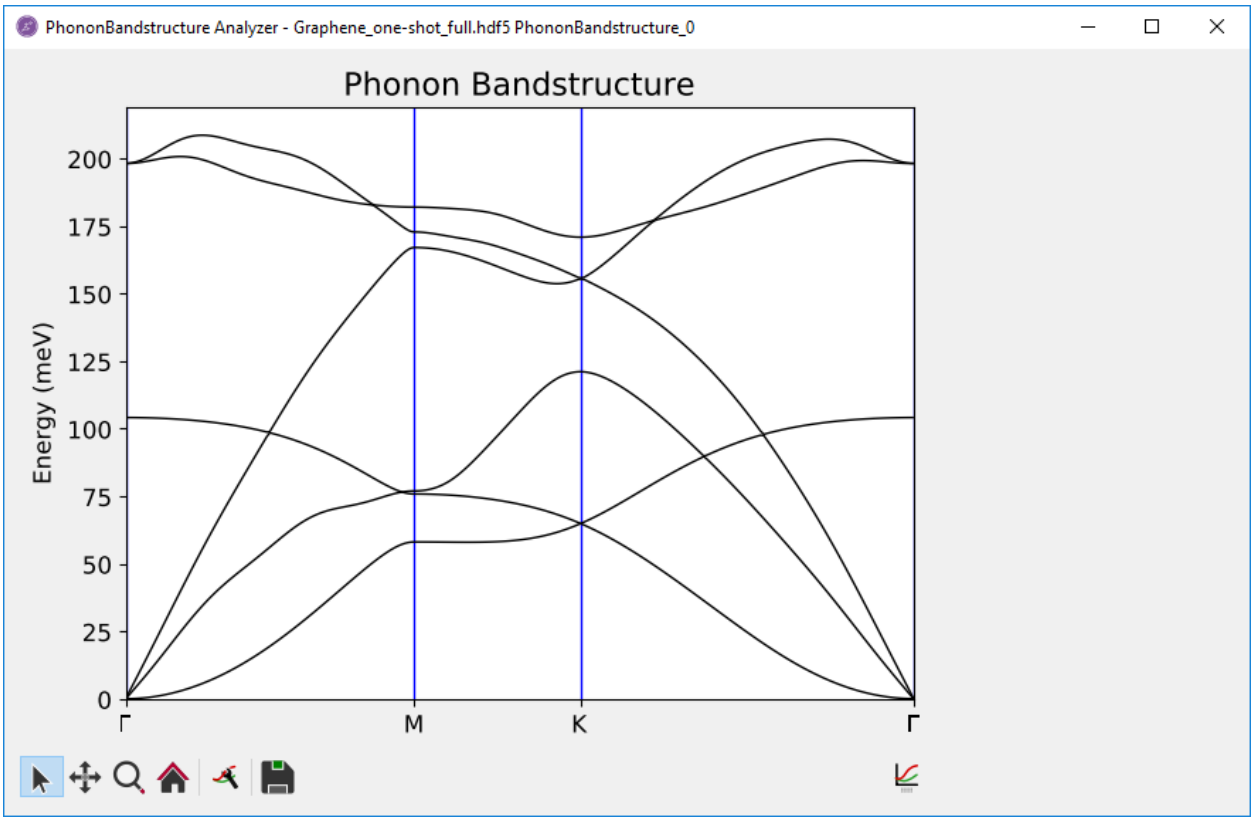


Fig. 3.18: Phonon bandstructure. There are three acoustic modes: The lowest, out-of-plane mode (ZA) has a $\propto \mathbf{q}^2$ for small $\mathbf{q}$, while the two next have linear $\mathbf{q}$-dependence with a constant velocity for small $\mathbf{q}$. The transverse acoustic (TA) mode is lower in energy than the longitudinal acoustic (LA) mode.

Mobility of graphene

In the following, the procedure to calculate the electron mobility in graphene will be described. Provided that one has already calculated the electronic structure and dynamical matrix of the system, the following three steps are necessary to evaluate the mobility:

1)Calculation of the **Hamiltonian derivatives**

2)Calculation of the **Electron-phonon couplings**

3)Calculation of the **Mobility**

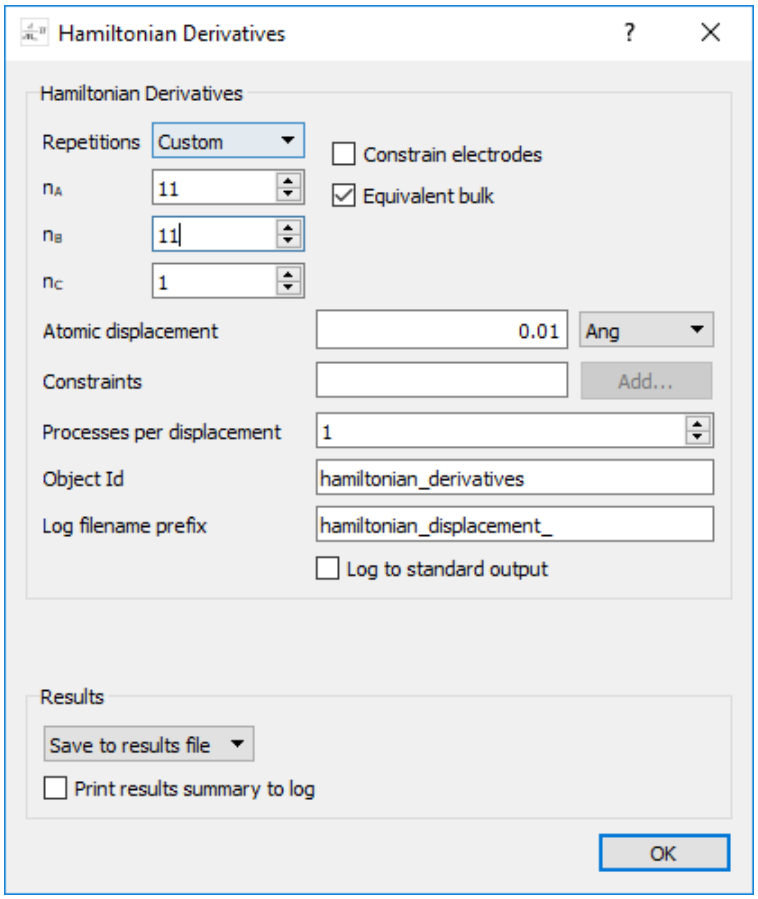
The two methods to calculate μ described above differ in the way in which steps 2 and 3 are carried out. These two steps will therefore be described separately for each method. The procedure for the **(k, q)-dependent method** is outlined in sections *1*, *2A* and *3A*, whereas the procedure for the **energy-dependent method** is outlined in in sections *1*, *2B* and *3B*

1. Hamiltonian derivatives

In order to calculate the electron-phonon coupling matrix, it is necessary to calculate the derivative of the Hamiltonian $\partial\hat{H}/\partial R_{i,\alpha}$ with respect to the coordinate of the i-th atom along the Cartesian direction α.

Open the **Script generator**, and modify the script as follows:

- Add an *Analysis from File* block and select *BulkConfiguration_1* in `Graphene_relax.hdf5`.
- Add a *Study Objects* → *HamiltonianDerivatives* block and set the **Number of repetitions** to:
  - n_A = 11
  - n_B = 11
  - n_C = 1



Warning: It is important that the number of repetitions matches the calculation of the dynamical matrix.

In the main panel of the **Script Generator**, set the **Results file** to `Graphene_dHdR.hdf5`, send the script to the **Job manager**, save it as `Graphene_dHdR.py` and click on the button to run the calculation.

Note: As for the **DynamicalMatrix**, the calculation of the **HamiltonianDerivatives** is rather time consuming, but can be parallelized over the atomic displacements. In the present case, the calculation takes around 1 hour if it is run on 24 cores with *Processes per displacement = 4* .

2A. Electron-Phonon couplings: (k,q)-dependent method

In order to calculate the lifetimes $\tau(\mathbf{k}, \mathbf{q})$ and the mobility μ, we need to calculate the electron-phonon coupling matrix on a fine grid of $\mathbf{k}$- and $\mathbf{q}$-points.

Open the **Script generator** , and modify the script as follows:

- Add an *Analysis from File* block and select *BulkConfiguration_1* in `Graphene_relax.hdf5`.
- Add an *Analysis → ElectronPhononCoupling* block.

You will notice that two additional blocks have been also added:

- *Study Objects → DynamicalMatrix* D_{ij}
- *Study Objects → HamiltonianDerivatives* $\frac{d}{d\mathbf{R}_\alpha}\hat{H}$

In the present case, both the dynamical matrix and the Hamiltonian derivatives have been calculated already and can be re-used. As they are study objects, this is automatically detected by QuantumATK if the provided filename and other input parameters are the same. Open each of them and change the repetitions to 11x11x1 and the filenames to `Graphene_dynmat.hdf5` and `Graphene_dHdR.hdf5`, respectively.

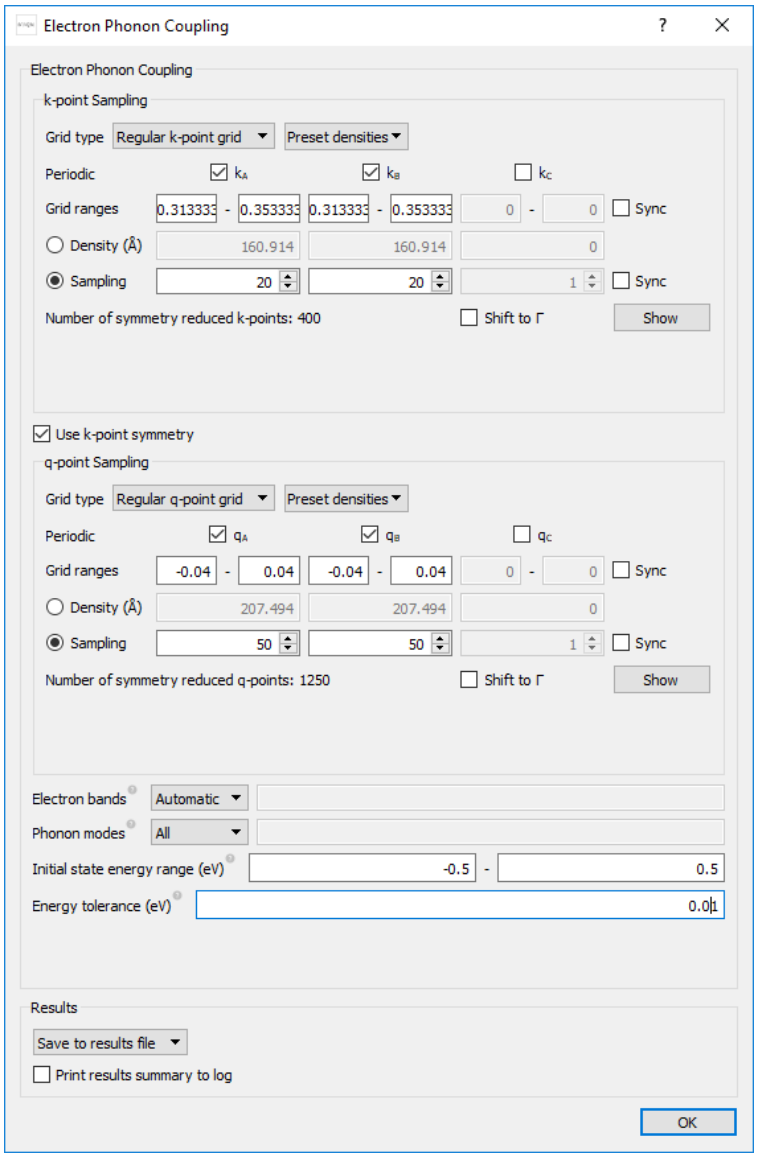
Now set the parameters in the *Analysis → ElectronPhononCoupling* block as shown below. We will sample k-points in a small area around the K-point at [1/3, 1/3, 0], and q-points in a small area around the Γ-point at [0,0,0].

Tip: If you are unsure of the coordinates of a particular symmetry point you may use the built-in functionality of the `BravaisLattice` class, f.ex. like this: `k0=bulk_configuration.bravaisLattice().symmetryPoints()['K']`)

Warning: For production calculations, it is strongly recommended to always converge the sampling resolution in q- and k-space. The above settings are a result of such a study, with more information shown in the appendix: *Convergence of q- and k-point sampling*

In the main panel of the **Script Generator**, set the **Results file** to `Graphene_eph.hdf5`. Send the script to the **Job manager**, save it as `Graphene_eph.py` and click on the button to run the calculation.

Warning: For graphene, we need quite dense samplings in $\mathbf{k}$- and $\mathbf{q}$-space, and therefore we have chosen to sample very finely a small area around the relevant high-symmetry points. This still includes the full angular dependence,


Electron Phonon Coupling
Electron Phonon Coupling
k-point Sampling
Grid type
Regular k-point grid
Preset densities
Periodic
Grid ranges
0.313333
0.353333
0.313333
0.353333
Sync
Density (Å)
160.914
Sampling
20
Number of symmetry reduced k-points: 400
Shift to Γ
Show
Use k-point symmetry
q-point Sampling
Regular q-point grid
-0.04
0.04
207.494
50
Number of symmetry reduced q-points: 1250
Electron bands
Automatic
Phonon modes
All
Initial state energy range (eV)
-0.5
0.5
Energy tolerance (eV)
0.01
Results
Save to results file
Print results summary to log
OK

but neglects inter-valley scattering. We therefore recommend to sample the entire Brillouin zone if this could be an important effect.

Note: Sampling the **k**- and **q**-space using dense meshes means that the calculation might become very time consuming. Contemporary versions of QuantumATK automatically detect the relevant regions of **k**- and **q**-space, and do not compute matrix elements which will not contribute, considerably decreasing the computational time and memory requirements. The two parameters *energy_tolerance* and *initial_state_energy_range* govern the filters which reduce the number of calculated coupling elements. More information about these two parameters can be found on the manual page: `ElectronPhononCoupling`.

As QuantumATK parallelizes over **k**- and **q**-points, a high number of MPI processes can be used if sufficient memory is available. In the present case, the calculation took approximately 10 hours on a 16-core node on a cluster.

3A. Mobility: (k,q)-dependent method

Now open a new Script Generator window and add an *Analysis from File* block and then a *Analysis → Mobility* block. In the *Analysis from File* block, select the file `Graphene_relax.hdf5` and load *BulkConfiguration_1* included in the file. Remove the *ElectronPhononCoupling*, *DynamicalMatrix* and *HamiltonianDerivatives* blocks, and replace them with a *Custom blocks → Code snippet* block. This allows us to easily add arbitrary code to any script without editing the full script manually. Open the block, and write the following:

```
electron_phonon_coupling = nlread('Graphene_eph.hdf5', ElectronPhononCoupling)[0]
```

Set the parameters for the *Analysis → Mobility* block as shown below:

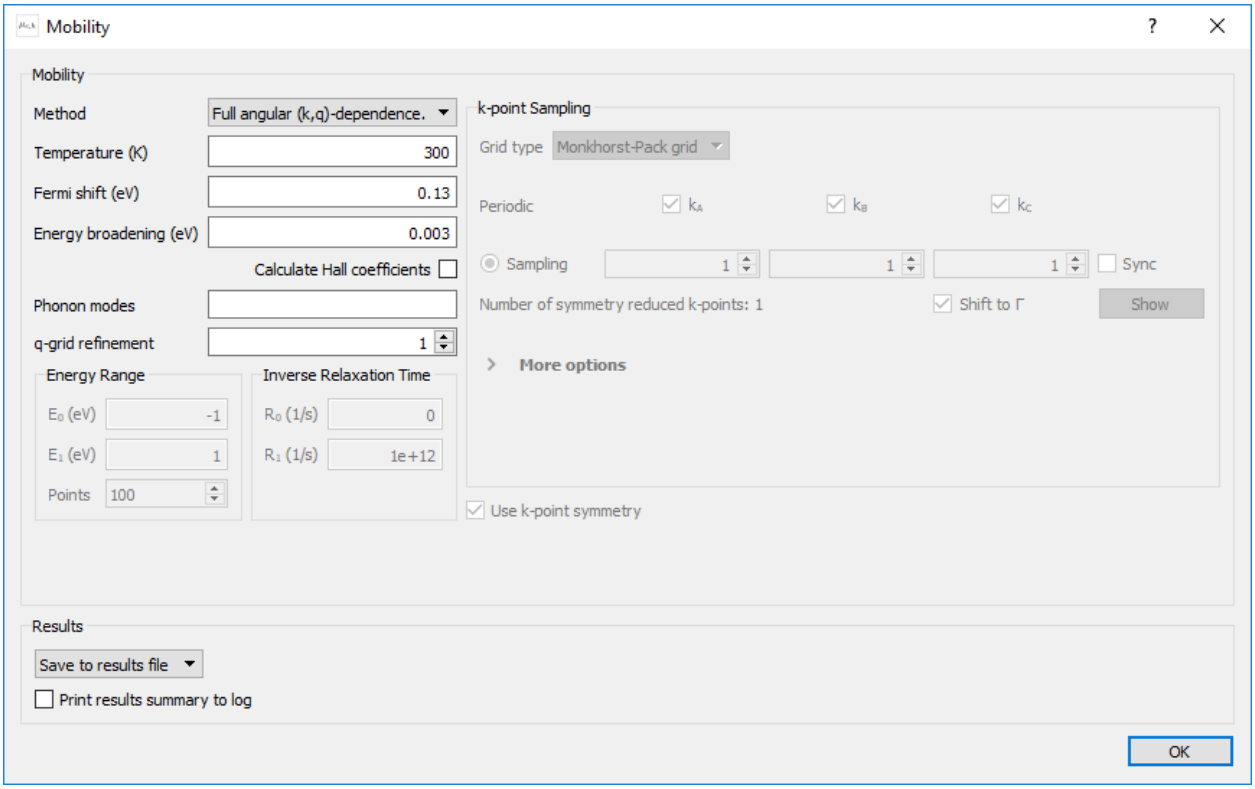
- Leave the **Method** at *Full angular (k,q)-dependence*
- Set the **Fermi shift** to *0.13 eV*
- Untick *Calculate Hall coefficients*

Note: The Fermi shift of 0.13 eV corresponds to a carrier concentration of $n = 10^{12}\mathrm{cm}^{-2}$

In the main panel of the **Script Generator**, set the **Results file** to `Graphene_mu-full.hdf5`, send the script to the **Job manager**, save it as `Graphene_mu-full.py` and click on the button to run the calculation. It should take no more than a few minutes on a laptop.

Note: You will get a notification that the Mobility block is invalid. You can ignore this warning, as NanoLab does not parse the *Code snippet* block and is thus unaware of the `ElectronPhononCoupling` object that will be loaded there.

Once the calculation is done, select the file `Graphene_mu-full.hdf5`, and on the **LabFloor**, select the $\mu_{e,h}$



Mobility object, and click on **Text Representation**. You should now see the following:

```
+-------------------------------------------------------------------------------+
| Mobility Report                                                               |
| ----------------------------------------------------------------------------- |
| Input parameters:                                                             |
|     Temperature       = 300.00 K                                              |
|     Fermi level shift = 0.13 eV                                               |
|     Energy broadening = 0.0030 eV                                             |
|     q-grid refinement = 1                                                     |
|                                                                               |
+-------------------------------------------------------------------------------+
| Trace of linear responce tensors:                                             |
+-------------------------------------------------------------------------------+
|                                                                               |
| Electrons:                                                                    |
|                                                                               |
|     Mobility                =  2.34e+05 cm^2/(V*s)                            |
|     Conductivity            =  4.19e+01 S/m                                   |
|     Seebeck coefficient     = -1.94e-05 V/K                                   |
|     Thermal conductivity    =  2.43e-04 W/(m*K)                               |
|     Carrier density (2D, xy) =  2.24e+06 cm^-2                                |
|                                                                               |
|                                                                               |
| Holes:                                                                        |
|                                                                               |
|     Mobility                =  8.66e-04 cm^2/(V*s)                            |
```

(continues on next page)

(continued from previous page)

```
|     Conductivity              =  1.19e-01 S/m                                  |
|     Seebeck coefficient       =  1.14e-03 V/K                                  |
|     Thermal conductivity      =  1.18e-05 W/(m*K)                              |
|     Carrier density (2D, xy) =  1.72e+12 cm^-2                                 |
|                                                                                |
+--------------------------------------------------------------------------------+
```

The calculated electron mobility, highlighted in yellow, is $2.34 \cdot 10^5$ $\mathrm{cm}^2\mathrm{V}^{-1}\mathrm{s}^{-1}$, and agrees well with previously reported data at room temperature and $n = 10^{12}\mathrm{cm}^{-2}$[2].

Note: Note that the carrier density listed here is significantly lower than $n = 10^{12}\mathrm{cm}^{-2}$. The carrier density is calculated independently from the mobility, and converges much more slowly than the mobility itself. In this tutorial, the focus is on achieving a converged value for the mobility - converging also the carrier density would require inclusion of more **k**-points and/or a larger region in **k**-space in the **ElectronPhononCoupling** calculation. Alternatively, you can calculate the carrier density with a finer sampling directly from the DensityOfStates object: `carrier_density_test.py`.

Tip: Starting from version Q-2019.12, QuantumATK NanoLab includes a Mobility Analyzer. This will be presented in a future update of this tutorial.

2B. Electron-Phonon couplings: energy-dependent method

We will now calculate the electron-phonon couplings to be used for the calculation of the energy-dependent relaxation times $\tau(E)$. We will assume that the relaxation times vary isotropically in **k**-space. This means that it will be sufficient to evaluate the electron-phonon coupling matrix for a number of **k**-points along a line through one of the Dirac (K) points, thereby reducing significantly the computational workload of the calculation.

Open the **Script generator** , and modify the script as follows:

- Add an *Analysis from File* block and select *BulkConfiguration_1* in `Graphene_relax.hdf5`.
- Add an *Analysis → ElectronPhononCoupling* block.

You will notice that two additional blocks have been also added:

- *Study Objects → DynamicalMatrix*
- *Study Objects → HamiltonianDerivatives*

In the present case, both the dynamical matrix and the Hamiltonian derivatives have been already calculated and can be re-used. As they are study objects, this is automatically detected by QuantumATK if the provided filename and other input parameters are the same. Open each of them and change the repetitions to 11x11x1 and the filenames to `Graphene_dynmat.hdf5` and `Graphene_dHdR.hdf5`, respectively.

In the *Analysis → ElectronPhononCoupling* block, modify the parameters as follows:

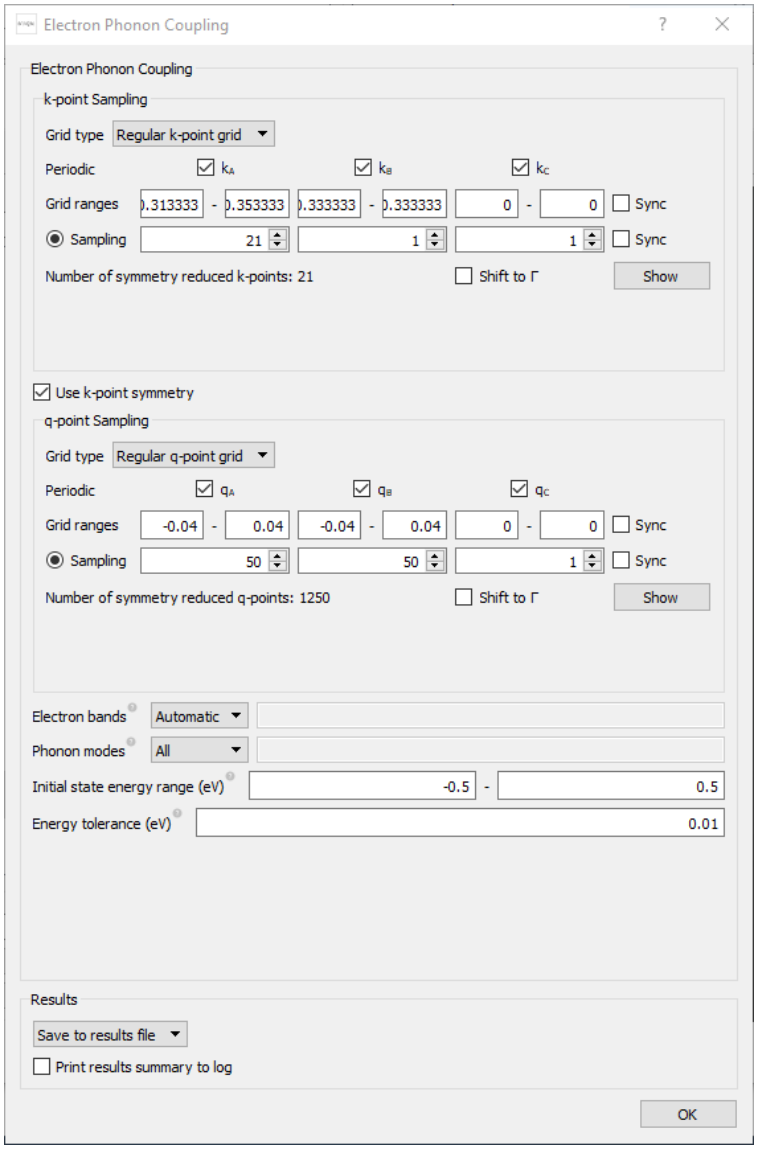
[2] Kristen Kaasbjerg, KristianăS Thygesen, and KarstenăW Jacobsen. Unraveling the acoustic electron-phonon interaction in graphene. *Physical Review B*, 85(16):165440, 2012.

•In the **k-point sampling**, set the **Grid type** to *Regular k-point grid*, and the sampling parameters as shown in the figure below.

Note: As you can see from the figure, the **k**-space is sampled only along a line!

•In the **q-point sampling**, set the **Grid type** to *Regular q-point grid*, and the sampling parameters as shown in the figure below.

•Set the **Energy tolerance** to *0.01* (eV) and the **Initial state energy range** to go from *-0.5* to *0.5*.



In the main panel of the **Script Generator**, set the **Results file** to `Graphene_eph-line.hdf5`, send the script to the **Job manager**, save it as `Graphene_eph-line.py` and click on the button to run the calculation.

Note: Since here we sample the **k**-space only along a line, the calculation will in general be faster than the one described in section 2A. In the present case, the calculation took approximately 50 minutes on a 16-core node on a

cluster, or about 1/10th of the full calculation in section 2A.

3B. Electron mobility: energy-dependent method

We will now use a two-step procedure to evaluate the room-temperature mobility μ based on the energy-dependent relaxation times $\tau(E)$:

- In the first step, the $\mathbf{k}$- and $\mathbf{q}$-dependent relaxation times $\tau(\mathbf{k}, \mathbf{q})$ are evaluated along the line outward from the K-point.
- In the second step, the values of $\tau(\mathbf{k}, \mathbf{q})$ are averaged in $\mathbf{k}$-space to obtain the values of $\tau(E)$, which are then used to calculate the mobility μ.

Now open a new Script Generator window and add an *Analysis from File* block and then a *Analysis → Mobility* block. In the *Analysis from File* block, select the file `Graphene_relax.hdf5` and load *BulkConfiguration_1* included in the file. Remove the *DynamicalMatrix* and *HamiltonianDerivatives* objects and replace the *Analysis → ElectronPhononCoupling* with a *Custom blocks → Code snippet* block. Open the block, and write the following:

```
electron_phonon_coupling = nlread('Graphene_eph-line.hdf5', ElectronPhononCoupling)[0]
```

Finally, set the parameters in the *Analysis → Mobility* block as shown below:

- *Analysis → Mobility*
  - Set the **Method** to *Full angular (k,q)-dependence*
  - Set the **Fermi shift** to *0.13 eV*
  - Untick *Calculate Hall coefficients*

Note: Note that we choose a $\mathbf{k}$-line direction somewhat arbitrarily here. The point is that a low number of $\mathbf{k}$-points can be used to generate an energy dependent rate. An alternative to a line could be to coarsely sample the entire BZ to capture the anisotropy in $\mathbf{k}$-space, but with fewer points than needed in the full ($\mathbf{k}$, $\mathbf{q}$)-dependent method.

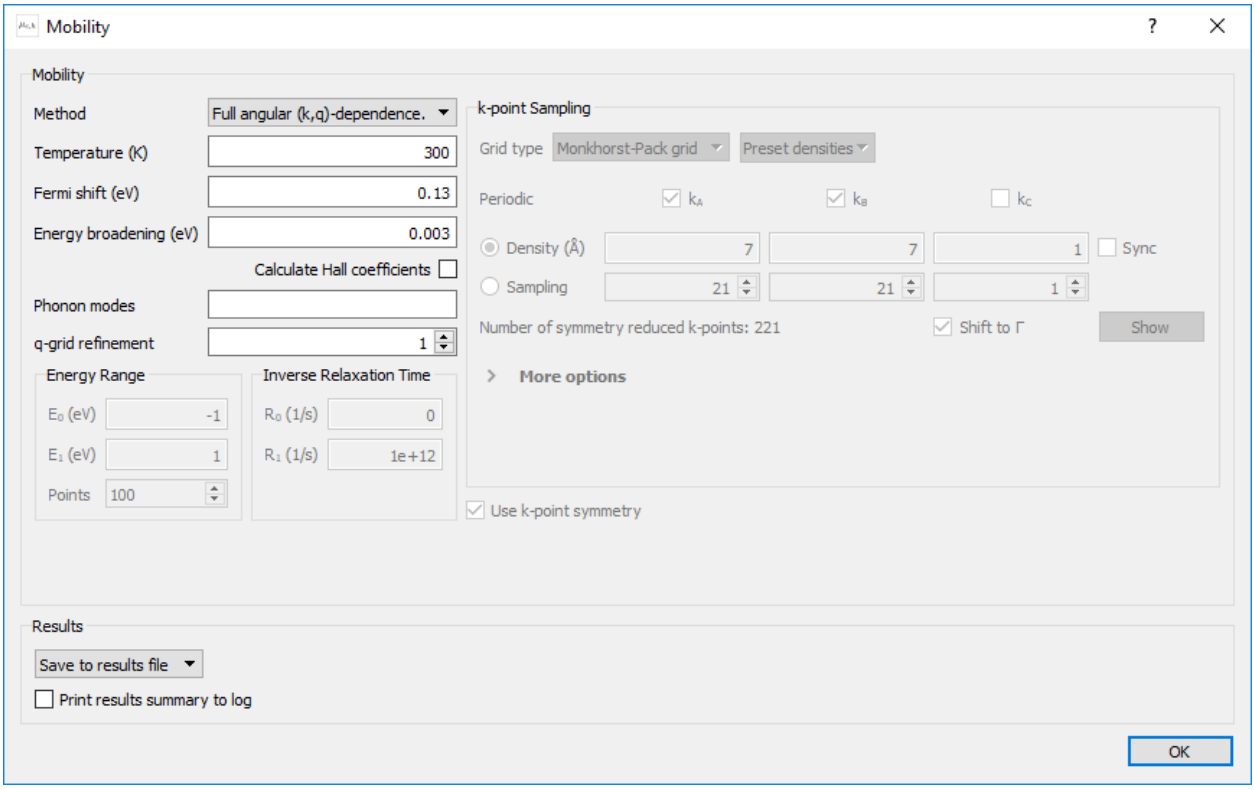
In the main panel of the **Script Generator**, set the **Results file** to `Graphene_mu-line-full.hdf5`, send the script to the **Job manager**,save it as `Graphene_mu-line-full.py` and click on the button to run the calculation. You will again see a warning that the script might be invalid, and you should ignore it this time as well.

Once the calculation is done, you are ready to calculate the mobility using the isotropic scattering rate method. In this case, you will re-use the bulk configuration and the full angular ($\mathbf{k}$, $\mathbf{q}$) -dependent mobility from the `Graphene_relax.hdf5` and `Graphene_mu-line-full.hdf5` files.

Go back to the **Script generator**, and open the *Custom → Code Snippet* block before the Mobility icon. Change the contents to:

```
mobility_full = nlread('Graphene_mu-line-full.hdf5', Mobility)[0]
```

Then modify the *Analysis → Mobility* block as follows:



- *Analysis* → *Mobility* $\mu_{e,h}$

–Set the **Method** to *Isotropic scattering rate*

–Set the **Fermi shift** to *0.13 eV*

–Untick *Calculate Hall coefficients*

–Set the following values of the **Energy Range**:

∗E_0 = -0.24 eV

∗E_0 = 0.24 eV

∗Points = 100

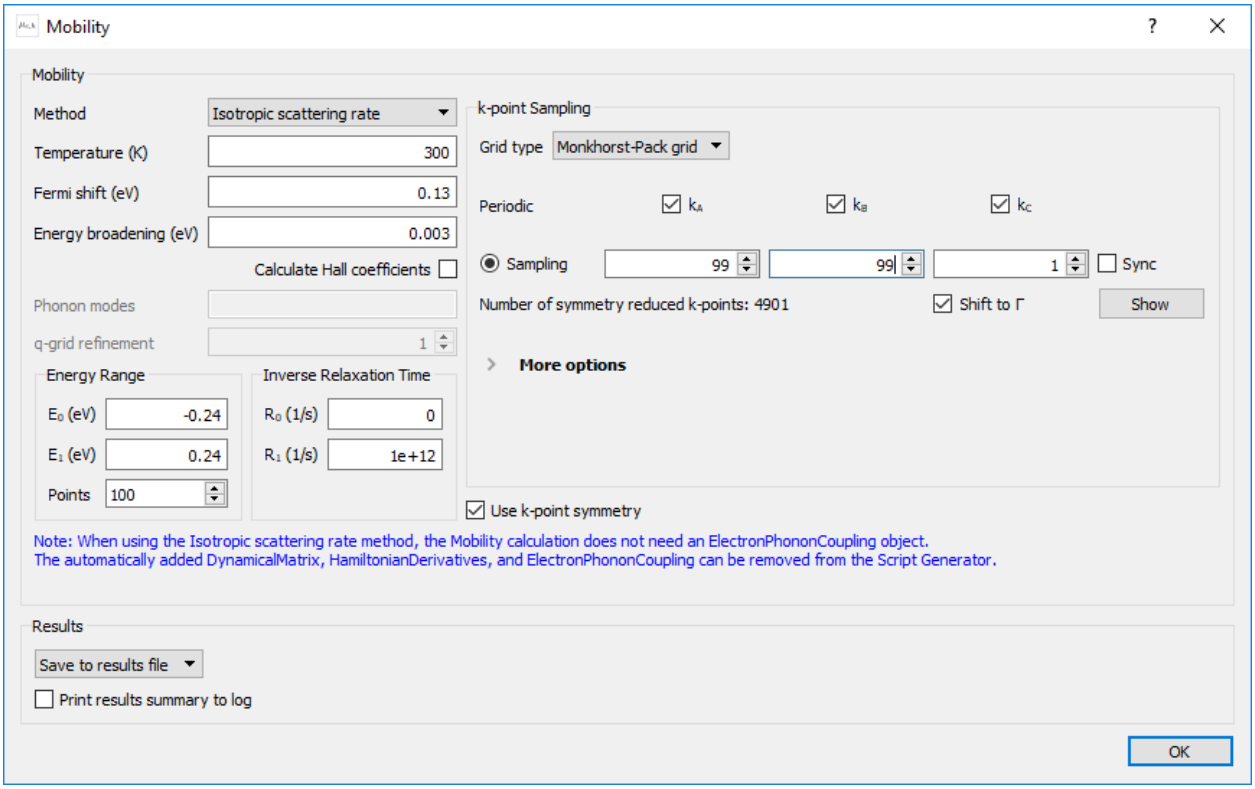
–Set the **k-point Sampling** to $99 \times 99 \times 1$

In the main panel of the **Script Generator**, set the **Results file** to `Graphene_mu-line-iso.hdf5`. Send the script to the **Editor**, change `mobility_object=None,` to `mobility_object=mobility_full,` and remove this line:

```
inverse_relaxation_time=numpy.linspace(0, 1e+12, 100)*Second**-1,
```

Send it to the **Job manager**, save it as `Graphene_mu-line-iso.py` and click on the button to run the calculation. This should take no more than a minute on a modern laptop.

Once the calculation is done, open the file `Graphene_mu-line-iso.hdf5`, select the $\mu_{e,h}$ **Mobility** object in the LabFloor and click on **Text Representation**.




```
+-------------------------------------------------------------------------------+
| Mobility Report                                                               |
| ----------------------------------------------------------------------------- |
| Input parameters:                                                             |
|     Temperature       = 300.00 K                                              |
|     Fermi level shift = 0.13 eV                                               |
|     Energy broadening = 0.0030 eV                                             |
|     q-grid refinement = 1                                                     |
|                                                                               |
+-------------------------------------------------------------------------------+
| Trace of linear responce tensors:                                             |
+-------------------------------------------------------------------------------+
|                                                                               |
| Electrons:                                                                    |
|                                                                               |
|     Mobility                  =  2.53e+05 cm^2/(V*s)                          |
|     Conductivity              =  1.73e+07 S/m                                 |
|     Seebeck coefficient       = -3.00e-05 V/K                                 |
|     Thermal conductivity      =  9.31e+01 W/(m*K)                             |
|     Carrier density (2D, xy) =  8.55e+11 cm^-2                                |
|                                                                               |
|                                                                               |
| Holes:                                                                        |
|                                                                               |
|     Mobility                  =  1.89e+06 cm^2/(V*s)                          |
|     Conductivity              =  5.72e+04 S/m                                 |
```

(continues on next page)

(continued from previous page)

```
|     Seebeck coefficient      =  1.14e-03 V/K                                  |
|     Thermal conductivity     =  5.74e+00 W/(m*K)                              |
|     Carrier density (2D, xy) =  3.78e+08 cm^-2                                |
|                                                                               |
+-------------------------------------------------------------------------------+
```

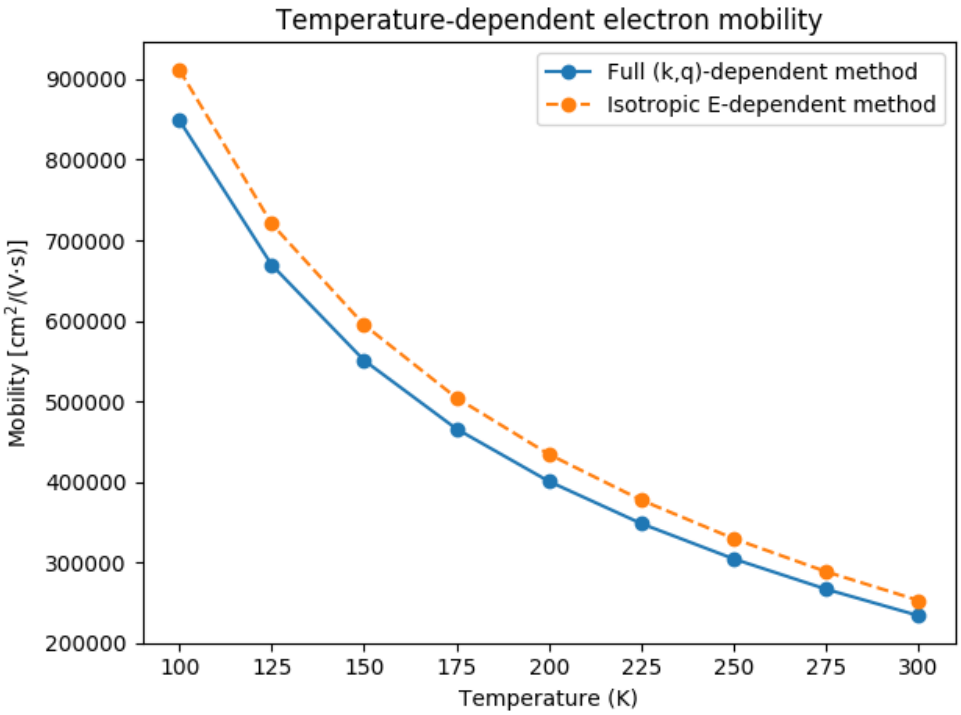
The calculated value for the electron mobility, highlighted in yellow above, is $2.53 \cdot 10^5$ $\mathrm{cm^2 V^{-1} s^{-1}}$, in good agreement with the value obtained by using the full $(\mathbf{k}, \mathbf{q})$ -dependent method.

Temperature dependence of the mobility: (k,q)-dependent method vs. energy-dependent method

A more stringent test for the reliability of the energy-dependent method is to calculate the temperature dependence of the mobility in an energy range up to room temperature.

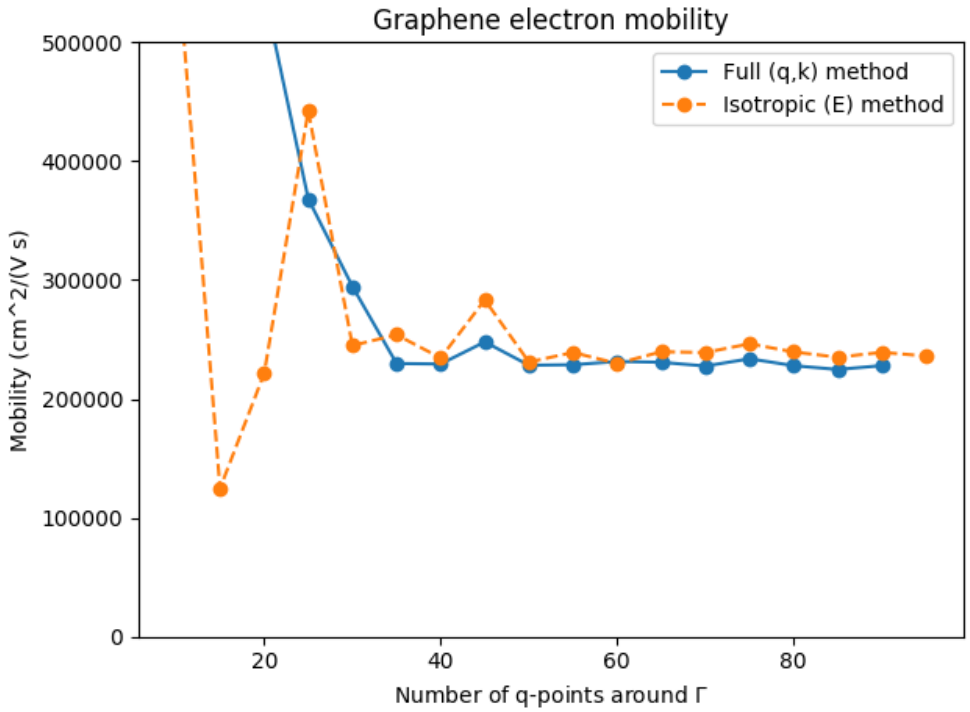
- In the **(k,q)-dependent method**, this can be done by simply modifying the value of the target temperature in the Mobility analysis object. `temperature_dependence_full_mobility.py`
- In the **energy-dependent method**, the two-step procedure must be repeated for each temperature, and the target temperature has to be set both when calculating the values of the $\mathbf{k}$- and $\mathbf{q}$-dependent relaxation times $\tau(\mathbf{k}, \mathbf{q})$ along the line, and when calculating the energy-dependent relaxation times $\tau(E)$. `temperature_dependence_isotropic_mobility.py`

As it can be seen from the figure below, where the T-dependency of μ is calculated in the temperature range $100\mathrm{K} \leq T \leq 300\mathrm{K}$, both methods reproduce well the expected $1/T$ behavior of μ. Script to create the plot: `plot_temperature_dependence_mobility.py`



Convergence of q- and k-point sampling

In order to select the appropriate samplings in $\mathbf{k}$- and $\mathbf{q}$-space, we first studied the mobility as a function of the number of $\mathbf{q}$-points, for a fixed sampling of $\mathbf{k}$-points:



We see that the mobility is reasonably converged for 50 $\mathbf{q}$-points and above. This corresponds to a density of at least 200 . We then study the mobility as a function of $\mathbf{k}$-points, and see that it converges very quickly in this case. We chose to use 20 k-points, corresponding to a density of about 160 .

Warning: This convergence study is only valid for this combination of system and computational settings. You should always study convergence for your system of interest and chosen computational model.

Theory section

The mobility μ is related to the conductivity σ via

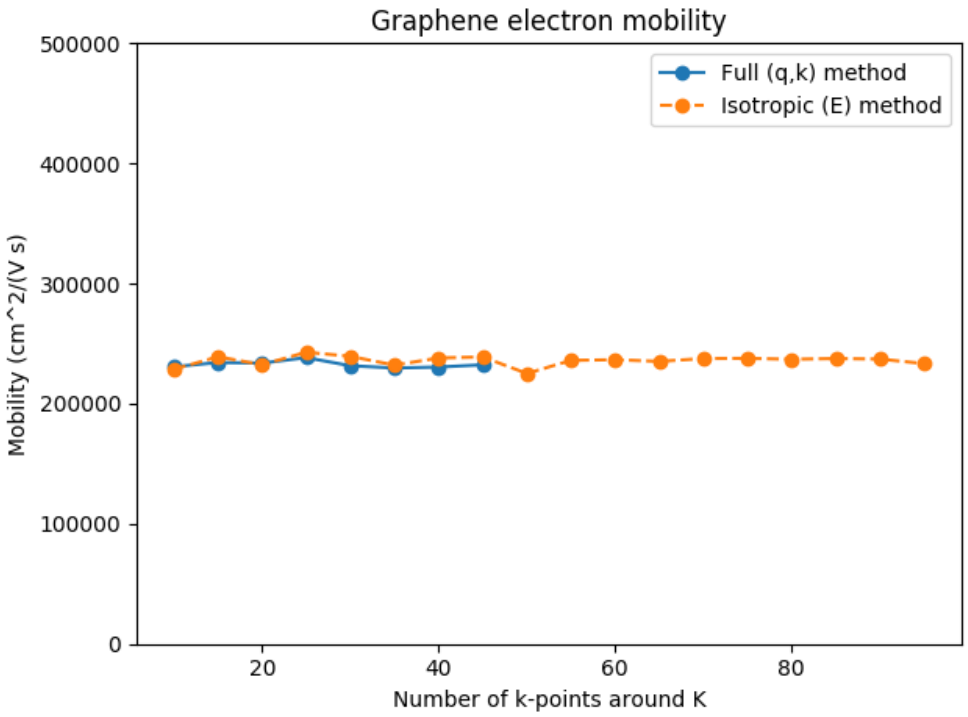
$$\mu = \frac{\sigma}{ne}, \tag{3.8}$$

where n is the carrier density, e is the electronic charge.

In this tutorial, you will calculate the conductivity using the semi-classical Boltzmann transport equation (BTE). In the case of zero magnetic field and a time-independent electric fields in the steady state limit the BTE simplifies to:

$$\frac{q\mathbf{E}}{\hbar} \cdot \nabla_{\mathbf{k}} f_{\mathbf{k}n} \approx \frac{q\mathbf{E}}{\hbar} \cdot \nabla_{\mathbf{k}} f^0_{\mathbf{k}n} = \left.\frac{\partial f_{\mathbf{k}n}}{\partial t}\right|_{coll} \tag{3.9}$$

The right hand side (RHS), often called the collision integral, describes different sources of scattering and dissipation that drives the system towards steady state. The left hand-side is approximated to linear order in the electric field by changing to the equilibrium distribution.



The electron-phonon scattering is described by the collision integral. Most commonly this is approximated by a relaxation time:

$$\left.\frac{\partial f_{\mathbf{k}n}}{\partial t}\right|_{coll} = \frac{\delta f_{\mathbf{k}n}}{\tau_{\mathbf{k}n}}$$

describing quasielastic scattering on acoustic phonons (relaxation time approximation). We let λ label the phonon modes, n the electronic bands, $\mathbf{k}$ the electron momentum. The transition rate from a state $|\mathbf{k}n\rangle$ to $|\mathbf{k}'n'\rangle$ is obtained from Fermis golden rule (FGR):

$$P^{\lambda nn'}_{\mathbf{k}\mathbf{k}'} = \frac{2\pi}{\hbar}|g^{\lambda nn'}_{\mathbf{k}\mathbf{k}'}|^2[n^{\lambda}_{\mathbf{q}}\delta\left(\epsilon_{\mathbf{k}'n'} - \epsilon_{\mathbf{k}n} - \hbar\omega_{q\lambda}\right)\delta_{\mathbf{k}',\mathbf{k}+\mathbf{q}}$$

$$+ (n^{\lambda}_{-\mathbf{q}} + 1)\delta\left(\epsilon_{\mathbf{k}'n'} - \epsilon_{\mathbf{k}n} + \hbar\omega_{-q\lambda}\right)\delta_{\mathbf{k}',\mathbf{k}-\mathbf{q}}],$$

where the first and last terms in the square brackets describes phonon absorption and emission, respectively. The electron-phonon coupling matrix elements $|g^{\lambda nn'}_{\mathbf{k}\mathbf{k}'}|$ are in **ATK** calculated using the **ElectronPhononCoupling** analysis module.

The general electron-phonon collision integral is given by

$$\left.\frac{\partial f_{\mathbf{k}n}}{\partial t}\right|_{coll} = -\sum_{\mathbf{k}'n'}\left[f_{\mathbf{k}n}\left(1 - f_{\mathbf{k}'n'}\right)P^{nn'}_{\mathbf{k}\mathbf{k}'} - f_{\mathbf{k}'n'}\left(1 - f_{\mathbf{k}n}\right)P^{n'n}_{\mathbf{k}'\mathbf{k}}\right] \tag{3.10}$$

Relaxation time approximation

In the relaxation-time approximation (RTA) one assumes a special simplified form of the RHS/collision integral:

$$\left.\frac{\partial f_{\mathbf{k}n}}{\partial t}\right|_{coll} = -\frac{\delta f_{\mathbf{k}n}}{\tau_{\mathbf{k}n}} \tag{3.11}$$

where $\delta f_{\mathbf{k}n} = f_{\mathbf{k}n} - f^0_{\mathbf{k}n}$. The linearized BTE becomes:

$$q\mathbf{E}\cdot\mathbf{v}_{\mathbf{k}n}\frac{\partial f^0_{\mathbf{k}n}}{\partial \epsilon_{\mathbf{k}n}} = -\frac{\delta f_{\mathbf{k}n}}{\tau_{\mathbf{k}n}}.$$

Hereby we find the solution:

$$\delta f_{\mathbf{k}n} = -q\mathbf{E}\cdot\mathbf{v}_{\mathbf{k}n}\frac{\partial f^0_{\mathbf{k}n}}{\partial \epsilon_{\mathbf{k}n}}\tau_{\mathbf{k}n}.$$

The relaxation-time can be evaluated from the general collision integral, (3.10). However, the specific form in eqn. (3.11) only applies in the limit of quasielastic scattering ($\omega_{q\lambda} \to 0$), in which case the collision integral, in eqn. (3.10), simplifies significantly:

$$\left.\frac{\partial f_{\mathbf{k}n}}{\partial t}\right|^{RTA}_{coll} = \sum_{\mathbf{k}'n'}\left(f_{\mathbf{k}'n'} - f_{\mathbf{k}n}\right)P^{nn'}_{\mathbf{k}\mathbf{k}'} \equiv -\frac{\delta f_{\mathbf{k}n}}{\tau_{\mathbf{k}n}},$$

since $P^{nn'}_{\mathbf{k}\mathbf{k}'} = P^{n'n}_{\mathbf{k}'\mathbf{k}}$ in this limit.

We then obtain the expression for the scattering rate or inverse relaxation-time:

$$\frac{1}{\tau_{\mathbf{k}n}} = \sum_{\mathbf{k}'n'}\left(1 - \frac{\delta f_{\mathbf{k}'n'}}{\delta f_{\mathbf{k}n}}\right)P^{nn'}_{\mathbf{k}\mathbf{k}'} \tag{3.12}$$

where we applied the detailed balance equation in this limit, $P^{nn'}_{\mathbf{k}\mathbf{k}'}(f^0_{\mathbf{k}'n'} - f^0_{\mathbf{k}n}) = 0$, and assumed isotropic scattering.

To evaluate the relaxation time we introduce the scattering angle

$$cos(\theta_{\mathbf{k}\mathbf{k}'}) = \frac{\mathbf{v}_{\mathbf{k}'n'}\cdot\mathbf{v}_{\mathbf{k}n}}{|\mathbf{v}_{\mathbf{k}'n'}||\mathbf{v}_{\mathbf{k}n}|}$$

where $\mathbf{v}_{\mathbf{k}n}$ is the electron velocity. Following[Page 434, 1] we obtain

$$\frac{1}{\tau_{\mathbf{k}n}} = \sum_{\mathbf{k}'n'}\frac{(1 - f^0_{\mathbf{k}'n'})}{(1 - f^0_{\mathbf{k}n})}\left(1 - cos(\theta_{\mathbf{k}\mathbf{k}'})\right)P^{nn'}_{\mathbf{k}\mathbf{k}'}$$

Mobility

Once knowing the relaxation times one obtain the mobility as:

$$\mu = -2q\frac{\sum_{\mathbf{k}n} v^2_{\mathbf{k}n}\frac{\partial f^0_{\mathbf{k}n}}{\partial \epsilon_{\mathbf{k}n}}\tau_{\mathbf{k}n}}{\sum_{\mathbf{k}n} f^0_{\mathbf{k}n}} \tag{3.13}$$

where the factor 2 accounts for spin degeneracy.

References

3.2.2 Effective mass of electrons in silicon

In this tutorial, you will learn how to compute the electron effective mass for silicon.

Introduction

It is well known that DFT methods, or to be more specific the LDA and GGA exchange-correlation functionals, are not very adept at predicing the band gap of semiconductors. They do, however, in many cases give rather accurate curvatures of the bands. We can use this to compute the effective mass of holes and electrons by fitting a parabola to the minimum/maximum of the conduction/valence bands.

The effective mass is a parameter that depends on several assumptions and definitions, so it will be important to properly understand the underlying concept, instead of just using it as a black box. For other materials, these assumptions may be different!

Background

The lowest conduction band of Si has 6 equivalent minima, located along (101) (called Δ) and its permuted equivalent directions in reciprocal space. The minimum is located at (x,0,x) where x is about 0.425, or 85% of the distance to the first Brillouin zone boundary at X=(1/2,0,1/2). At this point, the energy isosurfaces are ellipsoids, thus you get different values for the effective mass depending on if you look in the longitudinal (along) or transverse (perpendicular to Δ) directions. See Refs.[1] and[2] for more details.

By (1/2,0,1/2) we mean a k-vector $K = (\mathbf{G}_A + \mathbf{G}_C)/2$, where $\mathbf{G}_A$, $\mathbf{G}_B$, $\mathbf{G}_C$ are the three unit vectors that span the reciprocal unit cell. If we express this point in Cartesian coordinates (still in reciprocal space) we find that the Δ direction is parallel to $\mathbf{K}_Y$. Obviously then, $\mathbf{K}_X$ and $\mathbf{K}_Z$, which are parallel to $\mathbf{G}_B + \mathbf{G}_C$ and $\mathbf{G}_A + \mathbf{G}_B$, respectively, are perpendicular to $\mathbf{K}_Y$. Thus the transverse directions are (011) and (110), and the longitudinal one is (101) - see the figure below illustrating the Brillouin zone of silicon.

What we now need to do is:

1. generate a sequence of points k around a specified point k_0, in a given direction;
2. compute the energy eigenvalues $E(k)$ for the lowest conduction band for these k-points;
3. perform a numerical (finite difference) second order derivative of the band, $d = \partial^2 E(k)/\partial k|_{k=k_0}$;
4. compute the effective mass as $m^* = \hbar^2/(2d)\, m_e$ where m_e is the free electron mass.

If we do this for the longtitudinal (L) and transverse (T) directions, we will obtain three values, m^*_L, m^*_{T1}, and m^*_{T2} (actually, for Si $m^*_{T1} = m^*_{T2} = m^*_T$), which we can insert in the well-known expressions for the conductivity and density of states effective masses:

$$m^*_C = 3/(1/m^*_L + 2/m^*_T)$$
$$m^*_{DOS} = 6^{2/3}(m^*_L (m^*_T)^2)^{1/3}$$

where the factor 6 which enters the expression for the conductivity mass is the number of equivalent Δ valleys.

[1] http://nanohub.org/resources/1522/download/simplifiedbandstructurecarrierdynamics_word.pdf
[2] http://www.iue.tuwien.ac.at/phd/dhar/node18.html

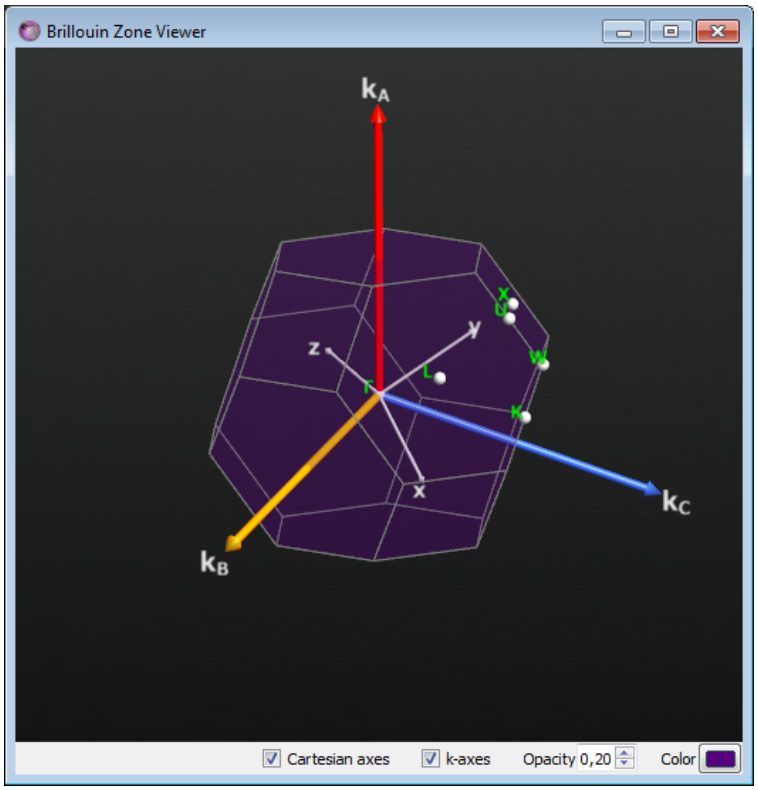


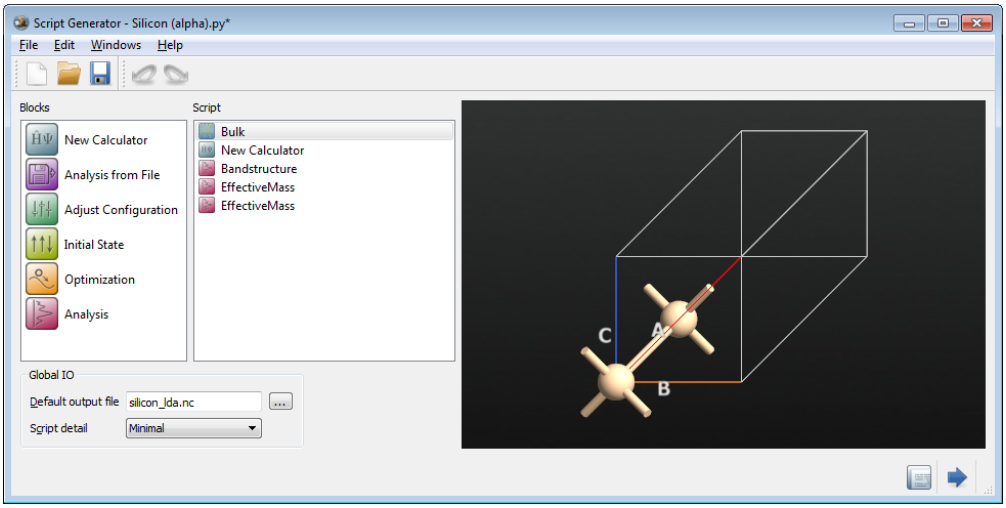
Fig. 3.19: Brillouin zone of silicon. In the **Builder** use *Bulk Tools* → *Brillouin Zone Viewer* to visualize the Brillouin zone, fractional- and Cartesian directions and the high-symmetry points.

Set up the calculation

The first step is to do a self-consistent DFT calculation for Si.

The details involved in such a task are described in the *Calculate the band structure of a crystal*. There, the extended Hückel method was used, so you should modify the calculator to use **ATK-DFT** instead. Note that by default the LDA exchange correlation functional is used and you just need to specify a proper k-point mesh, i.e. 15x15x15.

Before saving and running your DFT calculation, you can set up the effective mass analysis directly from the Scripter. Add *Analysis* → *EffectiveMass* analysis after the calculator:

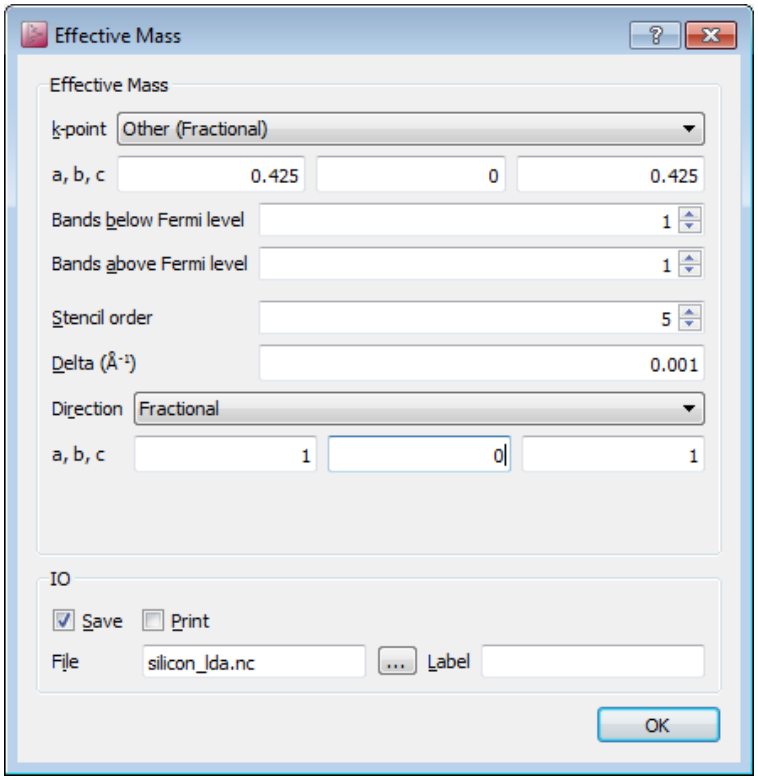


Double-click on EffectiveMass to set up the effective mass calculation. Here, you have several options and you can find all the details for each of these options in the `EffectiveMass`.

In order to calculate the effective mass at the conduction mass minimum enter the following parameters:

- Change the k-point to Other (Fractional) and use the [0.425, 0, 0.425] coordinates.
- Select one band above the Fermi level (conduction band only) and one band below the Fermi level. However, feel free to select as many bands as you wish!
- As you will see later they will be conveniently listed in the LabFloor. Only remember that the effective masses calculated for all these bands correspond to the specified [0.425, 0, 0.425] point.
- Set the Direction to **Fractional** and set a=1, b=0, c=1, corresponding to the longitudinal direction, c.f. *Background*.
- Leave all the other default parameters.
- Add a second EffectiveMass analysis object with the same settings, except the direction which should be a=1, b=1, c=0, corresponding to one of the transverse directions.

Finally, set the output filename, `silicon_lda.hdf5` for example, save and run the script. The job will take just a few seconds.



Analyze the results

Bandstructure

You will now find the Bandstructure and EffectiveMass objects in the LabFloor. Lets first check the bandstructure: double-click on it and use the Bandstructure Analyzer tool to plot it.

Inspecting the bandstructure its clear that the band gap is severly underestimated, but the shape is essentially correct; the gap is indirect and we do have a conduction band minimum at about 85% of the distance to X.

Effective mass 1

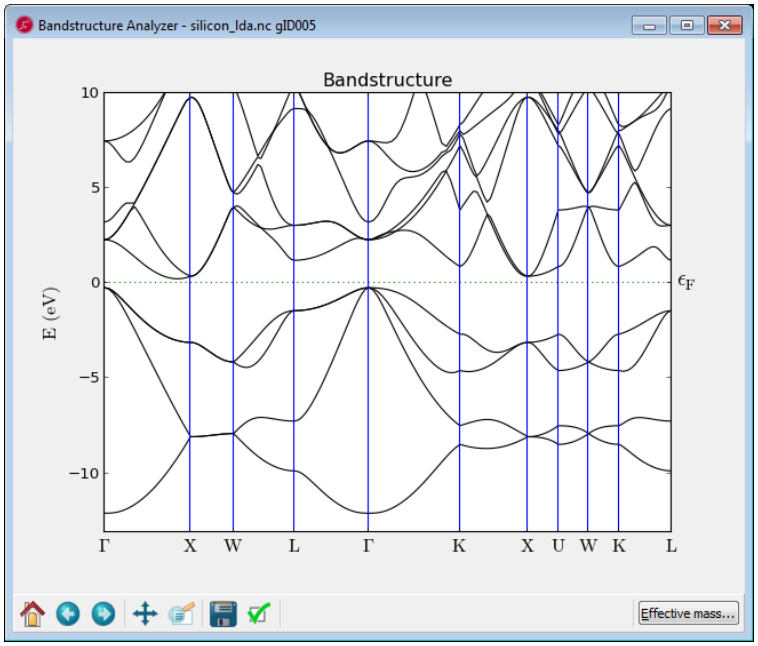
Select one of the **EffectiveMass** object from the LabFoor and use the **Effective mass analyzer** to inspect it:

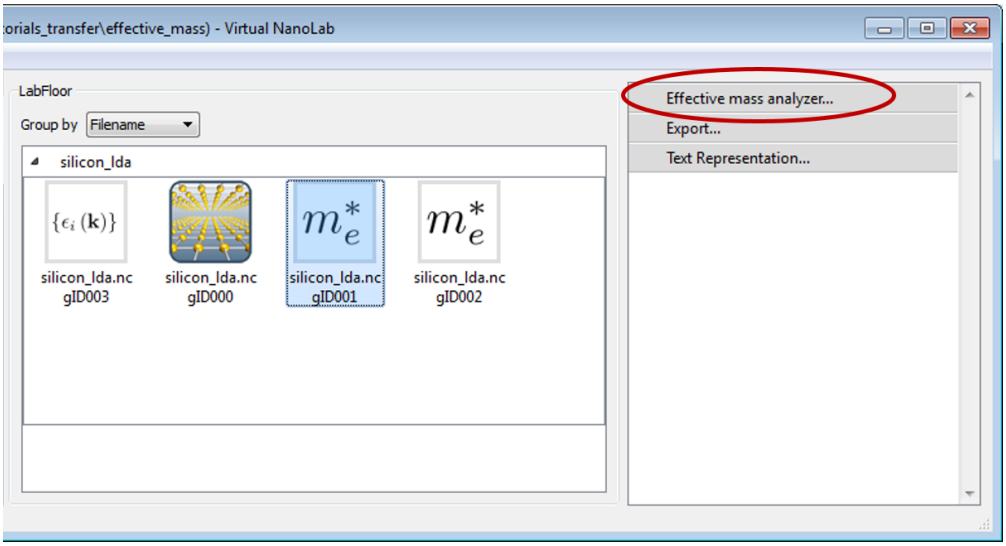
The **Effective mass analyzer** window will then open:

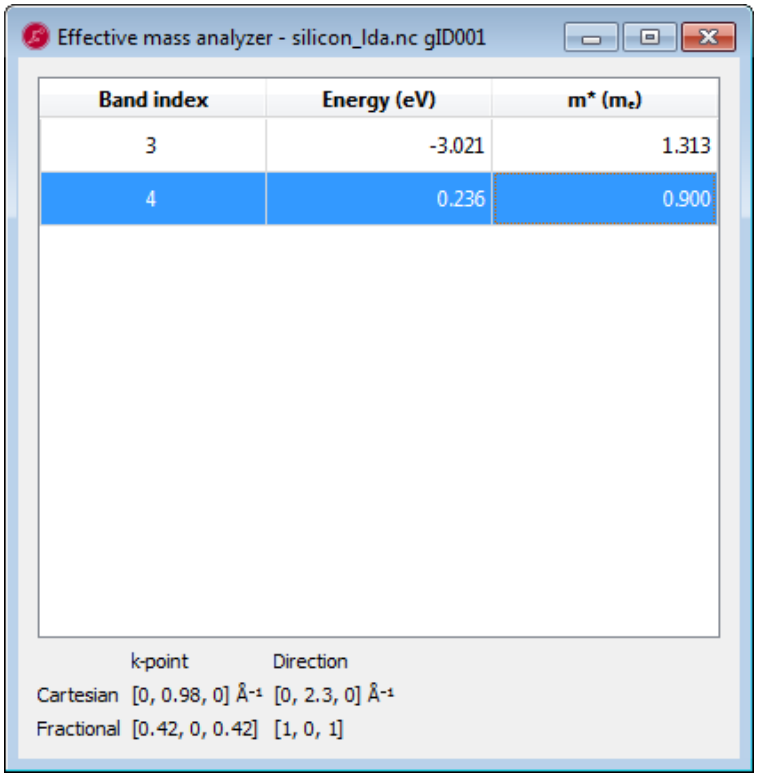
Here, the effective mass corresponding to the specified band and direction is reported. From the two **EffectiveMass** objects you will obtain:

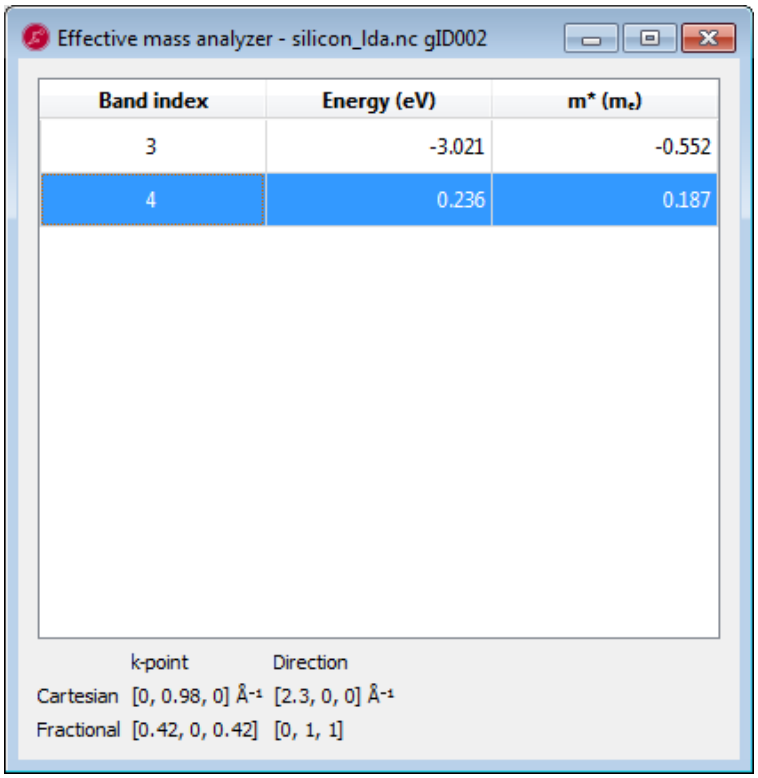
- Longitudinal effective mass, [1,0,1] direction, $m^* = 0.90\, m_e$.
- Transverse effective mass, [1,1,0] direction, $m^* = 0.187\, m_e$.

With these values we can now compute the density of states mass $m^*_{DOS} = 1.05$ and $m^*_C = 0.26$, very close indeed to the reference values 1.08 and 0.26 (see e.g. http://ecee.colorado.edu/~bart/book/effmass.htm). Thus DFT actually does reproduce the effective mass of the electrons in the Δ valley very well.


Bandstructure Analyzer - silicon_lda.nc gID005
Bandstructure
10
5
0
−5
−10
E (eV)
ϵ_F
Γ X W L Γ K X U W K L
Effective mass...


orials_transfer\effective_mass) - Virtual NanoLab
LabFloor
Group by Filename
silicon_lda
$\{\epsilon_i(\mathbf{k})\}$
silicon_lda.nc gID003
silicon_lda.nc gID000
m_e^*
silicon_lda.nc gID001
m_e^*
silicon_lda.nc gID002
Effective mass analyzer...
Export...
Text Representation...


Effective mass analyzer - silicon_lda.nc gID001
Band index
Energy (eV)
m* (m$_e$)
3
-3.021
1.313
4
0.236
0.900
k-point
Direction
Cartesian
[0, 0.98, 0] Å^{-1}
[0, 2.3, 0] Å^{-1}
Fractional
[0.42, 0, 0.42]
[1, 0, 1]

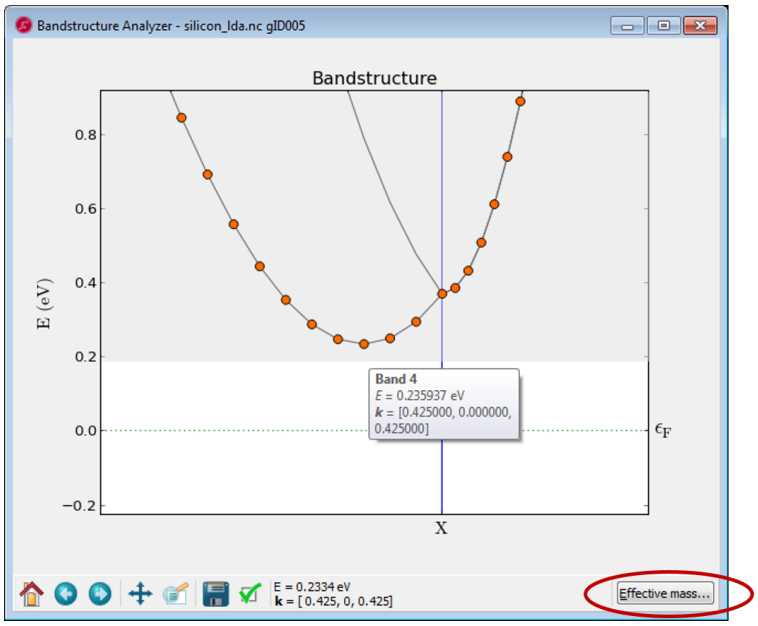

Effective mass analyzer - silicon_lda.nc gID002
Band index
Energy (eV)
m* (m$_e$)
3
-3.021
-0.552
4
0.236
0.187
k-point
Direction
Cartesian
[0, 0.98, 0] Å^{-1}
[2.3, 0, 0] Å^{-1}
Fractional
[0.42, 0, 0.42]
[0, 1, 1]

Effective mass 2

You may have noticed at the bottom right corner of the **Bandstructure analyzer** window the **Effective mass** button. You do have indeed a second interactive tool at your disposal to calculate the effective masses.

This is a very convenient tool since you can select the k-point for which the effective mass will be calculated directly from the plot:

•zoom in in the desired region of the plot and left-click to the desired point



•Click on Effective mass to open the effective mass tool:

Here, all the relevant parameters such as bandindex and k-point coordinates corresponding to the clicked point will be automatically set. Note that you can change and refine these parameters.

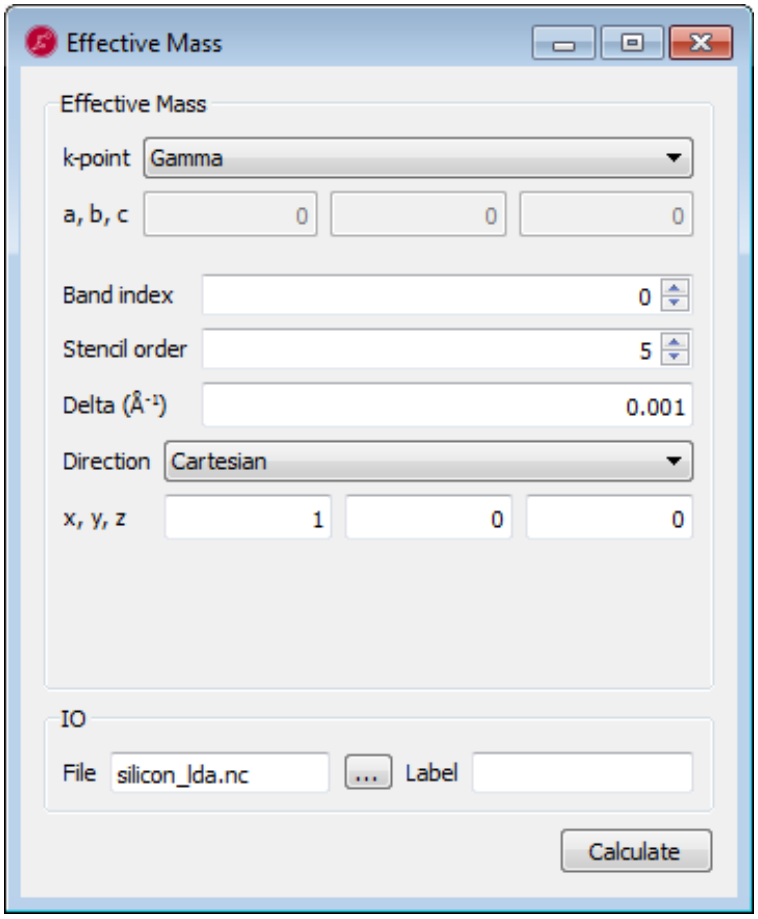
•Press Calculate to calculate the effective mass.

•Check on the LabFloor where the Effective Mass object is stored and follow the steps above to get the effective mass.

The effective mass tools contain a couple of additional parameters which we have not mentioned.

•Stencil order: it is possible to select the number of k-points that will be used to generate the band structure for which we perform the numerical derivative.

•Delta: this parameter controls how far away from the chosen k-point we go. The default is typically fine; if the band structure is strongly non-parabolic, the value can be decreased a bit perhaps.


Effective Mass
Effective Mass
k-point
Gamma
a, b, c
0
0
0
Band index
0
Stencil order
5
Delta (Å⁻¹)
0.001
Direction
Cartesian
x, y, z
1
0
0
IO
File
silicon_lda.nc
...
Label
Calculate

Going further

You can now also try the same calculation with the extended Hückel method.

If you followed the *Calculate the band structure of a crystal* you already have the necessary HDF5 file. The band minimum is not at x=0.452 this time, but closer to x=0.445. The results are very similar, however ($m_L^* = 1.00$ and $m_T^* = 0.17$).

Hole masses in Si

For Si, we compute a light hole mass (at the point [0,0,0]) $m_{LH}^* = -0.16$ (band index=1), and for the two degenerate heavy hole bands the result is $m_{HH}^* = -0.25$ (band indices 2 and 3, respectively). These values are obtained in the directions of the Cartesian axes (i.e. if we use the same directions as for the electron, along Δ from Γ to X). If, however, we look in the Λ direction from Γ to L (by setting the direction in the Analyzer to [1,1,1]), we find $m_{HH}^* = -0.64$ and $m_{LH}^* = -0.09$, a result of fact that the constant-energy surfaces for holes are strongly warped in Si, especially for the heavy hole bands[Page 457, 1Page 457, 2].

To get proper hole masses that can be compared to experiments one should however also include spin-orbit interaction.

References

3.2.3 Spin-orbit splitting of semiconductor band structures

The spin-orbit coupling (SOC) is a relativistic effect that causes splitting of the electronic bands in many materials, including semiconductors. This effect is not included in standard calculations.

In this tutorial, you will learn how to take spin-orbit coupling into account in density functional theory calculations (ATK-DFT) and also in semi-empirical Slater-Koster calculations (ATK-SE) for semiconductors.

Outline

- We start off with a short introduction to relativistic effetcs in electronic structure theory, including spin-orbit coupling.
- You will then use ATK-DFT to study how the SOC splits the electronic bands of Silicon around the Γ-point, leading to the so-called split-off valence band and bands with heavy and light holes.
- You will then employ a SO+MGGA method to get not only the correct band splits but also a reliable estimate of the indirect band gap.
- Lastly, you will consider the gallium-arsenide crystal, and use the Slater–Koster method with SOC parameters to compute the GaAs band structure.

Relavistic effects in Kohn-Sham DFT

The standard time-independent Kohn-Sham Hamiltonian describes *non-relativistic* electrons moving in an external field set up by atomic nuclei (and possibly any other external time-independent fields). Relavistic effects are therefore completely disregarded. This is usually a very good approximation for valence electrons outside the atomic cores, but for heavy elements, like gold and lead, relativistic contributions to the electronic structure can be crucial. Moreover, **spin-orbit coupling**, which cannot be captured in a strictly non-relativistic description, will often break degeneracy in solid band dispersions, and thereby lead to experimentally observed band splittings.

Electronic-structure software employing pseudopotentials, like the ATK-DFT engine does, will usually incorporate relativistic effects on core electrons in an approximate manner by using scalar-relativistic pseudopotentials. This is a computationally efficient and quite reliable approach.

However, calculations including spin-orbit coupling require fully relativistic pseudopotentials and a **noncollinear** representation of the electronic spin degrees of freedom. This can be computationally intense, but is mandatory if relativistic effects on the electronic ground state are to be taken fully into account.

Silicon band splitting with ATK-DFT

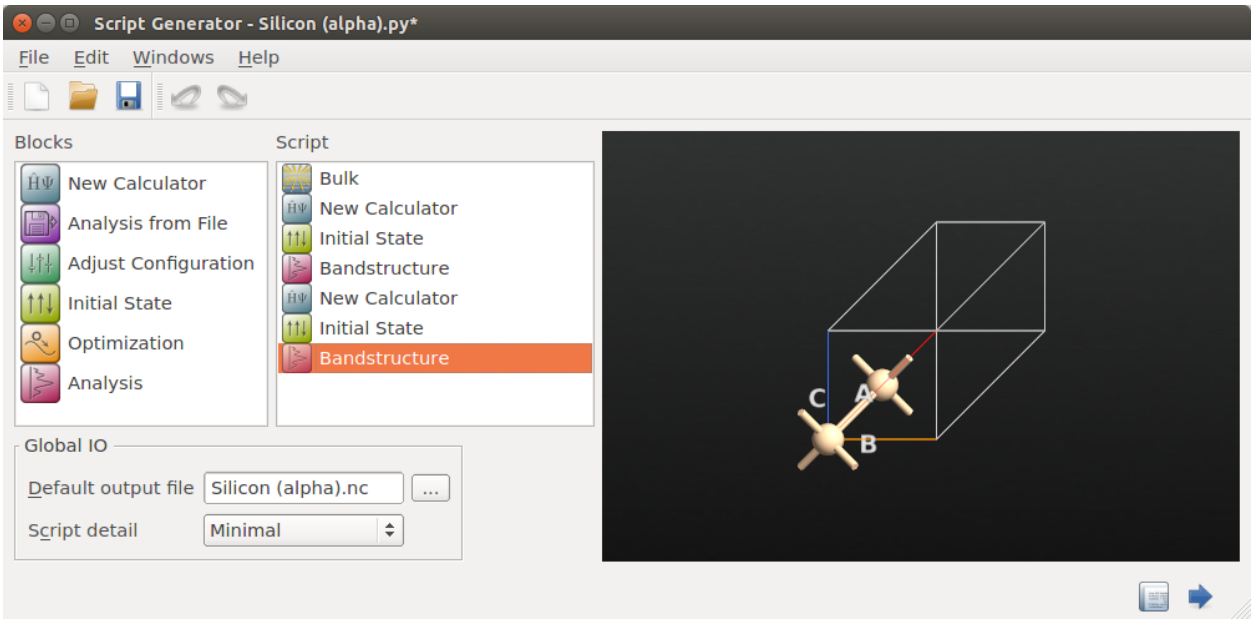
You will now use ATK-DFT to calculate the LDA bandstructure of silicon with and without spin-orbit coupling.

Note: Spin-orbit DFT calculations are more expensive than standard LDA or GGA methods and will often require more SCF iterations to converge. Usually, it is therefore more efficient to first do a spin-polarized ground state calculation, and then use this state as an initial guess for the spin-orbit DFT calculation. This procedure reduces the number of spin-orbit SCF steps and significantly reduces the overall computational time.

Warning: Not all pseudopotentials can be used for spin-orbit DFT calculations. The OMX potentials supplied with QuantumATK contain the required SO terms, while the FHI potentials do not.

Hint: Please remember that OMX pseudopotentials have in general more valence electrons in the cores than the FHI potentials do. Therefore, the OMX potentials usually require good basis sets and higher density-mesh cutoff energies (e.g. 200 Hartree).

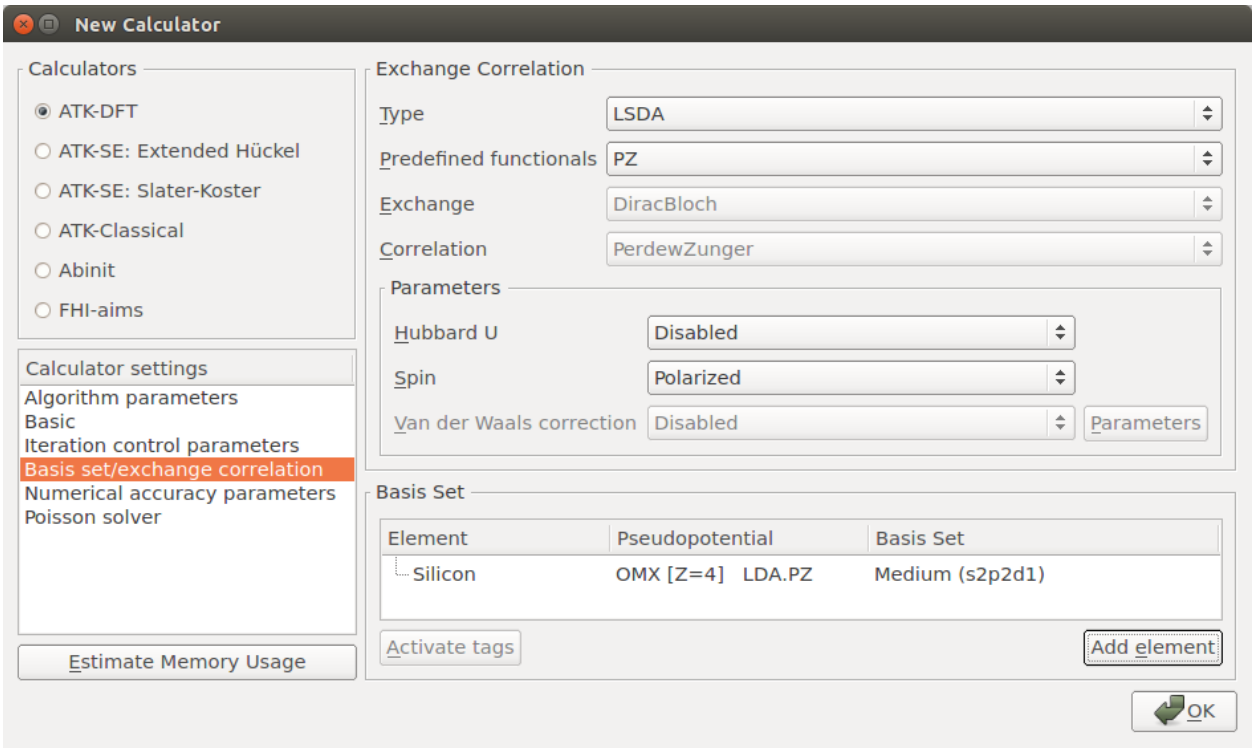
- Use the **Builder** to set up the silicon crystal with the default (experimental) lattice constant.
- Send the bulk configuration to the **Script Generator** and add two sets of the following three blocks: New Calculator, Initial State, and Bandstructure.



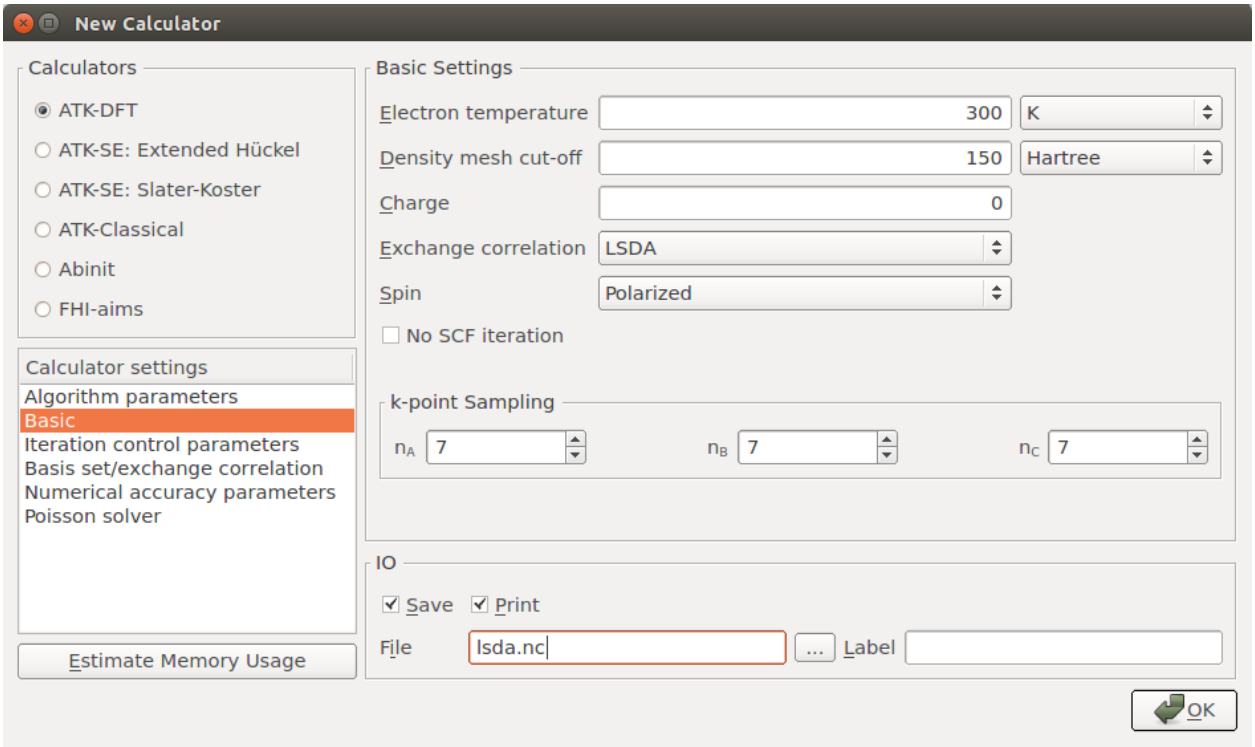
LSDA initial guess for the ground state

•Double-click the first New Calculator block, and make the following changes to it:

–Select the OMX pseudopotential with 4 valence electrons, and use the Medium accuracy basis set:



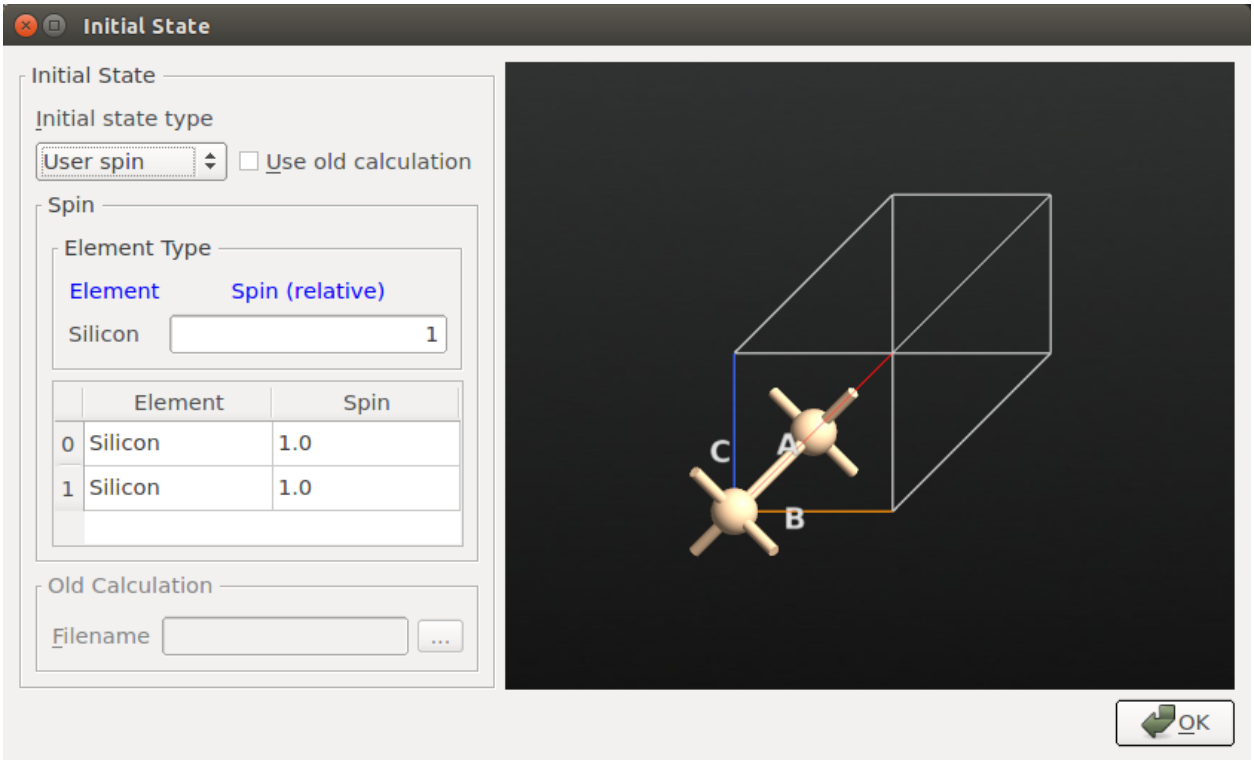
–In the Basic calculator settings, increase the density-mesh cutoff to 150 Hartree, select Polarized spin, set up a 7x7x7 k-point grid, and use `lsda.nc` as output file:



–Click OK to close the widget.

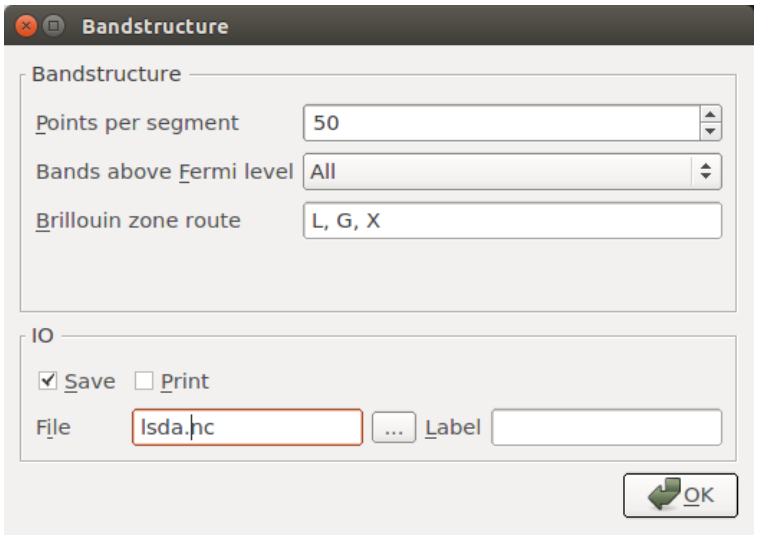
•Double-click the first Initial State block, and set up an initial guess for spin polarization of the silicon crystal:

–Change Initial state type to User spin. Then click OK.



Note: This defines a non-zero initial guess for the silicon polarization, even though we know that silicon is not magnetic. Indeed, we will find that it is not, because the self-consistent polarization will be close to zero. However, a self-consist LSDA calculation is usually a good starting point for noncollinear calculation, with or withour spin-orbit.

•Edit the first Bandstructure block, such that it maps out the $L-\Gamma-X$ path with 50 points per segment, and saves the data in the `lsda.nc` file:



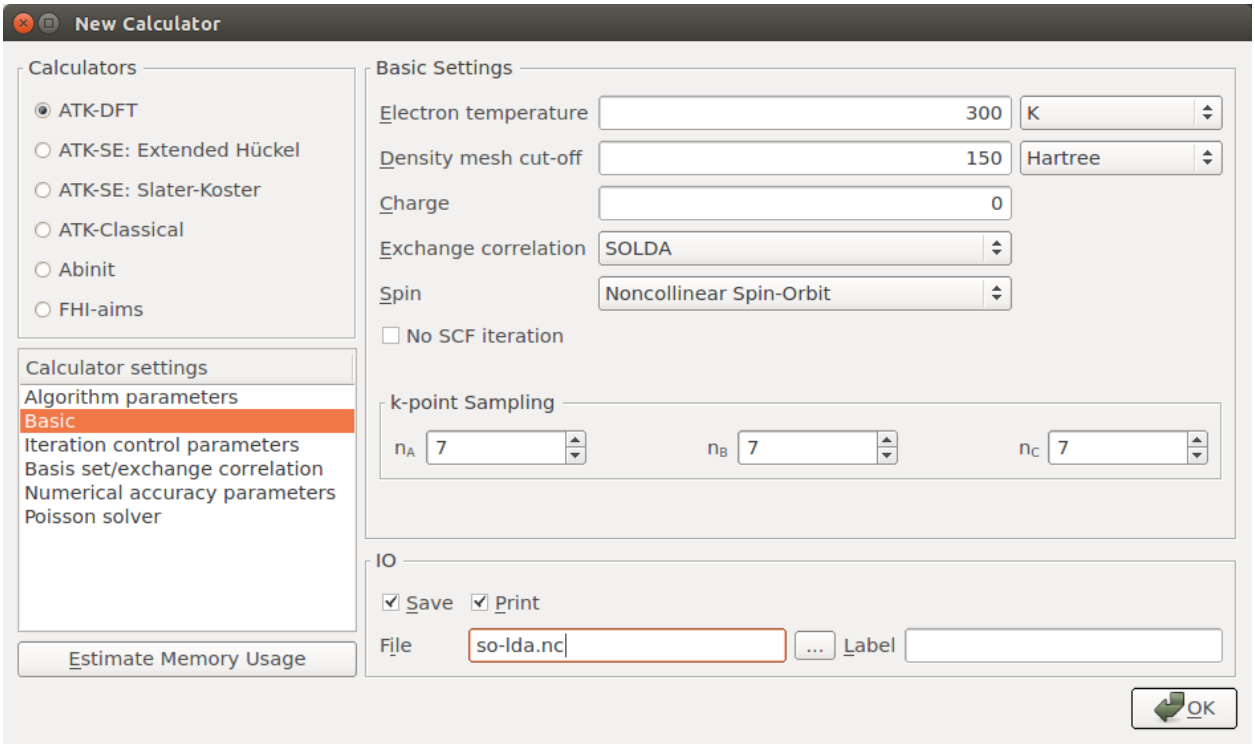
•That fully defines the LSDA ground state calculation followed by a band structure analysis.

SOLDA calculation

Next, you will set up a similar workflow for SOLDA calculations, but use the LSDA ground state as an initial guess for the SOLDA ground state.

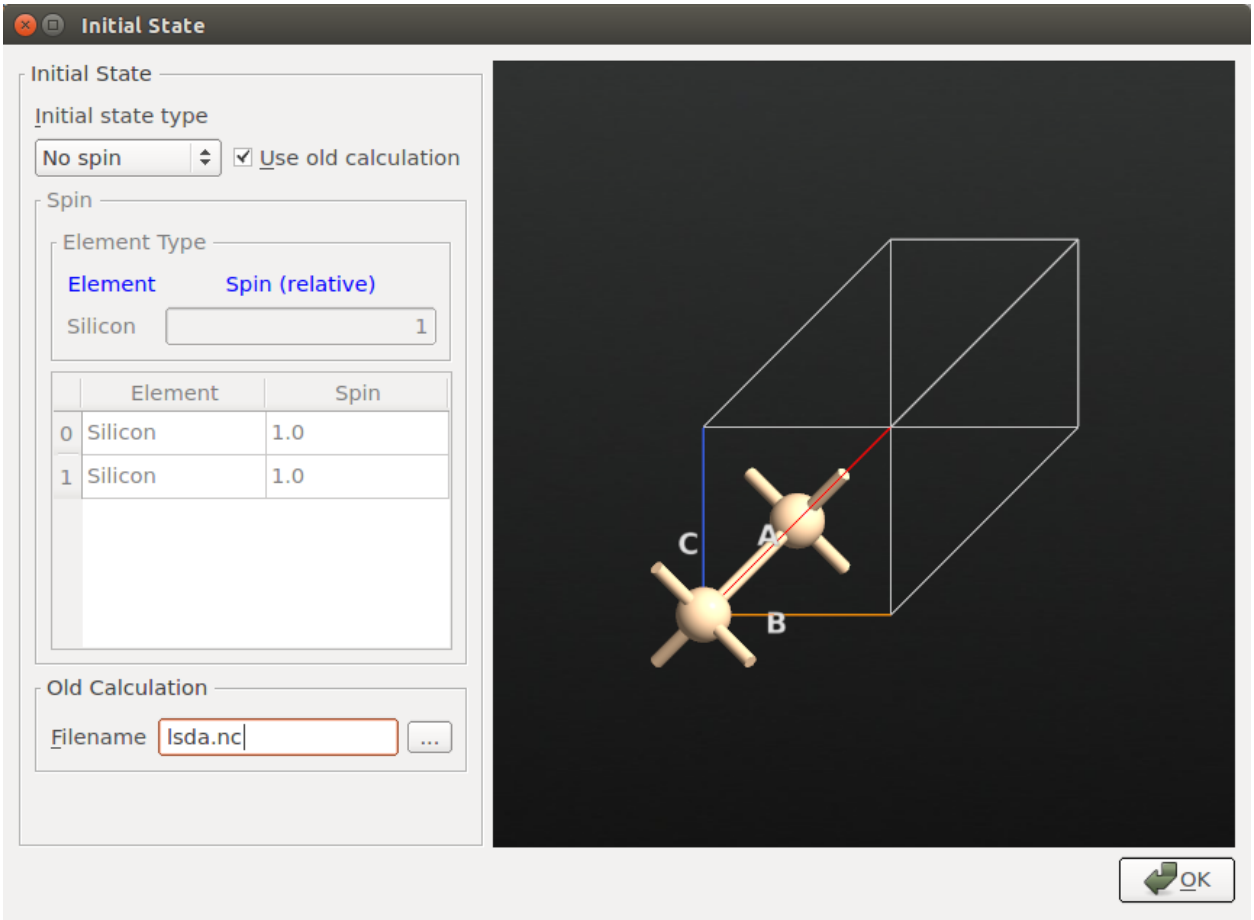
•Modify the second New Calculator block such that it is similar to the first one, but choose the Noncollinear Spin-Orbit spin option and save the results in `so-lda.nc`.

Hint: Remember to also choose the correct basis set and pseudopotential.

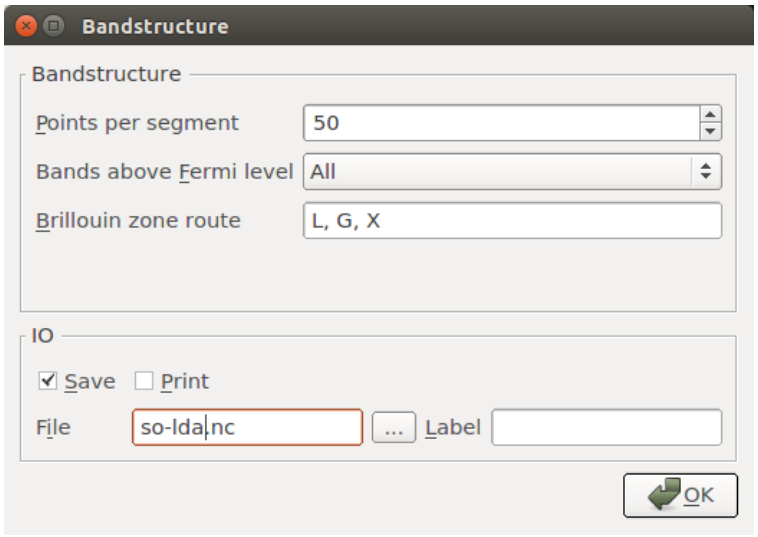


•In the second Initial State block, select Use old calculation and point to `lsda.nc` as the file from where to read the initial state:

Warning: If you use a non-polarized calculation as initial state for the spin-orbit calculation, you should set the initial spin to zero.

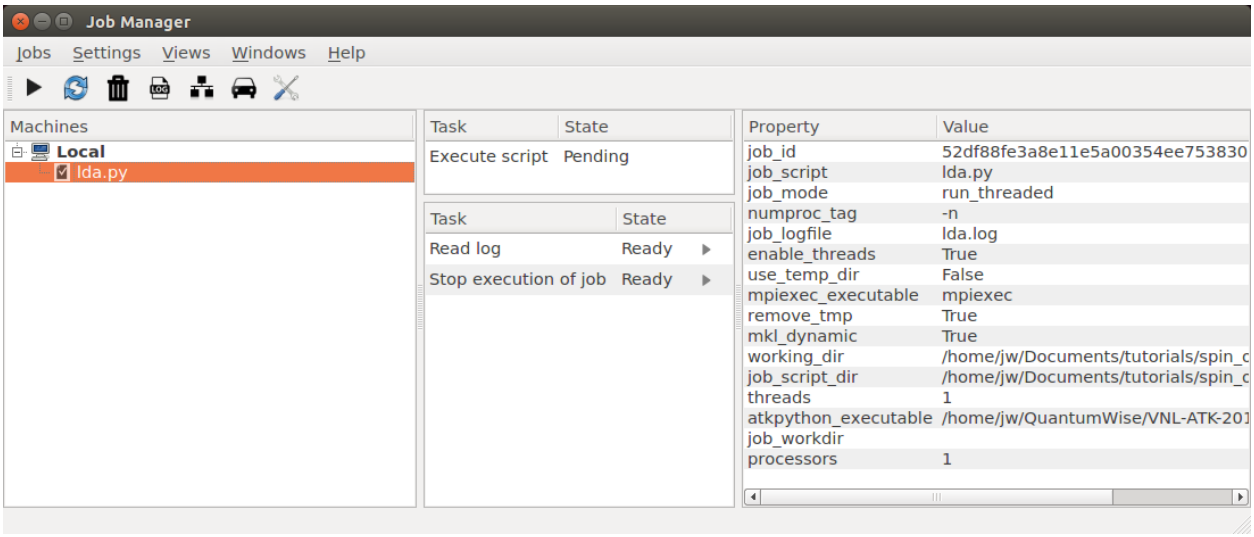


•Set up the last Bandstructure block similarly to the first one, and save the results in `so-lda.nc`:

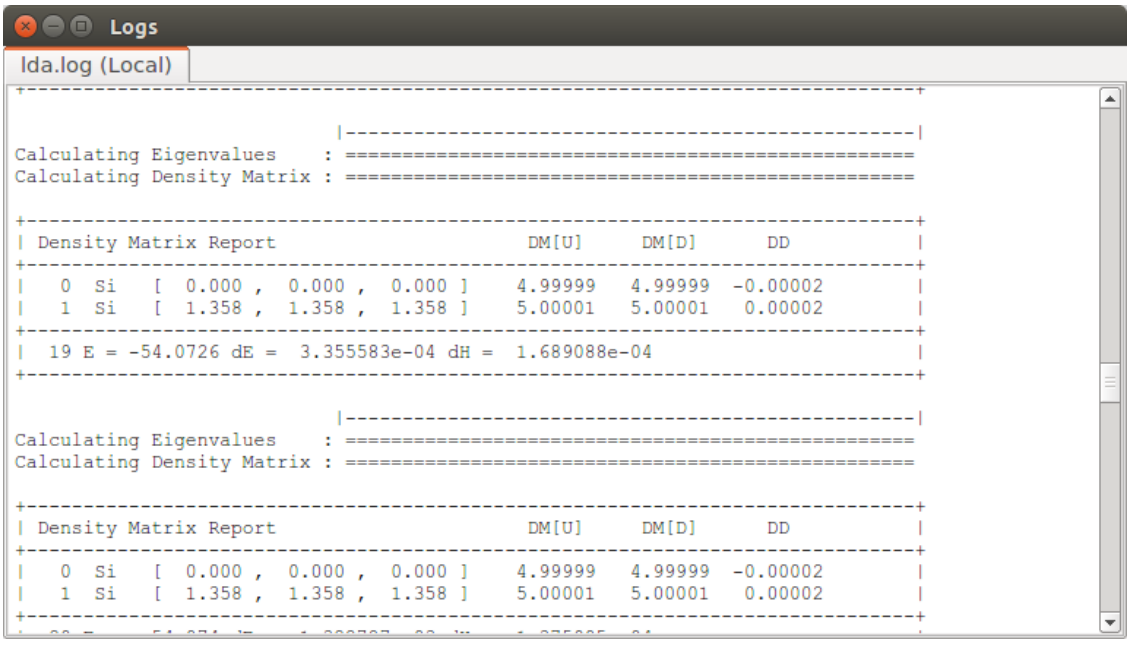


The ATK-DFT script is now done. It can also be downloaded from here: `lda.py`.

Send the script to the **Job Manager** , save it as `lda.py`, and run it in serial on your local machine. It should take only a few minutes to finish.

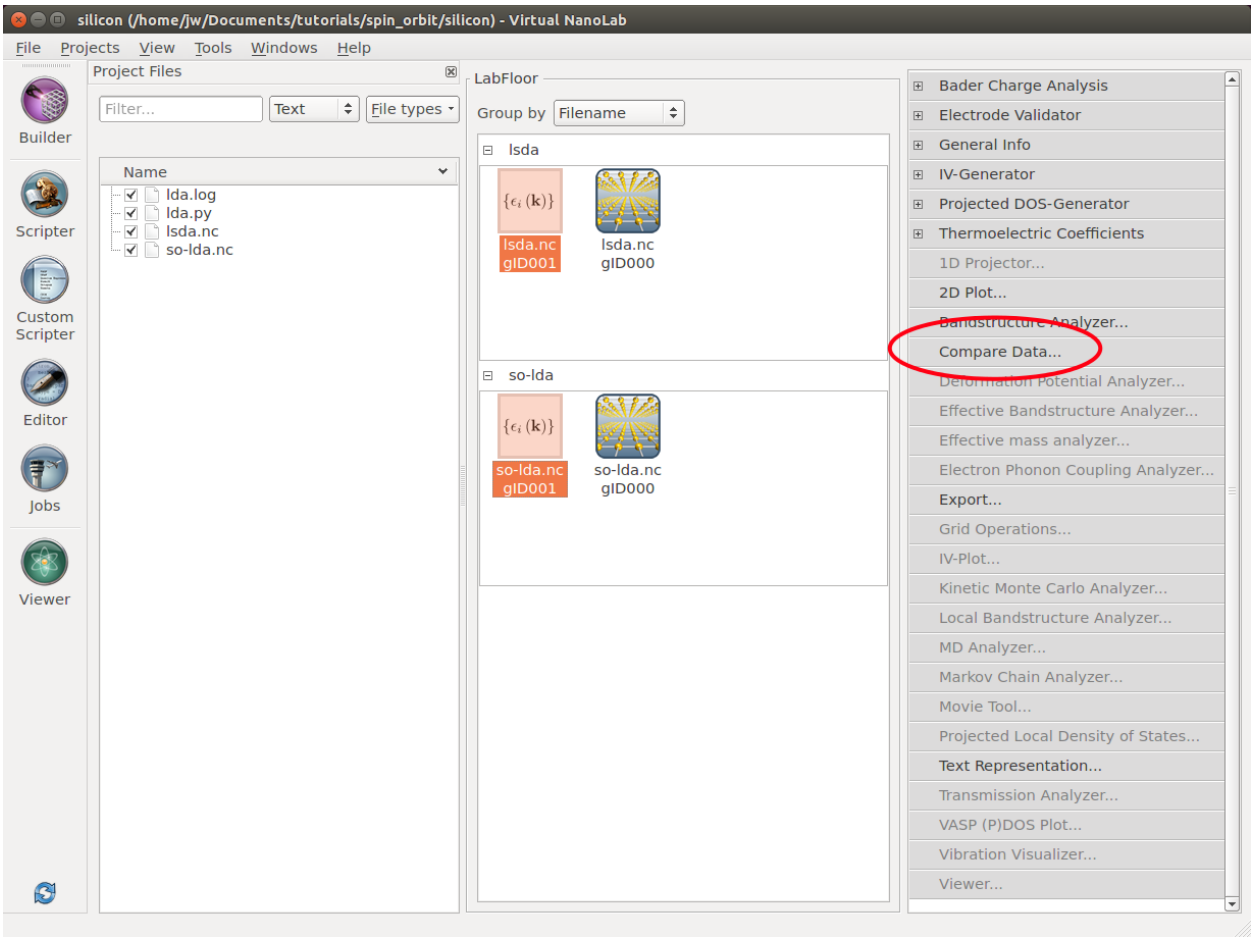


You should be able to monitor the progress of the job while it is running:



Results

Once the results have appeared on the **LabFloor**, highlight the two bandstructure objects and use the **Compare Data** plugin to directly visualize the LSDA and SO+LDA band structures in the same plot:



Several observations can be made:

- As expected, the LSDA spin-up and spin-down bands are identical (blue lines).
- At first sight, SO+LDA and LSDA bands appear indistinguishable. In particular, they both yield and indirect band gap of around 0.5 eV.
- However, if you zoom in around the highest valence bands at the Γ-point (see below), the SO coupling has partly lifted the degeneracy of those bands by splitting them:
  - The top band has split into two bands, corresponding to heavy and light holes.
  - The third band is now separated from the two others by the so-called split-off energy, experimentally measured to be 42.6 meV[1]. With SO+LDA we find 44 meV.

[1] Zhiyi Yu, YXăHuang, and SCăShen. Spin-orbit splitting of the valence bands in silicon determined by means of high-resolution photoconductive spectroscopy. *Physical Review B*, 39(9):6287, 1989.

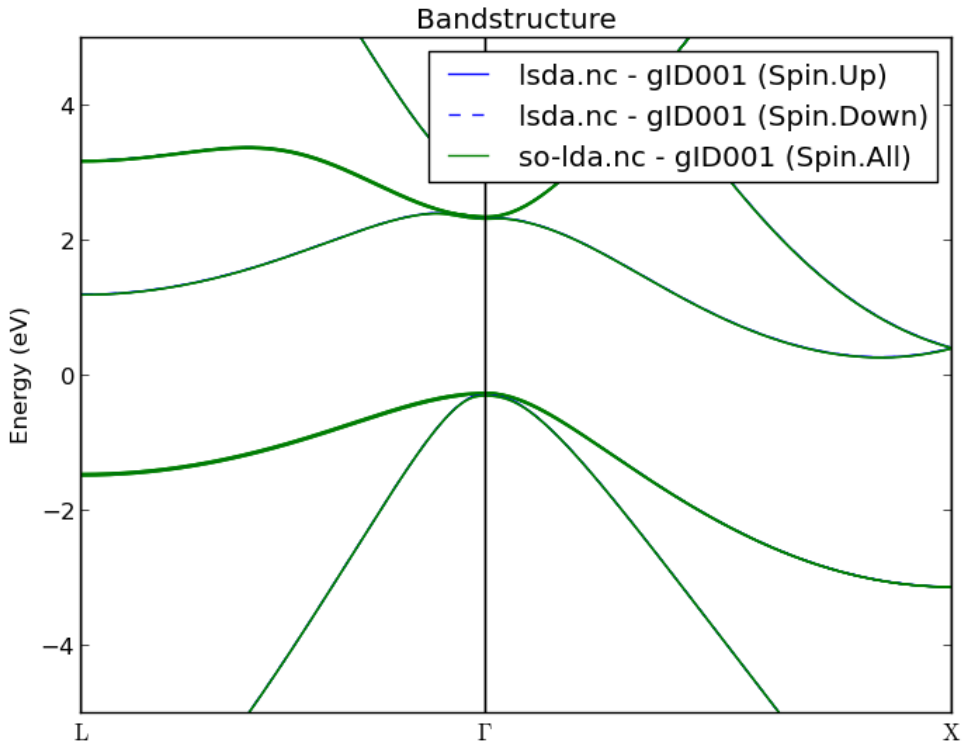


Fig. 3.20: Comparison of the LSDA (blue) and SOLDA (green) band structures of silicon.

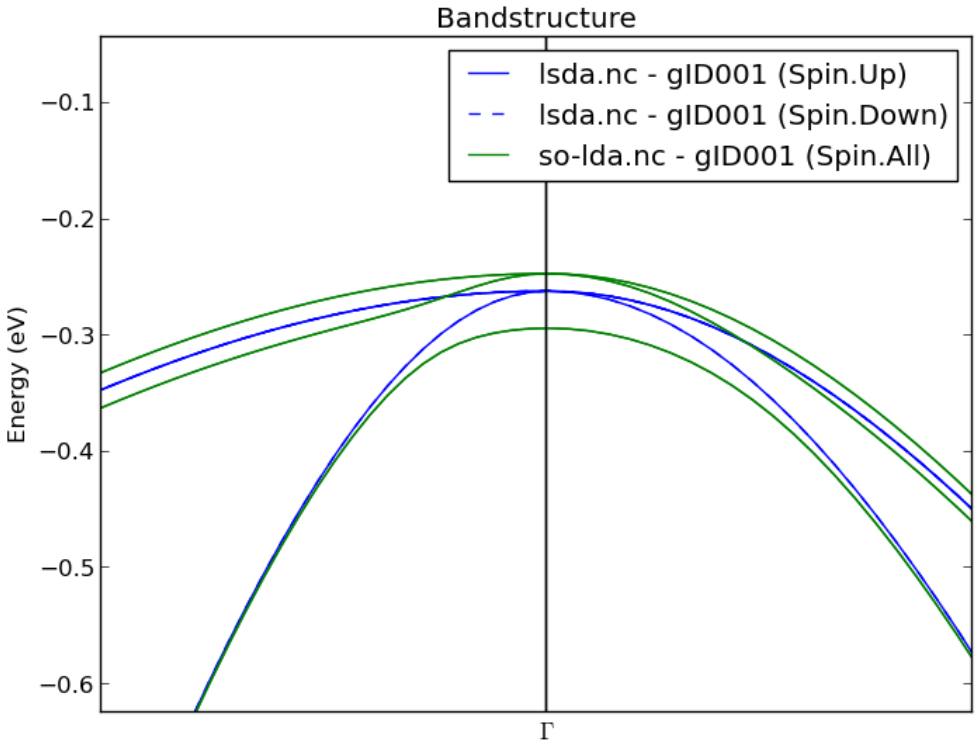


Fig. 3.21: Valence-band top at the silicon Γ-point calculated with LSDA (blue) and SOLDA (green). The latter leads to significant spin-orbit splitting of the bands, and predicts a split-off energy of 44 meV.

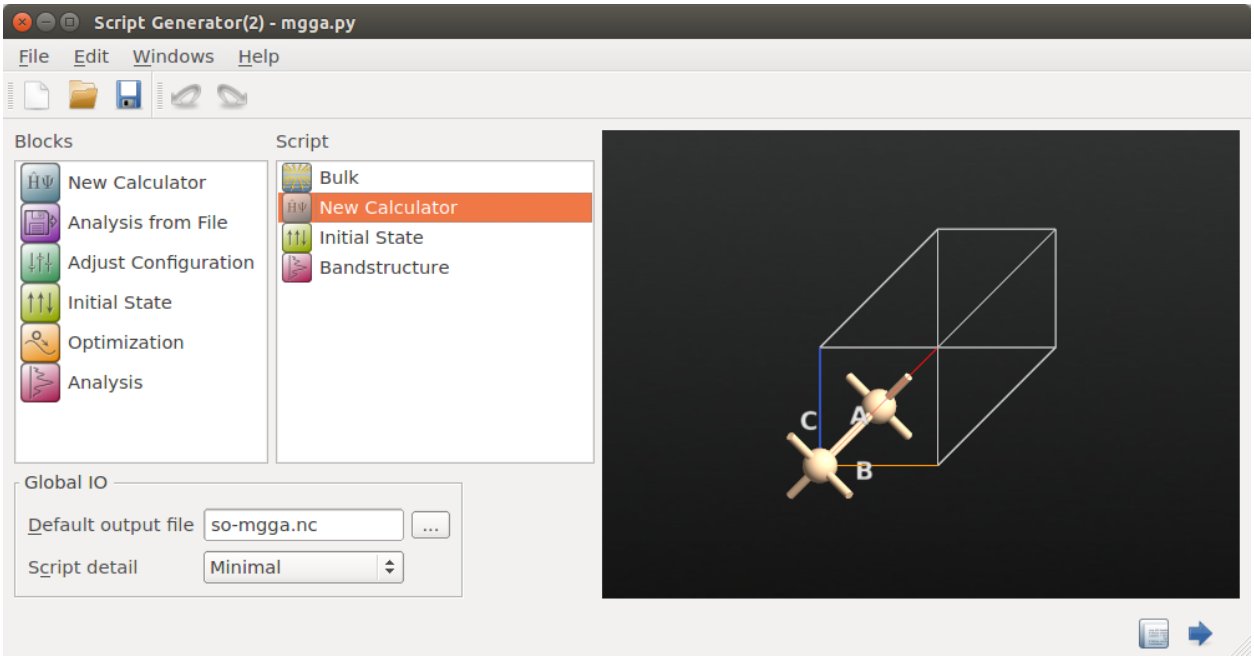
SO+MGGA band gap

The band splitting obtained above with the SO+LDA method is in very good agreement with the experimental splitting (42.6 meVPage 473, 1) However, the band gap is less than half of the experimental value (1.12 eV).

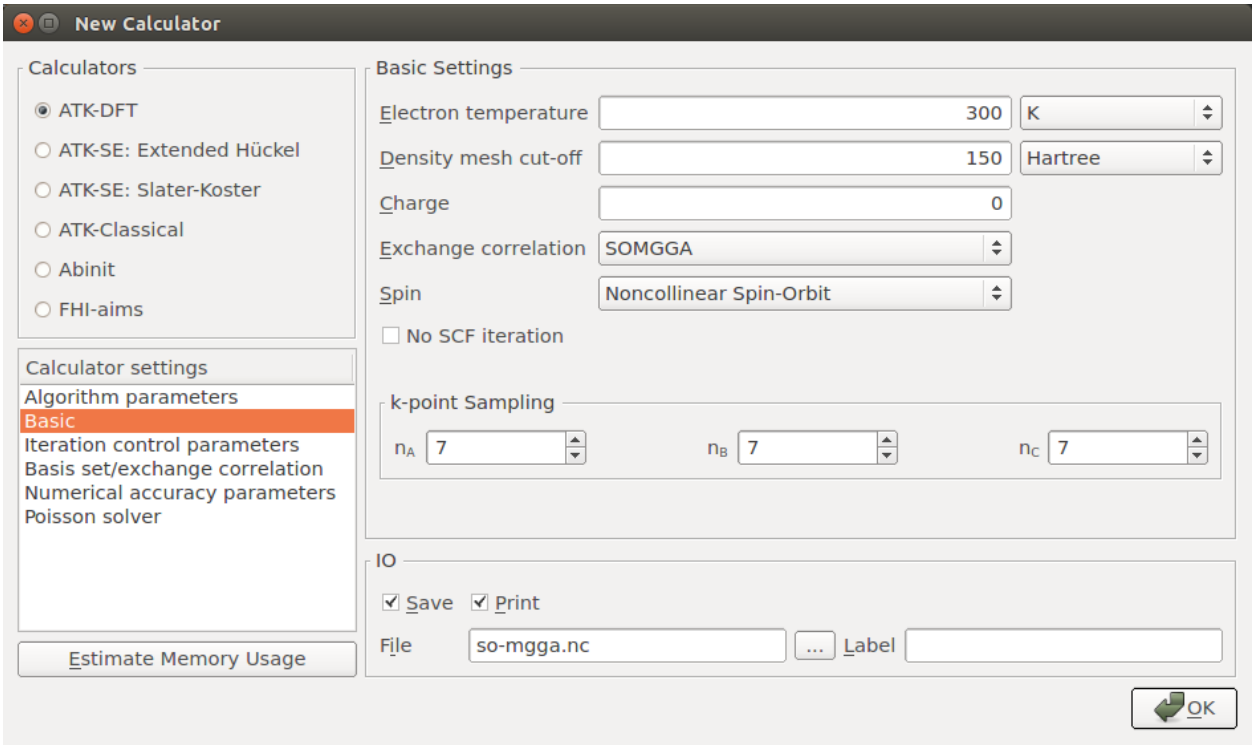
This is largely remedied by applying the TB09LDA method, which employs a meta-GGA exchange-correlation potential and gives remarkably accurate band gaps for semiconductors[2]. We should therefore expect the SO+MGGA method to accurately predict band splittings as well as band gaps.

- Go back to the **Script Generator** and delete the first set of three blocks from the script.
- Change the exchange correlation functional to SOMGGA and use `so-mgga.nc` as output.
- Make sure that the Initial State block reads in the ground state from `lsda.nc`, and that the Bandstructure analysis block saves its output in `so-mgga.nc`.
- Save the script as `mgga.py`. You can also download it if needed: `mgga.py`.
- Run the job – it may take 5-10 minutes to finish if run in serial. This can be reduced to 3 minutes if the job is run in parallel with 4 MPI processes.

Tip: The scf convergence can for small bulk systems be speed-up by increasing the damping factor to 0.5 in the iteration control parameters setup.



[2] F.ăTran and P.ăBlaha. Accurate band gaps of semiconductors and insulators with a semilocal exchange-correlation potential. *Phys. Rev. Lett.*, 102:226401, 2009. doi:10.1103/PhysRevLett.102.226401.



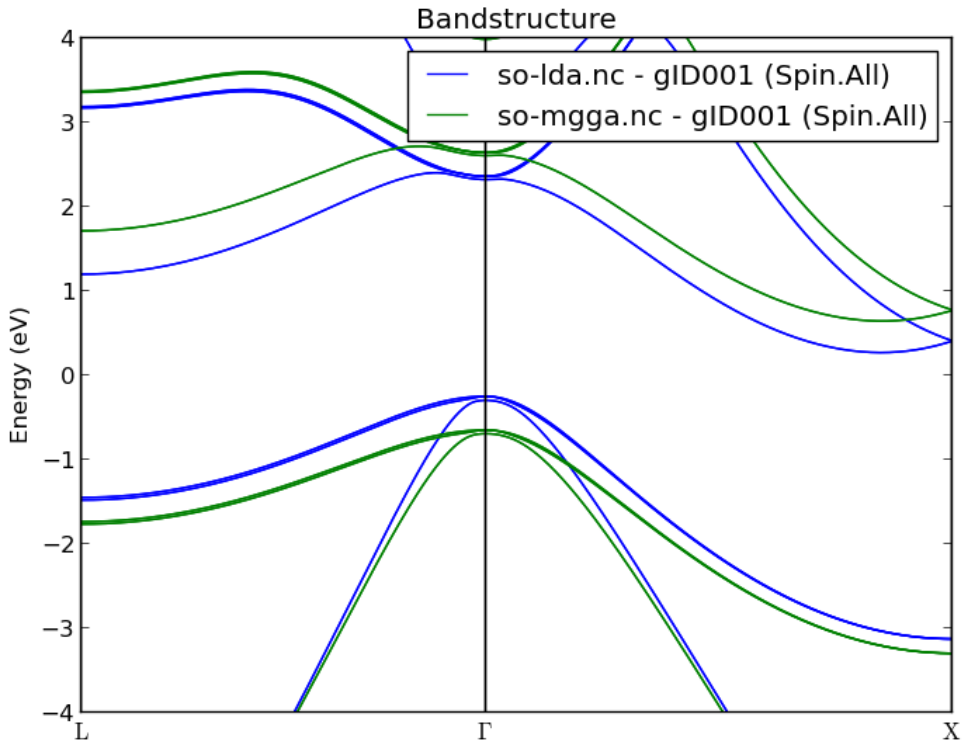


Fig. 3.22: Comparison of the SOLDA and SOMGGA bandstructures of silicon. The SOMGGA indirect band gap of 1.3 eV is close to experimental values.

Results

Using again the Compare Data plugin, you can now compare the SO+LDA band structure to the SO+MGGA one.

In the figure above it is clear that SO+MGGA has increased the distance from valence to conduction bands. Try to use the **Bandstructure Analyzer** plugin to measure the SO+MGGA indirect band gap. You should find ~1.3 eV.

The figure below zooms in on the split valence bands. They have not changed much from the SO+LDA results, apart from a vertical shift.

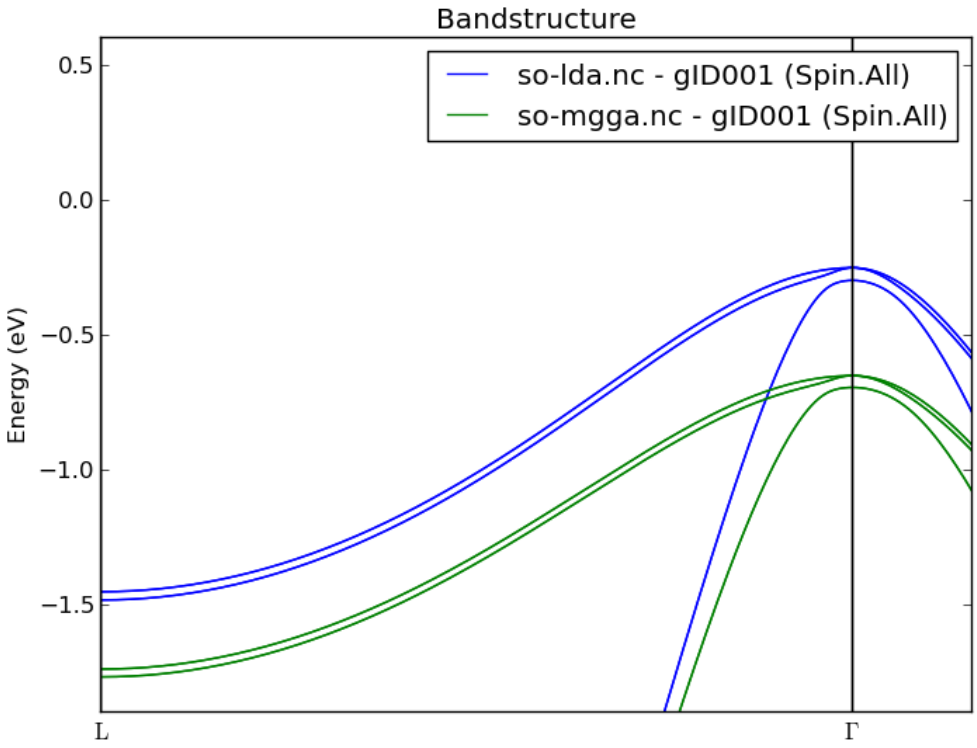


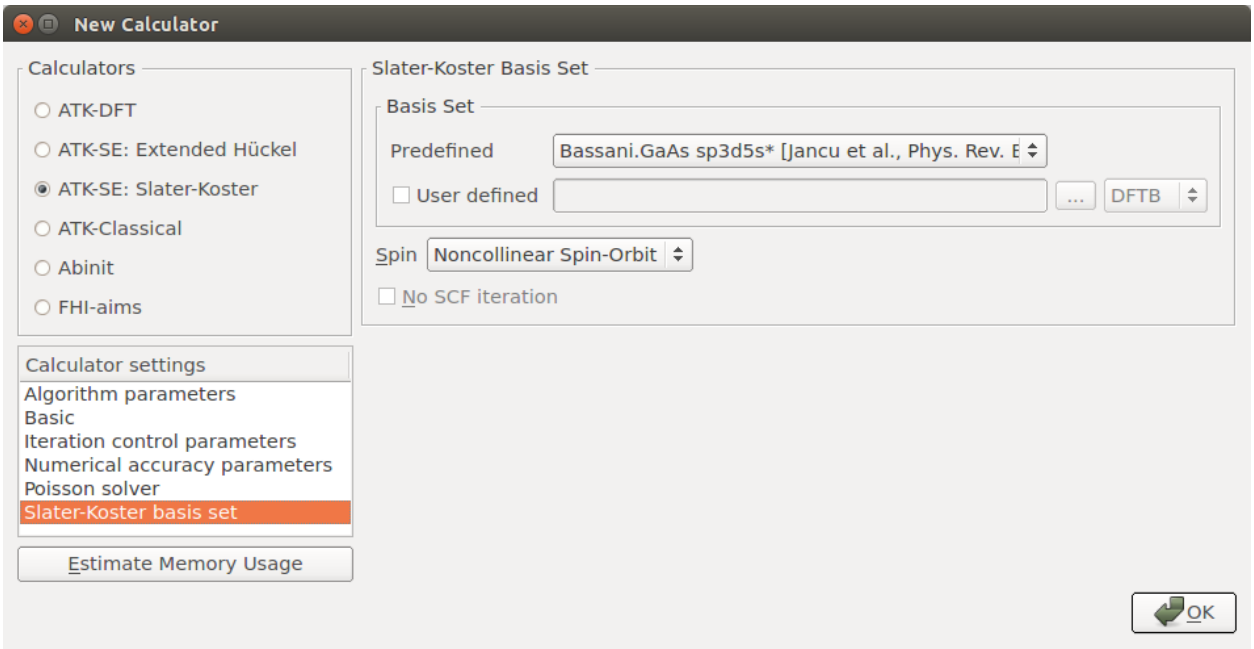
Fig. 3.23: Zoom-in on the split valence bands of silicon.

GaAs band structure with ATK-SE and SO coupling

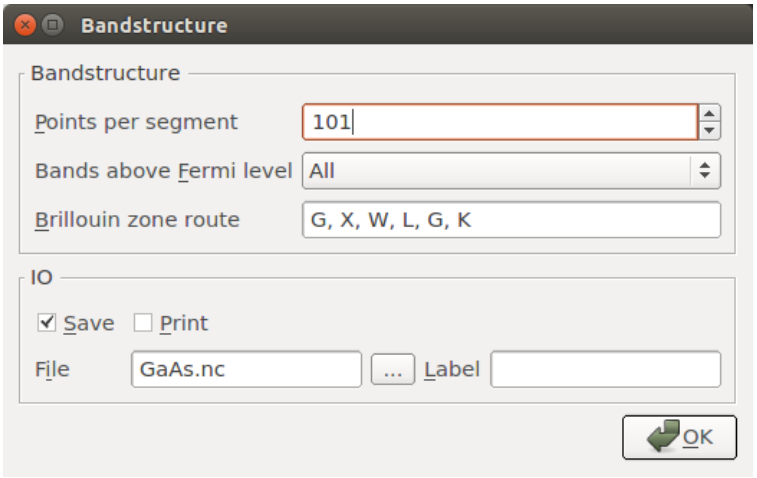
The ATK-SE engine offers tight-binding models for electronic structure calculations as a parametrized alternative to DFT. The Slater–Koster (SK) model is one such method, and includes spin-orbit coupling through a speciel SOC parameter, which is implemented in the ATK-SE engine.

Use the Builder to create the GaAs bulk configuration with experimental lattice constant, and use the Scripter to set up the Slater–Koster calculation:

- Add the New Calculator and Bandstructure blocks.
- Choose the **ATK-SE: Slater-Koster** calculator engine and the *Bassani.sp3d5s* basis set, and choose *Noncollinear Spin-Orbit* for the spin.



•In the Bandstructure block, increase the *Points per segment* to 101 in order to finely sample the band structure between each high-symmetry point, and trim the default *Brillouin zone route* such that it reads G, X, W, L, G, K.



Run the calculation – it will only take a few seconds – and then use the Bandstructure Analyzer to plot the results. The SO induced spin-splitting of the GaAs band structure is clearly visible. If you zoom in a bit you can measure a direct band gap of 1.51 eV at the Γ-point, which is virtually identical to the experimental gap at zero Kelvin.

References

3.2.4 Silicon p-n junction

In this tutorial you will:

•Learn how to build a p-n junction device.

•Learn how to study the current-voltage characteristic of such a device.

•Compare two different methods: **ATK-SE** and **ATK-DFT**.

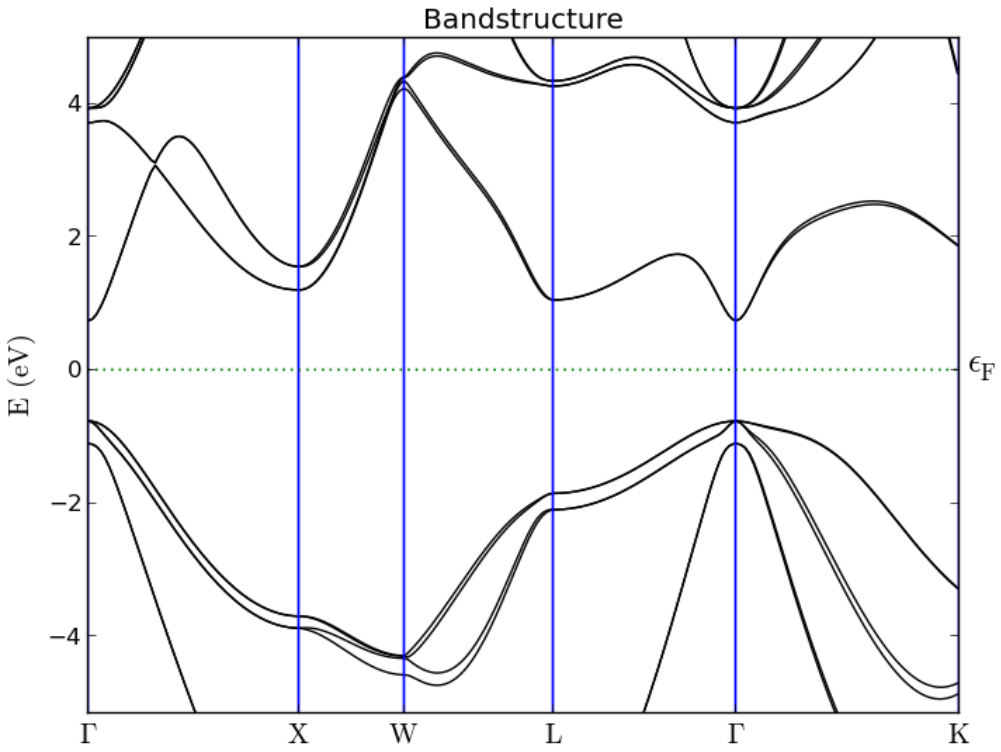
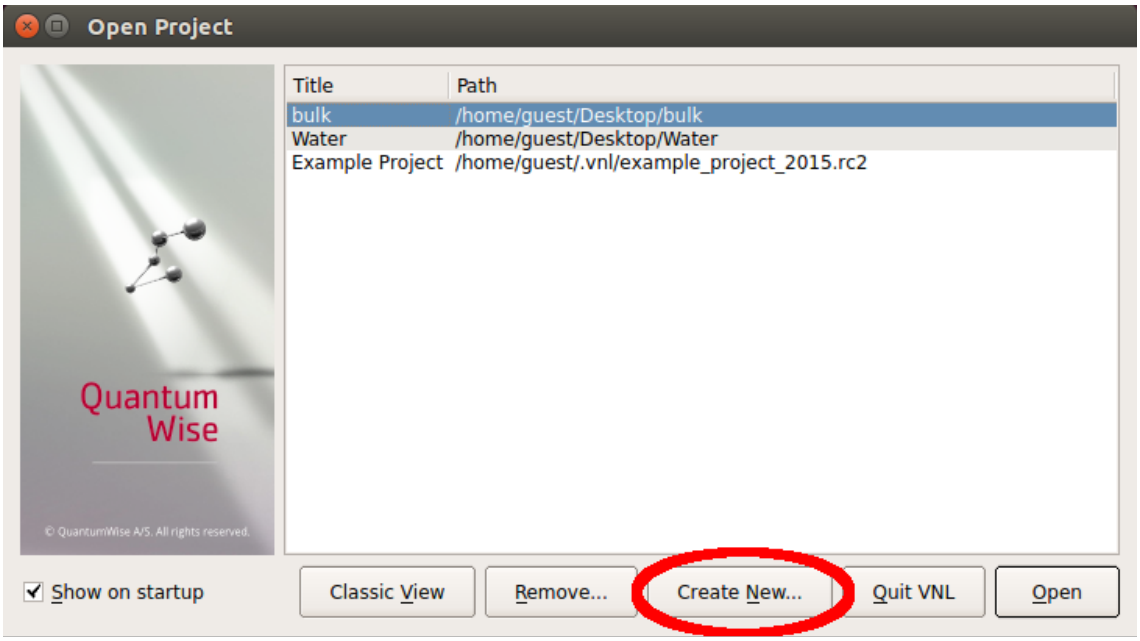


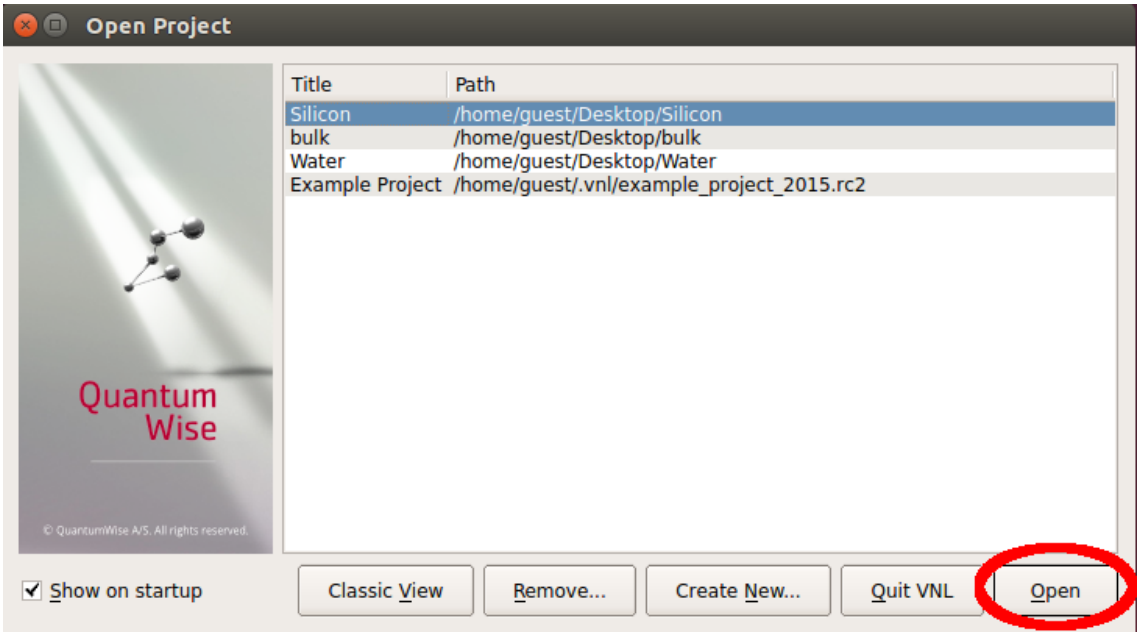
Fig. 3.24: Slater–Koster spin-orbit band structure of GaAs.

•Analyze the electronic structure of the p-n junction by studying and plotting the device density of states at zero bias and at reverse and forward bias.

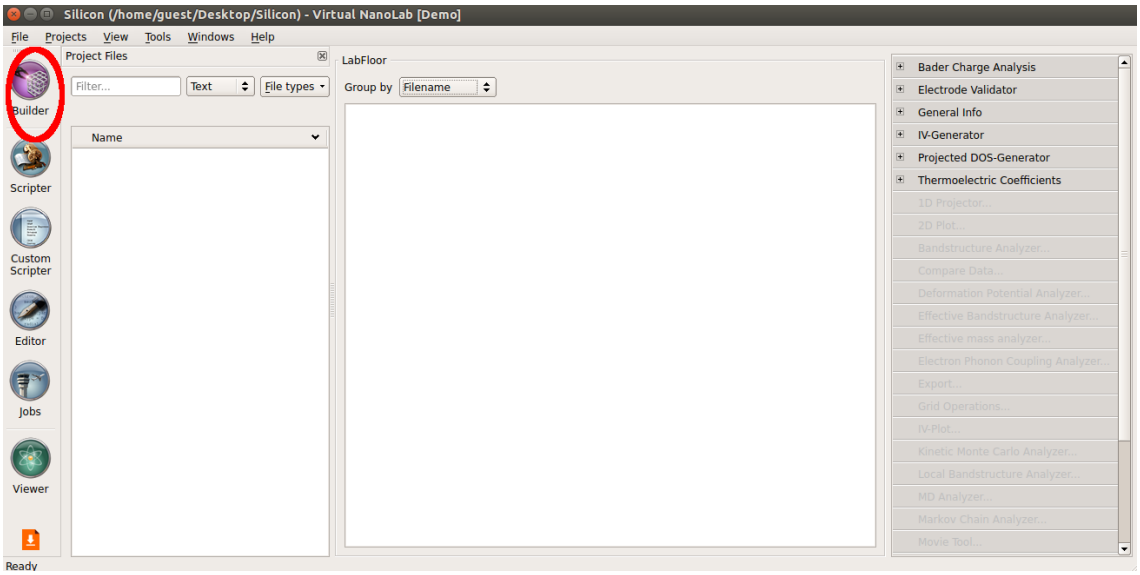
Silicon bulk: Slater-Koster vs DFT-MGGA

1.Open **QuantumATK** and create a new project by clicking on **Create New**. Give the project a *Title* (here: *Silicon*), select a folder for the project, click **OK** and then **Open** to start the project.

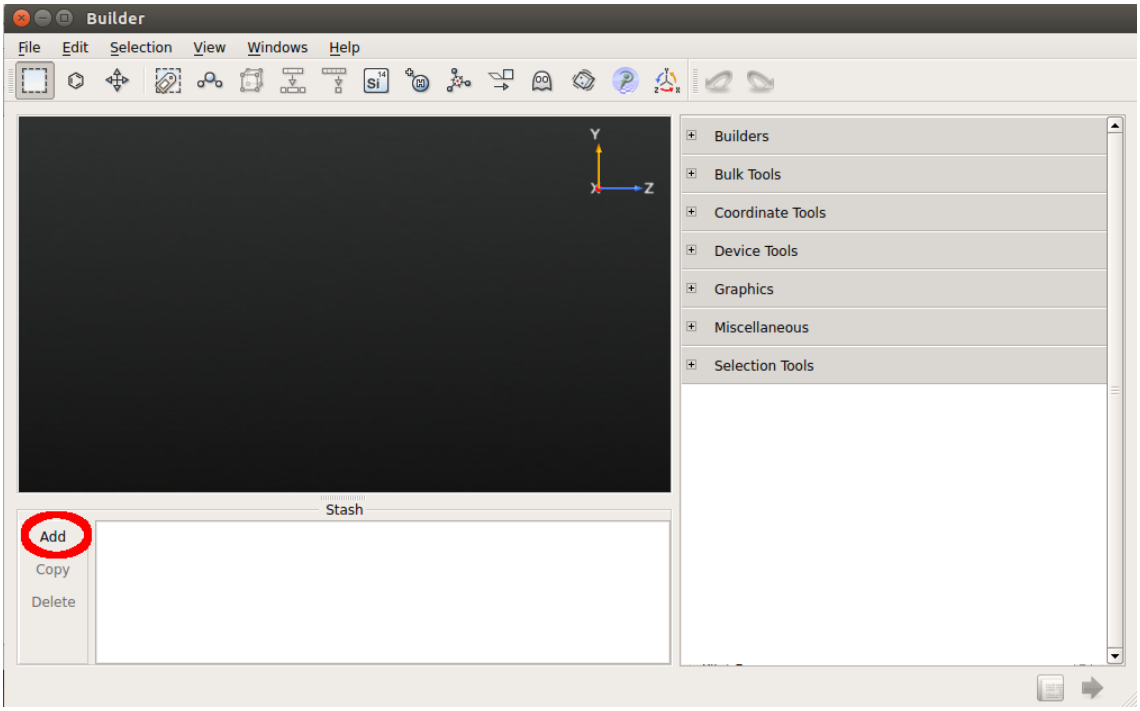


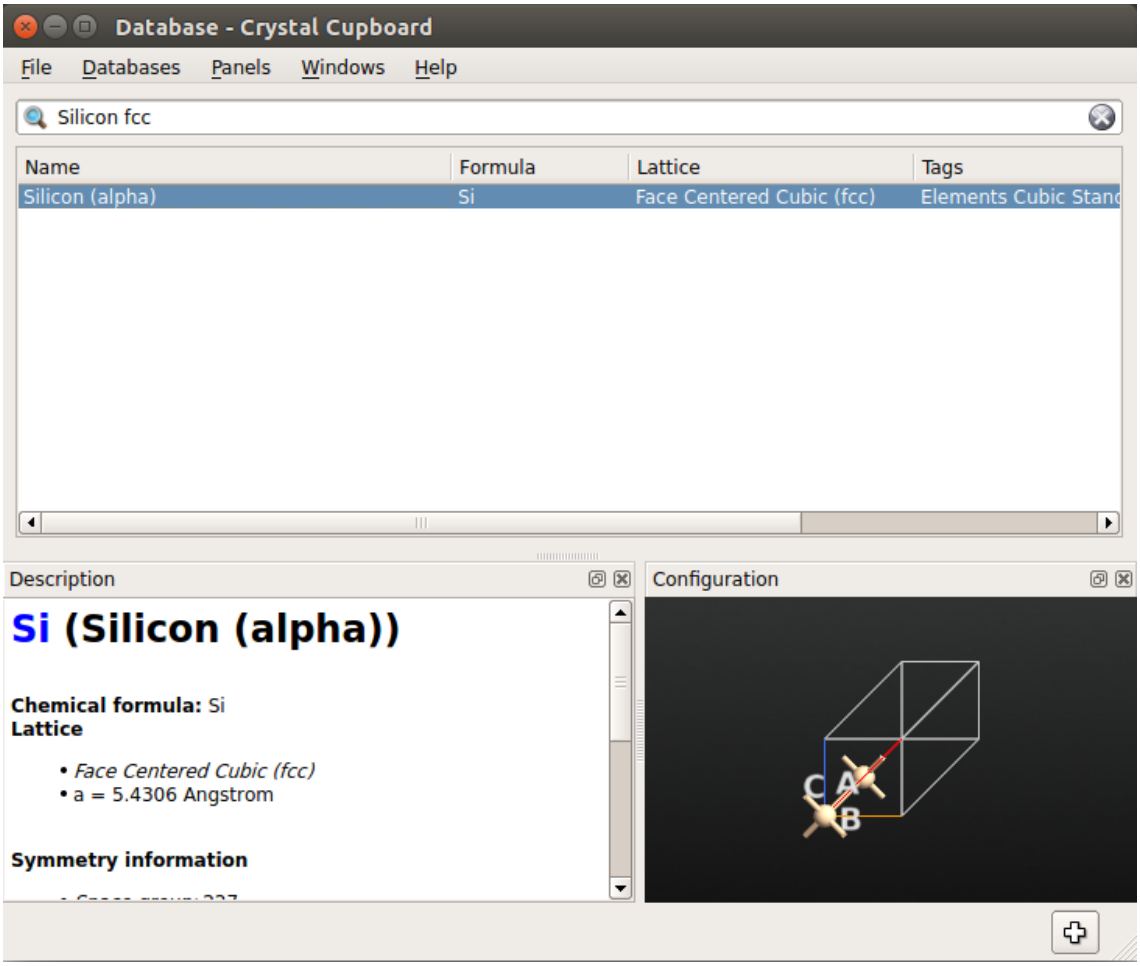


2.In the **QuantumATK** main window, click on the icon to open the **Builder**.



3. In the **Builder**, go to the **Stash**, click *Add* → *From Database* and from the menu navigate to *Databases* → *Crystals* and search for *silicon* in the search window. Then, select *Silicon (alpha)* and click the icon, or double-click its line in the list, to send it to the **Stash**.

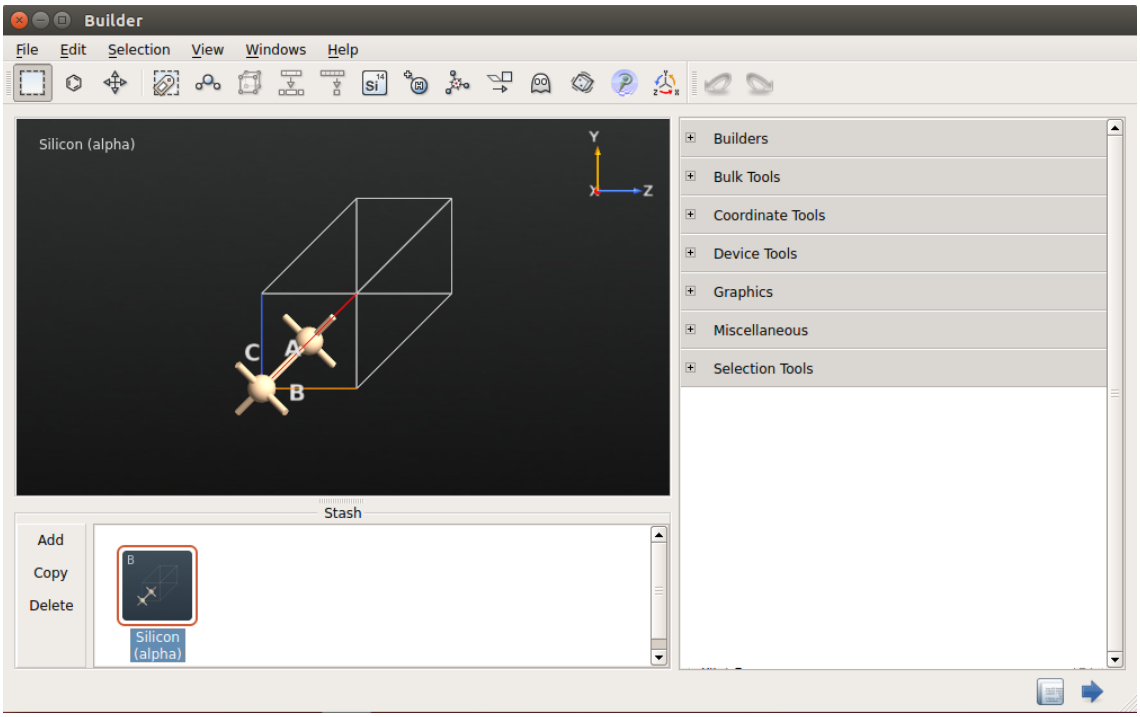




Slater-Koster calculation

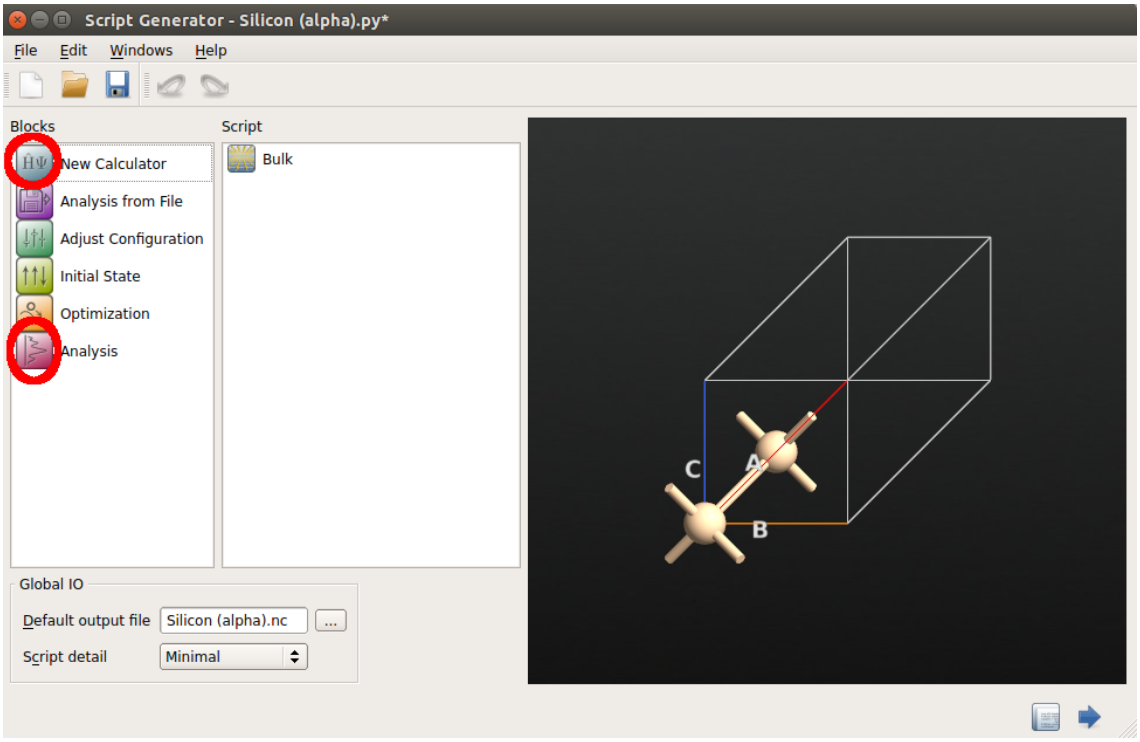
You will now set up an **ATK** semiempirical (**ATK-SE**) calculation using the Slater-Koster model Hamiltonian.

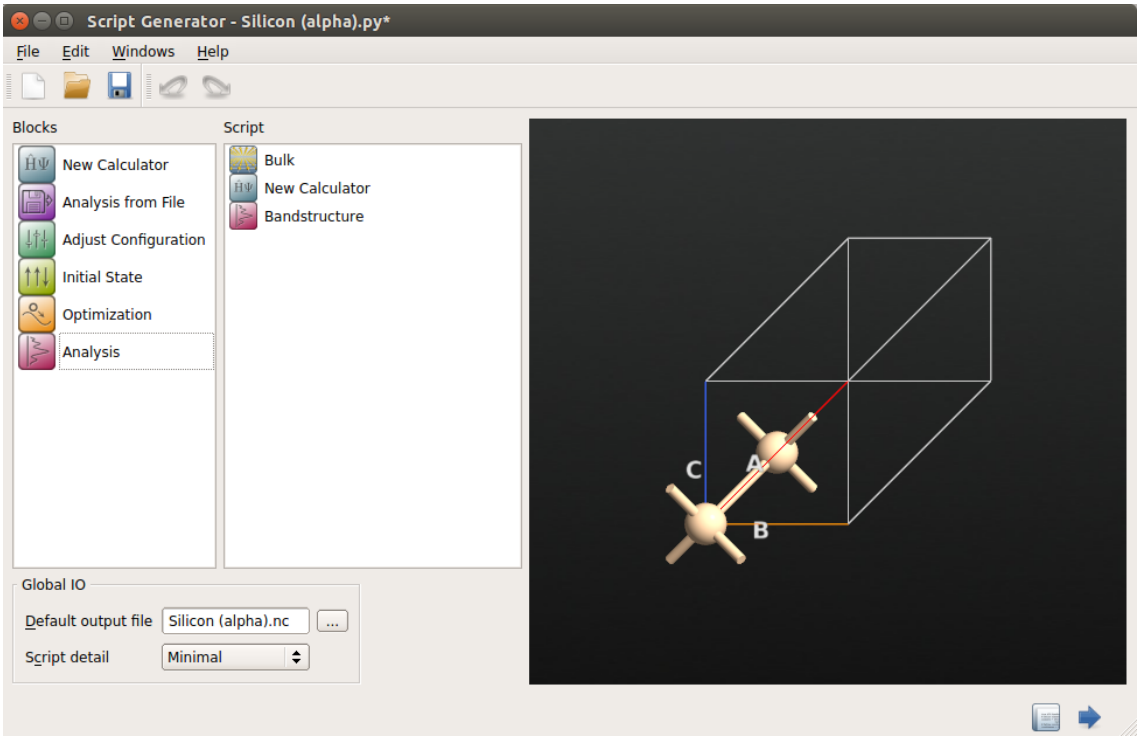
1. Send the bulk configuration from the **Builder** to the **Script Generator** using the button.



2.Add the following blocks to the *Script* by double-clicking the correspoding icons in the *Blocks* panel:

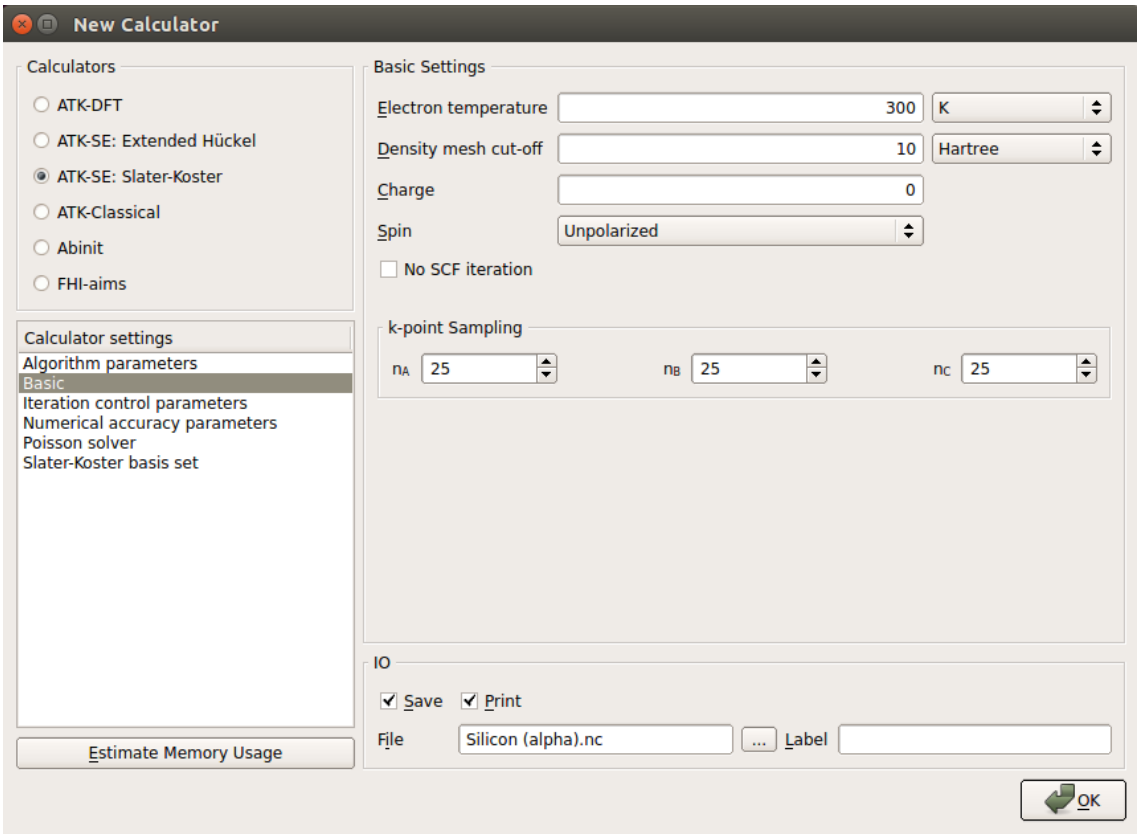
- New Calculator.
- *Analysis* → *Bandstructure*.

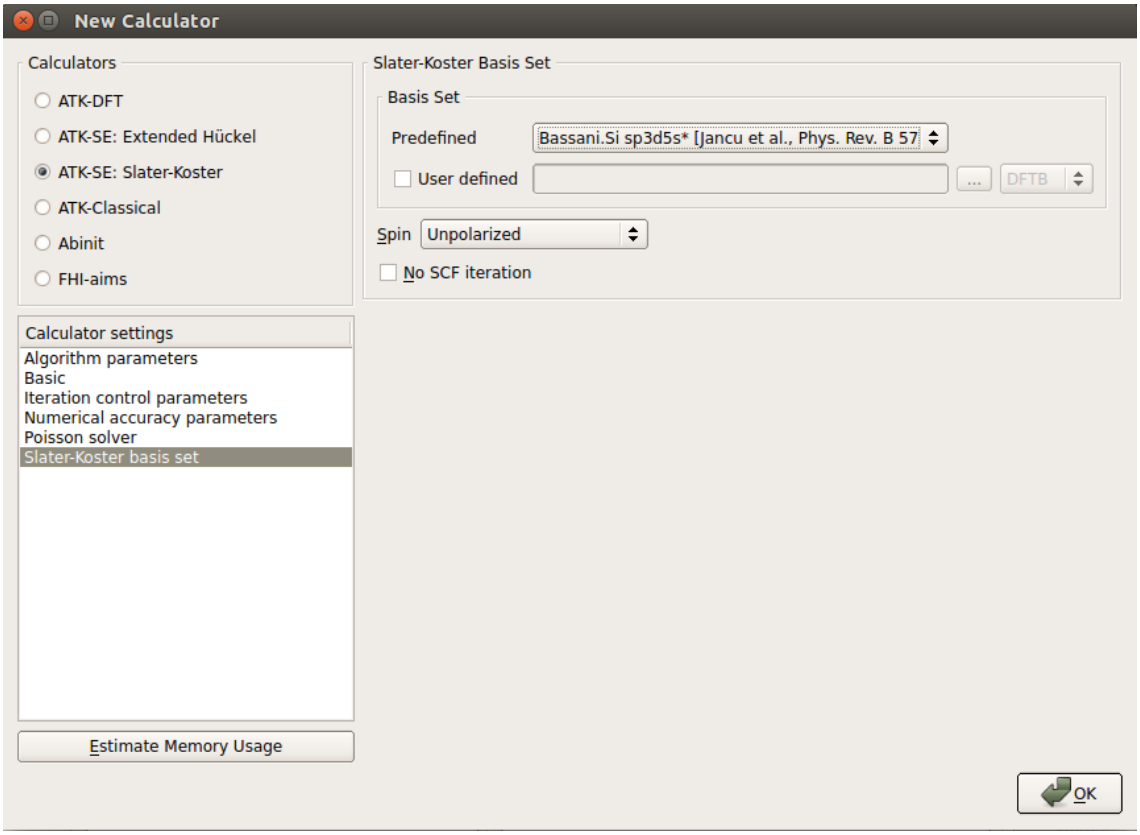




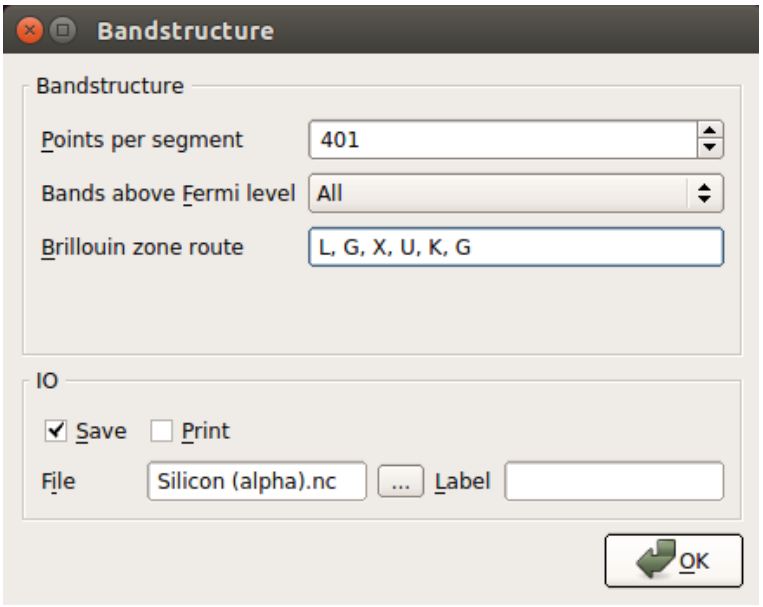
3.Open the **New Calculator** by double-clicking it.

- Select the *ATK-SE Slater-Koster* calculator.
- Use a 25x25x25 k-point sampling.
- Uncheck the *No SCF iteration* box.
- Select the *Bassani.Si* basis set.

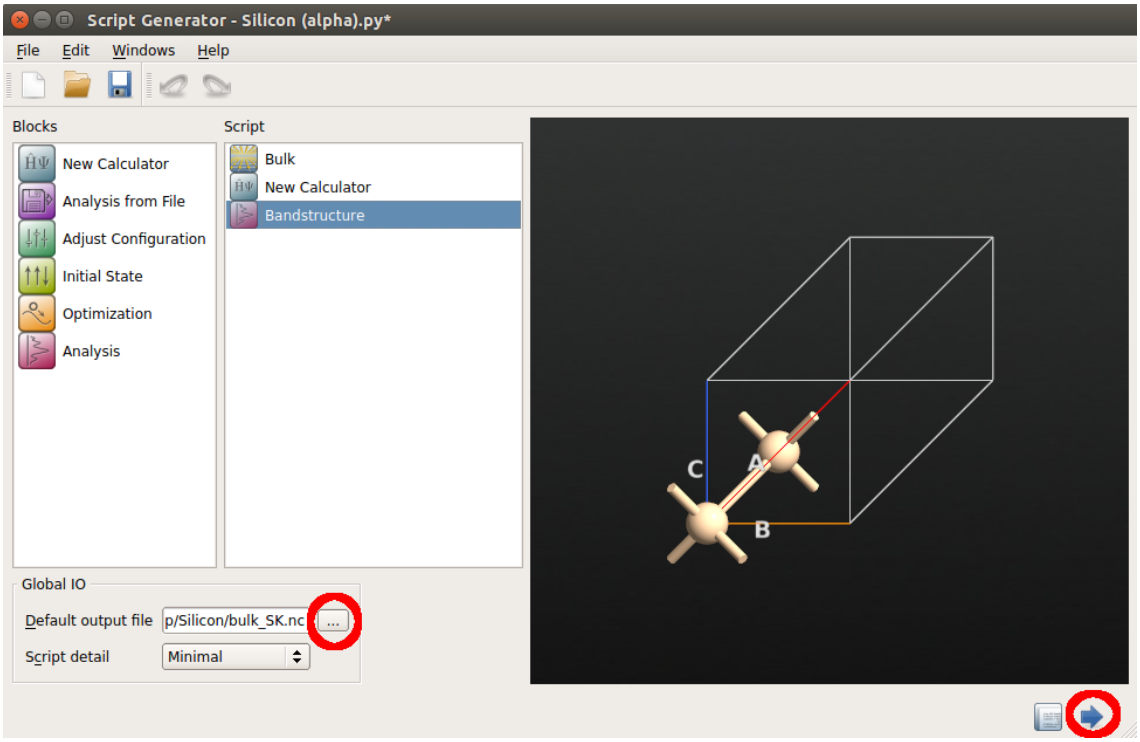

New Calculator
Calculators
ATK-DFT
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-Classical
Abinit
FHI-aims
Calculator settings
Algorithm parameters
Basic
Iteration control parameters
Numerical accuracy parameters
Poisson solver
Slater-Koster basis set
Estimate Memory Usage
Basic Settings
Electron temperature
300
K
Density mesh cut-off
10
Hartree
Charge
0
Spin
Unpolarized
No SCF iteration
k-point Sampling
nA
25
nB
25
nC
25
IO
Save
Print
File
Silicon (alpha).nc
...
Label
OK



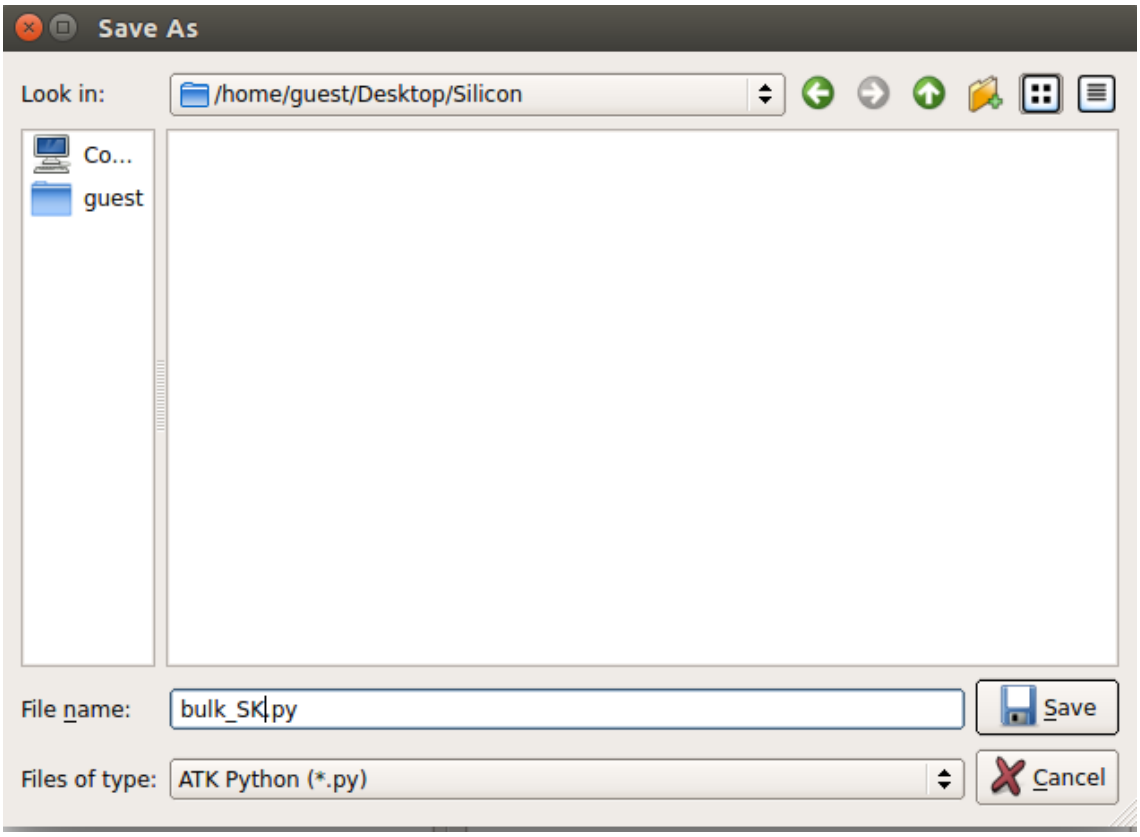
4.In **Bandstructure**, select the L, G, X, U, K, G Brillouin zone route and 401 points per segment.



5.Back in the **Script Generator**, set the output filename to `bulk_SK.hdf5`, and send the script to the **Job Manager** using the button.



6.Save the script in the window that shows up and use the ▶ botton to run the job.

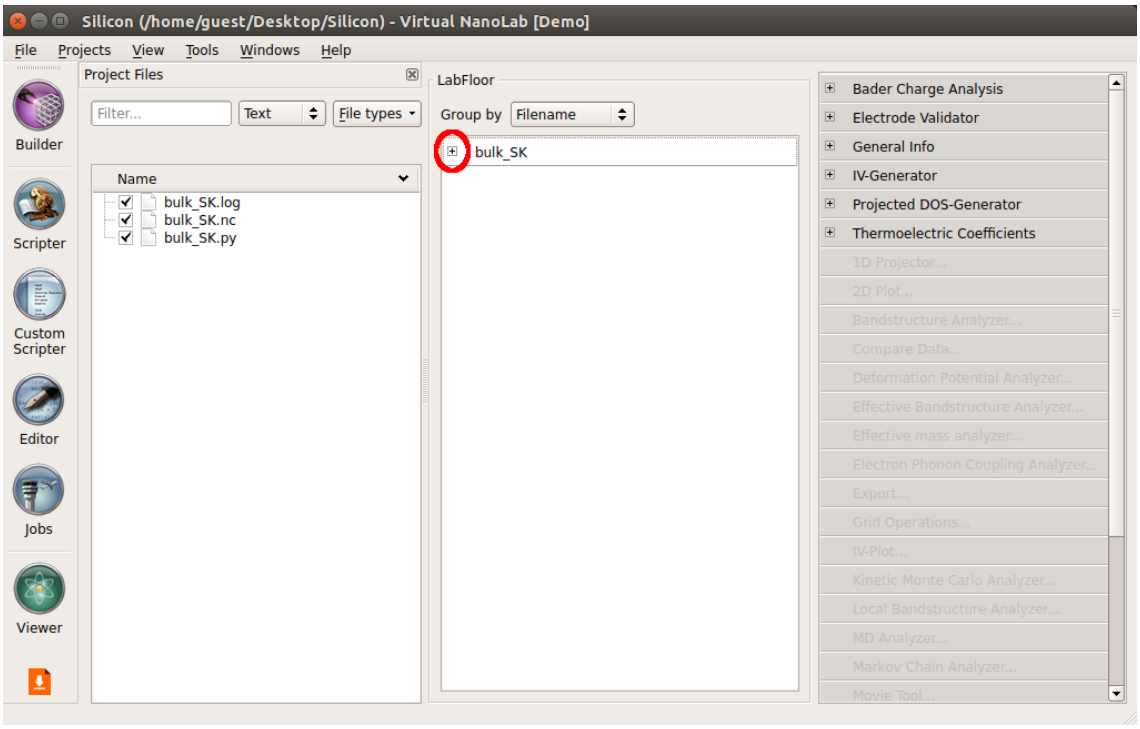


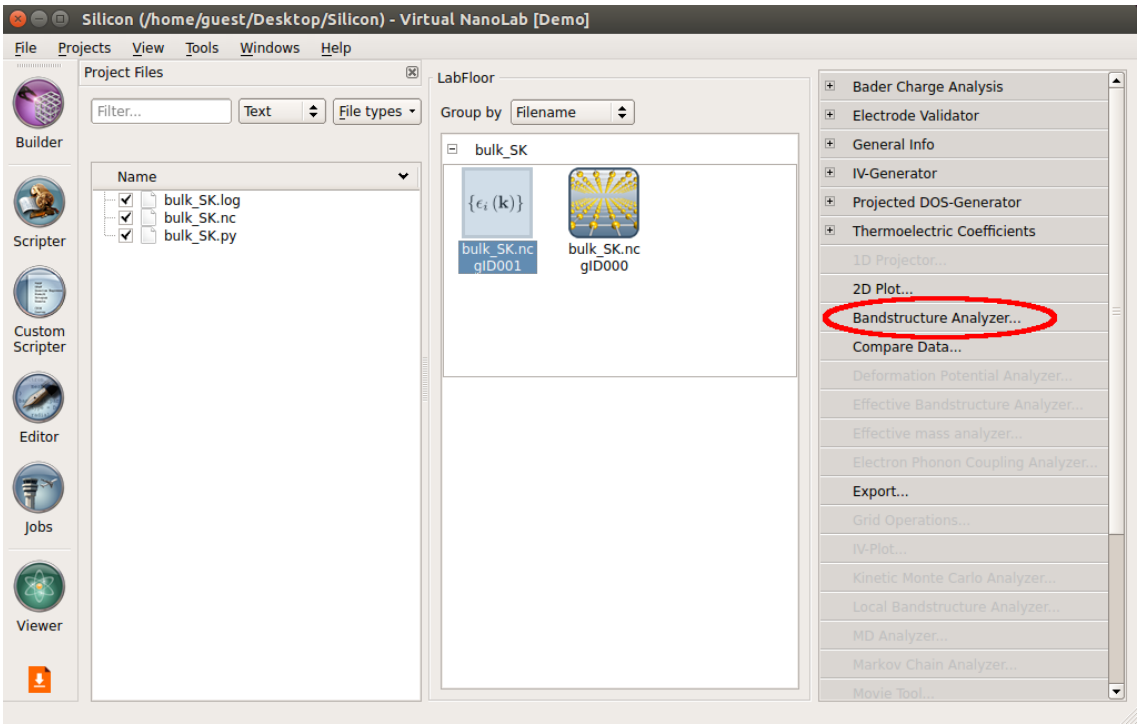
7.Once the calculation is done (it will only take a few seconds) you can find in the **LabFloor** your **Bandstructure** object,

which is saved in `bulk_SK.hdf5`:

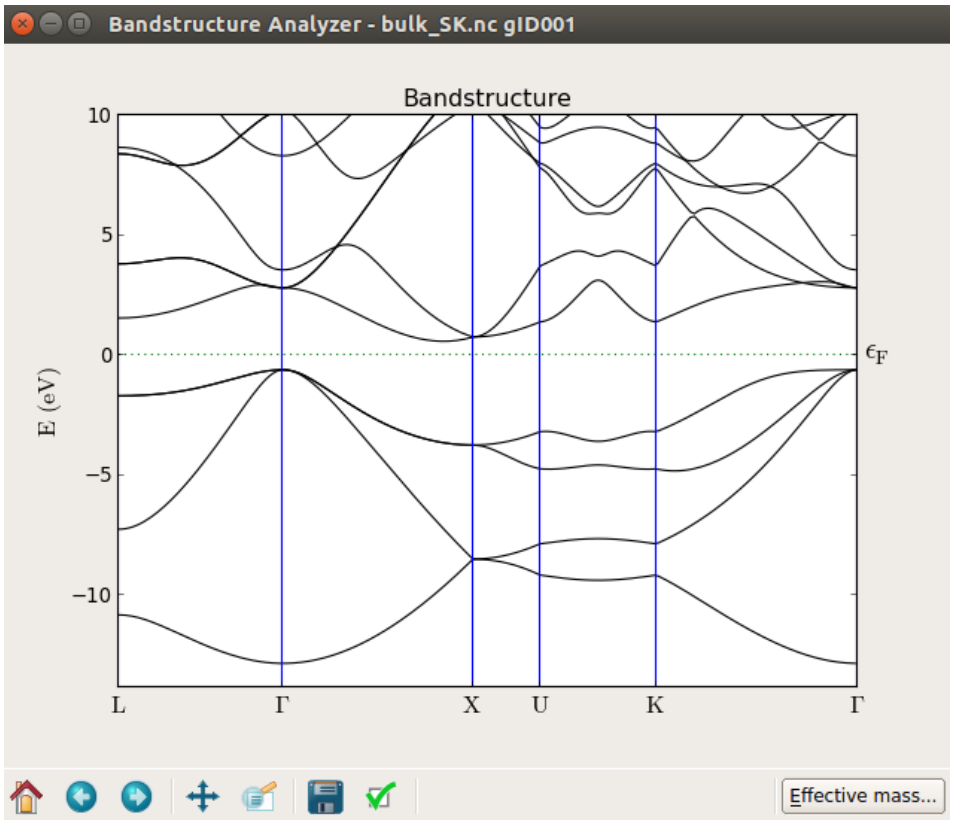
$$\{\epsilon_i(\mathbf{k})\}$$

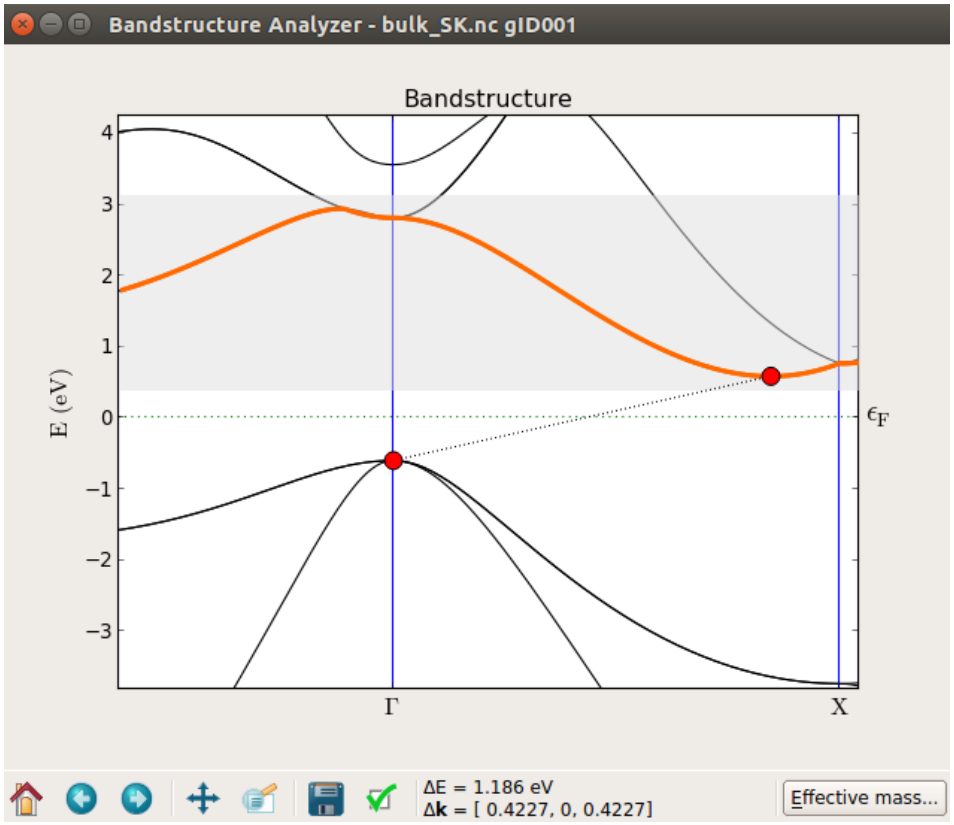
Highlight the object and use the **Bandstructure Analyzer** plugin to plot the bandstructure.





8. You can zoom into a region close to the band edges and precisely measure the calculated indirect band gap, 1.186 eV.





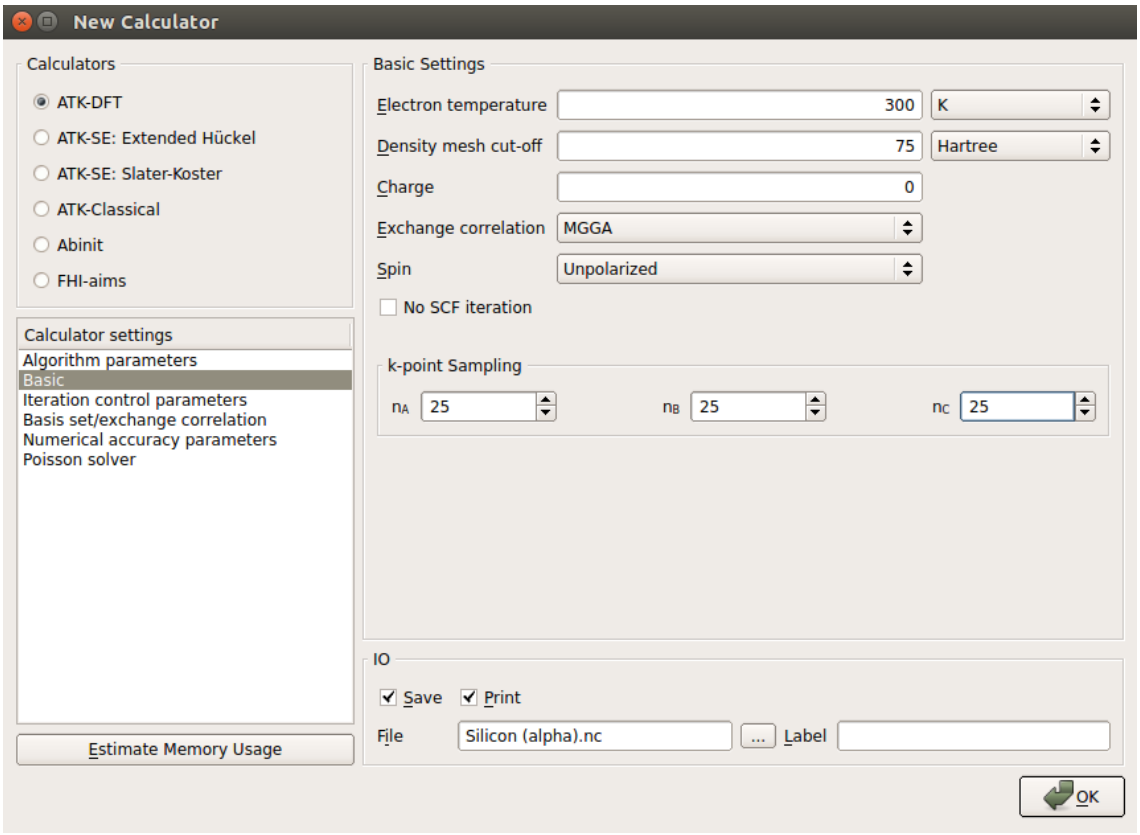
DFT-MGGA calculation

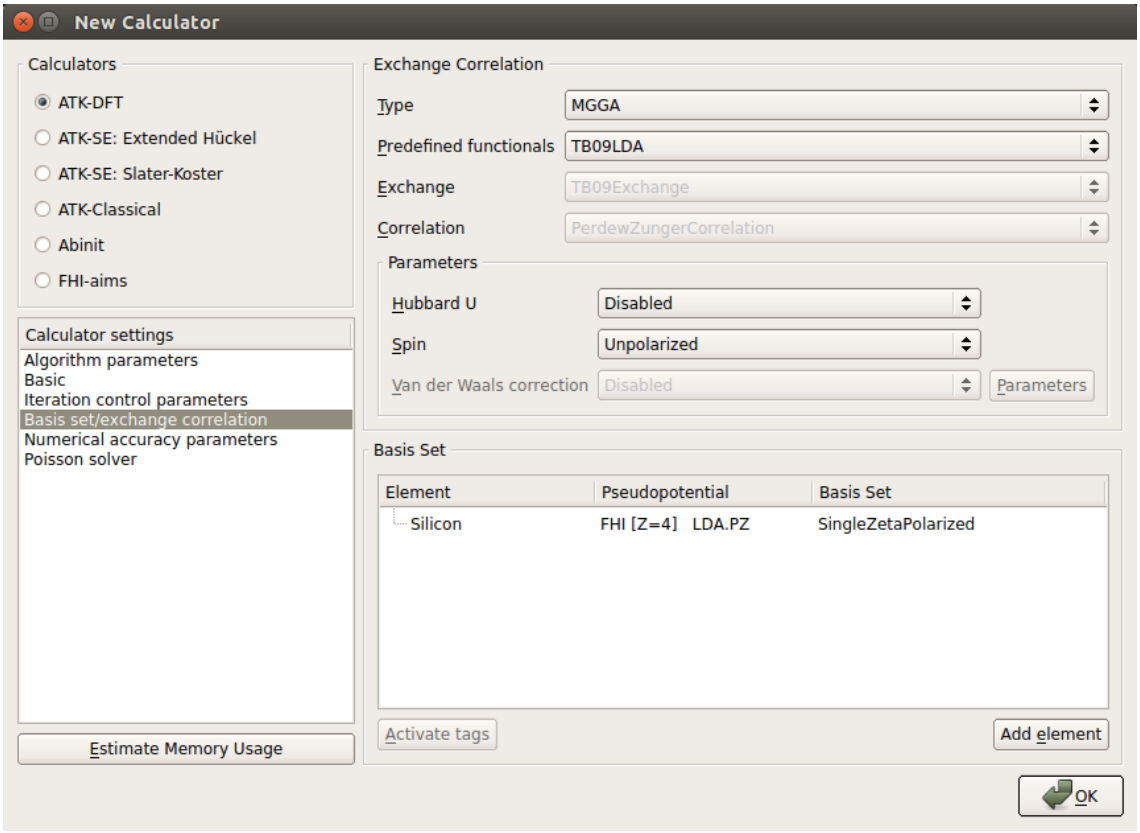
The TB09-MGGA approximation[1] available in **ATK-DFT** is fully explained in the tutorial *Meta-GGA and 2D confined InAs*. In order to compare the Slater-Koster results with results from the TB09-MGGA method, you will fit a parameter (c) such the TB09-MGGA band gap of silicon matches that calculated with Slater-Koster. Throughout this tutorial you will use this fitted c parameter for the TB09-MGGA calculations.

1. From the **Builder**, send the *Silicon bulk configuration* to the **Script Generator** and add a **New Calculator** to the script. Then edit the calculator parameters:

- Select *ATK-DFT* and the *MGGA* exchange-correlation functional.
- Use a 25x25x5 k-point sampling.
- Select the *SingleZetaPolarized* basis set in order to speed up calculations.

[1] F.ăTran and P.ăBlaha. Accurate band gaps of semiconductors and insulators with a semilocal exchange-correlation potential. *Phys. Rev. Lett.*, 102:226401, 2009. doi:10.1103/PhysRevLett.102.226401.


New Calculator
Calculators
ATK-DFT
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-Classical
Abinit
FHI-aims
Calculator settings
Algorithm parameters
Basic
Iteration control parameters
Basis set/exchange correlation
Numerical accuracy parameters
Poisson solver
Estimate Memory Usage
Basic Settings
Electron temperature
300
K
Density mesh cut-off
75
Hartree
Charge
0
Exchange correlation
MGGA
Spin
Unpolarized
No SCF iteration
k-point Sampling
nA
25
nB
25
nC
25
IO
Save
Print
File
Silicon (alpha).nc
...
Label
OK

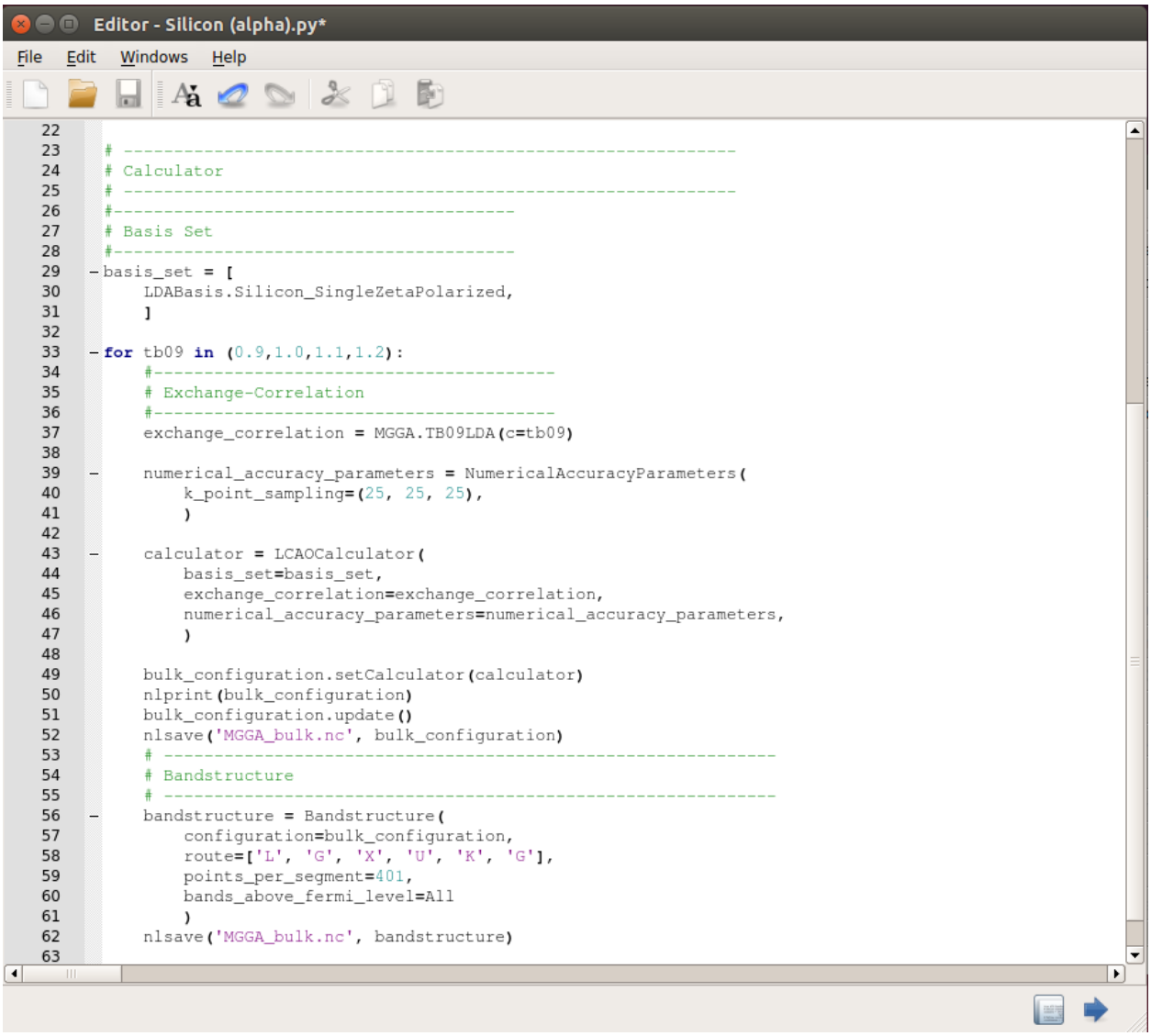


2.Add also a **Bandstructure** analysis object and set it up as you did for the Slater-Koster calculation.

3.Back in the **Script Generator**, set the output filename to `MGGA_bulk.hdf5`.

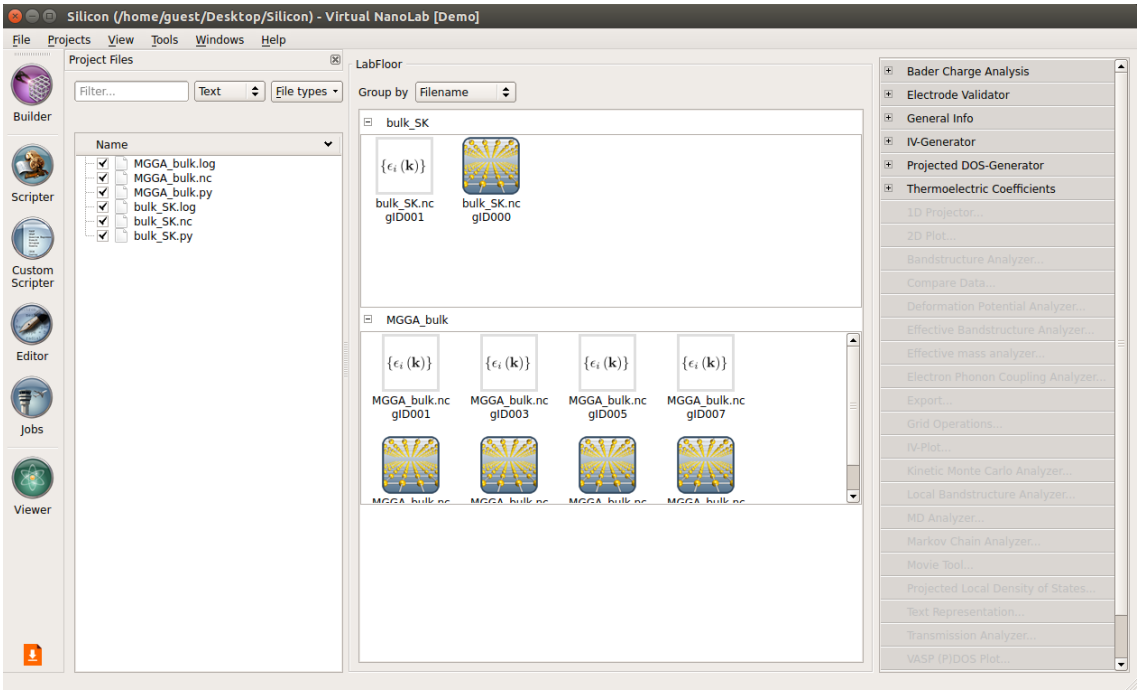
4.In order to fit the c parameter to the Slater-Koster band gap you will need to run a few calculations for different c parameters and make a linear interpolation. Here, you will run four TB09-MGGA calculations for c = 0.9, 1.0, 1.1 and 1.2. A little manual editing of the script is needed:

•Send the script to the **Editor** using the button, and modify the exchange-correlation section as shown in the image below. Also, add a loop over the c values to run the four calculations with a single Python script (you can download a copy of the script here: `MGGA_bulk.py`):



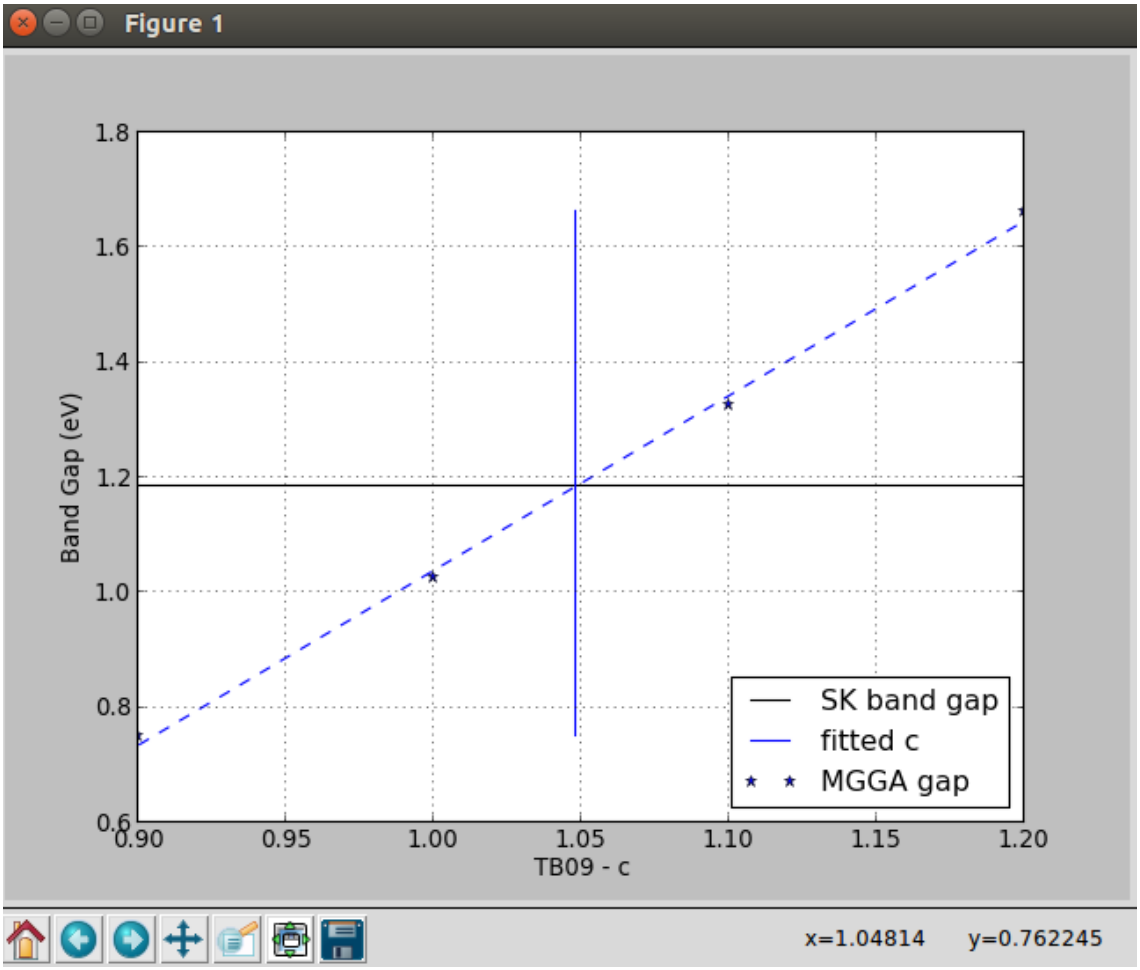
5.Send the script to the **Job Manager** and run the job.

6.On the **LabFloor** you will now have four different **Bandstructure** objects included in the *MGGA_bulk.hdf5*.



7.You will use the following script to fit the c-parameter: `bandstructure_fit.py`. Save it in the project directory.

8.Execute the fitting script using the **Job Manager**. The plot shown below should pop up.



It is clear that the TB09 c-parameter has a significant influence on the calculated band gap, and that the dependence is quite linear. The fitted value of c is 1.048.

Note: If you take a closer look at the script `bandstructure_fit.py`, You will see that band gaps are calculated from a **Bandstructure** object like this:

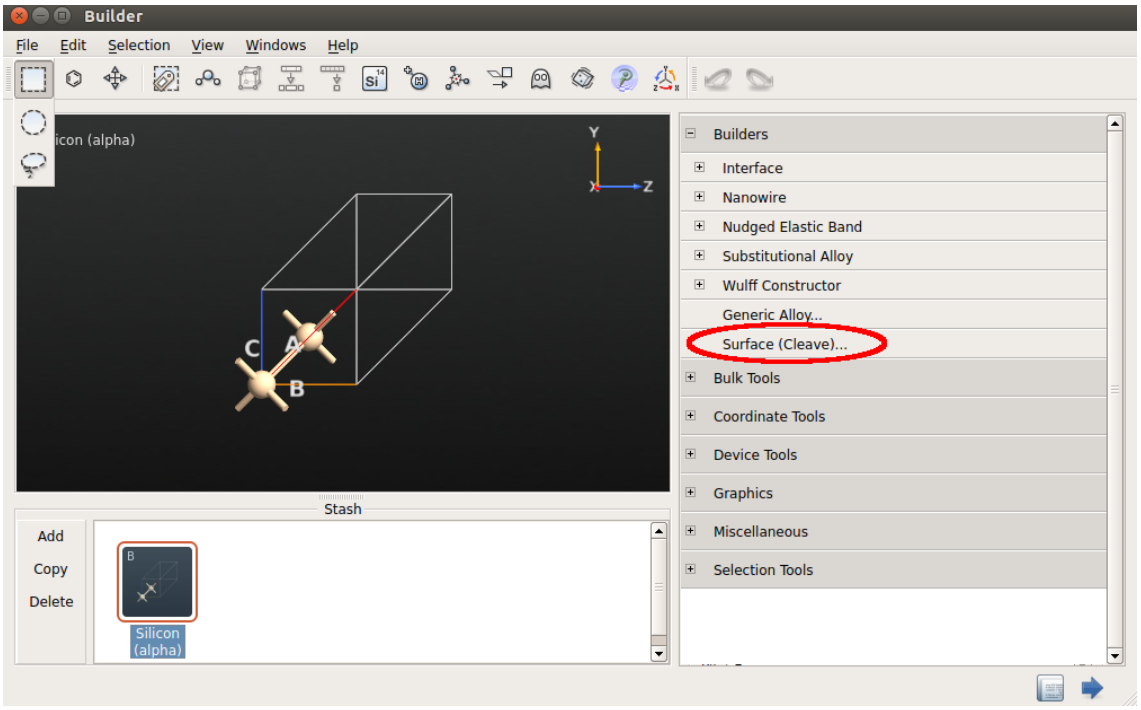
```
bandstructure = nlread("file.hdf5", Bandstructure)[0]
gap_direct   = bandstructure.directBandGap().inUnitsOf(eV)
gap_indirect = bandstructure.indirectBandGap().inUnitsOf(eV)
print "direct gap:   %.2f eV" % gap_direct
print "indirect gap: %.2f eV" % gap_indirect
```

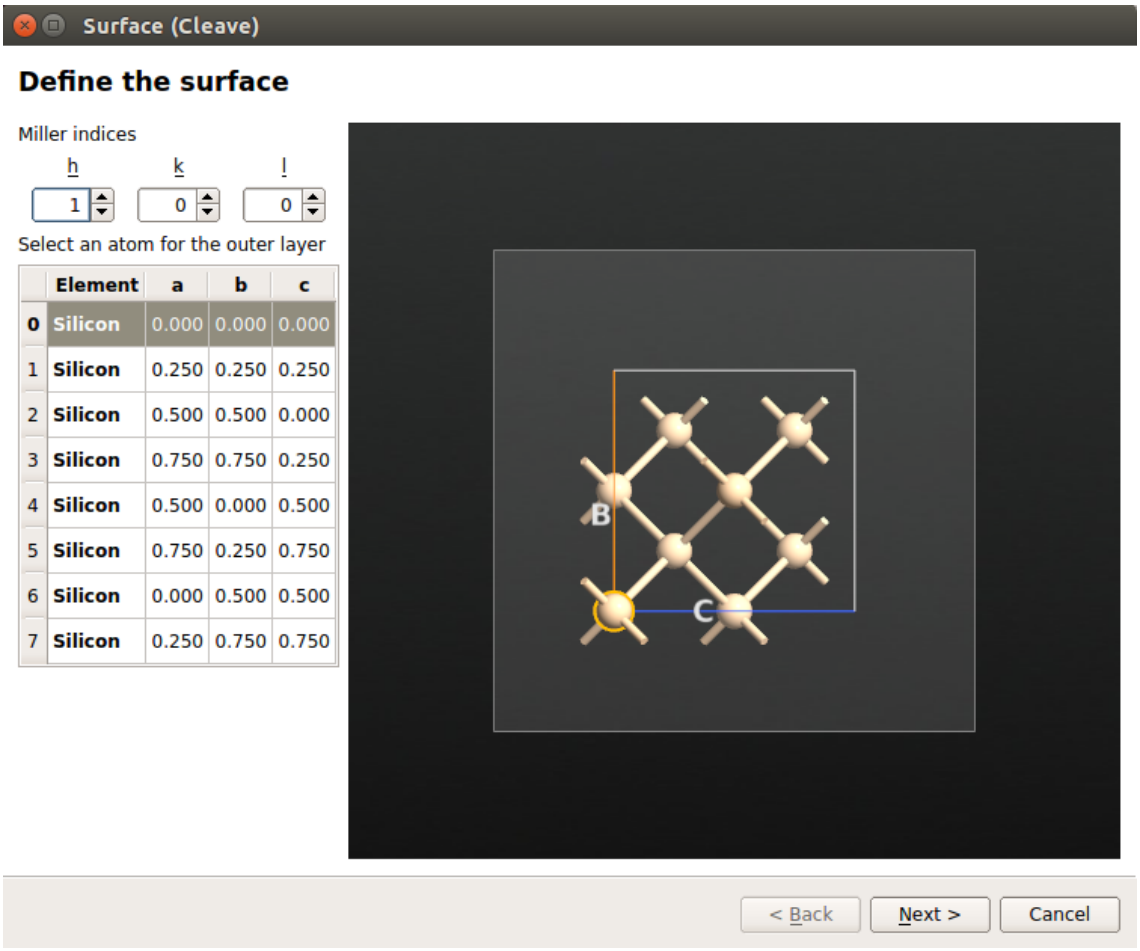
Silicon device

You will now set up the silicon device.

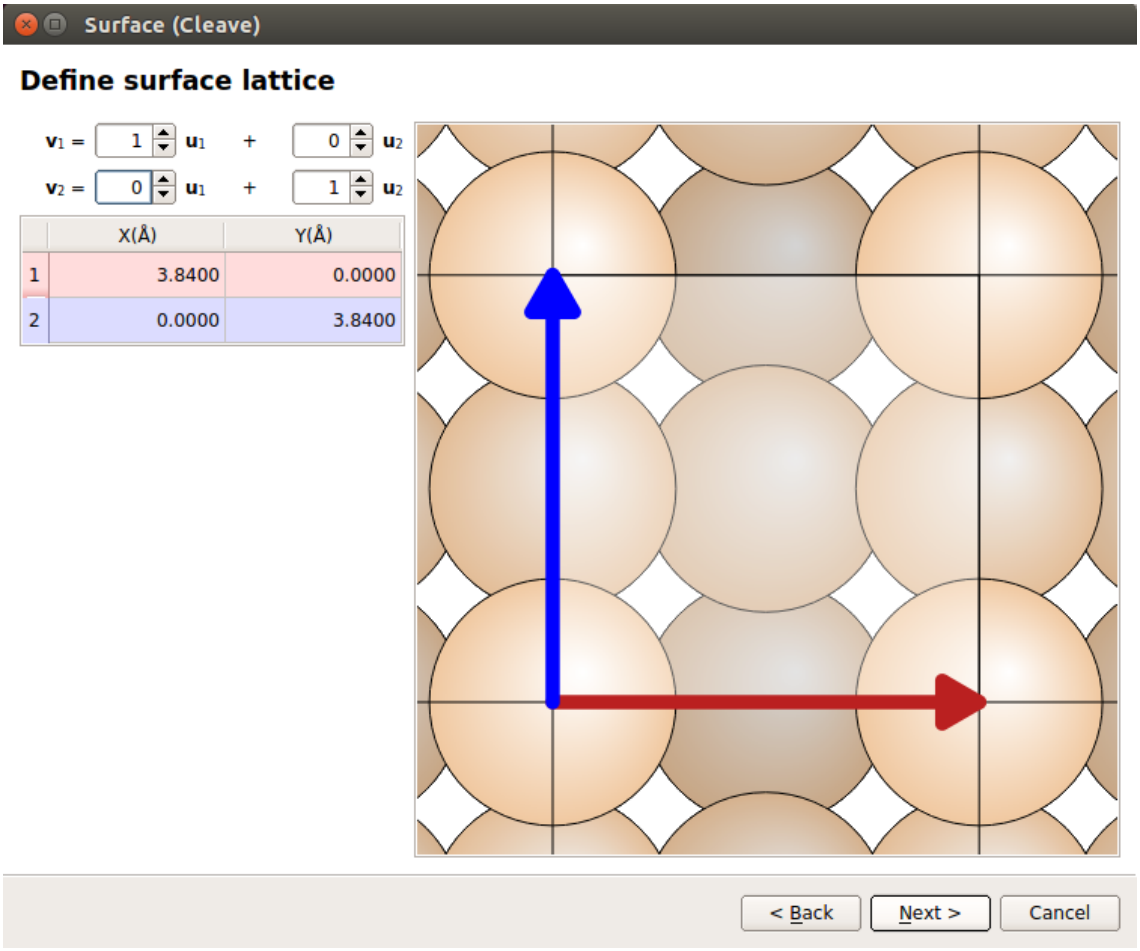
1. Open the **Builder**, highlight *Silicon (alpha)* in the **Stash**, and start creating a Si(100) surface:

- Navigate to *Builders* → *Surface (Cleave)*.
- In the **Surface (Cleave)** window, choose the [1,0,0] Miller indices and click *Next*.

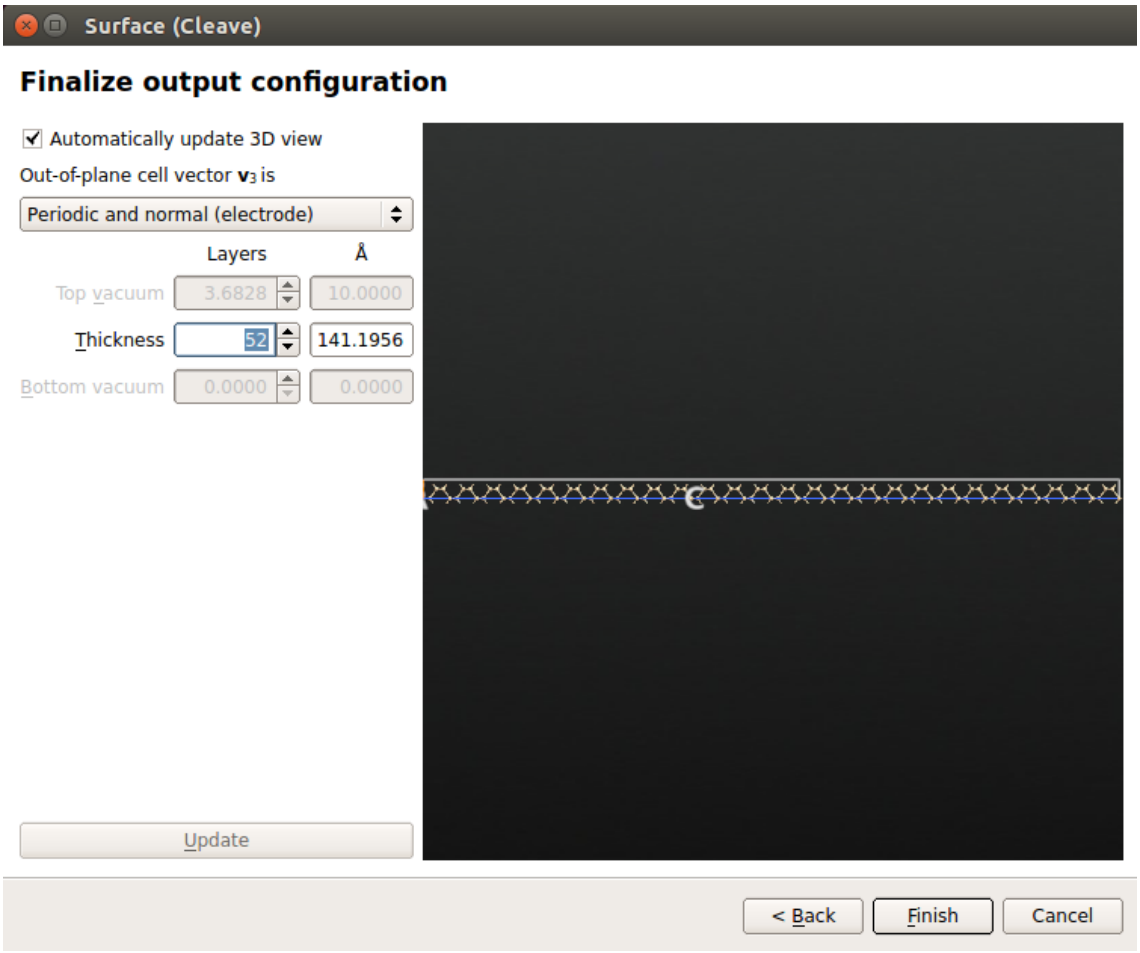




2.Keep the 1×1 surface unit cell and click *Next*.

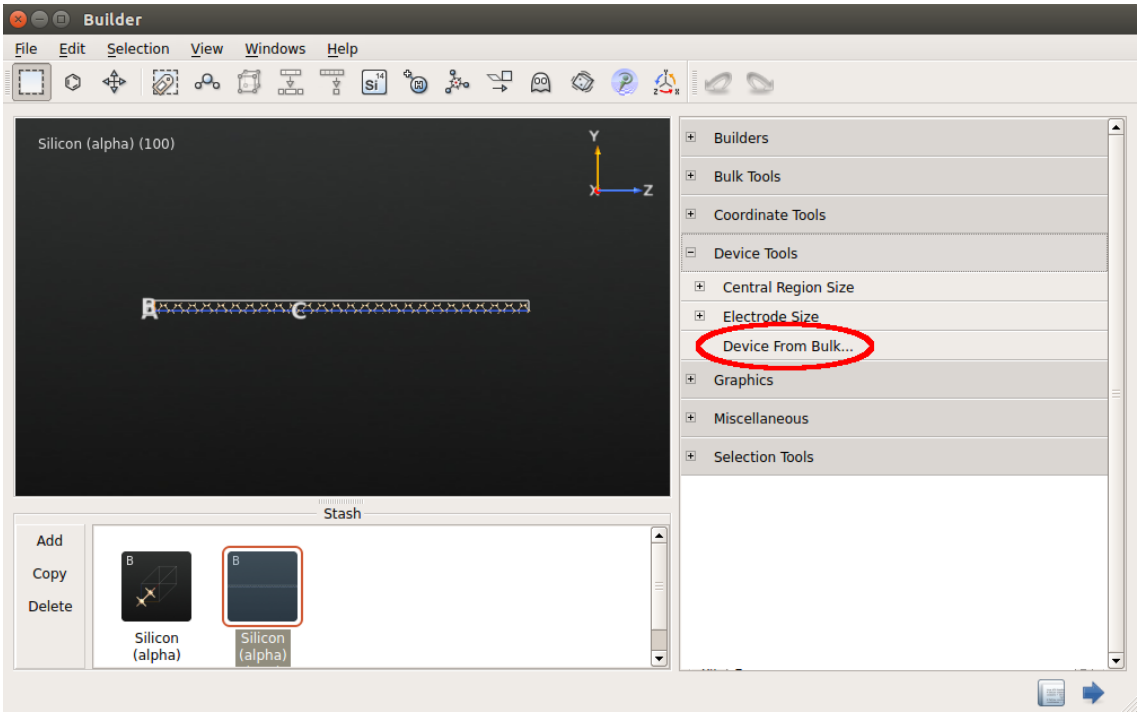


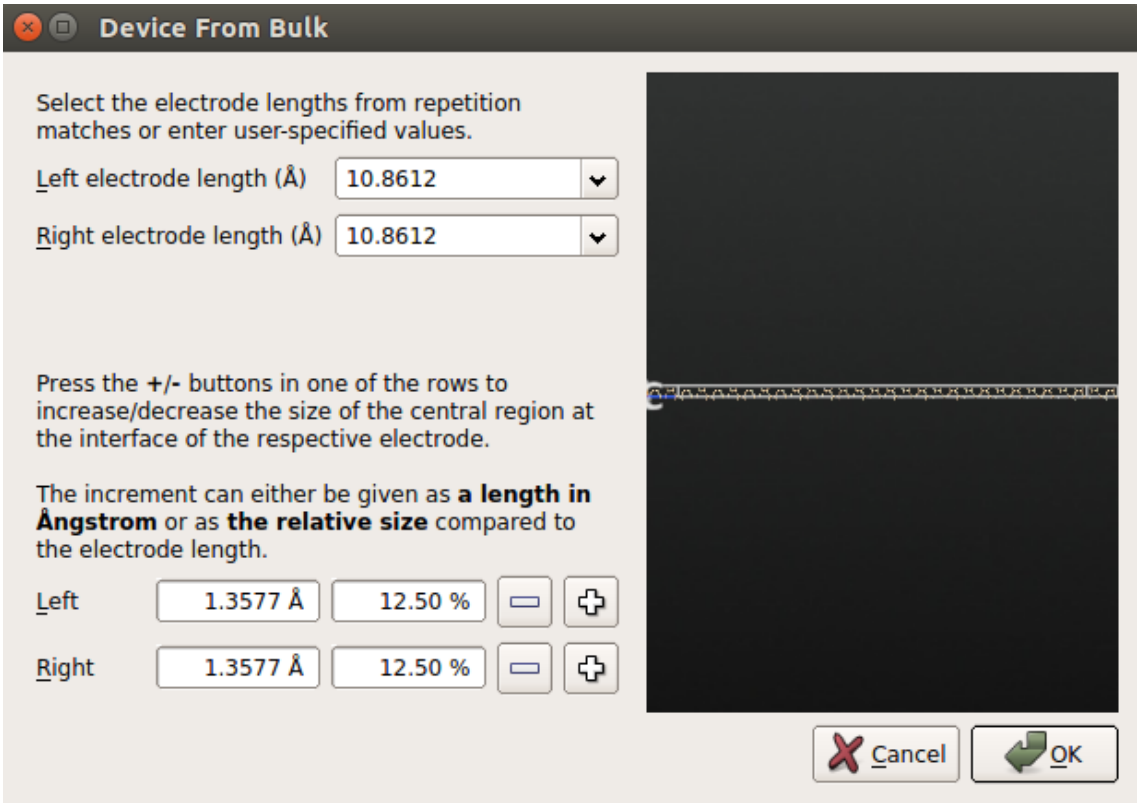
3.Select a thickness of 52 layers and click *Finish*.



Note: The 52 layers will constitute the central region of your device. You need such a relatively long device (~14 nm) because of the long screening length in the silicon semiconductor, as explained in the *NiSi2–Si interface* tutorial. Still, as you will see later, this device is not long enough to have perfectly converged results. However, you will use the 52 layers device throughout this tutorial to be able to perform the TB09-MGGA calculations faster.

4. Your Si(100) configuration is now constructed, and you will use it to construct a device with the transport direction along (100). Navigate to *Device Tools* → *Device From Bulk*, keep the default parameters, and simply click *OK*.





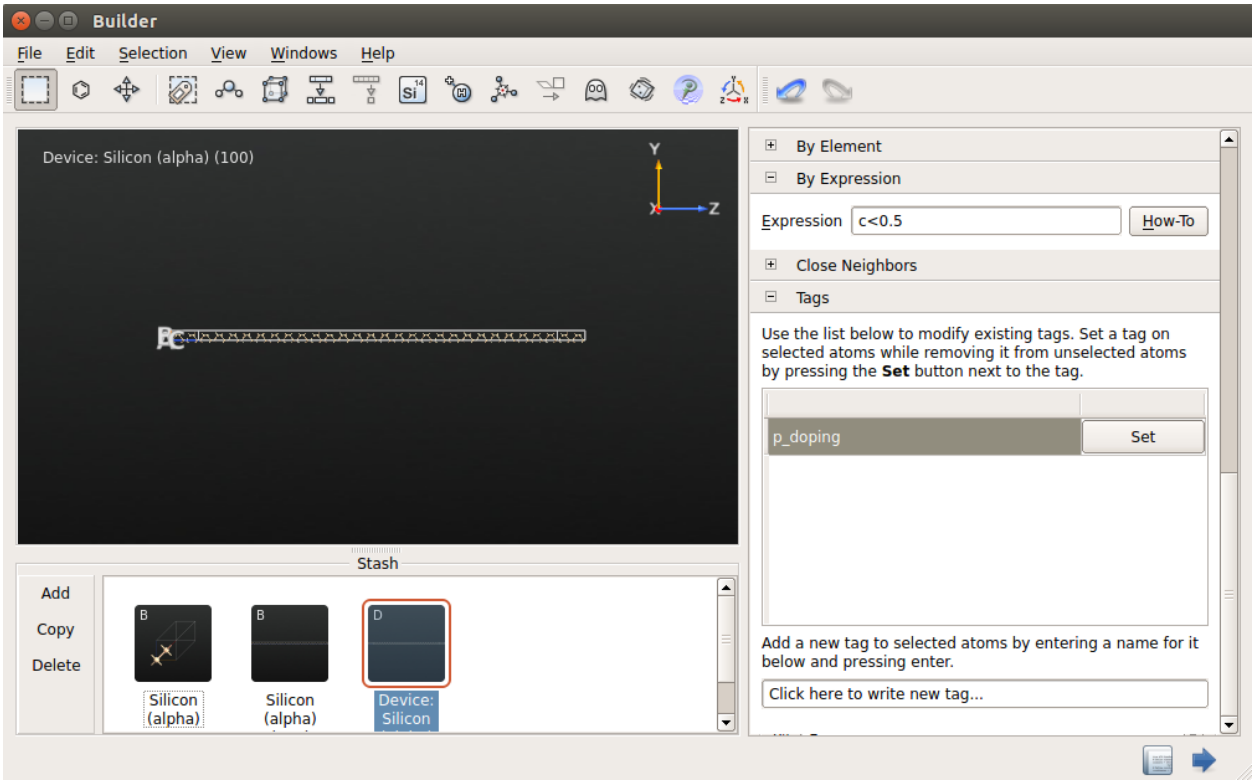
5.It will prove convenient to assign tags to atoms in the regions that should be doped p-type and n-type, respectively.

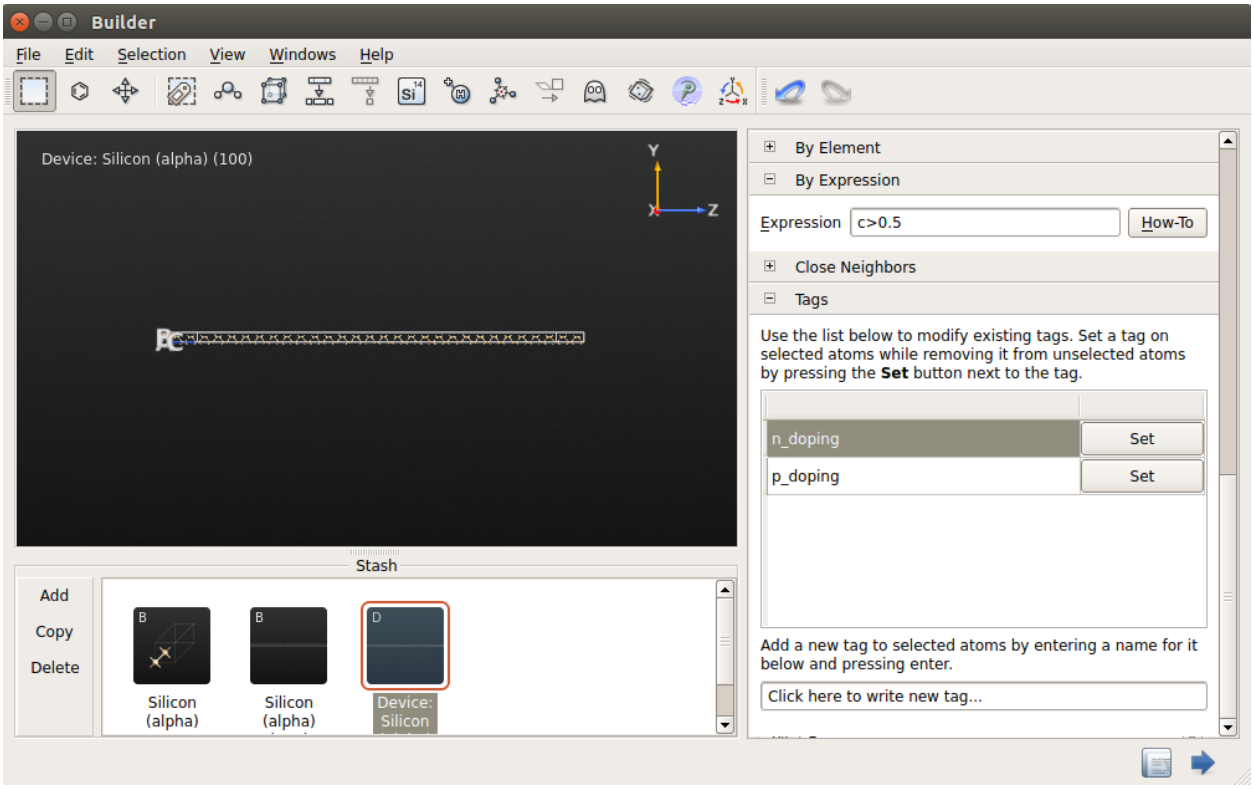
•Highlight the device configuration in the **Stash** and go to *Selection Tools* → *By Expression*.

•Type in $c < 0.5$ and press *Enter* to select all atoms in the left hand side of the device.

•Use *Selection Tools* → *Tags* to assign the tag *p_doping* to the selected atoms.

•Follow the same procedure to select atoms with $c > 0.5$ and assign the tag *n_doping* to those atoms.





Slater-Koster IV curve

You will use the Slater-Koster model to calculate the I-V characteristics for the p-n doped silicon junction.

1. Send the device configuration from the **Builder** to the **Script Generator**.

2. Add a **New Calculator** to the script and set the following parameters:

   - Select the *ATK-SE Slater-Koster* calculator.
   - Use a 7x7x100 k-point sampling.
   - Select the *Bassani.Si basis set* and uncheck the *No SCF iteration* box.

3. You can now add all the analysis objects needed. Add the following ones:

   - **ElectronDensity**
   - **DeviceDensityofStates**
   - **ElectrostaticDifferencePotentials**
   - **IVCurve**

4. Edit the **DeviceDensityofStates** parameters:

   - Increase considerably the k-point sampling. Use a 21x21 grid.

•Energy: from -2 to 2 eV with 401 points.

5.Edit the **IVCurve** parameters:

•*Voltage Bias*: sample from -1 V to 1 V with 21 points.

•*Energy*: from -1.1 eV to 1.1 eV with 401 points.

•*k-point sampling*: 21x21.

Note: A full explanation of all the parameters is given in the *Doping in QuantumATK: the Ni silicide - :ref:`nisi2-si* and in the ATK Reference Manual.

6.Change the output filename to something like `IV_2e19_SK.hdf5`.

Doping the silicon junction

You need to set up the proper doping to finally have a p-n junction. To this end, send the script to the **Editor** and locate the part of the script that defines the device configuration:

```
device_configuration = DeviceConfiguration(
central_region,
[left_electrode, right_electrode]
)
```

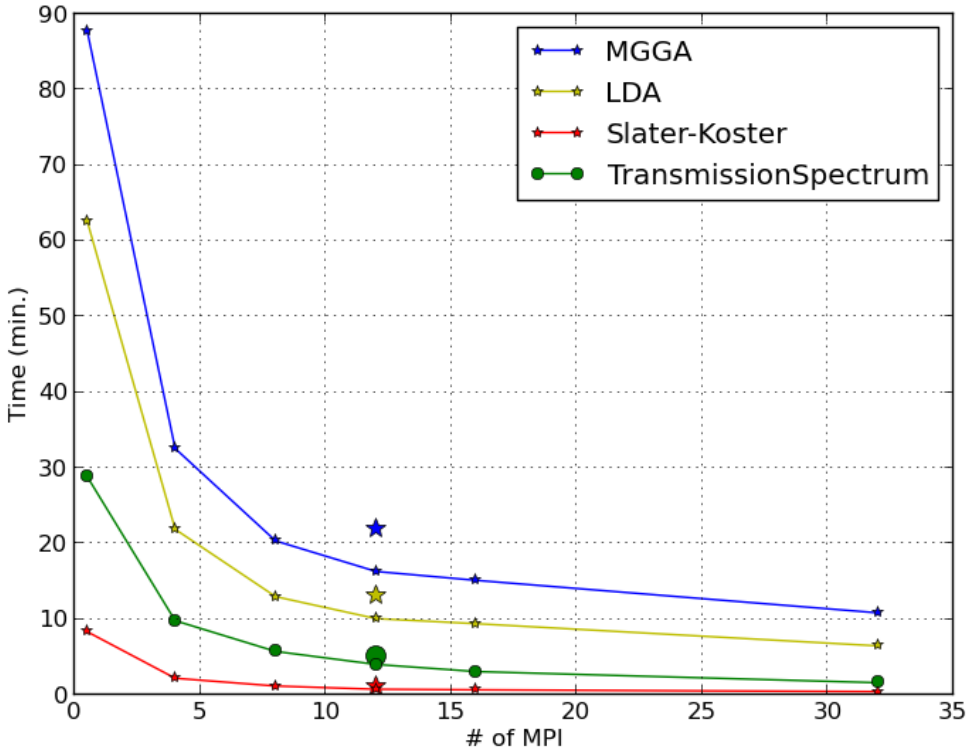
Just before those lines, add the following lines to define a p-n junction with a doping concentration of $2 \cdot 10^{19}$ cm^{-3}:

```
# ------------------------------------------------------------
# Add Doping
# ------------------------------------------------------------
doping_density = 2e+19
# Calculate the volume  and convert it to cm^-3
# Note: right and left electrodes have the same volume,
volume = right_electrode_lattice.unitCellVolume()
volume = float(volume/(0.01*Meter)**3)
# Calculate charge per atom
doping = doping_density * volume / len(right_electrode_elements)
# Add p- and n-type doping to the central region
external_potential = AtomicCompensationCharge([
     ('p_doping', -1*doping), ('n_doping',doping)
     ])
central_region.setExternalPotential(external_potential)
# Add doping to left and right electrodes
left_electrode.setExternalPotential(
   AtomicCompensationCharge([(Silicon, -1*doping)]))
right_electrode.setExternalPotential(
   AtomicCompensationCharge([(Silicon, doping)]))
```

Running the calculation

Send the script to the **Job Manager** and run the calculation.

Depending on the analysis objects included in the script, and on the number og parallel processes executed by the **Job Manager**, the calculation may take a few minutes for the zero bias part and up to few hours to calculate the whole IV curve. The performance of **ATK 2014** for different methods (**ATK-SE** and **ATK-DFT**) and for a transmission spectrum calculation are reported in the figure below.



Note: The calculations are run in parallel on different Intel Xeon e5472 3.0 GHz machines. The single points at 12 MPI are run with QuantumATK 13.8. Overall, QuantumATK 2014 is 25-30% faster compared to QuantumATK 13.8.

Calculation details: zero-bias, 7x7x100 k-point sampling for SCF part and 21x21 for the transmission spectra. All other parameters are set as in this tutorial.

Note that the Slater-Koster SCF calculation is much faster than the post-SCF calculation of the transmission spectrum.

DFT-MGGA IV curve

While the Slater-Koster job is running you can set up the DFT-MGGA calculation. Simply go back to the **Script Generator** and change the **Calculator** block:

- Choose *ATK-DFT*.
- Exchange correlation: *MGGA*.
- *SingleZetaPolarized* basis set.

Remove all analysis objects except the **IVCurve**. Then set up the device doping and run the calculation.

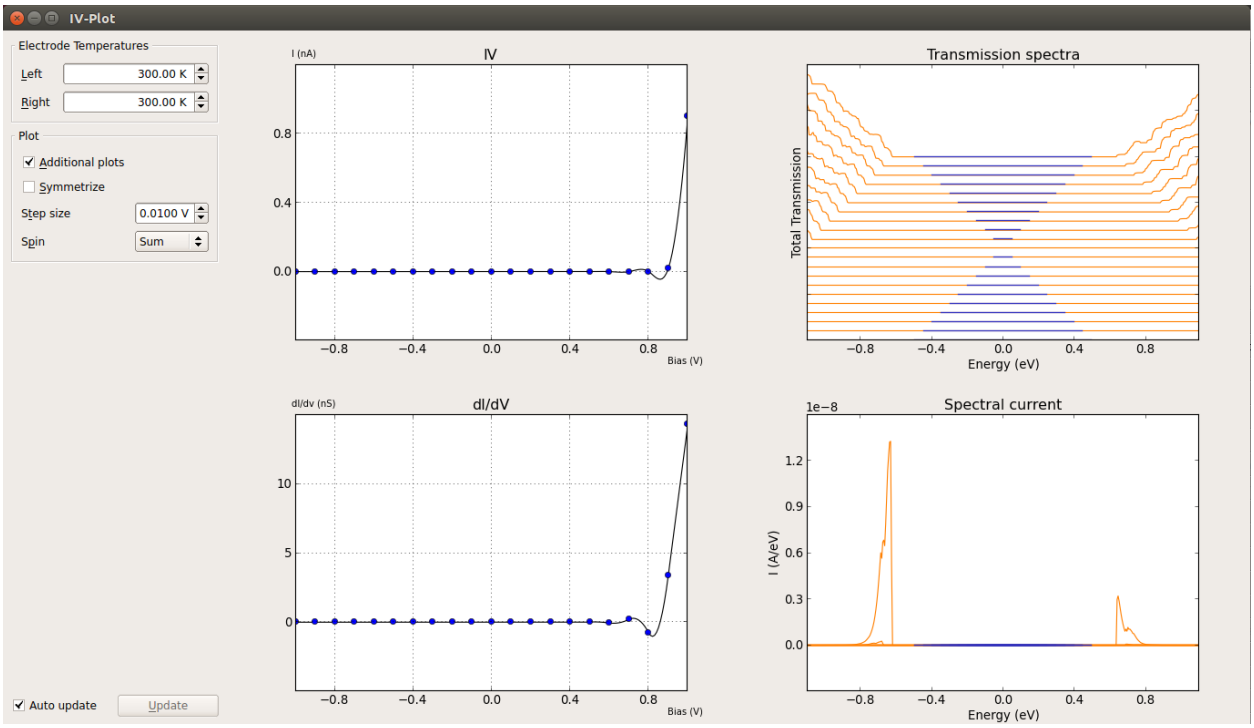
Analyzing the results

IV Curve

Once the two calculations are done and you have the output nc files in the project directory you will see the two **IVCurve** objects loaded in the **LabFloor**:

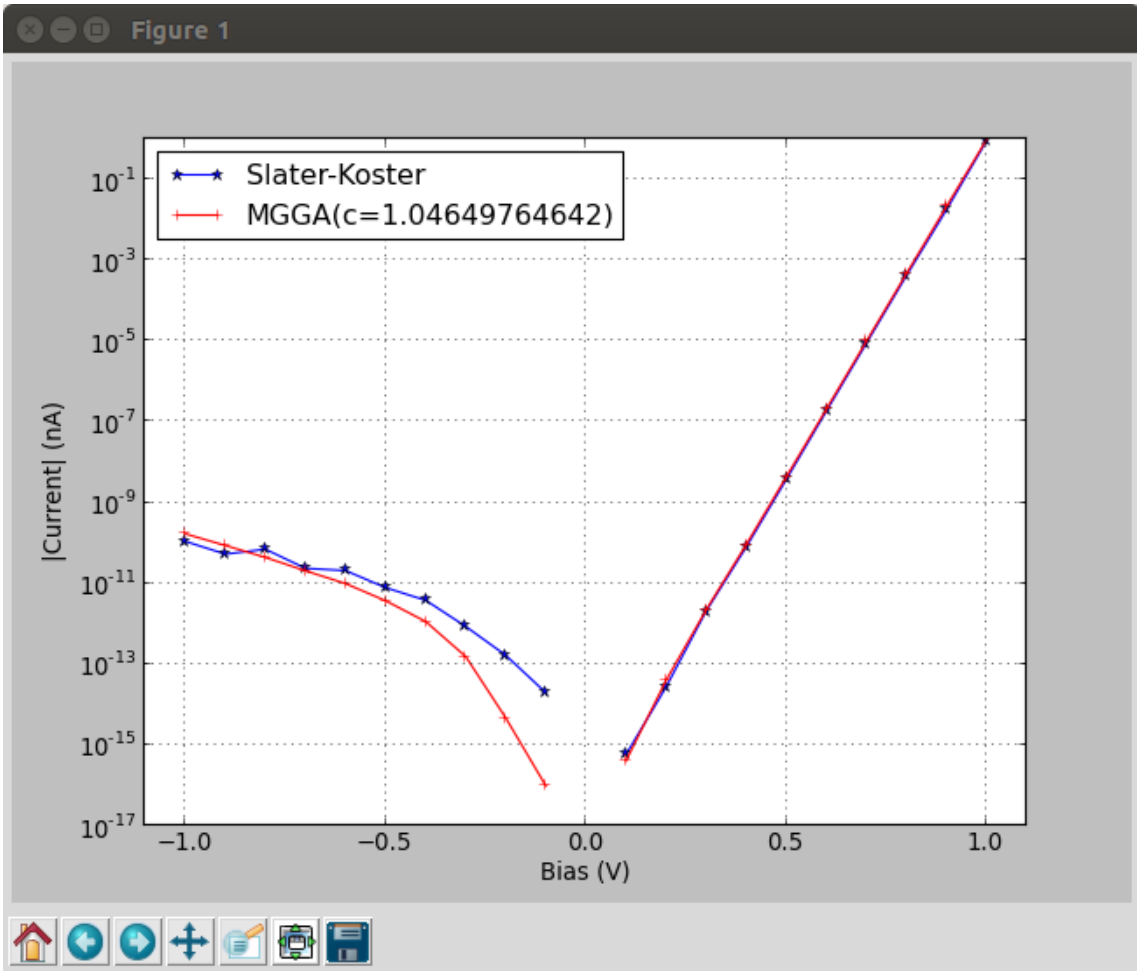
$$I\,(V)$$

1. Select one of the **IVCurve** objects and use the **IV-Plot** plugin to plot the IV curve. If you check the *Additional plots* option, you will also see plots of dI/dV, transmission spectra, and the spectral current.



2. In order to compare the **Slater-Koster** and **MGGA** IV curves, and send the script to the **Job Manager** to create

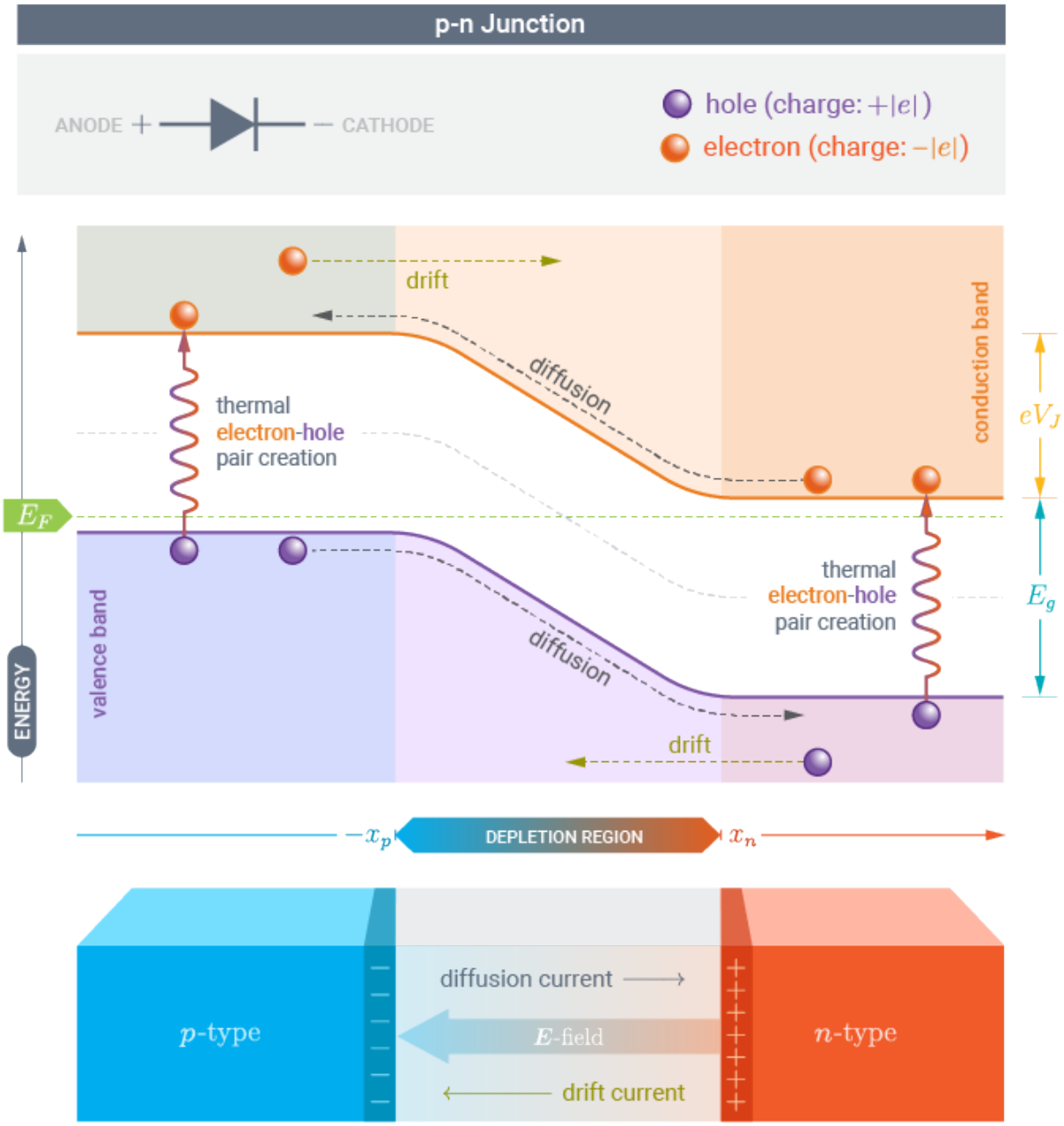
the figure below, where you can compare directly the two IV curves. You can also open the script with the **Editor** and modify it to tune the plot to your best convenience.



Attention: From the plot above you can see that the results from the semiempirical **Slater-Koster** model and the **DFT-MGGA** method are virtually the same.

Device Density of States

A classical picture of the electronic structure of a p-n junction is the one reported below, where the potential energy is plotted as a function of position along the transport direction of the device.



Using **QuantumATK** and **ATK** it is possible to perform this analysis through the **DeviceDensityOfStates** or the **LocalDensityOfStates** analysis objects.

Attention: Here, you will analyze the **DeviceDensityOfStates** and plot it along the device transport direction using a script.

In order to run the script and make the plots shown below you need an `.hdf5` file containing the analysis objects **DeviceConfiguration**, **DeviceDensityOfStates** and **ElectrostaticDifferencePotential**. You can open the script with the **Editor** and customize the plot according to your wish.

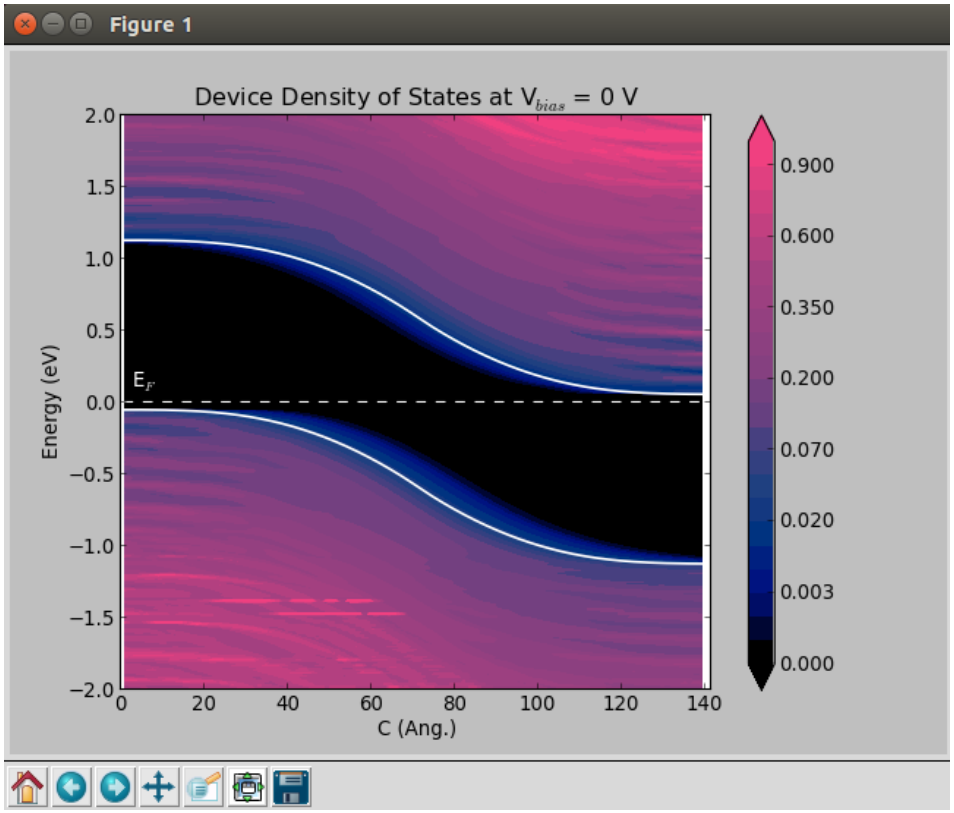
In particular, the `ddos_edp.py` script will read:

- **DeviceConfiguration**: to determine the spatial resolution along the device by defining a projection list used to plot the DDOS. Also, the electrode voltages are extracted from this object.
- **DeviceDensityOfStates**: besides reading and plotting the projected density of states, this object is also used to exctract the band edges at the left electrode. These informations are printed and used to align the electrostatic difference potential plots.
- **ElectrostaticDifferencePotential**: to plot the average electrostatic difference potential across the device.

In order to get the **DeviceDensityOfStates** plot without an applied bias, run the script from a terminal:

```
atkpython ddos_edp.py IV_2e19_SK.hdf5
```

where `IV_2e19_SK.hdf5` contains the analysis objects described above. You can also create plots for finite bias. See the section *Finite-bias calculations*. Results are illustrated below.



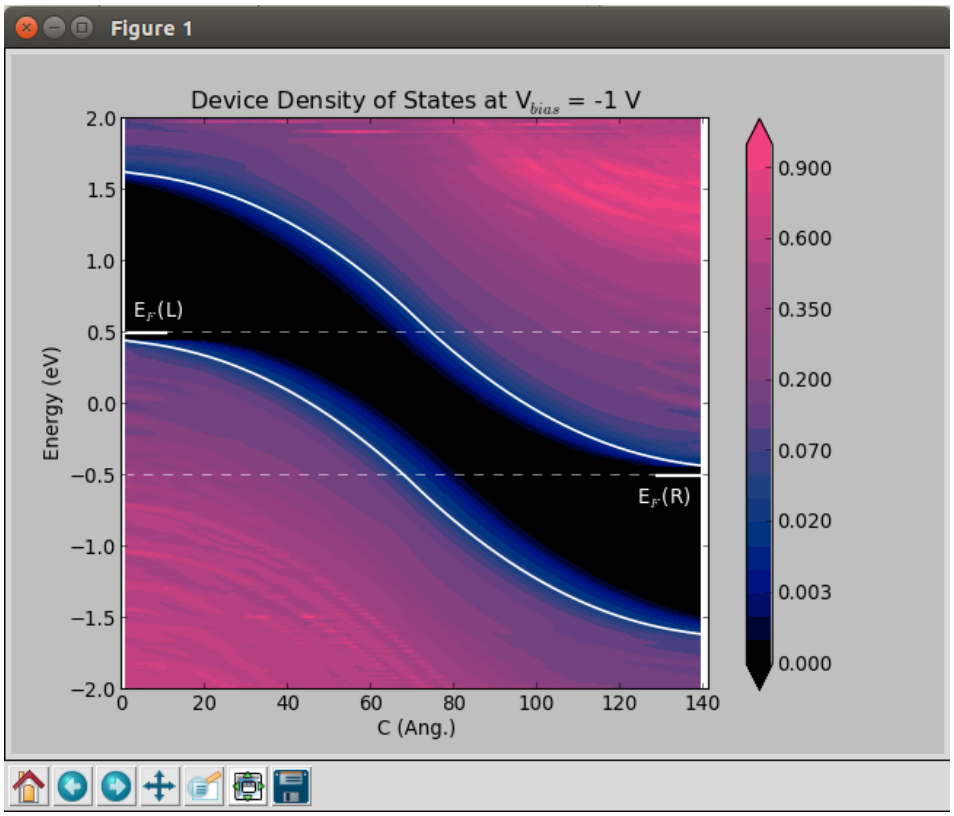


Figure 1
Device Density of States at V_{bias} = -1 V
Energy (eV)
C (Ang.)
E_F(L)
E_F(R)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
20
40
60
80
100
120
140
0.900
0.600
0.350
0.200
0.070
0.020
0.003
0.000

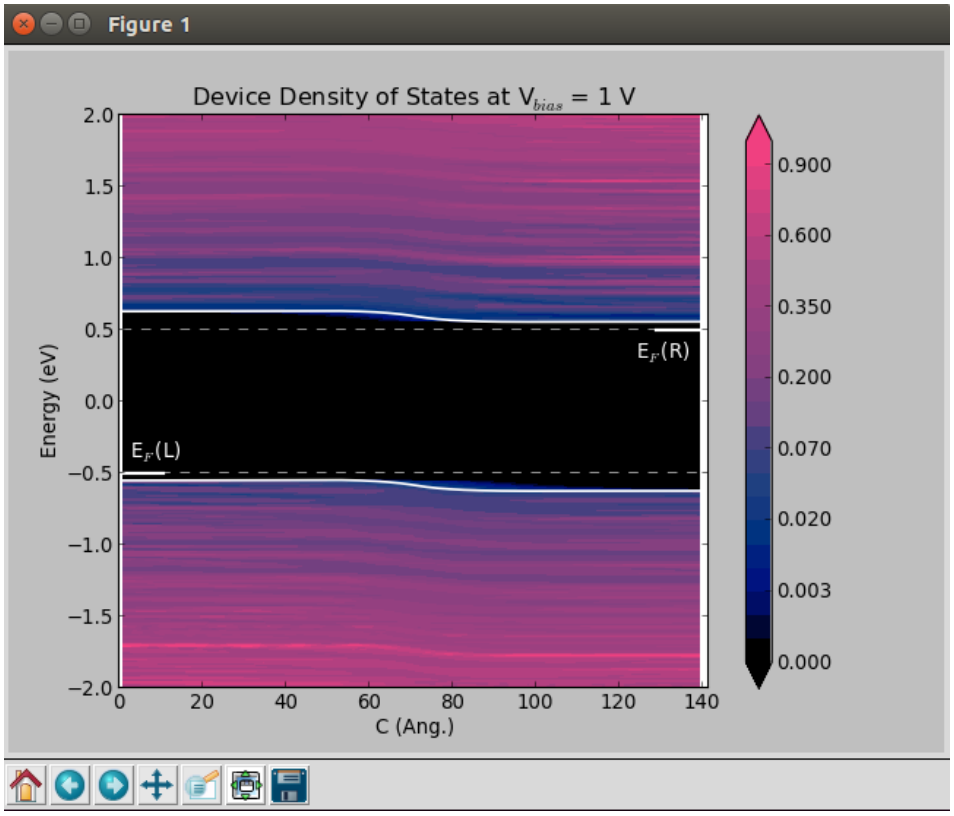
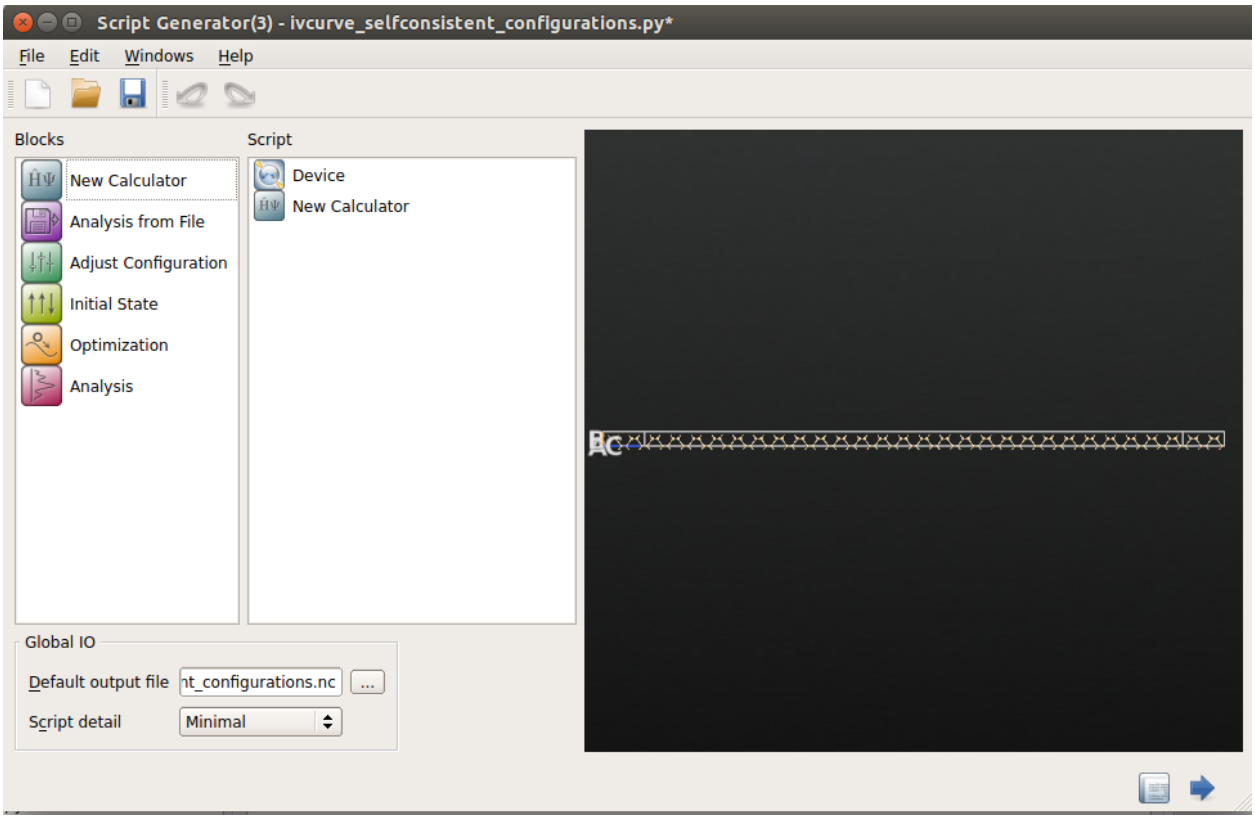


Figure 1
Device Density of States at V_{bias} = 1 V
Energy (eV)
C (Ang.)
E_F(R)
E_F(L)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
20
40
60
80
100
120
140
0.900
0.600
0.350
0.200
0.070
0.020
0.003
0.000

Finite-bias calculations

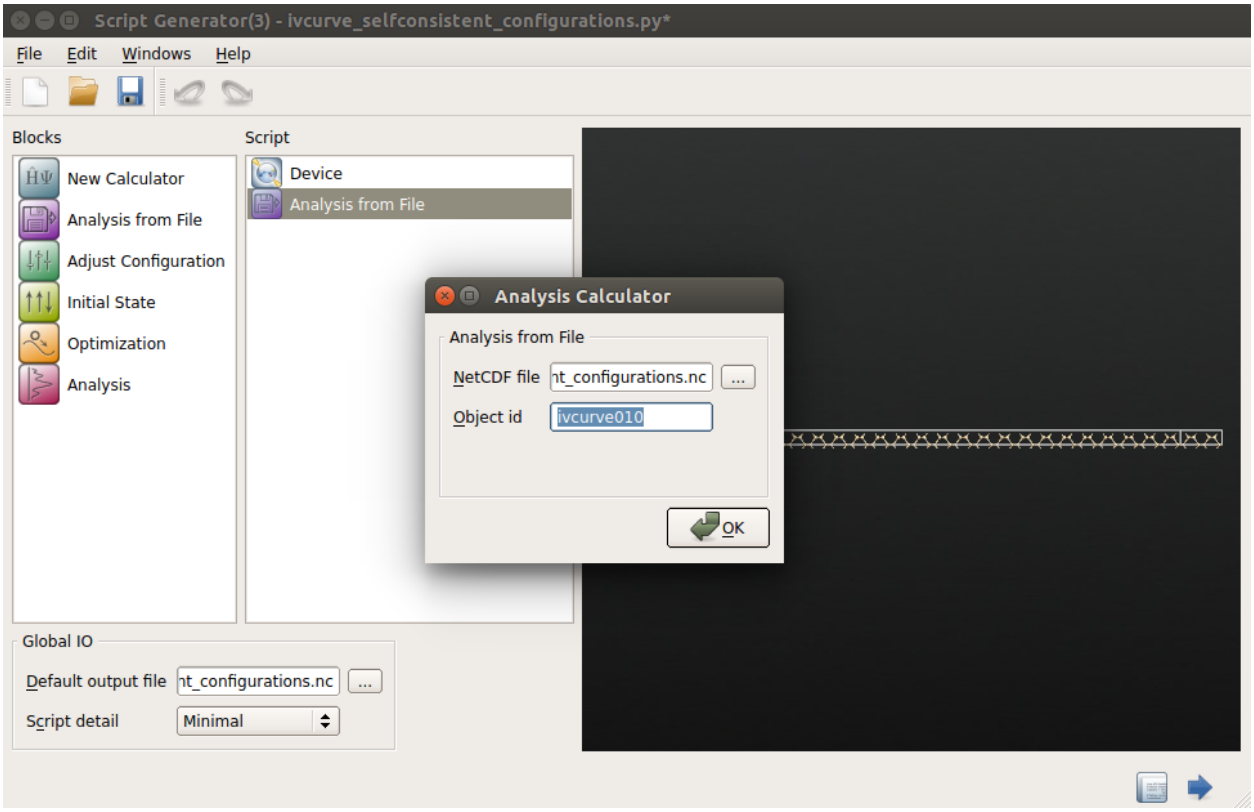
In order to create the finite-bias plots (V_{bias} = -1 and 1 V), do as follows:

1. Go to the **Labfloor** and from the `ivcurve_selftconsistent_configuration.hdf5` file select the device configuration obtained at V_{bias} = -1 V, and drag and drop it onto the **Scripter**. The following window will show up.



2. Remove the **New Calculator** object and add instead **Analysis from File**.

3. In **Analysis from File**, set the *HDF5* file and the *Object id* of the device configuration object.



4. Then, add the **Electrondensity**, **DeviceDensityOfStates** and **ElectrostaticDifferencePotential** analysis objects and set the parameters as described above.

5. Change the output file to `Analysis_ivcurve_10.hdf5`.

6. Before running the job, send the script to the **Editor**, and add the following line to get the *device_configuration* file within the `Analysis_ivcurve_10.hdf5` output file.

```
# ------------------------------------------------------------
# Analysis from File
# ------------------------------------------------------------
device_configuration = nlread('ivcurve_selfconsistent_configurations.hdf5', object_id=
↪'ivcurve010')[0]
```

7. After this part and just before **ElectronDensity**, add the following line:

```
nlsave('ivcurve_selfconsistent_configurations.hdf5', device_configuration)
```

8. Lastly, save the script and run the job. Once the job have finished, run the script `ddos_edp.py` in order to get the *Device Density of States* plot. Run the script as follows:

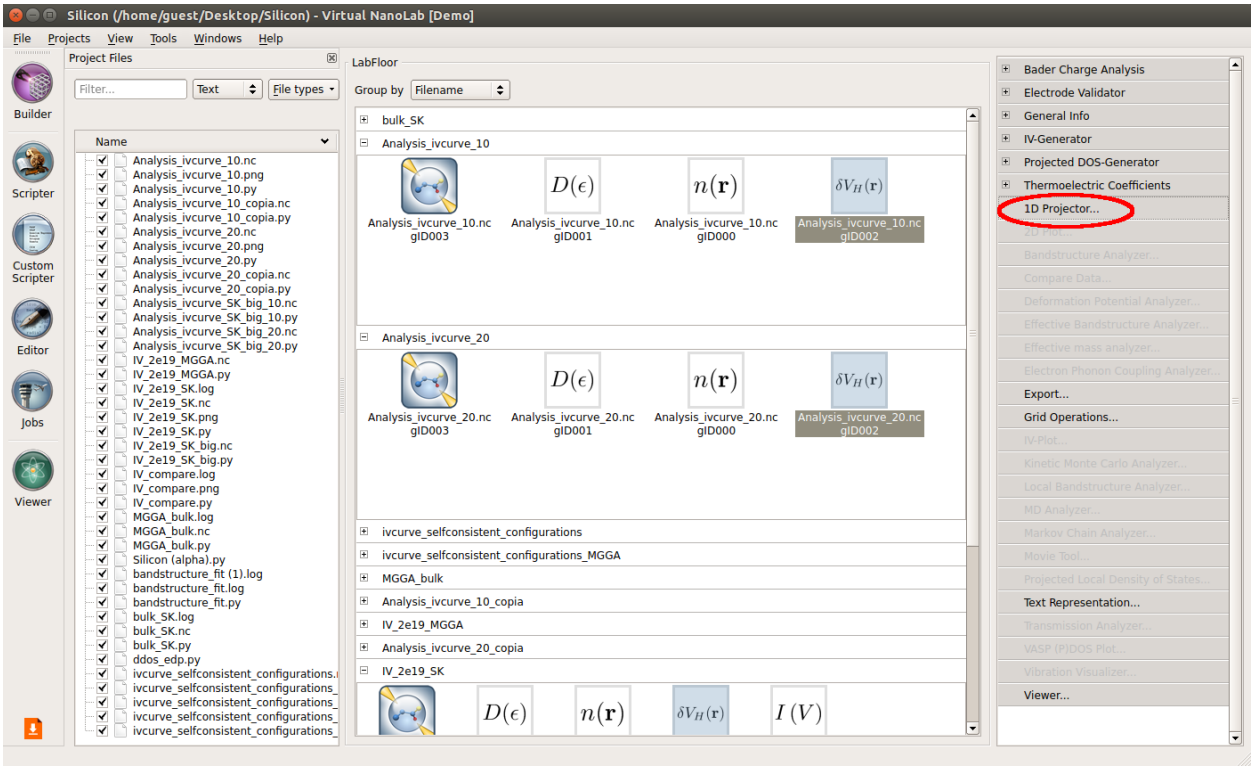
```
atkpython ddos_edp.py Analysis_ivcurve_10.hdf5
```

You can plot the *Device Density of States* plot following the same procedure for the `ivcurve_selftconsistent_configuration.hdf5ivcurve020` file.

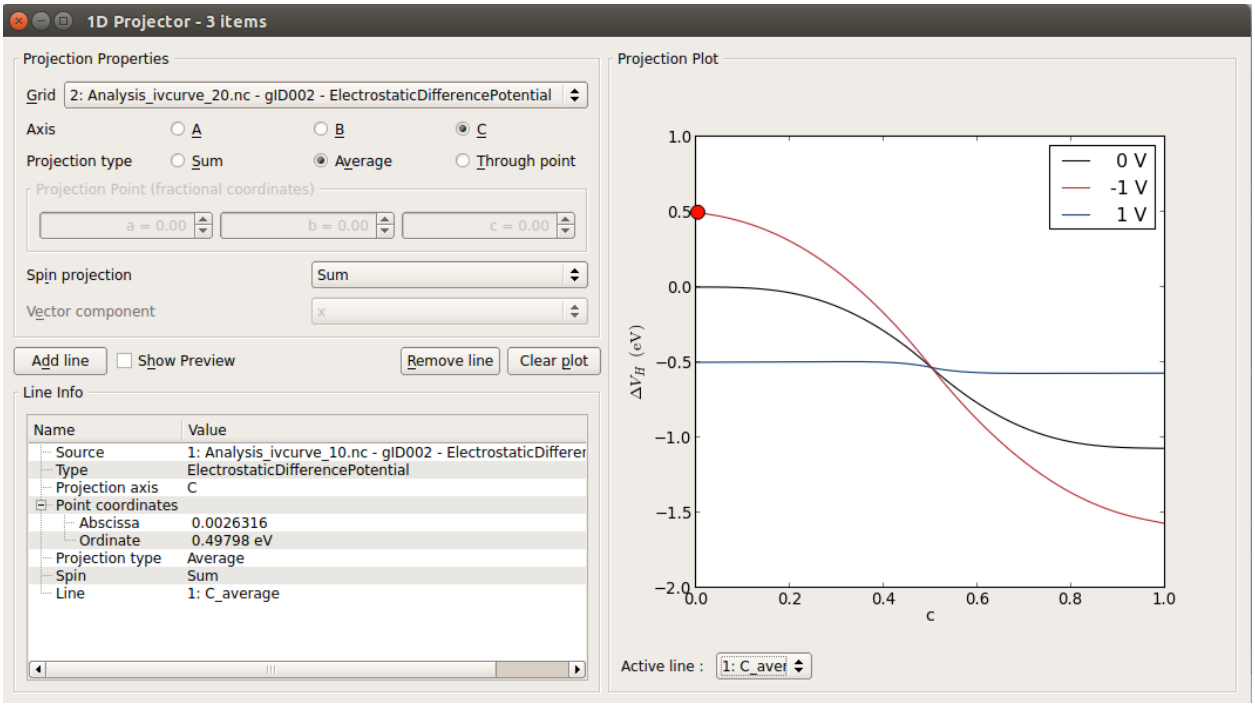
Electrostatic potential

$\delta V_{\mathbf{E}}(\mathbf{r})$

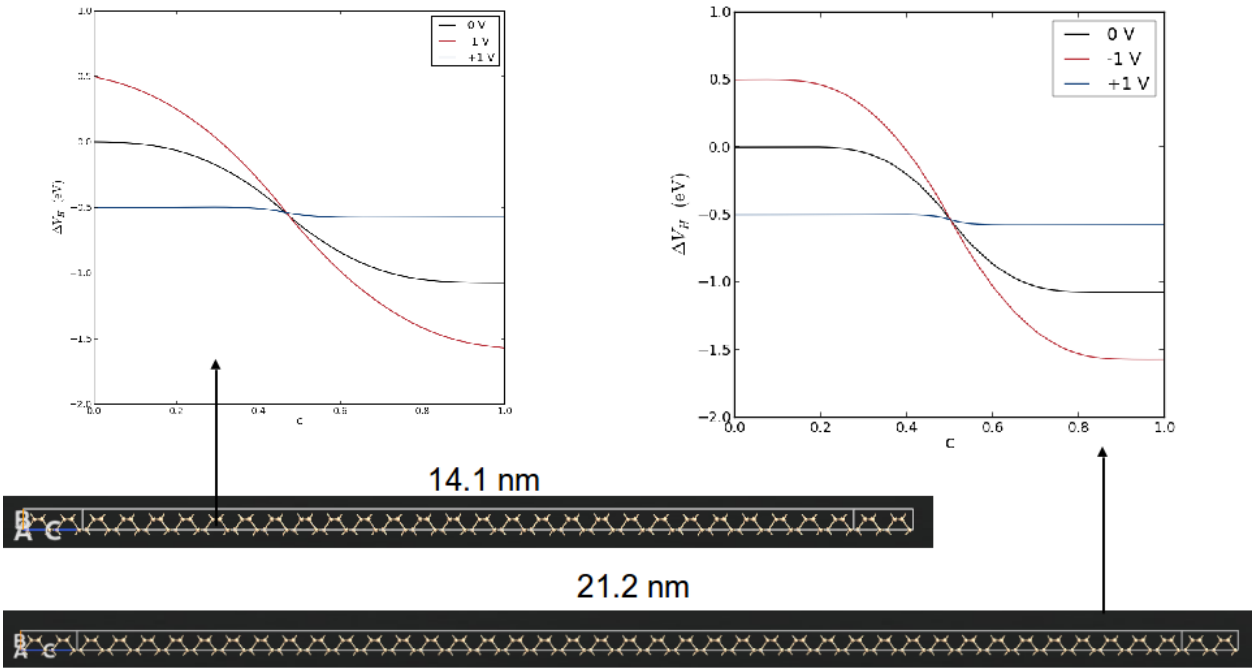
1.In the **Labfloor**, select the **ElectrostaticDifferencePotential** objects obtained for V_{bias} = 0, -1, 1,



2.Use the **1D Projector** plugin to plot the data.



You can see that the results are not perfectly converged wrt. the length of the device, because the gradient of the electrostatic potential is not zero at the ends of the device central region. The figure below compares the **ElectrostaticDifferencePotential** of the device used in this tutorial with results for a longer one.



References

3.2.5 Optical Properties of Silicon

Introduction

The purpose of this tutorial is to show how **ATK** can be used to compute accurate electronic structure, optical, and dielectric properties of semiconductors from DFT combined with the meta-GGA functional by Tran and Blaha[1] (TB09). TB09 is a semi-empirical functional that is fitted to give a good description of the band gaps in non-metals. The results obtained with the method are often comparable with very advanced many-body calculations, however with a computational expense comparable with LDA, i.e. several order of magnitudes faster. Thus, the TB09 meta-GGA functional is a very practical tool for obtaining a good description of the electronic structure of insulators and semiconductors.

It is important to note that the TB09 functional does not provide accurate total energies[Page 515, 1], and it can therefore only be used for calculating the electronic structure of the materials, while the GGA-PBE functional should be used for computing total energies and atomic geometries.

It is assumed that you are familiar with the general workflow of **QuantumATK**, as described in the Basic QuantumATK Tutorial.

This tutorial uses silicon as an example. As for most semiconductors, GGA/LDA both severely underestimate the Si band gap (between 0.5 and 0.6 eV), while the experimental band gap of 1.18 eV is reproduced almost exactly by TB09. The experimental dielectric constant is also reproduced by the optical properties module in **ATK** to within a few percent.

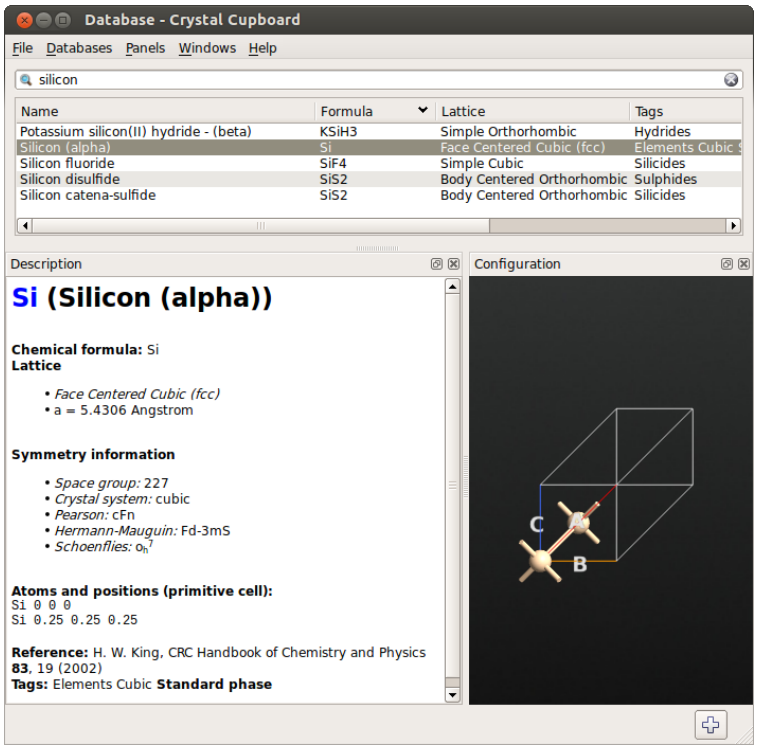
Electronic structure and optical properties of silicon

Setting up the calculation

Start **QuantumATK**, create a new project and give it a name, then select it and click Open. Launch the Builder by pressing the icon on the toolbar.

In the builder, click *Add* → *From Database*. Type silicon in the search field, and select the silicon standard phase in the list of matches. Information about the lattice, including its symmetries (e.g. that the selected crystal is face centered cubic), can be seen in the lower panel.

[1] F.Tran, P. Blaha, *Phys. Rev. Lett.*, **102**, 226401, 2009.

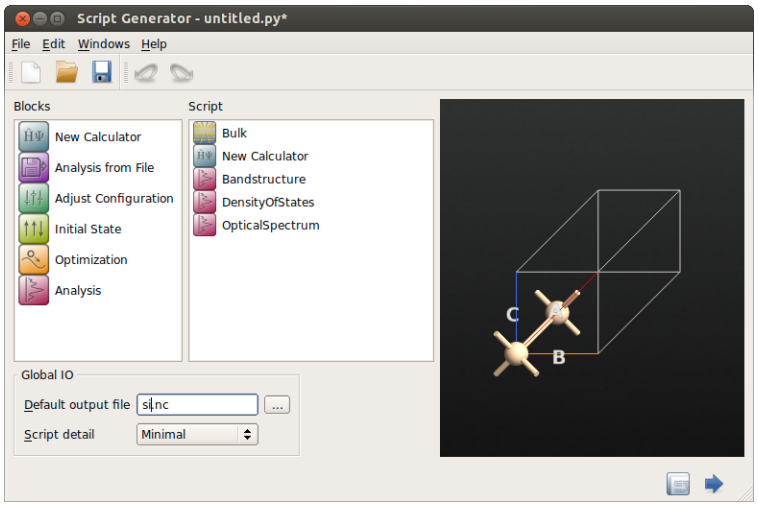


Double-click the line to add the structure to the Stash, or click the icon in the lower right-hand corner.

Now send the structure to the **Script Generator** by clicking the Send To icon in the lower right-hand corner of the window, and select **Script Generator** (the default choice, highlighted in **bold**) from the pop-up menu.

In the Script Generator,

- Add *New Calculator*.
- Add *Analysis* → *Bandstructure*.
- Add *Analysis* → *DensityOfStates*.
- Add *Analysis* → *OpticalSpectrum*.
- Change the output filename to `si.nc`.



The next step is to adjust the parameters of each block, by double-clicking them:

•Open the New Calculator block, and

–select the ATK-DFT calculator (selected by default),

–set the k-points to (4,4,4),

–select the exchange-correlation functional to MGGA,

–under Basis set/exchange correlation, set Pseudopotential to **HGH[Z=4] LDA.PZ**,

–and finally select the **Tier 3** basis set for Si.

–Close the dialogue by clicking OK in the lower right-hand corner

•Open the DensityOfStates block, and

–select 15 x 15 x 15 kpoints.

•Open the OpticalSpectrum block, and

–select 15 x 15 x 15 k-points,

–use 10 bands below and 20 above the Fermi level (this controls how many bands are included in the calculation of the optical matrix elements).

Note: For this calculation you will use the Hartwigsen, Goedecker, Hutter (HGH) pseudopotentials[2]. The calculation of optical properties requires a good description of virtual states far above the Fermi level. The Tier 3 basis set for Si consists of optimized 3s, 3p (2 orbitals), 3d, 4s orbitals. Going higher to Tier 4 would add another 3s orbital, and so on, but this appears to have no significant influence on the band gap (just taking longer time). With a smaller basis set, however, the band gap comes out incorrectly even with TB09-MGGA.

Save the script from the Editor for future reference.

2 C.Hartwigsen, S. Goedecker, J. Hutter, *Phys. Rev. B*, **58**, 3641, 1998.

Running and analyzing the calculation

Transfer the script to the **Job Manager** and start the calculation. The job will finish after a few minutes. The file `si.nc` should now be visible under Project Files in the main **QuantumATK** window. On the LabFloor select Group by Item Type.

DOS of silicon

Select the **DensityOfStates** (gID002), then click the **2D Plot** button on the plugin panel. From the density of states it is possible to determine the band gap of silicon.

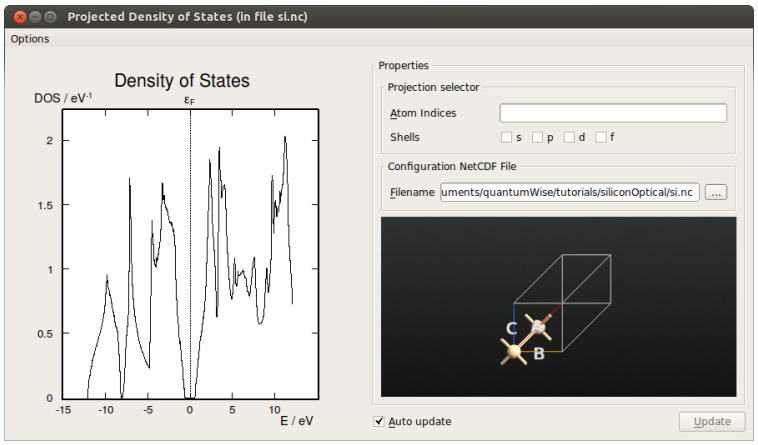


Fig. 3.25: Density of states (DOS) of Si, computed with meta-GGA.

To read off the band gap, you may zoom the plot or export the DOS data to a file (or simply select **Text Representation** instead of **2D Plot**). The band edges are located around -0.59 eV and 0.57 eV, resulting in a band gap of 1.16 eV. This is in excellent agreement with the experimental band gap of 1.17 eV (at 0 Kelvin), in contrast to the LDA band gap of 0.55 eV.

Optical spectrum

Next select the **OpticalSpectrum** (gID003), then click the **2D Plot** button on the plugin pane. The plot is shown below.

By zooming in (right-click on the plot), in the figure, you can determine the static dielectric constant, $\mathrm{Re}[\epsilon(\omega = 0)] = 10.9$, in qualitative agreement with the experimental value of 11.9 (with a bit larger basis set we can get values around 12.2, but also note that we did not optimize the k-point sampling).

The absorption coefficient and refractive index are related to the dielectric constant, see for instance the ATK Reference Manual.

The script below calculates the absorption coefficient and refractive index from the dielectric constant and plots it as a function of the wavelength.

```
# Load the optical spectrum
spectrum = nlread('si.nc', OpticalSpectrum)[-1]

# Get the energies range
energies = spectrum.energies()
```

(continues on next page)

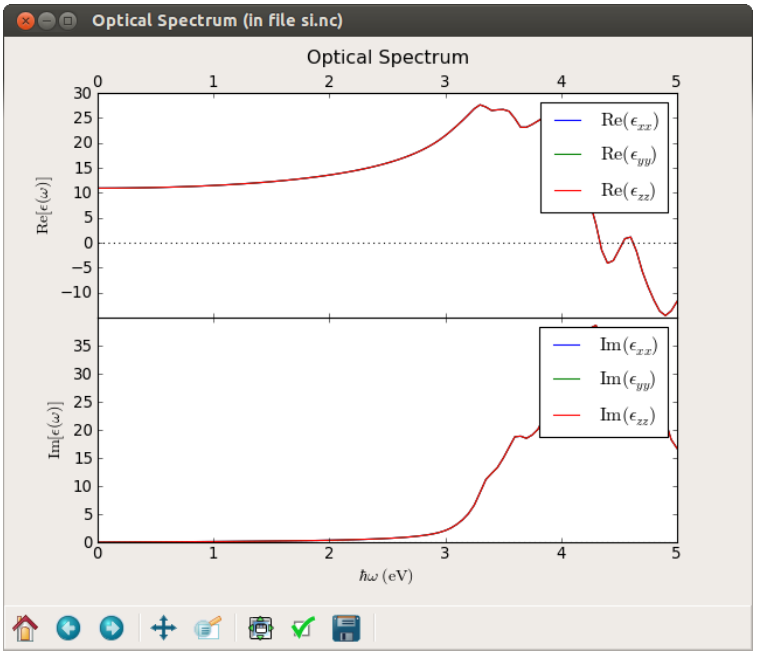


Fig. 3.26: Real and imaginary parts of the diagonal components of the dielectric constant. Since silicon has cubic symmetry the dielectric constant is isotropic, i.e. $\epsilon_{xx} = \epsilon_{yy} = \epsilon_{zz}$.

(continued from previous page)

```
# get the real and imaginary part of the e_xx component of the dielectric tensor
d_r = spectrum.evaluateDielectricConstant()[0, 0, :]
d_i = spectrum.evaluateImaginaryDielectricConstant()[0, 0, :]

# Calculate the wavelength
l = (speed_of_light*planck_constant/energies).inUnitsOf(nanoMeter)

# Calculate real and complex part of the refractive index
n = numpy.sqrt(0.5*(numpy.sqrt(d_r**2+d_i**2)+d_r))
k = numpy.sqrt(0.5*(numpy.sqrt(d_r**2+d_i**2)-d_r))

# Calculate the adsorption coefficient
alpha = (2*energies/hbar/speed_of_light*k).inUnitsOf(nanoMeter**-1)

# Plot the data
import pylab
pylab.figure()
pylab.subplots_adjust(hspace=0.0)
ax = pylab.subplot(211)
ax.plot(l, n, 'b', label='refractive index')
ax.axis([180, 1000, 2.2, 6.4])
ax.set_ylabel(r"$n$", size=16)
ax.tick_params(axis='x', labelbottom=False, labeltop=True)
ax = pylab.subplot(212)
ax.plot(l, alpha, 'r')
ax.axis([180, 1000, 0, 0.24])
ax.set_xlabel(r"$\lambda$ (nm)",  size=16)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r"$\alpha$ (1/nm)", size=16)
pylab.show()
```

Save the script to the project folder and execute the script by dropping it on the **Job Manager** icon on the toolbar.

The script will generate the figure below.

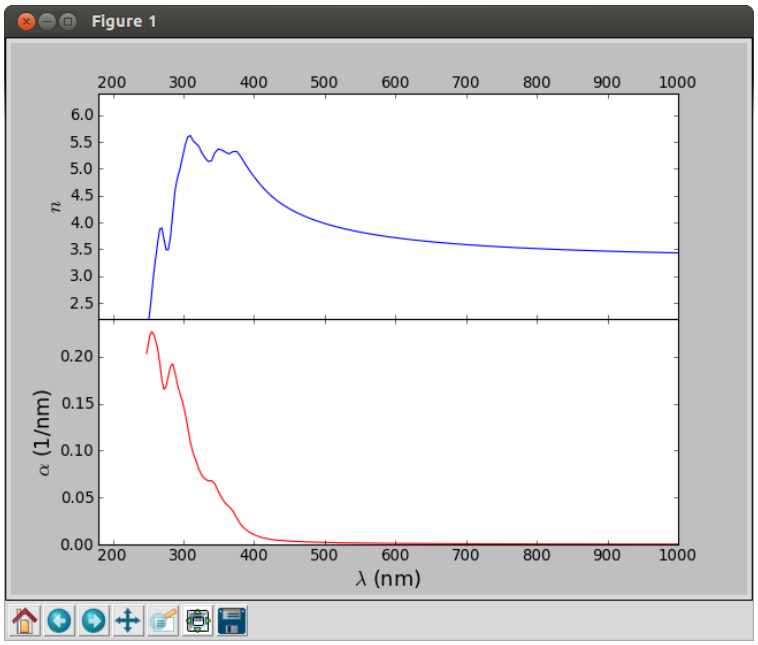


Fig. 3.27: Refractive index (blue) and adsorption coefficient (red) of silicon as function of the wavelength of the incoming light.

References

3.2.6 $NiSi_2$–Si interface

In this tutorial you will study a doped $NiSi_2$-Si interface and its electronic transport properties. In particular you will:

- create the device to calculate the transport properties of such interface;
- learn how to add the proper doping to simulate a real device;
- analyze the results in terms of electrostatic potentials, electron density and density of states, by using the tools included in **QuantumATK**.

Warning: If you are running this tutorial with QuantumATK v2017.2 or higher, be aware that you will have to set explicitly the electronic temperature to $T = 300$ K in all the electronic structure calculations performed for either a *BulkConfiguration* or for a *DeviceConfiguration*, in order to reproduce the results presented in the case study.

Create the $NiSi_2$/Si device

1. Create a new empty project and open the **Builder** (icon).

2. The first thing you have to do is to create a device configuration starting from the LDA-optimized $NiSi_2$-Si bulk configuration of the interface. The latter can be downloaded here: `Interface.py`.

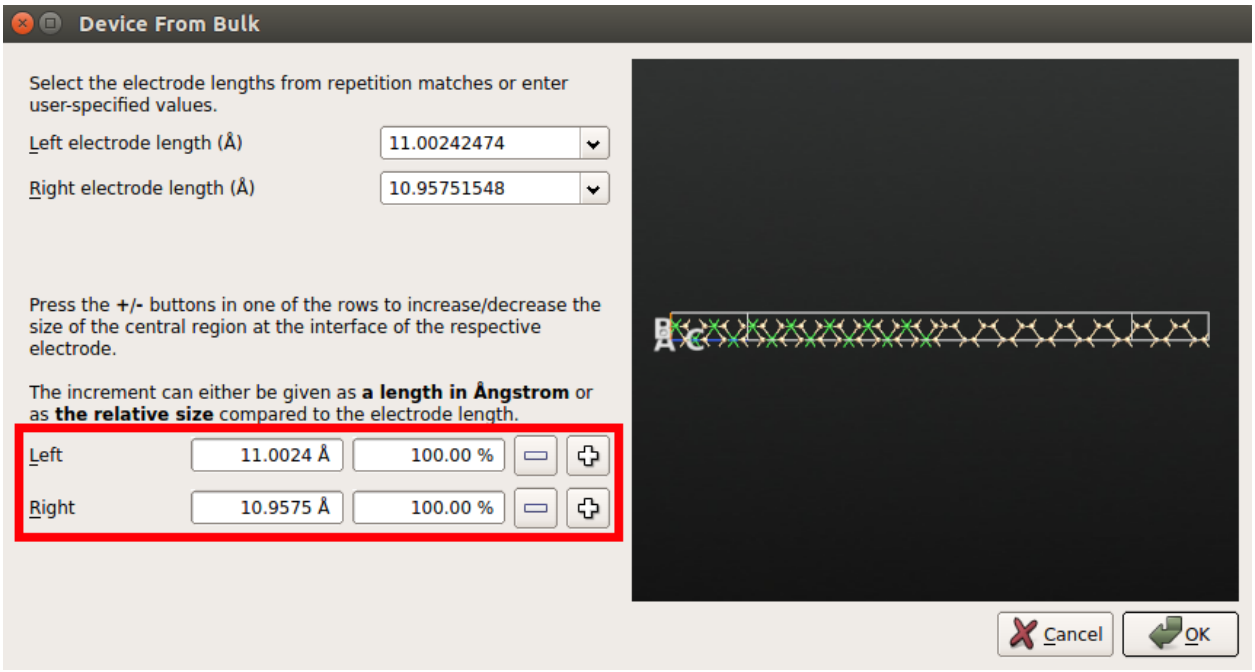
Note: You can learn how to create an interface from the tutorials *Building a Si-Si3N4 Interface*, *Graphene–Nickel interface* and *Building an interface between Ag(100) and Au(111)*.

---

3. Use *Add* → *From Files* to add the newly downloaded interface structure to the **Stash**.

4. Go to *Device Tools* → *Device From Bulk* to create the device configuration from the bulk configuration.

5. Do not modify the electrode lengths but increase the size of the central region on the left and right side by 100% compared to the size of left and right electrodes, respectively.



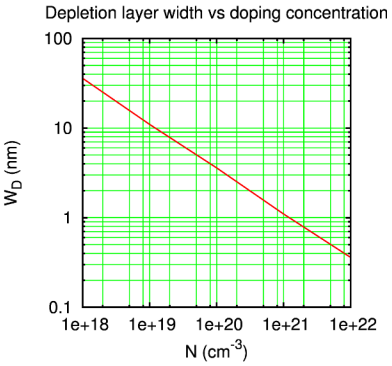
The screening region

What should be your doping concentration? In principle, you can use the experimental one. However, you need to be very careful about another quantity: the width of the depletion layer, usually denoted as W_D, which is related to the screening length of the semiconductor.

The width of the depletion layer depends on:

- the dielectric constant of the semiconductor;
- the doping concentration (number of carriers);
- the potential barrier at the interface;
- the applied voltage.

The plot below shows typical values of W_D for a metal-semiconductor junction (as reported in[1] pag. 119) as a function of the doping concentration N.



For the purpose of your simulations you need to ensure that the silicon part in the central region is larger than the width of depletion layer. Throughout this tutorial you will use doping concentrations ranging from 10^{19} cm^{-3} to 10^{20} cm^{-3}. This means that you will need a silicon region of about 30 Å to 100 Å wide. A good way to check if your system is large enough is to plot the Electrostatic Difference Potential along the device (see below).

Note: The screening length of a metal is much smaller that that of a semiconductor, only few atomic layers. Therefore, you can keep the $NiSi_2$ side much shorter than the Si side to save computational time.

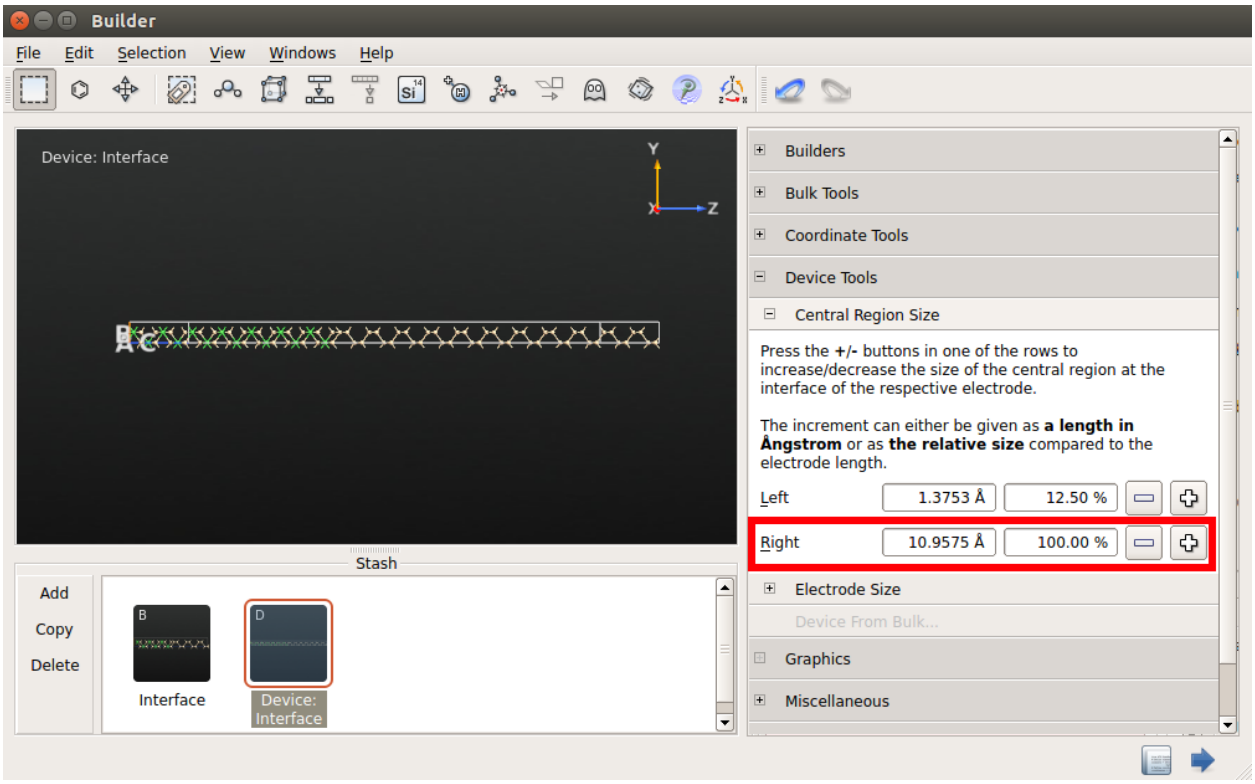
Note: In this tutorial you will use a highly doped device. The electronic transport properties of this device are significantly different from those a device with a much lower doping concentration, e.g. 10^{17} cm^{-3} or 10^{18} cm^{-3}. As you just learned, if you want to simulate a device with a lower doping concentration, you will also need a much longer device.

---

[1] E.ăKasper and D.ăJ. Paul. *Silicon Quantum Integrated Circuits*. Springer-Verlag Berlin Heidelberg, 2005. doi:10.1007/b137494.

Increase the length of the central region

1. Select again the device structure from the **Stash**.
2. Click on *Device Tools* → *Central Region size* and increase the dimension of right side (silicon) by 100% compared to the electrode length.



3. Send the structure to the **Script Generator** using the button to calculate some interesting properties of the undoped device.

Set-up the calculation for the undoped device

Once you have sent the structure to the **Script Generator**, you are ready to set-up the calculation. Add the following blocks to the script by double-clicking them:

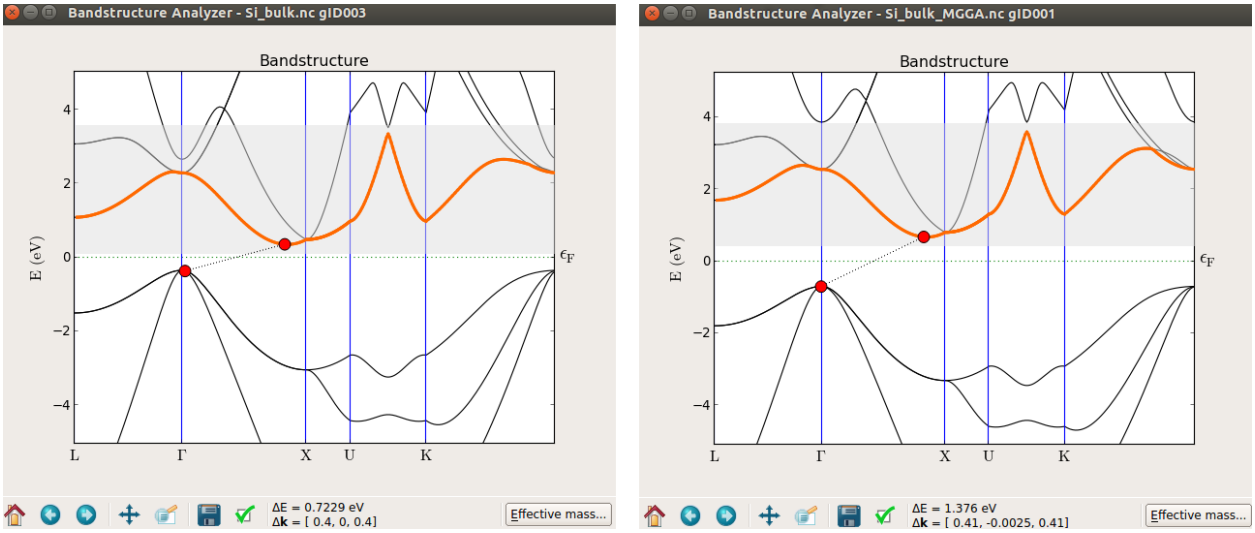
1. **New Calculator**. Double-click on it to modify the following parameters:

- change the exchange correlation parameter to MGGA.
- *k-point sampling*: 5x5x100 k-point sampling.
- *SingleZetaPolarized* basis set for both Si and Ni atoms.
- You need to set manually the TB09-MGGA c parameter in the **Editor**. Locate the exchange-correlation block in your python script and modify it as follows:

```
exchange_correlation = MGGA.TB09LDA(c=1.09368541059162)
```

Attention: We have obtained the c parameter self-consistently from a single-point calculation performed using the TB09-MGGA functional on the LDA-optimized structure of bulk silicon. If needed, you can obtain more details about how to obtain the c parameter or how to fit it from the tutorial *Meta-GGA and 2D confined InAs*. We have also compared the band structure of silicon at LDA and TB09-MGGA level of theory:

Figure 1: Band structure of bulk silicon computed using LDA (left) and TB09-MGGA (right).



From the picture it can be seen that the band structures are similar but the band-gap obtained using TB09-MGGA is closer to the experimental value (1.17 eV).

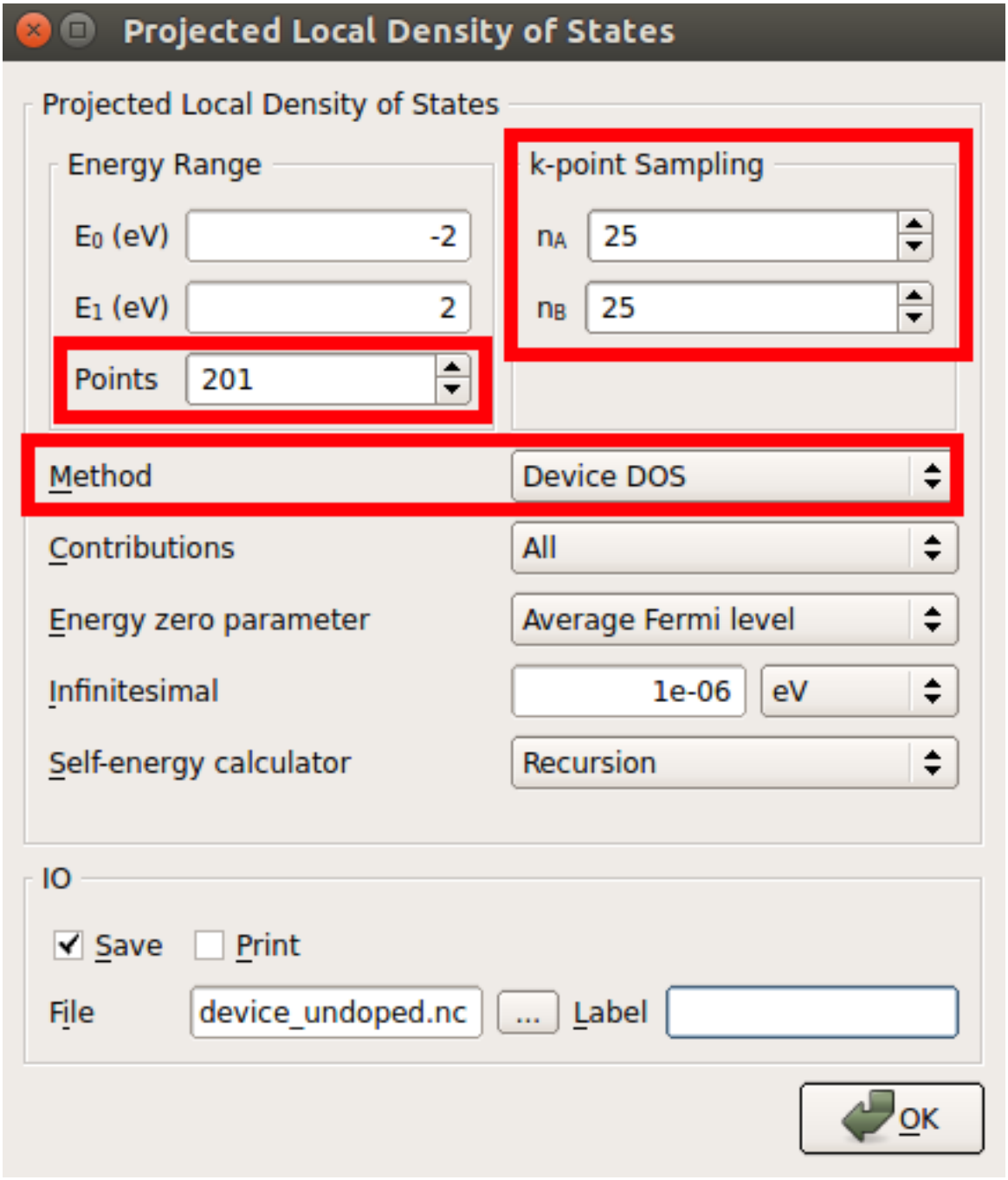
2. **Analysis**. In the menu that appears, choose the following analysis objects and modify the default parameters as shown below:

- **ElectrostaticDifferencePotentials**
- **ElectronDensity**
- **ElectronDifferenceDensity**.
- **ProjectedLocalDensityOfStates.**

– 25x25 k-points sampling

– 201 energy points

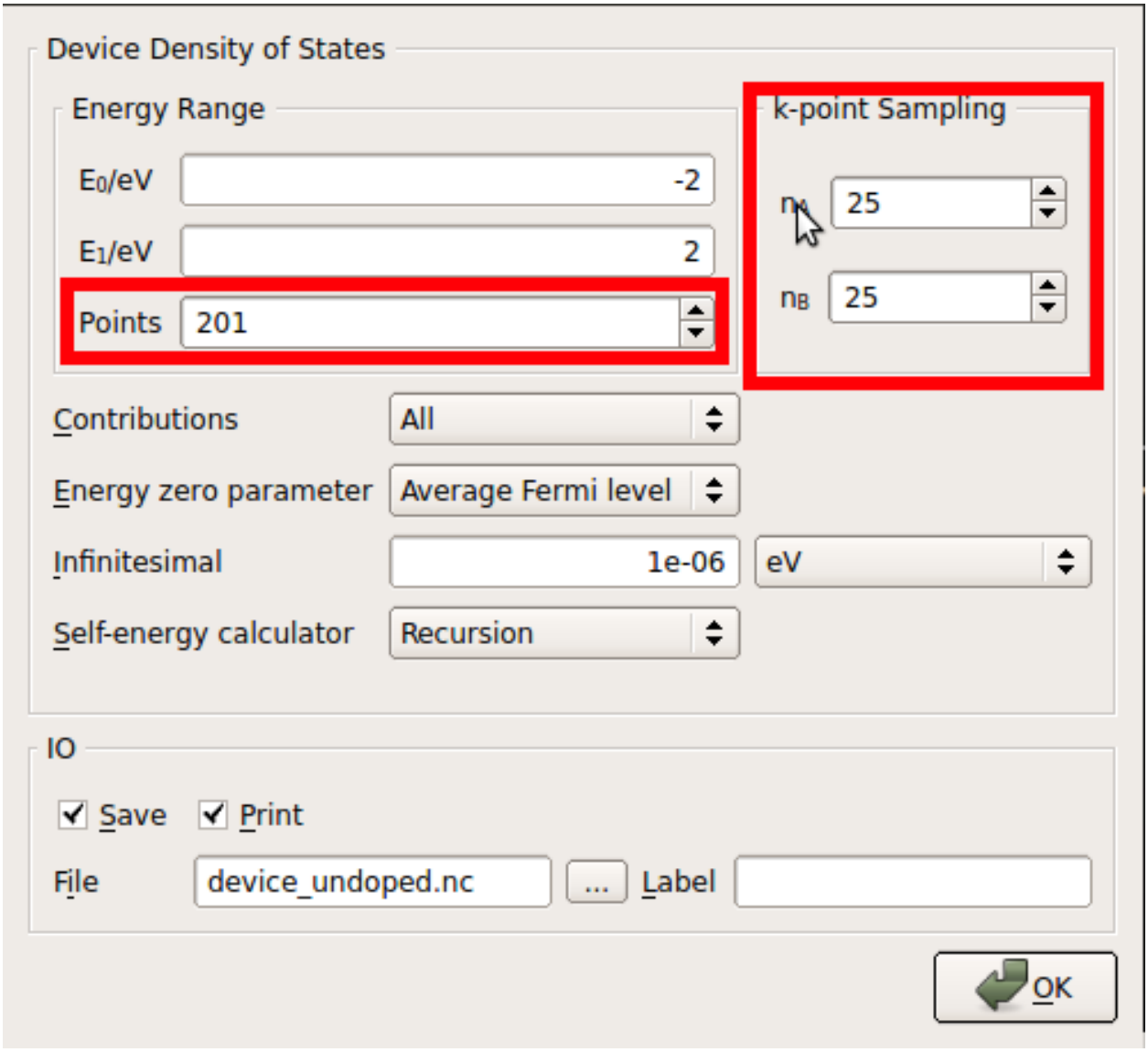
– Choose the *Device DOS* method.



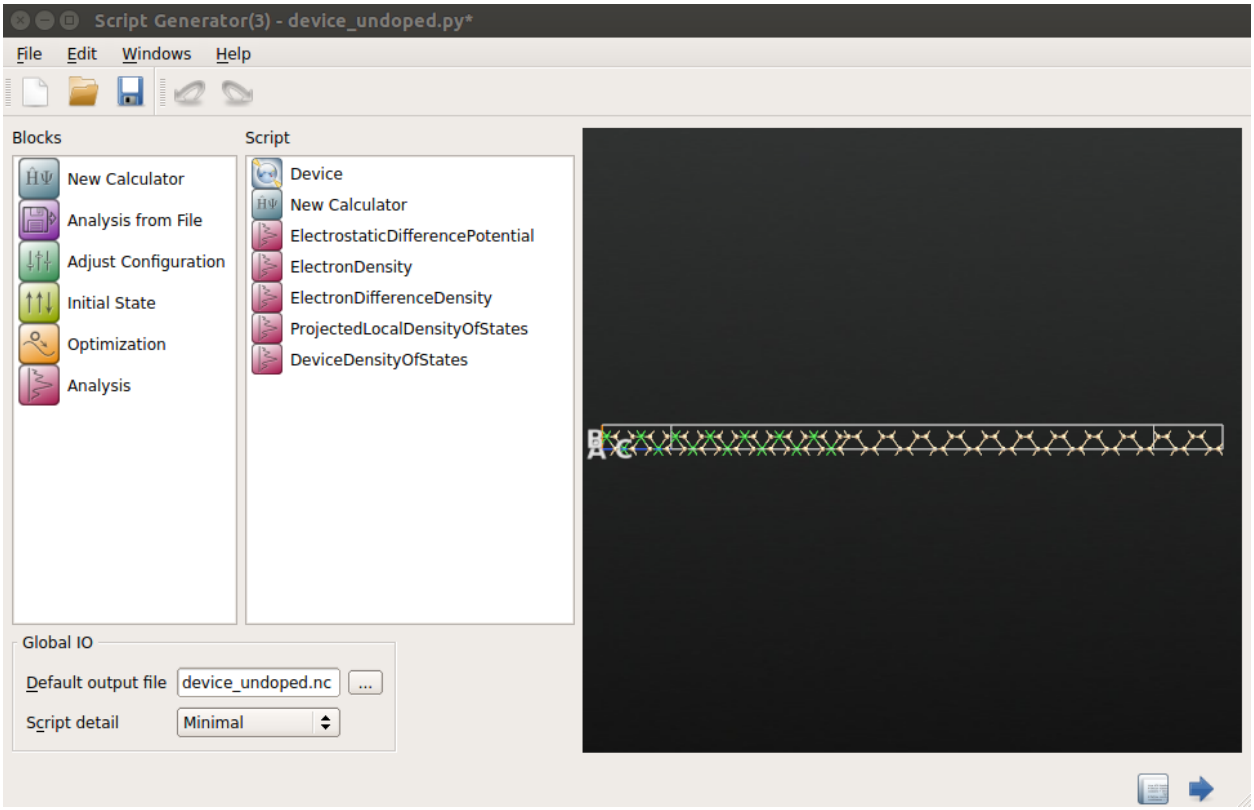
•**DeviceDensityOfStates.**

–25x25 k-points sampling

–201 energy points.



3. Change the name of the default output file to `device_undoped.hdf5` and save the script with the same name, `device_undoped.py`. The **Script Generator** panel should look as in the figure below.

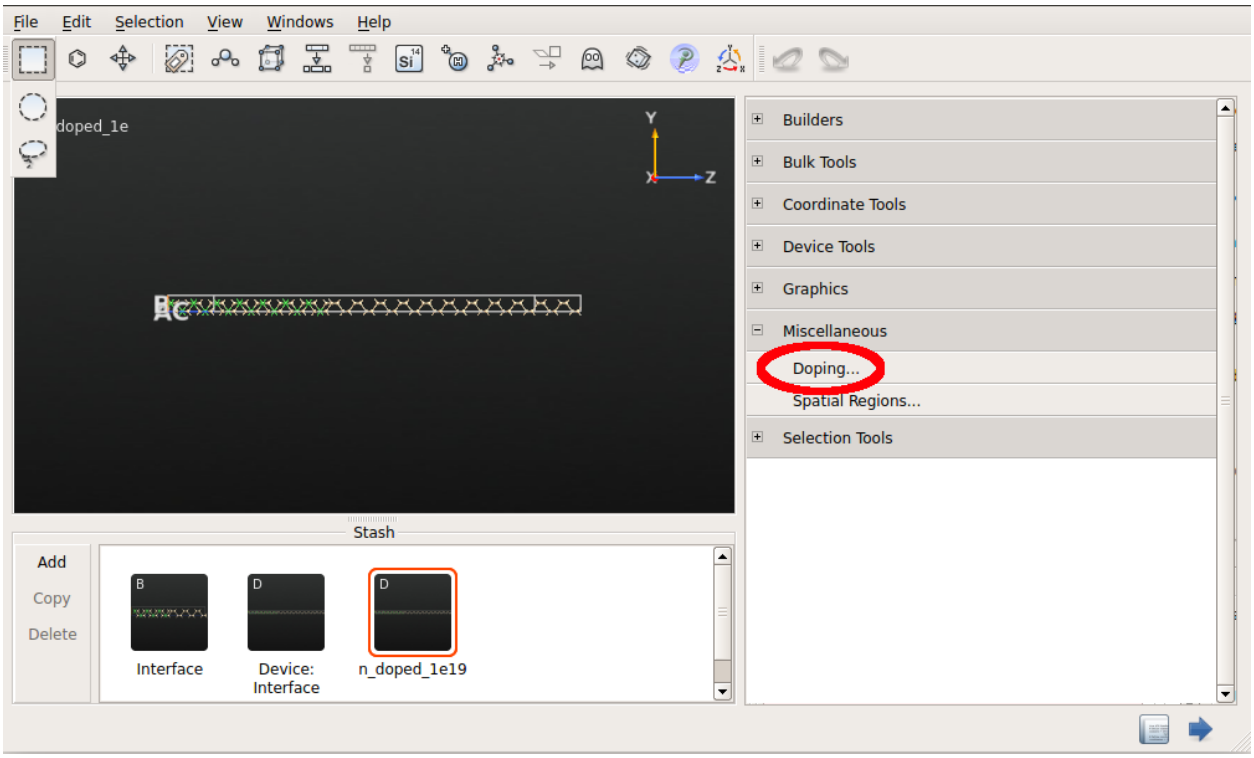


4. Send the script to the **Job Manager** and run it. The entire job will take around two hours by using 4 MPI processes. The calculation of each DOS takes around four hours on a single processor.

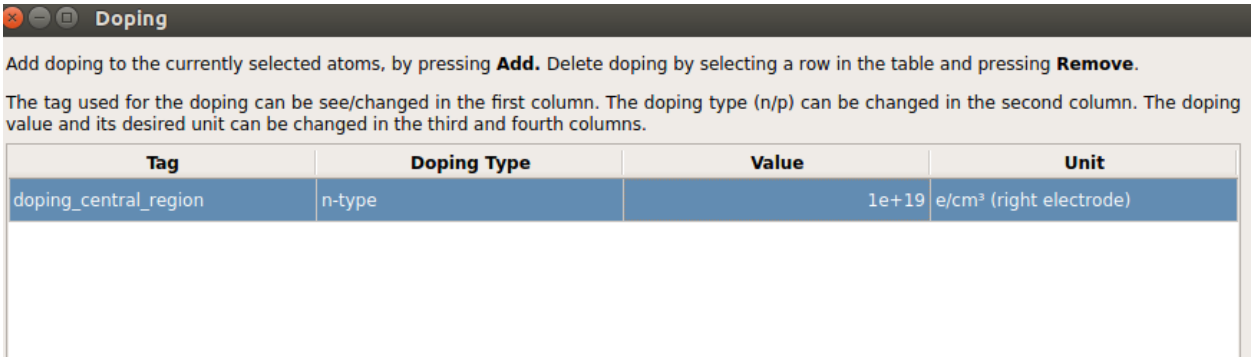
Note: The DOS calculations take quite a lot of time, especially with a 25x25 k-points sampling. You can reach reasonable good results also with a smaller mesh, e.g. 15x15 k-points. You can also run the DOS analysis at a later stage by reading the converged device configuration.

Dope the device

1. In the **Stash**, copy and paste the device configuration and rename it as `n_doped_1e19`.
2. Select the semiconductor atoms using the mouse by drawing a rectangle.
3. Go to *Miscellaneous* → *Doping*.



4.In the **Doping** panel that shows up set the doping as shown in the figure below.



5.Close the **Doping** panel and send the structure to the **Script Generator**.

6.Set up the calculation as for the undoped device and run the script.

7.Repeat the same steps for the p-type doped (10^{19} cm^{-3} and 10^{20} cm^{-3}) devices.

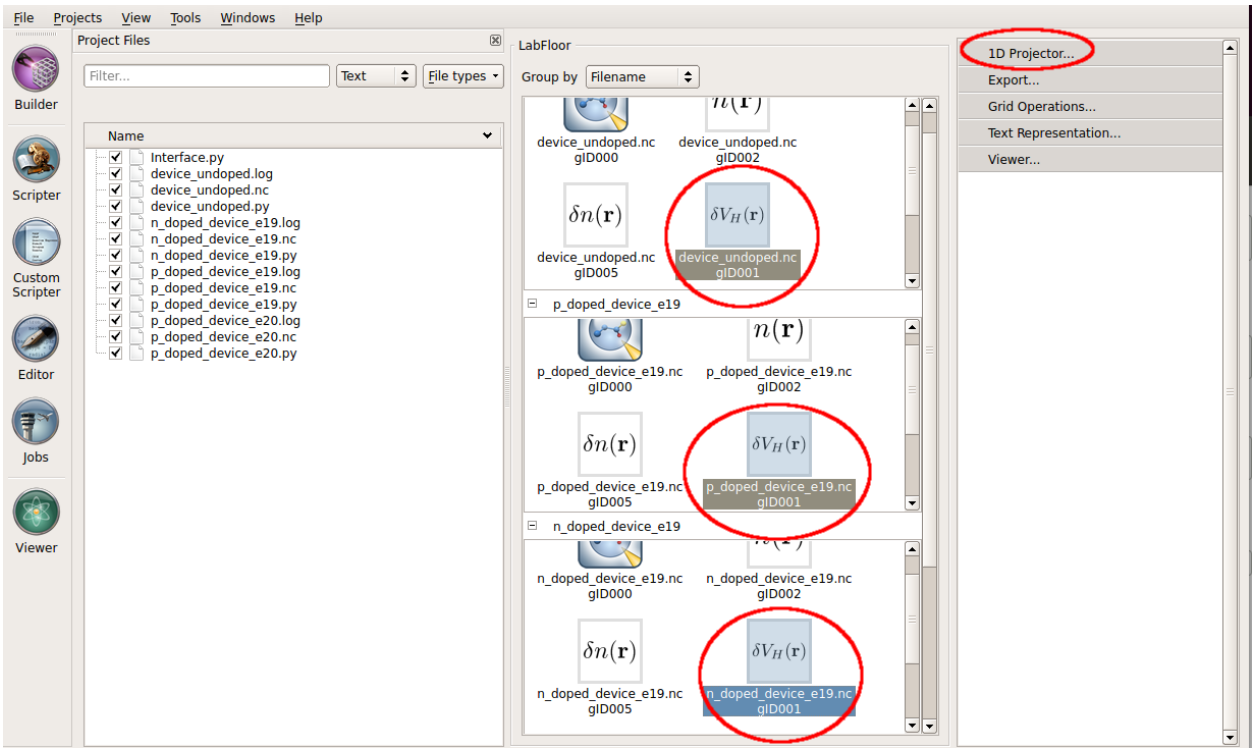
Analysis of the results

Once the calculations are finished, we are ready to analyze and compare the results obtained from the four calculations above.

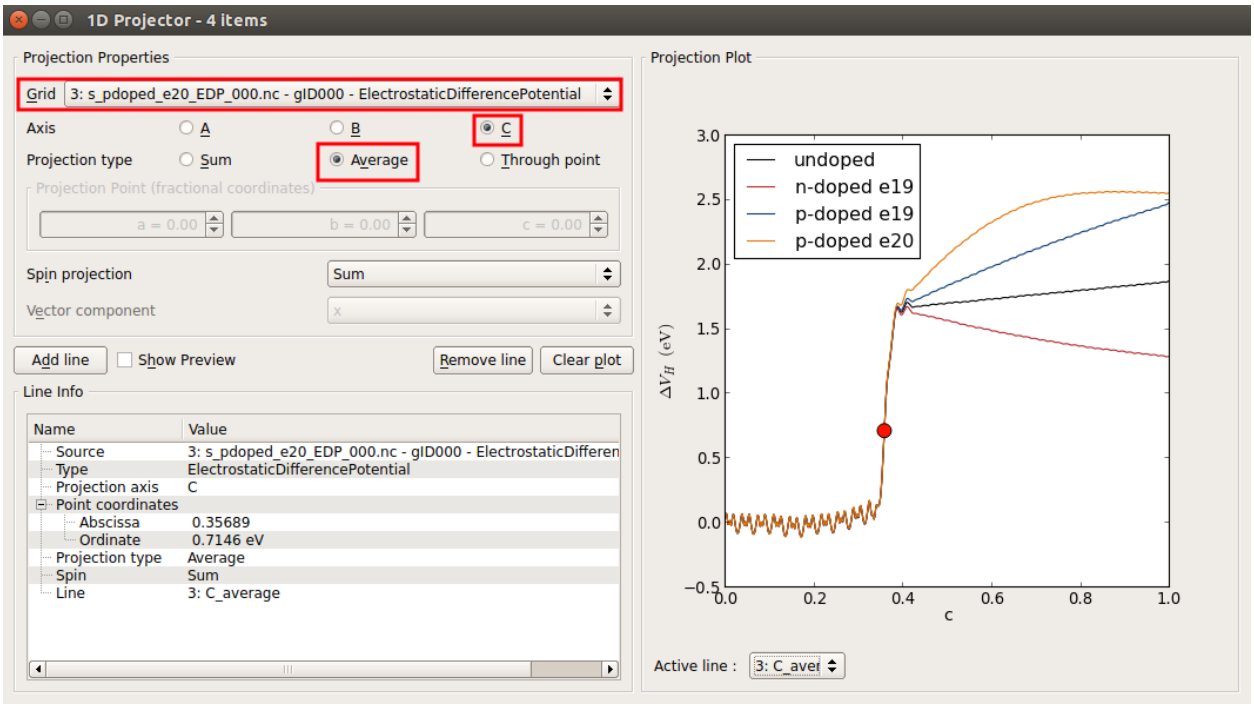
Electrostatic difference potential

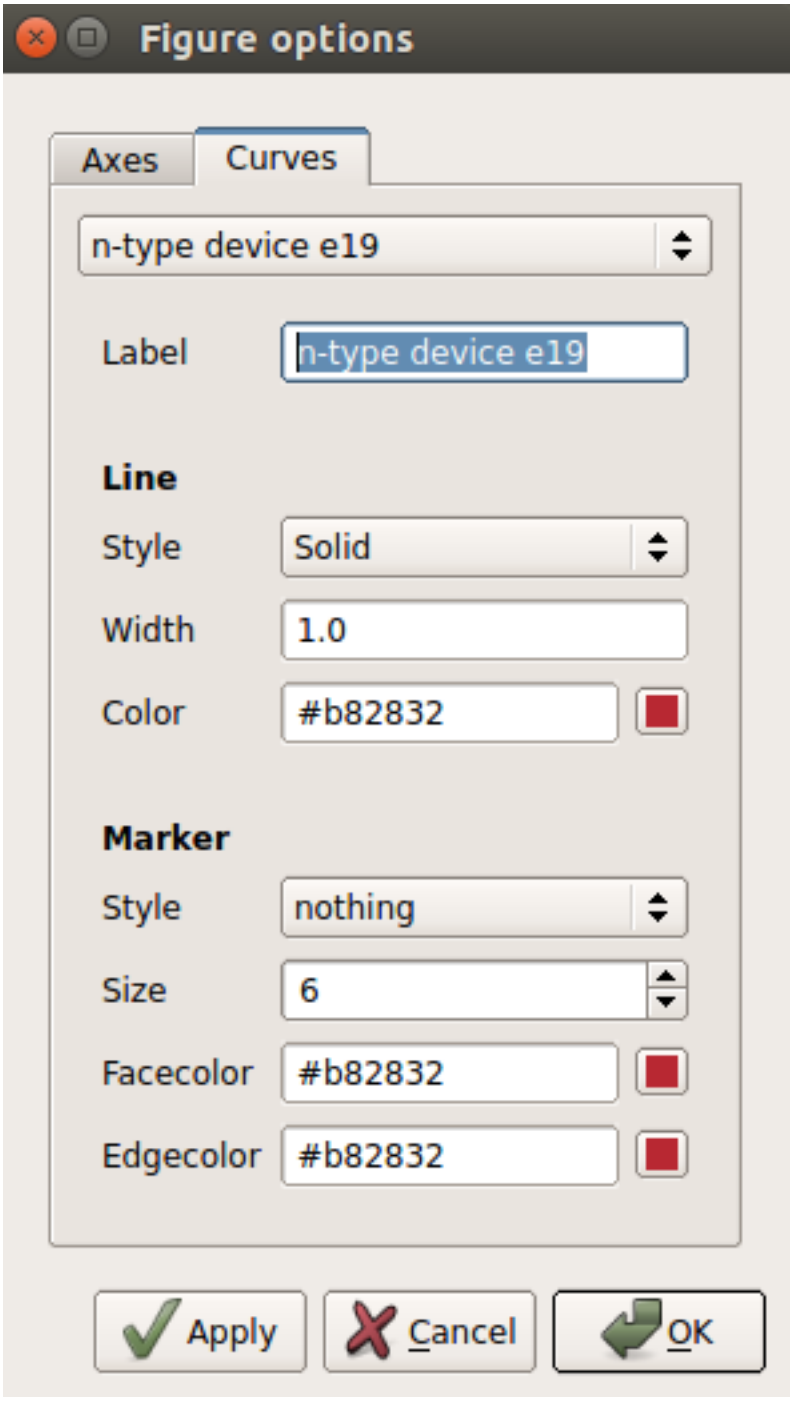
The electrostatic difference potential (EDP) corresponds to the difference between the electrostatic potential of the self-consistent valence electronic density and that obtained from a superposition of atomic valence electronic densities (see also the Reference Manual, *ElectrostaticDifferencePotential*). The analysis of the averaged EDP along the C axis of the device allows to see if the device is long enough, i.e. if the potential drop due to the interface is fully screened in the electrode regions.

1. In the **LabFloor**, select the four *Electrostatic Difference Potentials* objects obtained for the undoped and the three doped devices by holding Ctrl, and click on the **1D Projector** tool.



2. In the window that shows up, plot the average of the three potentials along the C direction as shown in the figure below. Right click in the graph and choose **Figure options** to change the legend.


1D Projector - 4 items
Projection Properties
Grid
3: s_pdoped_e20_EDP_000.nc - gID000 - ElectrostaticDifferencePotential
Axis
A
B
C
Projection type
Sum
Average
Through point
Projection Point (fractional coordinates)
a = 0.00
b = 0.00
c = 0.00
Spin projection
Sum
Vector component
x
Add line
Show Preview
Remove line
Clear plot
Line Info
Name
Value
Source
3: s_pdoped_e20_EDP_000.nc - gID000 - ElectrostaticDifferen
Type
ElectrostaticDifferencePotential
Projection axis
C
Point coordinates
Abscissa
0.35689
Ordinate
0.7146 eV
Projection type
Average
Spin
Sum
Line
3: C_average
Projection Plot
undoped
n-doped e19
p-doped e19
p-doped e20
ΔV_H (eV)
c
Active line :
3: C_aver



This plot shows that the device is not long enough for the undoped device and for those at a doping of 10^{19} cm^{-3}. (see below the convereged EDP calculated for the longer device at an n-type doping of 10^{19} cm^{-3}). The EDP of the p-type doped device at a doping of 10^{20} cm^{-3} is the only one showing the correct behaviour: it can be seen that the EDP it is flat in the electrode extension region (close to c = 1). The interface is located around the fractional coordinate c = 0.35, although it is better to say that the silicon atoms in contact with the Ni atoms are located at this coordinate since the interface is not simply a infinite thin 2D plane but correspond to a thick region of your device.

Electron Difference Density

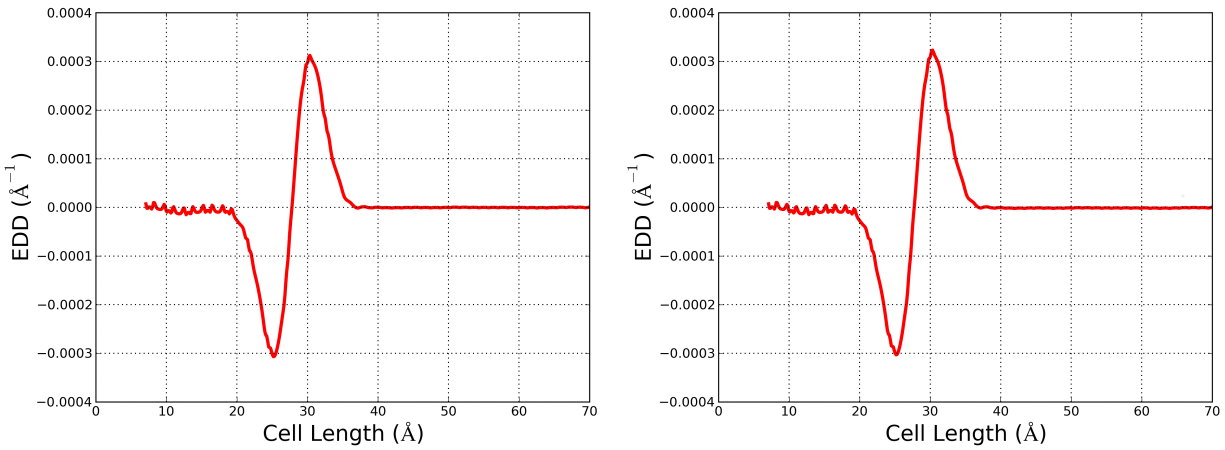
Electron Difference Density (EDD) is the difference between the self-consistent valence electronic density and the superposition of the atomic valence electronic densities.

1. Use the script provided (download the script here: `edp_macro_avg.py`) to plot the macroscopic average of the EDD.
2. Write the following commands in the terminal:

```
atkpython edp_macro_avg.py file.hdf5
```

where `file.hdf5` is the name of one of the HDF5 output files obtained from the device calculations.

Figure 2: Macroscopic average of the EDD of the undoped device (left) and of the n-type doped device (right).



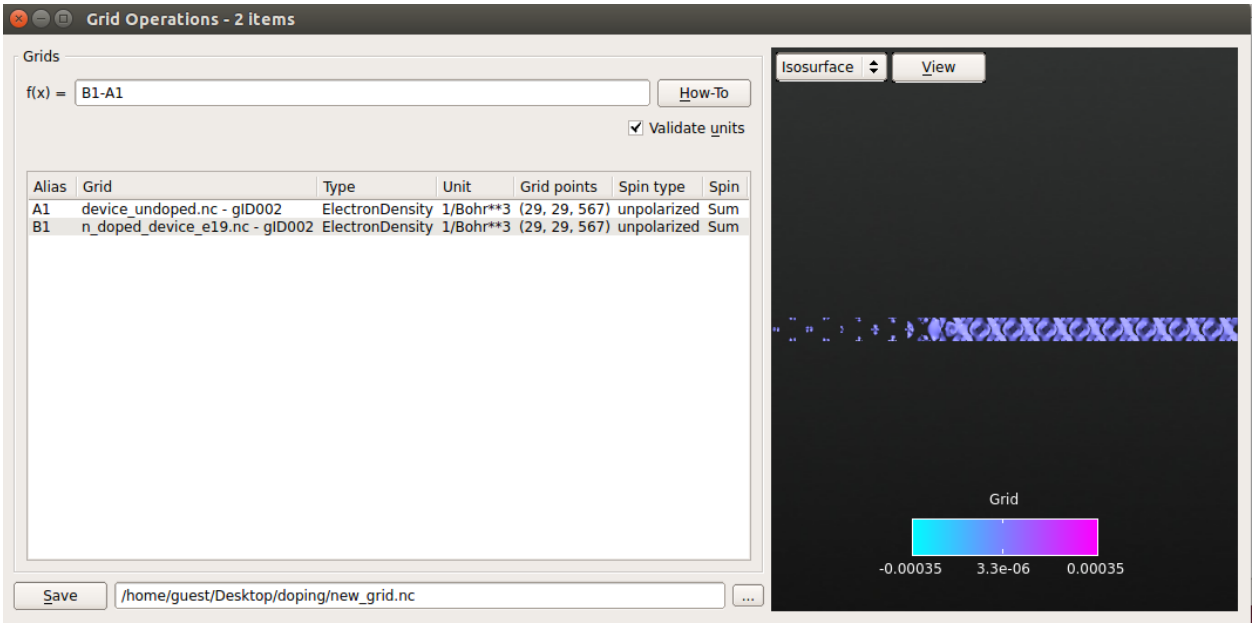
These two plots show the presence of an interfacial dipole which changes (although the change is small) unpon changing the doping in the semiconductor.

Electron density difference (doped - undoped)

You can analyze the difference in electron density between two device configurations in the following ways:

3D Analysis

1. Select the **ElectronDensity** objects of the following configurations from the **LabFloor**:
   - undoped device.
   - n-doped device (10^{19} cm^{-3})
2. Click the **Grid Operation** plug-in on the left-hand side of the **LabFloor**.
3. In the formula editor, insert B1-A1 and click *View* to visualize the isosurface of the electron density difference.



4.Save the generated density difference in a `new_grid.hdf5` file.

1D Analysis

Use the provided script to plot the macroscopic average of difference between the n-type doped and the undoped configurations.

1.Open the script and modify the line:

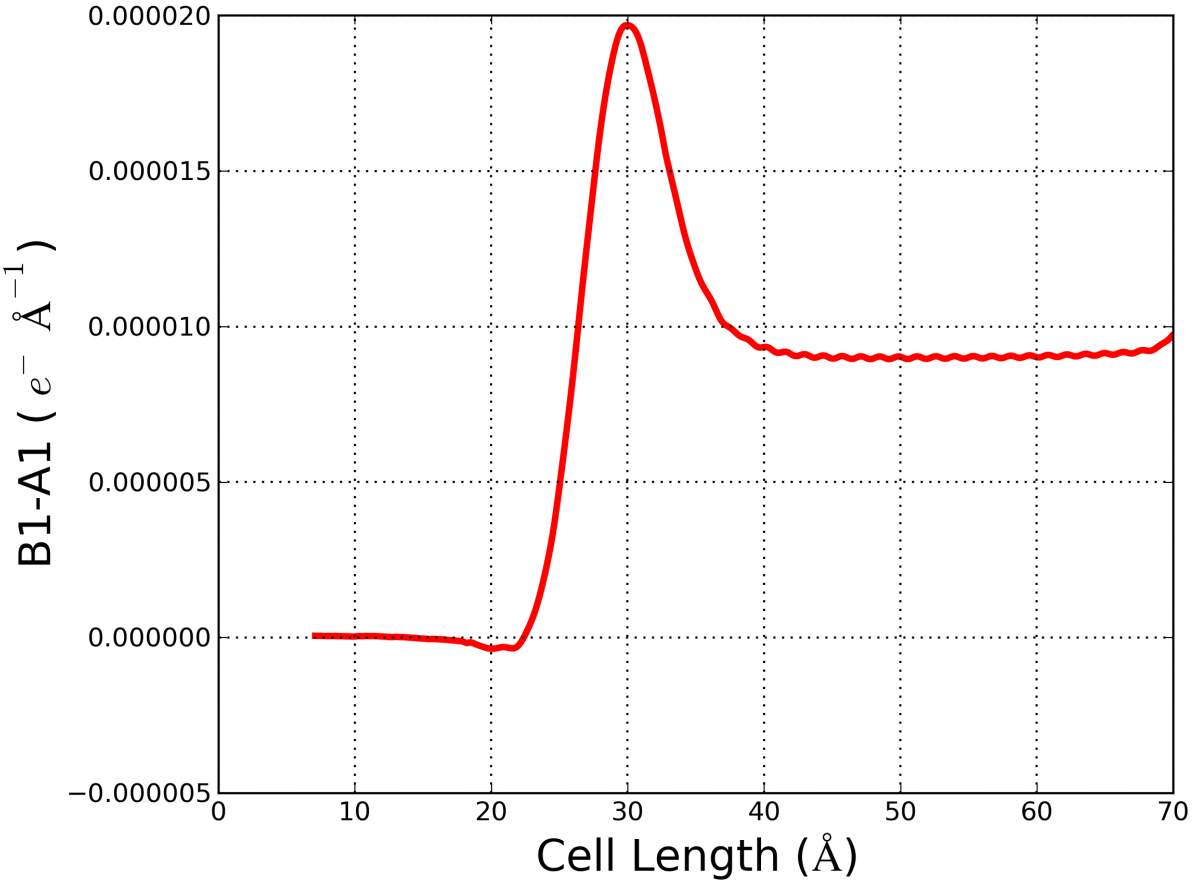
```
ax.set_ylabel(r'EDD ($\AA^{-1}$)', fontsize=20)
```

as follows:

```
ax.set_ylabel(r'B1-A1 ($\AA^{-1}$)', fontsize=20)
```

2.Run the script as shown previously:

```
atkpython edp_macro_avg.py new_grid.hdf5
```



The plot shows the difference B1-A1 between the electronic densities of the n-type doped (B1) and undoped (A1) devices. B1-A1 > 0 e^- Å $^-$ in the semiconductor region due to the extra electrons added to the silicon in the n-type doped device. Furthermore, the value of B1-A1 increases when approaching the interface, which can be understood taking into account the change of the interfacial dipole due to doping.

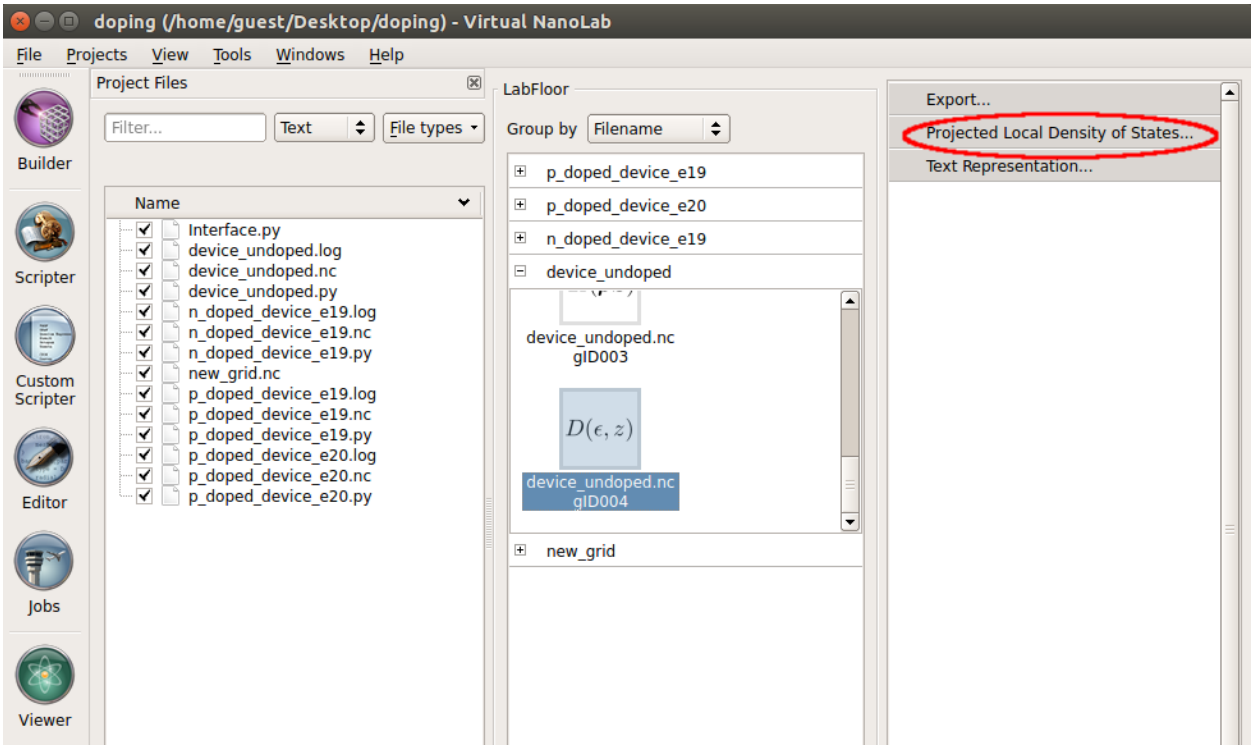
Density of states (DOS) analysis

In this tutorial we have computed two DOS analyses.

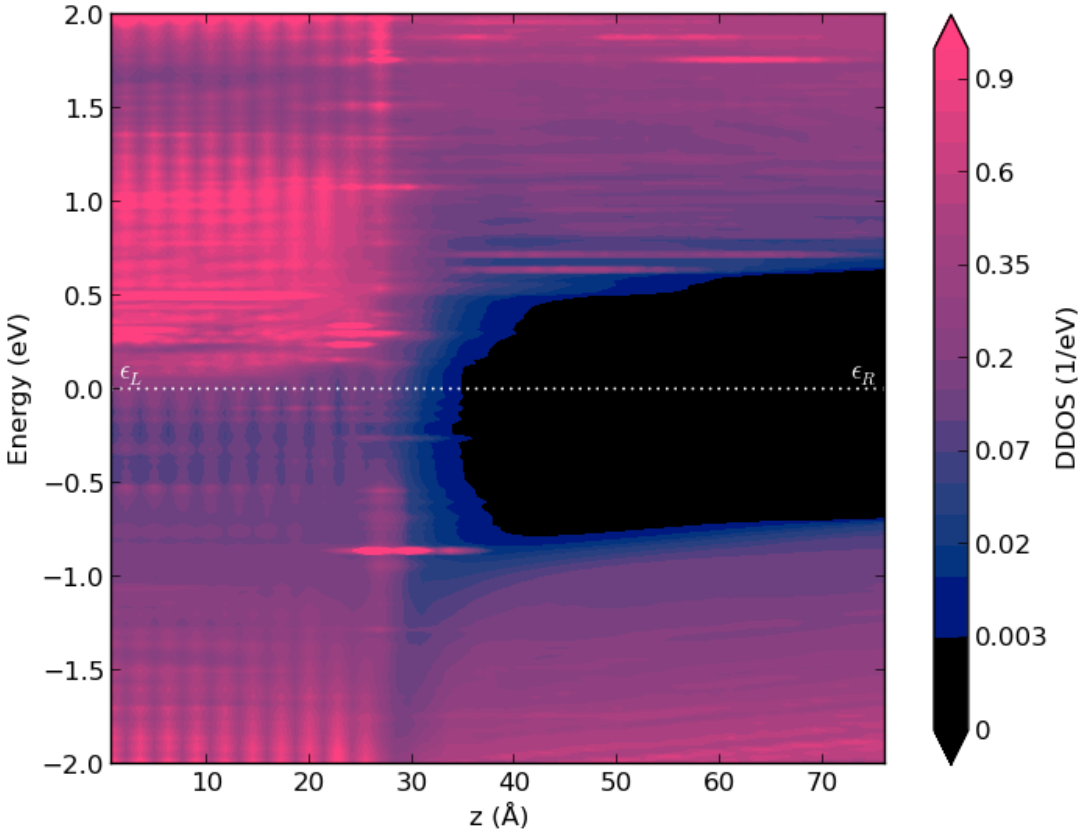
Projected Local Density Of States

1.Select the *Projected Local Density Of States* object of the undoped device from the **LabFloor**.

2.Click the **Projected Local Density of States** plug-in:



You will obtain the following plot:

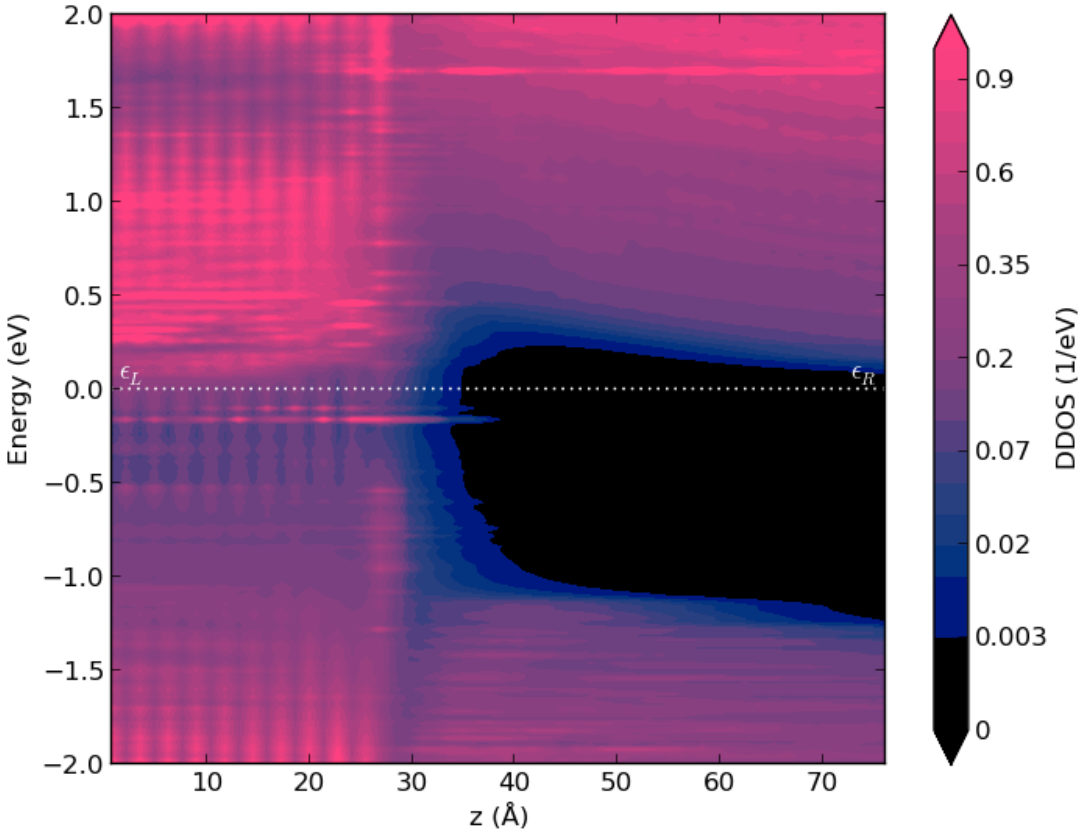


The abscissa indicates the Cartesian coordinate of the central region along the C direction. In this case, the interface is located around 27 Å. The energy on the Y-axis is relative to the Fermi energy. The color bar indicates the DOS amplitude.

It is clear from this plot that:

- The left side of the device is metallic, $NiSi_2$;
- The right side is a semiconductor with a band gap that resemble that of bulk undoped Si;
- There is a wide region of silicon atoms which show gap states induced by the metal-semiconductor interface. These are called metal-induced gap states (MIGS).

3. Follow the same procedure to plot the projected local DOS of the n-doped device.

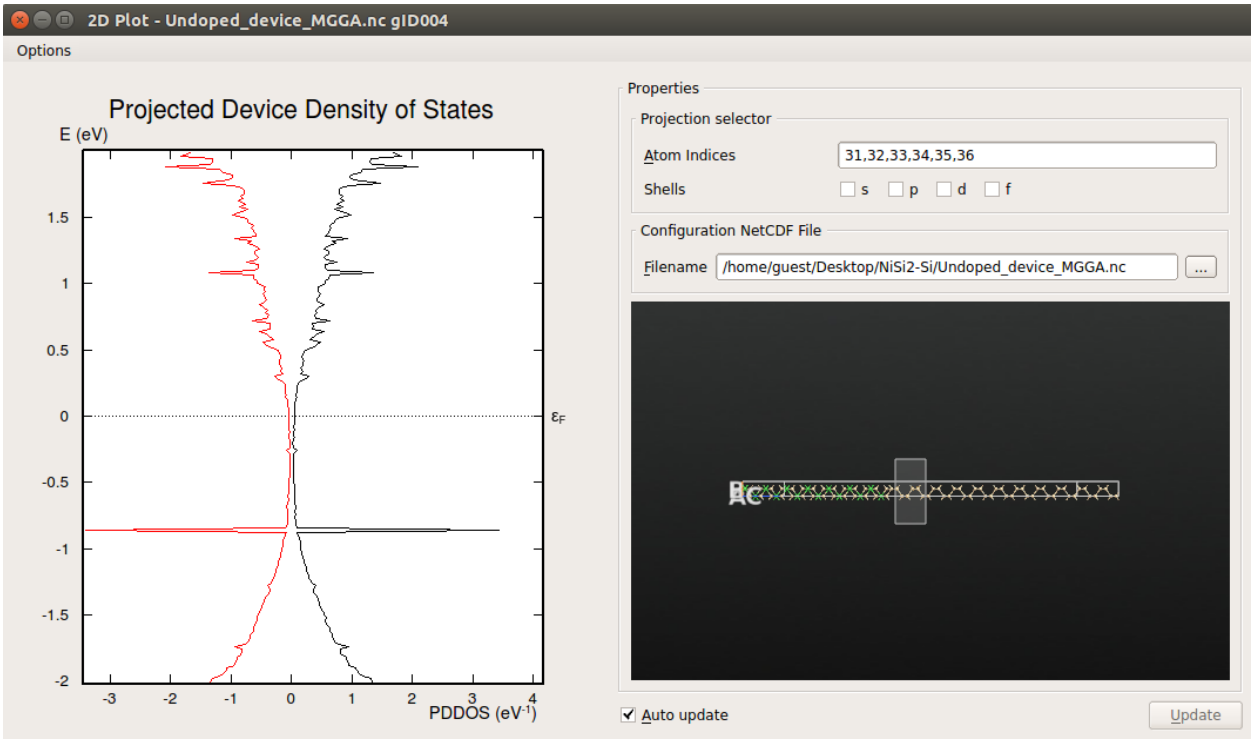


Note that in this case the Fermi level lies at the bottom of the silicon conduction band due to the doping of the semiconductor.

Device Density Of States

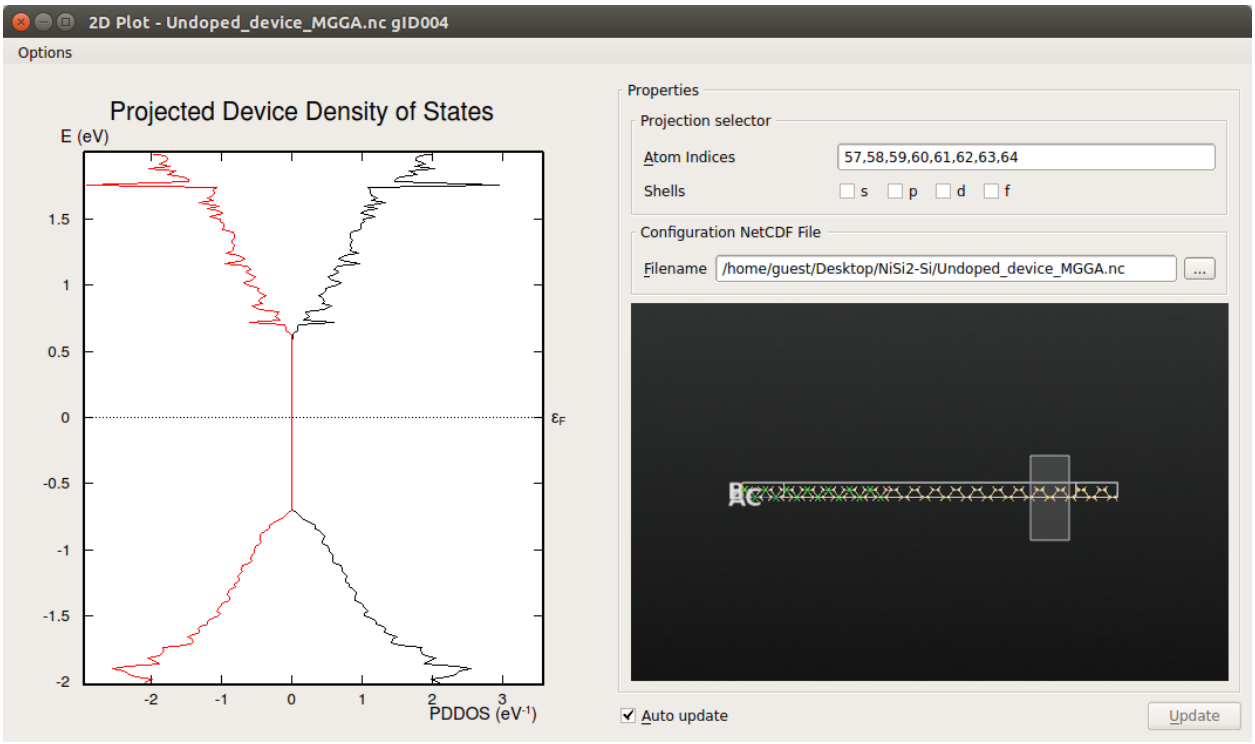
This tool allows the analysis of the density of states projected onto the atomic sites of the device.

1. Select the *Device Density Of States* object of the undoped device in the **Labfloor**.
2. Click the **2D Plot** plug-in and the window that appears will show the 2D plot of the total density of states together with a 3D view of the device.
3. Navigate to the 3D view and select the atoms close to the interface by drawing a rectangle with the mouse as shown below.



The 2D plot now shows the projected density of states onto the selected atoms. It is possible to see the presence of MIGS around the Fermi level.

4.Select the silicon atoms close to the electrode as shown below:



It can be seen that close to the silicon electrode the MIGS have disappeared and the electronic structure presents a band-gap similar to that calculated for bulk silicon, in agreement with the results obtained from the analysis of the projected local DOS.

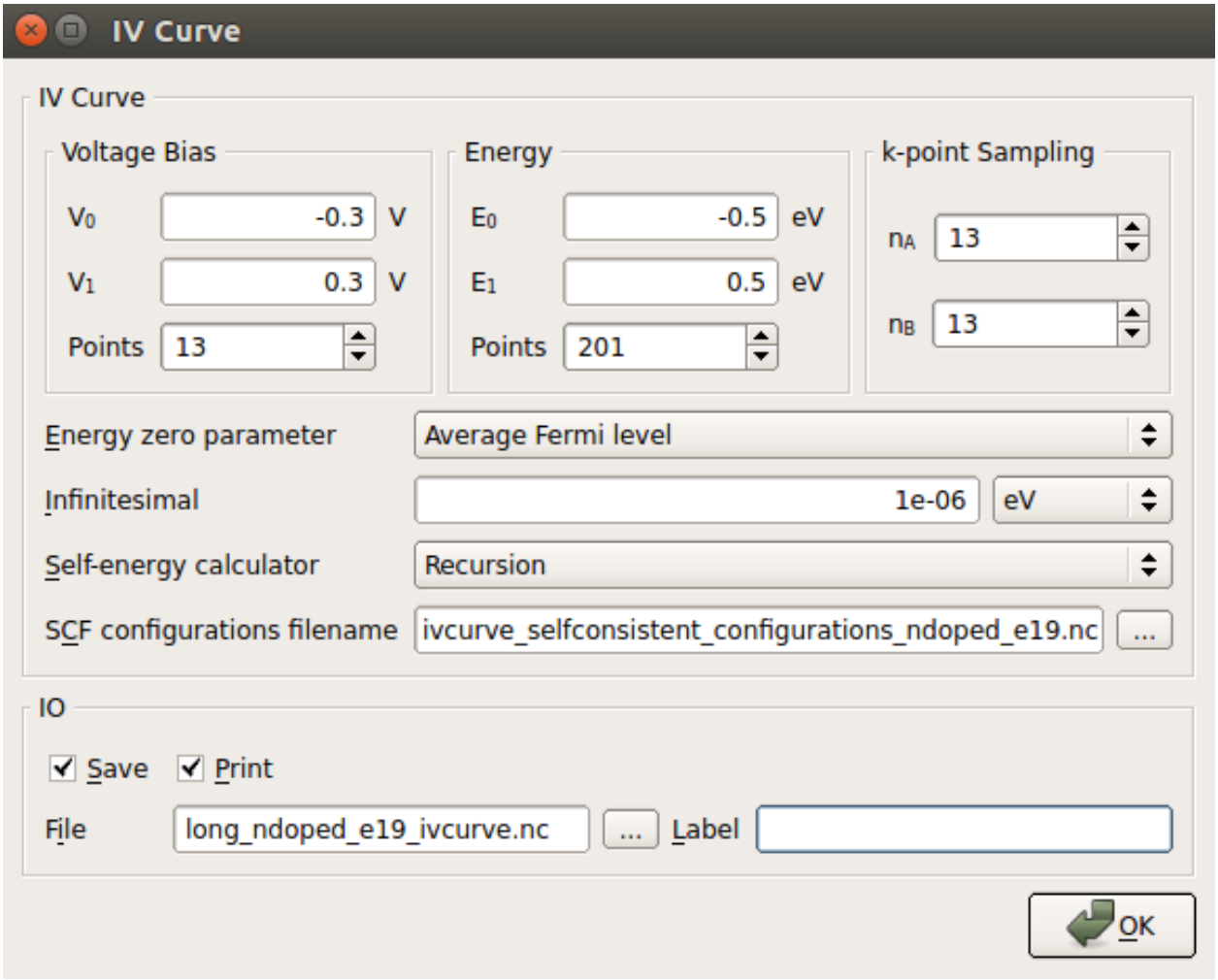
Finite-bias calculations

In this section we will calculate the IV curve for a n-type doped device (10^{19} cm^{-3}). As we have already mentioned, the device used above is not long enough for this doping concentration. In this section, we will perform the finite-bias calculations using a 100 Å long device. We have increased the lenght of the device following the steps described in the *Increase the length of the central region* section. You can download the structure here (`long_ndoped_e19_ivcurve.py`).

Set up the IV curve calculation

1. Send the structure to the **Script Generator** and add the following blocks to the script:

- **New Calculator** : Set the parameters as for 0 V_{bias} calculations.
- **IV curve**: Change the following default parameters as in the figure below:



It is worth adding some comments regarding the meaning of the most important parameters:

– *Voltage Bias* : 13 different calculations will be performed at V_{bias} = 0.0, -0.05, -0.1, -0.15, -0.2, -0.25, -0.3, 0.05, 0.1, 0.15, 0.2, 0.25 and 0.3 V, in this exact order. V_{bias} is defined as V_L - V_R.

Note: Each calculation will use the self-consistent configuration of the previous one as starting guess. This procedures improves quite a lot the convergence.

–*Energy* : These parameters refer to the enery range and the number of points within this range at which the transmission spectra will be calculated at each bias voltage.

Warning: The convergence of the energy range is very important when calculating IV curves! Check the *Transmission spectrum convergence* and The *IV curve object analysis* sections.

–*k-point Sampling* : This is the k-point mesh used to calculate the transmission spectra. It must correspond to a sampling for which the transmission curve is converged. In general a greater k-point sampling will be needed with respect to the one used for the SCF part.

Warning: The convergence of the k-point mesh can affect the calculated current! In this tutorial we have considered that at 13x13 the transmission is converged, check the *Transmission spectrum convergence* section.

–*IO File name* : Change the default name if you are going to do more than one IV curve. This is the file in which all the converged configurations will be stored. As you will learn in the *Post analysis of finite-bias calculations* section, you can use this converged states to:

∗re-calculate all, or selected transmission spectra with different parameters (k-point sampling, energy range,)

∗perform additional analysis at any specific bias (DOS, electron density, effective potential,)

∗extend your bias range by restarting from a well converged finite bias calculation.

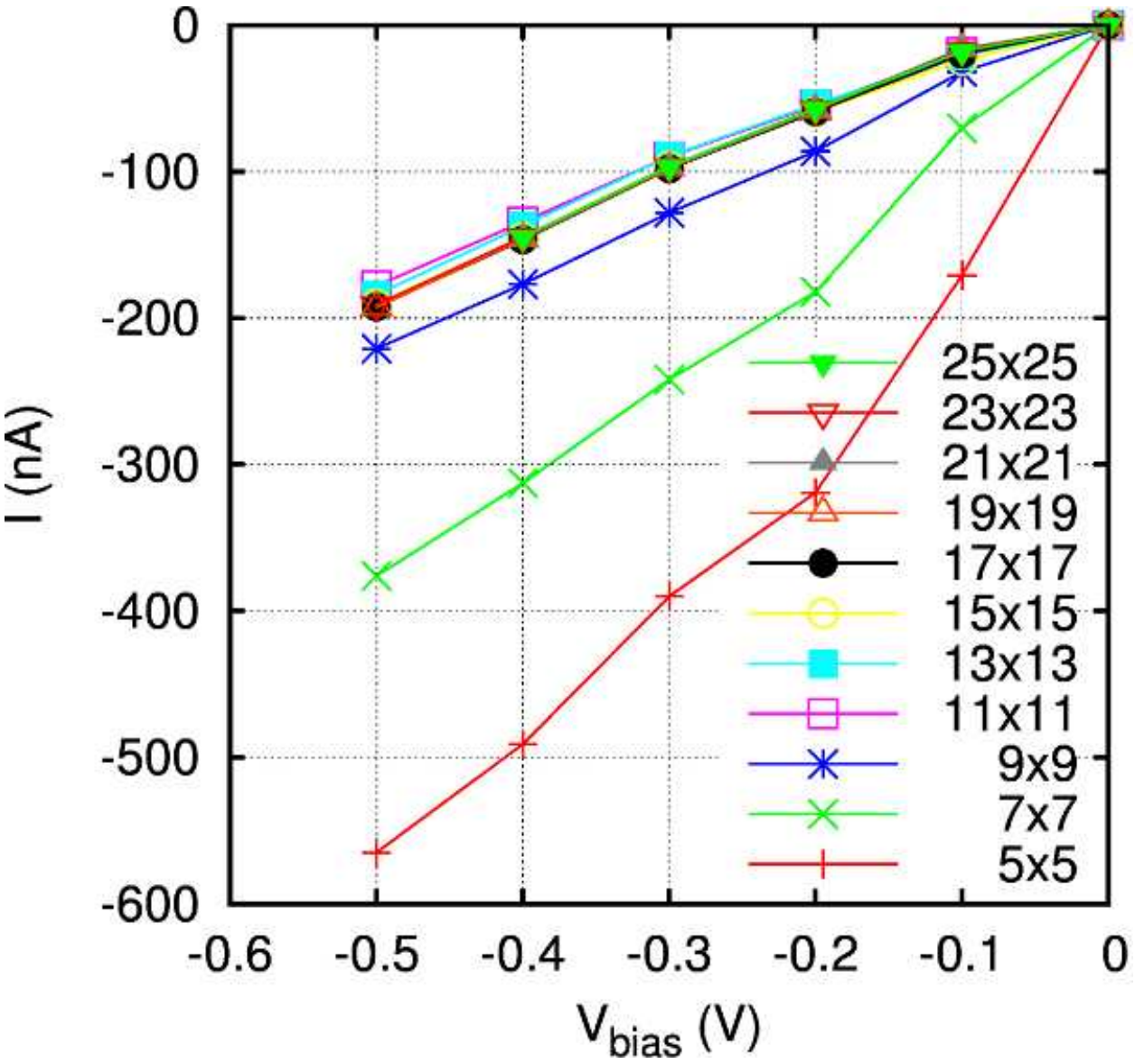
2.Once you have set all the parameters, you are ready to send the script to the **Job Manager** and run the calculation. This calculation will take around ten hour using 4 MPI processors.

Transmission spectrum convergence

Two parameters are very importat to determine the convergence of the transmission spectrum:

•**Number of energy points:** The first parameter you can tune to improve the balance between the precision of your transmission and computational efficiency is the number of energy points used to evaluate the transmission spectrum. Ideally, the range of the transmission spectrum should span at least the entire region where the spectral current is finite. In the present case the minimum range would be between +0.1 eV and +0.4 eV (see below). However, it is often safer to consider a larger and symmetric energy range.

•**K-points sampling:** The second and most important parameter to consider is the k-point sampling used for the evaluation of the transmission spectra. As already mentioned, you usually need more k-points than for the DFT calculation, since the Greens functions (which determine the transmission) have a stronger variation with k than the Fourier components of the electron density do. The plot below shows the IV curves at negative bias for the n-type doped device (at 10^{20} cm^{-3} doping concentration) calculated using different k-points samplings for the evaluation of the transmission spectra. In general, it is enough to converge the current at a specific applied bias within the range considered.



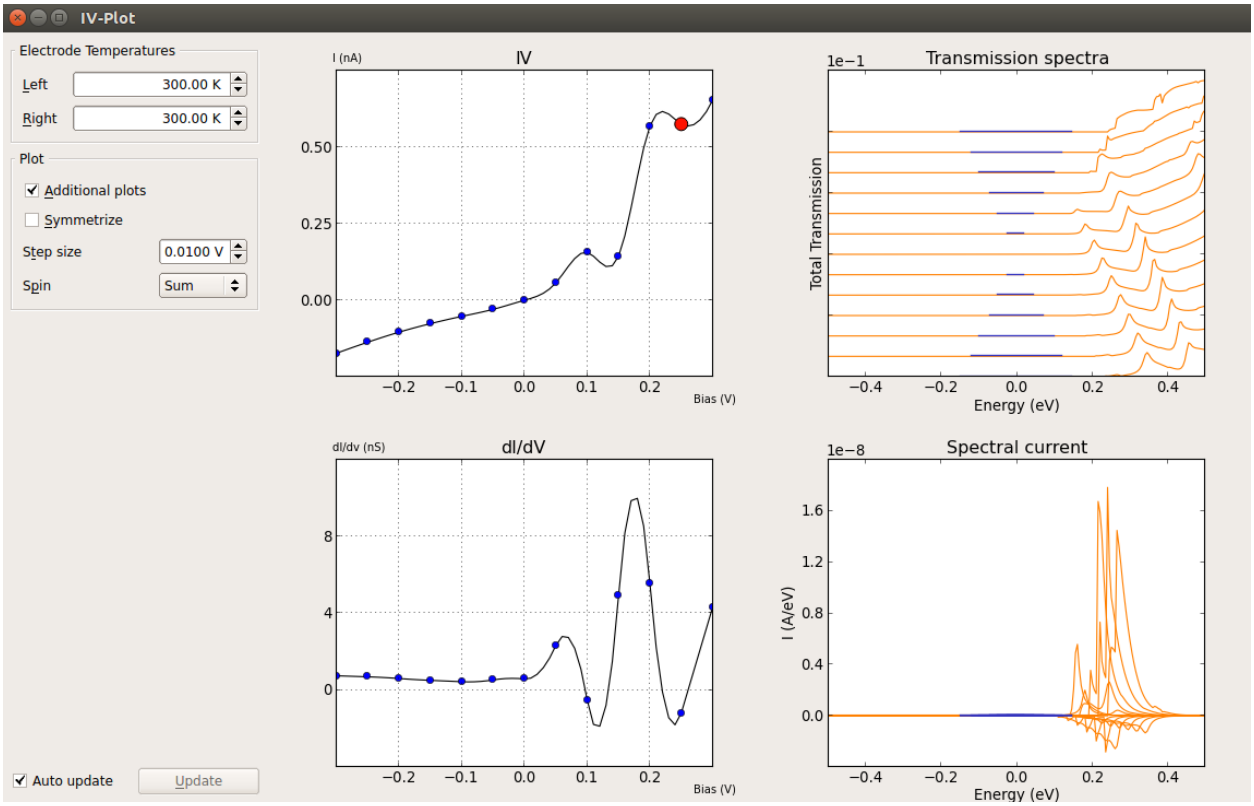
From this plot we can deduce that a 13x13 k-points mesh is good enough to give converged results, since the IV curve obtained at this mesh is very close to the one obtained at the highest sampling, 25x25.

Warning: This assessment have been done for n-type doped device with a concentration of 10^{20} cm^{-3}. For a more rigorous study, an assessment for the device doped at 10^{19} cm^{-3} should be done.

IV curve object analysis

Once the IV curve calculation are finished we are ready to analyse the **IVCurve** object. You can find it loaded in the **LabFloor**.

- Select the **IVCurve** object and click the **IV-plot** plug-in.
- You will find the IV curve plot in the panel that shows up. Click on *Additional Plots* and you will get the dI/dV, transmission spectra and spectral current plots.



Important: The spectral current is the energy resolved current:

$$I(E) = e/h[T(E, \mu_R, \mu_L)(f(E - \mu_R)/(k_B T_R) - f(E - \mu_L)/(k_B T_L))]$$

This is the quantity which is integrated to get the current at a given bias. As such, it is good practice to check that the energy range set for the evaluation of the transmission spectra spans the entire region where the spectral current differs from zero. You can refer to the *TransmissionSpectrum* entry in the Reference Manual for more information on how the current is evaluated in QuantumATK.

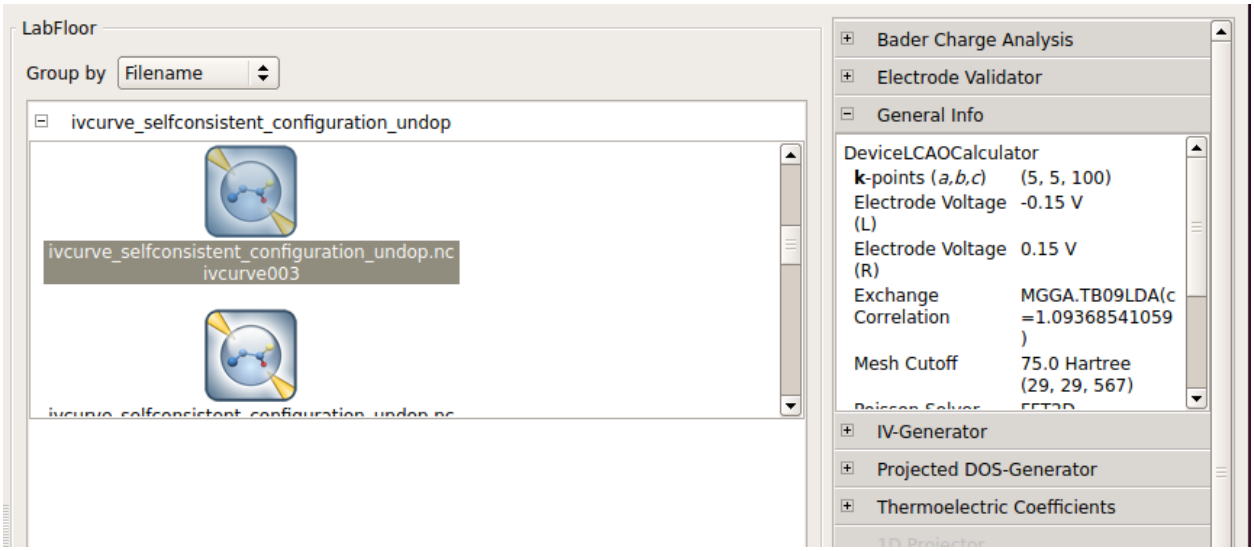
Note: You can right-click on any of the plot windows to export the data and use your favourite utility to plot your results.

Post analysis of finite-bias calculations

As we have already mentioned, using the selfconsistent configurations (stored in the `ivcurve_selfconsistent_configurations.hdf5` file) it is possible to perform further analysis at any specific bias. In this section we will explain how to inspect this objects and how to set-up the script for the additional analysis of a configuration.

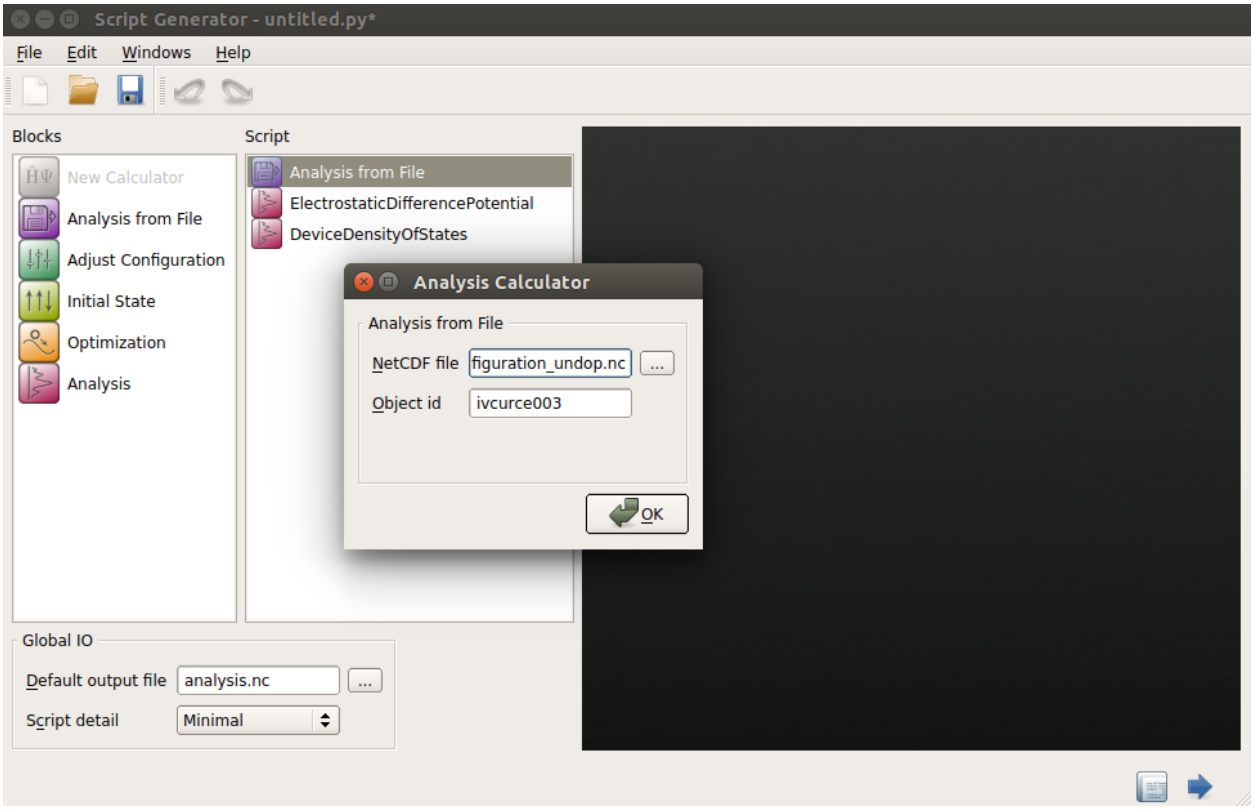
Analysis of a single self-consistent configuration

1. Select a self-consistent configuration and click on the **General info** tab. In the panel that opens up, you can check the details of the self-consistent configurations. Once you have identified the configuration you are interested in, you are ready to set up the script.



2. Go to **Script Generator** and add the **Analysis from File** block.

3. Double-click the **Analysis from File** block to open it and insert the name of the HDF5 file and the Object ID of the self-consistent configuration you want to analize.



4. Add the analysis blocks you need to the script and once you have set the parameters of these additional blocks, send the script to the **Job Manager** to run it.

Analysis of multiple self-consistent configurations

Here we will show how to perform an additional evaluation of the EDP at different bias voltages starting from the self-consistent configurations obtained previously.

1. Identify the self-consistent configurations you want to analyze using the **General info** tab as explained in the previous section.

2. Use the provided script (`loop.py`) to perform the analysis you want for multiple self-consistent configurations in the same calculation. Select the self-consistent configurations by including the relevant Object IDs in the id_list in the script.

```
id_list = [
      'ivcurve000',
      'ivcurve001',
      'ivcurve002',
      'ivcurve003',
      'ivcurve004',
      'ivcurve005',
      ]
for id_item in id_list:
   # Read the DeviceConfiguration
   configuration = nlread('ivcurve_selfconsistent_configurations.nc', object_id="%s" %id_
↪item)[0]

```

(continues on next page)

(continued from previous page)

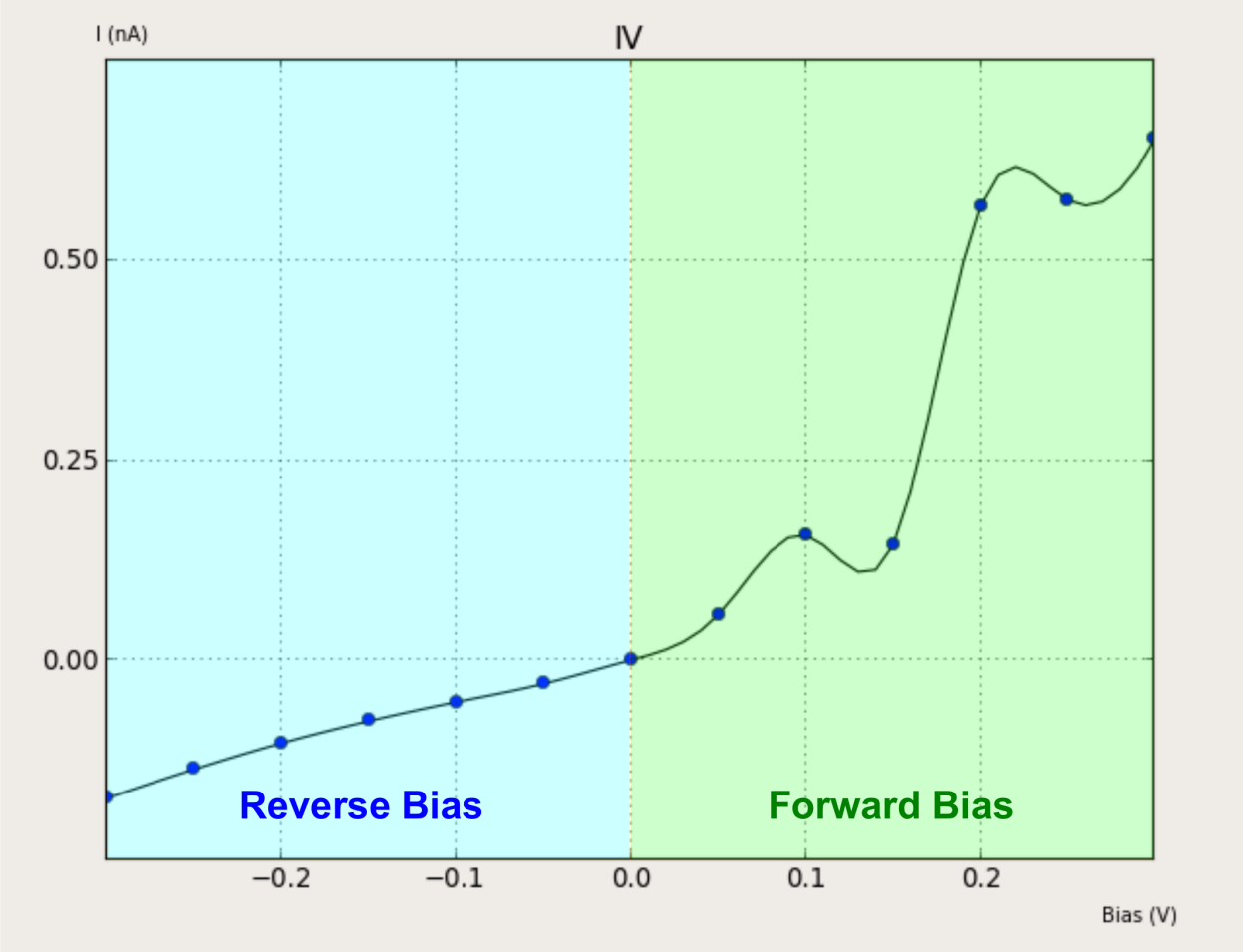
```
# ------------------------------------------------------------
# Electrostatic Difference Potential
# ------------------------------------------------------------
electrostatic_difference_potential = ElectrostaticDifferencePotential(configuration)
nlsave('analysis.nc', electrostatic_difference_potential)
```

3.Send the sript to the **Job Manager** and run the calculation.

Analysis of the IV curve

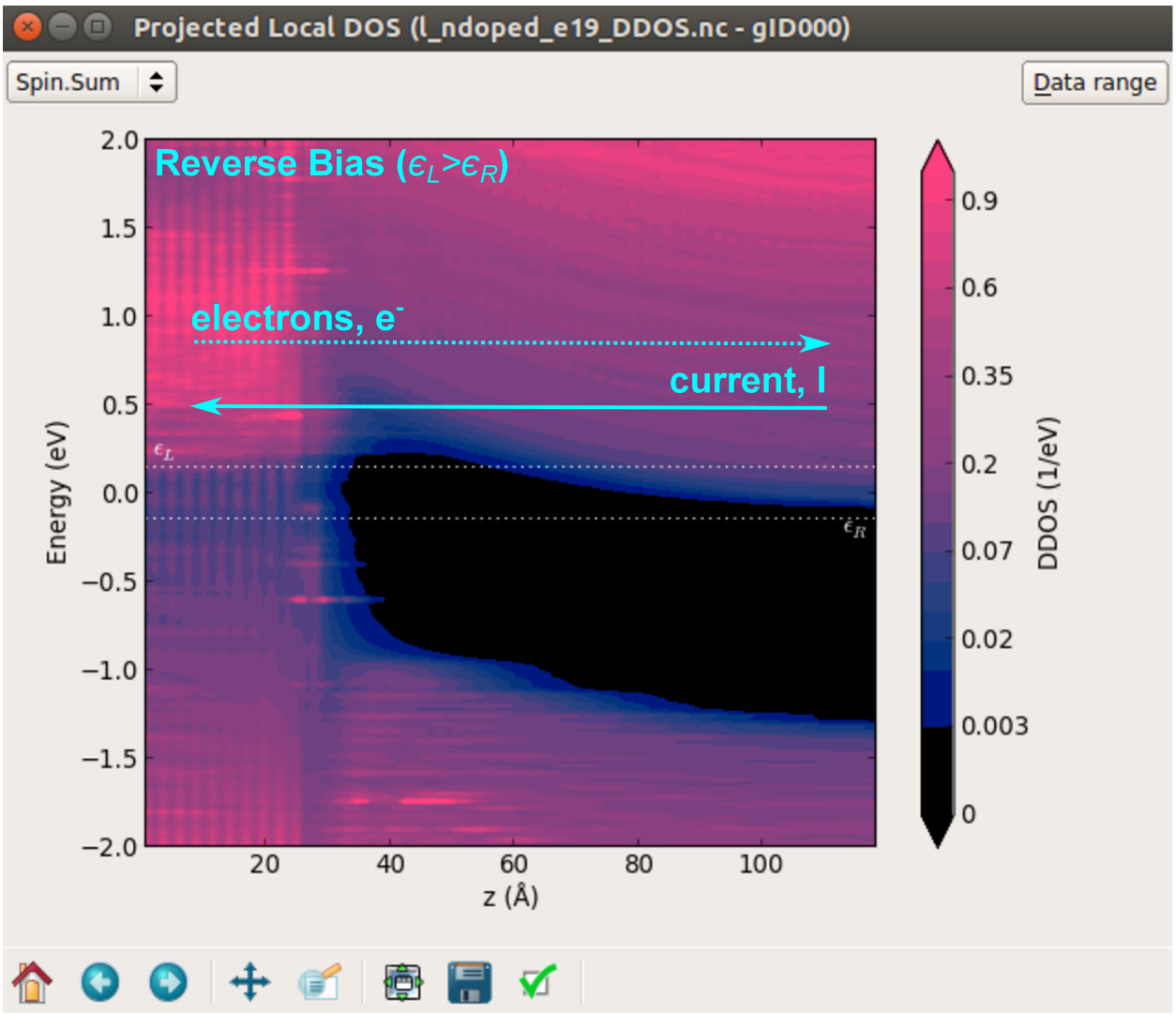
In this section we are going to analyze the EDP of several self-consistent configurations at different V_{bias} and the projected local DOS of the self-consistent configurations at -0.3 V and 0.3 V. You can perform these analyses following the steps described above. You can download the required scripts here (`l_ndoped_e19_EDP.py` `l_ndoped_e19_DDOS.py`).

The IV curve obtained can be divided in two regions, Forward Bias (V_{bias} > 0 V) and Reverse Bias (V_{bias} < 0 V):



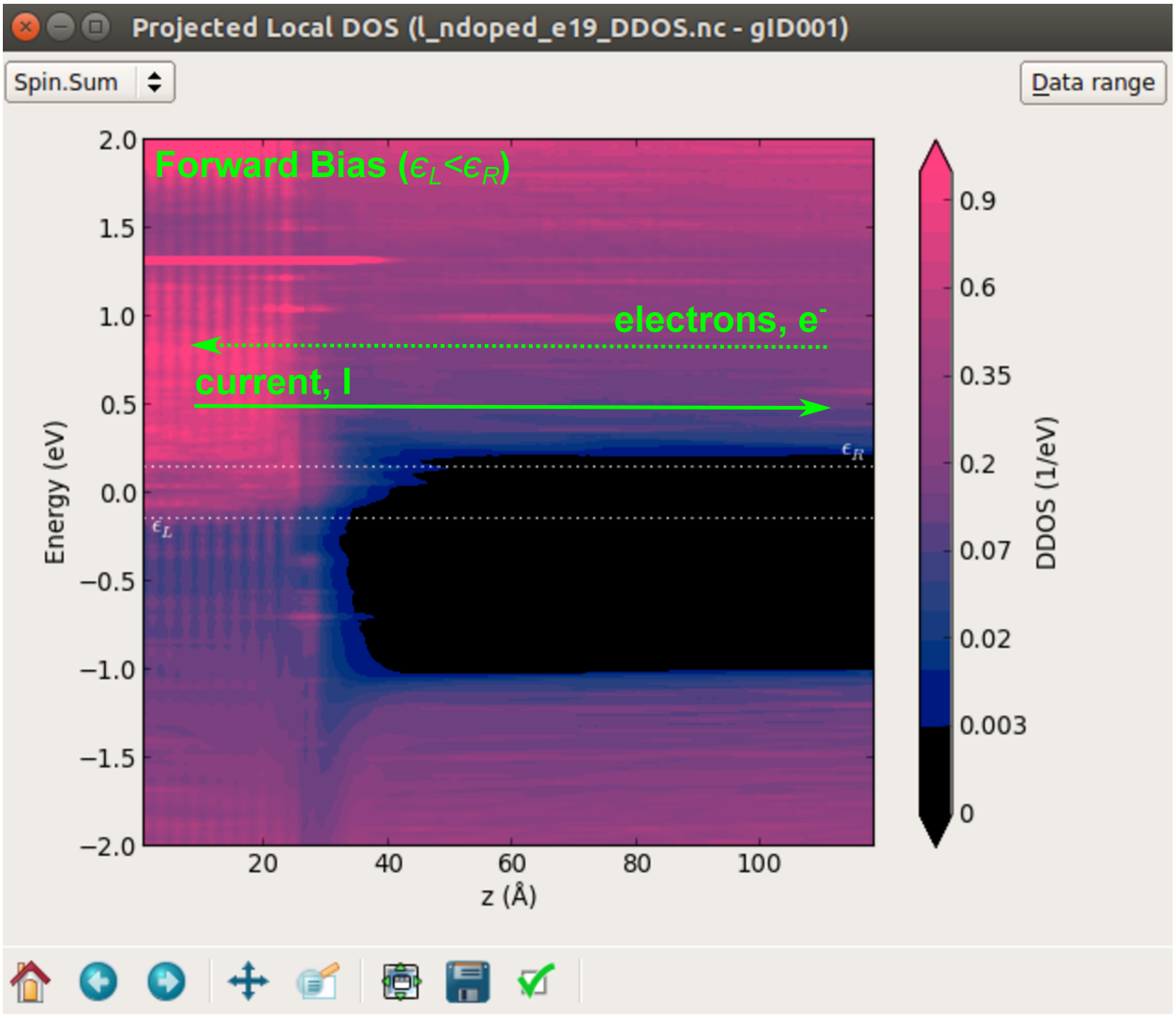
The IV curve shows that the reverse bias current is lower than the forward bias current. This can be undertood looking to the projected local DOS plots.

The figure below shows the projected local DOS at -0.3 V.



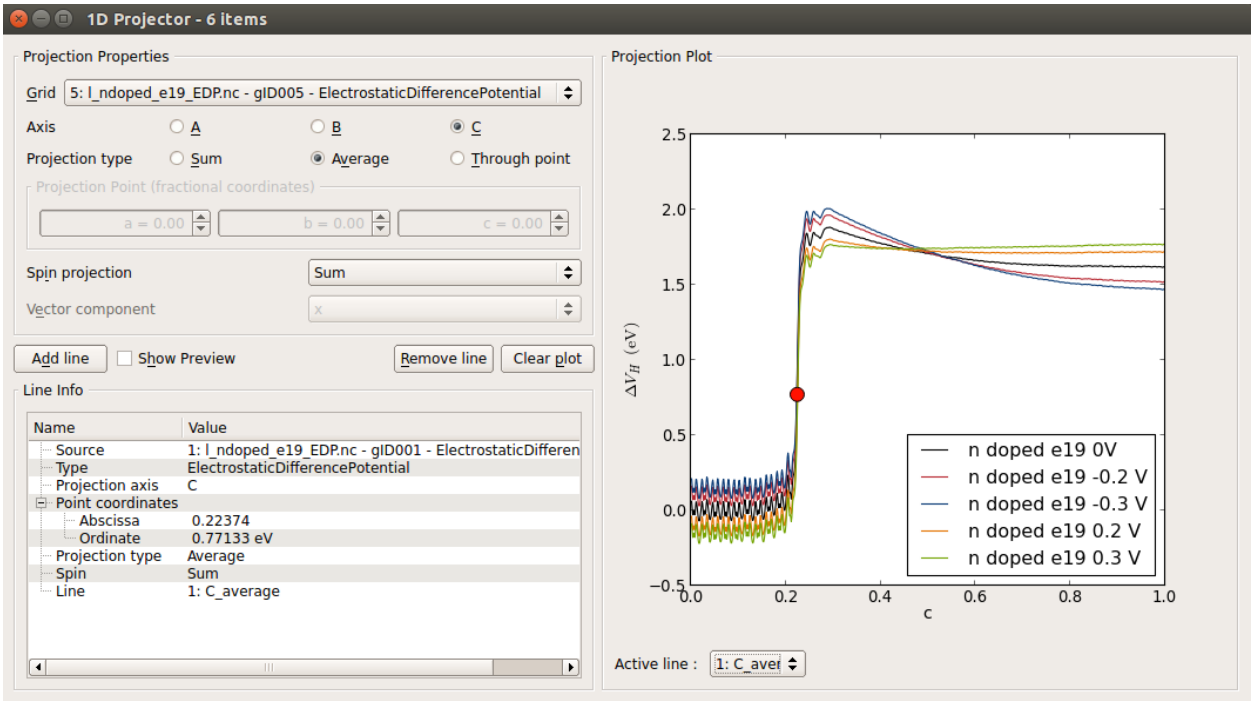
In reverse bias conditions, the chemical potential of the left electrode is higher in energy than that of right electrode. Consequently, there will be a net flow of electrons from the left to the right electrode, so that the current will propagate in the opposite direction. However, the electrons have to overcome a potential barrier at interface due to the presence of the depletion region, and consequently the current will be low.

In the next plot we show the projected local DOS at 0.3 V.



It can be seen that in forward bias conditions, the chemical potential in the right electrode is larger than that of the left electrode. Consequently, there will be a net flow of electrons from the right to the left electrode, so that the current will propagate in the opposite direction. However, at difference with the reverse bias case, here the electrons do not have to overcome any potential barrier, so that the current will be higher.

Finally, in the figure below the EDP along the the C axis of the long n-type doped device at different V_{bias} is shown:



Notice how applying a bias has also an effect on the EDP and on the width of the depletion region.

Warning: In the above plot, the EDP profiles are relative to $(\mu_L + \mu_R)/2$, which is how the potential are defined in the calculation. For a more intuitive picture, one should plot the EDP profiles relative to μ_L.

References

3.2.7 Bi_2Se_3 topological insulator

Version: 2016.2

Topological insulators are electronic materials that have a bulk band gap like an ordinary insulator, but still support conducting states on their surface. It can be shown that these states are topologically protected due to a non-trivial topological index of the material and cannot be removed by any distortions. The 2D energy–momentum relation of these surface states has a Dirac cone structure similar to that of graphene. Topological insulators therefore constitute a new class of quantum matter governed by exotic physics that may some day be exploited in electronic devices. Refs.[1] and[2] provide recent reviews of this highly active field of research.

In this tutorial you will learn how to use ATK-DFT to study the Bi_2Se_3 compound, which is a 3D strong topological insulator. Nonequilibrium Greens function DFT calculations were recently reported for a Bi_2Se_3 thin film connected to leads in a two-terminal device setup[3]. However, this tutorial focuses on bulk calculations and properties of the surface states. In particular, you will:

•Use the QuantumATK Crystal Builder to construct the Bi_2Se_3 crystal structure.

[1] M.ăZ. Hasan and C.ăL. Kane. Colloquium: Topological insulators. *Rev. Mod. Phys.*, 82:3045–3067, Nov 2010. doi:10.1103/RevModPhys.82.3045.

[2] Xiao-Liang Qi and Shou-Cheng Zhang. Topological insulators and superconductors. *Rev. Mod. Phys.*, 83:1057–1110, Oct 2011. doi:10.1103/RevModPhys.83.1057.

[3] Po-Hao Chang, Troels Markussen, Soren Smidstrup, Kurt Stokbro, and BranislavăK. Nikolic. Nonequilibrium spin texture within a thin layer below the surface of current-carrying topological insulator Bi2Se3: A first-principles quantum transport study. *Phys. Rev. B*, 92:201406, 2015. doi:10.1103/PhysRevB.92.201406.

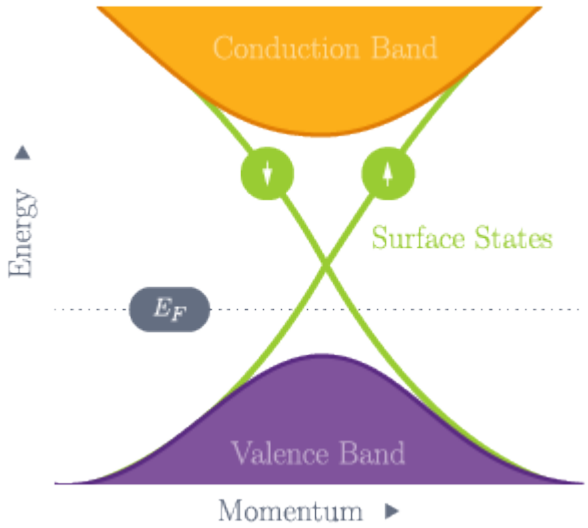


Fig. 3.28: The valence and conduction bands of a topological insulator are separated by a non-zero band gap, but bands due to surface states cross the band gap and lead to electronic conduction at the surface of the material.

•Investigate the bulk Bi_2Se_3 band structure with and without the SOC.

•Construct a Bi_2Se_3 slab configuration and compute the SOC band structure, which exhibits the surface states mentioned above.

•Compute the density of states (DOS) around the Fermi energy, which nicely illustrates the Dirac cone inside the bulk band gap.

•Investigate the surface state penetration depth into the material.

•Plot the spin-resolved Fermi surface of surface states, and investigate how the in-plane spin orientation varies on the Fermi surface.

•Calculate the topological invariants for Bi_2Se_3.

Note: It is essential to include the spin-orbit coupling (SOC) to correctly describe the electronic structure of a topological insulator.

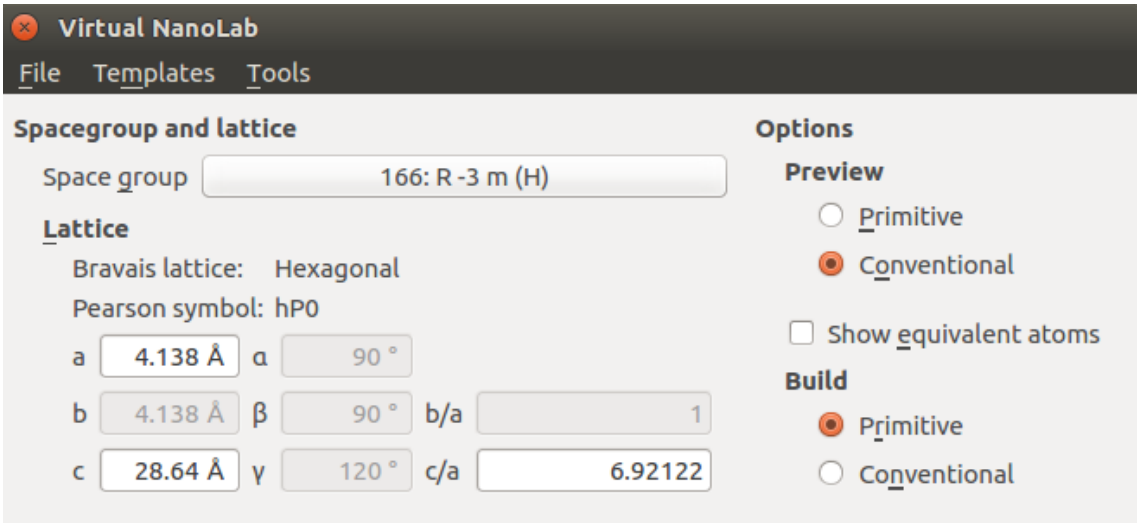
Build the Bi_2Se_3 crystal

Open the QuantumATK Builder and click *Add* → *From Plugin* → *Crystal Builder*. As described in[4], the Bi_2Se_3 crystal is rhombohedral and belongs to space group $R\bar{3}m$. It has lattice constants a=4.138 Å and c=28.64 Å, and internal coordinates μ=0.399 for Bi sites and ν=0.206 for Se sites. The basic Bi_2Se_3 unit cell, called a quintuple layer (QL), contains 5 atoms: One Se atom at coordinate (0, 0, 0), Bi atoms at internal coordinates $(\pm\mu, \pm\mu, \pm\mu)$, and Se atoms at $(\pm\nu, \pm\nu, \pm\nu)$.

[4] Wei Zhang, Rui Yu, Hai-Jun Zhang, XiăDai, and Zhong Fang. First-principles studies of the three-dimensional strong topological insulators Bi2Te3, Bi2Se3 and Sb2Te3. *New Journal of Physics*, 12(6):065013, 2010. doi:10.1088/1367-2630/12/6/065013.

Lattice

In the Crystal Builder, choose the *Trigonal* space group number 166 with *Hexagonal* setting, and enter the lattice parameters:



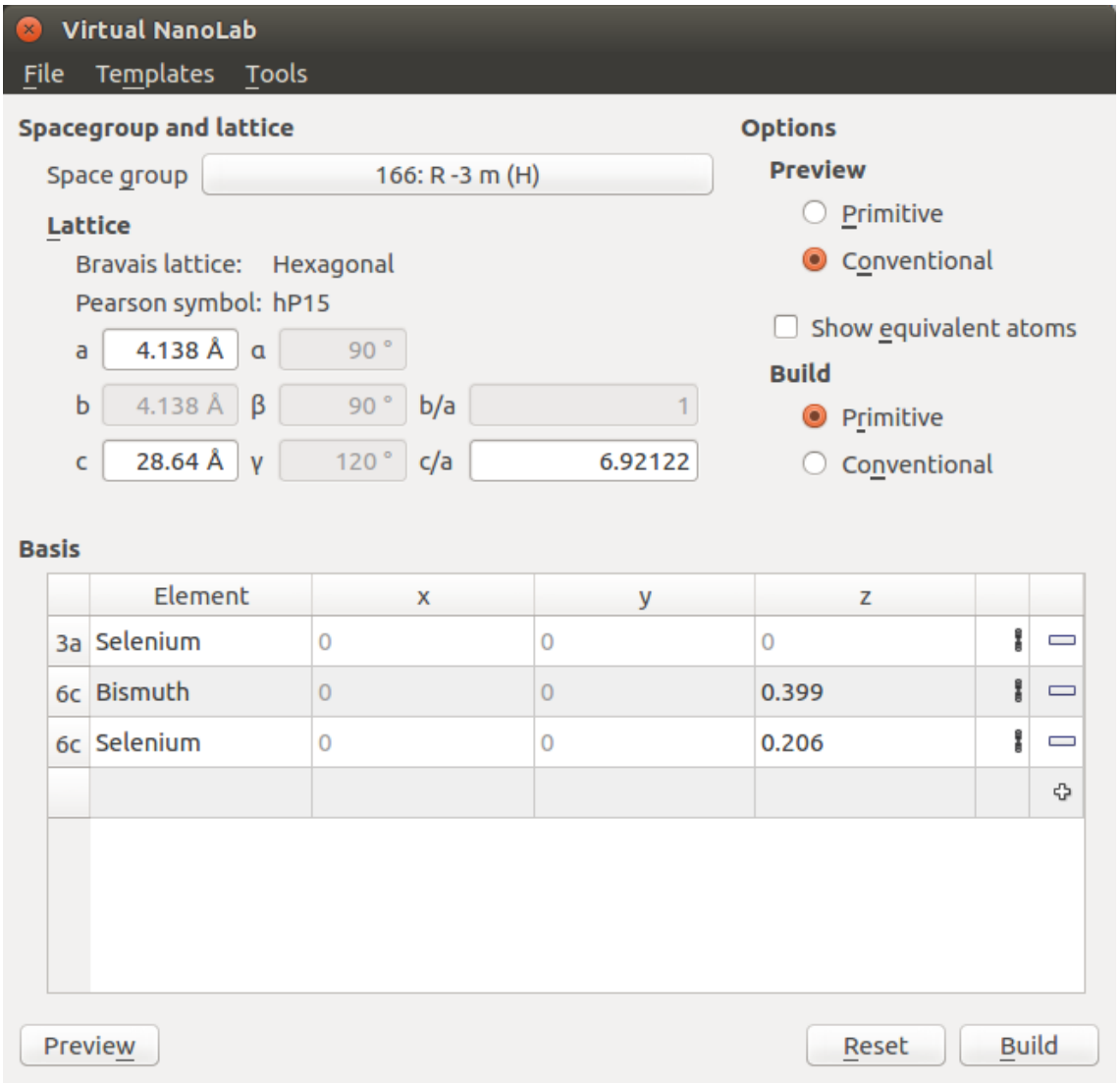
Basis

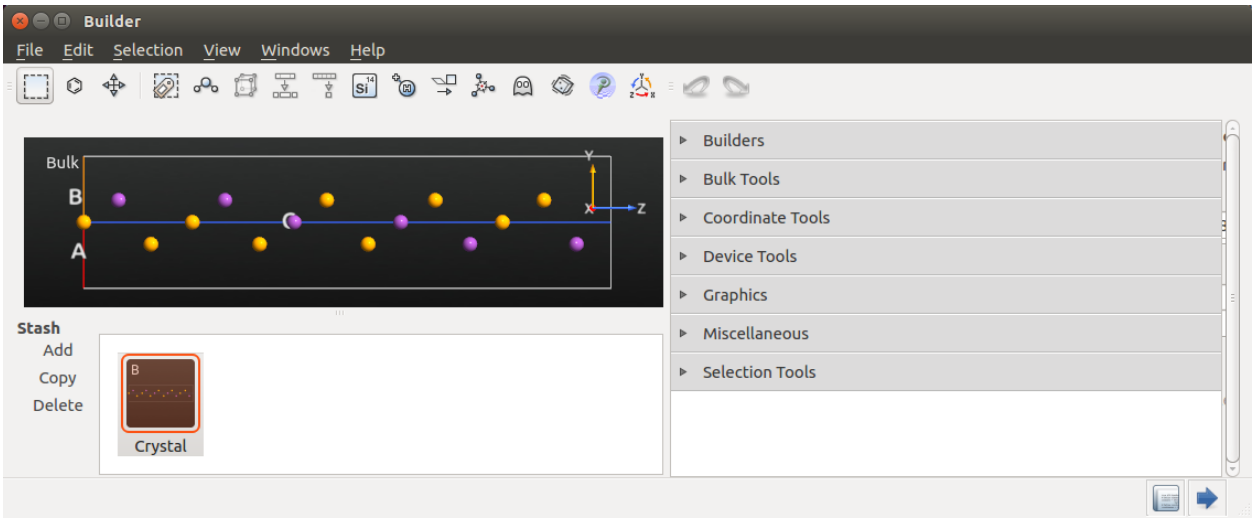
Then specify the atomic basis with which to decorate the lattice, i.e. enter the internal coordinates of the unique sites of the Bi_2Se_3 crystal and the elements that should be at those sites:

1. Click the icon and select site *3a: (0,0,0)*, and change the element from hydrogen to selenium.

2. Use the same procedure twice for specifying the remaining sites, but choose *6c: (0,0,z)* for the site type, and enter the internal coordinates z=0.399 for bismuth and z=0.206 for selenium.

   The Crystal Builder window should now look like illustrated below. Click *Build* to build the bulk configuration – it will then appear in the Builder Stash.





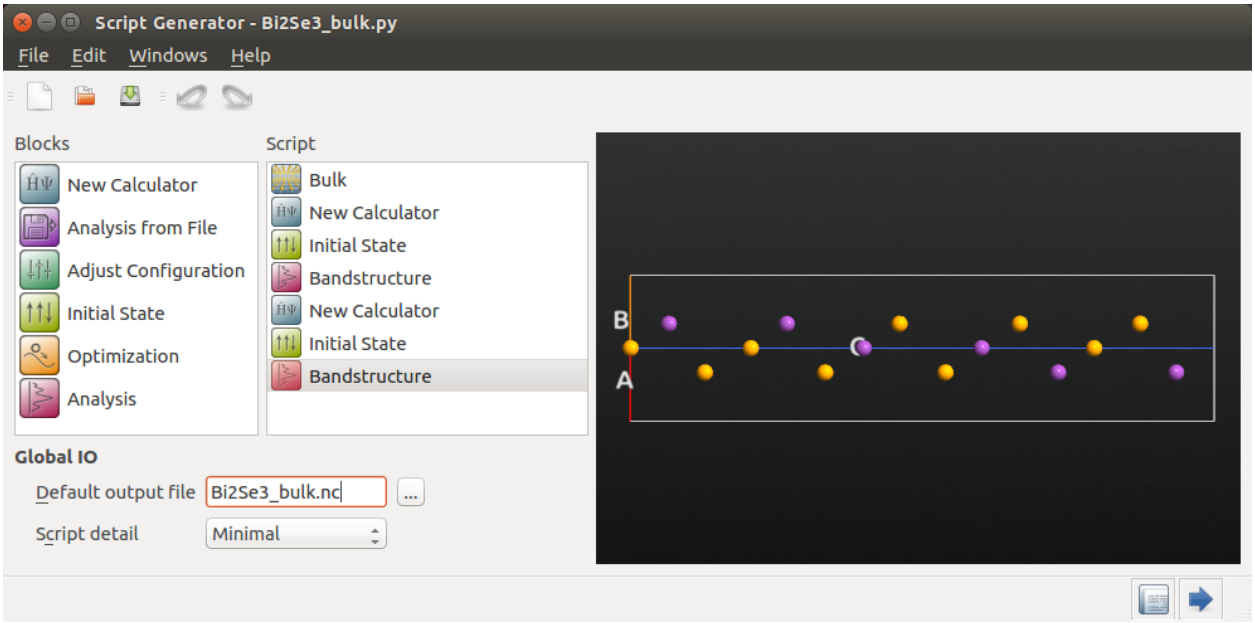
You have now built the Bi_2Se_3 bulk configuration with 3 QLs and 15 atoms in total. Continue to the next section to start doing some DFT calculations for this material. If needed, you can also download the bulk configuration here:

`Bi2Se3_bulk_configuration.py`

Bi_2Se_3 bulk band structure

Send the bulk configuration to the Script Generator, and set up a Python script that uses ATK-DFT to calculate the band structure with and without the spin-orbit coupling. The script should first calculate the self-consistent GGA state and perform band structure analysis, and then use that state as an initial guess for the self-consistent SOGGA state, again followed by band structure analysis.

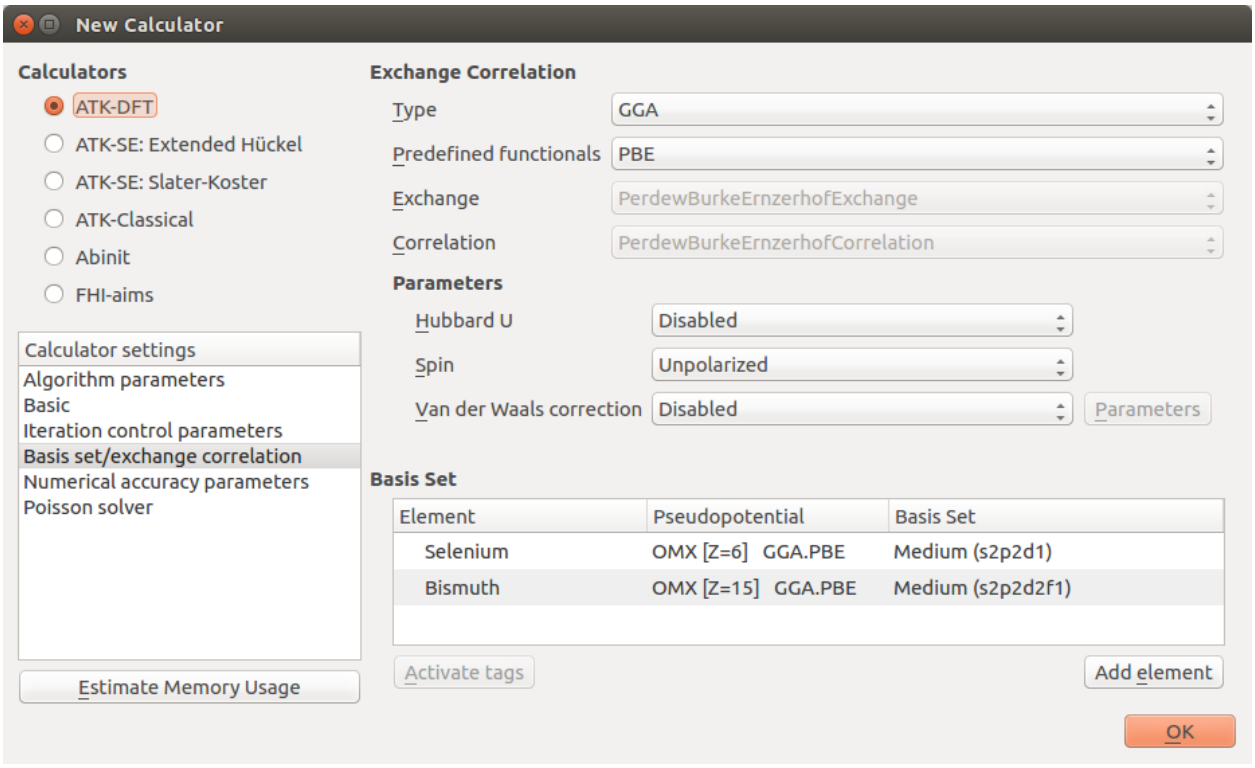
The Script Generator now contains the bulk configuration. Add the New Calculator, *Analysis* → *Bandstructure*, and Initial State blocks, as illustrated below. Also change the default output file to `Bi2Se3_bulk.nc`.

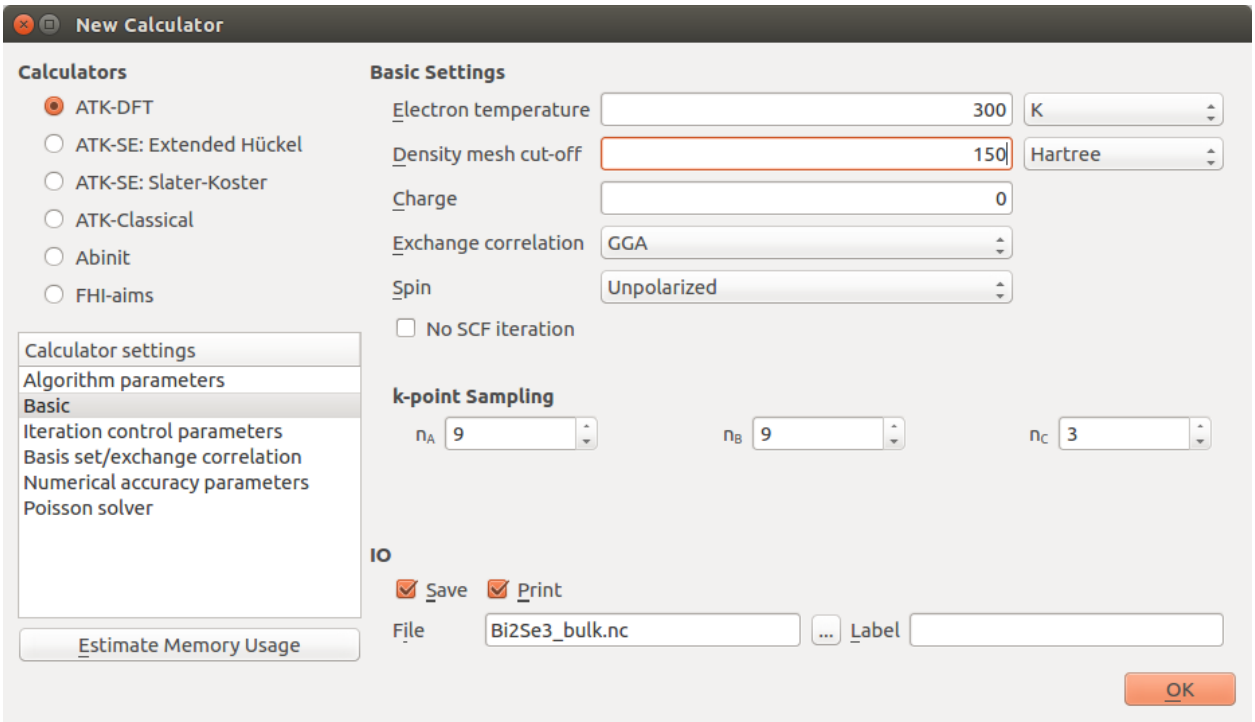


GGA calculation

Open the first New Calculator block to edit it:

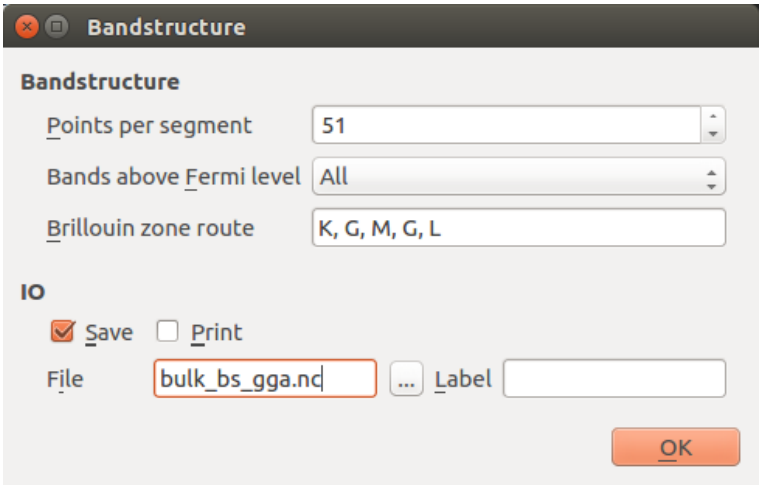
- Make sure to choose the ATK-DFT calculator.
- In the *Basis set/exchange correlation* settings, choose *GGA* and change the pseudopotentials to the *OMX* family with *Medium* basis sets for both Se and Bi.
- In the *Basic settings* tab, increase the density mesh cut-off to 150 Hartree, and set up a 9x9x3 k-point grid. Click *OK* to save the settings.





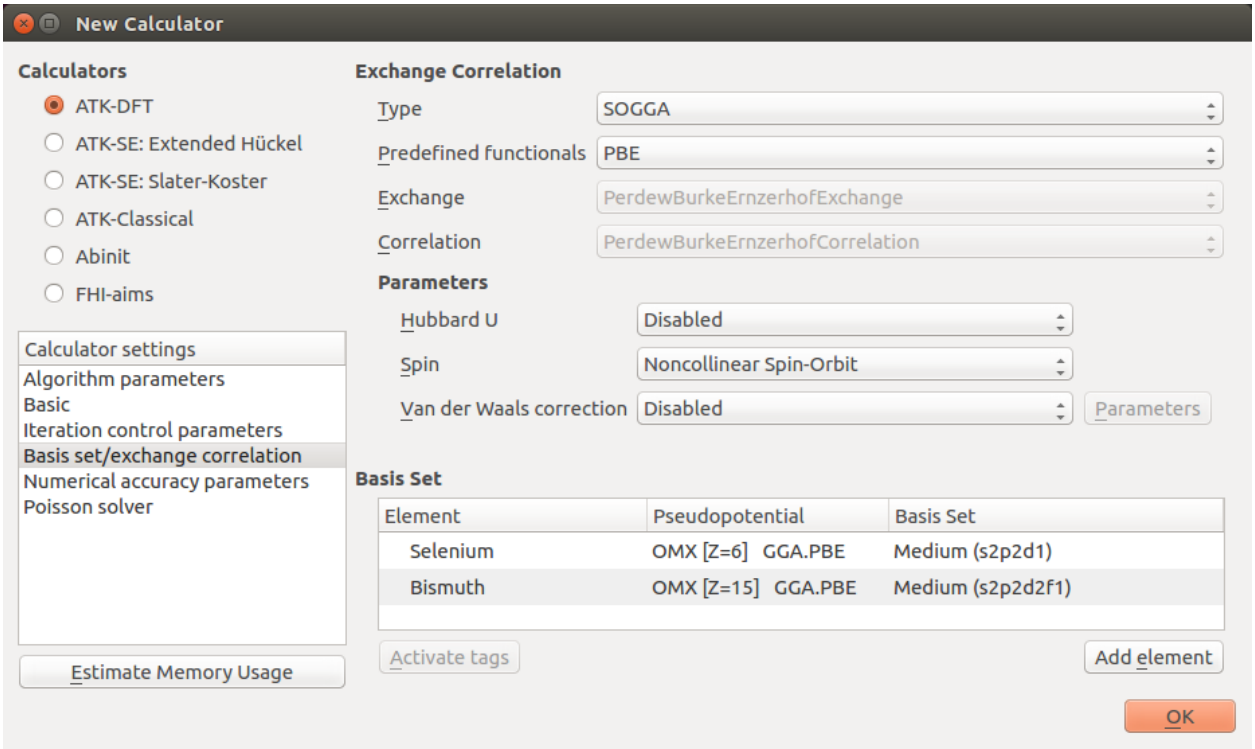
Then open the first Bandstructure block and edit it:

- Increase *points per segment* to 51 to get a nicely resolved band structure.
- Change the Brillouin zone route to *K, G, M, G, L*.
- Change the IO file to `bulk_bs_gga.nc`.

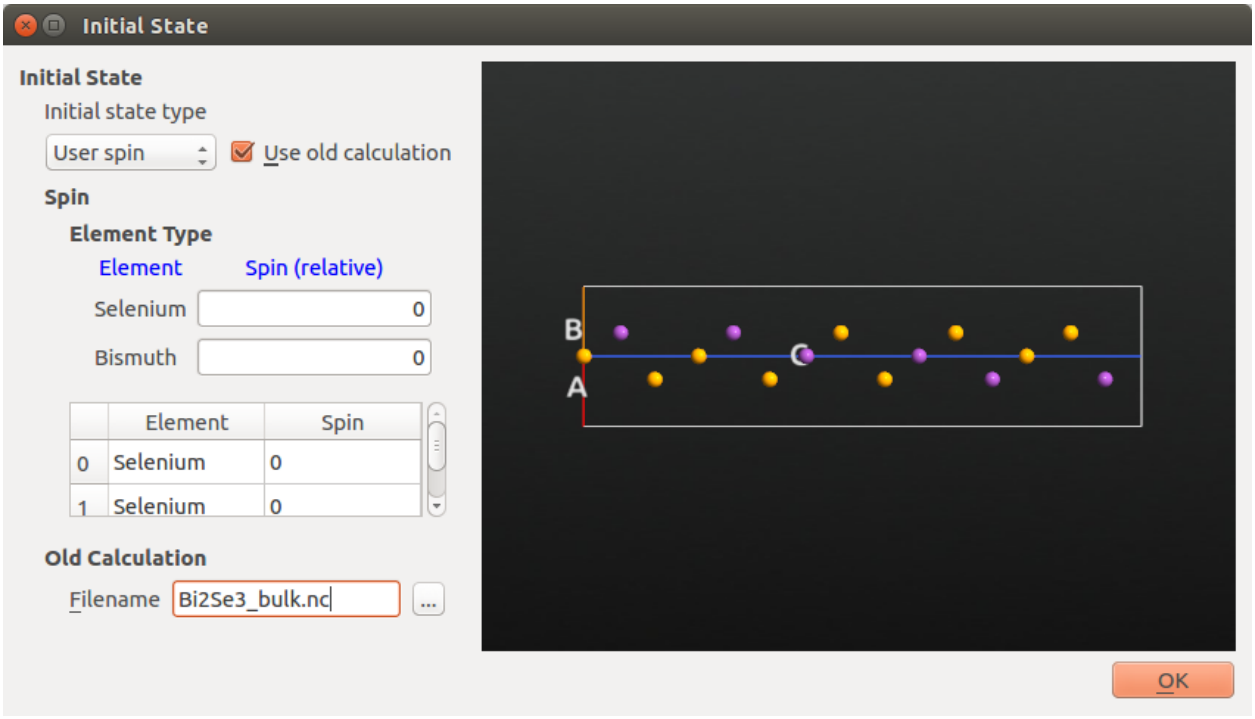


SOGGA calculation

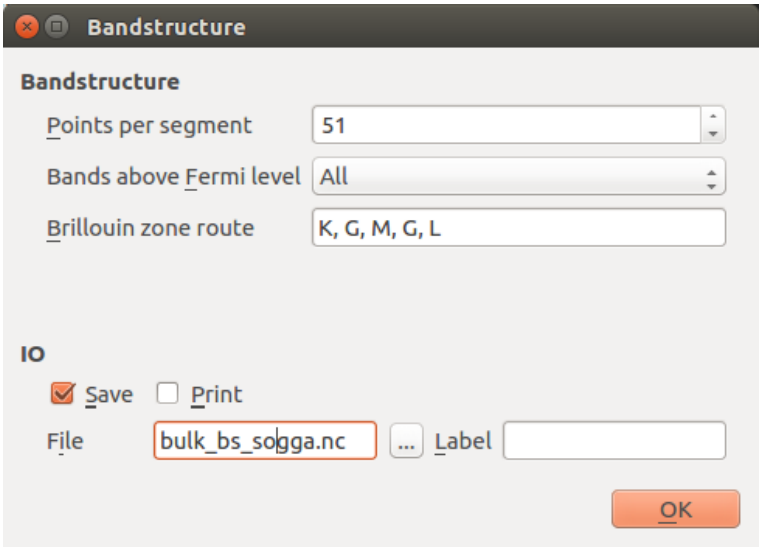
Open the second New Calculator block, and edit it such that the settings are identical to those in the first calculator, but set the *Spin* parameter to *Noncollinear Spin-Orbit* in order to do a SOGGA calculation. Remember to also set the pseudopotentials, density cut-off, and k-points.



Next, open the second Initial State block, and select *User spin* for *Initial state type*, tick the *Use old calculation* option, and enter the file name `Bi2Se3_bulk.nc` from which the GGA state should be read.



Finally, set up the second Bandstructure block identically to the first one, but use `bulk_bs_sogga.nc` for the IO file.



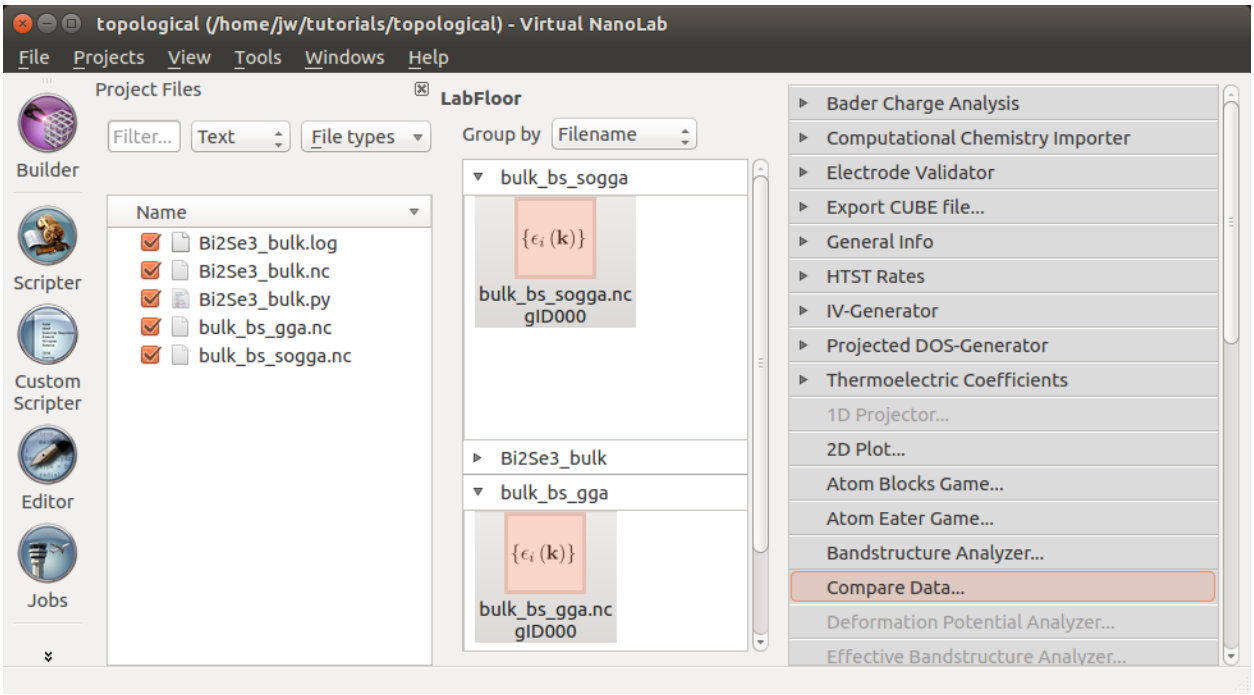
Results

The Python script is now ready. Save it as `Bi2Se3_bulk.py`. If needed, you can also download it here: `Bi2Se3_bulk.py`

Execute it using the Job Manager or in a Terminal. The job will take a while, but the wall time can be brought down to around 40 minutes if executed in parallel with 2 MPI processes on 2 CPUs. Note that this requires in total about 15 GB of available memory!

Tip: The tutorial *Basic QuantumATK Performance Guide* offers tips on QuantumATK performance, including how to deal with memory issues.

Once the calculation has finished, the QuantumATK **LabFloor** contains three sets of data: *Bi2Se3_bulk*, *bulk_bs_gga*, and *bulk_bs_sogga*. The last two contain the GGA and SOGGA band structures. Highlight both band structure objects, and use the **Compare Data** plugin to plot both of them.



The plot is shown below. Including spin-orbit coupling in the calculations (SOGGA) has a significant impact on the band structure, and widens the direct band gap at the Γ-point However, we do not see any bands crossing the Fermi level (the smallest SOGGA band gap is 0.3 eV), so the Bi_2Se_3 bulk material is an insulator.

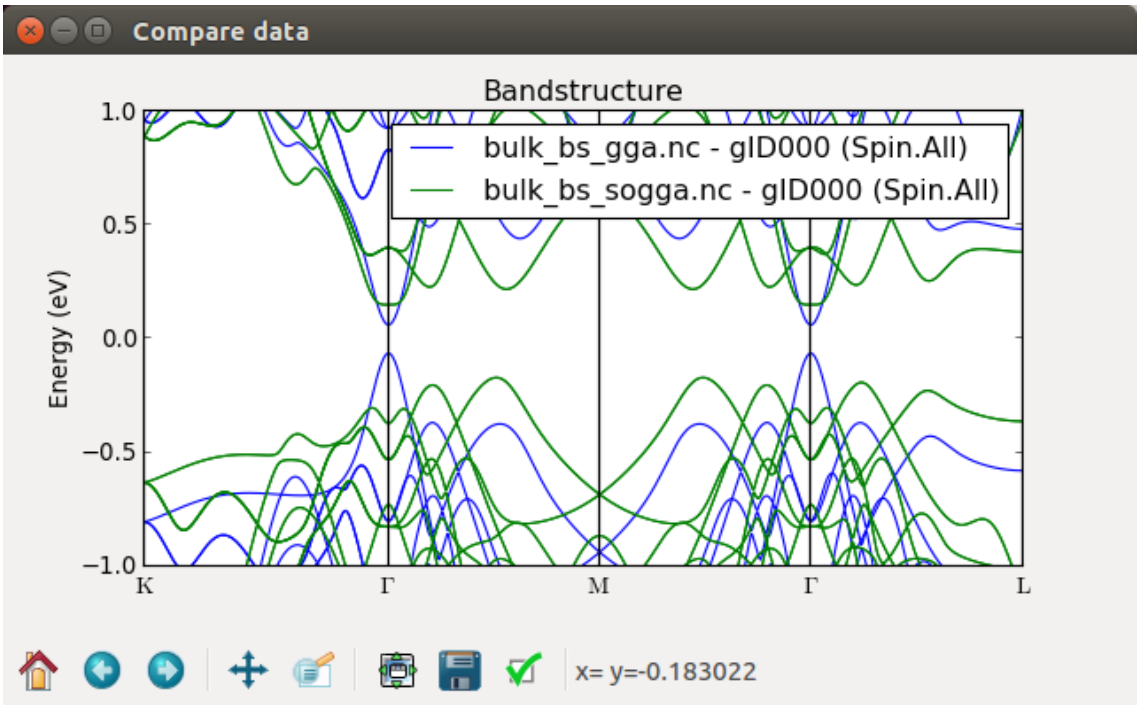


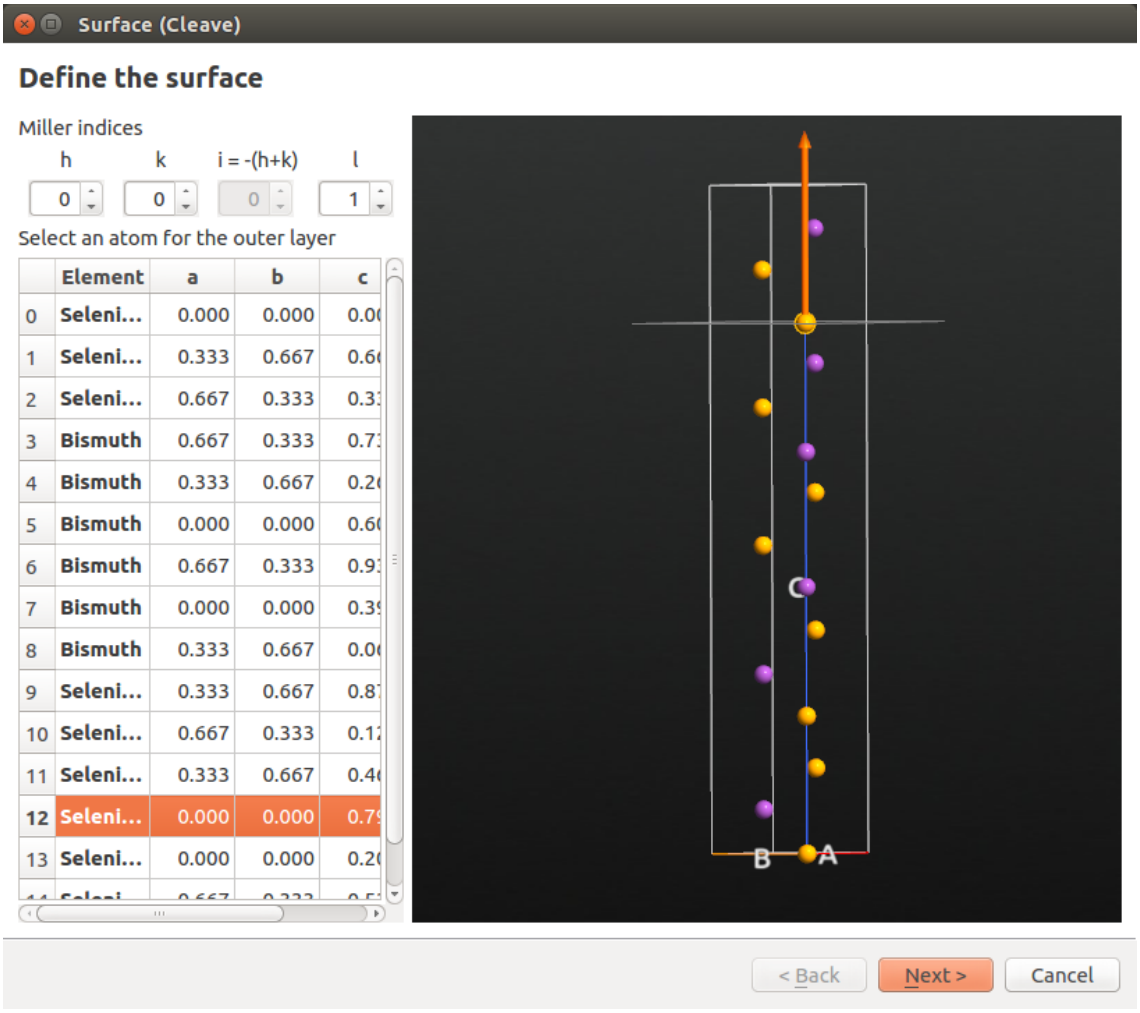
Fig. 3.29: Comparison of the GGA (blue) and SOGGA (green) band structures of bulk Bi_2Se_3.

Bi2Se3 surface: Spin-orbit band structure

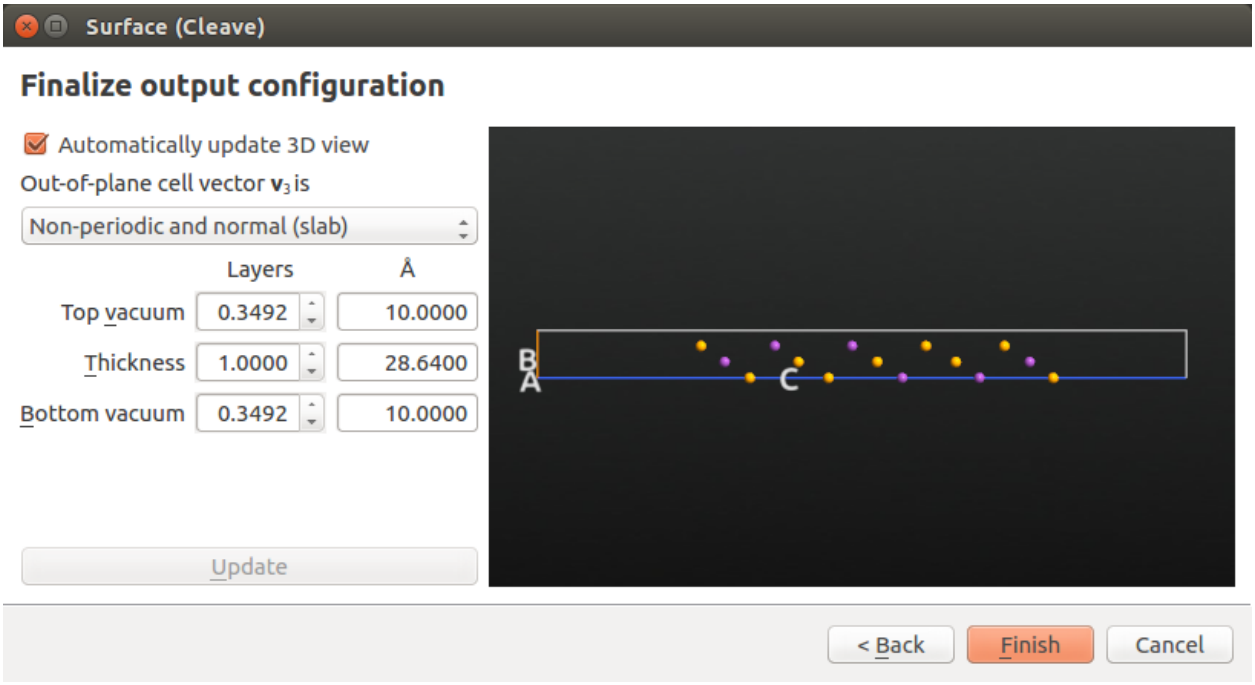
You will now create a Bi_2Se_3 slab configuration, and compute the SOGGA band structure, which will exhibit the topologically protected surface states crossing the band gap around the Γ-point.

Constructing the surface

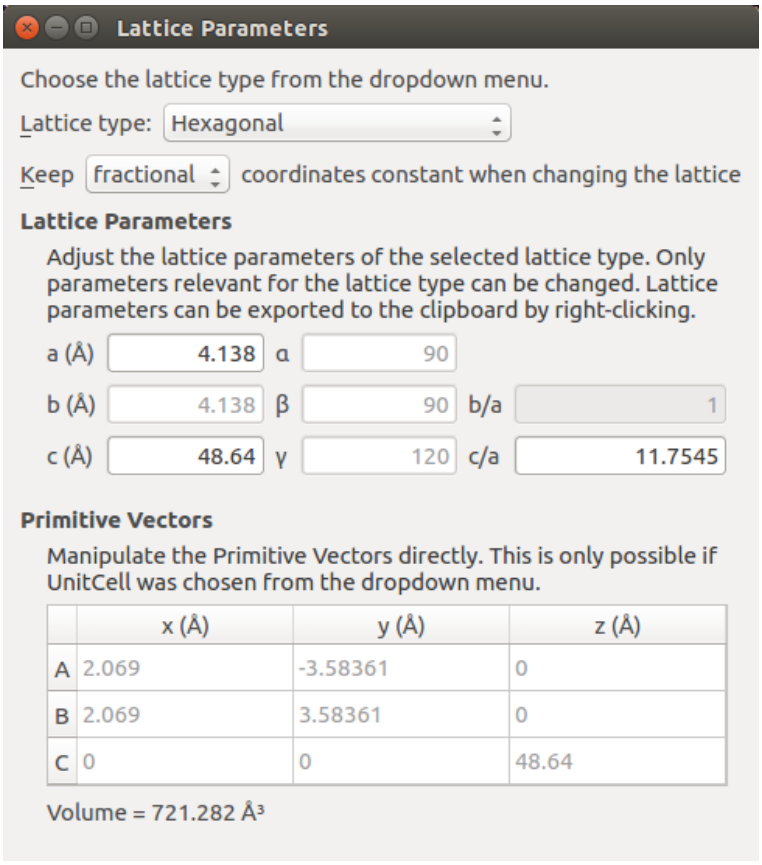
Back in the **Builder**, highlight the Bi_2Se_3 bulk configuration (named Crystal), and open the *Builders* → *Surface (Cleave)* plugin. Choose Miller indices (0001), and select selenium atom #12 as the top atom in the slab. Then click *Next*.



Leave settings at defaults in the Define surface lattice options window. In the Finalize output configuration options, choose a *Non-periodic and normal (slab)* out-of-plane cell vector, and enter 10 Å for both top and bottom vacuum. Then click *Finish*.



The slab configuration, named Crystal (0001), has now appeared in the Stash. As a final touch, highlight the item and use the *Bulk Tools* → *Lattice Parameters* plugin to convert the lattice type to *Hexagonal* while keeping the fractional coordinates constant:

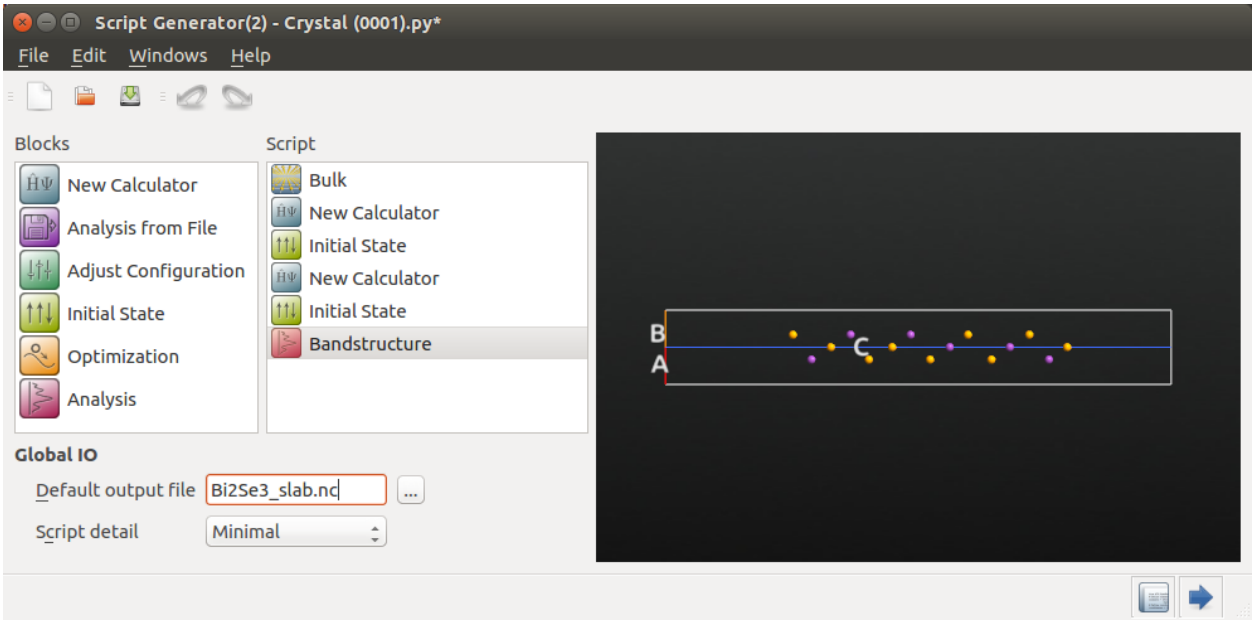


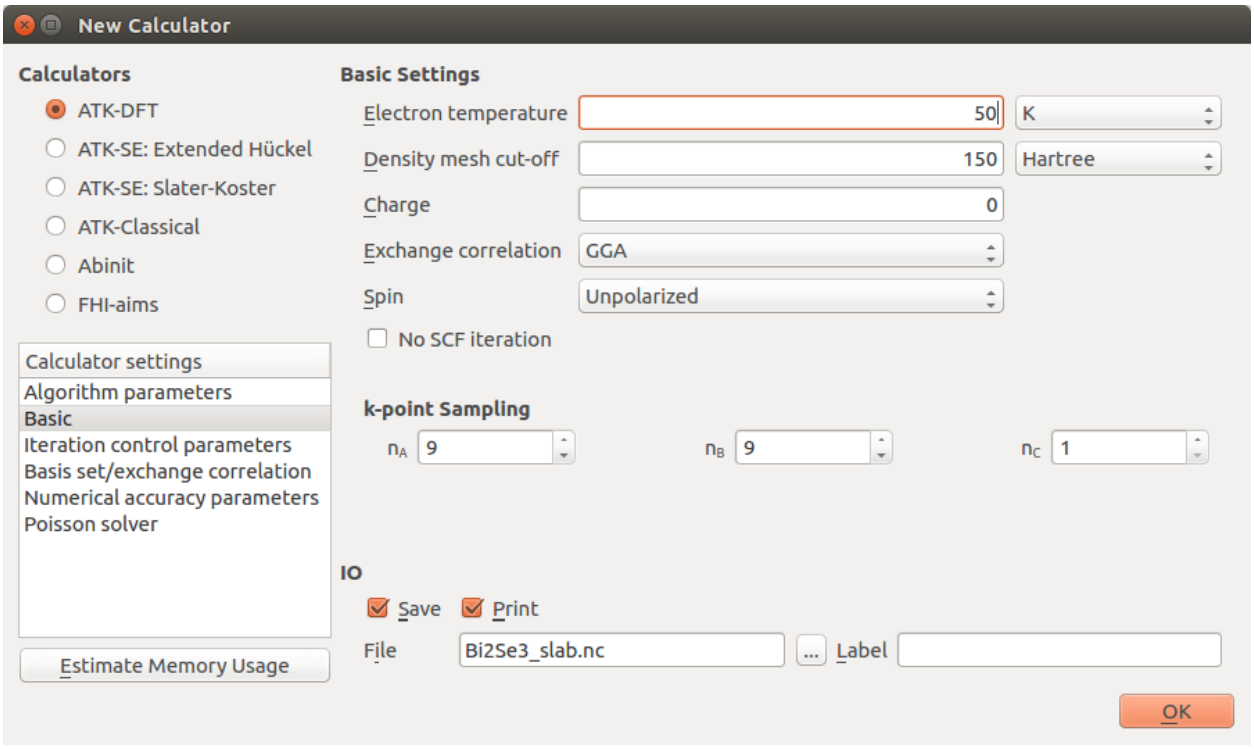
DFT-SOGGA calculations

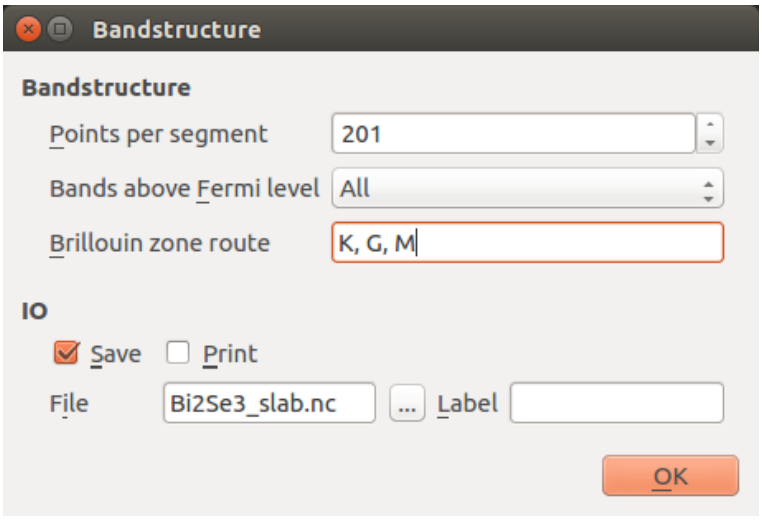
You can now send the slab configuration to the **Script Generator**, and set up the SOGGA band structure calculation similarly to what you did above for the bulk, but with slight changes:

- You will not need the GGA band structure analysis.
- Use `Bi2Se3_slab.nc` as default output file.
- Use a 9x9x1 k-point grid (the slab is not periodic along the C-direction).
- Decrease the electron temperature from 300 to 50 K. This will improve the DFT description of electron occupations close to the Fermi level.
- Use 201 points per segment for the SOGGA band structure analysis, and the *K–G–M* Brillouin zone route.

Alternatively, you can simply skip the above and go directly to the next section to use a pre-made Python script with a few additional features.







Convenient Python script

Click here: `Bi2Se3_slab.py` to download a Python script that was made using the procedure outlined above, but also has a few extra features:

- The MemoryUsage functionality is used to estimate the memory required for the GGA and SOGGA calculations immediately before they are executed.
- The SOGGA calculator is constructed as a copy of the GGA calculator, and is then modified accordingly.
- A special density mixing algorithm is used for the non-collinear calculation.

Run the calculation using the Job Manager or in a Terminal. The job will take roughly 30 minutes if executed in 2

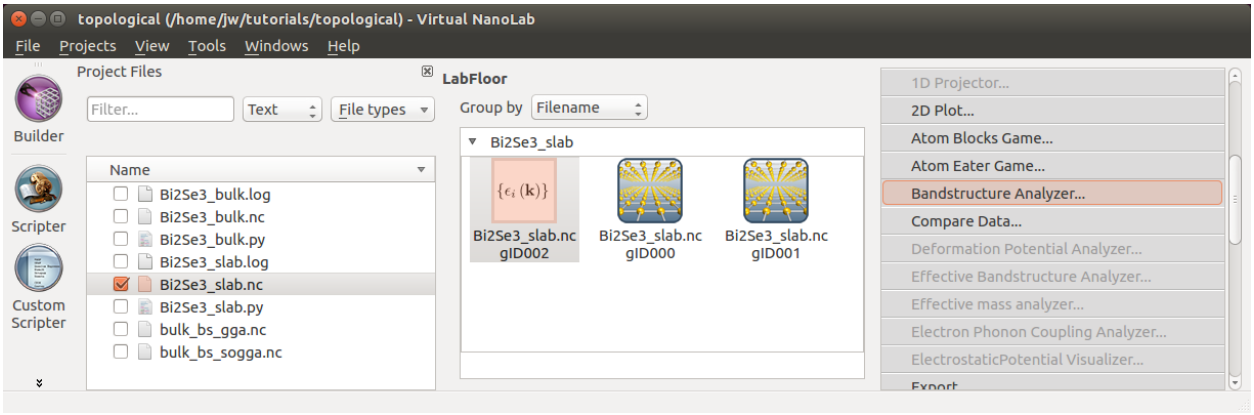
parallel MPI processes on 2 CPUs.

Hint: If you look at the log file during the calculation, you will see that the estimated peak memory requirements are 756 MB per MPI process for the GGA calculation, and 3.8 GB per MPI process for the SOGGA calculation. These are estimates, and in practice you should expect the full calculation to require almost 5 GB per MPI process.

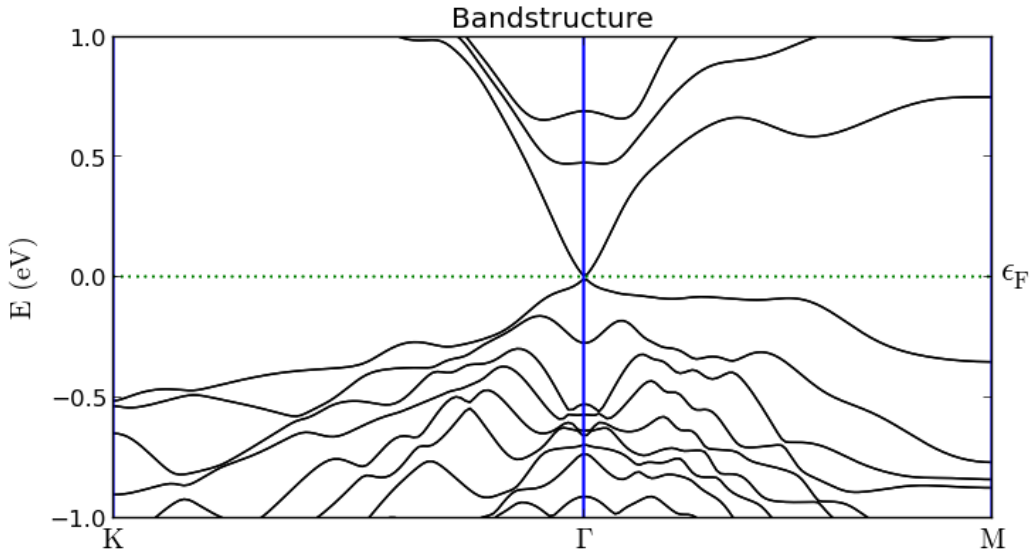
```
GGA:
+-------------------------------------------------------+
| Memory usage estimate                                 |
| ----------------------------------------------------- |
| Base: Sparse matrices                          44 Mbyte  |
| Base: GridTool+Basis set                      100 Mbyte  |
| Base: SCF history                             442 Mbyte  |
| Base: NLEngine                                 70 Mbyte  |
| Base: Grid terms                               39 Mbyte  |
| Sum of the base terms                         696 Mbyte  |
| ----------------------------------------------------- |
| Peak: Diagonalization                          60 Mbyte  |
| Peak: Grid terms                               19 Mbyte  |
| ----------------------------------------------------- |
| Estimated Peak memory requirement             756 Mbyte  |
+-------------------------------------------------------+

SOGGA:
+-------------------------------------------------------+
| Memory usage estimate                                 |
| ----------------------------------------------------- |
| Base: Sparse matrices                         272 Mbyte  |
| Base: GridTool+Basis set                      100 Mbyte  |
| Base: SCF history                            2726 Mbyte  |
| Base: NLEngine                                 70 Mbyte  |
| Base: Grid terms                              157 Mbyte  |
| Sum of the base terms                        3326 Mbyte  |
| ----------------------------------------------------- |
| Peak: Diagonalization                         446 Mbyte  |
| Peak: Grid terms                               58 Mbyte  |
| ----------------------------------------------------- |
| Estimated Peak memory requirement            3772 Mbyte  |
+-------------------------------------------------------+
```

Results

The QuantumATK **LabFloor** should now contain the results of the calculation. Select the band structure object, and use the **Bandstructure Analyzer** to plot it.



The calculated band structure of the Bi_2Se_3 slab is shown below. Bands are now present around the Fermi level at the Γ-point – these are surface states. There are four such surface states crossing the Fermi level (indicated by red dots); bands 142 and 143 form a single degenerate band immediately below the Fermi energy, while bands 144 and 145 form a degenerate band above E_F. The bulk valence bands are below those surface states (indicated by blue dot). It is clear that the dirac cone is located inside the bulk band gap. There is a tiny gap between the valence and conduction surface bands of size 7 meV. This is a finite size effect from the use of a slab model, which arise from the interaction of the surface bands on opposite sides of the slab. For a larger slab the gap will be reduced.



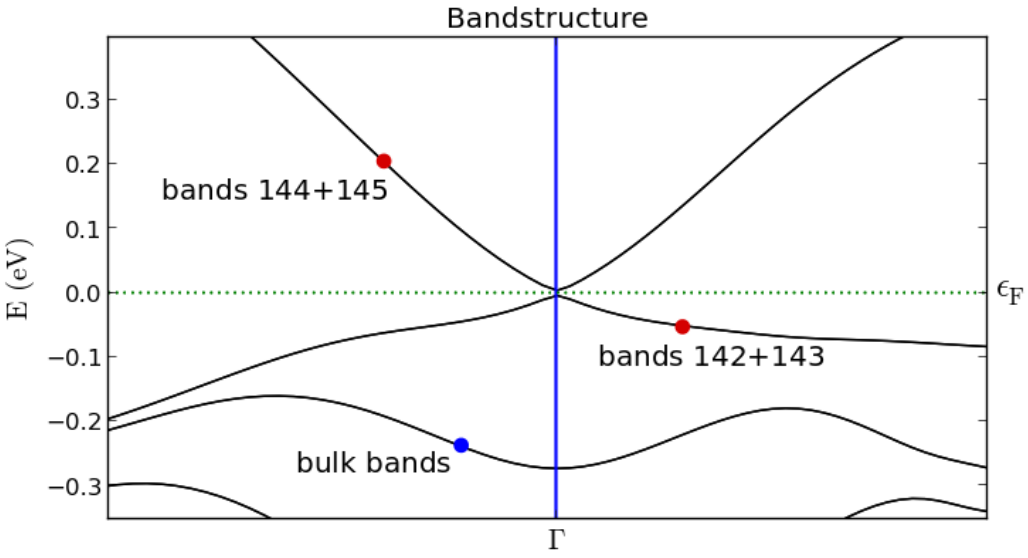
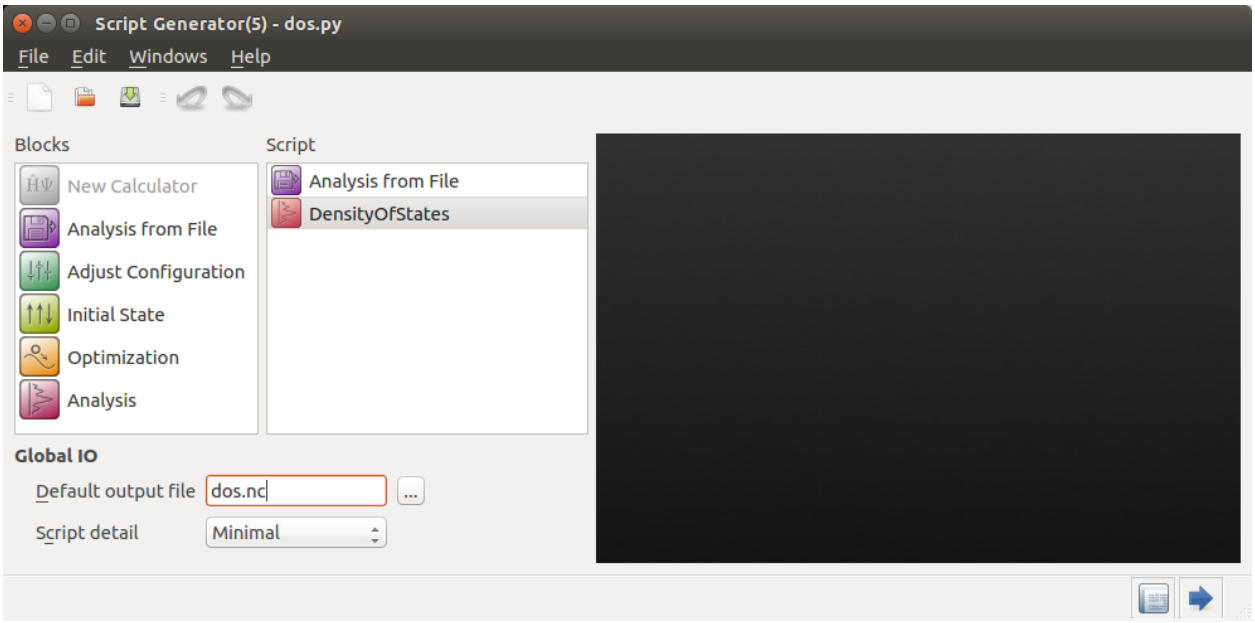


Fig. 3.30: SOGGA bandstructure of the Bi_2Se_3 slab configuration. The bottom panel is a zoom-in on the topologically protected Dirac cone. Both surface state bands (red dots) are both doubly degenerate and are located above the bulk valence bands (blue dot).

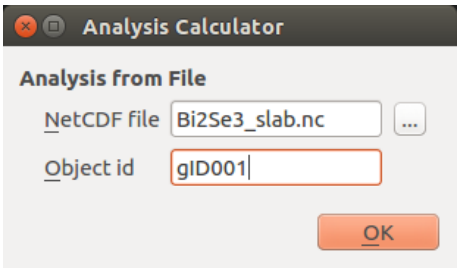
DOS analysis: Dirac cone finger print

As stated in the introduction, the electronic structure of the surface states close to the Fermi level resembles that of a Dirac cone, where the electron momentum depends linearly on the energy. Since the surface states are the only states present inside the bulk energy gap, we should expect the electronic density of states close to E_F to be linear. It is easy to compute and plot the DOS as a post-SCF analysis:

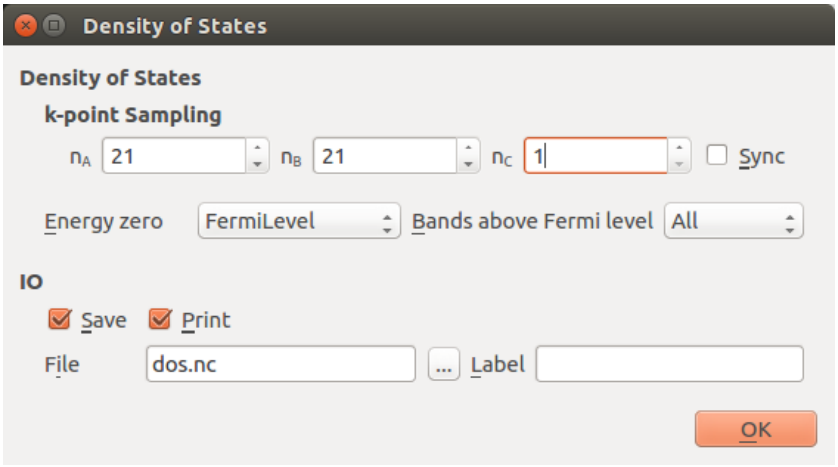
Open the **Script Generator** and add the Analysis from File and *Analysis* → *DensityOfStates* blocks. Change the default output file to `dos.nc`.



Open the first block, and point to the `Bi2Se3_slab.nc` file and Object id *glD001*, which is the self-consistent SOGGA state.



Then open the DensityOfStates analysis block, and set up a dense k-point sampling along k_x and k_y, 21x21x1. You will need to uncheck the *sync* option. Remember: It is important in this case to sample the Γ-point, so we use use an odd k-grid.



Save the script as `dos.py` and run it. You can also download it here: `dos.py`.

Then locate the DOS object on the **LabFloor**, and use the **2D Plot** plugin to visualize it. With a bit of zooming you should be able to see the expected linear dependence of the DOS on energy, as illustrated below. The sharp increase in DOS at E=0.47 eV are due to the lowest bulk conduction bands.

Penetration depth of surface states

An essential feature of the topologically protected surface states close to the Dirac point is that they are highly localized to the surface region. The penetration depth into the bulk material may be as small as 2–3 nm[Page 549, 4]. Moreover, the degenerate surface states observed in a band structure calculation are in each specific k-point composed of states with opposite spin and localized on opposite sides of the slab.

You will now investigate this by performing a **BlochState analysis** in a specific k-point on the Dirac cone for bands 144 and 145. You will then project both Bloch states onto the C-axis of the Bi_2Se_3 slab unit cell, and plot the magnitude and direction of the non-collinear spin vectors as a function of the C-coordinate.

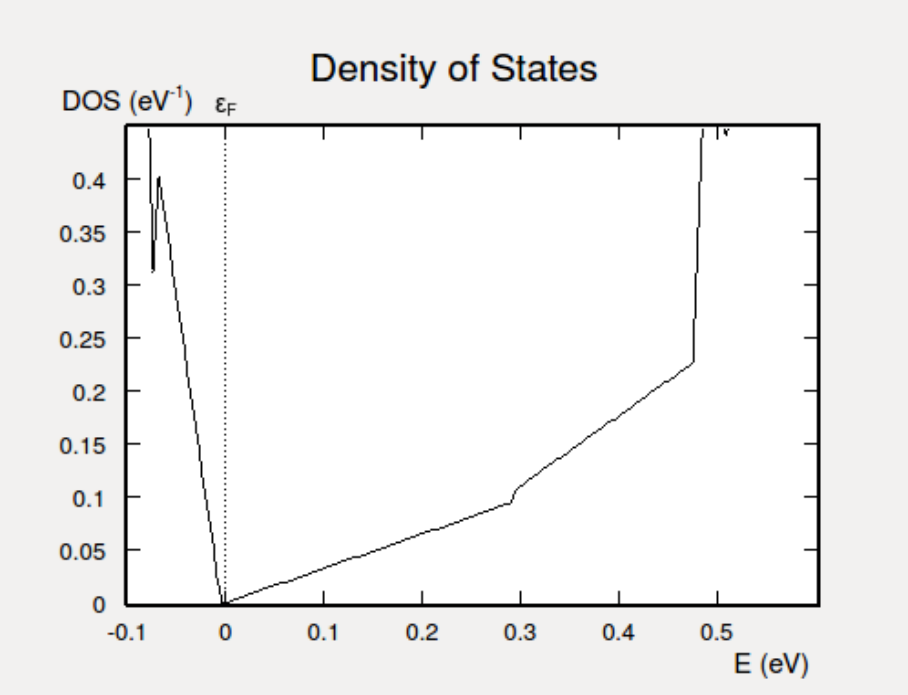


Fig. 3.31: Electronic DOS for the Bi_2Se_3 slab. Due to the Dirac cone, the DOS depends linearly on the energy in the vicinity of the Fermi level.

Calculations

You will need the following two scripts: `bloch_states.py` and `spinVector.py`. Download the scripts and execute the first one (`bloch_states.py`) using the Job Manager or in a Terminal. It reads in the SOGGA state from `Bi2Se3_slab.nc`, performs the two BlochState analyses for k-point [0, 0.04, 0], and then computes the length r and polar angles θ and ϕ for both Bloch states and plots them.

The resulting figure is shown above. Orange and purple dots at the bottom indicate the position of the Se and Bi atoms, respectively. The two states are clearly localized on opposite sides of the slab (red curves) and have very little overlap in the middle of it. The polar angle ϕ is 270ř for both states (blue curves), but the in-plane angle ϕ is 120ř for band 144 and rotated by 180ř to 300ř for band 145 (green curves). The two spin states therefore point in opposite directions!

Fermi surface and spin directions

Finally, it is interesting to study the Fermi surface of the Bi_2Se_3 slab in the vicinity of the Dirac cone. This is easily done by sampling the band structure on a dense k-grid centered at the Γ-point.

Download and execute this pre-made script: `fermi_surface.py`. It reads in the SOGGA state from `Bi2Se3_slab.nc`, performs band structure analysis in all points on a $k_x \times k_y$=51×51 grid, and creates a contour plot of the eigen-energies of the surface state in energy band 144.

Note: For users of QuantumATK 2017: Use instead the script `fermi_surface_2017.py`.

The resulting plot is illustrated above. As expected for a Dirac cone, the Fermi surface is circular close to the apex (red dot), but in this case it turns more hexagonal in shape for higher energies.

The Python script has also extracted the contour corresponding to E_F=0.15 eV and computed the Bloch state for every

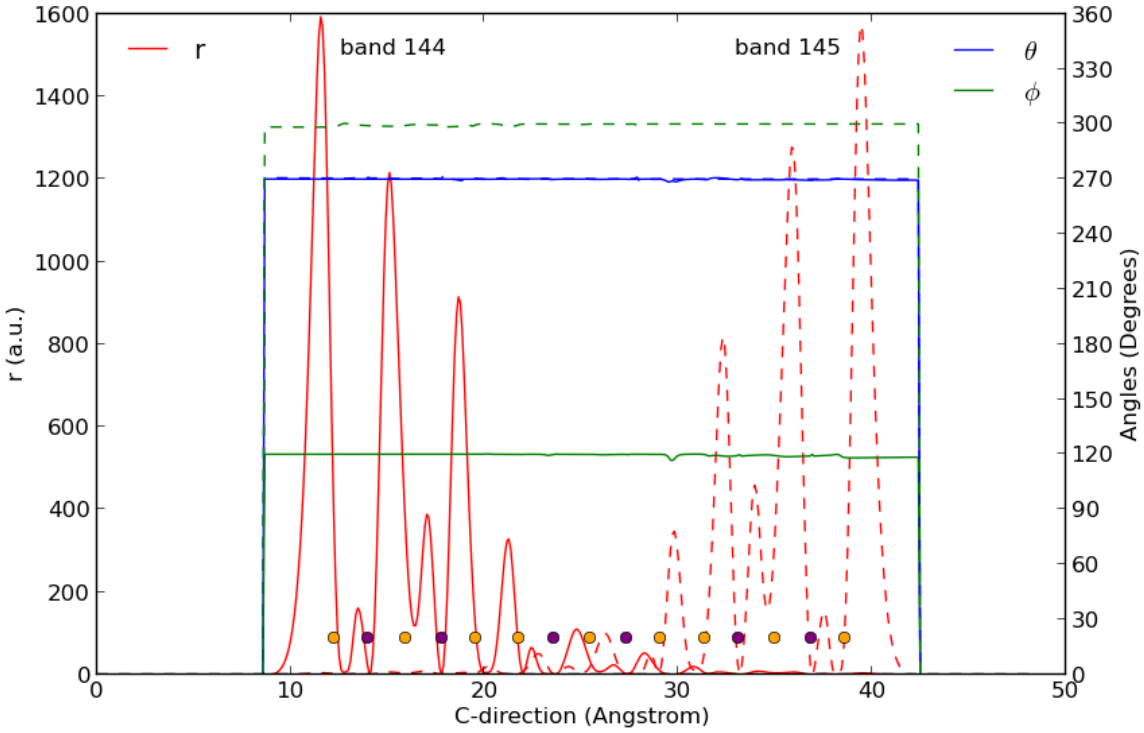


Fig. 3.32: Surface Bloch states in bands 144 and 145 evaluated in k-point [0, 0.04, 0]. The states are projected onto the C-direction. The states are localized on opposite sides of the slab configuration, and their spin directions are opposite.

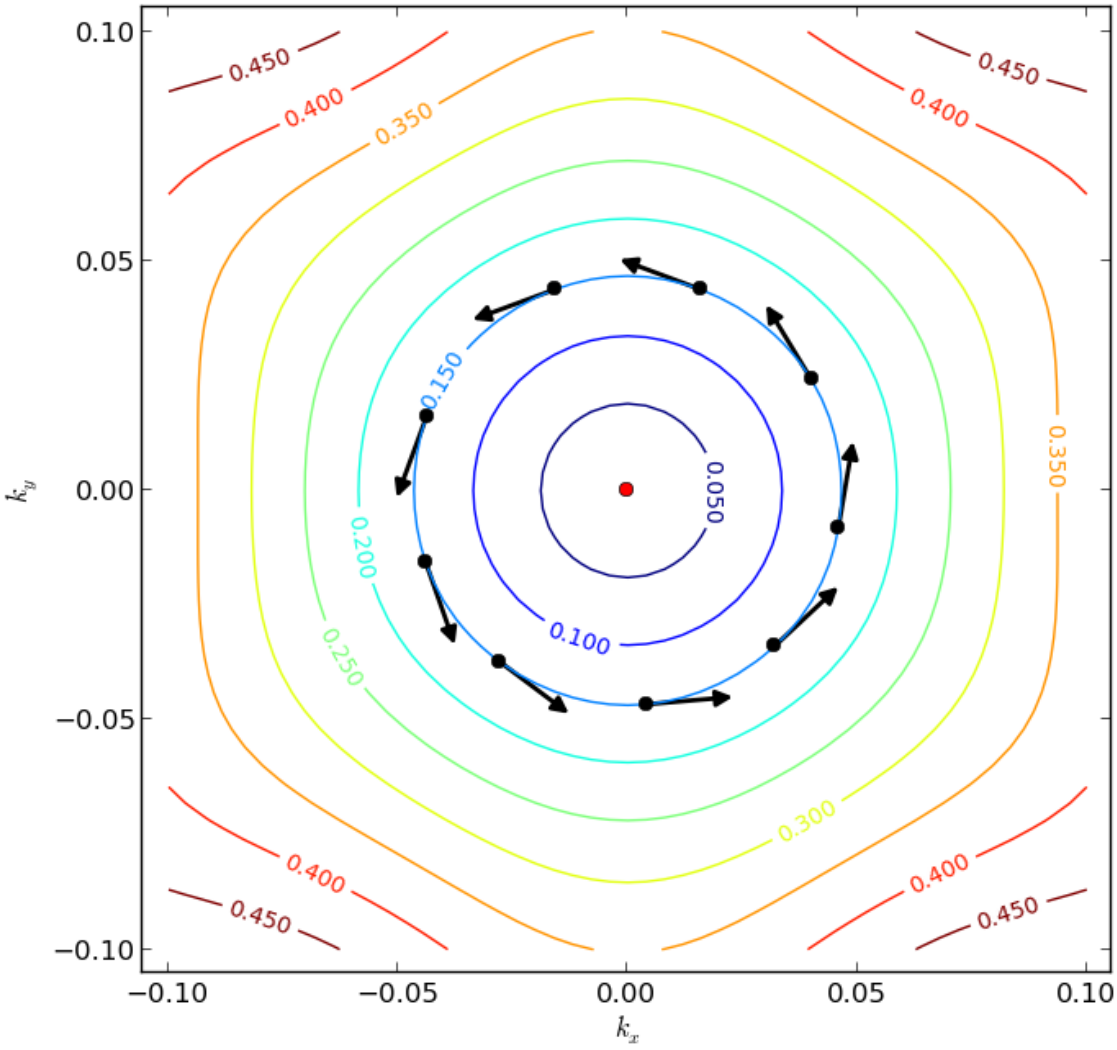


Fig. 3.33: Fermi surfaces on the Dirac cone centered at the Γ-point. The arrows indicate how the in-plane spin direction varies as an electron circles around the Dirac point.

10th (k_x, k_y) point around that contour. In each point, an arrow indicates the in-plane spin orientation (ϕ). It is clear that the spin rotates by 2π as it circles around the Dirac point.

The reason for this is simple: Time-reversal symmetry requires that states at momenta **k** and -**k** have opposite spin. However, the Dirac cone for surface states originating from a *single* surface is *not* spin degenerate. The spin must therefore rotate with **k** as it goes around the Fermi surface. See Ref.[Page 548, 1] for more details.

Topological Invariants

The properties of a 3-D topological insulator is determined by four topological invariants[5],[6],[7], which describe properties of the band structure at special k-points, i.e. the Γ-point and the Brillouin zone edges.

The first topological invariant describes the difference in the wavefunction symmetry at the different special k-points. This invariant distiguishes strong topological insulators, and when present, the system will have topologically protected surface bands. The topological invariant is robust towards atomic disorder and any time-invariant perturbation cannot remove the surface bands.

For a system with inversion symmetry the topological invariants can be calculated from the symmetry of the Bloch function at 8 special Brillouin zone points[8],

$$\Gamma_{i=(n_1,n_2,n_3)} = \frac{1}{2}(n_1\mathbf{b_1} + n_2\mathbf{b_2} + n_3\mathbf{b_3}),$$

where $\mathbf{b_i}$ are the reciprocal lattice vectors and $n_i = 0, 1$.

Let $\psi_{i,n}$ be the nth occupied Bloch function at Γ_i, then we define the symmetry function

$$\delta_i = \Pi_n \sqrt{\langle\psi_{i,n}|\Theta|\psi_{i,n}\rangle}$$

where Θ is the inversion operator. Once the symmetry functions are known the strong topological invariant, ν_0, is given by

$$(-1)^{\nu_0} = \Pi_{i=1}^{8}\delta_i.$$

There are three weak invariants, ν_1, ν_2, and ν_3, given by products of four δ_is, for which Γ_i reside in the same plane:

$$(-1)^{\nu_k} = \Pi_{n_k=1,n_{j\neq k}=0,1}\delta_{i=(n_1,n_2,n_3)}.$$

The weak invariants are not robust towards atomic disorder, and do not guarantee the existence of metallic surface bands.

Python script for the calculating the topological invariants

Download the script `TopologicalInvariant3D.py`, which implements the equations given above for calculating the topological invariants. The script will only work for systems with inversion symmetry. An example of usage is given with the python script below.

[5] Liang Fu, CharlesăL Kane, and EugeneăJ Mele. Topological insulators in three dimensions. *Physical Review Letters*, 98(10):106803, 2007. doi:10.1103/PhysRevLett.98.106803.

[6] JoelăE Moore and Leon Balents. Topological invariants of time-reversal-invariant band structures. *Physical Review B*, 75(12):121306, 2007. doi:10.1103/PhysRevB.75.121306.

[7] Rahul Roy. Topological phases and the quantum spin Hall effect in three dimensions. *Physical Review B*, 79(19):195322, 2009. doi:10.1103/PhysRevB.79.195322.

[8] Liang Fu and CharlesăL Kane. Topological insulators with inversion symmetry. *Physical Review B*, 76(4):045302, 2007. doi:10.1103/PhysRevB.76.045302.

```
from TopologicalInvariant3D import topologicalInvariant3D

bulk_configuration = nlread('Bi2Se3_bulk.nc', BulkConfiguration)[-1]

print '-----------------------------------'
print 'Topological Invariants = ', topologicalInvariant3D(bulk_configuration)
print '-----------------------------------'
```

which when run will produce the output

```
-----------------------------------
Topological Invariants =  [1 0 0 0]
-----------------------------------
```

We see that Bi_2Se_3 is a strong 3D topological insulator, because $\nu_0 = 1$, while the three weak invariants are all 0.

You can now easily screen a range of 3D materials with inversion symmetry for their topological indices to figure out if they are strong or weak topological insulators.

Note: For users of QuantumATK 2017 and later: The script `TopologicalInvariant3D.py` will **not** work with QuantumATK 2017 and later versions. Download instead the script `TopologicalInvariant3DATK2017.py` and use it like this:

```
from TopologicalInvariant3DATK2017 import topologicalInvariant3D

bulk_configuration = nlread('Bi2Se3_bulk.nc', BulkConfiguration)[-1]

print '-----------------------------------'
print 'Topological Invariants = ', topologicalInvariant3D(bulk_configuration)
print '-----------------------------------'
```

References

3.2.8 Effective band structure of random alloy InGaAs

Version: 2015.1

Random alloys are becoming technologically important materials. For example, SiGe is one of the best known thermoelectric materials, and InGaAs is a prime candidate to substitute or complement silicon in future CMOS devices. Simulations of both thermoelectric elements as well as transistor devices often refer to band structure parameters such as band gaps and effective masses.

In this tutorial, you will lean how to setup the structure of a random alloy and calculate the so-called effective bandstructure. The tutorial will focus on $In_{0.53}Ga_{0.47}As$ and calculations will be performed with DFT using the TB09-MGGA exchange-correlation functional, in order to obtain accurate band gaps.

Note: In a normal band structure, each band is a line with zero width/broadening in both the E- and **k**-directions:

Blochs theorem applies and the wavenumber **k** is a good quantum number. In a disordered alloy, Blochs theorem strictly does not apply, and one cannot *a priory* know if **k** is still a good quantum number or if the whole concept of a band structure is well defined.

In QuantumATK 2015 it is possible to calculate effective band structures of random alloys, following the method described in Refs.[1][2][3]. The purpose is to investigate whether or not the random alloy band structure is well defined, and if it is reasonable to extract band parameters.

Methodology

In the calculation of effective band structures, one considers a large super-cell containing many atoms. Here we consider a 3x3x3 repetition of the primitive FCC unit cell with 64 atoms. We model $In_{0.53}Ga_{0.47}As$ as InAs with some of the indium atoms replaced at random with gallium atoms. The In/Ga ratio is kept fixed at 0.53/0.47 in all calculations.

From the band structure of the super-cell containing more than 500 electronic bands, one can unfold the band structure such that it only contains bands corresponding to the primitive cell. In the unfolding procedure, one calculates a spectral weight $| \langle e^{i\mathbf{k}\cdot\mathbf{r}} \mid \psi_{j,K} \rangle |$, where $\mid \psi_{j,K} \rangle$ is an eigenstate of the super-cell at k-point K. If the super-cell is simply a copy of a simple unit cell (e.g. a super-cell of InAs), the spectral weights will either be 0 or 1, and the band unfolding can be done explicitly. However, in the case of a disordered alloy, such as $In_{0.53}Ga_{0.47}As$, the band weights can be any real number between 0 and 1.

Band structures of InAs

You will first calculate the normal band structure of a InAs, using the primitive (1x1x1) unit cell. Then a 3x3x3 InAs configuration is created, and you will show that the effective band structure of this repeated super-cell reproduces that of the primitive cell.

1x1x1 InAs

•Open the **Builder** and go through the following steps:

–Use the *Add* → *From Database* tool to create the InAs bulk configuration.

–Use the *Bulk Tools* → *Lattice Parameters* tool to change the lattice constant to the room temperature value:

∗Choose keep *fractional*;

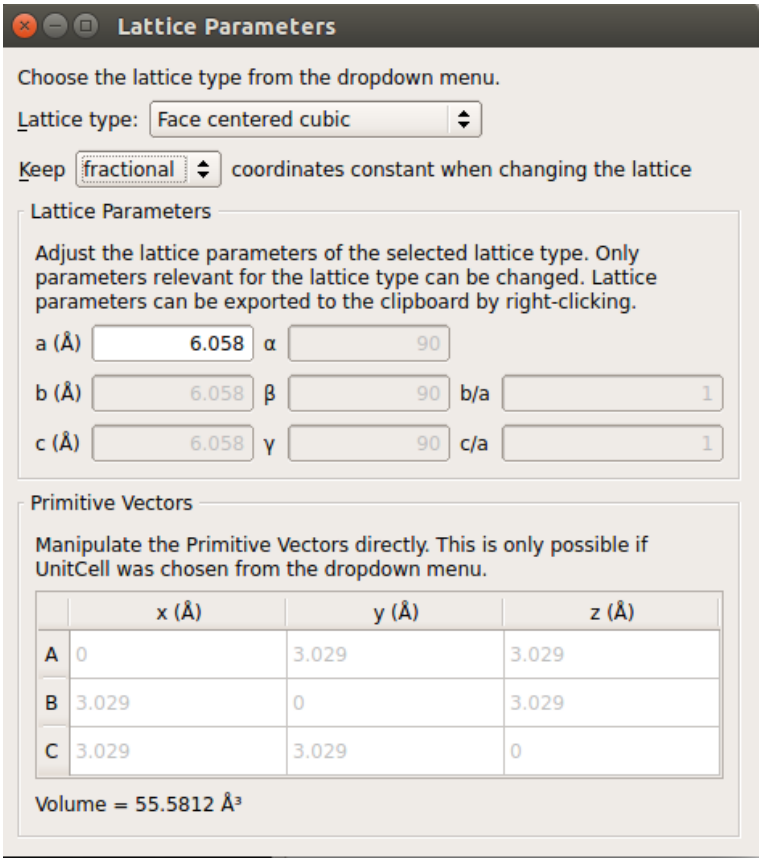
∗Type in a = 6.058 Å.

–Click *OK* and send the configuration to the **Scripter** .

[1] V.ăPopescu and A.ăZunger. Effective Band Structure of Random Alloys. *Phys. Rev. Lett.*, 104:236403, Jun 2010. doi:10.1103/PhysRevLett.104.236403.

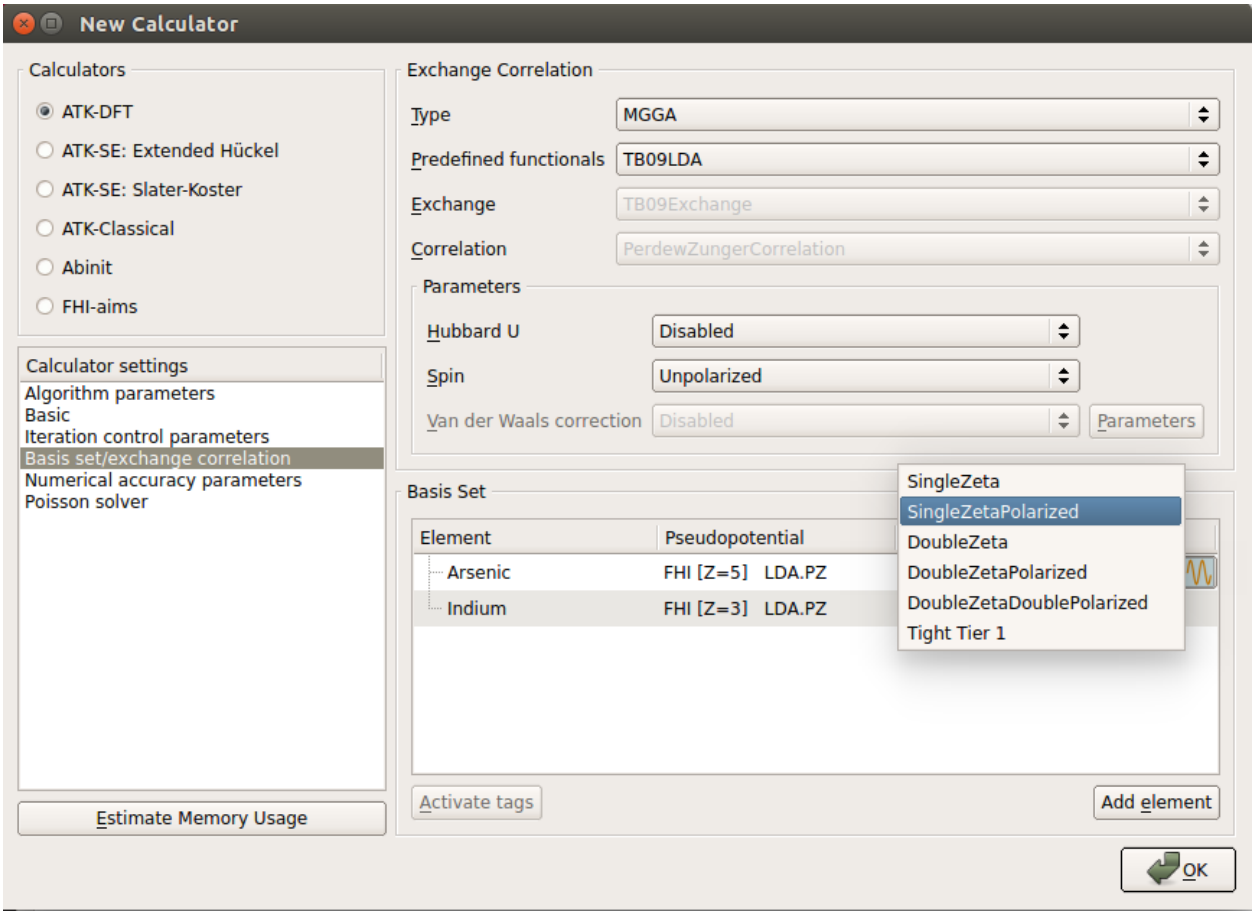
[2] V.ăPopescu and A.ăZunger. Extracting E versus P k effective band structure from supercell calculations on alloys and impurities. *Phys. Rev. B*, 85:085201, Feb 2012. doi:10.1103/PhysRevB.85.085201.

[3] M.ăW. Haverkort, I.ăS. Elfimov, and G.ăA. Sawatzky. Electronic structure and self energies of randomly substituted solids using density functional theory and model calculations. *arXiv*, pages 1109.4036, 2011. URL: http://arxiv.org/abs/1109.4036.



- In the **Scripter** add a New Calculator block. Then open and edit it:

  - Select the *ATK-DFT* calculator.
  - Use *MGGA* for exchange-correlation potential.
  - Set up a 9x9x9 grid for k-point sampling.
  - Use a *SingleZetaPolarized* basis set to speed up calculations.



•Next, add an *Analysis → Bandstructure* block. Open it and set the Brillouin zone route to L, G, X.

•Change the Default output file name to `InAs_1x1x1.hdf5`.

•Transfer the script to the **Editor** in order to specify the TB09-MGGA c-parameter. Locate the line

```
exchange_correlation = MGGA.TB09LDA
```

and change it to

```
exchange_correlation = MGGA.TB09LDA(c=0.91)
```

•Run the calculation using the **Job Manager** or in a Terminal. It takes about eight minutes to complete the simulation.

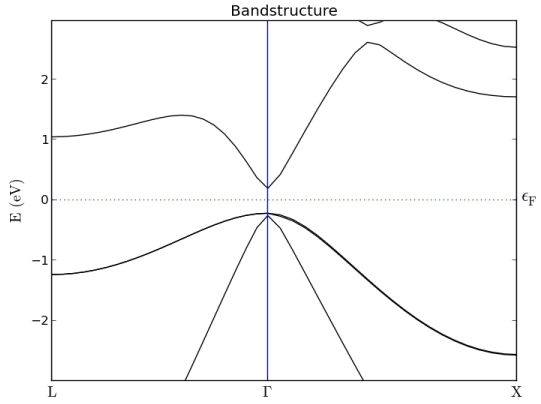
Note: You can learn more about the TB09-MGGA exchange-correlation potential and the TB09-MGGA c-parameter from the tutorial *Meta-GGA and 2D confined InAs*.

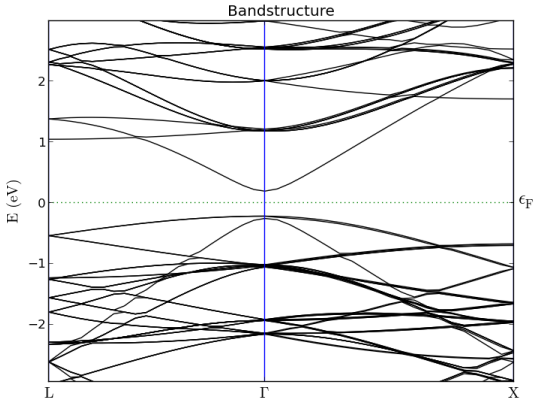
3x3x3 InAs

- Back in the **Builder** , repeat the 1x1x1 InAs configuration 3 times in each direction using the *Bulk Tools* → *Repeat* tool.
- Send this configuration to the **Scripter** and repeat the procedure outlined above to set up a script, buth with the following changes:
  - The k-point sampling in the calculator should be 3x3x3.
  - The default output file name should be `InAs_3x3x3.hdf5`.
- Use again the **Editor** to explicitly set the TB09-MGGA c-parameter, and run the calculation.

Results

The band structures for the 1x1x1 (left) and 3x3x3 (right) InAs configurations are shown below. The 3x3x3 band structure is folded at the zone boundaries, which makes it hard to locate the valleys corresponding to the X- and L-valley minima of the 1x1x1 configuration.

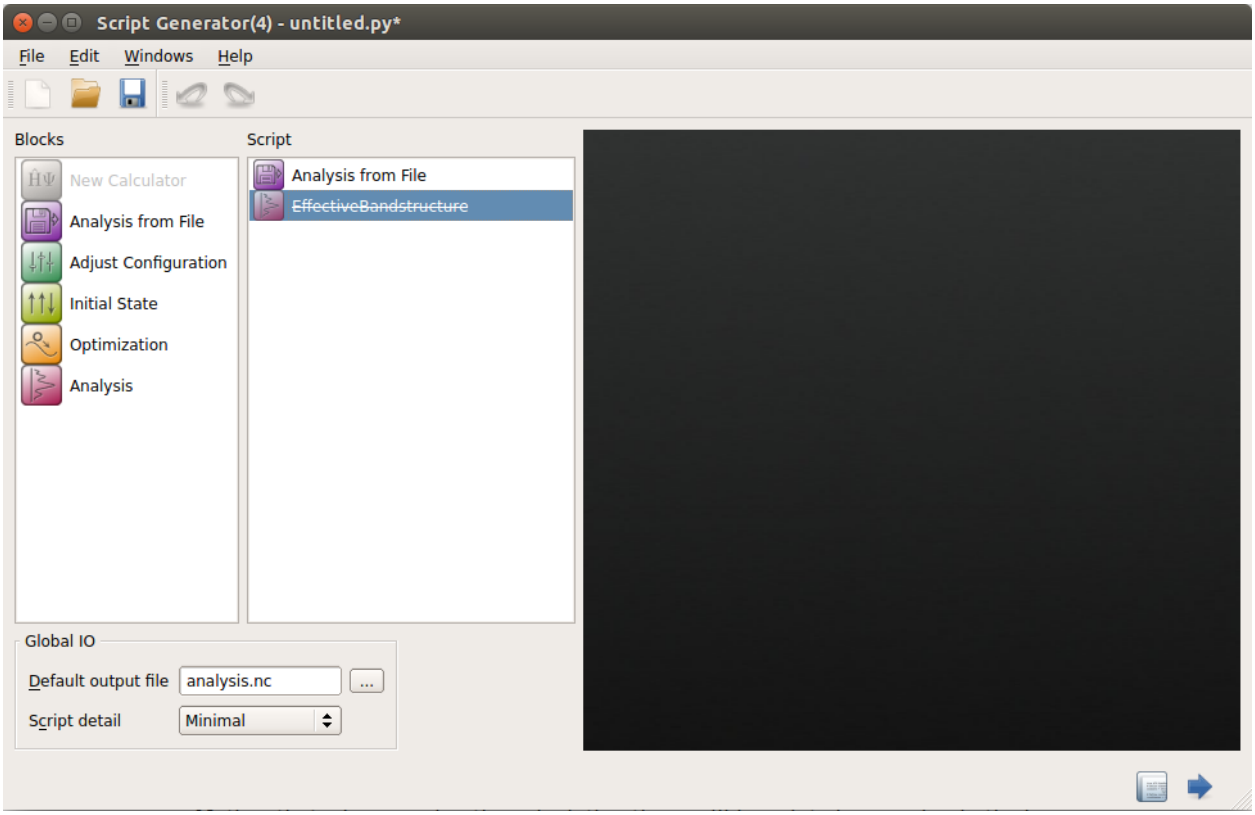




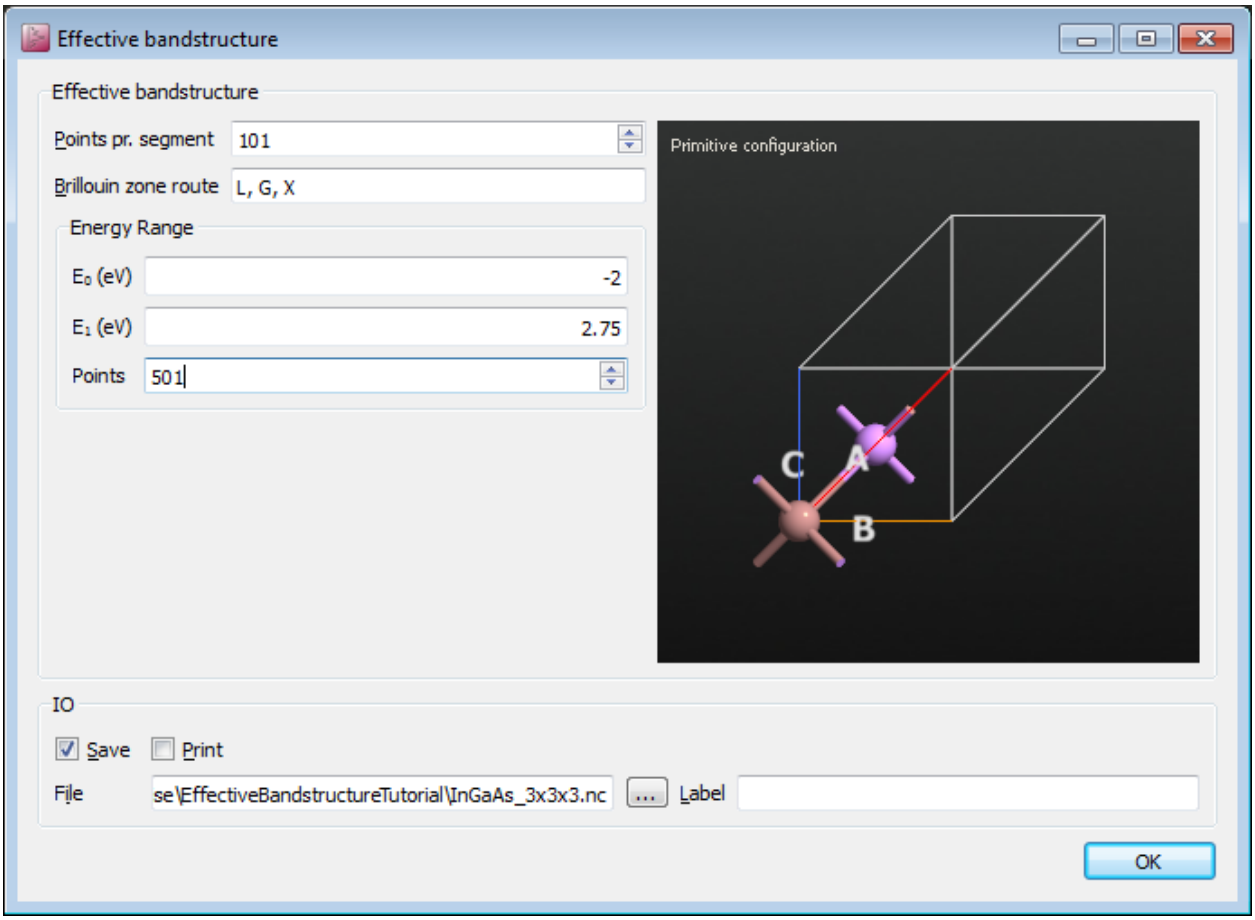
Effective band structure of 3x3x3 InAs

Open a new **Scripter** window, and follow these steps:

- Add the Analysis from File block, open it, and select the existing HDF5 file `InAs_3x3x3.hdf5`.
- Add also the *Analysis* → *EffectiveBandstructure* block. It will be crossed over, indicating that the block needs further editing.



- Open the EffectiveBandstructure block and start editing it.
- Select a 1x1x1 InAs configuration, either from the **Builder** or from the **LabFloor**, and drag it to the black area named Primitive configuration.
- Apply the following settings:
  - Points pr. segment: 101
  - Brillouin zone route: L, G, X
  - E0: -2.0 eV
  - E1: 2.75 eV
  - Points: 501



•The script is now done, so save it and run the calculation. It takes about three minutes to complete on a laptop.

Note: When running the calculation the folllowing warning will appear in the log file:

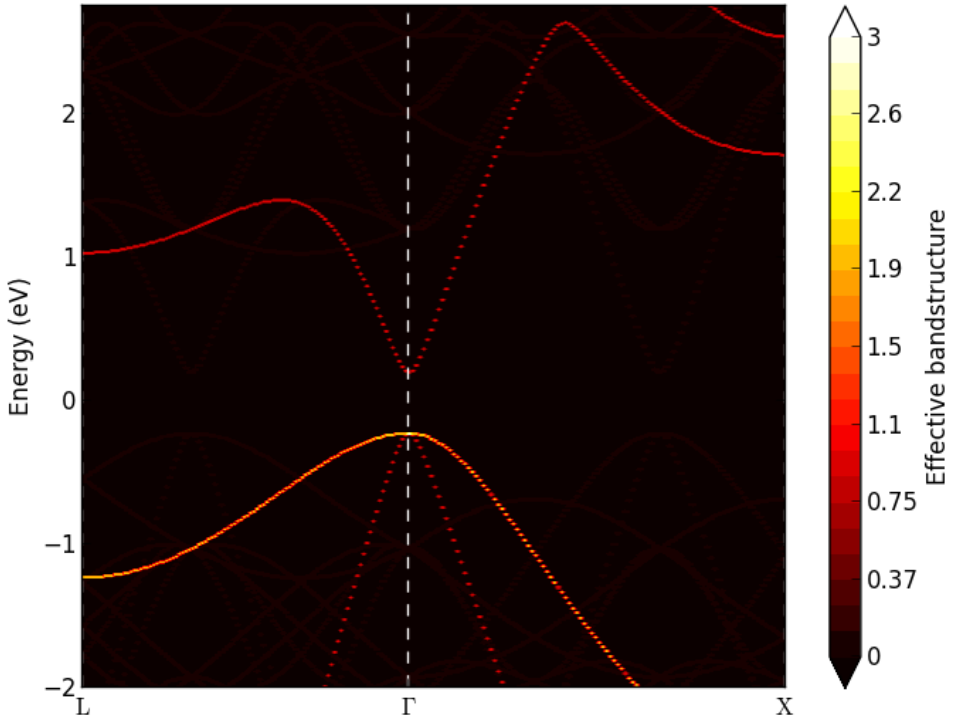
```
Warning: The calculation did not converge to the requested tolerance!
```

This is perfectly OK. A single SCF iteration is executed to obtain the basis functions in the primitive cell, which are used in the effective band structure calculation.

•The EffectiveBandstructure object will now have appeared on the **LabFloor**. It looks like this:

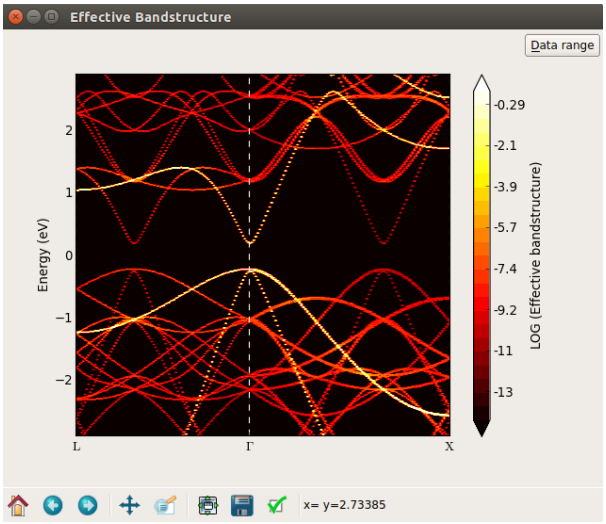
$$\epsilon(E, \mathbf{k})$$

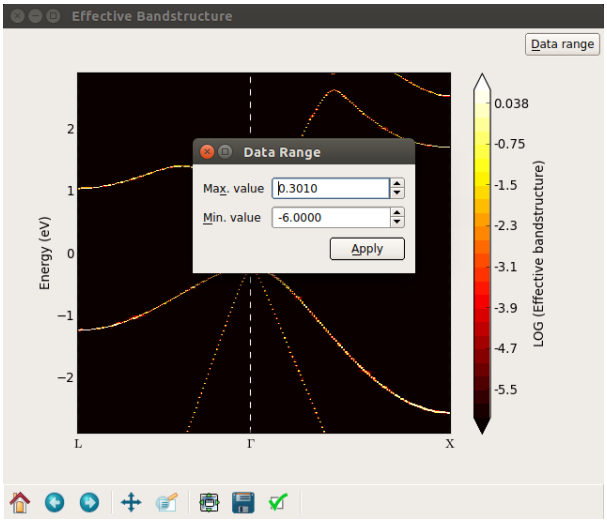
•Select the object and use the **Effective Bandstructure Analyzer** plugin to visualize it. Compare the figure (shown below) with the normal band structure plot of the 1x1x1 InAs shown above.



The analyzer shows a band density map of the **k**,E(**k**) plane. Each band will add a weight between 0 and 1 to the band density. At the Γ-point there are three degenerate valence bands, and the band density becomes 3 (bright yellow in the figure above). The conduction band has a weight of 1.

In order to see all the bands with a weight close to 0, you can plot the band density on a logarithmic scale. Right-mouse click on the plot and select *Log scale* in the bottom of the menu, and the plot is shown as the left figure below. A lot of bands appear in red color. By inspecting the colorbar you notice that these bands have weights around 10^{-9}, which is essentially just numerical noise. Try to adjust the Data range (upper right corner) and set the minimum value to -6. The noise bands now disappear, as shown at the right figure below.





Important: So far, you have seen that the effective bandstructure works, and give what we expect: The effective band structure of the 3x3x3 repeated InAs is the same as the normal bandstructure of the simple 1x1x1 InAs.

In the next section you will learn how to setup a random InGaAs alloy – a system where the normal bandstructure cannot be calculated for a simple 1x1x1 configuration, and where it is not *a priori* obvious if the whole concept of a band structure is well-defined or not.

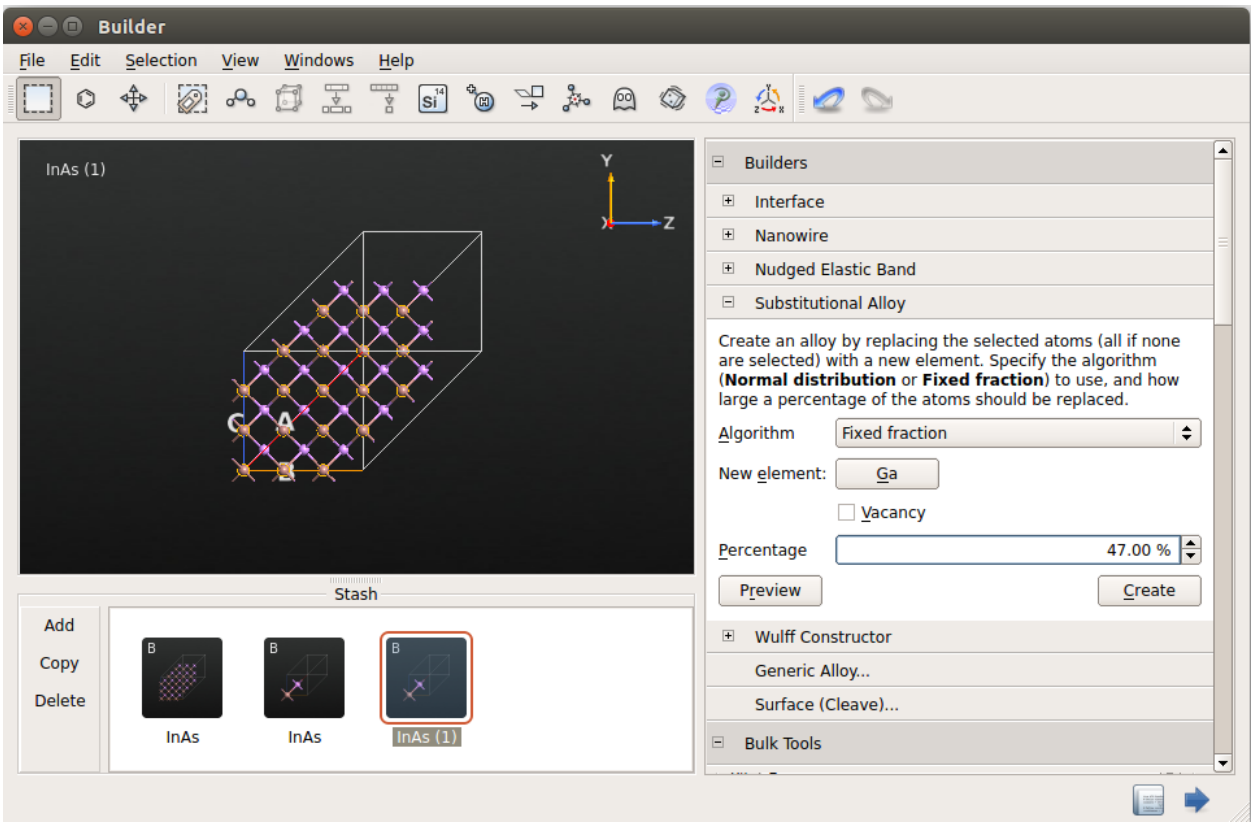
$In_{0.53}Ga_{0.47}As$ random alloy

You will now set up a $In_{0.53}Ga_{0.47}As$ random alloy structure and calculate the effective bandstructure. We shall also consider the sample averaged effective band structure, and analyze the consequences of finite band broadening.

Random alloy

Open the **Builder** to create the alloy configuration:

- Add a new InAs configuration, and change the lattice constant to 5.8687 Å, which is the experimental value for $In_{0.53}Ga_{0.47}As$.
- Right-click the new Stash item and copy it (you will need the copy later on for calculating the effective bandstructure).
- Repeat the InAs configuration 3 times in each direction.
- Next, select all the indium atoms by using the *Select* → *By Element* tool, and open the *Builders* → *Substitutional Alloy* plugin. Apply the following settings:
  - Algorithm: *Fixed fraction*
  - New element: *Ga*
  - Percentage: *47%*
- Then click *Create*.



In the new configuration, approximately 47% of the indium atoms have been substituted with gallium atoms in a random way. Transfer this $In_{0.53}Ga_{0.47}As$ random alloy configuration to the **Script Generator** .

Effective band structure calculation

- Add a **New Calculator** block with the same settings as used for InAs:

  – MGGA exchange correlation.

  – 3x3x3 k-point sampling.

  – SingleZetaPolarized basis set.

- Add the *Analysis* → *EffectiveBandstructure* block, and open it for editing.

- Drag-and-drop the copied 1x1x1 InAs configuration with the 5.8687 Å lattice constant from the **Builder** onto the black area.

- Apply these settings:

  – Points pr. segment: 101

  – Brillouin zone route: L, G, X

  – E0: -2 eV

  – E1: 2.75 eV

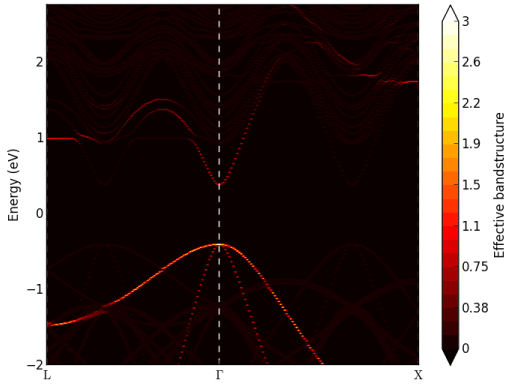
  – Points: 501

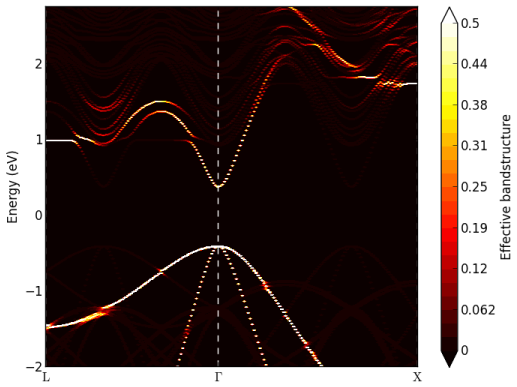
•Transfer the script to the **Editor** and add the TB09-MGGA c-parameter. For $In_{0.53}Ga_{0.47}As$ with a SingleZetaPolarized basis set, you should use c=1.01 to get approximately the experimental band gap of 0.74 eV.

•Run the job using **Job Manager** or in a Terminal. It takes about 10 minutes to complete the job.

Results

Investigate the calculated effective band structure using the **Effective Bandstructure Analyzer**. It should be similar to the figure on the left below. The bands are no longer weighted at either 0 or 1, but with a value in between. In particular around the L- and X-point conduction band minima, there are clear deviations from the simple band structure seen for InAs. Try to adjust the data range and set the maximum value to 0.5. Then some of the bands become more visible, as shown below on the right. Near the Γ-point, both the valence band and conduction bands are well-defined, and the effect of the random disorder seems to be rather limited.





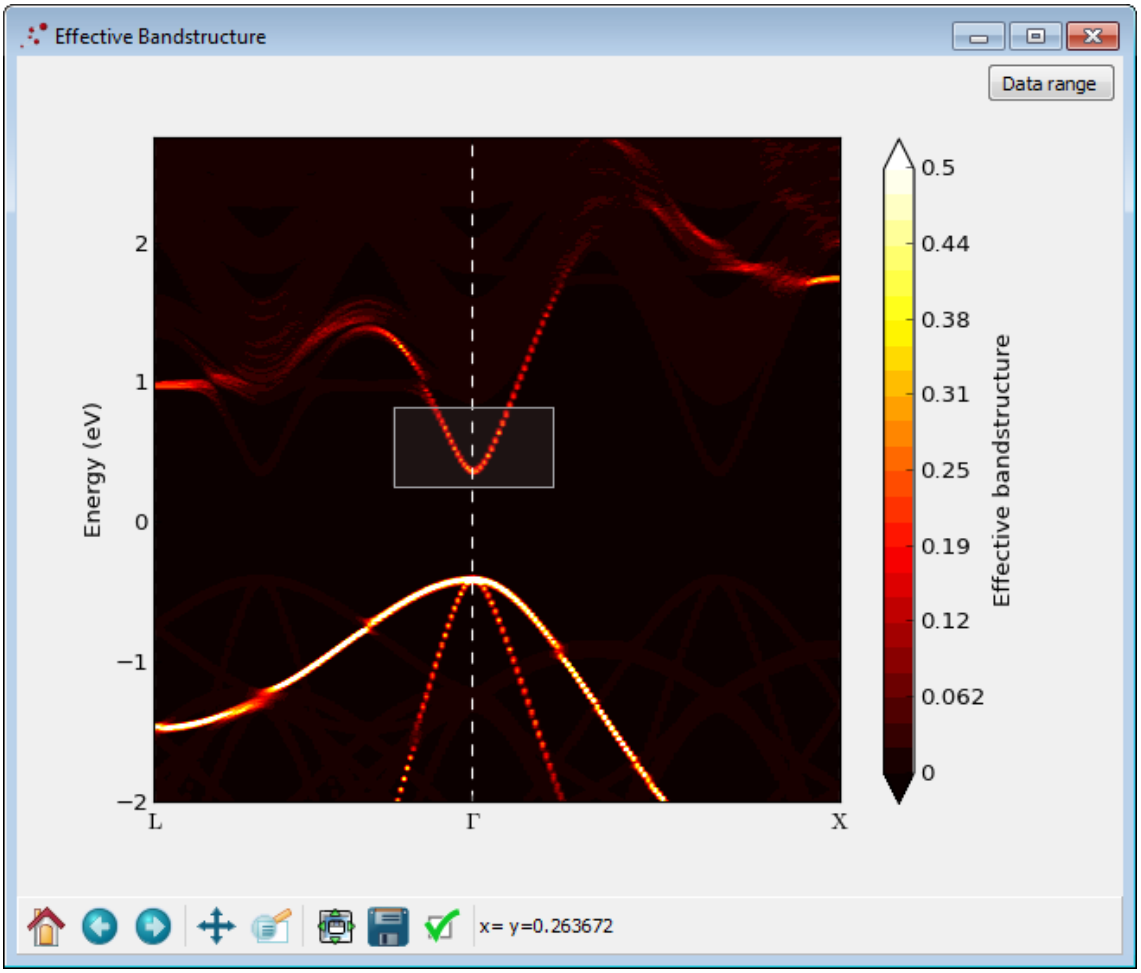
Note: The resulting effective bandstructure is not necessarily exactly the same as in the figure above due to the random alloy generation.

Sample averaged effective band structure

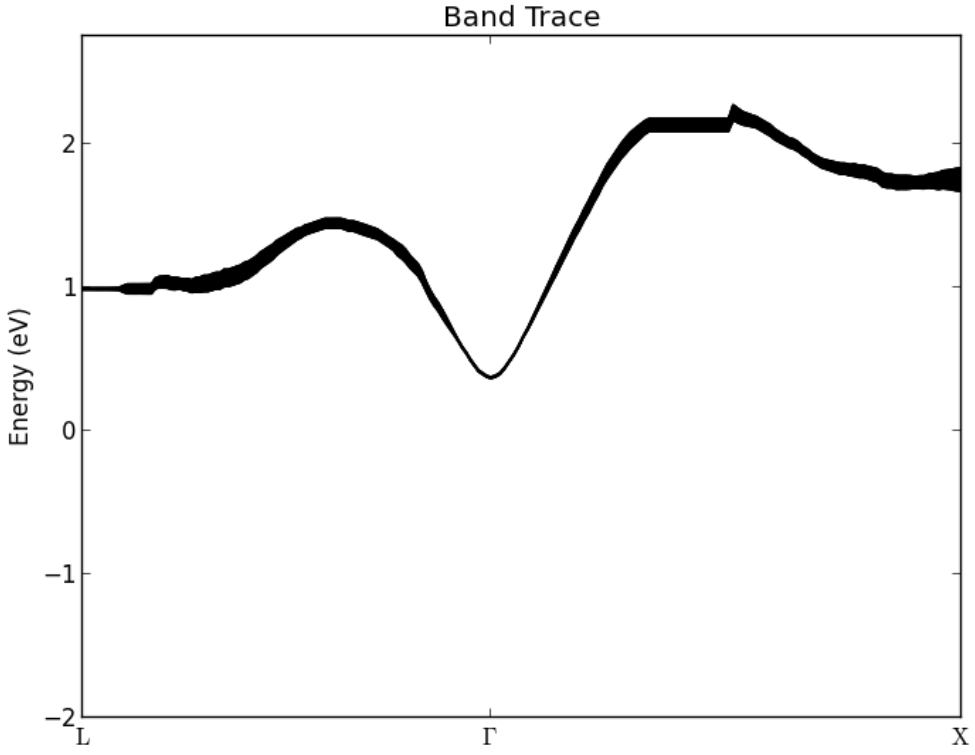
A quantitatively more reliable effective band structure should be obtained by repeating the above procedure a number of times to create different random configurations. Some python scripting can help to finish the task elegantly and efficiently.

Download the Python script `ingaas_ebs.py`, which creates a number of random $In_{0.53}Ga_{0.47}As$ alloys and performs the effective band structure calculation for each of them. You can then average over the results. The default number of generated alloys (`number_of_runs`) is 10. You will get even better results by increasing that number.

Run the script, and observe the EffectiveBandstructure objects appear in the file `InGaAs_3x3x3.hdf5` on the **LabFloor**. Mark all the generated data objects and click the **Effective Bandstructure Analyzer**. The analyzer will now show the average of all the effective band structure densities. Try to adjust the colours by changing the data range and maximum value:



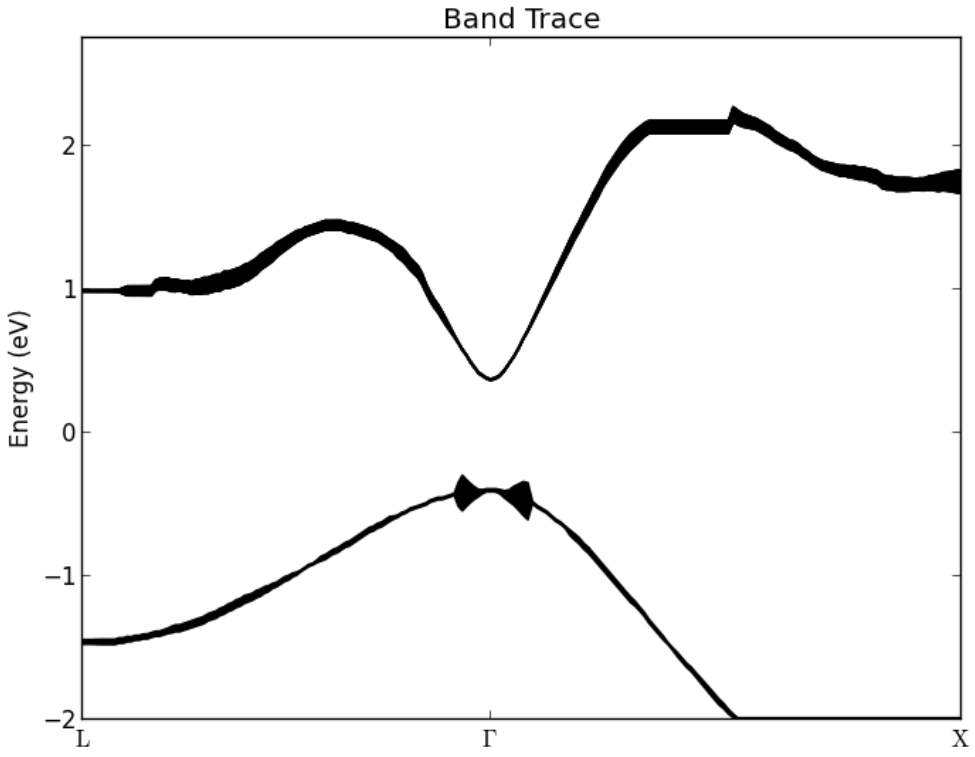
It is possible to trace the most significant bands with the heighest weight: Draw a box with the mouse like shown in the figure above. A progress bar will appear in the bottom of the window signalling that the effective band structure data is being analyzed for band tracing. After a little while a new window will appear, like the one shown below. The figure shows the traced bands plotted with a finite with. The center of the line is identified as a band. It can be exported to a text file by right-clicking on the figure and choosing *Export data*.



Tip: The band is calculated in the following way:

1. For each **k**, the band density is first smoothed by performing a convolution with a Lorentzian function (the broadening in energy is 0.2 eV).
2. The peaks of this smoothed band density is found.
3. A band is formed by selecting the peak energies closest to the peak energy found at the previous k-point (or, for the first k-point, closest to the center of the box you draw).
4. Proceed to the next k-point and repeat the process.

The width is calculated as the second moment of the raw data (i.e. not smoothed) in an interval of -0.15–0.15 eV around the peak energy. The maximum width is thus 0.3 eV. Note that if two or more bands are close to each other, the calculated width will be slightly wrong, since the density from the other bands will contribute to the width. Try to draw a box around the highest valence band. The Band Trace figure will be quickly updated to appear like the figure below. Close to the Γ-point the senond-highest valence band erroneously contribute to the width of the first one.



Finite broadening

Contrary to a normal band structure, the effective bands will have a finite broadening. If the broadening is small, the band structure is well-defined; if the broadening is large, the concept of bands is no longer valid. The average effective band structure of an $In_{0.53}Ga_{0.47}As$ random alloy, as shown in the figure above, is well-defined close to the Γ-point. However, the conduction band is not well-defined around the L- and X-valley minima, where the bands have a finite broadening of $w \approx 0.06$ eV.

The well-defined effective band structure around the Γ-point can be understood in terms of the long wavelength present there. The local random variations are not seen by a long wavelength electron. On the contrary, at the X and L points, the wave functions vary on the atomic scale, and are affected by the disorder.

Analysis and consequences of finite band broadening

One can interpret the finite band broadening in terms of a life-time $\tau = \hbar/w = 11$ fs. Together with an approximate longitudinal effective mass m_l=1.5 m_e and transverse effective mass m_t=0.2 m_e, both obtained from virtual crystal approximation (VCA) calculations, we can estimate the effective mean free path (MFP) as:

$$\mathrm{MFP} = v_k \tau,$$

where the band velocity $v_k = 1/\hbar \cdot \partial E(k)/\partial k$. Assuming a parabolic band $E(k) = \hbar^2 k^2/2m$ around the valley minima, the mean free path as a function of energy, relative to the X- or L-valley minimum, is then:

$$\mathrm{MFP} = \frac{\hbar}{\omega}\sqrt{\frac{2E}{m}}.$$

At the valley minima, the mean free path vanishes, and at a constant energy surface of 0.01 eV above the X- or L-valley minima we obtain estimated mean free paths of 1.5 nm and 0.5 nm in the transverse and longitudinal directions, respectively. Clearly, the random disorder will be a limiting factor for transport in these valleys. The MFP vs. energy can be plotted with the python script `mean_free_path.py`.

Final comments

For scientific use we recommend to use even larger super-cells, e.g. 4x4x4 or even larger, which will be more like a real random alloy. The relatively small 3x3x3 cell used in this tutorial was chosen to make the calculations relatively fast. The small cell size might affect the results in a quantitive manner, but we do expect the main conclusions to remain unchanged.

Since the effective band structure close to the Γ-point has a narrow broadening, it is reasonable to assume that Blochs theorem applies here and that the random disorder has a small effect on the transport properties. In this case, more efficient studies of the band structure of random alloys can be performed with the virtual crystal approximation as explained in the tutorial *Virtual Crystal Approximation for InGaAs random alloy simulations*.

References

3.2.9 Complex bandstructure of Si(100)

Version: 2015.1

Downloads & Links

`si_100_cbs.py` `3D_plot.py` `2D_plot.py` *`ComplexBandstructure`*

In this tutorial you will calculate the complex bandstructure of a silicon crystal along the (100) direction.

In particular you will:

1. create the Si(100) surface;
2. set up and run the calculation;
3. plot the complex bandstructure (3D and 2D).

Background

In a periodic solid the eigenstates $\psi_{n\mathbf{k}}$ of the Schrödinger equation $H\psi_{n\mathbf{k}} = E_{n\mathbf{k}}S\psi_{n\mathbf{k}}$, ($S$ is the overlap matrix) can be written as $\psi_{n\mathbf{k}}(\mathbf{r}) = e^{-i\mathbf{k}\cdot\mathbf{r}}U_{n\mathbf{k}}(\mathbf{r})$, where $U_{n\mathbf{k}}(\mathbf{r})$ is a periodic function with the same periodicity as the crystal itself. In a usual bandstructure calculation, the wave vector $\mathbf{k}$ is a real number, and by solving the Schrödinger equation above for various fixed values of $\mathbf{k}$ (typically located along different symmetry lines in the first Brillouin zone) an eigenproblem is defined, from which the eigenenergies $E_{n\mathbf{k}}$ (i.e. the bandstructure) can be determined.

When calculating the complex bandstructure another approach is taken[1]. Instead the energy E is fixed, and the values of $\mathbf{k}$ which solve the Schrödinger equation are sought. Such solutions will be found with both real and complex $\mathbf{k}$; the

[1] Yia-Chung Chang and J.ăN. Schulman. Complex band structures of crystalline solids: An eigenvalue method. *Phys. Rev. B*, 25:3975–3986, Mar 1982. doi:10.1103/PhysRevB.25.3975.

solutions with a real $\mathbf{k}$ are the usual Bloch states, while the solutions with an imaginary part are exponentially decaying in one direction and increasing in the other. Such solutions cannot exist in a bulk material, and so they are normally ignored in a bandstructure calculation. They may however exist at a surface or interface, and they give information about how electronic states decay in the material, for instance through a thin tunneling barrier.

Note: You will primarily use the graphical user interface QuantumATK for setting up and analyzing the results. If you are new to QuantumATK, it is recommended to go through the Basic QuantumATK Tutorial.

The calculations in this tutorial will be performed using the semi-empirical models of QuantumATK. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK Reference Manual. In particular, the Reference Manual entry for complex bandstructure calculations is of relevance: `ComplexBandstructure`.

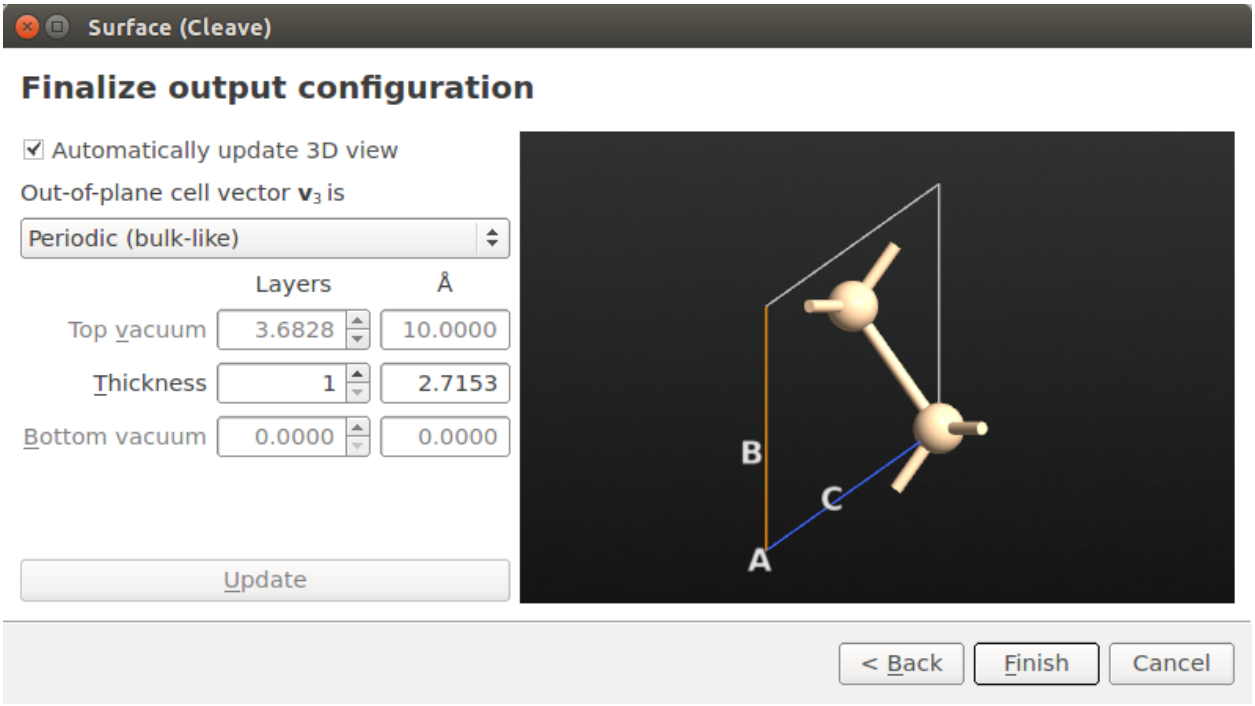
Si(100) surface

Start QuantumATK and create a new project named ComplexBandstructure. Then click *Open* and launch the **Builder** via the icon on the toolbar.

In the Builder, click *Add* → *From Database* and locate the Silicon (alpha) structure. Double-click the line to add the structure to the Stash, or click the icon in the lower right-hand corner.

Unfold *Builders* from the plugin panel and open the Surface (Cleave) tool, and follow the next steps to set up the Si(100) surface:

1. Keep the default Miller indices (100) and click *Next*.
2. Keep the default lattice vectors and click *Next*.
3. Select *Periodic (bulk-like)* for the out-of-plane cell vector.
4. Set the thickness of the surface to 1 layer.
5. Click *Finish* to add the Si(100) surface structure to the Stash.



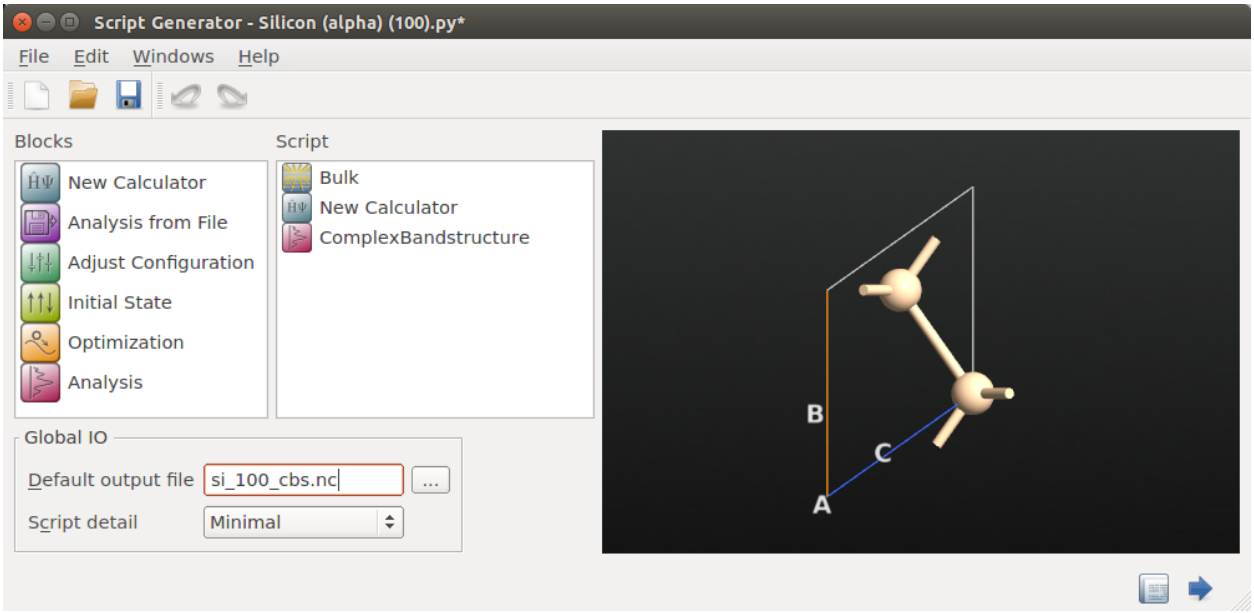
With these settings we use a surface representation with a minimum number of layers in order to reduce the calculation time and avoid zone folding.

Note: The cleave plane – in the present case (100) – is always spanned by the two first unit cell vectors, and $\mathbf{B}$. In the electrode mode, the normal to this plane coincides with $\mathbf{C}$, the third unit cell vector, but in the bulk-like mode this is not the case. In QuantumATK the complex bandstructure is always computed along the third reciprocal vector, $\mathbf{g}_C$, which is of course parallel to $\mathbf{A} \times \mathbf{B}$. With the present geometry you will therefore obtain the complex bandstructure along (100).

Complex bandstructure calculation

Send the structure to the **Script Generator** using the icon in the lower right-hand corner of the Builder and follow the next steps:

1. Add a New Calculator block.
2. Add the *Analysis* → *ComplexBandStructure* block.
3. Change the default output filename to `si_100_cbs.hdf5`.

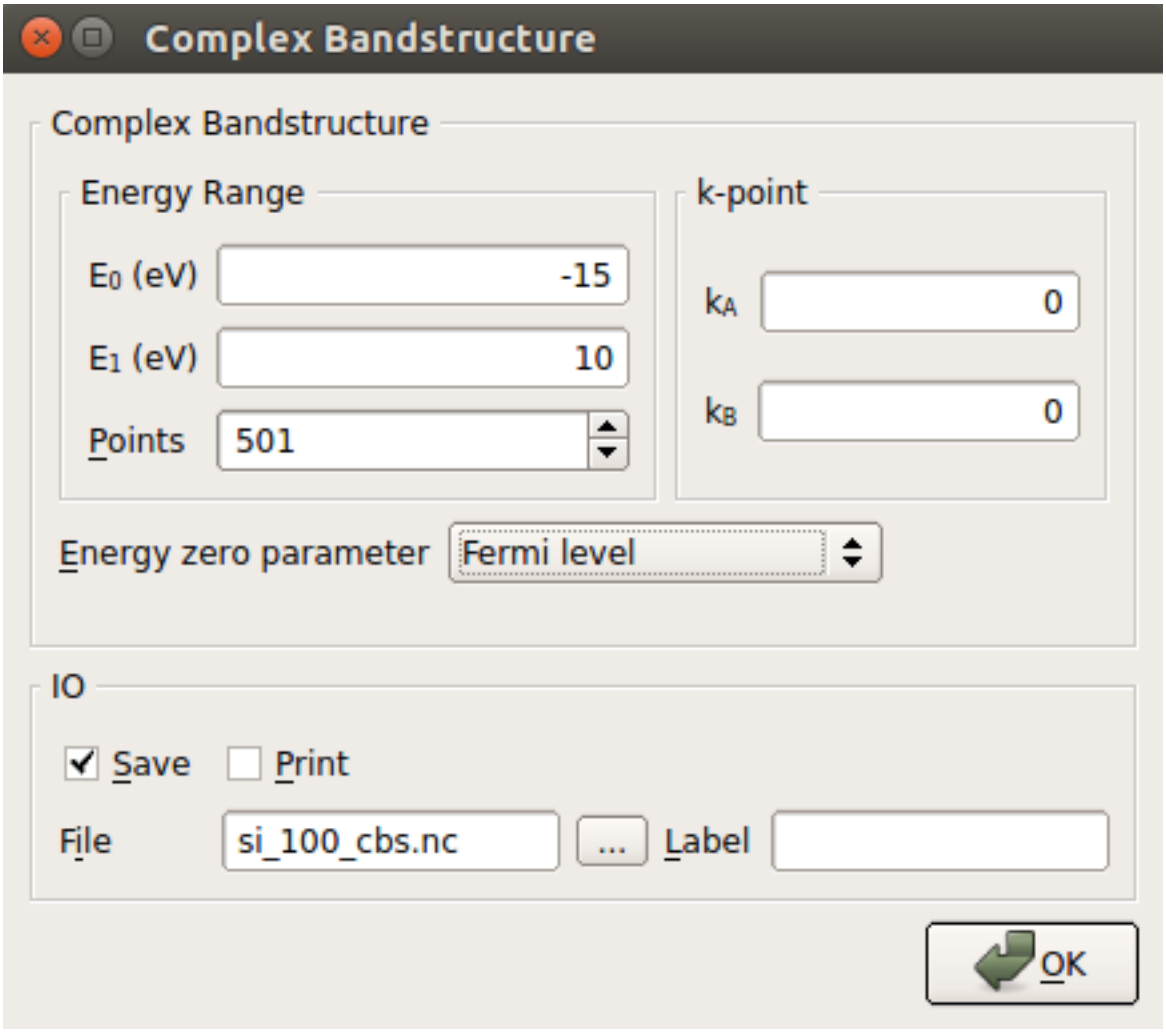


Next, open the inserted New Calculator block (double-click it), and set up a tight-binding calculation:

1. Select the *ATK-SE: Extended Hückel* calculator.
2. Set the k-point sampling to 5x5x5.
3. Under Huckel Basis set, select the *Cerda.Silicon [diamond GW]* basis set. This basis set has been fitted to GW calculations, and gives an excellent description of the bandstructure, including the size of the band gap.
4. Close the dialogue by clicking *OK*.

Next, open the ComplexBandstructure analysis block, and edit it:

- Set the energy range to -15 to 10 eV with 501 points.



Note: The projection of **k** on the cleave plane is kept constant in the complex andstructure calculation, and the value of the projection is specified by the (k_A, k_B) parameters, which are given in units of the two first reciprocal lattice vectors, $\mathbf{g}_A$ and $\mathbf{g}_B$. Therefore, the obtained solutions will lie along the third reciprocal lattice vector, $\mathbf{g}_C$, which is parallel to the cleave plane normal. A new set of solutions $k_C + i\kappa_C$ is obtained for each value of (k_A, k_B).

Please note that k_C is therefore the *real* part of the complex bandstructure, while $kappa_C$ is the *complex* part.

You have now finished the Python script. Save it as `si_100_cbs.py` and send it to the **Job Manager** for execution.

It should only take a few minutes for this small system. If needed, you can also download the script here: `si_100_cbs.py`.

Tip: This type of calculation parallelizes extremely well, so for larger structures it is warmly recommended to execute the script in parallel.

Analysing the results

The file `si_100_cbs.hdf5` should now have appeared on the QuantumATK **LabFloor**. It contains the saved Hückel calculation and the ComplexBandstructure analysis object:

ComplexBandstructure

$\mathbf{k}_\perp(\epsilon, \mathbf{k}_\parallel)$

Select that analysis object and click **Show 2D Plot** tool from the right-hand side plugins panel. A window with a plot of the complex bandstructure pops up. Zoom in a bit to filter out the solutions with large values of the complex part κ_C, since these are not really relevant:

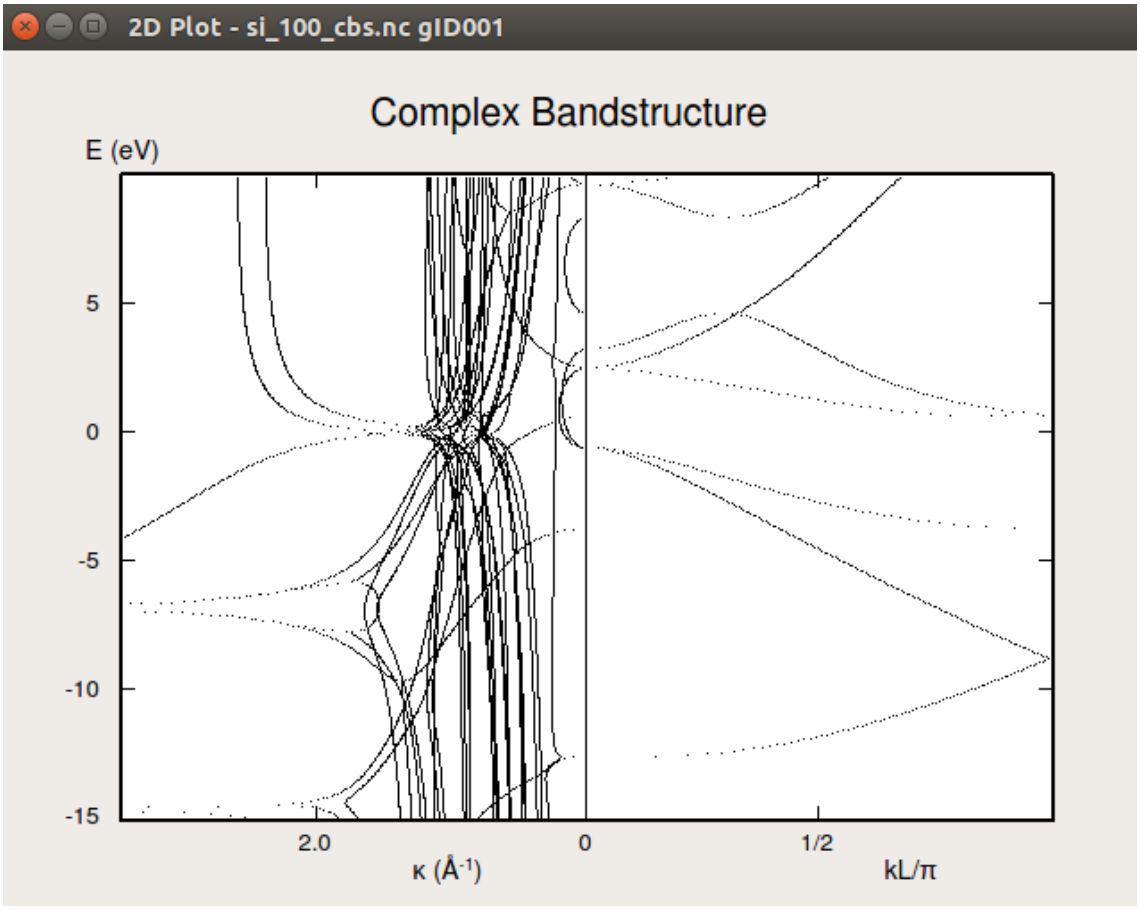


Fig. 3.34: Si(100) bandstructure. The right-hand part of the plot shows the *real* bands – note the indirect band gap of ~1.2 eV. The left-hand part of the plot shows the *complex* bands, plotted against the imaginary part, κ_C. The bands with a small imaginary part are the most important ones, since they are less damped if you should try to tunnel electrons through a thin slice of silicon in the (100) direction.

Note: In the plot above, the solutions κ_C are given in reciprocal Cartesian coordinates, to make it easier to compare different structures. For the real part of the bandstructure, on the other hand, the solutions k_C are normalized by L, the layer separation perpendicular to the cleave plane. In the case of a fcc crystal cleaved along (100), the layer separation is $L = a/2$, where a is the lattice constant. The layer separation can be extracted from the ComplexBandstructure object using the *layerSeparation()* method, see `ComplexBandstructure`.

3D and 2D visualizations

In the complex part of the bandstructure, the solutions actually have a real part too. Thus, they could be visualized in a three-dimensional plot, $(x, y, z) = (k_C, \kappa_C, E)$. This is possible with the following script. Open the QuantumATK **Editor** and copy-paste the script and save it as `3D_plot.py`. Make sure that the indentation is correct. Alternatively, you can simply download it here: `3D_plot.py`.

```
from QuantumATK import *
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import math

# Read the complex bandstructure object from the NC file
cbs = nlread('si_100_cbs.nc', ComplexBandstructure)[-1]
energies = cbs.energies().inUnitsOf(eV)
k_real, k_complex = cbs.evaluate()
L = cbs.layerSeparation()

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# First plot the real bands
kvr = numpy.array([])
e = numpy.array([])

# Loop over energies, and pick those where we have solutions
for (j, energy) in enumerate(energies):
    k = k_real[j]*L/math.pi
    if len(k)>0:
        e = numpy.append(e,[energy,]*len(k))
        kvr = numpy.append(kvr,k)

# Plot the bands with red
ax.scatter([0.0,]*len(kvr), kvr, e, c='r', marker='o', linewidths=0, s=10)

# Next plot the complex bands
kvr = []
kvi = []
e = []

# Again loop over energies and pick solutions
for (j, energy) in enumerate(energies):
    if len(k_complex[j])>0:
        for x in numpy.array(k_complex[j]*L/math.pi):
            kr = numpy.abs(x.real)
            ki = -numpy.abs(x.imag)
            # Discard rapidly decaying modes which clutter the plot
            if ki>-0.3:
                e += [energy]
                kvr += [kr]
                kvi += [ki]

```

(continues on next page)

(continued from previous page)

```
# Plot the complex bands with blue
ax.scatter(kvi, kvr, e, c='b', marker='o', linewidths=0, s=10)

# Put on labels
ax.set_xlabel('$\kappa$ (1/Ang)')
ax.set_ylabel('$kL/\pi$')
ax.set_zlabel('Energy / eV')

plt.show()
```

Execute the script using the **Job Manager** or from a Terminal. The plot will resemble the figure below, but your points will be more scattered, since the figure below was produced from a Hückel calculation with 10,001 energy points instead of 501.

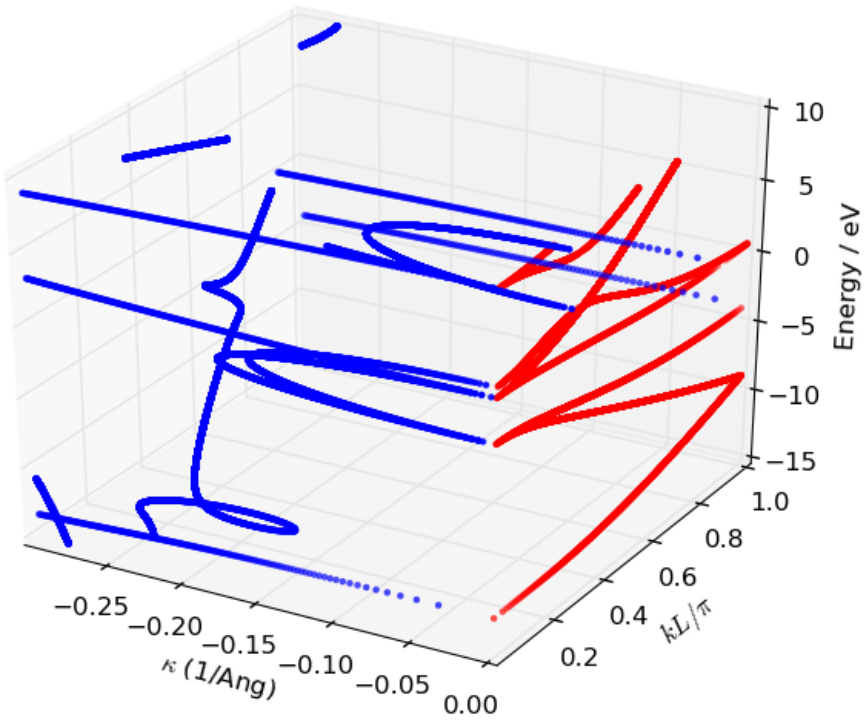


Fig. 3.35: 3D visualization of the complex bandstructure of Si(100). The real bands are plotted with red dots and the complex part in blue. Note how the complex bands connect with the real bands.

Another way to visualize the real part of the complex bandstructure is using colors. The script `2D_plot.py` does this (you can see the script below). It is a bit complicated towards the end, but that part is just for placing the colorbar, and could be omitted.

The forest of complex bands with a rather large value of κ are not usually seen in tight-binding simulations. However,

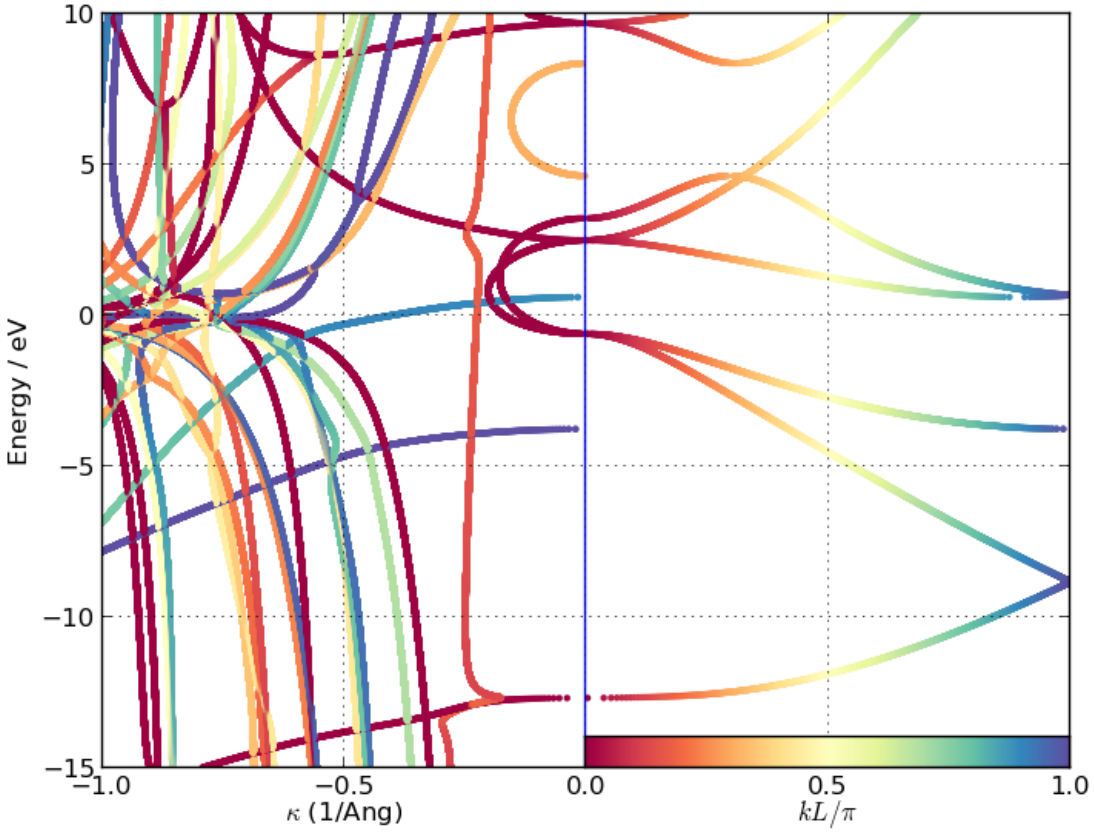


Fig. 3.36: 2D visualization of the complex bandstructure. Note how the color-coding of the k-values applies both to the real and the complex parts of the bandstructure, which makes it easier to see where the complex bands are attached to the real ones.

for DFT and Hückel, the basis sets are larger, so there are more complex bands as well, connecting unoccupied levels with various valence bands.

```
from QuantumATK import *
import matplotlib.pyplot as plt
import math

# Read the complex bandstructure object from the NC file
cbs = nlread('si_100_cbs.nc', ComplexBandstructure)[-1]
energies = cbs.energies().inUnitsOf(eV)
k_real, k_complex = cbs.evaluate()

ax = plt.axes()
cmap="Spectral"

# First plot the real bands
kvr = numpy.array([])
e = numpy.array([])
for (j, energy) in enumerate(energies):
    k = k_real[j]*cbs.layerSeparation()/math.pi
    if len(k)>0:
        e = numpy.append(e,[energy,]*len(k))
        kvr = numpy.append(kvr,k)

# Plot
ax.scatter(kvr, e,
           c=numpy.abs(kvr),
           cmap=cmap, marker='o', linewidths=0, s=10)

# Next plot the complex bands
kvr = numpy.array([])
kvi = numpy.array([])
e = numpy.array([])

for (j, energy) in enumerate(energies):
    if len(k_complex[j])>0:
        kr = [numpy.abs(x.real) for x in k_complex[j]]
        ki = [numpy.abs(x.imag) for x in k_complex[j]]
        e = numpy.append(e,[energy,]*len(kr))
        kvr = numpy.append(kvr,kr)
        kvi = numpy.append(kvi,ki)

# Plot with color depending on the imaginary part (corresponding to real k-points)
sc = ax.scatter(-kvi, e,
                c=kvr,
                cmap=cmap, marker='o', linewidths=0, s=10)

# Put on labels and decorations
ax.axvline(0,color='b')
ax.grid(True, which='major')
ax.set_xlim(-1, 1)
ax.set_ylim(-15, 10)
ax.annotate('$\kappa$ (1/Ang)', xy=(0.25,-0.07), xycoords="axes fraction", ha="center")
```

(continues on next page)

(continued from previous page)

```
ax.annotate('$kL / \pi$', xy=(0.75,-0.07), xycoords="axes fraction", ha="center")
ax.set_ylabel('Energy / eV')

# Add a colorbar
fig = plt.gcf()
x1, x2, y1, y2 = 0., 1, ax.get_ylim()[0], ax.get_ylim()[0]+1
trans = ax.transData + fig.transFigure.inverted()
ax_x1, ax_y1 = trans.transform_point([x1, y1])
ax_x2, ax_y2 = trans.transform_point([x2, y2])
ax_dx, ax_dy = ax_x2 - ax_x1, ax_y2 - ax_y1
cmap_axes = plt.axes([ax_x1, ax_y1, ax_dx, ax_dy])
a = numpy.outer(numpy.arange(0,1,0.01),numpy.ones(10)).transpose()
cmap_plt = plt.imshow(a,aspect='auto',cmap=plt.get_cmap(cmap),origin=(0,0))
ax = plt.gca()
ax.set_xticks([])
ax.set_yticks([])
ax.set_xticklabels([])
ax.set_yticklabels([])

plt.show()
```

Hint: You may note that there are gaps in the visualizations. The reason is that, unlike a normal bandstructure plot, the data is not plotted with lines, but with dots. In a standard bandstructure plot you can – with some level of confidence – define bands, which continuously run between the symmetry points (only band crossings can create some small problems). In the present case, the number of solutions is first of all different at each energy (particularly on the complex side), and depending on the density of the energy sampling, you may not hit a particular band close to points where the bands are very flat (heavy effective mass). This can to some extent be alleviated by increasing the number of energy points.

References

3.2.10 InAs p-i-n junction

Version: 2017

The purpose of this tutorial is to show how to set up and perform calculations for a device based on an InAs slab. You will define the structure of a H-passivated InAs slab along the (100) direction, and set up a p-i-n junction.

The underlying calculation engine for this tutorial is ATK-SE. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK-SE section of the *ATK Reference Manual*.

Tip: For more basic properties of InAs slabs, see this tutorial: *Meta-GGA and 2D confined InAs*

Setting up the device geometry

Creating a passivated InAs slab

Start QuantumATK and launch the **Builder** by pressing the icon on the toolbar.

•In the builder, click *Add* → *From Database*.

•Type InAs in the search field and double-click on the row with InAs.

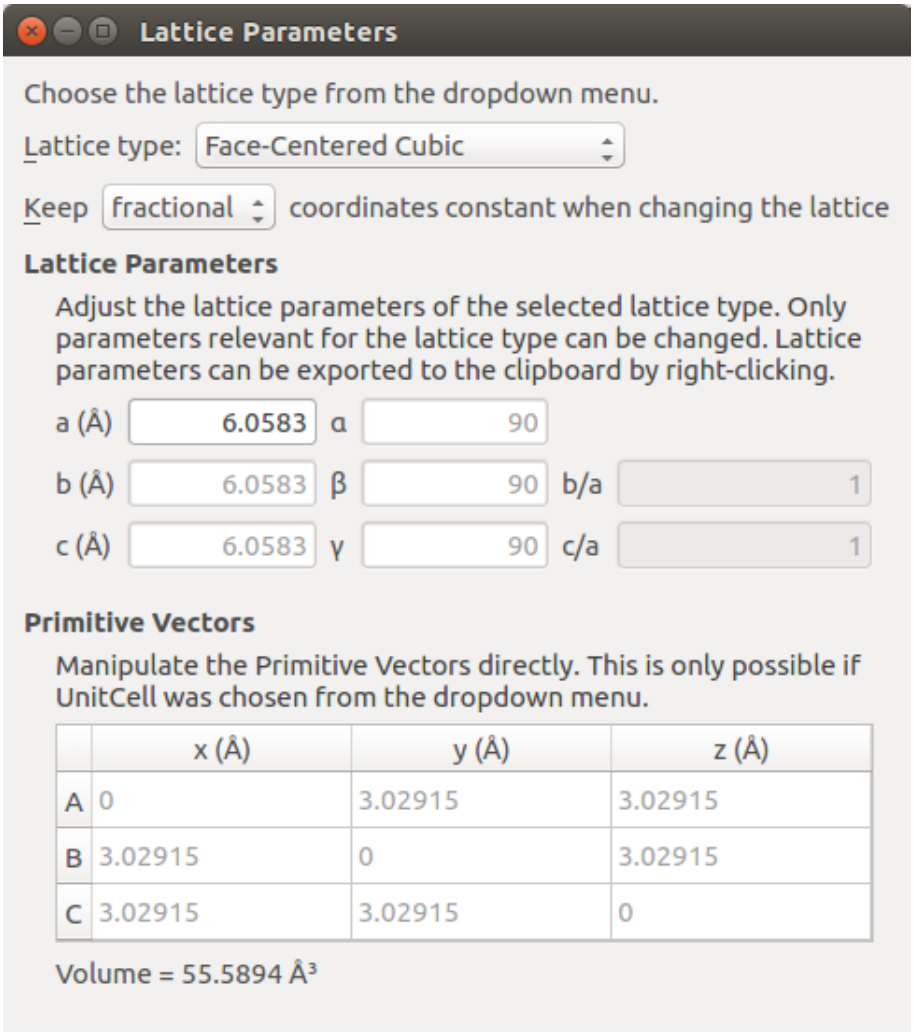
•Click the icon in the lower right-hand corner of the **Database** window to add the structure to the **Stash** in the **Builder**.

The next step is to change the lattice constant to the experimental room temperature lattice constant, a = 6.0583 Å[1].

•In the panel to the right open *Bulk tools* → *Lattice Parameters*.

•Set *Keep fractional coordinates*.

•Set a = 6.0583 Å.



[1] New Semiconductor Materials - Characteristics and Properties. InAs. http://www.ioffe.rssi.ru/SVA/NSM/Semicond/InAs/index.html. Accessed: 2017-05-31.

We now have a bulk configuration with the correct lattice parameter, and the next step is to create a slab with the right orientation. Open *Builders* → *Surface (Cleave)* to cleave the crystal. In this case, along the (100) direction.

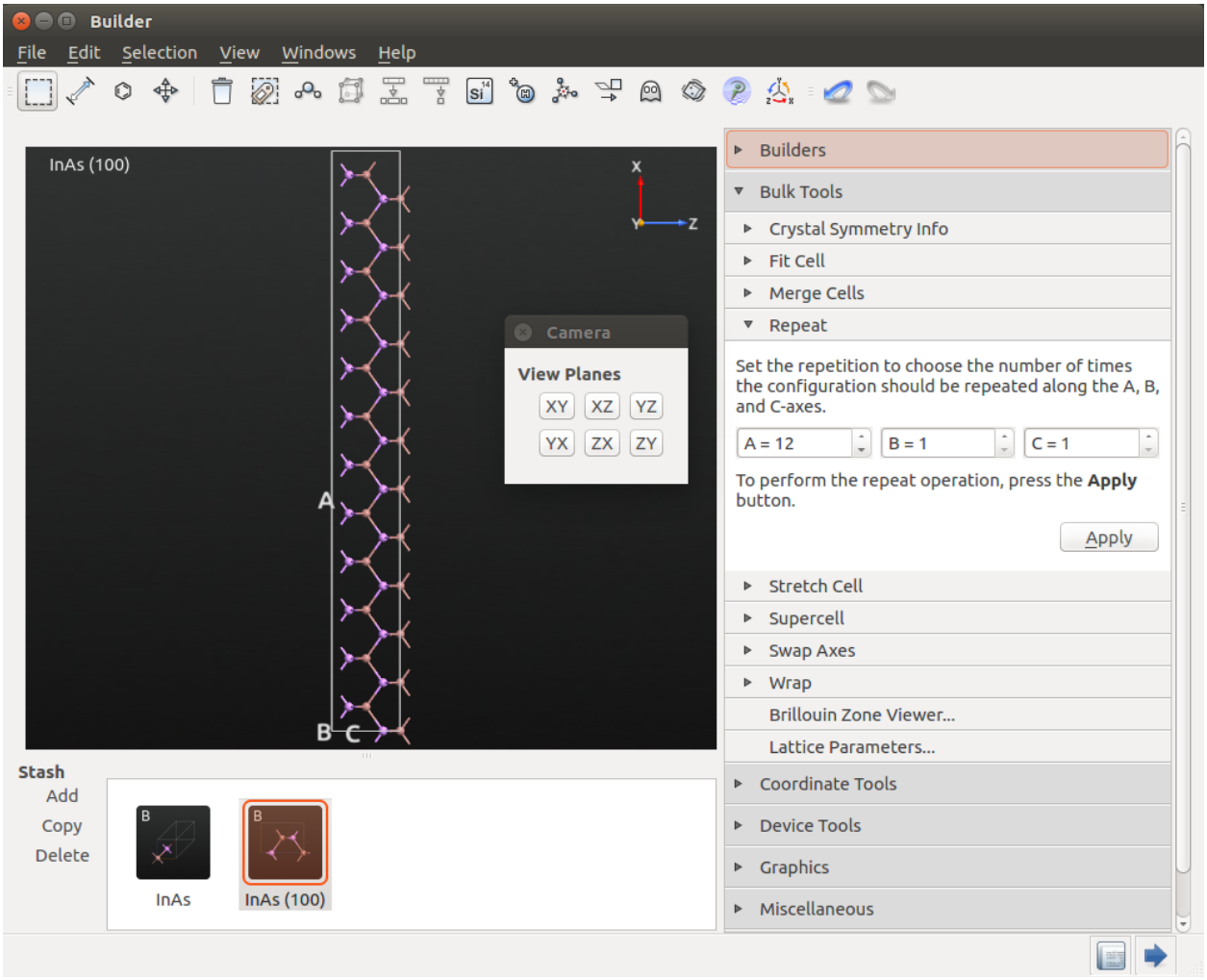
•Keep the default (100) cleave direction, and press *Next* >.

•Keep the default surface lattice, and press *Next* >.

•Keep the default settings on the last page, and press the *Finish* button to add the cleaved structure to the **Stash**.

•Open *Bulk Tools* → *Repeat* and enter A = 12, B = 1, and C = 1, and press *Apply*.

To get a good look at the structure, click on the viewplane icon and choose *ZX*. You should now see the following:



At this point you should register the volume of this InAs cell. Open *Bulk Tools* → *Lattice Parameters*. The volume is printed in the **Lattice Parameters** window: 1334.15 Å^3. You will need this value in the next section when we set up the doping of the p-i-n device.

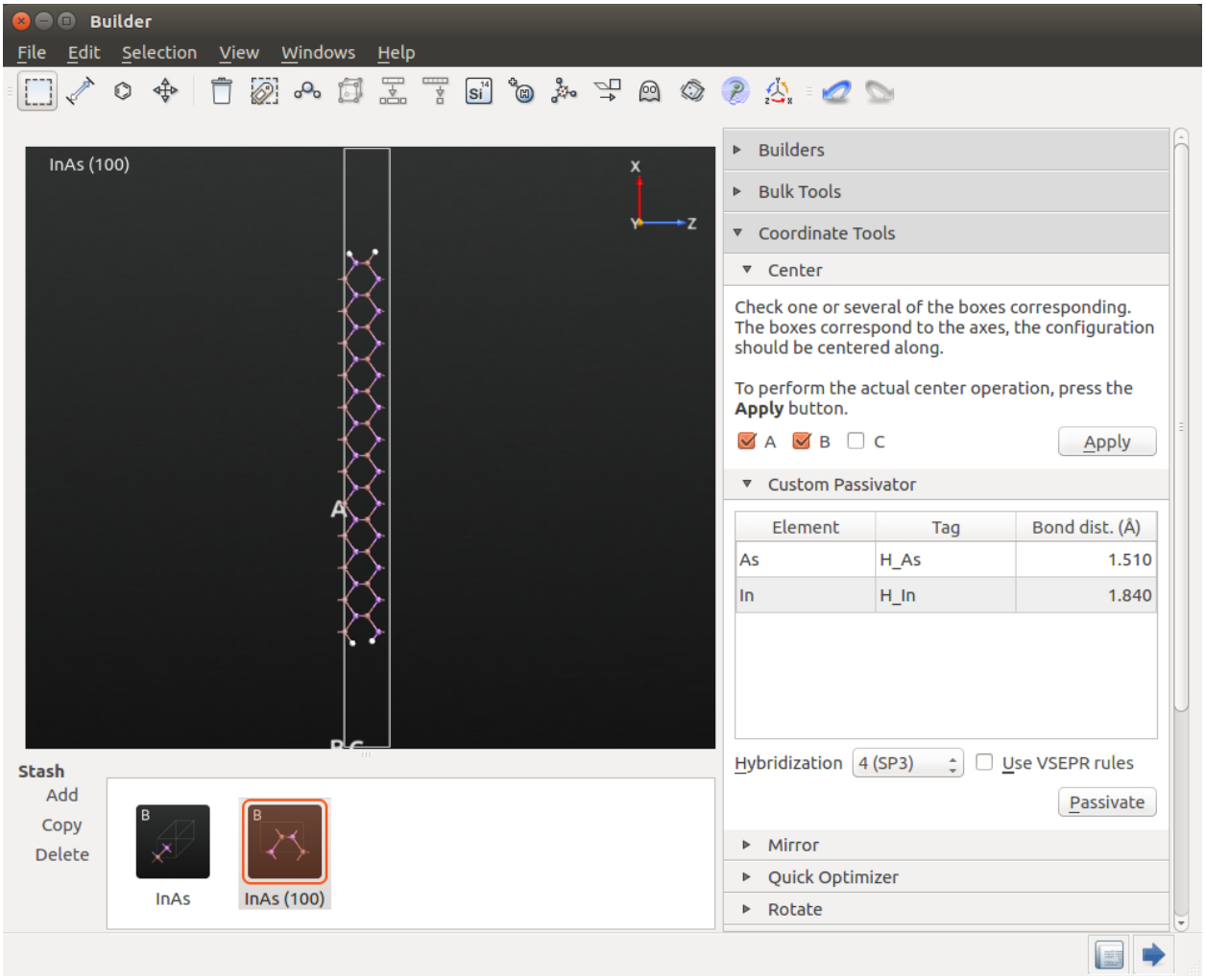
To finalize the setup, perform the following steps:

•Open *Bulk Tools* → *Lattice Parameters*. Select *Keep Cartesian*, and set the length of the *A* vector to 80 Å.

•Open *Coordinate Tools* → *Center* and center the structure in the *A* and *B* directions.

•Open *Coordinate Tools* → *Custom Passivator*. Select Hybridization *4(SP3)*, and passivate the structure.

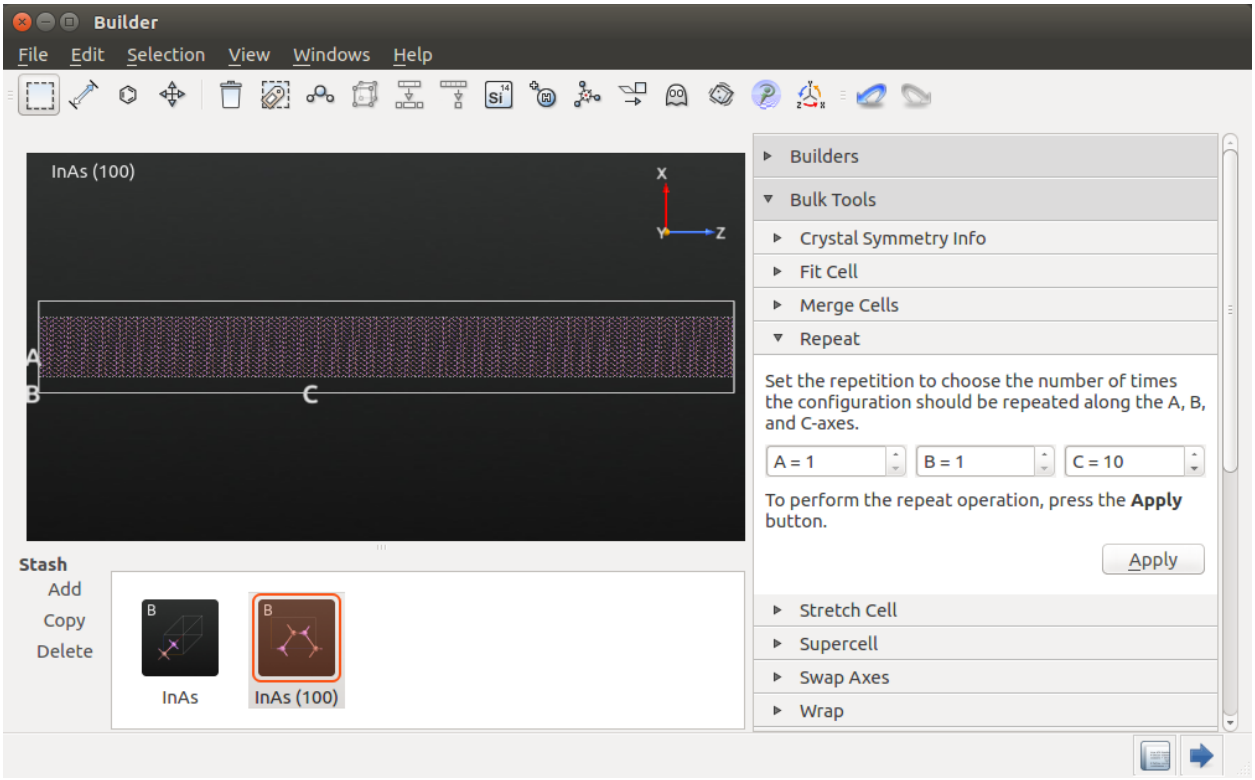
Your structure should now resemble the one shown below.



Warning: In previous versions of QuantumATK (2016 and earlier), we have identified a bug which caused some of the H atoms to miss their tags in some cases. If you use a version where this bug might apply, please check the tags of the H atoms, and manually apply the correct tag to any that are missing. We need these tags later when applying compensation charges and doping.

Creating the FET device geometry from the slab

We will make a 60 nm long device, thus the unit cell must be repeated 100 times along C. Therefore, open the *Bulk Tools* → *Repeat* panel, set A = 1, B = 1 and C = 10, and press *Apply* twice. Your structure should now look like this:



Warning: The repeat panel remembers your settings from last time, so remember to change the value for *A* back to 1. If you forgot this, you can use the undo button .

Adding dielectric regions and gates

The next step is to add 1 nm thick dielectric regions on both sides of the device with a double gate in the center.

•Open *Miscellaneous* → *Spatial Regions*.

•Right-click in the white area of the widget and click *Insert region* to insert a new spatial region. You can also click on the icon below the white area.

•Change the spatial region into a dielectric region and set the dielectric constant to 3.9. Define in fractional coordinates:

–*a*: 0.0375–0.165,

–*b*: 0–1,

–*c*: 0–1.

•Add another dielectric region with dielectric constant 3.9. Define in fractional coordinates:

–*a*: 0.8375–0.9625

–*b*: 0–1

–*c*: 0–1.

•Add a metallic region with value 0.0 Volt. Define in fractional coordinates:

–*a*: 0–0.0375

–*b*: 0–1

–*c* :0.33–0.67.

•Add another metallic region with value 0.0 Volt. Define in fractional coordinates:

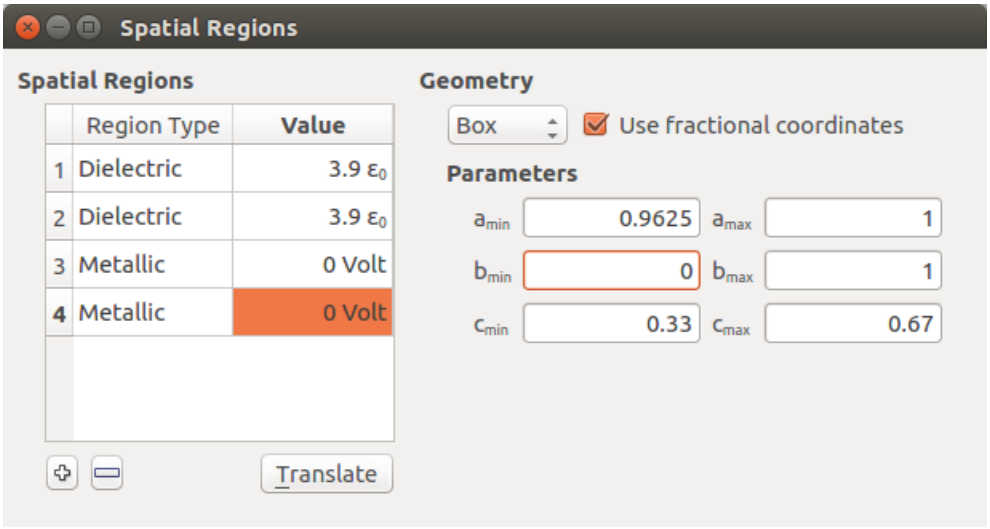
–*a*: 0.9625–1,

–*b*: 0–1

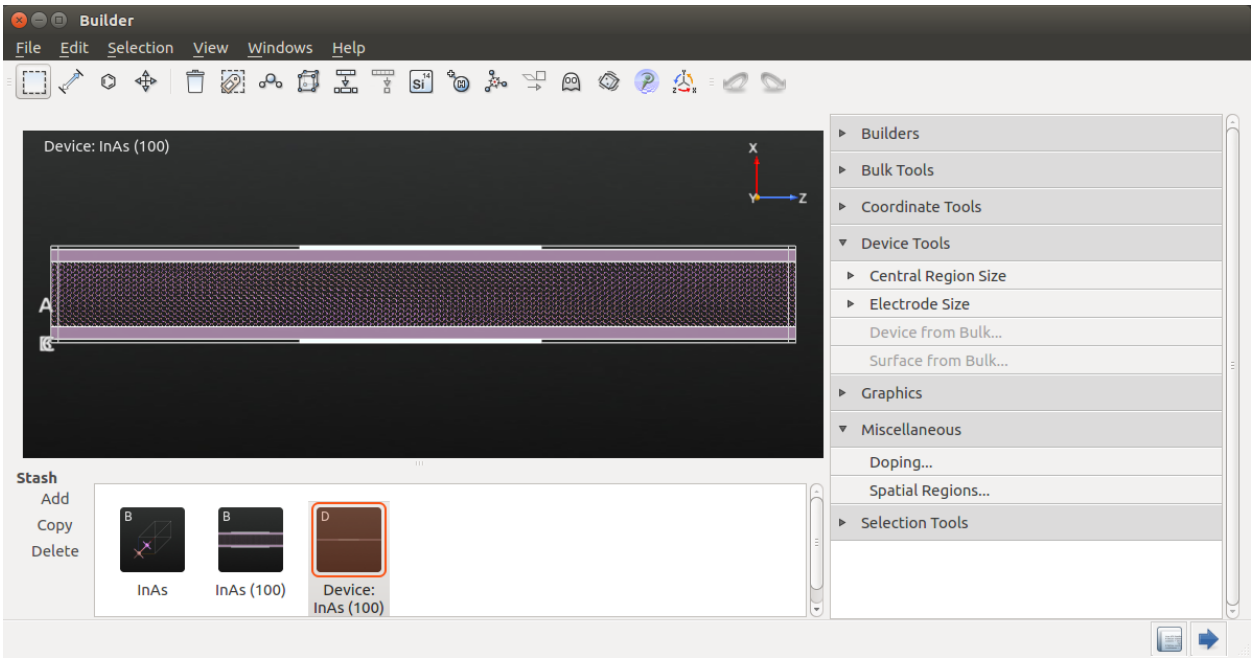
–*c*: 0.33–0.67.

Note: Note that you can switch between fractional and cartesian coordinates, if you prefer a particular representation of coordinates.

At the end, the widget should look like this, and you can close it:



Finally, transform the structure into a device configuration by opening *Device Tools* → *Device From Bulk*. Accept the default suggestion for the electrode lengths of a = 6.0583 Å. The dielectric regions are automatically extended to the electrodes as well.



Warning: In previous versions of QuantumATK (2016 and earlier), this extension was not done automatically. If you use an older version, you can add the dielectric regions to the electrodes in the script, by inserting the contents of this file somewhere after the two electrodes have been defined: `dielectric-regions.py`

Adding doping to the configuration

Finally, we add the doping to the configuration, which includes both regions of In and As atoms, as well the passivating hydrogen atoms. As InAs is a III-V material, the passivating hydrogens should have fractional charges, as explained in this tutorial: *Meta-GGA and 2D confined InAs*.

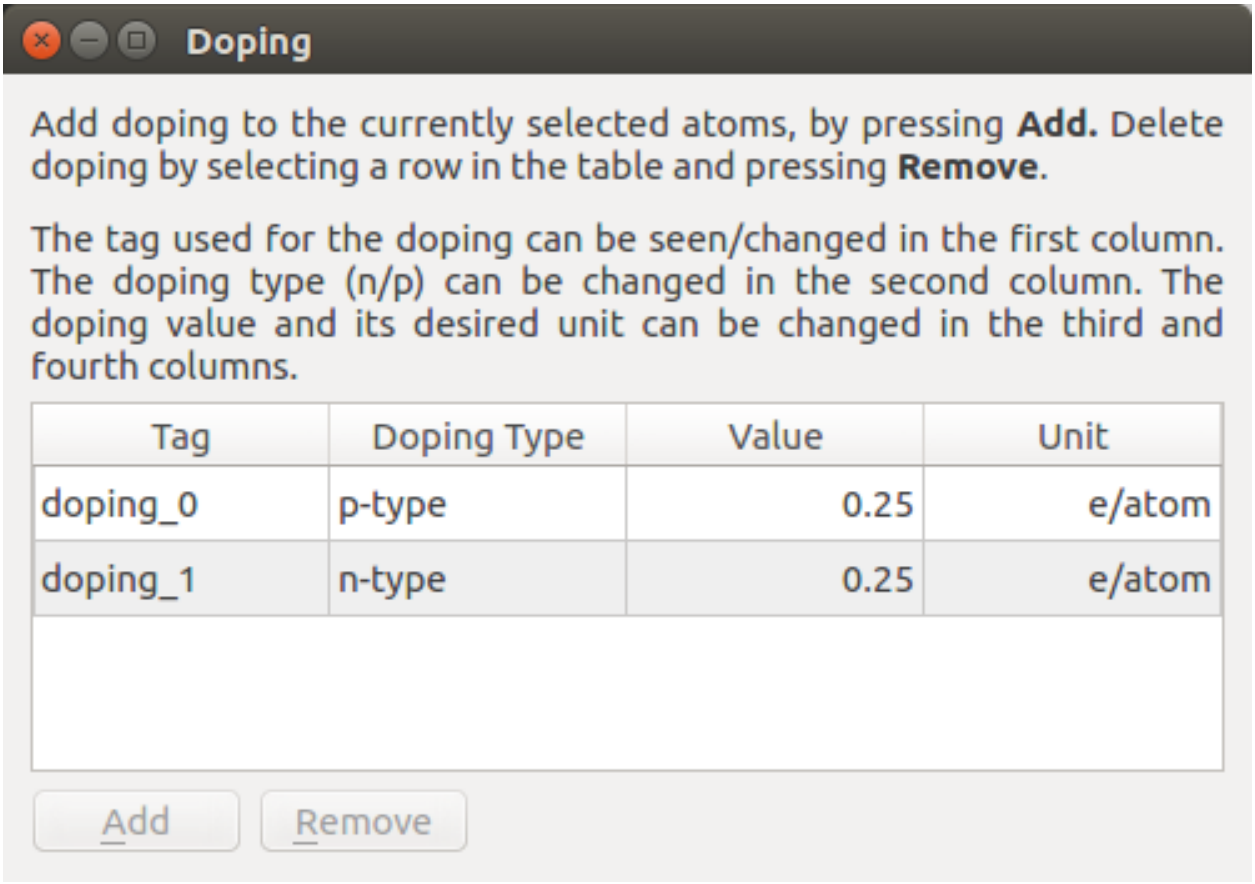
Note: The doping can also be applied via a script. Send the device configuration to the **Editor**, and append the contents of this script at the end: `add-doping.py`. You have now created the device, and are ready to set up the actual calculations.

Adding fractional charges to the H atoms

To add the correct fractional charges to the H atoms, go to the panel on the right, and select *Selection Tools* → *Tags*.

- Click on *H_As*, which will select the H atoms bonded to As atoms.
- Open *Miscellaneous* → *Doping*, click **Add**, change the **Unit** to *e/atom* and the **Doping Type** to *p-type*.
- Double-click on the number under **Value** and type 0.25 to apply a fractional charge of -0.25 electron to these H atoms.

Now, go back to *Selection Tools* → *Tags* and select *H_In*. Repeat the process above, but choose *n-type* to apply a charge of +0.25 electron instead. The doping widget should now look like this:



Tip: When using the doping widget, QuantumATK automatically assigns a tag to each atom, and you can easily edit these tags to make them more meaningful for your application.

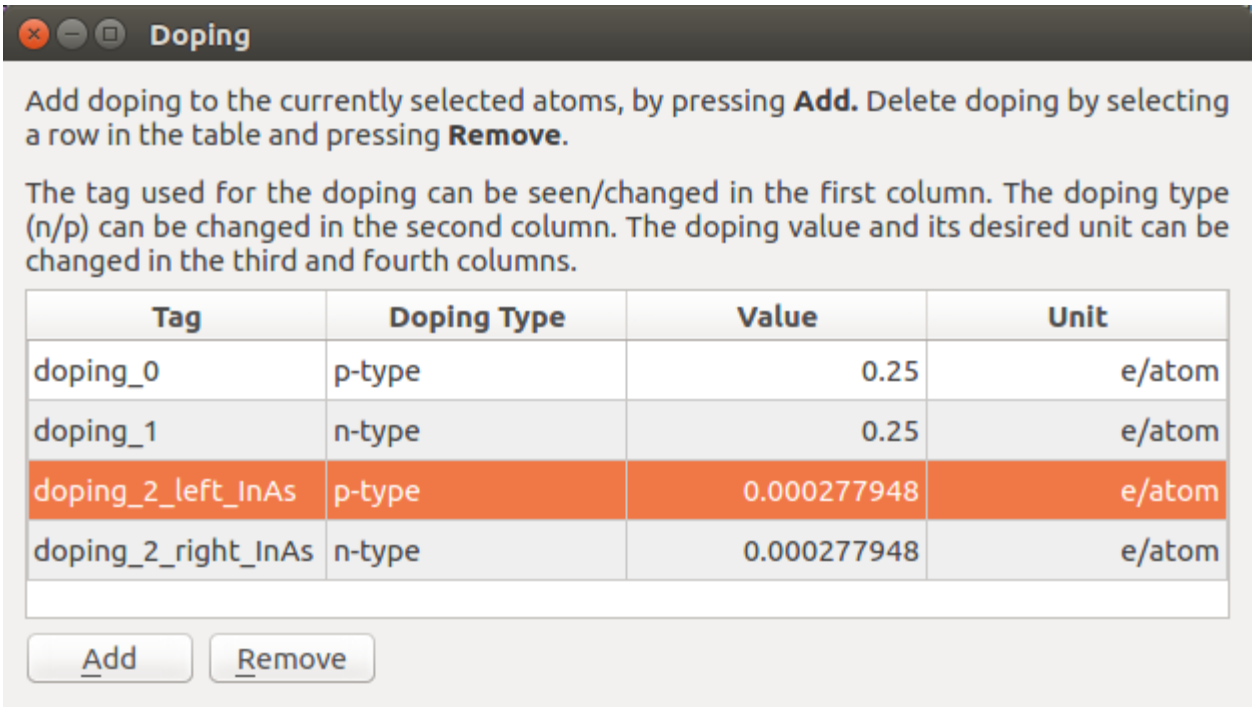
Adding p-type and n-type doping to InAs

Secondly, we dope the InAs with p-type doping at one end and n-type at the other, with an intrinsic region in the middle. Here, we will also use the doping widget, which can handily apply a doping with a specific volume concentration. However, because we already have vacuum in the configuration, we cannot use this feature, as it would be based on a too large volume. We would like a doping concentration of 10^{19} per cm^3, and with the 48 atoms and 1334.15 Å^3 volume of the un-repeated slab, this works out to a doping value of 0.0002779479 e/atom, irrespective of the vacuum.

First, we will apply the p-type doping to the left side of the device. We will use the gate to define the intrinsic region, so zoom in on the left side of the device so that you can still see the whole electrode and the edge of the gate.

Tip: If you hold down the `shift` key and the right mouse button at the same time, you can move the configuration around the window. This is very useful in this case. See also the *Mouse and key operations* section of the *Builder Manual*.

Now select all the In and As atoms to the left of the gate, while not hitting any H atoms. Open the doping widget again, and apply a p-type doping of 0.0002779479 e/atom. Do the same to the right of the gate, but instead selecting n-type here. The doping widget should now look like this:



The device is now finished, and you are ready to begin calculations. You can download the full geometry here: `full-device.py`.

Running the calculations

Send the updated script from the **Editor** to the **Script Generator**.

1. Add a **New Calculator**

- Select the *ATK-SE: Slater-Koster* calculator
- Set the *k-point sampling* to 1, 11, 100 and uncheck the *No SCF iteration* box.
- **Iteration control parameters**: set the number of *history steps* to 10. This will slightly improve the convergence for this system and it will reduce the memory consumption.
- **Device algorithm parameters**: Set the *Self energy* method to *Sparse Recursion* for calculating both equilibrium and non-equilibrium contour integrals.
- **Poisson Solver**: Select the *[Parallel] Conjugate Gradient* method and *Neumann* boundary conditions in the A-direction, where the gates are located.
- **Slater-Koster basis set**: Select the *Bassani.InAsH* basis set.

2. Add an **ElectrostaticDifferencePotential** object.

3. Add a **ProjectedLocalDensityOfStates** object.

- Choose *Device DOS* as the **Method**
- Set the k-point sampling to 1, 15.

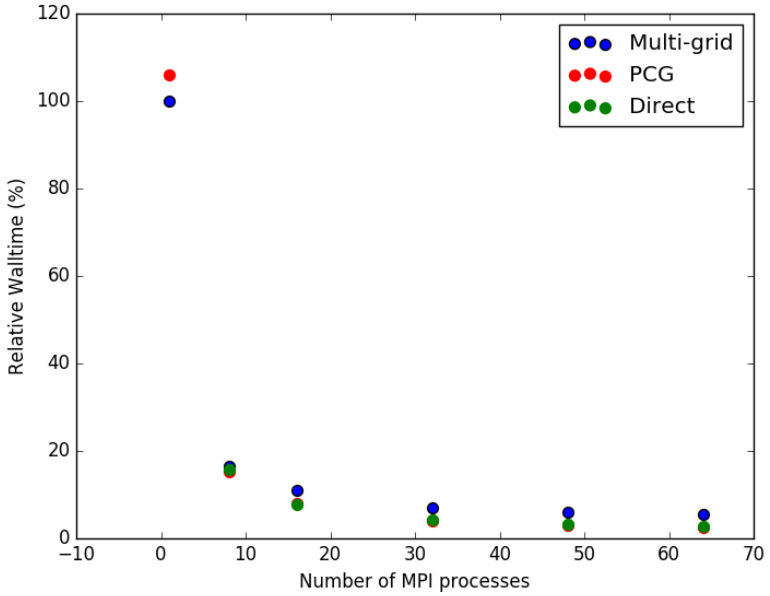
4. Set the output filename to `InAs_5nm_pin.nc`

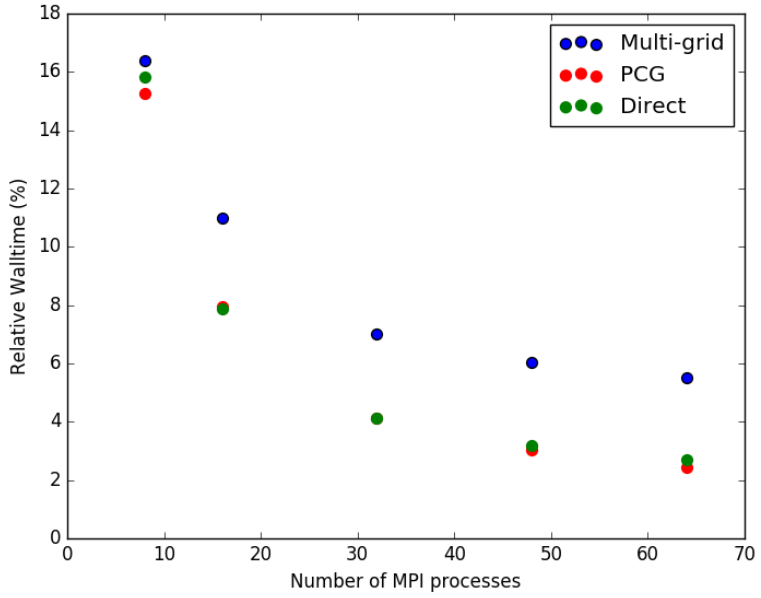
5.Run the script.

You can also download the full script here: `full-script.py`.

Choice of the Poisson Solver

For this system, the choice of Poisson solver is quite important, and is directly related to the number of cores or MPI processes the script will be run on. In this case threading is less efficient than MPI parallelization, so we will not treat it here. See figures below for an overview of the performance for the three methods when running on different numbers of cores. First is the timing relative to the Multi-grid method in serial, and second is the exact same figure, but with the serial data-points omitted. The index number (equivalent to 100 %) is 67,532 seconds, or a little less than 19 hours.





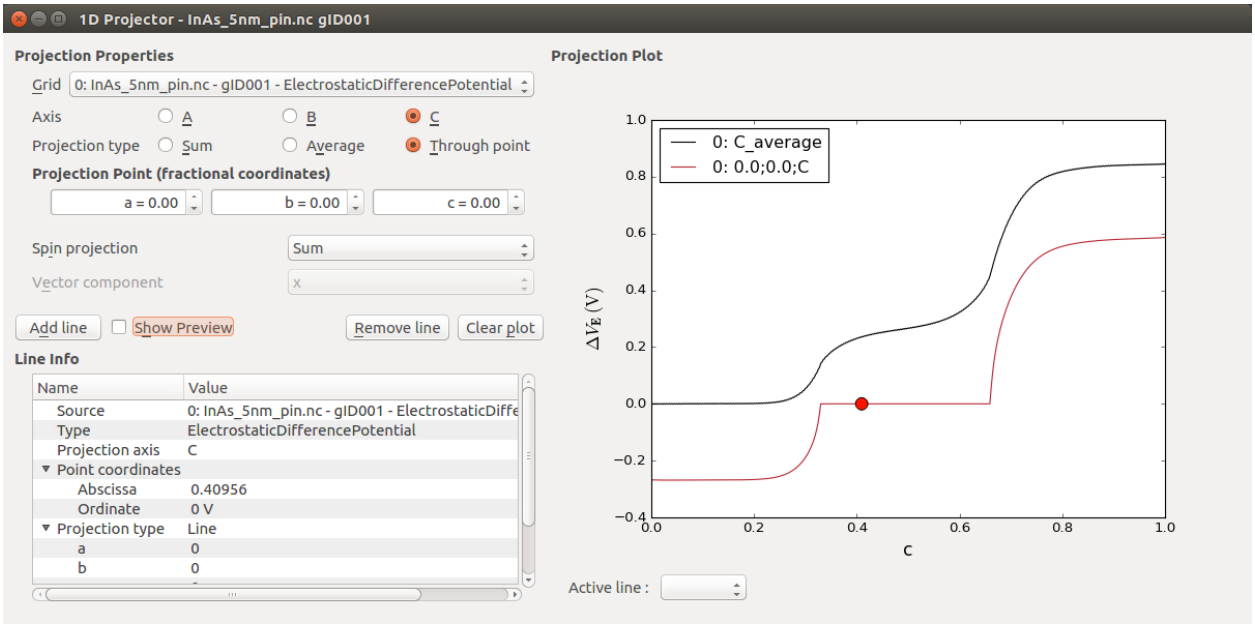
To summarize, one should use the Multi-grid solver if running in serial and either the Parallel Conjugate Gradient or the Direct method if running in parallel. However, the memory footprint of the PCG method is much smaller than for the Direct solver, so in most cases PCG will be preferable of the two.

Analysis of the results

You can now plot the electrostatic potential. From the **LabFloor**, select the *ElectrostaticDifferencePotential* object contained in the file `InAs_5nm_pin.nc`. Afterwards, use the 1D Projector on the right-hand side and plot the average potential along the C axis.

In the calculation, you have used a gate potential of zero volt. Since you use Neumann boundary conditions you have an arbitrary vacuum level. To see the vacuum level of the left or the right electrode, plot the electrostatic potential at the edge. As **projection type** select *through point* and use the *0, 0, 0* fractional coordinates. This will now be added as a red line, in addition to the black line showing the average.

You can inspect the potential at the edge of the cell (edge of the vacuum region), in addition to the average plotted previously. From the plot you can extract the two vacuum levels, left = -0.28 V, right = 0.58 V.



The zero energy reference for this vacuum level is defined by the construction of the basis set. In the next section, you will learn how to align the vacuum level with the Fermi level and the band edges of your device. By doing so you will be able to calculate the work function of your metal gate and finally tune it to match the work function of a gold gate, for example.

Finally, it is worthwhile to plot the electrostatic potential directly on top or your device configuration. Select the *ElectrostaticDifferencePotential* object and click on the **Viewer** (or drag and drop it on top of the Viewer icon:).

Here, you can choose to plot it as an isosurface or as a cut plane. The figure below shows a 3D view of a cut plane along the *CA* plane where you can nicely see the atomic resolution of the electrostatic potential of your 60 nm InAs device.

Tip: If you wish to re-create the figure, you need to do the following:

1. Drop the *ElectrostaticDifferencePotential* object on the Viewer and choose *Cut plane*.
2. Click on *Properties* on the right.

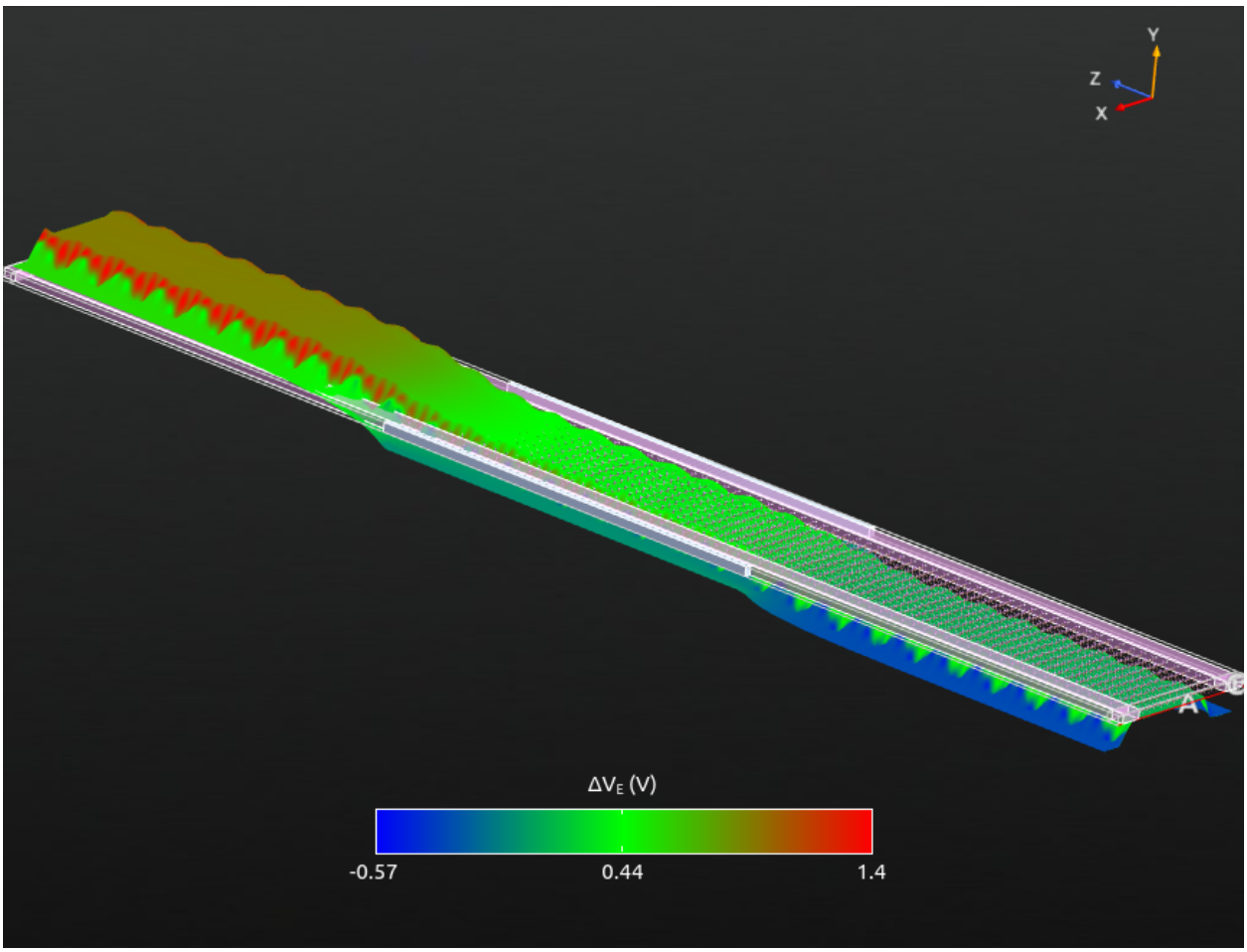


Fig. 3.37: 3D view of the ElectrostaticDifferencePotential across the device.

3.Under **Surface Properties** change the value of *3D* to 0.03.

4.In the bottom left, click the button *AC*.

5.On the lab floor, locate the *DeviceConfiguration* object, and drag it onto the cut plane window.

6.Use the mouse to re-orient the figure to your liking.

Defining the work function of the metal gate

Note: In the following, gate usually refers to the entire gate region, and the gate voltage is thus applied to both metallic regions.

In order to calculate the metal gate work function you will need to align the vacuum levels extracted in the previous chapter with the Fermi level and the band edges of your InAs device. As described in the first section, you need to perform this operation since these levels are not related to each other within the definition of this Slater-Koster basis set.

At zero applied voltage between the left and right electrodes, the Fermi levels will coincide and the band diagram of the whole system corresponds to the one shown in the following figure.

The choice of your zero energy reference is quite arbitrary. In the following, you will consider the Fermi level at zero bias to coincide with the zero energy in the scheme above, $E_F = 0$ eV. In order to set the vacuum levels, and in particular the left electrode vacuum level, you need to know:

1.The positions of the band edges, valence band maximum (VBM) and conduction band minimum (CBM), see the next section: *Band gap and band edges*.

2.The electron affinity (E_{aff}) defined as the energy difference between the CBM and the vacuum level. This can be computed in a manner similar to the metal work function, as described in this tutorial: *Greens function surface calculations*.

As a first approximation you could use the values reported in the literature for InAs which are: 4.55 eV[2] and 0.451 eV at 0 K[Page 595, 1] for the electron affinity and band gap of InAs, respectively. However, the band gap and the electron affinity of your device can be quite different from the bulk value due to finite size effects. Here, you will instead use a calculated value of the electron affinity for a 2d-InAs slab of 5.09 eV, and you will determine the band edges and band gap of your gated device.

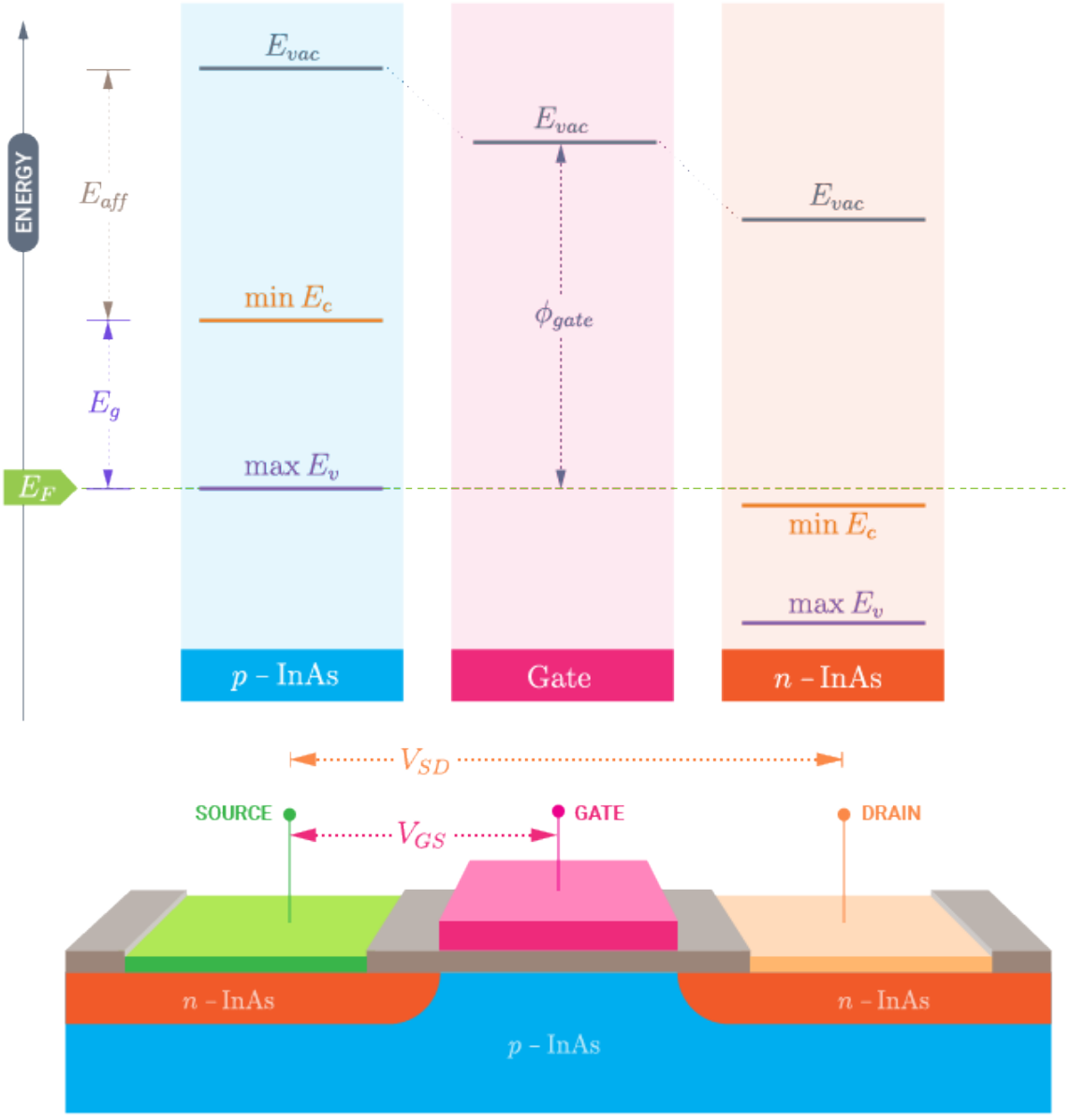
Band gap and band edges

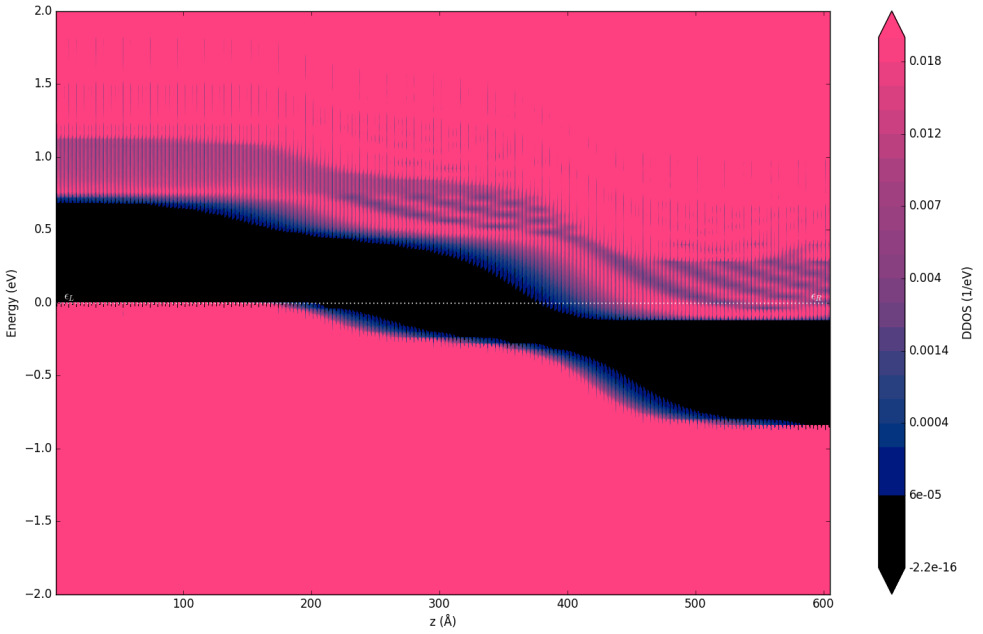
You can easily plot the density of states across your device from the *DeviceDensityOfStates*, *LocalDeviceDensityOfStates* or *ProjectedLocalDeviceDensityOfStates* analysis objects available in QuantumATK. Here you will use the *ProjectedLocalDeviceDensityOfStates* tool available from the QuantumATK 2015 release, and which we already calculated previously. Select this object on the LabFloor and click on the *Projected Local Density of States* plugin on the right-hand side to open the corresponding window. In order to highlight the band gap and band edges, the figure above has been created by narrowing the data range for the DDOS values to 0-0.02 1/eV Å^3 (click on the *Data range* button in the upper right corner).

This plot nicely shows the electronic structure across the full length (about 60 nm) of your device. In particular, you can immediately see the band bending due to different doping types in the left and right parts of the device.

From this plot, you can extract the band gap of 0.68 eV by zooming in on the p-doped region, where the position of the VBM is exactly on the Fermi level.

[2] T.ăE. Fischer, F.ăG. Allen, and G.ăW. Gobeli. Photoelectric Emission from InAs: Surface Properties and Interband Transitions. *Phys. Rev.*, 163:703–711, Nov 1967. doi:10.1103/PhysRev.163.703.


ENERGY
E_{aff}
E_g
E_F
E_{vac}
min E_c
max E_v
E_{vac}
ϕ_{gate}
E_{vac}
min E_c
max E_v
p – InAs
Gate
n – InAs
V_{SD}
SOURCE
V_{GS}
GATE
DRAIN
n – InAs
n – InAs
p – InAs



Tip: The projected local density of states window also shows the corresponding transmission spectrum, if present in the LabFloor, on the left-side of the PLDOS plot.

Tuning the work function of the metal gate

You now have all the ingredients to define the electronic structure of your device and align the Fermi levels, vacuum levels and band edges.

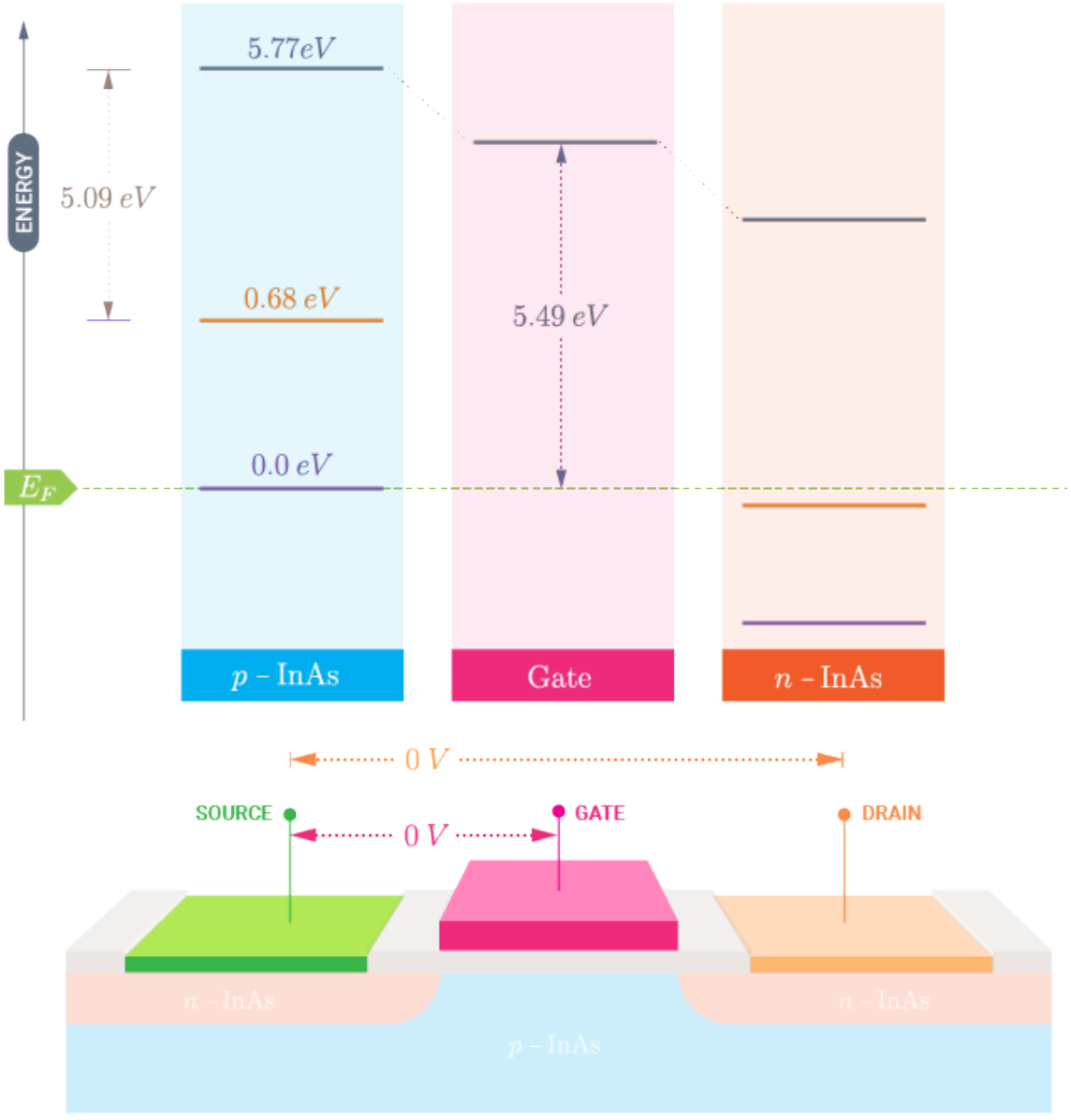
From your actual set-up you thus have a metal gate with an intrinsic work function of 5.5 eV. This can be calculated as: $E_G + E_{aff} - E_{vac,left}$. Note that the value for the vacuum level from earlier needs to be multiplied by -*e* before being inserted here. This value may correspond to the work function of Pt, 5.12/5.93 eV as reported in the CRC Handbook[3].

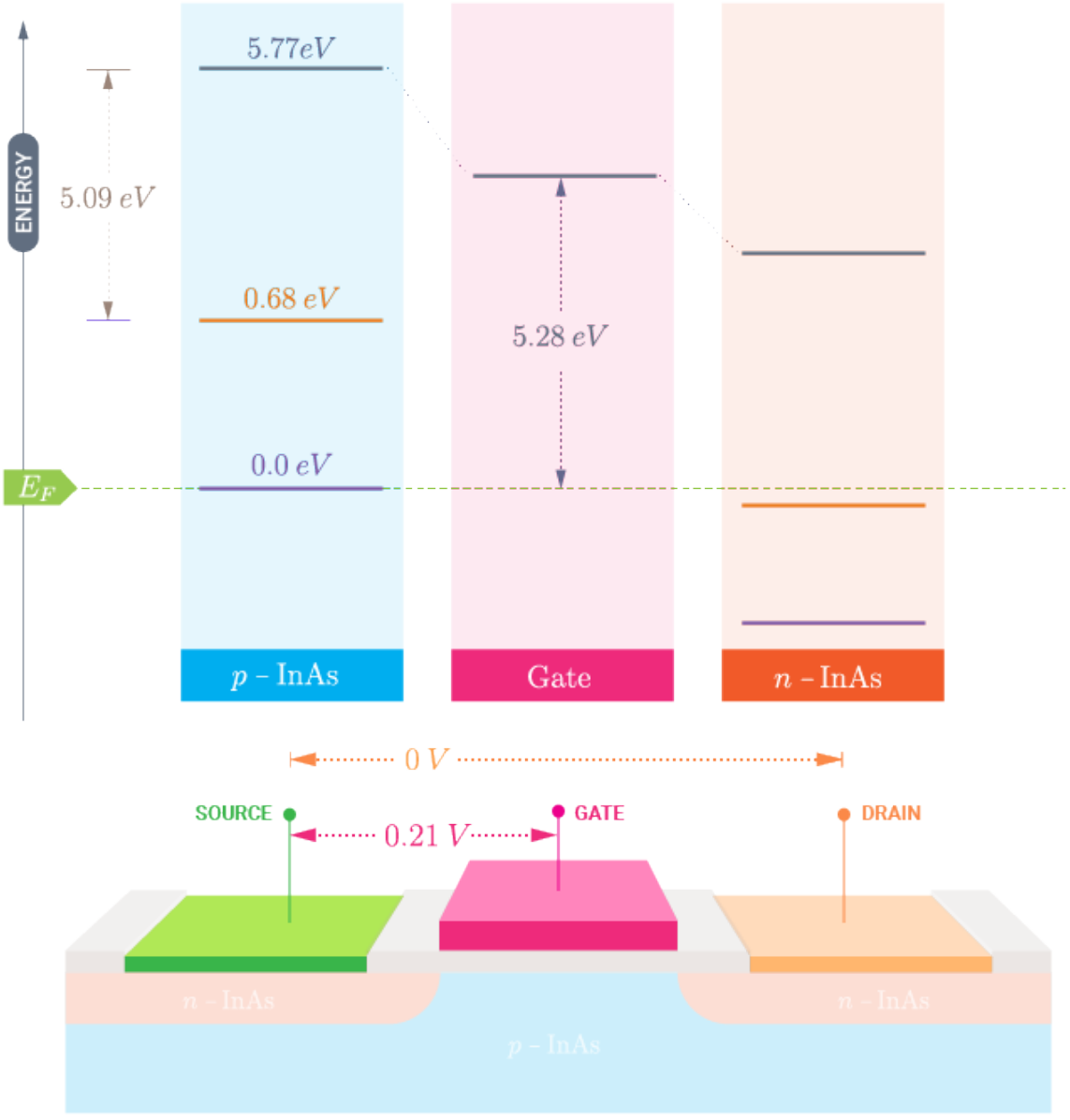
You can now tune the work function of your metal gate by applying a gate voltage to adjust to your experimental conditions. In particular, you can lower or increase the metal gate work function by applying a positive or negative gate bias, respectively.

For example, in case of a gold gate, with a work function of 5.28 eV[Page 608, 3], you will need to apply a +0.21 V gate voltage to have the correct energy reference for a grounded gold left electrode.

In the next chapter you will perform the gate scan at finite bias.

[3] W.ăM. Haynes, editor. *Electron Work Function of the Crystalline Elements, in CRC Handbook of Chemistry and Physics*. CRC Press/Taylor & Francis, Boca Raton, FL, 97th edition (internet version 2017) edition, 2017.


ENERGY
5.09 eV
5.77eV
0.68 eV
0.0 eV
E_F
5.49 eV
p – InAs
Gate
n – InAs
0 V
SOURCE
0 V
GATE
DRAIN
n – InAs
n – InAs
p – InAs


ENERGY
5.77eV
5.09 eV
0.68 eV
5.28 eV
0.0 eV
E_F
p – InAs
Gate
n – InAs
0 V
SOURCE
0.21 V
GATE
DRAIN
n – InAs
n – InAs
p – InAs

Performing a gate scan

You will now apply a source-drain voltage, V(SD), of -0.5 V and perform a gate scan. To this end use this script: `gatescan.py`, which uses the converged zero bias calculation as a starting point to generate converged calculations at each bias. Again, you can get a good speed up by parallelizing over several MPI processes.

You can now analyze the behavior of the electrostatic difference potential as a function of the gate bias. From the figure below you can see that the effect of applying a positive gate voltage is to shift up the potential at the metal gates.

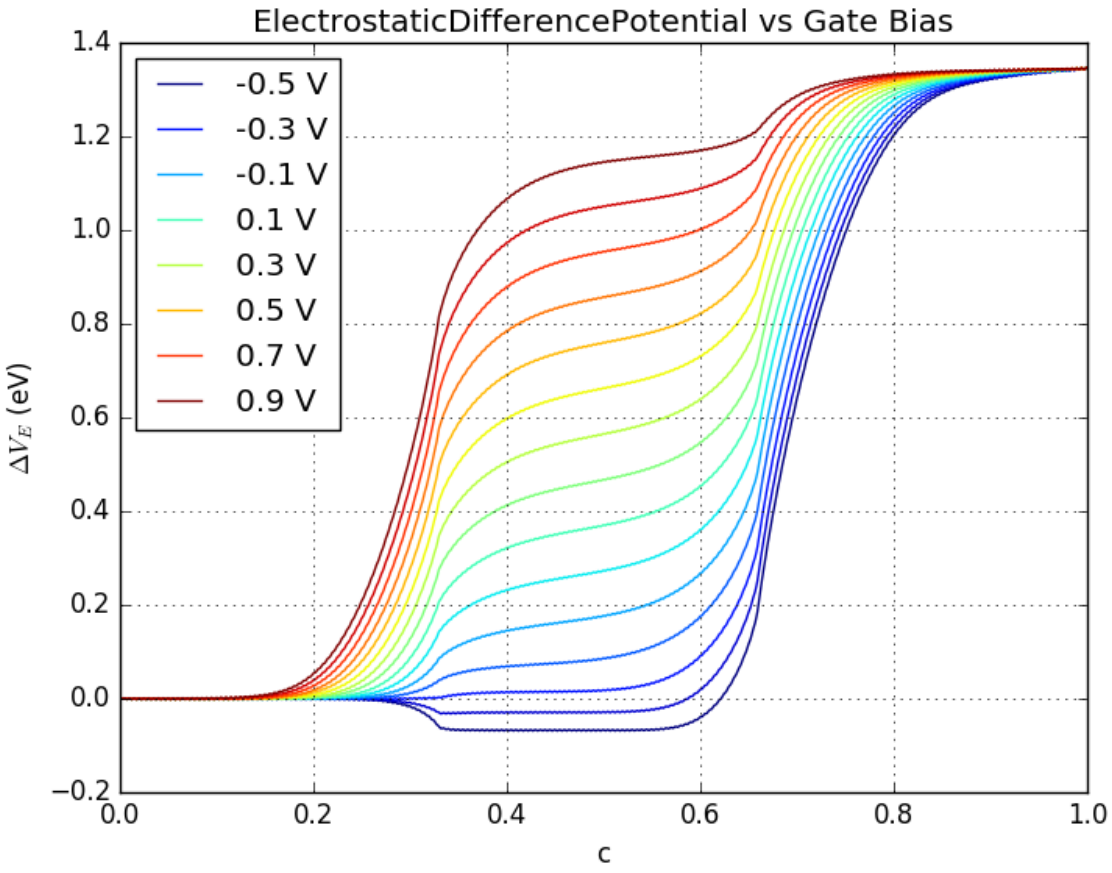


Fig. 3.38: ElectrostaticDifferencePotential vs gate bias between -0.5 and 0.9 V. You can recreate the plot with this script: `plot-gatescan-edps.py`.

You can now calculate the transmission spectra for the converged configurations. This script will calculate the transmission spectrum at each gate voltage: `transmission.py`. This calculation can be performed with high parallelization. We will return to these results after a small detour to look at a system where there is a smaller doped region on the right side, towards the *Drain*.

Simulating a drain underlap

You will now simulate a system where there is a drain underlap, i.e. the atoms 10 nm to the right of the central gate are not doped. In the following figure, this situation is shown, by comparing sketches of the doped areas in both the normal p-i-n case and with a drain underlap.

Fig. 3.39: Structure of the gated InAs p-i-n device and of the same device with a drain underlap of 1 nm. Source (S), drain (D) and gates (G) are indicated.

In the calculation, the drain underlap is created by only adding a compensation charge for the last 860 atoms. You can set up this system in several ways:

- If you are using the script you can set `n_central_right = 860`.
- Using the direct procedure of selection in the Builder is a little harder in this case, because we do not have the gate to use as guideline for the edge of the region. On the other hand, it is not that important to hit exactly in the middle in this case.
- You can also manually modify the tags for the n-doping in the script and delete the first half of the indexes (should be 3484 to 4344).

Also change the output filename to `InAs_5nm_pin_du.nc`. Once the calculation is finished, you can repeat the gatescan with the calculation of the electrostatic difference potential and transmission spectra as explained in the previous section.

Analysis of the results

We can now plot the I-V curve for both the regular p-i-n junction, and the system with a drain underlap, using an applied gate voltage from -0.5 V to 0.9 V. In the figure, the gate voltage is shifted to account for the work function of the metal gate, as explained previously. We also calculate the subthreshold slope, as indicated by the dashed black lines. We get values of 58 mV/dec for the regular device and 43 mV/dec for the device with a drain underlap. You can generate the plot yourself with the following script: `plot-iv-curve.py`

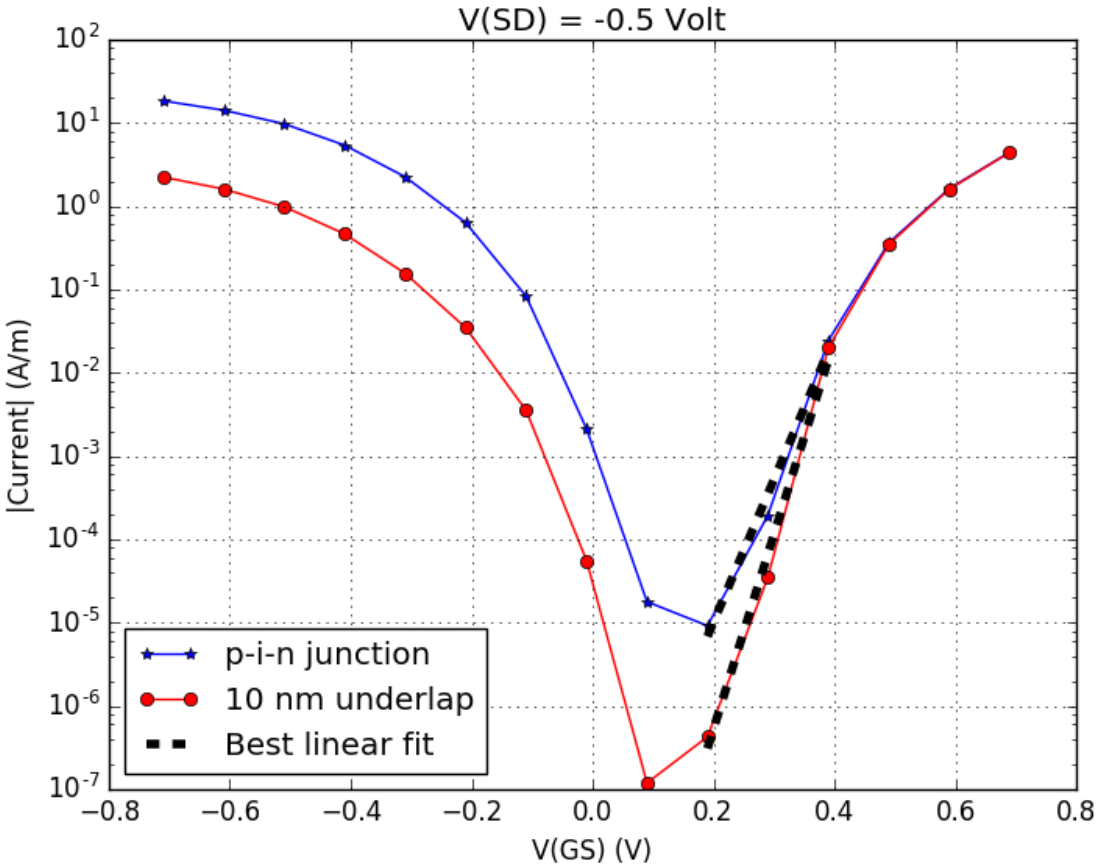


Fig. 3.40: Gate-Source bias, V(GS), dependence of the current for symmetric InAs-2D p-i-n junction. The gate voltage is reported relative to a gold metal gate with a work function equal to 5.28 eV, i.e. V(GS) is shifted by 0.21 V (see the section *Band gap and band edges*.

Finally, you can perform the projected local density of states analysis to any of the configurations saved in `gatescan_conf.nc` or `gatescan_conf_du.nc` as explained in the previous section. In the figure below the PL-DOS of the p-i-n junction and the one with the 10nm drain-underlap region are shown.

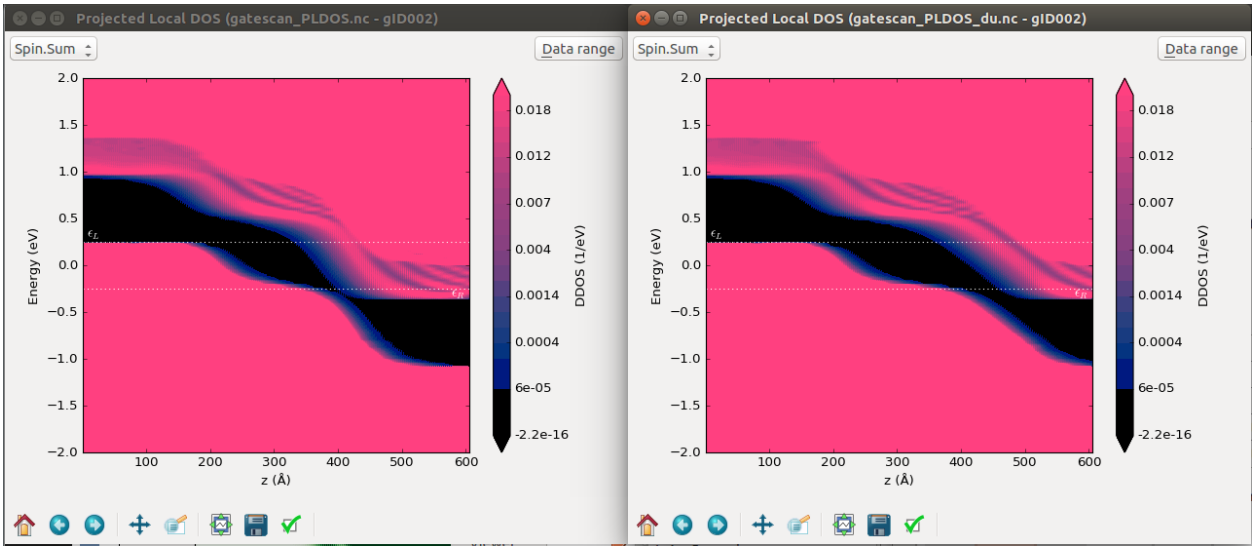


Fig. 3.41: Projected density of states across the 60 nm p-i-n junction device (left) and with 10 nm drain-underlap region (right). A -0.5 V source-drain bias voltage and a 0.4 V gate voltage are applied, respectively. The 0.4 V gate bias corresponds to the points around 0 V(GS) in the previous figure.

References

3.2.11 Inelastic current in a silicon p-n junction

Version: 2017.2

Inelastic scattering effects due to electron-phonon interactions can play a fundamental role in determining the carrier transport through a device. In this tutorial, you will investigate the impact of such effects on the current through a silicon p-n junction in reverse bias. In this system, carrier transport is dominated by band-to-band tunnelling between the valence band of the p region and the conduction band of the n region. This process can be enhanced considerably by inelastic scattering effects.

Specifically, you will use the *InelasticTransmissionSpectrum* module implemented in **ATK** to calculate the transmission spectrum of the device in the presence of electron-phonon interactions. You will also use the **Inelastic Transmission Spectrum Analyzer** plugin in **QuantumATK** to perform a detailed analysis of the current and determine which phonon mode is the main responsible for the current enhancement.

The methodology implemented in QuantumATK is based on the **Lowest Order Expansion (LOE)**[1] and the **eXtended LOE (XLOE)**[2] methods. In this tutorial, you will use the **XLOE** method, which treats the finite energy difference between initial and final states in the transition $\mathbf{k} \rightarrow \mathbf{k} \pm \mathbf{q}$ due to electron-phonon coupling in a more accurate manner. However, the tutorial can be carried out also using the simpler **LOE** method, which will give qualitatively similar results. You can find more details about the theory underlying the two methods in the *Notes* sections of the *InelasticTransmissionSpectrum* module in the **ATK manual**.

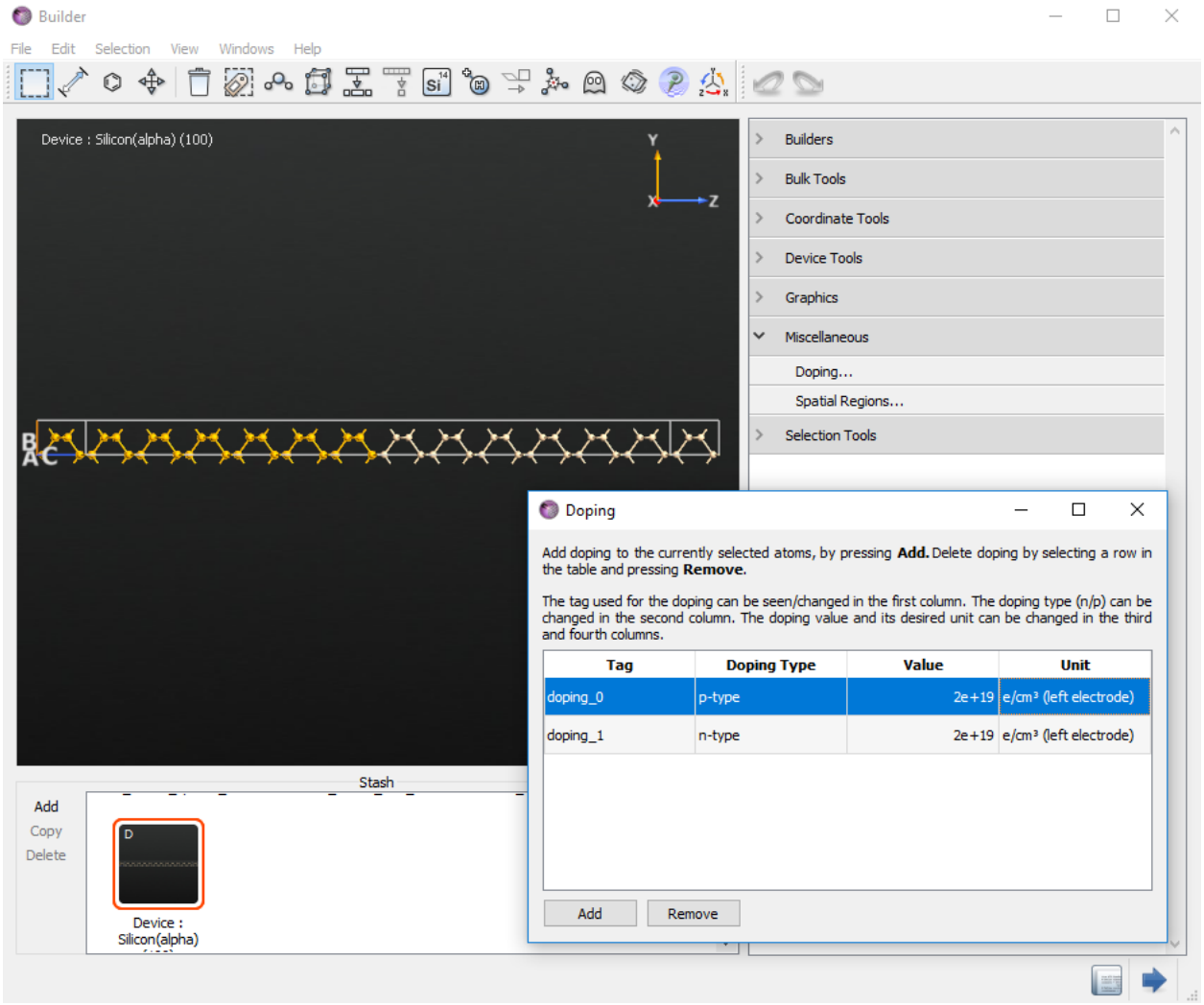
[1] T.ăFrederiksen, M.ăPaulsson, M.ăBrandbyge, and A.-P. Jauho. Inelastic transport theory from first principles: Methodology and application to nanoscale devices. *Phys. Rev. B*, 75:205413, May 2007. doi:10.1103/PhysRevB.75.205413.

[2] J.-T. Lu, R.ăB. Christensen, G.ăFoti, T.ăFrederiksen, T.ăGunst, and M.ăBrandbyge. Efficient calculation of inelastic vibration signals in electron transport: Beyond the wide-band approximation. *Phys. Rev. B*, 89:081405, Feb 2014. doi:10.1103/PhysRevB.89.081405.

Creating the silicon p-n junction

To create the device, you can follow the instruction in the *Create device* section of the *Silicon p-n junction* tutorial. However, in the present tutorial you will create a shorter device to speed up the calculations. In the **Builder**, contruct the device with the following changes:

- When using the **Surface (Cleave)** plugin to create a <100>-oriented structure, set the **thickness** to 24 layers instead of 52 layers;
- When using the **Device from Bulk** plugin, choose 5.4306 Å as the **electrode length**;
- Click the **Doping** plugin in *Miscellaneous* → *Doping* to set the doping of the p and n regions. In this case, select the half of device to set the p doping region and select the remainder part using the `ctrl+i` which will select the inversion part to set the n doping region. In both regions, set a doping of $2 \cdot 10^{19} \mathrm{e/cm^3}$.



You can download the resulting device configuration from here: `Si_pn_junction.py`.

Transmission calculation without electron-phonon interactions

In this section, you will calculate and analyze the electronic structure of the p-n junction at a reverse bias voltage of $V_{\mathrm{bias}} = -0.4$ V, and the associated transmission spectrum without electron-phonon interactions included, using the conventional Landauer formula as implemented in **ATK** in the *TransmissionSpectrum* module.

Setting up the calculation

From the **Builder**, send the structure to the **Script generator** using the button. Add the following blocks and set their parameters as follows:

- *Calculators* → *SemiEmpiricalCalculator*

–In **Main**:

∗Set the **Left electrode voltage** to -0.2 V

∗Set the **Right electrode voltage** to 0.2 V

–In **Hamiltonian**:

∗Untick the *No SCF iteration* option

–In **Numerical Accuracy**, set the **k-point Sampling** to:

∗$\mathrm{k_A} = 7$

∗$\mathrm{k_B} = 7$

∗$\mathrm{k_C} = 101$

–In **Iteration control parameters**, set the **Tolerance** to *4e-05*

- *Analysis* → *ProjectedLocalDensityOfStates*

–Set the **k-point Sampling** to:

∗$\mathrm{k_A} = 21$

∗$\mathrm{k_B} = 21$

- *Analysis* → *TransmissionSpectrum*

–Set the **k-point Sampling** to:

∗$\mathrm{k_A} = 21$

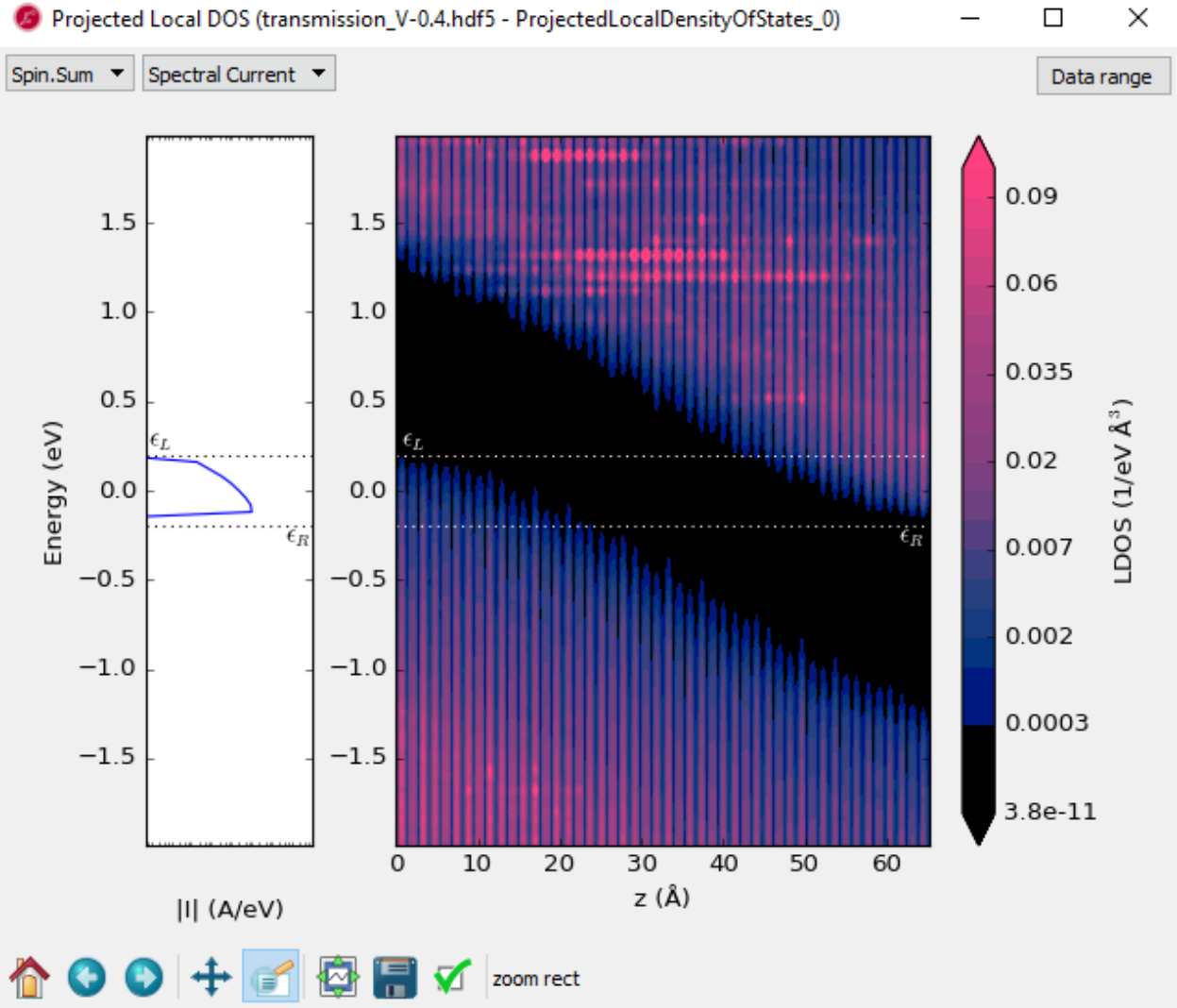
∗$\mathrm{k_B} = 21$

Finally, in the **Global IO** options, change the name of the **Default output file** to `transmission_V-0.4.hdf5`.

Send the script to the **Job manager** using the button, save it as `transmission_V-0.4.py` and press the button to run the calculation. It will take less than 2 minutes on 4 CPUs. You can also download the full script from here: `transmission_V-0.4.py`.

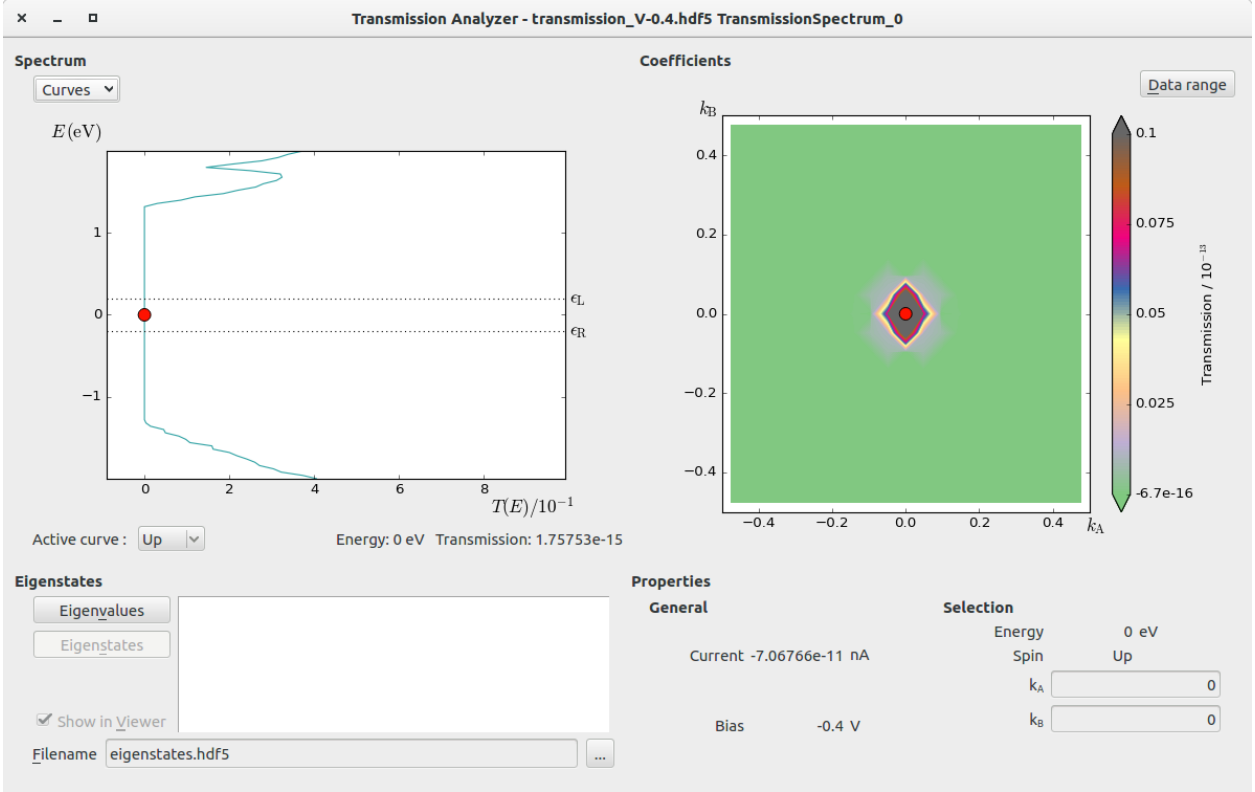
Analysis of the results

In the **LabFloor**, select the $D(\epsilon, z)$ **ProjectedLocalDensityOfStates** object from the file `transmission_V-0.4.hdf5`, and click on the **Projected Local Density of States** plugin. In the plugin window, select *Spectral Current* from the drop-down menu on the top left, and set the maximum value of the **Data range** to 0.1. In the panel on the left-hand side of the screen, use the **Zoom** button to make sure that the minimum value of the current is about $10^{-24}\mathrm{A/eV}$. The features below this value can be associated to numerical noise.



From the left panel of the figure above, it can be seen that the maximum of the spectral current occurs inside the bias window (-0.2 eV $\leq$ Energy ≤ 0.2 eV). Furthermore, from the right panel, which shows the density of states across the device, it can be seen that within the bias window, there are no electronic states in central region ($20 \leq$ z ≤ 45) of the device, so that tunnelling between the left and the right electrodes is the only possible transport mechanism. This indicates that electronic transport in the device in reverse bias is dominated by tunnelling.

To analyze the probability of tunnelling in **k**-space, select the $T(\mathbf{k}, \epsilon)$ **Transmission Spectrum** object from the same file and click on the **Transmission Analyzer** plugin. In the plugin window, set the maximum value of the **Data range** to 0.1.



From the figure above, it can be seen that the tunnelling probability is strongly peaked at the Γ-point of the Brilluoin zone at the Fermi level.

Transmission calculation with electron-phonon interactions

In this section, you will calculate the transmission spectrum of the silicon p-n junction at a reverse bias voltage of $V_{\text{bias}} = -0.4$ V including the effect of the electron-phonon interactions within the **XLOE** approximation[Page 614, 2].

To calculate the *InelasticTransmissionSpectrum*, you first need to calculate the **DynamicalMatrix** and the **HamiltonianDerivatives** of the system.

Setting up the dynamical matrix calculation

In the **LabFloor**, drag and drop the **DeviceConfiguration** object contained in the file `transmission_V-0.4.hdf5` to the **Script generator**. Remove the **DeviceSemiEmpiricalCalculator** and replace it with a **ForceFieldCalculator**

•Select *StillingerWeber_Si_1985* from the list of available classical potentials in the **Parameter set** list.

Add the following blocks and set their parameters as follows:

• *Study Objects → DynamicalMatrix*

–Set **Repetitions** to *Custom*

–Set the **number of repetitions** to:

$*n_A = 3$

$*n_B = 3$

$*n_C = 1$

–Untick the *Acoustic sum rule* option

–Tick the *Constrain electrodes* option

–Untick the *Equivalent bulk* option

• *Analysis → VibrationalMode*

Finally, in the **Global IO** options, change the name of the **Default output file** to `dynmat.hdf5`.

Send the script to the **Job manager** using the button, save it as `dynmat.py` and press the button to run the calculation. You can also download the full script from here: `dynmat.py`.

Note: Alternatively, you can also use the *special thermal displacement - Landauer method* (STD-Landauer) to include part of the electron-phonon coupling effects in the transmission calculation at a much lower computational cost. *The STD-Landauer case study* treats a system which is similar to that discussed in this tutorial.

Setting up the Hamiltonian derivatives calculation

In the **LabFloor**, drag and drop the **DeviceConfiguration** object contained in the file `transmission_V-0.4.hdf5` to the **Script generator**, and modify the **New Calculator** block as follows:

•In **Hamiltonian** tick the *No SCF iteration* option.

Add an *Study Objects → HamiltonianDerivatives* block and set the parameters as follows:

•Set **Repetitions** to *Custom*

•Set the **number of repetitions** to:

–$n_A = 3$

–$n_B = 3$

– $n_C = 1$

- In **Constraints**, click **Add** and constrain the electrode repetitions with *Fixed* constrains. The resulting list of constrained atoms should be: *0,1,2,3,44,45,46,47*.
- Untick the *Equivalent Bulk* option

In the **Global IO** options, change the name of the **Default output file** to `dHdR_V-0.4.hdf5`.

Send the script to the **Job manager** using the button, save it as `dHdR_V-0.4.py` and press the button to run the calculation. The calculation will take around 20 minutes on 8 CPUs. You can also download the full script from here: `dHdR_V-0.4.py`.

Note: Here, the Hamiltonian derivatives are calculated non self-consistently to speed up the calculations.

Calculating the inelastic transmission spectrum

To calculate the inelastic part of the current, click the **Script generator**, and add the following blocks:

- *Analysis from File*
- *Analysis → InelasticTransmissionSpectrum*

Open the *Analysis from File* block, select the `transmission_V-0.4.hdf5` file and load the *DeviceConfiguration* object contained in the file .

You will notice that two additional blocks have also been added:

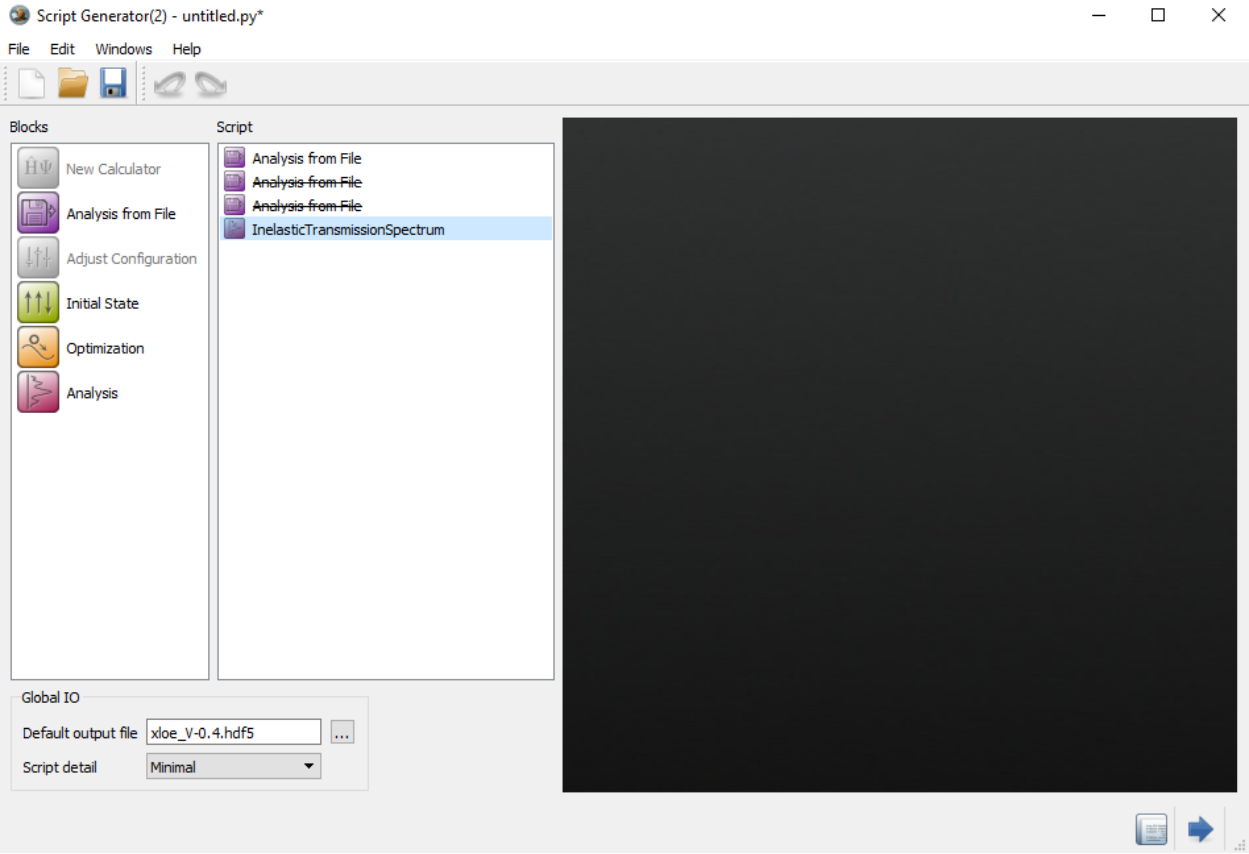
- *DynamicalMatrix*
- *HamiltonianDerivaties*

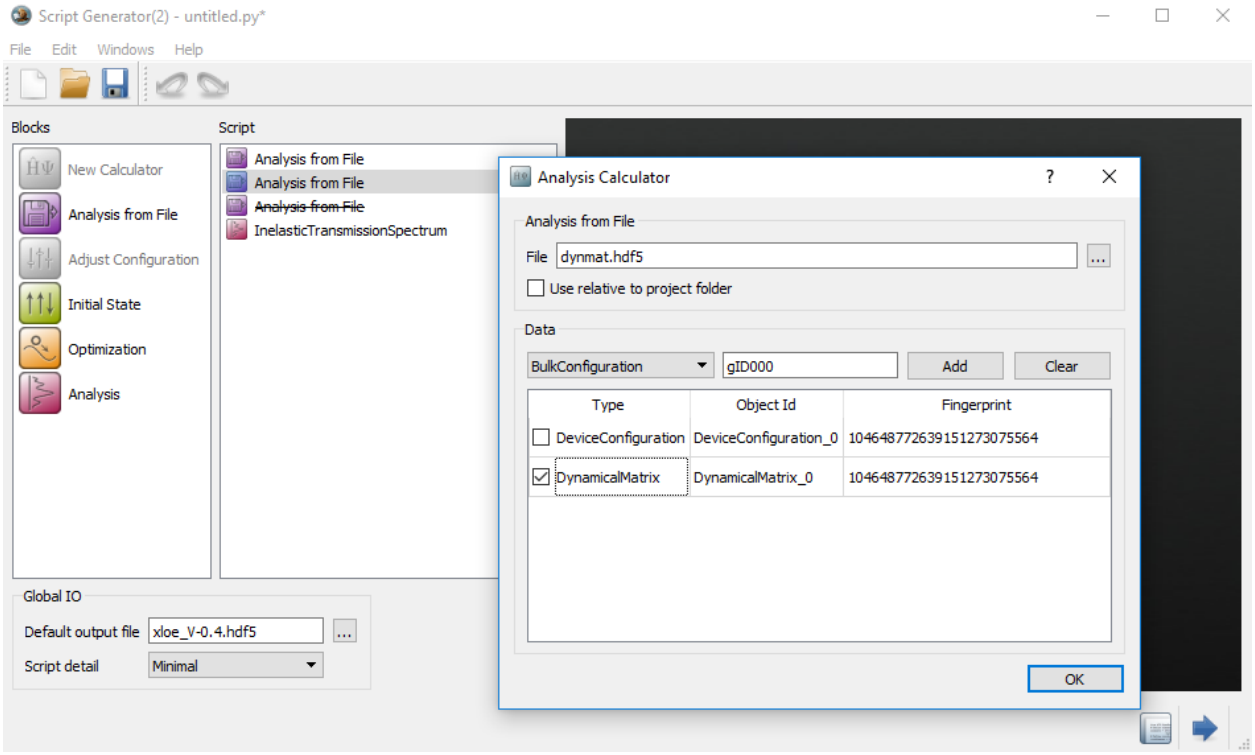
In our case, the dynamical matrix and the Hamiltonian derivatives have already been calculated previously and can be re-used. Delete the two blocks, add two more *Analysis from File* blocks and arrange them as shown in the figure below:

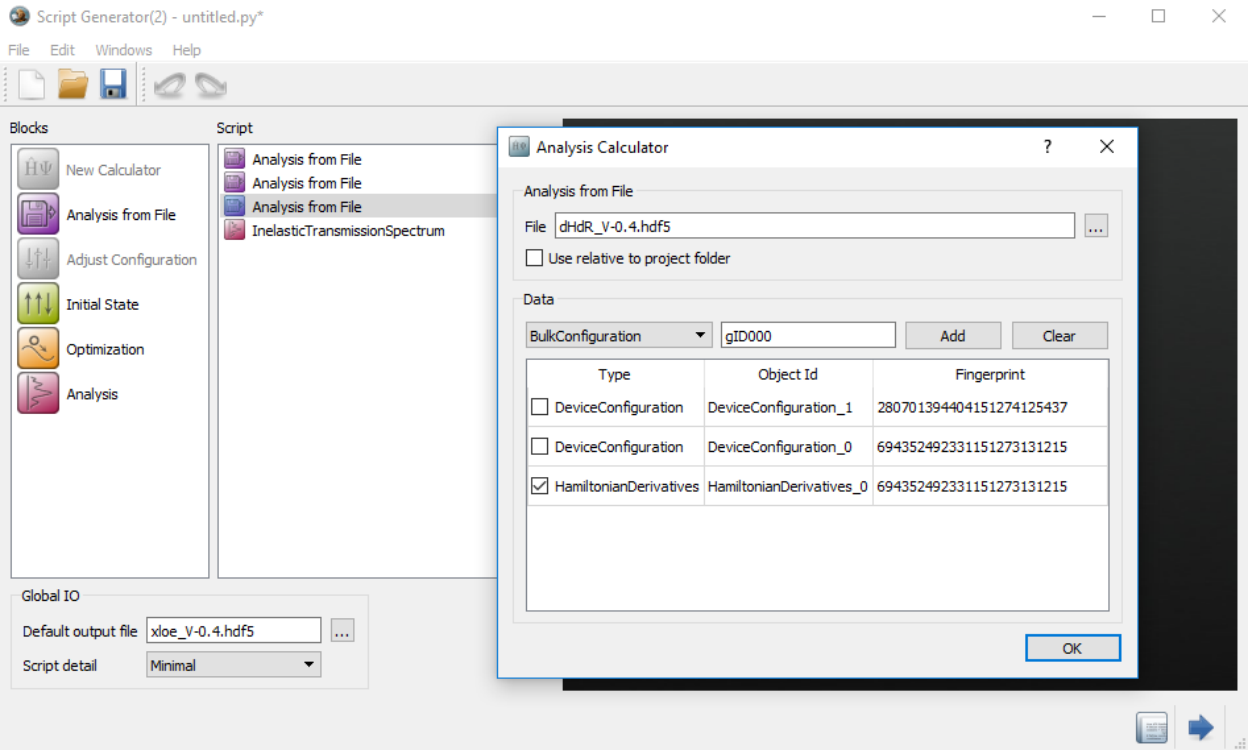
Click on the second *Analysis from File* block from the top. Select the file `dynmat.hdf5` and load the *DynamicalMatrix* object. Uncheck the *DeviceConfiguration* object included in the same file. The panel should look like as in the figure below:

Then, click on the third *Analysis from File* block from the top. Select the file `dHdR_V-0.4.hdf5` and load the *HamiltonianDerivatives* object. The panel should look like as in the figure below:

Finally, set the parameters for the *Analysis → InelasticTransmissionSpectrum* block as shown below:


Script Generator(2) - untitled.py*
File Edit Windows Help
Blocks
New Calculator
Analysis from File
Adjust Configuration
Initial State
Optimization
Analysis
Script
Analysis from File
Analysis from File
Analysis from File
InelasticTransmissionSpectrum
Global IO
Default output file
xloe_V-0.4.hdf5
Script detail
Minimal


Script Generator(2) - untitled.py*
File Edit Windows Help
Blocks
New Calculator
Analysis from File
Adjust Configuration
Initial State
Optimization
Analysis
Script
Analysis from File
Analysis from File
Analysis from File
InelasticTransmissionSpectrum
Global IO
Default output file
xloe_V-0.4.hdf5
Script detail
Minimal
Analysis Calculator
Analysis from File
File
dynmat.hdf5
Use relative to project folder
Data
BulkConfiguration
gID000
Add
Clear
Type
Object Id
Fingerprint
DeviceConfiguration
DeviceConfiguration_0
104648772639151273075564
DynamicalMatrix
DynamicalMatrix_0
104648772639151273075564
OK



•In the **Energy** part, set:

–$E_0 = -0.5$ eV

–$E_1 = 0.5$ eV

–Points = 25

•In the **k-point Sampling** part:

–set **Grid type** to *Regular k-point grid*

–set the grid range for k_A and k_B to $[-0.2 : 0.2]$

–set the **Sampling** to:

∗$k_A = 3$

∗$k_B = 3$

•In the **q-point Sampling** part:

–set **Grid type** to *Regular q-point grid*

–set the grid range for q_A and q_B to $[-0.5 : 0.5]$

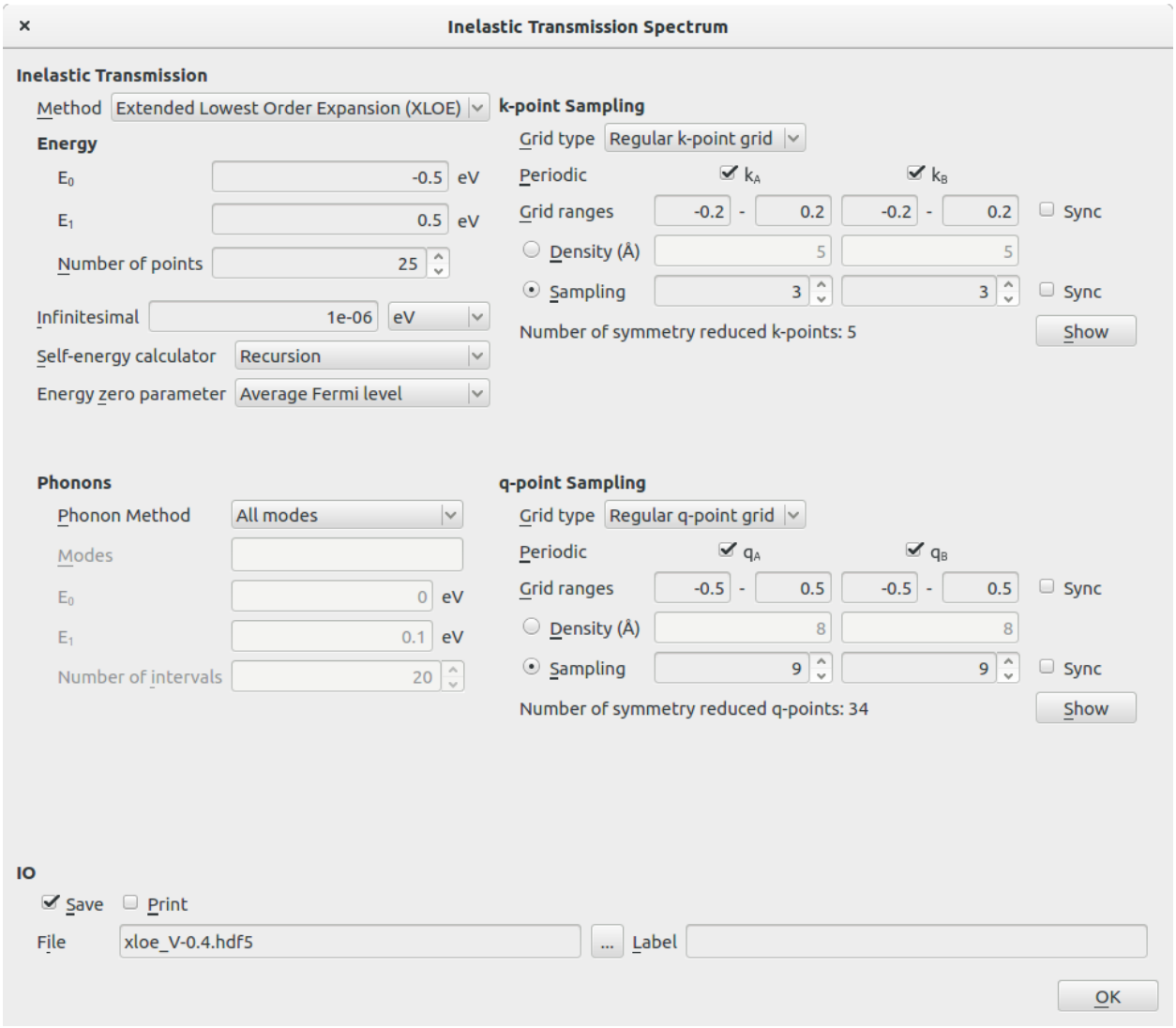
–set the **Sampling** to:

∗$q_A = 9$

∗$q_B = 9$

In the **Global IO** options, change the name of the **Default output file** to `xloe_V-0.4.hdf5`.

Send the script to the **Job manager** using the button, save it as `xloe_V-0.4.py` and press the button to run the calculation. The calculation will take around 1.5 hour on 24 CPUs. You can also download the full script from


Inelastic Transmission Spectrum
Inelastic Transmission
Method
Extended Lowest Order Expansion (XLOE)
Energy
E0
-0.5
eV
E1
0.5
eV
Number of points
25
Infinitesimal
1e-06
eV
Self-energy calculator
Recursion
Energy zero parameter
Average Fermi level
k-point Sampling
Grid type
Regular k-point grid
Periodic
kA
kB
Grid ranges
-0.2
0.2
-0.2
0.2
Sync
Density (Å)
5
5
Sampling
3
3
Sync
Number of symmetry reduced k-points: 5
Show
Phonons
Phonon Method
All modes
Modes
E0
0
eV
E1
0.1
eV
Number of intervals
20
q-point Sampling
Grid type
Regular q-point grid
Periodic
qA
qB
Grid ranges
-0.5
0.5
-0.5
0.5
Sync
Density (Å)
8
8
Sampling
9
9
Sync
Number of symmetry reduced q-points: 34
Show
IO
Save
Print
File
xloe_V-0.4.hdf5
...
Label
OK

here: `xloe_V-0.4.py`.

Analysis of the inelastic transmission spectrum

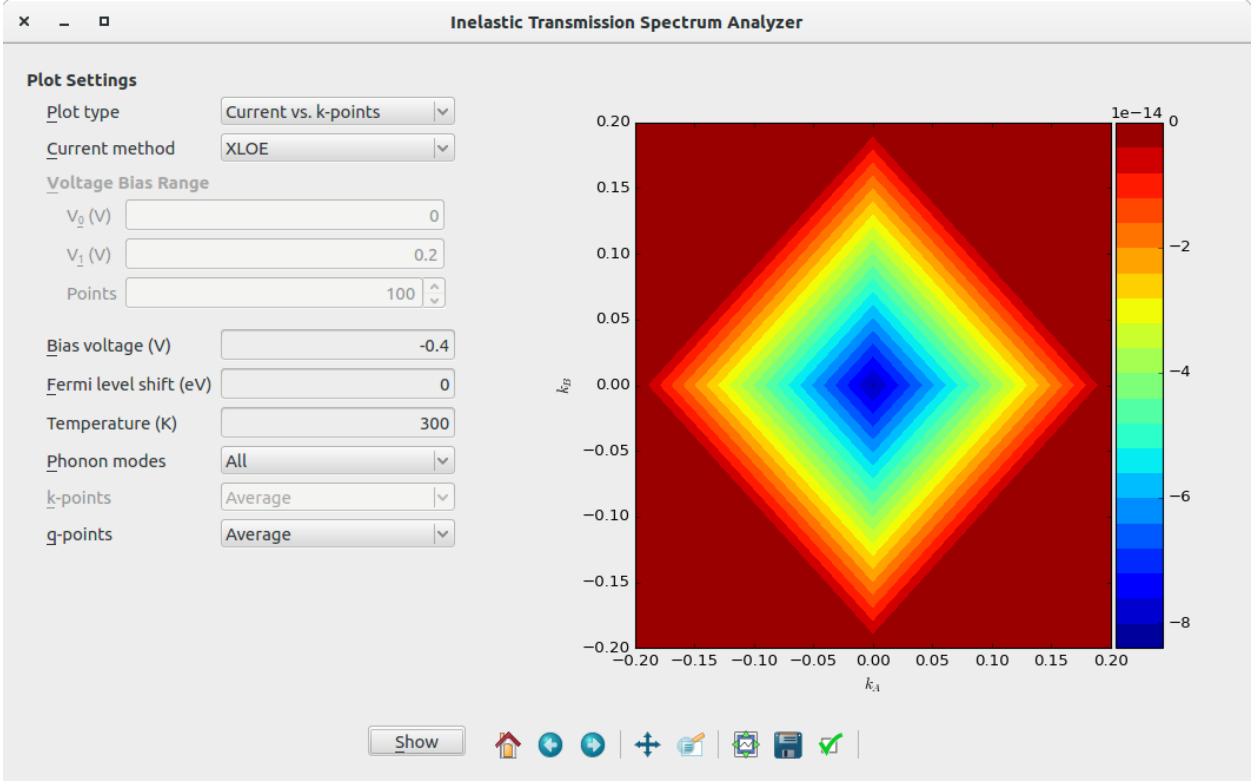
The results of the inelastic transmission spectrum can be analyzed in detail using the **Inelastic Transmission Spectrum Analyzer**.

In the **LabFloor**, select the $T(\mathbf{k},\mathbf{q})$ **InelasticTransmissionSpectrum** object contained in the file `xloe_V-0.4.hdf5`, and click on the **Inelastic Transmission Spectrum Analyzer** plugin in the plugins panel on the right-hand side of the screen.

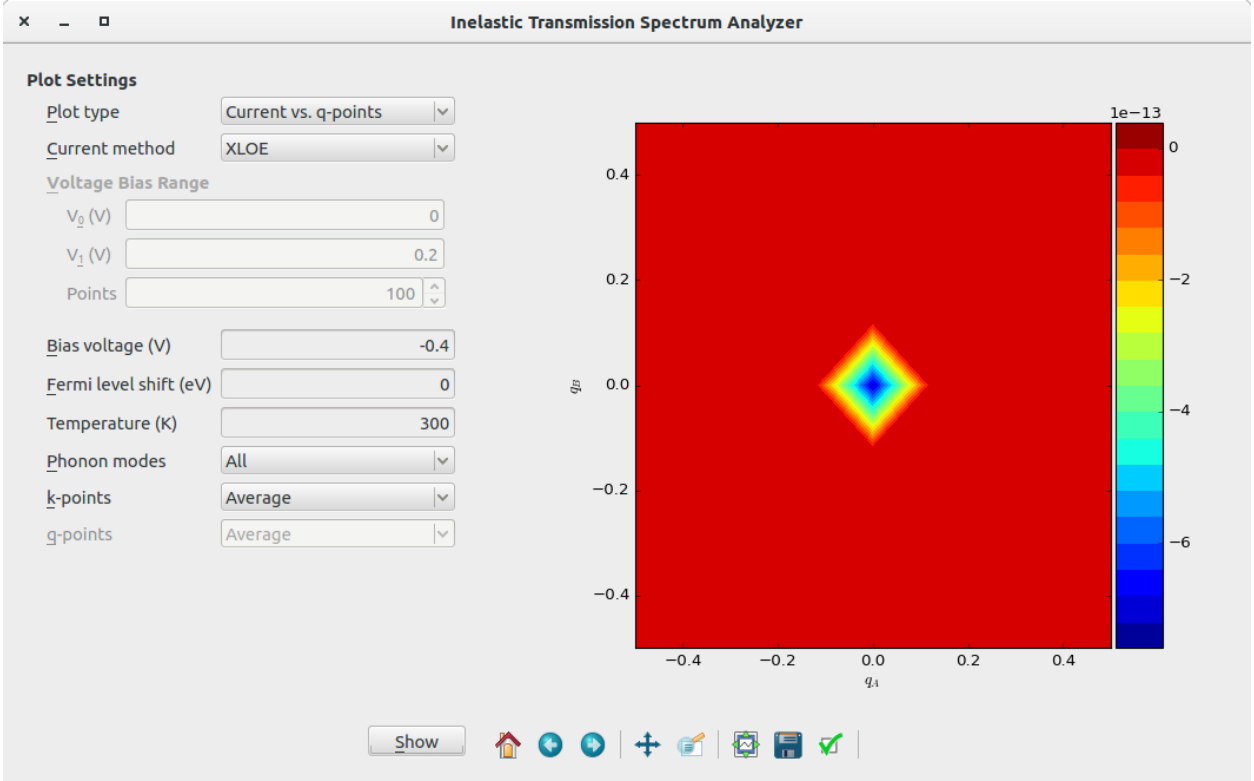
First of all, you will analyze the $\mathbf{k}$- and $\mathbf{q}$-dependency of the current. In the main window of the analyzer, set the following parameters:

- **Plot type** : *Current vs. k-points*
- **Bias voltage** : -0.4 V

You will obtain a figure similar to the one below, showing the $\mathbf{k}$-dependency of the current within the sampled range of k-points. From the figure, it is evident that the main contribution to the current comes from the Γ-point of the two-dimensional Brillouin zone defined by k_A and k_B, with a non-negligible contribution from finite $\mathbf{k}$-vectors.



Next, change the **Plot type** to *Current vs. q-points*. You will obtain a figure similar to the one below, showing the $\mathbf{q}$-dependency of the current. From the figure, it is evident that also in this case the main contribution to the current comes from the Γ-point of the two-dimensional Brillouin zone defined by q_A and q_B, , with a non-negligible contribution from finite $\mathbf{q}$-vectors.



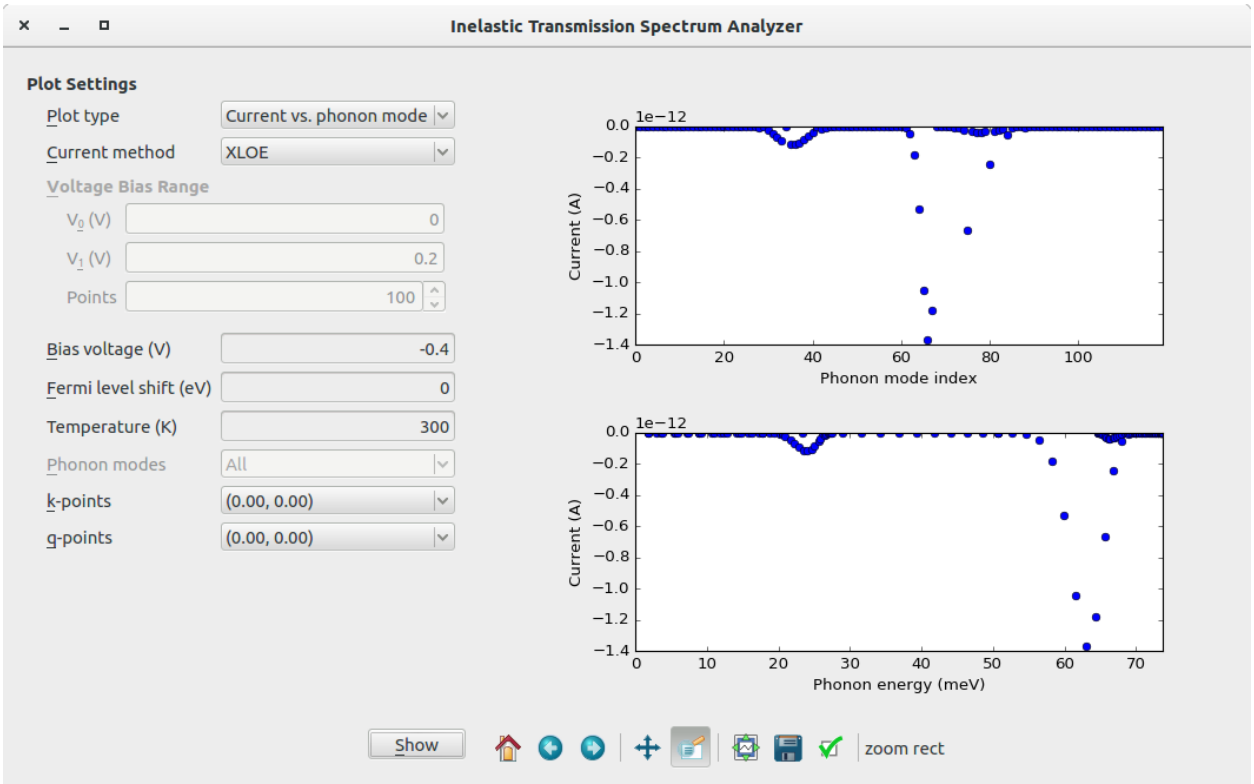
The two figures above indicate that the current is mainly associated with transitions occurring at the Γ-point of the two-dimensional Brillouin zone, both in **k**-space and in **q**-space. To analyze which of the phonon modes at the Γ-point is the one that contributes the most, set the following parameters in the main window of the analyzer:

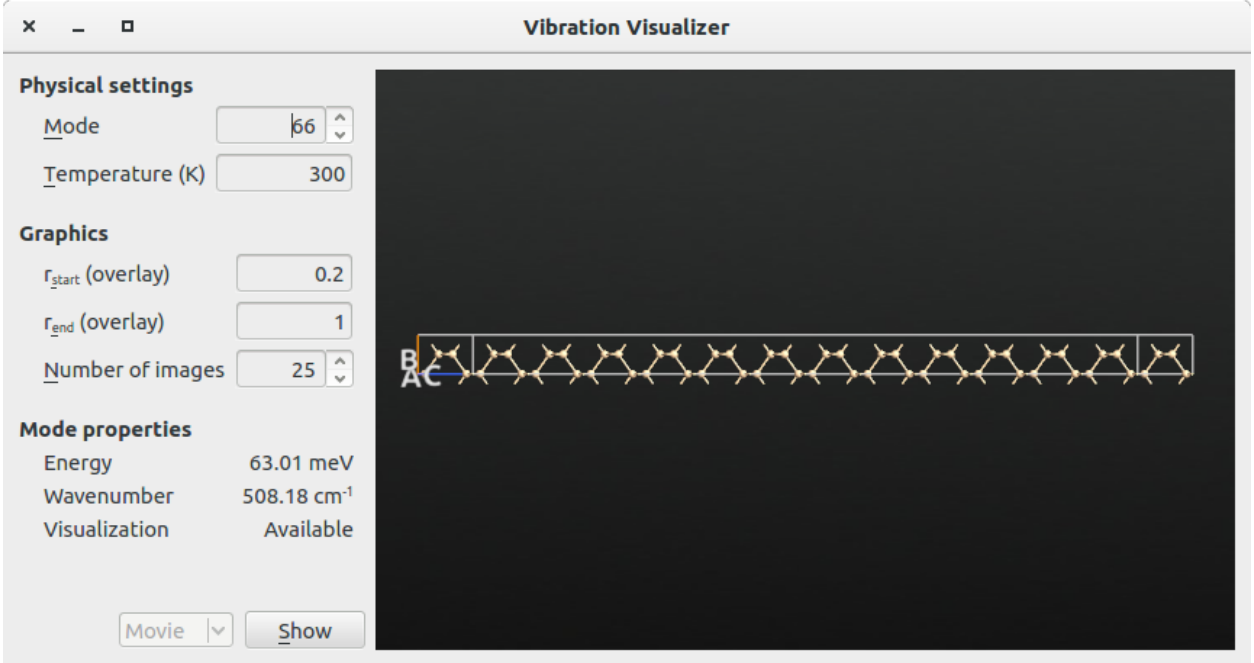
- **Plot type** : *Current vs. phonon mode*
- **k-points** : *(0.00, 0.00)*
- **q-points** : *(0.00, 0.00)*

From the figure, it can be seen that the phonon that contributes the most to the current is that with index 66, and energy $\hbar\omega = 63.01$ meV.

This phonon mode can be visualized by selecting the **VibrationalMode** object in the file `dynmat.hdf5` calculated in Section *Setting up the dynamical matrix calculation* and using the **Vibration Visualizer** plugin:

Note: In the figures presented above, the values of the current are negative, because the simulations are performed in reverse bias conditions.


Inelastic Transmission Spectrum Analyzer
Plot Settings
Plot type
Current vs. phonon mode
Current method
XLOE
Voltage Bias Range
V0 (V)
0
V1 (V)
0.2
Points
100
Bias voltage (V)
-0.4
Fermi level shift (eV)
0
Temperature (K)
300
Phonon modes
All
k-points
(0.00, 0.00)
q-points
(0.00, 0.00)
1e−12
Current (A)
Phonon mode index
1e−12
Current (A)
Phonon energy (meV)
Show
zoom rect


Vibration Visualizer
Physical settings
Mode
66
Temperature (K)
300
Graphics
rstart (overlay)
0.2
rend (overlay)
1
Number of images
25
Mode properties
Energy
63.01 meV
Wavenumber
508.18 cm-1
Visualization
Available
Movie
Show

Calculation of the current with and without electron-phonon interactions

In order to calculate the current without and with electron-phonon interactions, download the script `current.py` and run it from the terminal as:

`atkpython current.py transmission_V-0.4.hdf5 xloe_V-0.4.hdf5`.

Note: On windows, it is not convenient to run `current.py` script from the terminal. In this script, you can replace filename1 and filename2 to the `transmission_V-0.4.hdf5` and `xloe-V-0.4.hdf5`. In order to run it, drag it on the **Job manager**.

The script will calculate the elastic (I_{el}) and inelastic (I_{inel}) components of the current. The total current without and with electron-phonon interactions will then be calculated as:

$$I(\mathrm{w/o\ interactions}) = I_{\mathrm{el}}$$
$$I(\mathrm{w\ interactions}) = I_{\mathrm{el}} + I_{\mathrm{inel}}$$

The output of the calculation will show the name of the files used as input, as well as the values of the calculated currents (highlighted in yellow):

```
+-------------------------------------------------------------------------------+
|                                                                               |
| QuantumATK 2017.2 [Build 997d195]                                  |
|                                                                               |
+-------------------------------------------------------------------------------+
-------------------------------------------------------------------------------
 TransmissionSpectrum read from file :
 transmission_V-0.4.hdf5
 InelasticTransmissionSpectrum read from file :
 xloe_V-0.4.hdf5
-------------------------------------------------------------------------------
 Current (w/o interactions)    = -3.71e-09 nA
 Current (w interactions)      = -9.03e-06 nA
-------------------------------------------------------------------------------

Timing:                          Total     Per Step        %

-------------------------------------------------------------------------------

Loading Modules + MPI   :       1.33 s       1.33 s       2.58% |==========|
-------------------------------------------------------------------------------
Total                   :      51.58 s
```

Note: The above calculated currents are different with the *currents* before analysis of the InelasticTransmissionSpectrum because of different k-points samplings. Also in this case, the value of the current is negative, because the simulations are performed in reverse bias conditions.

It can be seen that electron-phonon scattering leads to an increase in the reverse bias current of about three orders of magnitude!

Speeding up the calculations

ATK implements several methods which can be used to speed up the calculations considerably and enable the calculation of the inelastic transmission for devices comprised of thousands of atoms. These methods are:

- *The Phonon energy intervals method*
- The possibility of *using the bulk dynamical matrix and Hamiltonian derivatives*

For a more detailed description of these methods, see the section *Large Device Calculations* of the *InelasticTransmissionSpectrum* module in the **ATK manual**.

The Phonon energy intervals method

This method allows us to group $3N$ phonon modes of a device with N vibrating atoms into M energy intervals to form M new effective phonon modes, with $M << 3N$. The inelastic transmission spectrum will therefore be calculated only for these M effective phonon modes, greatly reducing the computational cost of the calculation.

In the **LabFloor**, drag and drop the **DeviceConfiguration** object contained in the file `transmission_V-0.4.hdf5` to the **Script generator**. Setup the calculation of the inelastic transmission spectrum as in the *Calculating the inelastic transmission spectrum* Section, but in this case set the **Phonon Method** in the **Phonons** part to *Phonon energy intervals*.

Note: The default parameters of the *Phonon energy intervals* are fine for the present calculation, but in general one should ensure that the energy range $[E_0 : E_1]$ spans that of the phonon modes of the calculated dynamical matrix.

In the **Global IO** options, change the name of the **Default output file** to `xloe_pheint_V-0.4.hdf5`.

Send the script to the **Job manager** using the button, save it as `xloe_pheint_V-0.4.py` and press the button to run the calculation. The calculation will take only 20 minutes on 24 CPUs. You can also download the full script from here: `xloe_pheint_V-0.4.py`.

Once the calculation is finished, run the script `current.py` as:

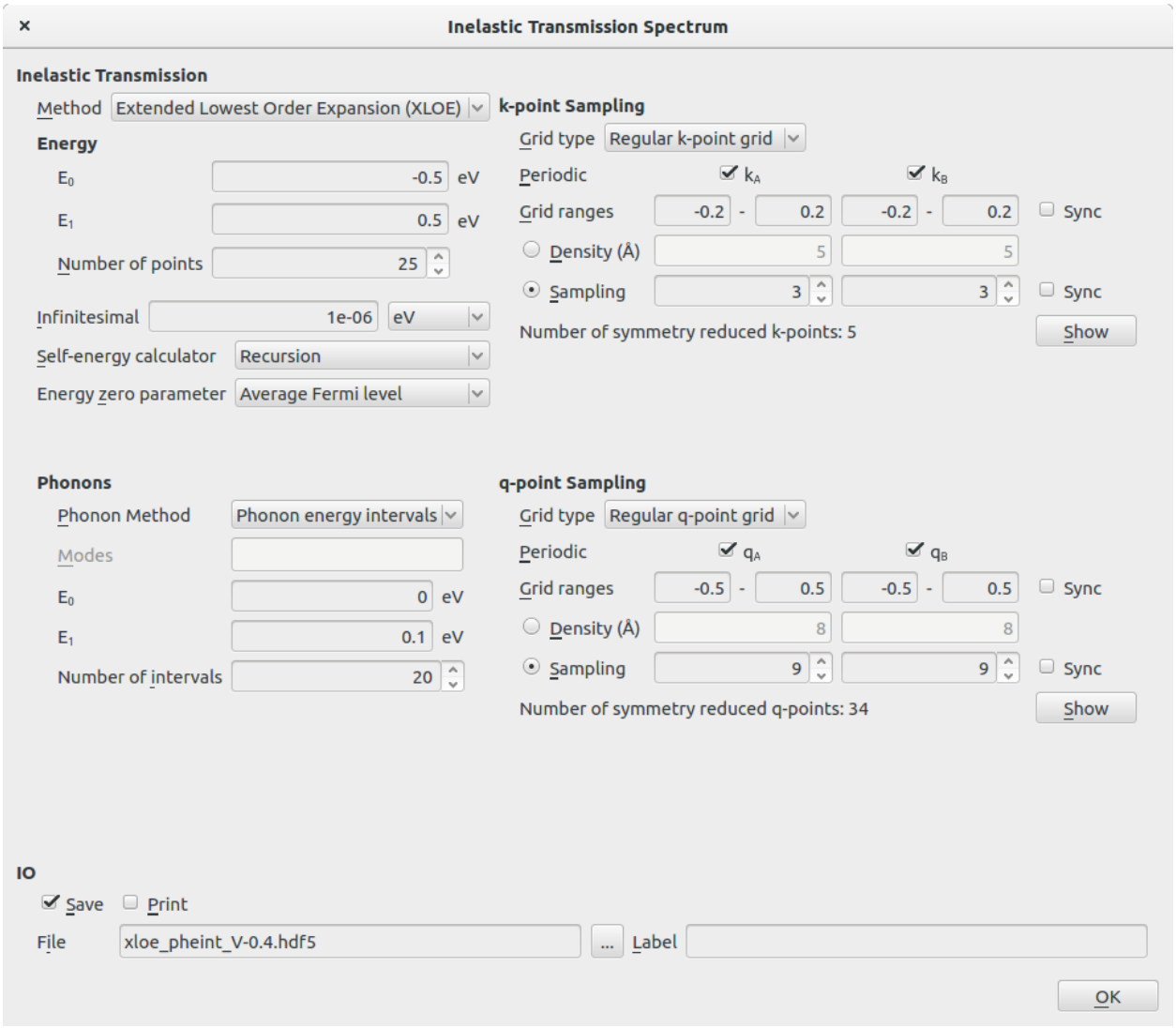
`atkpython current.py transmission_V-0.4.hdf5 xloe_pheint_V-0.4.hdf5`.

If you are a Windows user, see *how to run the script*.

The resulting output will be:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| QuantumATK 2017.2 [Build 997d195]                                  |
|                                                                               |
+-------------------------------------------------------------------------------+
-------------------------------------------------------------------------------
 TransmissionSpectrum read from file :
 transmission_V-0.4.hdf5
 InelasticTransmissionSpectrum read from file :
 xloe_pheint_V-0.4.hdf5
-------------------------------------------------------------------------------
 Current (w/o interactions)    = -3.71e-09 nA
```

(continues on next page)


Inelastic Transmission Spectrum
Inelastic Transmission
Method
Extended Lowest Order Expansion (XLOE)
Energy
E0
-0.5
eV
E1
0.5
eV
Number of points
25
Infinitesimal
1e-06
eV
Self-energy calculator
Recursion
Energy zero parameter
Average Fermi level
k-point Sampling
Grid type
Regular k-point grid
Periodic
kA
kB
Grid ranges
-0.2
0.2
-0.2
0.2
Sync
Density (Å)
5
5
Sampling
3
3
Sync
Number of symmetry reduced k-points: 5
Show
Phonons
Phonon Method
Phonon energy intervals
Modes
E0
0
eV
E1
0.1
eV
Number of intervals
20
q-point Sampling
Grid type
Regular q-point grid
Periodic
qA
qB
Grid ranges
-0.5
0.5
-0.5
0.5
Sync
Density (Å)
8
8
Sampling
9
9
Sync
Number of symmetry reduced q-points: 34
Show
IO
Save
Print
File
xloe_pheint_V-0.4.hdf5
...
Label
OK

(continued from previous page)

```
 Current (w interactions)      = -9.25e-06 nA
-------------------------------------------------------------------------------

Timing:                          Total     Per Step        %

-------------------------------------------------------------------------------

Loading Modules + MPI   :       1.34 s       1.34 s      19.27% |==========|
-------------------------------------------------------------------------------
Total                   :       6.94 s
```

One can see that the calculated values for the current are essentially the same as those calculated in the Section *Calculation of the current with and without electron-phonon interactions*, despite the fact that the present calculations are much faster.

Using the bulk dynamical matrix and Hamiltonian derivatives

Another option to speed up the calculations is to use the dynamical matrix and Hamiltonian derivatives of the bulk electrodes, instead of those of the device. However, this is possible only for devices with a structure which is translationally invariant along the C-direction, apart from doping, electrostatic regions and applied bias voltage.

The scripts needed to calculate the dynamical matrix and the Hamiltonian derivatives of the bulk electrodes can be downloaded from here: `dynmat_bulk.py`, `dHdR_bulk.py`. Running the two calculations on 4 CPUs will take less than 2 minutes.

In order to calculate the current inelastic transmission spectrum using the bulk dynamical matrix and Hamiltonian derivatives, repeat the steps followed in Section *Calculating the inelastic transmission spectrum*, with the changes described in the following.

First, click the **Script generator**, add the following blocks, and modify their parameters as follows:

- *Analysis from File*

–Load the *DeviceConfiguration* object from the `tranmission_V-0.4.hdf5` file.

- *Analysis → InelasticTransmissionSpectrum*

–Set parameters as described in the *Inelastic transmission spectrum* section.

Remove the *DynamicalMatrix* and *HamiltonianDerivatives* blocks, add two more *Analysis from File* blocks right after the *Analysis from File* already present, and set their properties as follows:

•Click on the second *Analysis from File* block from the top, select the `dynamic_bulk.hdf5` file and load the *DynamicalMatrix* object contained in the file.

•Click on the second *Analysis from File* block from the top, select the `dHdR_bulk.hdf5` file and load the *HamiltonianDerivatives* object contained in the file.

In the **Global IO** options, set the name of the **Default output file** to `xloe_V-0.4_bulk.hdf5`. In this case, you need to modify the **Inelastic Transmission Spectrum** block in the **Editor** as follows:

```
inelastic_transmission_spectrum = InelasticTransmissionSpectrum(
    configuration=device_configuration,
    bulk_dynamical_matrix=dynamical_matrix,
    bulk_hamiltonian_derivatives=hamiltonian_derivatives,
    energies=numpy.linspace(-0.5, 0.5, 25)*eV,
    kpoints=kpoints,
    qpoints=qpoints,
    self_energy_calculator=RecursionSelfEnergy(),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    phonon_modes=All,
    method=XLOE,
    spectral_representation=True,
    electrode_extensions=[0, 0],
```

Send the script to the **Job manager** using the button, save it as `xloe_V-0.4_bulk.py` and press the button to run the calculation. Also in this case, the calculation will take only 20 minutes on 24 CPUs. You can also download the full script from here: `xloe_V-0.4_bulk.py`.

Once the calculation is finished, run the script `current.py` as:

`atkpython current.py transmission_V-0.4.hdf5 xloe_V-0.4_bulk.hdf5`.

If you use Windows, look at the *how to run the script*.

The resulting output will be:

```
+------------------------------------------------------------------------------+
|                                                                              |
| QuantumATK 2017.2 [Build 997d195]                                  |
|                                                                              |
+------------------------------------------------------------------------------+
------------------------------------------------------------------------------
 TransmissionSpectrum read from file :
 transmission_V-0.4.hdf5
 InelasticTransmissionSpectrum read from file :
 xloe_V-0.4_bulk.hdf5
------------------------------------------------------------------------------
 Current (w/o interactions)    = -3.71e-09 nA
 Current (w interactions)      = -6.42e-06 nA
------------------------------------------------------------------------------

Timing:                          Total     Per Step        %

------------------------------------------------------------------------------

Loading Modules + MPI   :       1.33 s       1.33 s       2.17% |==========|
------------------------------------------------------------------------------
Total                   :      61.15 s (1m01.15s)
```

It can be seen that the calculated values for the current is similar to that calculated using the dynamical matrix and Hamiltonian derivatives of the device configuration, although there are differences due to the fact that in the present case, the loss translational invariance of the Hamiltonian derivatives due to the applied bias is not taken into account.

References

3.2.12 Elastic scattering, mean free path, mobility: Impurity scattering in a silicon nanowire

Version: 2017.0

Downloads & Links

`sinw_pristine_trans.hdf5` `sinw_P_dopant_trans_1234.hdf5` `T_and_mfp.py` `doped-bandstructures.zip` `mobility.py`

In this tutorial you will study elastic scattering by dopant atoms in a silicon nanowire. Based on calculated transmission spectra you will learn how to estimate the scattering mean free path and mobility, taking into account both the effect of doping and the explicit scattering due to the dopant atoms.

Introduction

In this tutorial you will learn how the transmission spectra can be used to estimate the elastic scattering mean free path and the doping dependent mobility.

The main focus of the tutorial is not on the actual calculations of the transmission spectra, but rather on the subsequent analysis. As some of the calculations are rather time-consuming, data-files with the results are provided as downloads, so you can focus on the analysis.

The methods used in the tutorial are based on Ref.[1], where it was shown that the transmission function of a very long silicon nanowires with several structural defects can be accurately reproduced using a statistical average of the transmissions calculated for relatively small nanowires with a single defect.

Theoretical background

At low temperatures, the conductance of a system can be approximated as $G = G_0 \cdot T(E_F)$, where $G_0 = 2e^2/h$ is the conductance quantum and $T(E_F)$ is the transmission at the Fermi energy. The resistance of the system is $R = 1/G$.

These considerations can be generalized to define an energy dependent and unitless resistance $R(E) = 1/T(E)$, which is just the inverse of the transmission. To get a physical resistance, $R(E)$ should be multiplied with $1/G_0$.

For a one-dimensional system without any defects (pristine), the transmission always takes integer values, $T_0(E) = N(E)$. We shall denote the corresponding resistance $R_c(E) = 1/T_0(E)$ as the contact resistance. If a structural defect (labeled as *1* in the following) is introduced to the system, the transmission will always be lower than the transmission of the pristine system, $T_1(E) < T_0(E)$. We will now write the resistance of the system with the defect as a sum of two contributions:

$$\frac{1}{T_1(E)} = R_1(E) = R_c(E) + R_{1,s}(E),$$

[1] T.ăMarkussen, R.ăRurali, A.-P. Jauho, and M.ăBrandbyge. Scaling Theory Put into Practice: First-Principles Modeling of Transport in Doped Silicon Nanowires. *Phys. Rev. Lett.*, 99:076803, Aug 2007. doi:10.1103/PhysRevLett.99.076803.

which defines the scattering resistance $R_{1,s}(E)$ of defect *1*. If multiple different defects are present – this could be dopant atoms in different positions – we can calculate an average transmission $\langle T(E)\rangle$. Using the same notation as in the equation above, we will define an average scattering resistance, $\langle R_s(E)\rangle$, as:

$$\frac{1}{\langle T(E)\rangle} = \langle R(E)\rangle = R_c(E) + \langle R_s(E)\rangle,$$
$$\langle R_s(E)\rangle = \langle R(E)\rangle - R_c(E) = \frac{1}{\langle T(E)\rangle} - \frac{1}{T_0(E)}$$

It was shown in Ref.[Page 632, 1] that the length- and energy-dependent resistance through a long wire of length L containing many, randomly placed defects with an average defect–defect distance d, can be well approximated as:

$$R(L,E) = R_c(E) + \langle R_s(E)\rangle \frac{L}{d},$$

which may be written as

$$R(L,E) = R_c(E)\left(1 + \frac{L}{L_{\mathrm{mfp}}(E)}\right),$$

which defines the energy-dependent scattering mean free path as

$$L_{\mathrm{mfp}}(E) = d\frac{R_c(E)}{\langle R_s(E)\rangle}.$$

Note: Note that we have now introduced the total length of the wire L, which in this formalism is an arbitrary parameter.

The corresponding transmission through the long wire is $T(L,E) = 1/R(L,E)$. This means that the conductance of a wire with length L can be estimated as

$$G(L,E_F) = \int_{-\infty}^{\infty} T(E,L)\frac{-\partial f(E,E_F)}{\partial E}dE,$$

which generalizes the previous relation $G = G_0T(E_F)$ to finite temperatures. The function $f(E,E_F)$ is the Fermi-Dirac distribution function for electrons in the left and right electrodes with average Fermi level E_F.

The conductivity σ of the system is given by

$$\sigma = G \cdot L,$$

and we can now estimate the mobility μ as

$$\mu = \frac{\sigma}{e \cdot n},$$

where e is the elementary charge and n is the charge density. The Fermi level at a given doping density will be estimated from band structure calculations with compensation charge doping.

Summary of required calculations

In summary, the calculations we need in order to calculate a mean free path are:

- Transmission spectrum, $T_0(E)$, of the pristine (defect free) nanowire.
- A transmission spectrum for each defect – in this case for the same element in four different dopant positions. From these calculations we get the average transmission $\langle T(E)\rangle$.

In order to calculate the mobility at a given doping density, we further need:

- A bandstructure calculation (also using the nanowire unit cell) with compensation charge doping, in order to calculate the Fermi level for the desired doping density.

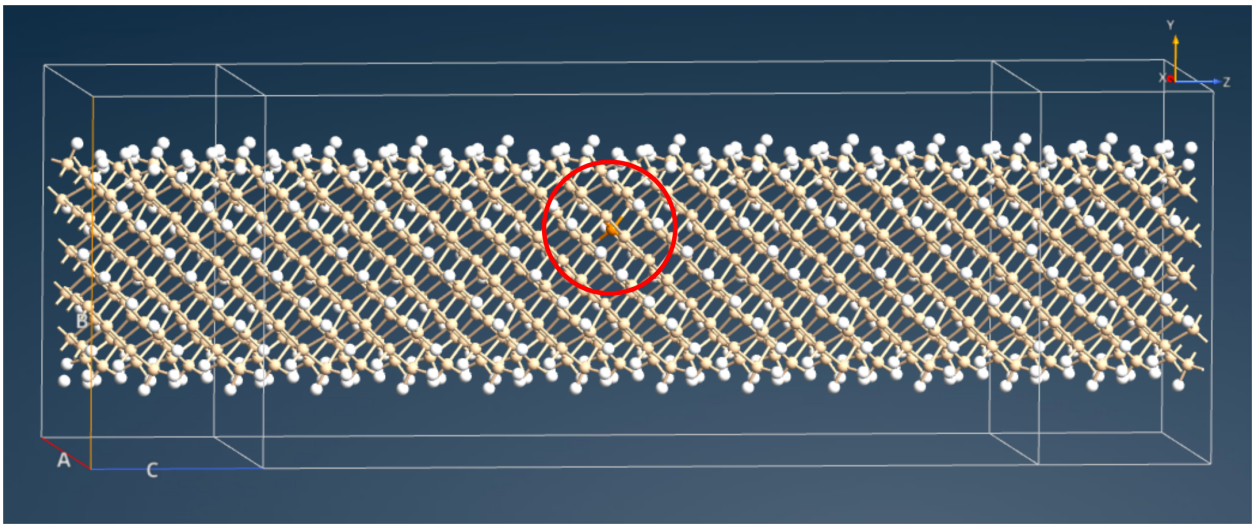
Defected silicon nanowires

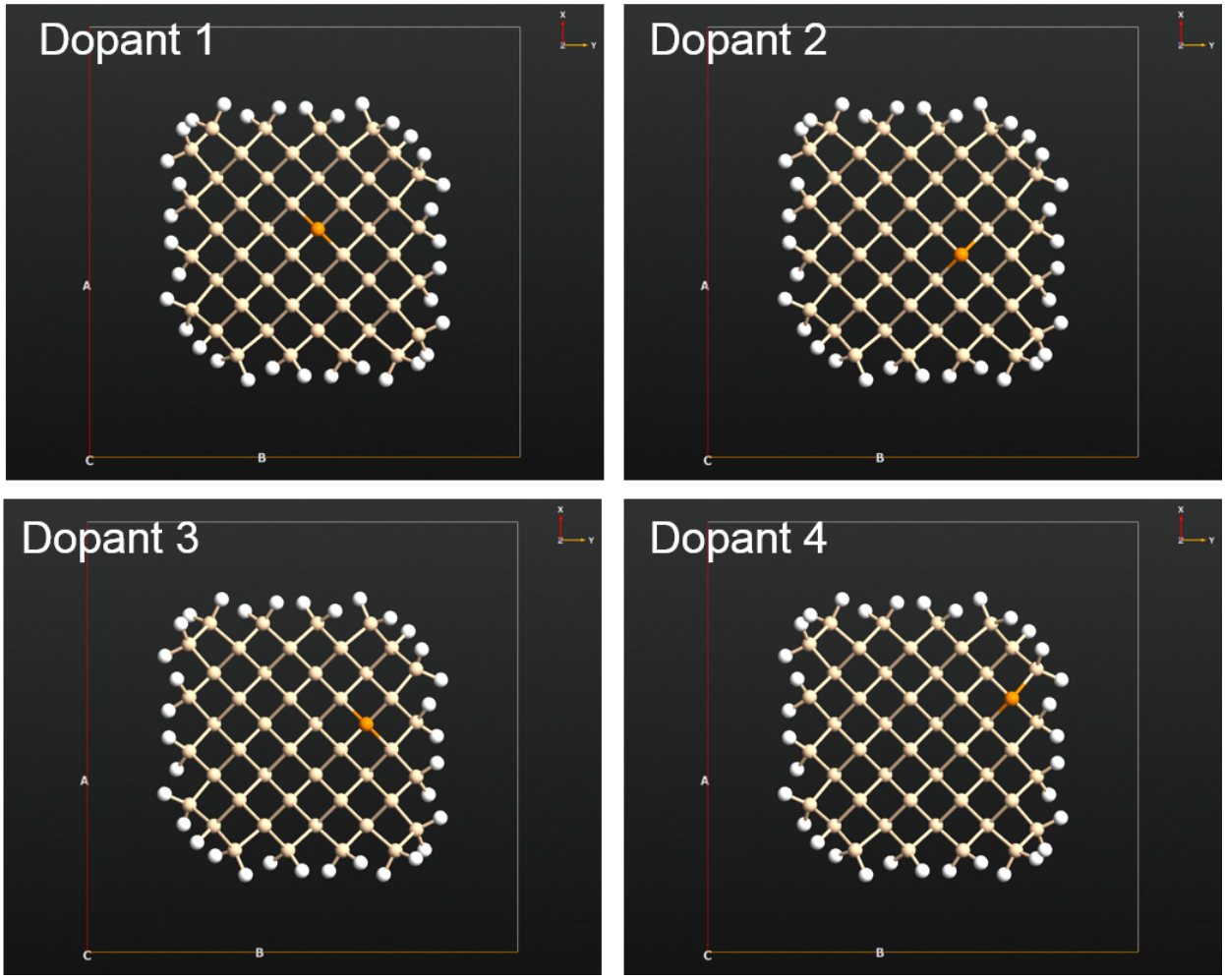
Two different types of nanowires will be considered:

1. The **pristine device** is constructed from pure, defect-free, relaxed silicon, passivated with hydrogen at the surface;
2. The **defected devices** are generated by substituting one Si atom with P (in 4 different positions) and are relaxed as bulk.

More information on how to build the structures can be found at the end of the tutorial, in the *Appendix: Building the nanowires*.

Computational method: The calculations are performed with DFT and NEGF, using a GGA xc-functional and the FHI-SZP basis sets. We use 1x1x51 k-points for the NEGF calculations.





Transmission spectra

The calculated transmission spectra are available via download. They have been calculated using 201 points in an energy range between $E - E_{\mathrm{F}} = 0.5eV$ and $E - E_{\mathrm{F}} = 2.0eV$.

- For the *pristine* nanowire, the transmission is calculated for the infinite bulk. `sinw_pristine_trans.hdf5`
- For the *defected* nanowires, we have used a two-probe device configuration. `sinw_P_dopant_trans_1234.hdf5`

Elastic scattering mean free path

In this section, we will analyze how the presence of an impurity atom impacts the transmission through the nanowire, and we will also estimate the mean free path for the conduction electrons. The script `T_and_mfp.py` calculates and plots the transmission and the mean free path using the theory described in the *theoretical background*.

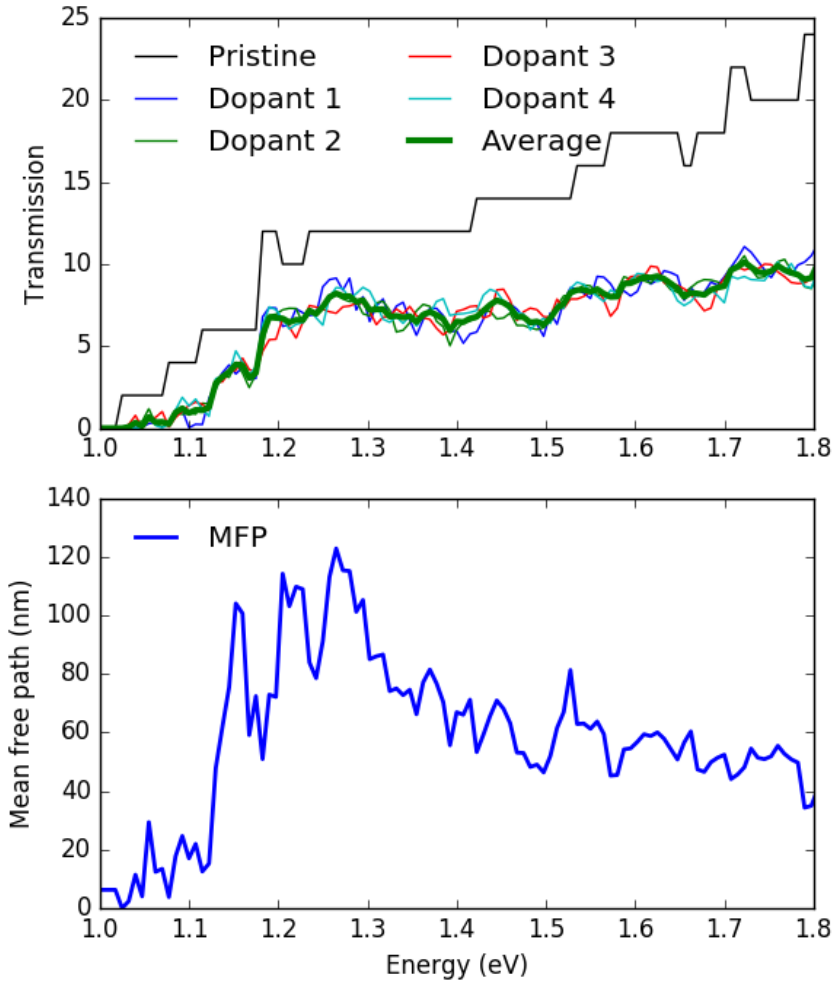
The script does the following:

1. Aligns the conduction band minima, to give the two datasets a common energy reference.
2. Calculates the scattering resistance $R_s(E)$ from the average transmission and the contact resistance $R_c(E)$ from the pristine transmission.
3. Saves the data to a file, for re-use later.

4.Calculates the mean free path from $R_c(E)$ and $R_s(E)$ and the average distance between dopant atoms.

5.Plots of transmissions and mean free path as functions of energy.

Run the script as `atkpython T_and_mfp.py` or with the Job Manager, in the directory where you have saved the files with the transmission spectra, and the following figure should appear:



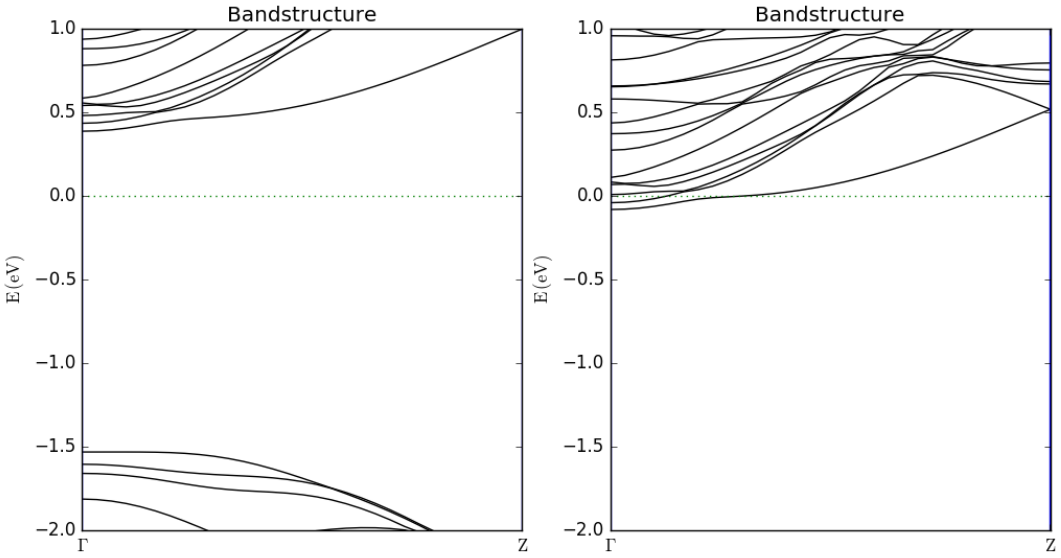
The first plot compares the transmission function for the pristine and the defected nanowires. It can be seen that the transmission of the defected nanowires is similar and always lower than that of the pristine nanowire. The second plot shows the mean free path as a function of energy.

Note how, close to the conduction band edge at energies between 1.0 eV and 1.2 eV, the electrons are strongly scattered by the dopant impurity, and the associated transmission functions are very low. This is reflected in a short mean free path. At higher energies E > 1.2 eV, there is less scattering and both the transmission and mean free path are larger.

Fermi levels in doped nanowires

The next task is to estimate the impurity limited mobility as a function of doping density. In order to do this, you need to know the position of the Fermi level relative to the conduction band edge. This can be achieved by calculating the band structure for silicon nanowires which are uniformly doped. This can be done in QuantumATK using atomic compensation charges. You can learn about doping in QuantumATK in the technical note here: *Doping methods available in QuantumATK* and in the tutorial here: *Silicon p-n junction*. The bandstructure calculations use 1x1x9 k-points and the $\Gamma \rightarrow$ Z route. You can download the data files here: `doped-bandstructures.zip`.

After unzipping the files you can compare the bandstructures in QuantumATK using the **Bandstructure Analyzer**:



Note that in this case, both bandstructures are referenced to the Fermi level, and it is therefore also easiest to align them here. This means that we need to look at the positions of the bands to study the shift in the Fermi level. We see that the conduction band has shifted about 0.5 eV down when going from a doping level of 10^{14} cm^{-3} to 10^{21} cm^{-3}, which is more properly described as the Fermi level shifting up with increasing doping.

Doping dependent mobility

You now have the ingredients needed to estimate the mobility as function of doping concentration. The python script below loads the calculated data and calculate the conductivity as well as mobility for a 1 μm long Silicon nanowire. By calculating the mobility in this way, as also described in the *theoretical background*, the following assumptions are made:

- Each dopant atom is fully ionized and contributes with one electron to the conduction band charge density.
- The transmission of the long wire can be estimated from the single-dopant transmissions.

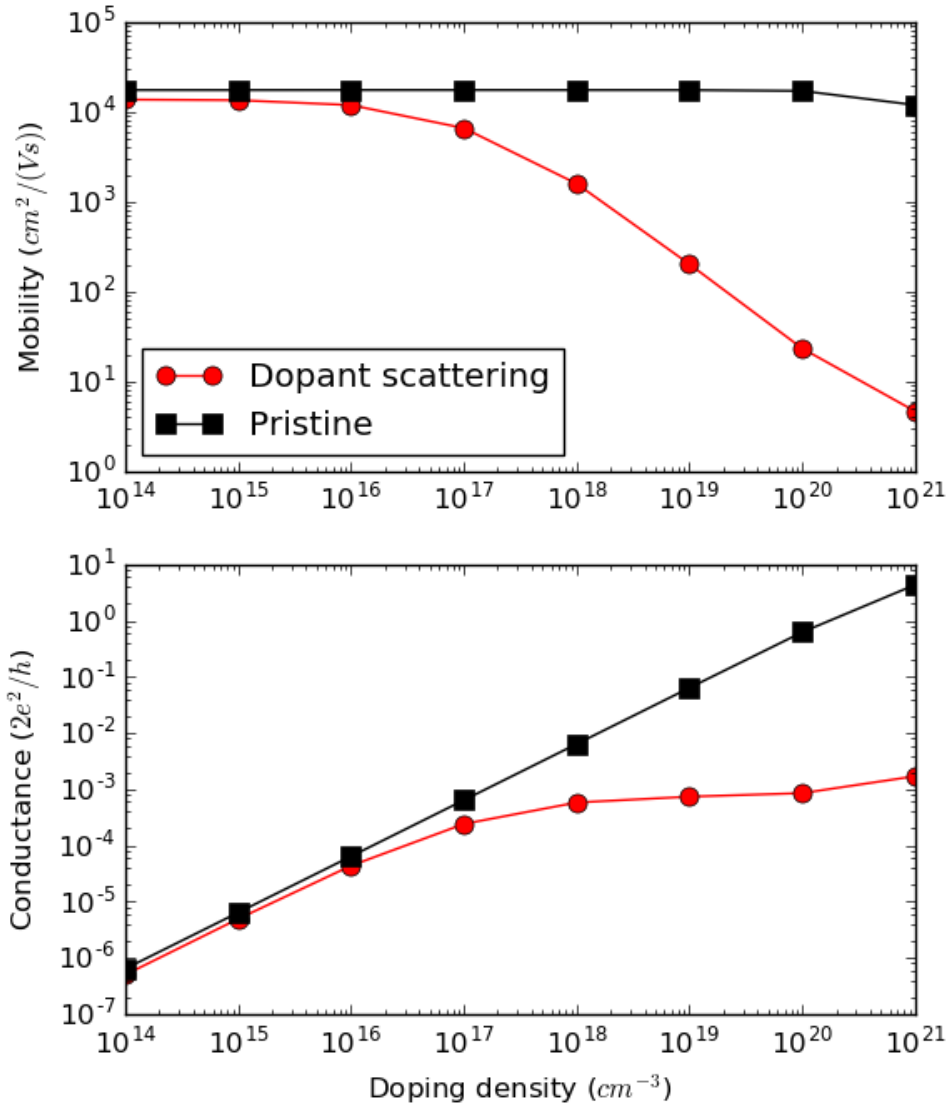
> **Warning:** Note that the mobility defined in this way is very much an estimate of the mobility of electrons for the pristine wire, as we do not include any explicit scattering mechanisms. It is not directly comparable with the, for the pristine wire more appropriate, mobility that is calculated by f.ex. the *Mobility* class.

Notice that although the transmission calculations with the dopant atoms to some extend include the doping effect - i.e. that the P atom becomes partly ionized - this effect is not included in the position of the Fermi level, which is determined

by the electrodes, which are un-doped in this calculation. The bandstructure calculations with the compensation charge doping are thus needed to consider this effect. You can download the script for these calculations here: `mobility.py`. The script does the following:

1. Calculates the conductances and mobilities starting from the transmission;
2. The conductances and mobilities are plotted as a function of doping level, both with and without taking scattering processes due to dopant atoms into account.

Run the script as `atkpython mobility.py`, and the following figure should appear:



Both the mobility and the conductance are calculated and plotted for two cases:

i. Including dopant scattering in the transmission calculation of the long wire (red circles).
ii. For a pristine, but uniformly doped, wire transmission (black squares).

When the dopant scattering is included, two factors affect the conductance:

- As for the pristine wire case, the Fermi level moved closer to the conduction band for increased doping density. This effect increases the conductance.

•Due to the scattering by the dopant impurities, the energy resolved resistance increases linearly with length according to the equations shown in the theory. This effect will decrease the conductance.

From the conductance plot (bottom), one can observe that the two effects approximately cancel out for doping densities $n > 10^{18}$ cm^{-3}, where the conductance is more or less constant. Since the conductance is constant, the mobility will decrease as $\mu \propto 1/n$ for $n > 10^{18}$ cm^{-3}.

For doping levels $n < 10^{18}$ cm^{-3}, the average dopant-dopant distance becomes larger than the wire length of 1 $\mu\mathrm{m}$ and there will on average be no dopants in the wire, which is the reason why the doped wire curves approach the pristine wire curves. If we had chosen a longer wire, this onset would happen at lower doping concentrations, as it is related to the ratio $L/L_{\mathrm{mfp}}(E)$.

Summary and discussion

As mentioned previously, the focus of this tutorial was not so much on how to calculate very accurate and detailed transmission spectra, but rather on how to extract information from them. In the present case of phosphorus dopants in a silicon nanowire we note the following points:

•The actual transmission functions and mobilities will depend on the diameter and orientation of the wire.

•The transmission might also depend on the basis set, and differ from the one presented here, if *e.g.* a larger basis set (like DoubleZetaPolarized) is used.

One should also be aware that the analysis presented in this tutorial is not intended to be very quantitative. It rather provides a relatively straightforward approach to estimate the order of magnitude of the elastic scattering by defects. Recall that the actual system considered would be a 1 $\mu\mathrm{m}$ long nanowire (this would include more than 150,000 atoms) with randomly placed dopants.

Finally, one should notice that the methods presented in this tutorial are applicable to other kinds of structural disorder such as surface defects[2], or surface roughness scattering[3], and can be applied to both electron and phonon transmission spectra[Page 639, 2, 3].

Appendix: Building the nanowires

Detailed instructions for building a Si nanowire can be found in this tutorial: *Silicon nanowire field-effect transistor*. However, you may also use the **Nanowire** plugin in the Builder. In this case, we have used a nanowire bounded by (111) and (100) facets and a radius of 7.0 Å.

References

3.2.13 Virtual Crystal Approximation for InGaAs random alloy simulations

Version: 2017

Downloads & Links

Effective band structure of random alloy InGaAs VirtualCrystalBasisSet

[2] T.ăMarkussen, A.-P. Jauho, and M.ăBrandbyge. Electron and phonon transport in silicon nanowires: Atomistic approach to thermoelectric properties. *Phys. Rev. B*, 79:035415, Jan 2009. doi:10.1103/PhysRevB.79.035415.

[3] Troels Markussen. Surface Disordered GeSi Core–Shell Nanowires as Efficient Thermoelectric Materials. *Nano Letters*, 12(9):4698–4704, 2012. doi:10.1021/nl302061f.

In this tutorial you will learn how to use the virtual crystal approximation (VCA) in order to simulate properties of random alloys with $In_xGa_{1-x}As$ as the specific system of interest. You will use VCA with the *PBEsol* exchange-correlation (xc) functional for determination of the lattice constant as a function of the In fraction x, and then use VCA with the TB09-MGGA xc-functional to compute the band structures.

Introduction

Random alloys are becoming technologically important materials. SiGe is one of the best known materials for thermoelectric elements and $In_xGa_{1-x}As$ is a prime candidate to substitute or complement silicon in future CMOS devices. Simulations of both thermoelectric elements as well as transistor devices often refers to band structure parameters such as band gaps and effective masses. In a random alloy, one can calculate the so-called *EffectiveBandstructure* (EBS), which includes effects of the random disorder. However, EBS calculations require large unit cells and are rather time consuming. For more details, see the tutorial: *Effective band structure of random alloy InGaAs*.

A different approach to modeling a random alloy is to use the virtual crystal approximation (VCA), which is the topic of the present tutorial. In a VCA simulation of, for instance, $In_xGa_{1-x}As$ one creates a new, virtual atom, which is a mixture of In and Ga with appropriate weights. The actual calculation of e.g. lattice constants or band structures can then be performed on a primitive cell with only two atoms: the normal As atom and the virtual In-Ga atom. See also the manual page: *VirtualCrystalBasisSet*.

Compared with EBS calculations, VCA simulations are much less computationally demanding, since there is no requirement for large super cells. Conversely, VCA simulations do not capture any effects of disorder, as in the EBS approach. However, as it is shown in the EBS tutorial, the random disorder has a small effect on the band structure around the Γ-point, which is also the most important region for InGaAs. For this reason it makes good sense to study the band structure with VCA.

Setting up the VCA calculations for $In_xGa_{1-x}As$

In the first part of the tutorial, you will calculate the lattice constant of $In_xGa_{1-x}As$ vs. In fraction x. You will be using the the GGA-type PBEsol exchange-correlation functional.

- Open the **Builder** and add an InAs bulk configuration:

  – *Add* → *From Database* → *InAs*

  – Send the InAs structure to the **Script Generator**.

In the **Script Generator** go through the following points:

- Add a **New Calculator**, open it and apply the settings:

  – Calculators: *ATK-DFT:LCAO* (default)

  – k-point sampling: 9x9x9.

  – Exchange correlation: *PBES*

  – Basis set: Change *Arsenic* to *High*.

- Add an **OptimizeGeometry** object and open it.

  – Un-check *Constrain Lattice Vectors* to also optimize the lattice parameter.

Note: Note that we use Medium basis sets for all elements. It was changed to High in the Script Generator to make sure the basis set was printed in the script, allowing us to easily modify it.

- Add a **Bandstructure** object and open it.
  - Set *Points pr. segment* to 100
  - Change the *Brillouin zone route* to **L, G, X** to just include a region around the Γ-point.
- Note the filename - you will need it later.

Note: The particular choice of filename is not important, as we will change it later. In this way it is easy to do an automatic search and replace without changing other things by accident.

In order to use the VCA basis set, you need to modify the python script. Send the script to the **Editor** and find the lines:

```
basis_set = [
    BasisGGASG15.Arsenic_High,
    BasisGGASG15.Indium_Medium,
    ]
```

and replace them with the following lines:

```
VirtualCrystalBasis = VirtualCrystalBasisSet([BasisGGASG15.Indium_Medium,BasisGGASG15.
↪Gallium_Medium], x=x)
basis_set = [
    VirtualCrystalBasis,
    BasisGGASG15.Arsenic_Medium,
    ]
```

In the first line the VirtualCrystalBasis is defined, based on regular basis sets of the SG15-Medium type. The VirtualCrystalBasis is constructed with:

- A list of two basis sets.
- A number $0x1$, defining the fraction of the first basis set.

The VirtualCrystalBasis will be associated with atoms in the configuration corresponding to the first element in the list - in this case Indium. So we now have a configuration containing regular Arsenic, and an atom which is a tunable mix of In and Ga, represented by the In atom in the actual BulkConfiguration.

You can now calculate the lattice constant and band structure for $In_xGa_{1-x}As$ with an arbitrary value of x. However, an advantage of VCA is the very short calculation time, so we would rather do calculations for many values of x to show the detailed dependence on x. We therefore need to modify the script to loop over x. Add the following lines at the top of the script:

```
# -*- coding: utf-8 -*-

x_range = numpy.linspace(0, 1.0, 11)
print('x_range', x_range)
for x in x_range:
    filename = 'ingaas-x%i.hdf5' % (x*100)
```

This will make a loop over x in intervals of 0.1, and create a separate file for each. Now go through the script (manually or with automatic replacement) and replace all instances of the filename (default is `'InAs.hdf5'`) with simply

`filename` and indent all the lines as required by Python in a loop. Finally, we need to add the TB09-MGGA calculator, to get the band gap calculated with TB09-MGGA instead of PBES. Insert these lines between the OptimizeGeometry and Bandstructure parts, and make sure they have the correct indentation

```
exchange_correlation = MGGA.TB09LDA

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
nlsave(filename, bulk_configuration)
```

Read the script and make sure you understand what is being done and then run the calculation. It should take a couple of hours depending on hardware. Your script should look like this: `vca-ingaas-full.py`.

Note: For scientific use you should always carefully investigate if your numerical settings are converged. This includes in particular:

•k-point sampling

•Density mesh cutoff.

•Choice of basis sets/pseudopotentials - and their influence on the required mesh cutoff.

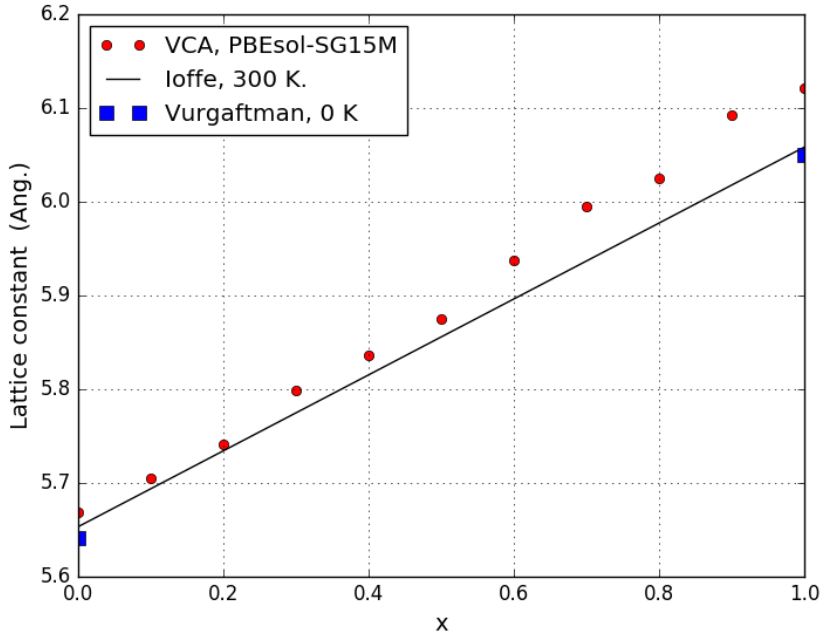
Analyzing the results for VCA with $In_xGa_{1-x}As$

Lattice constants

When the calculations are done, the results can be visualized with the `plot-lattice-constant.py` script. The lattice constant is plotted vs. the In fraction x. The solid line is an analytical expression which is based on experimental room temperature lattice constants[1], while the computed results correspond to zero temperature. Experimental low temperature lattice constants for GaAs ($x = 0$) and InAs ($x = 1$) lie slightly below the solid line[2].

[1] New Semiconductor Materials - Characteristics and Properties. GaInAs. http://www.ioffe.ru/SVA/NSM/Semicond/GaInAs/basic.html. Accessed: 2017-06-21.

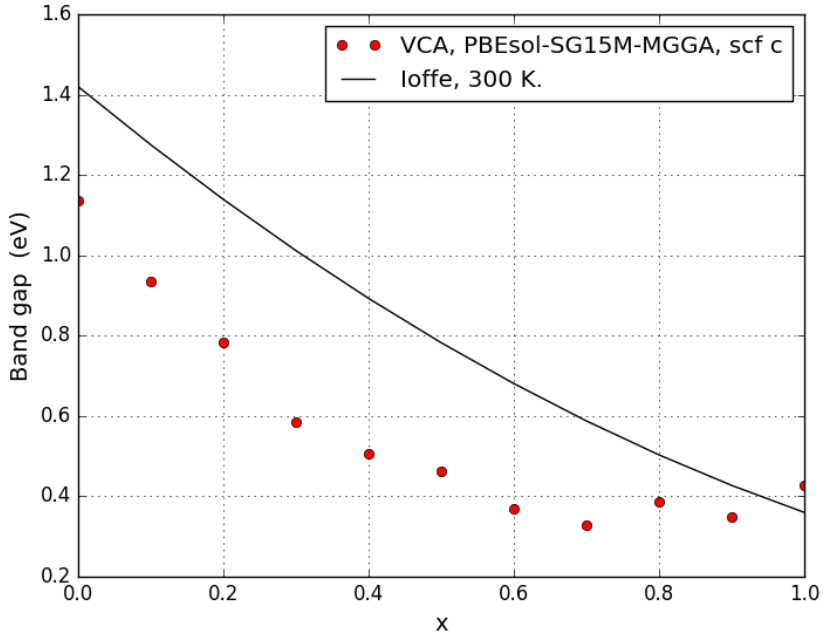
[2] I.ăVurgaftman, J.ăR. Meyer, and L.ăR. Ram-Mohan. Band parameters for III-V compound semiconductors and their alloys. *J. Appl. Phys.*, 89(11):5815–5875, jun 2001. arXiv:1206.0723, doi:10.1063/1.1368156.



We see that the experimental lattice constants are well reproduced in the Ga-rich region on the left of the figure, while the deviation is higher, about 1 %, in the In-rich region on the right. This is in line with general expectations, as PBEsol is designed to give accurate lattice constants, while GGA-type functionals generally overestimate them.

Band gaps - self-consistent TB09-MGGA

We also calculated bandstructures, from which we can extract the band gaps with the following script: `plot-band-gaps.py`. This will produce this plot:



We see that the band gaps show a qualitatively correct trend, but with rather large deviations in the Ga-rich region on the left and a bit of an oscillation on the right. Fortunately, it is possible to fit the c-parameter of TB09-MGGA to reproduce the correct band gaps of the end-points, which should also improve the band gaps in the intermediate region.

Band gaps - TB09-MGGA with fitted c-parameter

In order to fit the c-parameter, we need to re-visit the calculations for $x = 0$ and $x = 1$. The procedure for doing the fitting is rather straightforward and described in this tutorial: *Meta-GGA and 2D confined InAs*.

For our combination of computational parameters, we find that a c-parameter of 1.1331 optimizes the band gap of InAs and a c-parameter of 1.2680 optimizes the band gap of GaAs. We cannot know beforehand which parameter values optimize the band gap for the various compositions in between, so the most straightforward choice, is to simply make a linear interpolation between the two values.

We make a modified version of the previous script, which reads in the finished calculations, re-calculates with TB09-MGGA with a linear interpolation between the fitted c-parameter values and computes and saves a bandstructure:

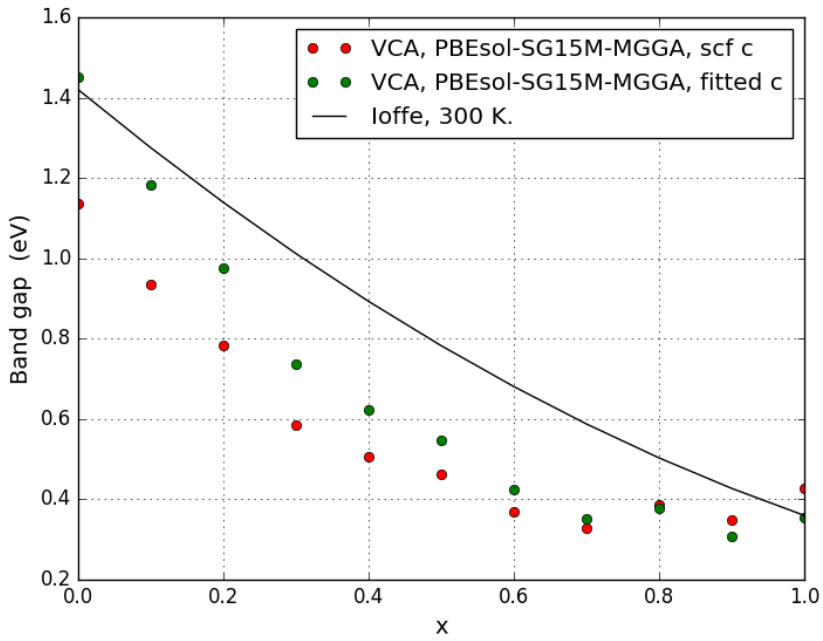
```
# -*- coding: utf-8 -*-
x_range = numpy.linspace(0, 1.0, 11)
print('x_range', x_range)
for x in x_range:
    filename = 'ingaas-x%i.hdf5' % (x*100)

    bulk_configuration=nlread(filename,BulkConfiguration)[-1]
    old_calculator=bulk_configuration.calculator()

    # Interpolated fitted c
    c = 1.2679609982 - 0.13488609 * x
    exchange_correlation = MGGA.TB09LDA(c=c)

    calculator = old_calculator(
```

(continues on next page)

(continued from previous page)

```
        exchange_correlation=exchange_correlation,
        )

    bulk_configuration.setCalculator(calculator)
    bulk_configuration.update()

    bandstructure = Bandstructure(
        configuration=bulk_configuration,
        route=['L', 'G', 'X'],
        points_per_segment=200,
        bands_above_fermi_level=All
        )
    nlsave(filename, bandstructure)
```

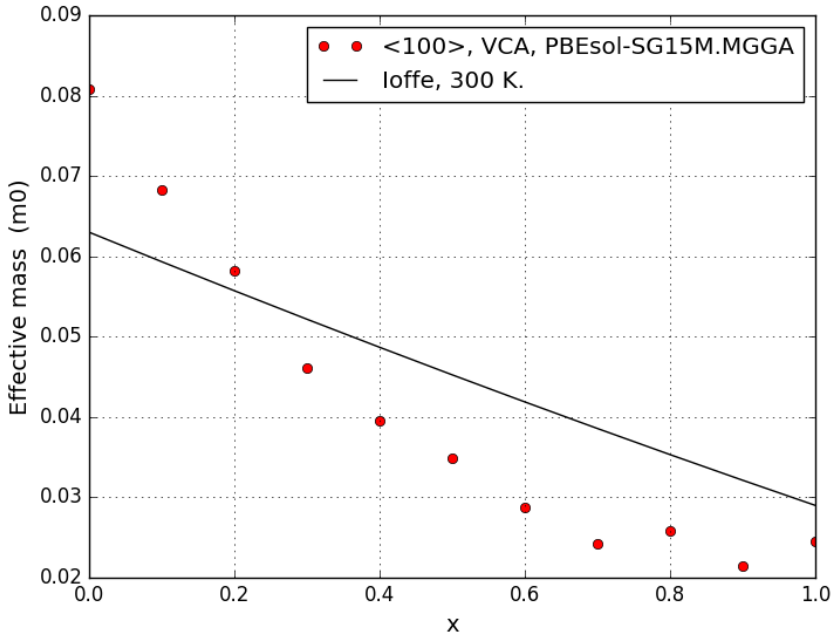
Run the script in the same folder, to re-do the TB09-MGGA calculations with a fitted c-parameter and get the corresponding bandstructures. We also modify the plotting script to include both datasets (`plot-both-band-gaps.py`), giving us this plot:



We see that this has improved the band gaps quite a bit in the Ga-rich part on the left, while the improvement is smaller in the In-rich part on the right.

Calculating effective masses

It is also possible to calculate the effective mass of the electrons as a function of x. See this tutorial: *Effective mass of electrons in silicon*, for general information on calculation of effective masses. The script is equivalent to the one used for the new TB09-MGGA calculations, and also plots the data, and can be found here: `effective-masses.py`. It will give this plot:



We see that the effective mass of GaAs is overestimated, leading to an overall distortion of the shape of the curve. However, it does capture the overall downwards trend in the effective mass from GaAs to InAs.

Summary and discussion

You have now learned how to use the VCA in order to calculate structural properties (lattice constants) and band structure properties (band gaps and effective masses) of $In_xGa_{1-x}As$. The lattice constants were well-reproduced, and while the fundamental band gap at the Γ point came out satisfyingly with the fitted TB09-MGGA c-parameter, the effective masses were not as accurate.

When using the VCA you must be aware that you are ignoring all effects of the random disorder. You are implicitly assuming that Blochs theorem applies and that the band structure is a well-defined quantity (or, alternatively that k is a good quantum number). This might, however, not always be the case, as shown in the calculations in the tutorial *Effective band structure of random alloy InGaAs*, which includes the random disorder. However, the effective band structure calculations also indicate that close to the Γ point, the band structure is indeed well-defined in $In_xGa_{1-x}As$. In that case it makes good sense to study the band structure details (effective mass, band gap, strain dependence, etc.) with the VCA, which is computationally much easier than the effective band structure calculation, which requires large supercells.

References

3.2.14 DFT-1/2 and DFT-PPS density functional methods for electronic structure calculations

Version: R-2020.09

Downloads & Links

DFTHalfParameters *PseudoPotentialProjectorShift*

QuantumATK offers two novel density functional corrections for computing the electronic structure of semiconductors and insulators: the DFT-1/2 and pseudopotential projector shift (DFT-PPS) methods. This tutorial gives an introduction to these methods.

DFT-1/2 is often also denoted LDA-1/2 or GGA-1/2, and is a semi-empirical approach to correcting the self-interaction error in local and semi-local exchange-correlation density functionals for extended systems, similar in spirit to the *DFT+U* method. As such, it can be viewed as an alternative to the TB09-*Meta-GGA* method with broadly the same aims; to improve the description of conduction-band energy levels and band gaps. Just like with TB09-MGGA, the DFT-1/2 method is **not** suitable for calculations that rely on total energy and forces, e.g., geometry optimization or molecular dynamics.

PPS is an abbreviation for pseudopotential projector shift. This method introduces empirical shifts of the nonlocal projectors in the SG15 pseudopotentials, in spirit of the empirical pseudopotentials proposed by Zunger and co-workers in[1]. The projector shifts have been adjusted to reproduce technologically important properties of semiconductors such as the fundamental band gap and lattice constant. The PPS method currently works out-of-the-box for the elements silicon and germanium, but may be used for other elements also by manually specifying the appropriate projector shifts (these must somehow first be determined). Importantly, the PPS method **does work** for geometry optimization of semiconductor material structures.

DFT-1/2 methods

The DFT-1/2 method attempts to correct the DFT self-interaction error by defining an atomic self-energy potential that cancels the electron-hole self-interaction energy. This potential is calculated for atomic sites in the system, and is defined as the difference between the potential of the neutral atom and that of a charged ion resulting from the removal of a fraction of its charge, between 0 and 1 electrons. The total self-energy potential is the sum of these atomic potentials. The addition of the DFT-1/2 self-energy potential to the DFT Hamiltonian has been found to greatly improve band gaps for a wide range of semiconducting and insulating systems[2]. For more information, see the QuantumATK Manual entry on the *DFT-1/2 method*.

Important: The method is not entirely free of empirical parameters. For eaxmple, a cutoff radius is employed. Following Refs.[3] and[Page 647, 2] it is chosen such as to maximize the band gap.

[1] L.-W. Wang and A.ăZunger. Local-density-derived semiempirical pseudopotentials. *Phys. Rev. B*, 51:17398–17416, 1995. doi:10.1103/PhysRevB.51.17398.

[2] LuizăG. Ferreira, Marcelo Marques, and LaraăK. Teles. Slater half-occupation technique revisited: The LDA-1/2 and GGA-1/2 approaches for atomic ionization energies and band gaps in semiconductors. *AIP Adv.*, 1(3):032119, 2011. doi:10.1063/1.3624562.

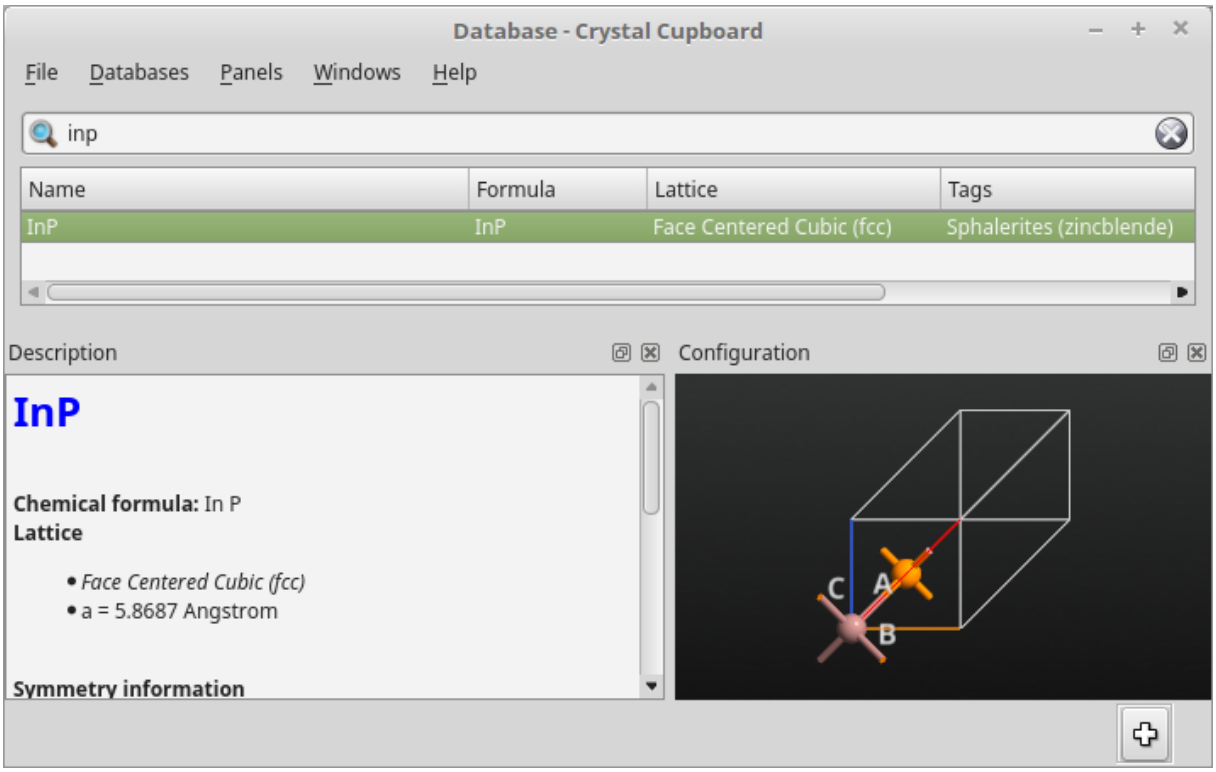
[3] LuizăG. Ferreira, Marcelo Marques, and LaraăK. Teles. Approximation to density functional theory for the calculation of band gaps of semiconductors. *Phys. Rev. B*, 78:125116, Sep 2008. doi:10.1103/PhysRevB.78.125116.

Note also that not all elements in the system necessarily require the DFT-1/2 correction; it is generally advisable only to add this to the anionic species, and leave the cationic species as normal.

Default DFT-1/2 parameters are available in QuantumATK; these have been optimized against a wide range of materials, and should improve upon the standard LDA or GGA band gap in most cases.

InP bandstructure using PBE-1/2

Create a new QuantumATK project (preferably in a new folder), and open the **Builder**. Then click *Add* → *From Database* and locate the InP crystal in the **Database**.

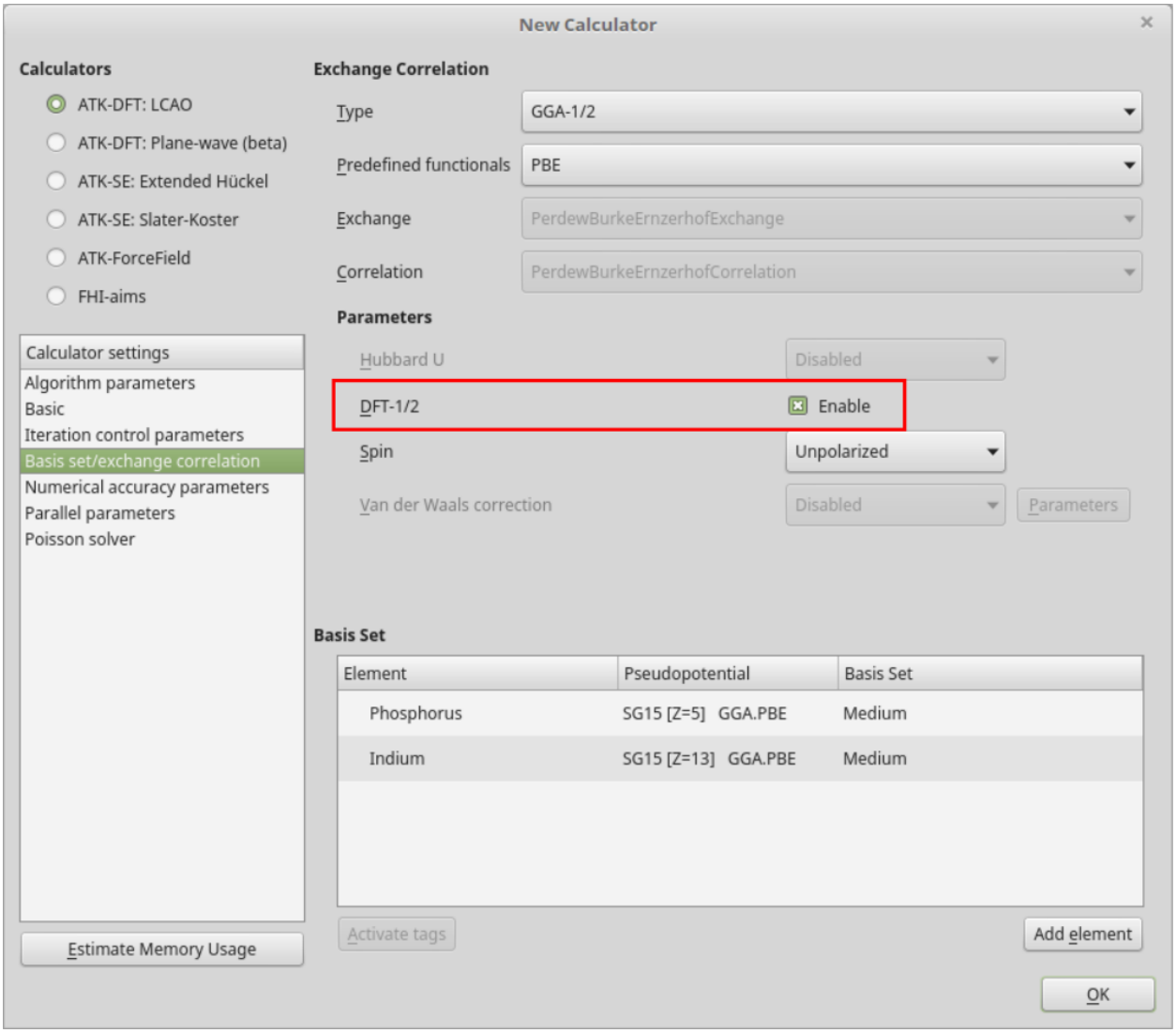


Add the InP cystal to the Builder **Stash**, and send the configuration to the **Script Generator**. Then add the following script blocks:

- **New Calculator**
- **Bandstructure**

and open the calculator widget to adjust the calculator parameters.

Select a 9x9x9 k-point grid, and navigate to the *Basis set/exchange correlation* tab. Note that the default exchange-correlation method is GGA (PBE). Enable the DFT-1/2 correction by ticking the checkbox indicated below.

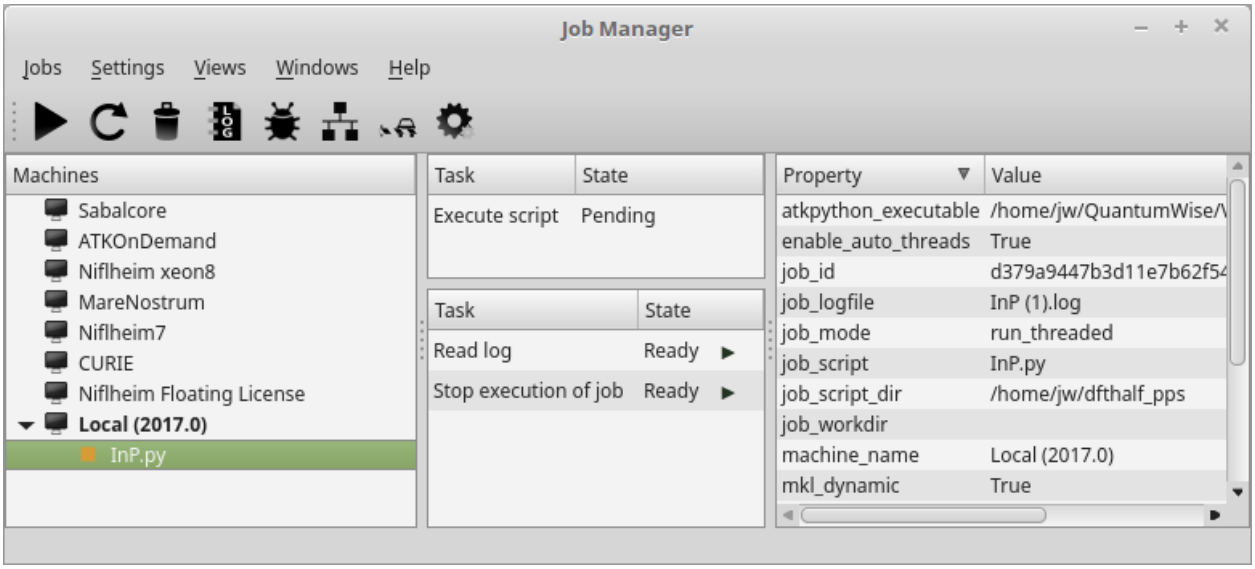


Close the calculator widget and save the script as `InP.py`. If you open the script using the **Editor**, you will see that the exchange-correlation method is `GAHalf.PBE`:

```
#------------------------------------------
# Exchange-Correlation
#------------------------------------------
exchange_correlation = GGAHalf.PBE
```

that is, GGA-1/2 using the PBE functional, which might also be denoted PBE-1/2. You may download the script here if needed: `InP.py`

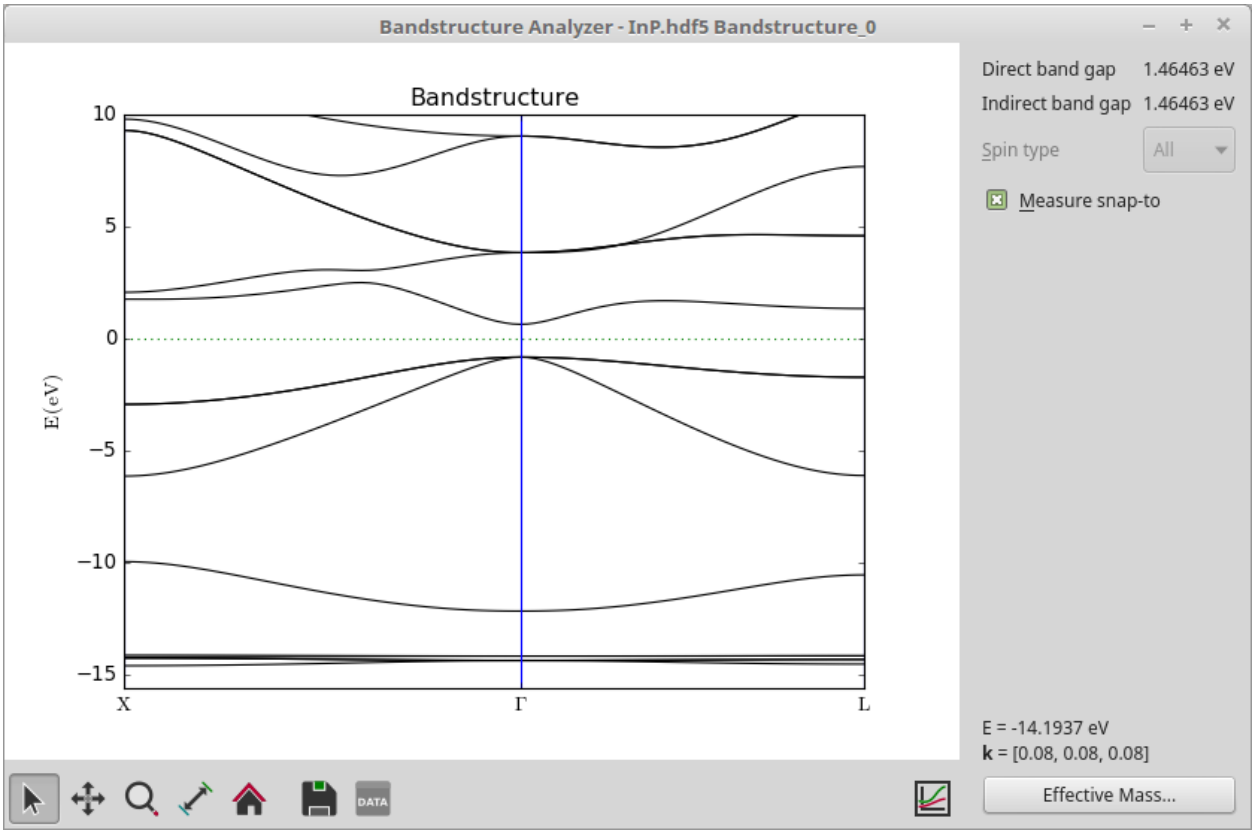
Run the script using either the **Job Manager**:



or from a terminal:

```
$ atkpython InP.py > InP.log
```

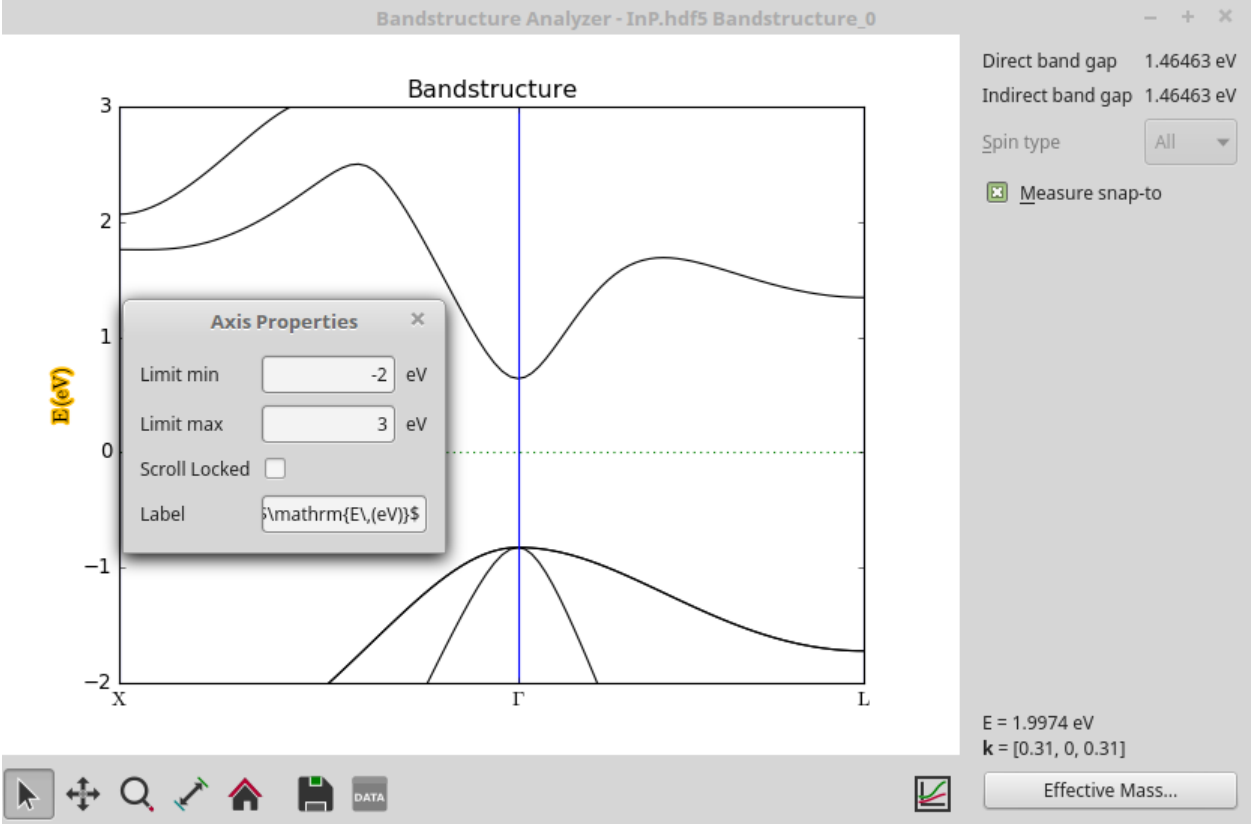
The calculation will be very fast. Once done, use the **Bandstructure Analyzer** to plot the resulting band structure. You will correctly find that InP is a direct-gap semiconductor with a band band gap of 1.46 eV, which is close to the experimental gap of 1.34 eV from Ref.[4].



[4] M.ăLevinshtein, S.ăRumyantsev, and M.ăShur, editors. *Handbook Series on Semiconductor Parameters*. Volumeă1. World Scientific Publishing Cp. Pte. Ltd., Singapore, 1996. ISBN 978-981-02-2934-4.

Tip: You may want to **zoom in** on the bands around the Fermi level for a better view. There are two ways to do this:

1. use the Zoom Tool (third icon from the left);
2. or click the y-axis label to select it, then right-click and select *Edit item* in order to open the *Axis Properties* widget, where you can adjust the limits of the y-axis.

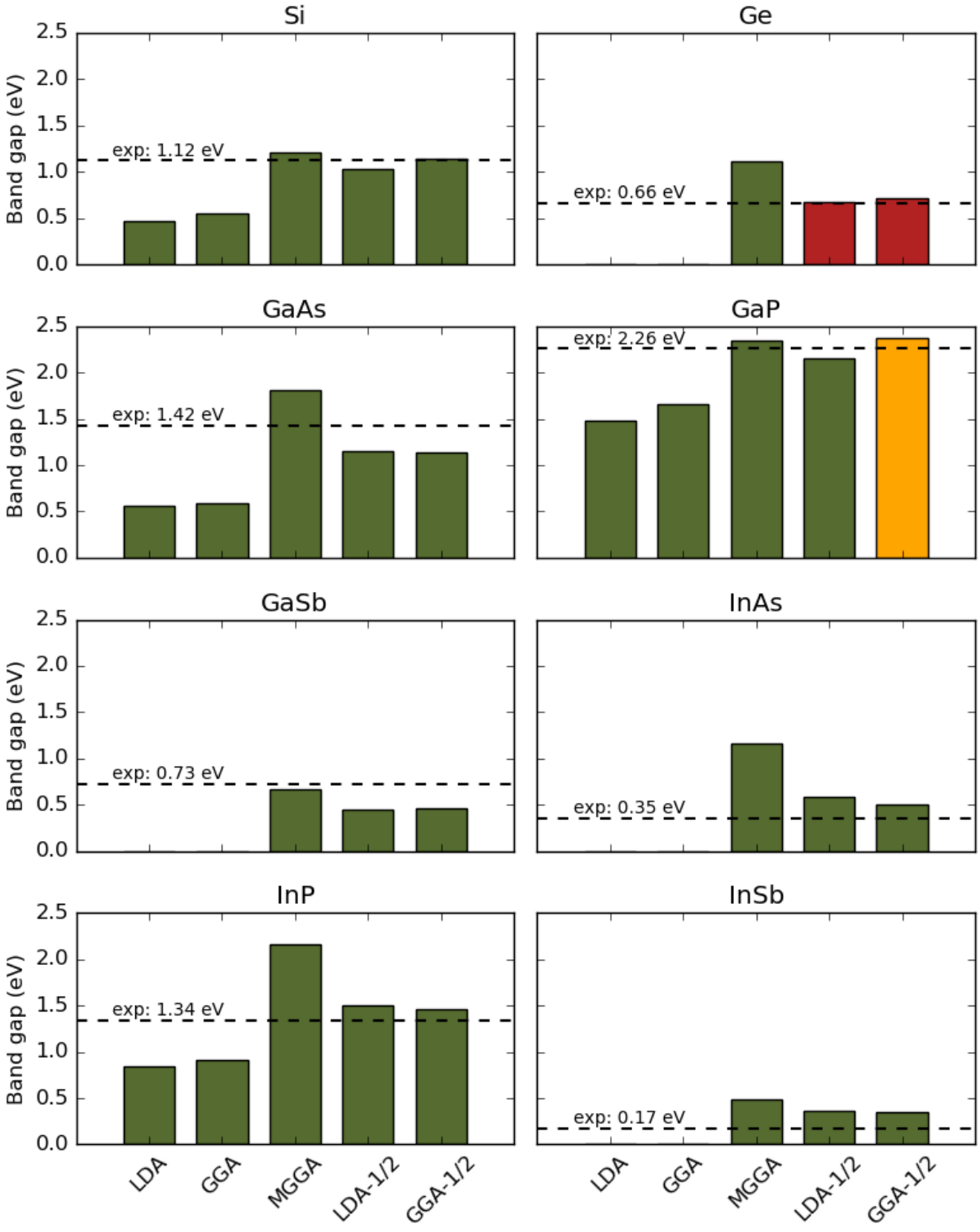


III-V type semiconductor band gaps

The plots below show how the DFT-1/2 methods (LDA-1/2 and GGA-1/2) compare to LDA, GGA, and TB09-MGGA standard band gap calculations using default pseudopotentials and basis sets. A 9x9x9 k-point grid was used in all calculations, and experimental band gaps were adapted from Ref.[Page 650, 4].

It is quite clear that the DFT-1/2 correction improves on standard LDA and GGA band gaps, which are usually too small or non-existing (no bar). TB09-MGGA calculations with self-consistently determined c-parameter also improves band gaps in general.

The red bars for Ge using LDA-1/2 and GGA-1/2 indicate that a *direct* band gap is wrongly predicted (Γ–Γ instead of Γ–L as in experiments). The orange bar for GaP using GGA-1/2 indicates that the gap is correctly predicted to be *indirect*, but that the band energy in the L-valley is wrongly predicted to be a little lower than in the X-valley.


Si
Ge
GaAs
GaP
GaSb
InAs
InP
InSb
Band gap (eV)
exp: 1.12 eV
exp: 0.66 eV
exp: 1.42 eV
exp: 2.26 eV
exp: 0.73 eV
exp: 0.35 eV
exp: 1.34 eV
exp: 0.17 eV
0.0
0.5
1.0
1.5
2.0
2.5
LDA
GGA
MGGA
LDA-1/2
GGA-1/2

Manually specifying DFT-1/2 parameters

It is of course possible to manually specify the DFT-1/2 parameters instead of using the default ones. This is done by creating an instance of the *DFTHalfParameters* class, which is then given as an argument to the basis set.

For example, lets consider the case of GaAs. The following script manually sets up the As DFT-1/2 parameters identical to the default ones, and leaves the DFT-1/2 correction disabled for Ga, which is also default behavior. Note the `dft_half_parameters` argument to the basis set:

```
#----------------------------------------
# Basis Set
#----------------------------------------
# LDA-1/2 parameters for As
dft_half_parameters = DFTHalfParameters(
    element=Arsenic,
    fractional_charge=[0.3, 0.0],
    cutoff_radius=4.0*Bohr,
    )
# No LDA-1/2 parameters are needed for Ga (Disabled)
basis_set = [
    LDABasis.Arsenic_DoubleZetaPolarized(
        dft_half_parameters=dft_half_parameters),
    LDABasis.Gallium_DoubleZetaPolarized(
        dft_half_parameters=Disabled),
    ]
```

For more information, see the sections *Input format for DFT-1/2* and *DFT-1/2 default parameters* in the QuantumATK Manual.

Warning: Choosing the appropriate DFT-1/2 parameters may be a very delicate matter, and great care has been taken in determining the default parameters. If you choose to use non-default DFT-1/2 parameters, the quality of those parameters is entirely your own responsibility as a user!

QuantumATK does not offer support for determining custom DFT-1/2 parameters; we recommend in general that users stick to the default ones.

DFT-PPS method

As already mentioned, the DFT-PPS method applies shifts to the nonlocal projectors in the SG15 pseudopotentials. The nonlocal part of the pseudopotential, $\hat{V}_{\mathrm{nl}}$, is modified according to

$$\hat{V}_{\mathrm{nl}} \rightarrow \hat{V}_{\mathrm{nl}} + \sum_{l} |p_l\rangle \alpha_l \langle p_l|,$$

where the sum is over all projectors p_l, and α_l is an empirical parameter that depends on the orbital angular momentum quantum number l. Note that this approach does not increase the computational cost of DFT calculations!

The required projector shift parameters have been optimized for silicon and germanium and for use with the PBE density functional and SG15 pseudopotentials only. These are implemented as separate basis sets:

```
BasisGGASG15.Silicon_LowProjectorShift
BasisGGASG15.Silicon_MediumProjectorShift
BasisGGASG15.Silicon_HighProjectorShift
```

(continues on next page)

(continued from previous page)

```
BasisGGASG15.Silicon_UltraProjectorShift

BasisGGASG15.Germanium_MediumProjectorShift
BasisGGASG15.Germanium_HighProjectorShift
BasisGGASG15.Germanium_UltraProjectorShift
```

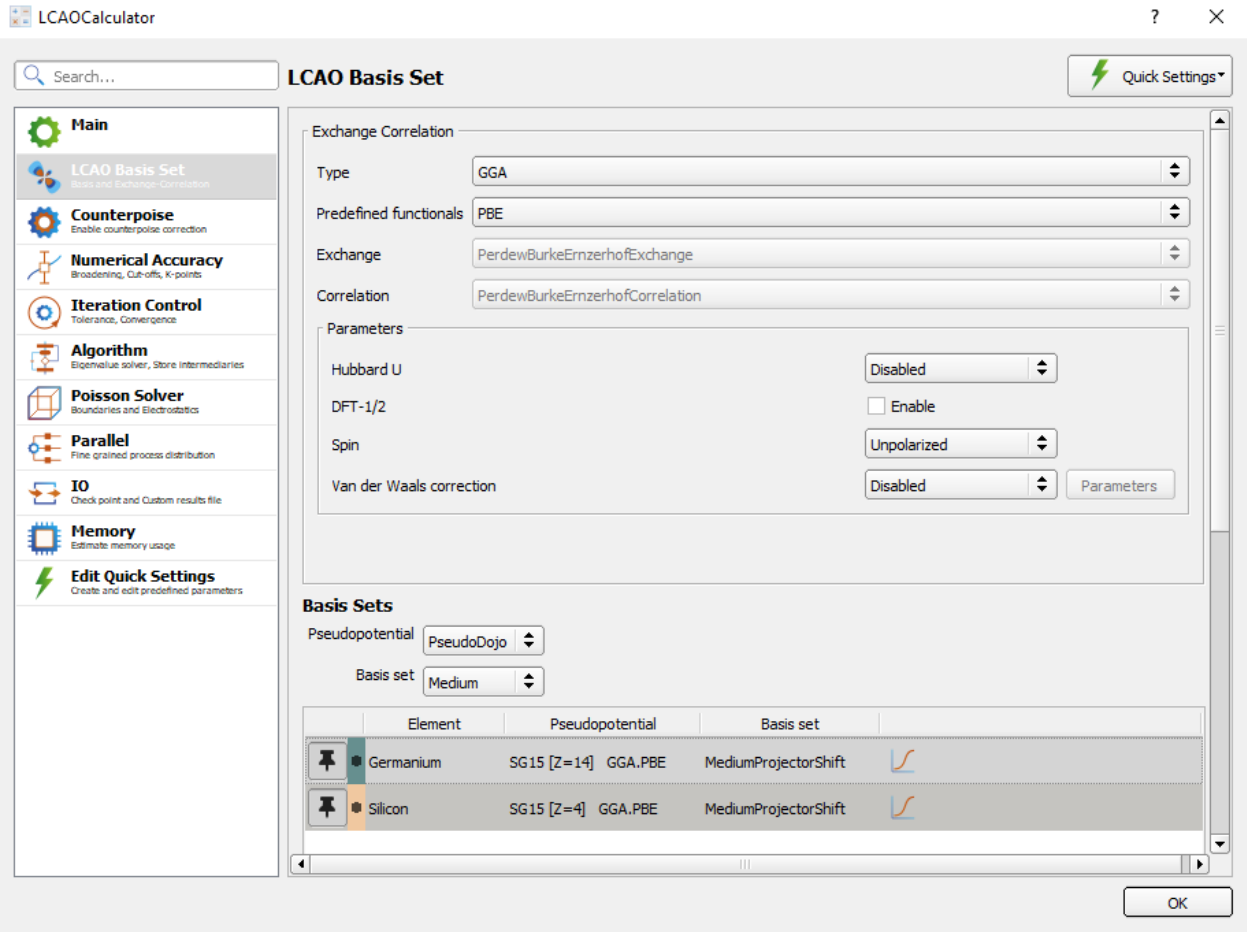
For each element, the same set of optimized projector shifts are applied in all SG15 basis sets. The script `projector_shifts.py` prints the built-in DFT-PPS parameters for Si and Ge:

```
Si
s-shift: +21.33 eV
p-shift: -1.43 eV
d-shift: +0.00 eV
Ge
s-shift: +14.55 eV
p-shift: +0.20 eV
d-shift: -2.17 eV
```

Note that the `d-shift` for Si is 0.0 eV since silicon has no d-electrons.

Si, SiGe, and Ge band gaps and lattice constants

The DFT-PPS method is enabled in the **Script Generator** by selecting one of the `ProjectorShift` basis sets for Si or Ge, as indicated below. No other calculator settings need to be changed in order to switch from ordinary PBE to DFT-PPS.



One very convenient aspect of the DFT-PPS method is that it allows for geometry optimization (forces and stress minimization) just like ordinary GGA calculations do – in fact, the DFT-PPS parameters may often be chosen such as to give highly accurate semiconductor lattice constants while also producing accurate band gaps.

In the following, you will consider bulk Si and Ge, and a simple 50/50 SiGe alloy. These 3 bulk configurations are defined in the script `bulks.py`. The script `pbe.py` runs geometry optimization and band structure analysis for all 3 configurations, while the script `pps.py` does the same using the DFT-PPS method.

The latter script looks like shown below. Note the lines from 2 to 31 in that particular script, where the bulk configurations are imported from the external script, and a Python loop over these configurations and basis sets is set up. Also, the DFT-PPS shifts are here manually specified.

```
1  # -*- coding: utf-8 -*-
2  from bulks import si, ge, sige
3  setVerbosity(MinimalLog)
4
5  configurations = [si,ge,sige]
6  labels         = ['Si','Ge','SiGe']
7
8  # Basis set for Silicon
9  SiliconBasis_projector_shift = PseudoPotentialProjectorShift(
10     s_orbital_shift=21.33*eV,
11     p_orbital_shift=-1.43*eV,
12     d_orbital_shift=0.0*eV,
```

(continues on next page)

(continued from previous page)

```
    f_orbital_shift=0.0*eV,
    g_orbital_shift=0.0*eV
    )
SiliconBasis = BasisGGASG15.Silicon_Medium(projector_shift=SiliconBasis_projector_shift)

# Basis set for Germanium
GermaniumBasis_projector_shift = PseudoPotentialProjectorShift(
    s_orbital_shift=14.55*eV,
    p_orbital_shift=0.20*eV,
    d_orbital_shift=-2.17*eV,
    f_orbital_shift=0.0*eV,
    g_orbital_shift=0.0*eV
    )
GermaniumBasis = BasisGGASG15.Germanium_Medium(projector_shift=GermaniumBasis_projector_
↪shift)

# Combined basis sets
basis_sets = [[SiliconBasis], [GermaniumBasis], [SiliconBasis, GermaniumBasis]]

for bulk_configuration,label,basis_set in zip(configurations,labels,basis_sets):
    outfile = "%s_PPS.hdf5" % label

    # ------------------------------------------------------------
    # Calculator
    # ------------------------------------------------------------
    k_point_sampling = MonkhorstPackGrid(
        na=9,
        nb=9,
        nc=9,
        )
    numerical_accuracy_parameters = NumericalAccuracyParameters(
        k_point_sampling=k_point_sampling,
        density_mesh_cutoff=90.0*Hartree,
        )

    calculator = LCAOCalculator(
        basis_set=basis_set,
        numerical_accuracy_parameters=numerical_accuracy_parameters,
        )

    bulk_configuration.setCalculator(calculator)
    nlprint(bulk_configuration)
    bulk_configuration.update()
    nlsave(outfile, bulk_configuration)

    # ------------------------------------------------------------
    # Optimize Geometry
    # ------------------------------------------------------------
    fix_atom_indices_0 = [0, 1]
    constraints = [FixAtomConstraints(fix_atom_indices_0)]

    bulk_configuration = OptimizeGeometry(
```

(continues on next page)

(continued from previous page)

```
        bulk_configuration,
        max_forces=0.05*eV/Ang,
        max_stress=0.1*GPa,
        max_steps=200,
        max_step_length=0.2*Ang,
        constraints=constraints,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
        constrain_bravais_lattice=True,
    )
    nlsave(outfile, bulk_configuration)
    nlprint(bulk_configuration)

    # ------------------------------------------------------------
    # Bandstructure
    # ------------------------------------------------------------
    bandstructure = Bandstructure(
        configuration=bulk_configuration,
        route=['L', 'G', 'X'],
        points_per_segment=50
        )
    nlsave(outfile, bandstructure)
```

Download all 3 scripts (`bulks.py`, `pbe.py`, `pps.py`) and run the latter two using either the **Job Manager** or from a terminal:

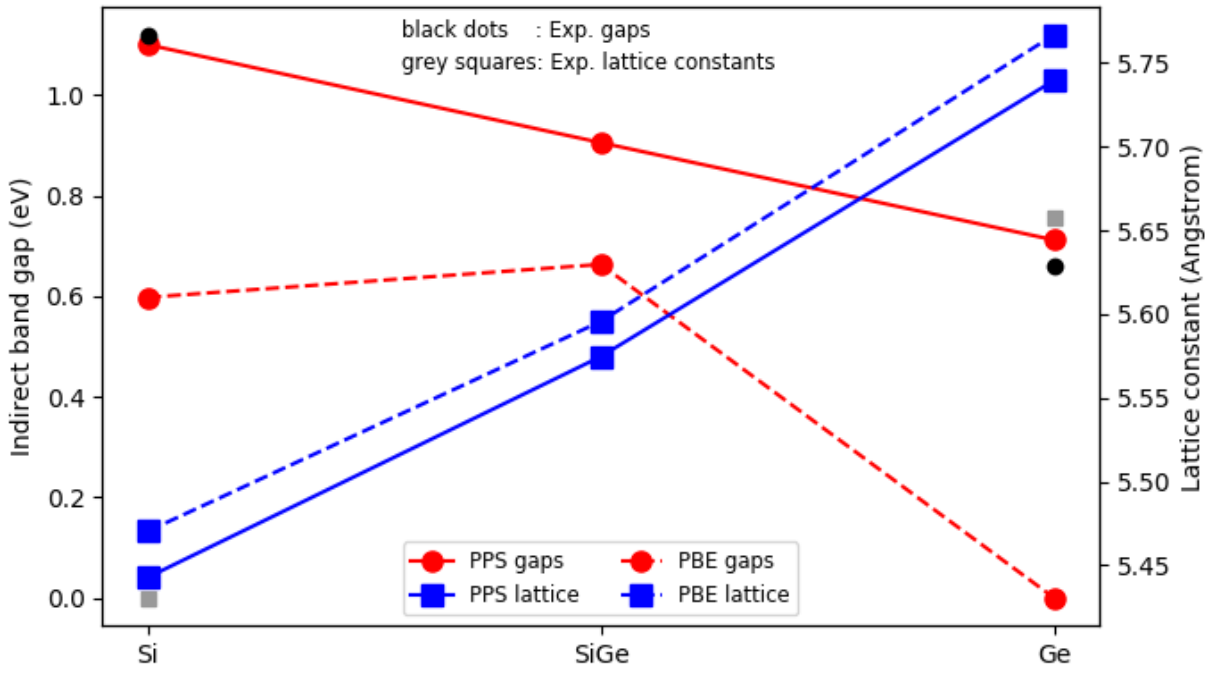
```
$ atkpython pps.py > pps.log
$ atkpython pbe.py > pbe.log
```

The jobs should take around 5 minutes each.

Use then the script `plot_pps.py` to plot the results. Running that script should produce the figure shown below, where the calculated indirect band gaps are shown in red circles, and the calculated lattice constants are shown in blue squares, both for PBE (dashed lines) and DFT-PPS (solid lines).

The DFT-PPS band gaps of Si and Ge compare very well to experiments (block dots; from Ref.[Page 650, 4]), and varies roughly linearly with germanium content. On the contrary, the ordinary PBE method predicts zero Ge band gap.

The DFT-PPS lattice constants of pure Si and Ge are also closer to experiments (grey squares) than found with non-corrected PBE.



Manually specifying DFT-PPS parameters

It can be very convenient to manually specify the DFT-PPS projector shift parameters. This may be especially useful in DFT-PPS calculations for elements where there are no default DFT-PPS parameters (only Si and Ge currently have defaults).

The projector shifts are supplied using an instance of the *PseudoPotentialProjectorShift* class, which is then given as an argument to the SG15 basis set. The following script manually sets the Si and Ge DFT-PPS parameters identical to the default ones:

```
#-----------------------------------------
# Basis Set
#-----------------------------------------
# Basis set for Silicon
SiliconBasis_projector_shift = PseudoPotentialProjectorShift(
    s_orbital_shift=21.33*eV,
    p_orbital_shift=-1.43*eV,
    d_orbital_shift=0.0*eV,
    f_orbital_shift=0.0*eV,
    g_orbital_shift=0.0*eV
    )
SiliconBasis = BasisGGASG15.Silicon_Medium(projector_shift=SiliconBasis_projector_shift)

# Basis set for Germanium
GermaniumBasis_projector_shift = PseudoPotentialProjectorShift(
    s_orbital_shift=14.55*eV,
    p_orbital_shift=0.20*eV,
    d_orbital_shift=-2.17*eV,
    f_orbital_shift=0.0*eV,
```

(continues on next page)

(continued from previous page)

```
    g_orbital_shift=0.0*eV
    )
GermaniumBasis = BasisGGASG15.Germanium_Medium(projector_shift=GermaniumBasis_projector_
↪shift)

# Total basis set
basis_set = [
    SiliconBasis,
    GermaniumBasis,
    ]
```

Warning: Choosing the appropriate DFT-PPS parameters may be a very delicate matter, and often requires a numerical optimization procedure. The SciPy software offers numerous such routines, but if you choose to use non-default DFT-PPS parameters, the quality of those parameters is entirely your own responsibility as a user!

QuantumATK does not offer support for optimizing DFT-PPS parameters; we recommend in general that users stick to the default parameters, if they exist.

References

3.2.15 Electrical characteristics of devices using the IVCharacteristics study object

Version: U-2022.12

Downloads & Links

IVCharacteristics Study objects `soi_device_configuration.py` `Workflow 1` `Workflow 2` `Workflow 3` `soi_device_ivc_through_scripting.py`

This tutorial introduces the *IVCharacteristics* **study object** used within the framework of the **Workflow Builder**. The *IVCharacteristics* study object enables the calculation and analysis of the most relevant electrical characteristics of field-effect transistor (FET) device models, including the on/off ratio (I_{on}/I_{off}), the subthreshold slope (SS), the transconductance (g_m) and the drain-induced barrier lowering (DIBL)[1].

In the present tutorial, you will learn how to use *IVCharacteristics* study object in the **Workflow Builder** to setup, calculate and analyze some of the electrical characteristics of a model silicon-on-insulator (SOI) device. These workflows will be demonstrated for a preconstructed SOI FET device modeled with the *DeviceSemiEmpiricalCalculator* adopted for the *IVCharacteristics* study object to calculate the SS, and DIBL parameter values and then benchmark them against the available experimental data.

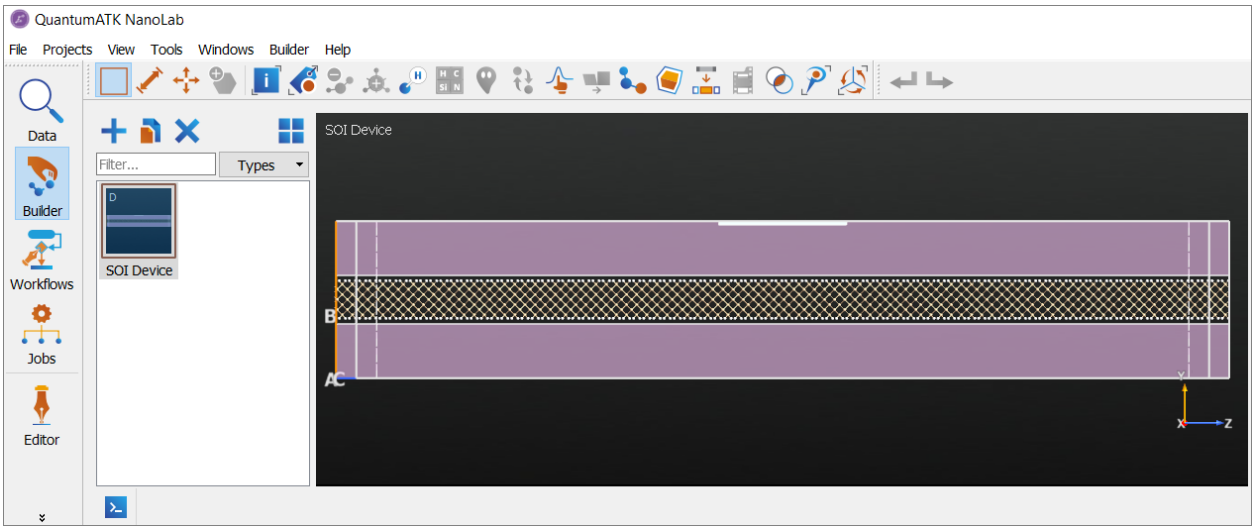
[1] S.ăM. Sze and K.ăN. Kwok. *Physics of Semiconductor Devices*. Wiley, 3rd edition, 2006.

Prerequisites

A general knowledge of the charge carrier transport in microelectronics devices and related IV characteristics is expected, e.g., IV curve, on/off ratio, subthreshold slope, transconductance, and drain-induced barrier lowering. If you are not familiar with these concepts you can still learn from this tutorial, but will not get the full benefit. If you wish to learn about them, we recommend reading this paper:[Page 659, 1].

Calculation and analysis of the $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve for the FET on-state

The SOI FET device model structure considered in the tutorial is shown in the figure below. It consists of a hydrogen-passivated **2D** silicon slab, which includes a Si channel region connected with an ideal contact to source and drain leads made of highly-doped Si; two continuum dielectric regions acting as the SiO_2 dielectric substrate or spacer (with $\epsilon = 3.9\epsilon_0$) between a continuum metal gate and atomistic Si channel region.



For the sake of computational efficiency, the electronic structure of the device model will be calculated using a semi-empirical tight-binding model[2].

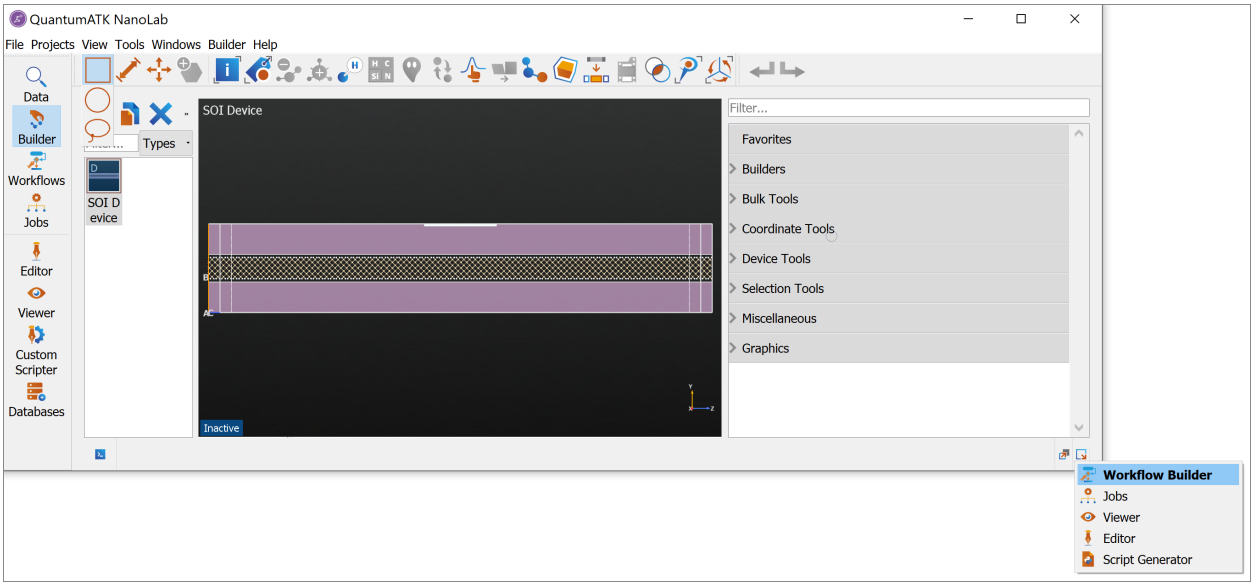
Setting up a Workflow

To create a workflow for *IVCharacteristics* calculations in the **Workflow Builder**, follow the step-by-step protocol described below. For a general introduction to the **Workflow Builder**, please see the dedicated guide: *Introduction to the Workflow Builder*.

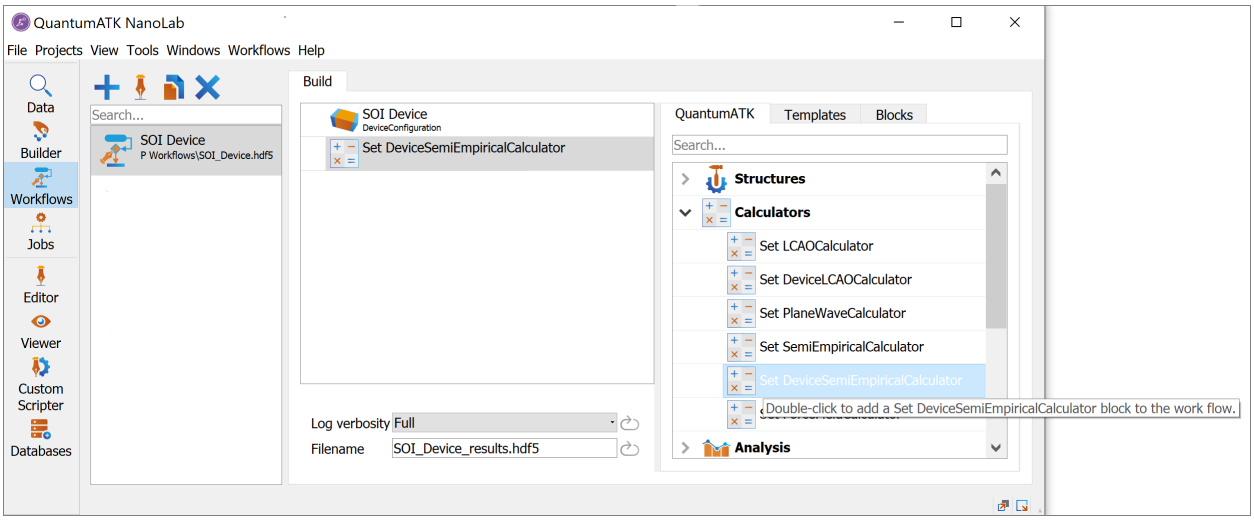
- To add the SOI FET device configuration (shown in the previous figure) to the NanoLab Builder, download the script `soi_device_configuration.py`, and drag-and-drop it on to the NanoLab Builder icon from the **Data View**. Or use the Add button in the NanoLab Builder to import the device structure from the script `soi_device_configuration.py`.

[2] T.ăB. Boykin, M.ăLuisier, M.ăSalmani-Jelodar, and G.ăKlimeck. Strain-induced, off-diagonal, same-atom parameters in empirical tight-binding theory suitable for [110] uniaxial strain applied to a silicon parametrization. *Phys. Rev. B*, 81:125202, 2010. doi:10.1103/PhysRevB.81.125202.

•In the NanoLab Builder, send the SOI `DeviceConfiguration` to the **Workflow Builder**.



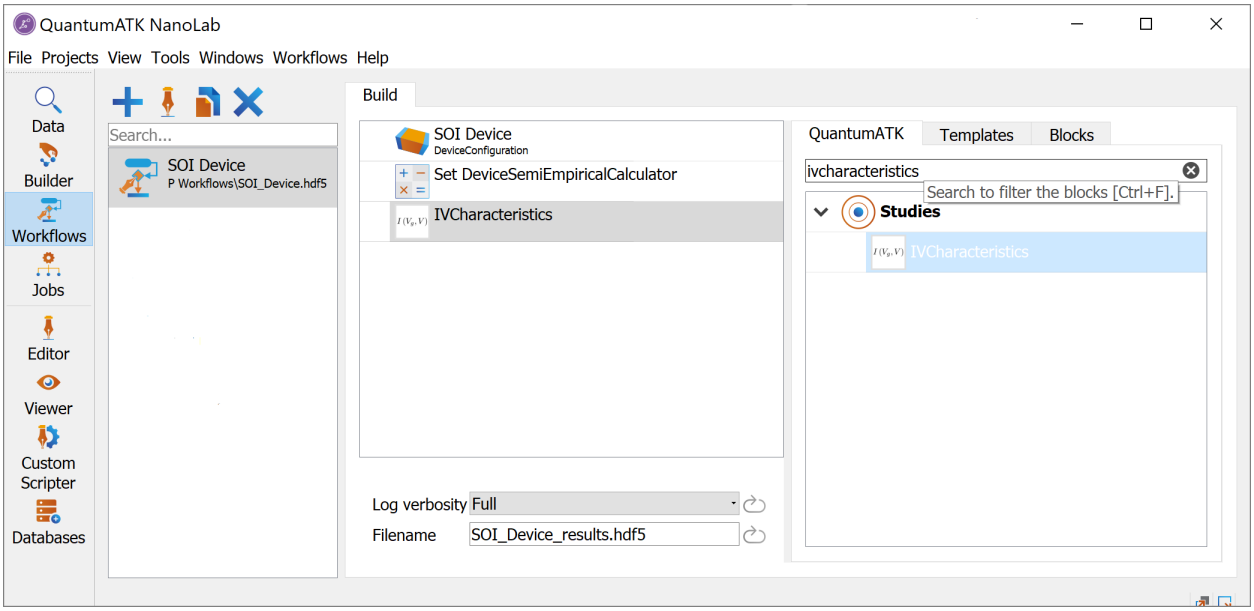
•In the **Workflow Builder** window, add a *Set SemiEmpiricalCalculator* to the *SOI Device* workflow.

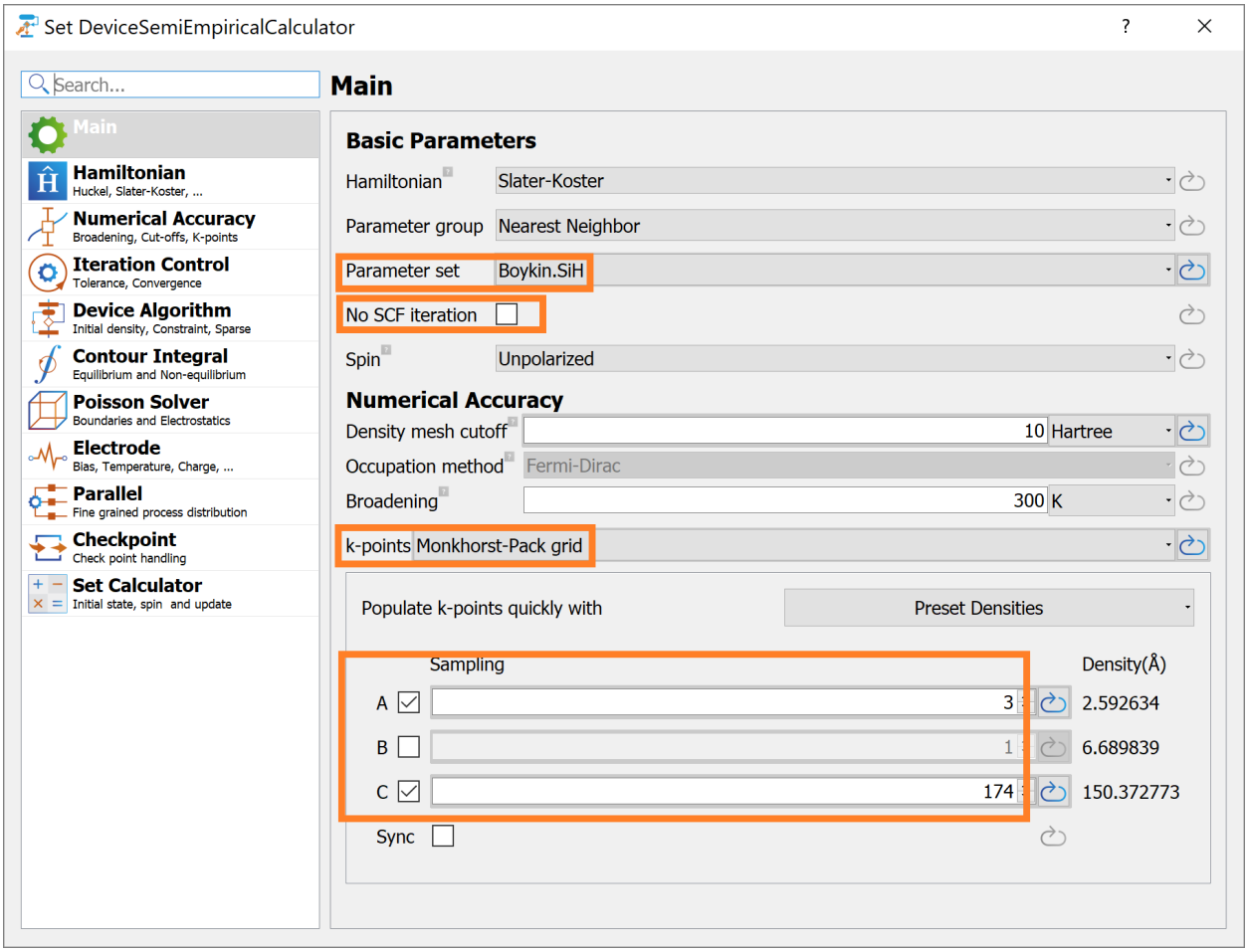


•In the QuantumATK Search bar, type *ivcharacteristics* to find the corresponding *IVCharacteristics* study $I(V_g, V)$ object and then add it to the workflow by double-clicking.

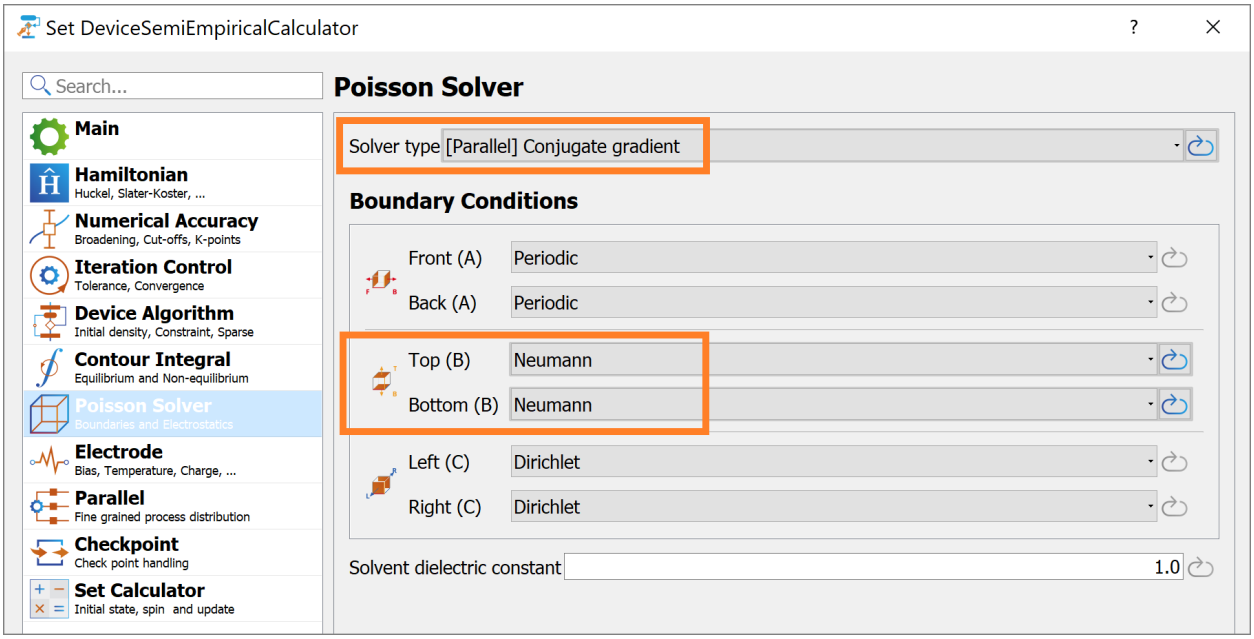
•In the Main section of the *Set SemiEmpiricalCalculator* block, change the *Parameter set* to *Boykin.SiH*, uncheck the *No SCF iteration* box, change *k-points* from *Density* to *Monkhorst-Pack k-grid*, and set the *k-point sampling* to 3x1x174.

•Change the **Poisson Solver** settings of the *Set SemiEmpiricalCalculator* block by switching to the [*Parallel*] *Conjugate gradient* Poisson solver, and set the top and bottom boundary conditions (in the B-direction) to Neumann,


QuantumATK NanoLab
File Projects View Tools Windows Workflows Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Custom Scripter
Databases
Search...
SOI Device
P Workflows\SOI_Device.hdf5
Build
SOI Device
DeviceConfiguration
Set DeviceSemiEmpiricalCalculator
IVCharacteristics
Log verbosity Full
Filename SOI_Device_results.hdf5
QuantumATK
Templates
Blocks
ivcharacteristics
Search to filter the blocks [Ctrl+F].
Studies
IVCharacteristics


Set DeviceSemiEmpiricalCalculator
Search...
Main
Hamiltonian
Huckel, Slater-Koster, ...
Numerical Accuracy
Broadening, Cut-offs, K-points
Iteration Control
Tolerance, Convergence
Device Algorithm
Initial density, Constraint, Sparse
Contour Integral
Equilibrium and Non-equilibrium
Poisson Solver
Boundaries and Electrostatics
Electrode
Bias, Temperature, Charge, ...
Parallel
Fine grained process distribution
Checkpoint
Check point handling
Set Calculator
Initial state, spin and update
Main
Basic Parameters
Hamiltonian Slater-Koster
Parameter group Nearest Neighbor
Parameter set Boykin.SiH
No SCF iteration
Spin Unpolarized
Numerical Accuracy
Density mesh cutoff 10 Hartree
Occupation method Fermi-Dirac
Broadening 300 K
k-points Monkhorst-Pack grid
Populate k-points quickly with
Preset Densities
Sampling
Density(Å)
A 3 2.592634
B 1 6.689839
C 174 150.372773
Sync

because there exists no periodicity in the direction perpendicular to the Si slab surfaces.



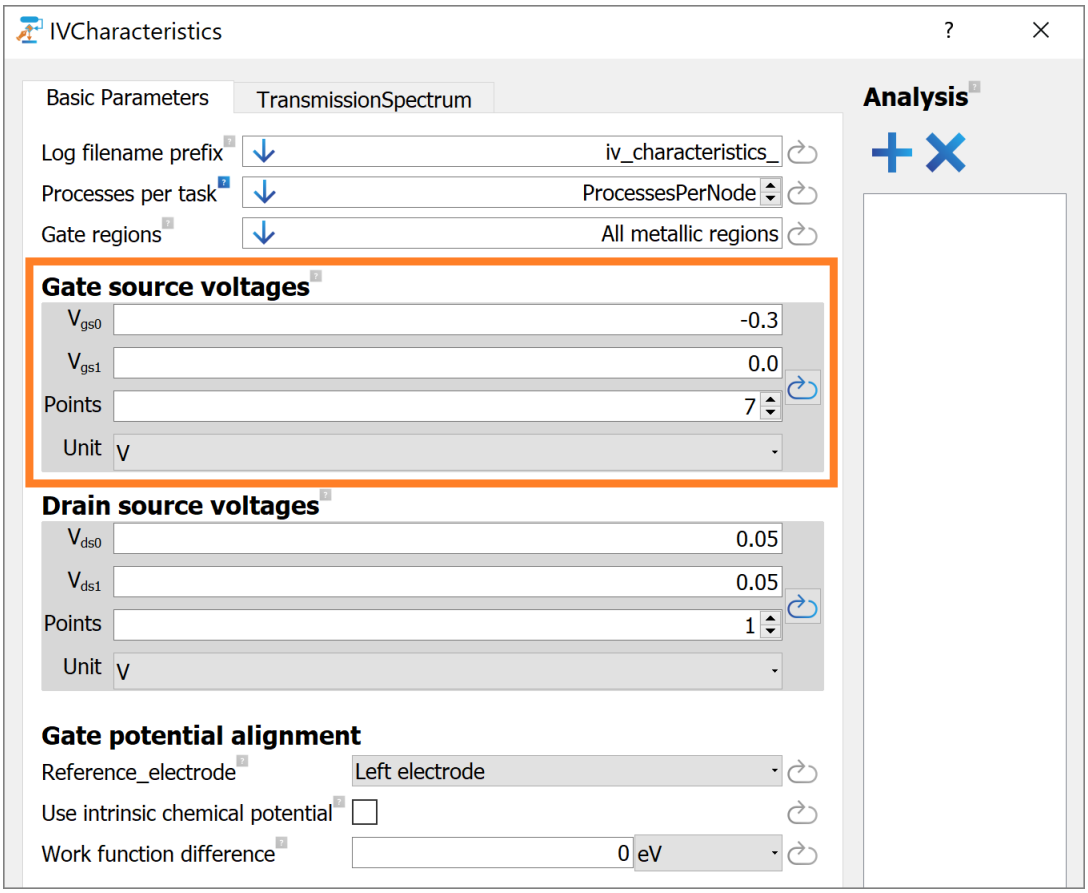
•Edit the *IVCharacteristics* settings in the workflow to set a range of gate voltage values for calculating the $I(V_g, V)$ $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve.

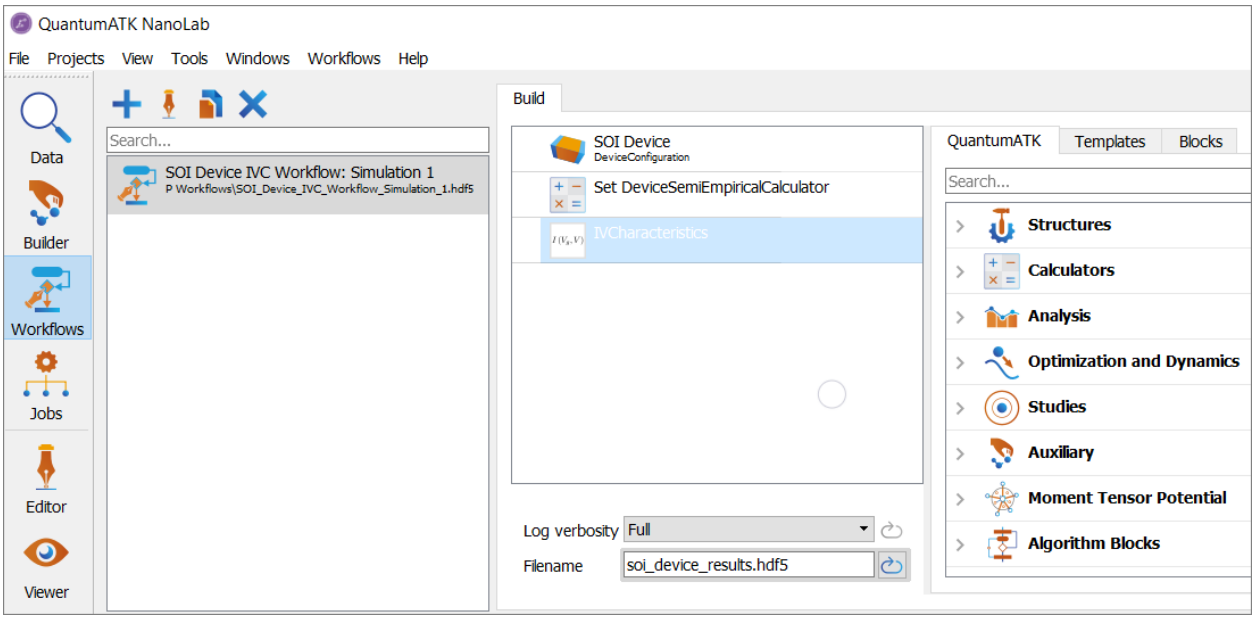
Note: Given these settings, the `IVCharacteristics` study object will do a scan over the gate-source voltage values from $\mathrm{V_{gs0}} = -0.3\mathrm{V}$ to $\mathrm{V_{gs1}} = 0.0\mathrm{V}$, at regular intervals of $\Delta\mathrm{V_{gs}} = 0.05\mathrm{V}$ and at a small drain-source bias voltage applied, $\mathrm{V_{ds}} = 0.05\mathrm{V}$.

Using a predefined Workflow

To verify if the *SOI Device* workflow in the previous section is created correctly, one can compare it to a predefined workflow, which can be downloaded at `Workflow 1`. To use the predefined workflow, one should place the corresponding workflow (hdf5) file in your project folder in the *Workflows* subfolder (create it, if missing), and it will then appear in the **Workflow Builder** window of the **QuantumATK NanoLab GUI** as follows.

The *SOI Device* workflow created in the previous section and predefined *SOI Device IVC Workflow: Simulation 1* are supposed to be identical. Rename the *SOI Device* workflow to *SOI Device IVC Workflow: Simulation 1* in the **Workflow Builder** window.


IVCharacteristics
Basic Parameters
TransmissionSpectrum
Analysis
Log filename prefix
iv_characteristics_
Processes per task
ProcessesPerNode
Gate regions
All metallic regions
Gate source voltages
V_{gs0}
-0.3
V_{gs1}
0.0
Points
7
Unit
V
Drain source voltages
V_{ds0}
0.05
V_{ds1}
0.05
Points
1
Unit
V
Gate potential alignment
Reference_electrode
Left electrode
Use intrinsic chemical potential
Work function difference
0
eV


QuantumATK NanoLab
File
Projects
View
Tools
Windows
Workflows
Help
Data
Builder
Workflows
Jobs
Editor
Viewer
Search...
SOI Device IVC Workflow: Simulation 1
P Workflows\SOI_Device_IVC_Workflow_Simulation_1.hdf5
Build
SOI Device
DeviceConfiguration
Set DeviceSemiEmpiricalCalculator
IVCharacteristics
Log verbosity
Full
Filename
soi_device_results.hdf5
QuantumATK
Templates
Blocks
Search...
Structures
Calculators
Analysis
Optimization and Dynamics
Studies
Auxiliary
Moment Tensor Potential
Algorithm Blocks

Running Workflows

To run the *SOI Device IVC Workflow: Simulation 1*, send it to the **Job manager** using the button, saving it, e.g., with a default filename *soi_device_results.py*. When submitting it to a *Local Machine*, then you might also need pressing the button in the **Job manager** to actually start running the calculation. This device calculation will take about 5 minutes using 40 MPI processes.

Once the device calculation is done (or started), go to the **Data View** (or to the working directory on a remote machine) and click on the log file `soi_device_results.log` in the *Data View* window (or use a text editor on the remote machine) to view the log information in the **Editor**. A list of all the planned tasks that are to be done by the *IVCharacteristics* study object can then be seen in the log, see the following figure with the first 3 (out of 14) workflow tasks highlighted in yellow.

```
+-------------------------------------------------------------------------------+
| IV Characteristics Study                                                      |
+-------------------------------------------------------------------------------+
| 14 task(s) will be executed.                                                  |
|                                                                               |
| * Update configuration                                                        |
|   Gate voltage: -0.3 V                                                        |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.05 V                                             |
| * Update configuration                                                        |
|   Gate voltage: -0.25 V                                                       |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.05 V                                             |
| * Update configuration                                                        |
|   Gate voltage: -0.2 V                                                        |
```

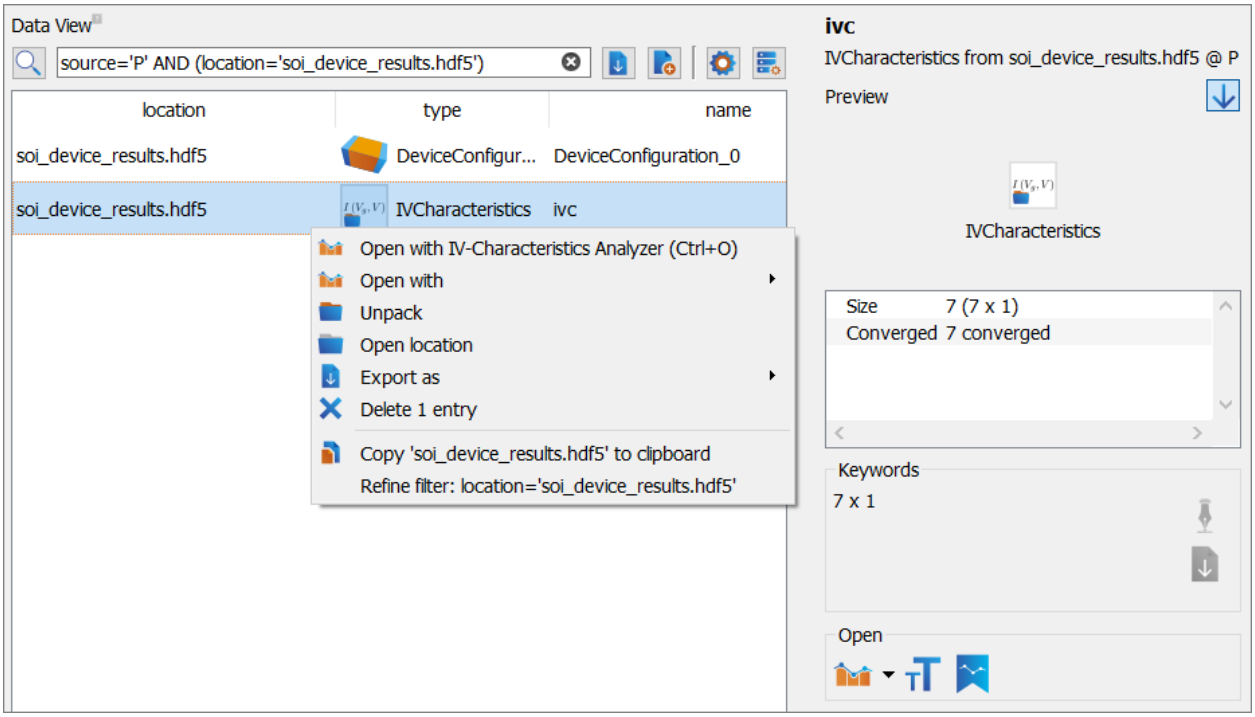
During the workflow execution, an additional information box will be printed out to the main log file for each started task of the workflow. In the following example of the main log file, there are 14 tasks to be executed, *task 1* is finished and *task 2* is in progress.

```
+-------------------------------------------------------------------------------+
| Executing task 1 / 14:                                                        |
|   Update configuration                                                        |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.05 V                                             |
|   Log to: iv_characteristics_Vgs_0.0_Volt_Vds_0.05_Volt.log                   |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Executing task 2 / 14:                                                        |
|   Calculate TransmissionSpectrum                                              |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.05 V                                             |
|   Log to: iv_characteristics_Vgs_0.0_Volt_Vds_0.05_Volt.log                   |
+-------------------------------------------------------------------------------+
```

Note: For each gate voltage point, the study object will create an additional log file with the usual NEGF-SCF convergence information, as well as information on related analysis object calculations, e.g., on the *Transmission Spectrum* calculation. We will not discuss that in more detail here.

Analyzing the Workflow results

Close the **Editor** and go back to the **Data View** of the **QuantumATK NanoLab GUI**. In the *Data Sources*, select the file `soi_device_results.hdf5` and then click on (or right-click to Open with) the $I\left(V_g, V\right)$ *IVCharacteristics* object in the *Data View* window to open the object, e.g., to view it with the *IV-Characteristics Analyzer*.

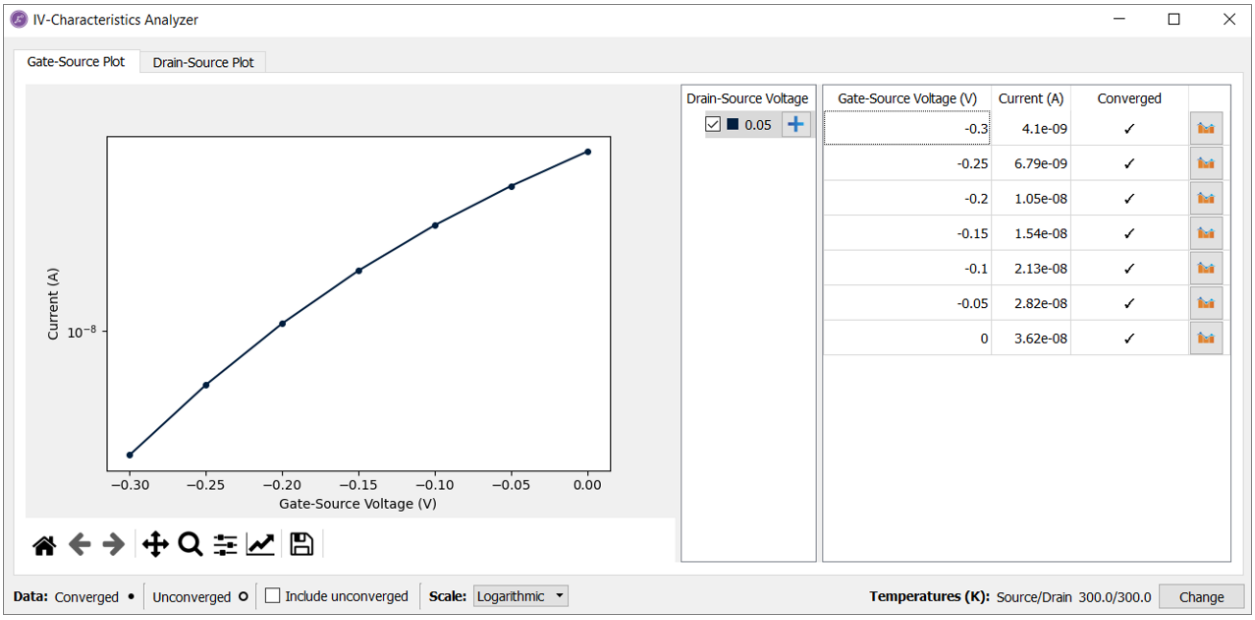


The *IV-Characteristics Analyzer* widget will then appear as follows.

The bottom part of the *IV-Characteristics Analyzer* widget contains the following supplemental information:

- **Data**: the convergence information for individual self-consistent device calculations at different voltages. In this case all bias points are converged.

Note: An unconverged point in the $\mathrm{I_{ds}}$ *vs.* $\mathrm{V_{gs}}$ data set means that the NEGF-SCF calculation did not reach the target convergence parameter value (which is set in the *Device Calculator*) for the corresponding gate voltage value and are thus unreliable. For tips on how to reach the convergence target, see the guidelines in the *NEGF Convergence Guide*.



•**Scale**: the plotting scale used on the Y-axis for the drain-source current $\mathrm{I_{ds}}$, which can be viewed as either *Linear* or *Logarithmic*.

•**Temperatures**: the electron temperature in the electrodes adopted to calculate the drain-source current $\mathrm{I_{ds}}$, using the `TransmissionSpectrum` calculated by the `IVCharacteristics` study object. Note that the temperature for each electrode can be modified by the user, and the current is then re-calculated accordingly.

The top part of the *IV-Characteristics Analyzer* shows the actual plots of the `IVCharacteristics` data, allowing one to switch between two different types of visualization:

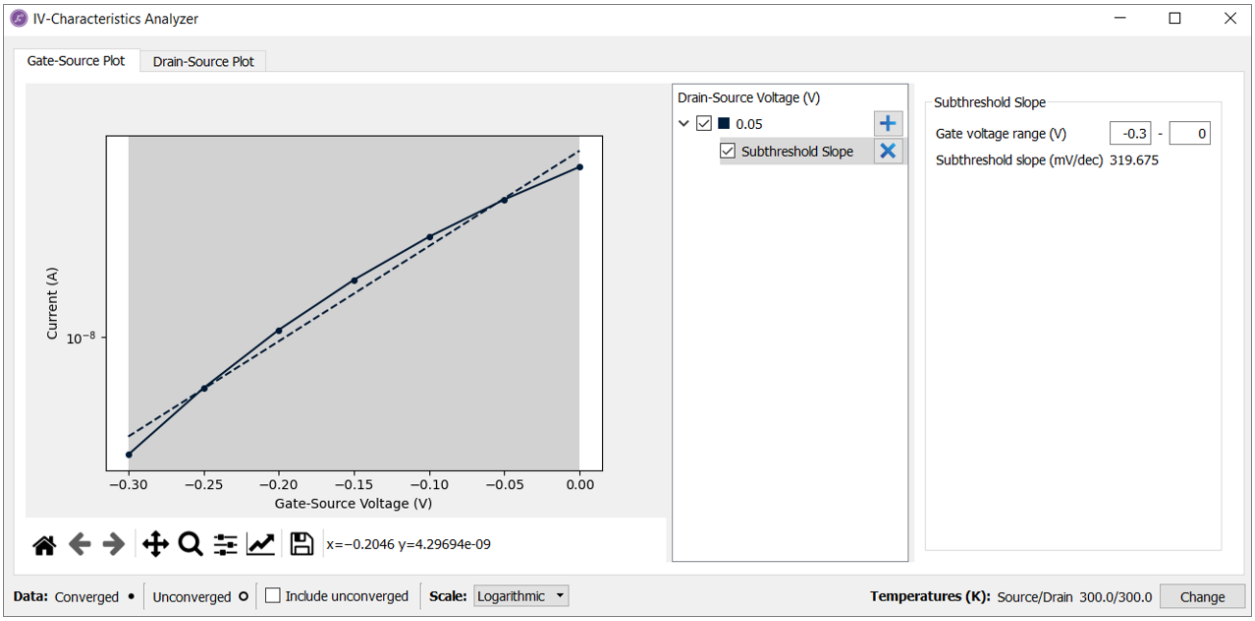
•*Gate-Source Plot* of the drain-source current vs. the gate-source voltage ($\mathrm{I_{ds}}$ $vs.$ $\mathrm{V_{gs}}$ curve),

•*Drain-Source Plot* of the drain-source current vs. the drain-source voltage ($\mathrm{I_{ds}}$ $vs.$ $\mathrm{V_{ds}}$ curve).

Given that a single bias voltage $\mathrm{V_{ds}}$ has been considered in this *Workflow 1*, only the $\mathrm{I_{ds}}$ $vs.$ $\mathrm{V_{gs}}$ characteristics will be analyzed at this time.

To calculate relevant characteristics of the FET device, e.g., the subthreshold slope SS, click on the + button in the *Drain-Source Voltage* section at the center of the panel and select the *Subthreshold Slope* option from the drop-down menu. The *Gate voltage range* over which the SS is calculated will be highlighted in gray in the $\mathrm{I_{ds}}$ $vs.$ $\mathrm{V_{gs}}$ plot. You can adjust this range to calculate the SS in the *Subthreshold Slope* panel on the right-hand side of the screen.

Note: For the obtained $\mathrm{I_{ds}} - \mathrm{V_{ds}}$ curve, the value of SS is highly overestimated. The SS value is also very sensitive to the actual choice of the *Gate voltage range*. This is because the curve section, which is used to compute the SS parameter here, lies outside the subthreshold region / FET off-state. In the latter, the current $\mathrm{I_{ds}}$ is supposed to vary linearly (on a logarithmic scale) with $\mathrm{V_{gs}}$. For a reliable calculation of the SS, the range of $\mathrm{V_{gs}}$ voltage values must therefore be extended to the subthreshold region, or, in other words, to the FET off-state, as done in the following section.



Extending the range of the $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve to the FET off-state

To sample the subthreshold region for the SOI FET device considered, the range of $\mathrm{V_{gs}}$ has to be extended to $-0.9\mathrm{V} \leq \mathrm{V_{gs}} \leq 0.0\mathrm{V}$. This can be achieved in two ways:

- *2A. Extending the gate-source voltage range through the QuantumATK NanoLab GUI in the Workflow Builder*
- *2B. Use scripting to add data points to the IVCharacteristics study object*

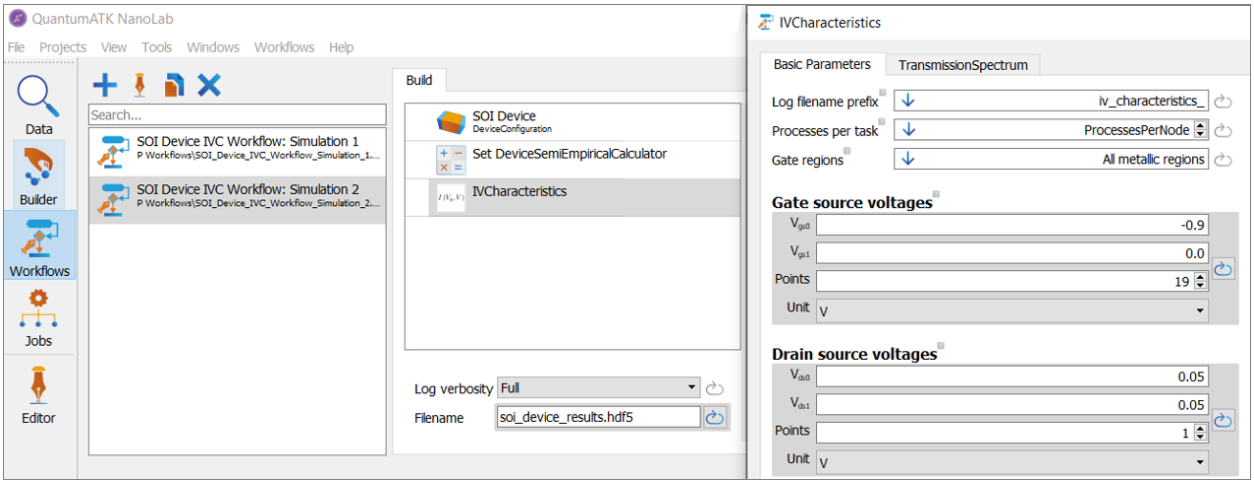
2A. Extending the gate-source voltage range through the QuantumATK NanoLab GUI in the Workflow Builder

To extend the gate-source voltage range in the *Workflow*, one can select the *SOI Device IVC Workflow: Simulation 1* (described in *Section 1*) and copy it with the *Duplicate* button, rename it to the *SOI Device IVC Workflow: Simulation 2* in the **Workflow Builder** window to then modify the **Gate source voltage range** settings in the `IVCharacteristics` study object as follows:

- Set the lower limit to $\mathrm{V_{gs0}} = -0.9\mathrm{V}$,
- Set the number of voltage points to 19 to also include the original gate voltage range (for the on-state), $-0.3\mathrm{V} \leq \mathrm{V_{gs}} \leq 0.0\mathrm{V}$, see also a note in the following.

Note: For the sake of comparison, the predefined *SOI Device IVC Workflow: Simulation 2* is enclosed to this tutorial and can be downloaded at `Workflow 2`.

To do the *Workflow 2* for the extended gate voltage range, send *SOI Device IVC Workflow: Simulation 2* to the **Job manager** using the button, save it as `soi_device_results.py` and press the button to run the corresponding job, if needed. The calculation will take about 5 minutes using 40 MPI processes.



Inspecting the log file of the follow-up *Workflow 2* calculation, one can notice that only the V_{gs} voltage points added to the original gate voltage range of *Section 1* have been calculated. It means that the `IVCharacteristics` study object automatically performs a check of which values of V_{gs} have been already considered, and run calculations only for the remaining values of V_{gs}.

It is important that:

- The output / results (hdf5) file obtained in *Section 1* is located in the same folder as the `soi_device_results.py` file,
- And the new output / results (hdf5) file for the *Workflow 2* calculation has the same filename as that used for the *Workflow 1* calculation in *Section 1*. In the present example, keep the *Results file* filename as *soi_device_results.hdf5*.

Note: The range of the gate-source voltage values and number of voltage points have been chosen such that the gate-source voltage values in the range of $-0.3\text{V} \leq V_{gs} \leq 0.0\text{V}$ are the same as those used in the section *Section 1*. This ensures that the study object will not repeat any calculation in that gate-source voltage range.

2B. Use scripting to add data points to the *IVCharacteristics* study object

There exists another option to instruct the `IVCharacteristics` study object to do calculations for additional values of V_{gs}. That can be done through Python scripting by adding the required voltage values by directly modifying the script, e.g., if the original workflow is missing or one cannot run the **QuantumATK NanoLab GUI** for some reason.

Note: An extensive list of all the commands available to interact with the IVCharacteristics study object is available in the `IVCharacteristics` entry of the reference manual.

To do this, click on (or right-click to open with) the script `soi_device_results.py` in the **Data View** window (or just open this script in some text editor) to open it the **Editor**. Add the line highlighted in yellow in the *IVCharacteristics* block.

```
iv_characteristics = IVCharacteristics(
    configuration=soi_device,
    filename='soi_device_results.hdf5',
```

(continues on next page)

(continued from previous page)

```
    object_id='ivc',
    gate_regions=None,
    gate_source_voltages=numpy.linspace(-0.3, 0.0, 7)*Volt,
    drain_source_voltages=numpy.linspace(0.05, 0.05, 1)*Volt,
    gate_potential_alignment=gate_potential_alignment,
    energies=None,
    kpoints=kpoints,
    kpoints_weights=None,
    self_energy_calculator=self_energy_calculator,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    log_filename_prefix='iv_characteristics_',
    number_of_processes_per_task=ProcessesPerNode,
    device_transmission_method=device_transmission_method,
    enforce_zero_transmission_in_band_gap=True,
    source_electrode=Left,
    max_initial_state_configurations=3
)
iv_characteristics.addVoltages(gate_source_voltages=numpy.linspace(-0.9, -0.35, 12)*Volt)
iv_characteristics.update()
```

To run the device calculation, send the script to the **Job manager** using the button, save it as `soi_device_ivc_through_scripting.py`, and press the button in the **Job manager**, if needed. The calculation will take about 5 minutes on 40 MPI processes. You can also download the full script at `soi_device_ivc_through_scripting.py`.

Note: In general, we recommend to save *Workflows* and use them in the **Workflow Builder**, as described in *Introduction to the Workflow Builder*. This allows for a higher flexibility in adjusting simulation settings and the simulation workflow itself, e.g., by adding new or removing old workflow steps through the **QuantumATK NanoLab GUI** in a simpler manner than in a script.

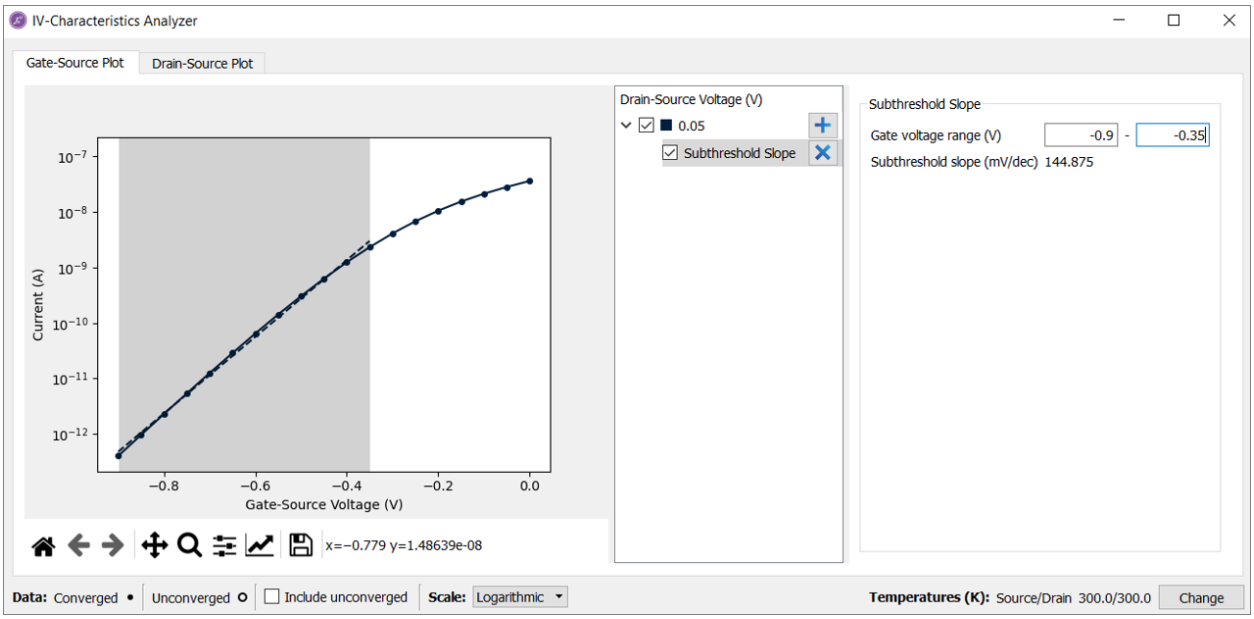
Analysis of the $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve in the subthreshold region

Once the device calculations are done, you are ready to analyze the characteristics of the device and extract the subthreshold slope SS, using the appropriate set of data, which includes the subthreshold region / FET off-state.

Similarly to the discussion in *Section 1*, click on (or right-click to open with) the *IVCharacteristics* object $I\left(V_g, V\right)$ contained in the file `soi_device_results.hdf5` present in the **Data view** window to open the *IV-Characteristics Analyzer* widget.

This time, the *IV-Characteristics Analyzer* will show an $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve exhibiting a virtually linear behavior with respect to $\mathrm{V_{gs}}$ in the gate voltage range of $-0.9\mathrm{V} \leq \mathrm{V_{gs}} \leq -0.35\mathrm{V}$. This indicates that this range of $\mathrm{V_{gs}}$ corresponds to the subthreshold region / FET off-state. Approximating the $\mathrm{I_{ds}} - \mathrm{V_{gs}}$ curve with a linear fit allows for extracting the subthreshold slope SS.

Click on the + button in the **Drain-Source Voltage** section at the center of the panel and select the *Subthreshold Slope* option from the available options in the drop-down menu. Adjust the *Gate voltage range* in the *Subthreshold Slope* panel to include only the subthreshold region by setting it from $-0.9\mathrm{V}$ to $-0.35\mathrm{V}$, as shown in the following figure.



The calculated value of the subthreshold slope is now $\mathrm{SS} = 145\ \mathrm{meV/dec}$, being of the same order of magnitude as a measured value of $\mathrm{SS}^{\mathrm{exp}} = 95\ \mathrm{meV/dec}$[3]. The discrepancy is related to a simplified model for the SOI FET device adopted in this tutorial for the sake of basic workflow demonstration.

Calculating the drain-induced barrier lowering

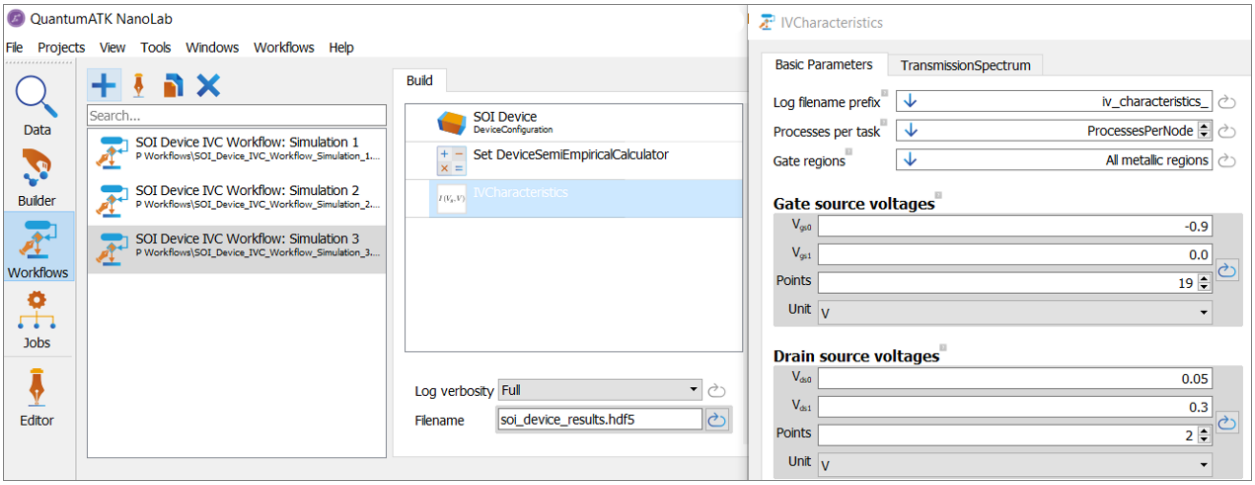
To calculate the drain-induced barrier lowering (DIBL), one needs to obtain a second $\mathrm{I_{ds}}\ vs.\mathrm{V_{gs}}$ curve at a somewhat higher value of the source-drain bias voltage, $\mathrm{V_{ds}} = 0.3\mathrm{V}$, for example.

To extend the range of the source-drain voltage values in the study object, one can select the *SOI Device IVC Workflow: Simulation 2* (described in *Section 2A*) and copy it with the *Duplicate* button, and rename it to the *SOI Device IVC Workflow: Simulation 3* in the **Workflow Builder** window to then modify the **Drain-source voltage range** settings in the `IVCharacteristics` study object as follows:

- Set the upper limit of the drain-source voltage range to $\mathrm{V_{ds1}} = 0.3\mathrm{V}$,
- Set the number of points to 2.

Note: For the sake of comparison, the predefined *SOI Device IVC Workflow: Simulation 3* is enclosed to this tutorial and can be downloaded at `Workflow 3`.

[3] Shinji Migita, Yukinori Morita, Meishoku Masahara, and Hiroyuki Ota. Electrical performances of junctionless-FETs at the scaling limit (LCH=3nm). *Proceedings of the Electron Device Meeting (IEDM) 2012*, 2012. URL: http://ieeexplore.ieee.org/document/6479006, doi:10.1109/IEDM.2012.6479006.



To do the *Workflow 3* for the extended gate voltage range, send *SOI Device IVC Workflow: Simulation 3* to the **Job manager** using the button, save it as `soi_device_results.py` and press the button to run the corresponding job, if needed. The calculation will take about 10 minutes using 40 MPI processes.

It is important that:

- The output / results (hdf5) file obtained in *Section 2A* is located in the same folder as the `soi_device_results.py` file,
- And the new output / results (hdf5) file for the *Workflow 2* calculation has the same filename as that used for the *Workflow 1* calculation in *Section 2A*. In the present example, keep the *Results file* filename as *soi_device_results.hdf5*.

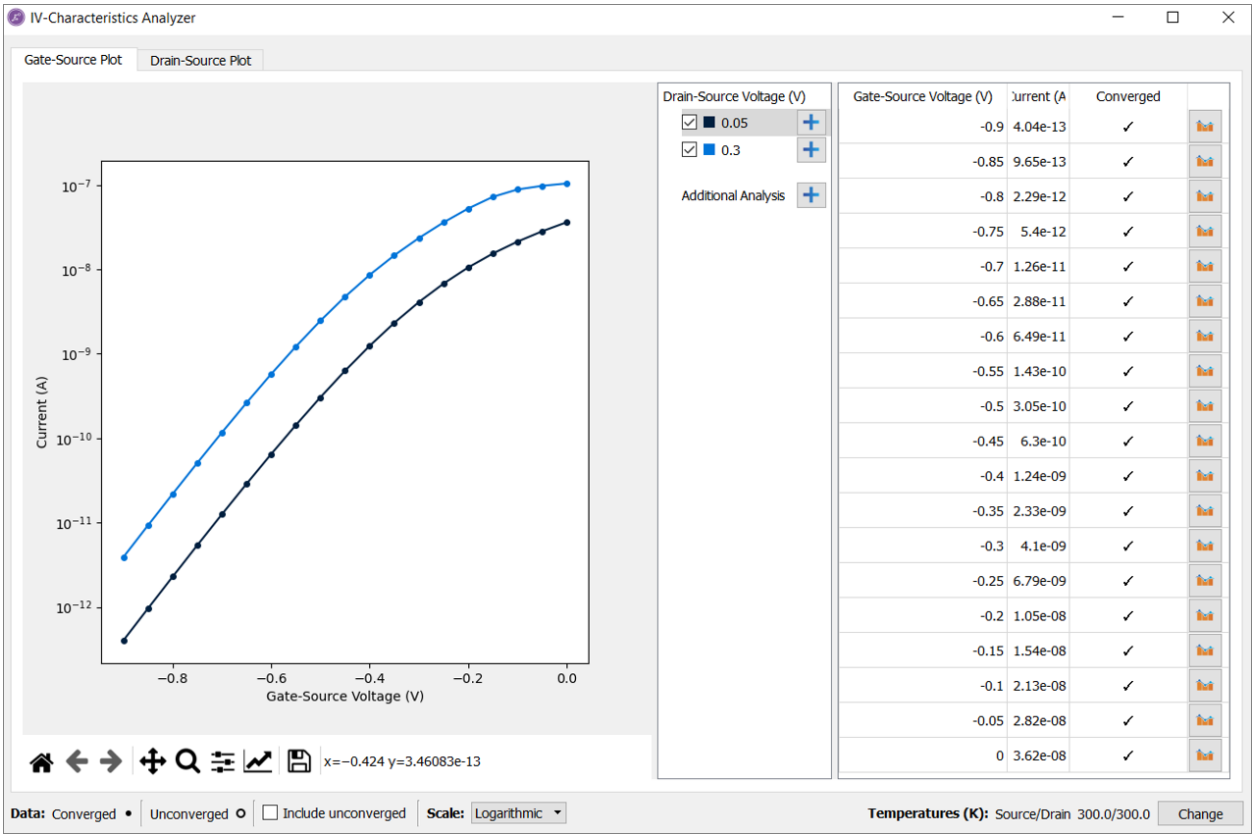
Note: By inspecting the log file, you will notice how, thanks to the smart handling of pre-existing calculations, the **IV Characteristics** object has calculated only the curve at $\mathrm{V_{ds}} = 0.3\mathrm{V}$, using the points calculated at $\mathrm{V_{ds}} = 0.05\mathrm{V}$ as initial states until there are $\mathrm{V_{ds}} = 0.3\mathrm{V}$ points available.

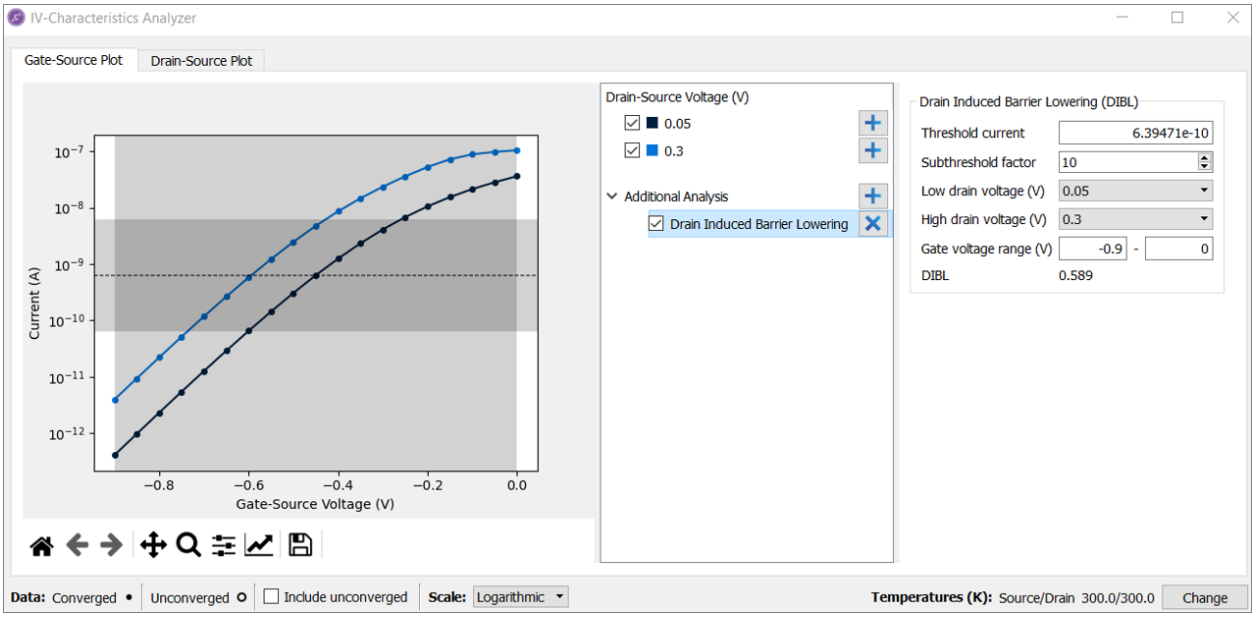
When the calculation is done, select the $I\left(V_g, V\right)$ *IVCharacteristics* object contained in the file `soi_device_results.hdf5` in the **Workflow Builder** window, and click on the *IV-Characteristics Analyzer* plugin in the right-hand side of the **Quantum ATK NanoLab GUI window**. The *IV-Characteristics Analyzer* plugin will now show also the two additional curves calculated at $\mathrm{V_{ds}} = 0.3\mathrm{V}$, in addition to that previously calculated at a lower bias voltage of $\mathrm{V_{ds}} = 0.05\mathrm{V}$, as shown in the following figure.

To calculate the DIBL parameter, click on the button next to the *Additional Analysis* option in the *Drain-Source Voltage* section at the center of the panel, and select *Drain Induced Barrier Lowering* from the drop-down menu. The calculated value of DIBL will be shown in the **Drain Induced Barrier Lowering** section, as shown in the figure below.

Note: Notice that the DIBL value of 0.589 is given as unitless quantity, which corresponds to $589\ \mathrm{mV/V}$ as often reported in literature.

There are a number of parameters that can be tuned in the **Drain Induced Barrier Lowering** panel to tune the calculation of the DIBL. In particular, it is worth mentioning that:


IV-Characteristics Analyzer
Gate-Source Plot
Drain-Source Plot
Current (A)
Gate-Source Voltage (V)
Drain-Source Voltage (V)
0.05
0.3
Additional Analysis
Gate-Source Voltage (V)
Current (A
Converged
-0.9 4.04e-13 ✓
-0.85 9.65e-13 ✓
-0.8 2.29e-12 ✓
-0.75 5.4e-12 ✓
-0.7 1.26e-11 ✓
-0.65 2.88e-11 ✓
-0.6 6.49e-11 ✓
-0.55 1.43e-10 ✓
-0.5 3.05e-10 ✓
-0.45 6.3e-10 ✓
-0.4 1.24e-09 ✓
-0.35 2.33e-09 ✓
-0.3 4.1e-09 ✓
-0.25 6.79e-09 ✓
-0.2 1.05e-08 ✓
-0.15 1.54e-08 ✓
-0.1 2.13e-08 ✓
-0.05 2.82e-08 ✓
0 3.62e-08 ✓
x=−0.424 y=3.46083e-13
Data: Converged • Unconverged ○ Include unconverged Scale: Logarithmic
Temperatures (K): Source/Drain 300.0/300.0 Change


IV-Characteristics Analyzer
Gate-Source Plot
Drain-Source Plot
Current (A)
Gate-Source Voltage (V)
Drain-Source Voltage (V)
0.05
0.3
Additional Analysis
Drain Induced Barrier Lowering
Drain Induced Barrier Lowering (DIBL)
Threshold current 6.39471e-10
Subthreshold factor 10
Low drain voltage (V) 0.05
High drain voltage (V) 0.3
Gate voltage range (V) -0.9 - 0
DIBL 0.589
Data: Converged • Unconverged ○ Include unconverged Scale: Logarithmic
Temperatures (K): Source/Drain 300.0/300.0 Change

- The *Threshold current*, which is the value of the current corresponding to a value of V_{gs} equal to the threshold voltage, is set by default at the mid-point between the maximum and the minimum values of the current I_{ds}, but can be modified by the user.
- The *Subthreshold factor* determines the current range used to fit the subthreshold characteristics of the device to accurately determine the threshold voltage. A range containing at least 3 points is then requested. If not enough points are included, the DIBL will not be calculated and an error message will be shown instead.
- The *Gate voltage range* can be modified to include only that part of the curve for which the DIBL should be calculated. This can be important for device models showing bipolar characteristics, where one has two subthreshold regions.

Warning: The value of the DIBL calculated for the FET device considered in the present tutorial is significantly larger than that measured experimentally. This can be ascribed to the high sensitivity of the DIBL parameter value to the doping profile and the geometrical characteristics of the device. For a FET device with an ultra-short metal gate length, one can expect the DIBL parameter value between 0.1 and 1, as obtained in the present example.

References

3.2.16 Formation energies and transition levels of charged defects

Version: O-2018.06

Downloads & Links

`bulk-diamond.py` `bulk-boron.py` `boron_in_diamond.py` *`ChargedPointDefect`*

In this tutorial you will learn how to calculate the formation energies of neutral and charged point defects, and the transition levels between stable charge states for a given defect centre. Specifically, we will work with the substitution of boron in diamond. We will use the **ChargedPointDefect** study object, to set up and analyze the calculations.

If you are unfamiliar with this topic, you might also want to consult the introductory tutorial on formation energies, which focuses on bulk systems and neutral defects: *Calculation of Formation Energies*.

Note: The **ChargedPointDefect** study object was introduced in the O-2018.06 release, and is the recommended way of studying charged point defects in QuantumATK. Users of previous versions can still do the calculations and analysis in a more complex, and manual, workflow, as described here: *Formation energies of charged defects - manual workflow*

Procedure for calculating the formation energy

The formation energy E^f of a defect X is defined as the energy difference between the investigated system and the components in their reference states. This is analogous to the formation energy of a bulk material, but for charged defects it is less clear how to choose the reference states. Here we follow the procedure outlined in the review by C. Freysoldt *et al.*[1], originally formualted by Zhang and Nrthrup[2]:

$$E^f[X^q] = E_{tot}[X^q] - E_{tot}[bulk] - \sum_i n_i\mu_i + q\left(E_{VBM} + \mu_e\right) + E_{corr}.$$

$E_{tot}[X^q]$ is the total energy of the host crystal with the defect with charge q, which is given in units of the elementary charge ($e > 0$), $E_{tot}[bulk]$ is the total energy of the same cell of crystal without the defect, and $n_i\mu_i$ is the reference energy of n_i added, or substracted with a change of sign, atoms of element i at chemical potential μ_i. The term in parenthesis accounts for the chemical potential of the electron(s) involved in charging the defect. E_{VBM} is the valence band maximum as given by the QuantumATK band structure calculation for the bulk material of the study (diamond in this tutorial), and μ_e is the electron chemical potential defined here with respect to the top of the corresponding valence band. The μ_e parameter can then be treated as a free parameter, allowing to account for a shift of the Fermi level, e.g., due to doping. Note that $\mu_e = E_{gap}/2$ corresponds to the undoped semiconductor case, where E_{gap} is the intrinsic semiconductor band gap. Finally, E_{corr} is a sum of relevant correction terms. In our case, it corrects for electrostatic interactions between periodic images of the charged defects and the shift in the electronic bands from the introduction of the defect, using the scheme of Freysoldt et al.[3]. Due to the long-ranged nature of the Coulomb interaction, it can be quite difficult to work with systems that are large enough not to need this correction. The **ChargedPointDefect** study object allows you to scale the system in an easy way, and see how the formation energy converges with system size. More details of the correction scheme can be found in the *Appendix*.

In order to calculate the formation energy, we need to compute each of the terms outlined above. The total workflow is listed below:

•Select the compound and defect to be investigated, and the computational model to be used (exchange-correlation, pseudopotentials, etc.).

•Relax the pristine bulk system using the chosen computational model.

•Determine the chemical potential of the elements in the defect. In this case we use the total energy of the bulk for the element added substitutionally.

•Setup the undefected supercell to calculate $E_{tot}[bulk]$ and E_{VBM}.

•Introduce the (charged) defect and relax the system to obtain $E_{tot}[X^q]$.

•Calculate the correction terms in E_{corr} and finally the formation energy.

The latter three, and most complicated items, will be handled by the study object. These three items must also be repeated for each supercell size, and the latter two also for each charge state. In comparison, the first three items must only be done once.

[1] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and ChrisăG. Van De Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86(1):253–305, 2014. doi:10.1103/RevModPhys.86.253.

[2] S.ăB. Zhang and JohnăE. Northrup. Chemical potential dependence of defect formation energies in GaAs: Application to Ga self-diffusion. *Phys. Rev. Lett.*, 67:2339–2342, Oct 1991. doi:10.1103/PhysRevLett.67.2339.

[3] Christoph Freysoldt, Jörg Neugebauer, and ChrisăG. Van De Walle. Fully Ab Initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, 2009. doi:10.1103/PhysRevLett.102.016402.

Setting up the calculation

Initial calculations - optimized bulk diamond, dielectric constant and boron chemical potential

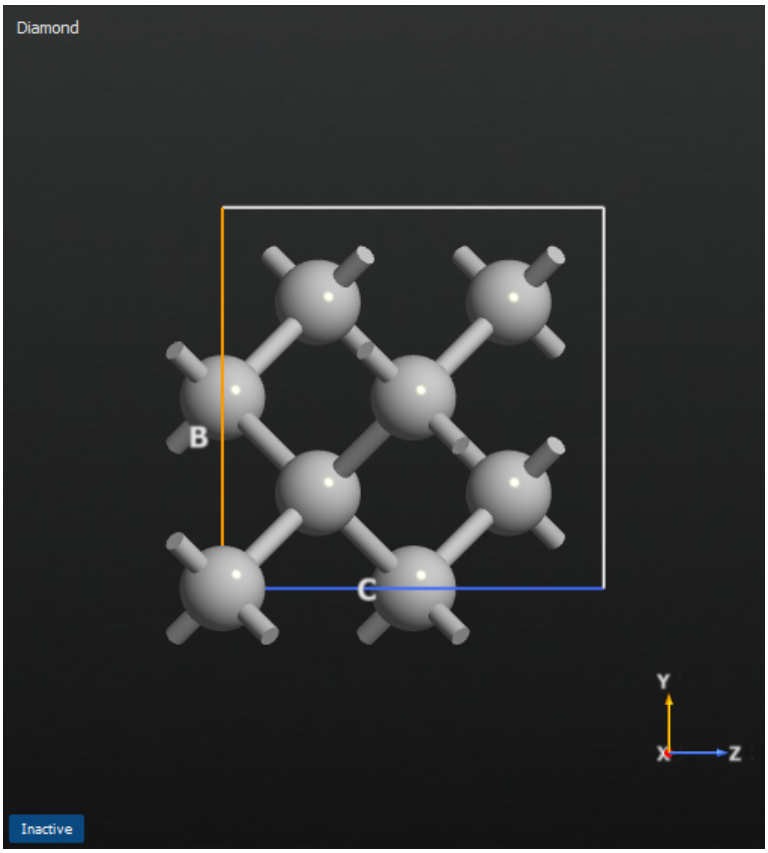
We first need to do some small initial calculations, to generate the input structure and the parameters required for the calculation of the formation energies, including corrections:

•Optimized structure of diamond

•Dielectric constant of diamond

•Chemical potential of boron atoms, from bulk boron

In order to make the interpetration easier when analyzing the results, we will use the cubic/conventional form of the unit cell of diamond. It can be generated from the primitive cell by using the function: *Bulk Tools* → *Supercell* tool in the Builder.



We will do the first two calculations in the same script. Set up a script to optimize the geometry of diamond and then calculate the optical spectrum. Use the following changes from default in the script for diamond:

•**New calculator**: k-point density of 4 Å.

•**Optimize geometry**: Untick *Constrain lattice vectors* and fix fractional coordinates.

•**Optical Spectrum**: k-point density of 12 Å and 100 bands above and below the Fermi level.

For boron, set up a script to optimize the geometry and calculate the total energy. Make the following changes from default values:

•**New Calculator**: k-point density of 4 Å and a density mesh cutoff of 55 Ha.

•**Optimize geometry**: Untick *Constrain lattice vectors* and fix fractional coordinates.

As the focus of this tutorial is on the **ChargedPointDefect** study object, you can save time and download the scripts for the preliminary calculations here: `bulk-diamond.py` and `bulk-boron.py`. However, if you are unsure of how to do these basic calculations, please see the appropriate tutorials to familiarise yourself with the concepts.

Tip: If you need more guidance on how to set up these calculations, please see these tutorials:

- *Geometry optimization: CO/Pd(100)*
- *Optical Properties of Silicon*
- *Molecular builder*

Extracting initial parameters

We get the dielectric constant from the OpticalSpectrum analysis object. Open the **OpticalSpectrum** object $\epsilon_{i,j}(\hbar\omega)$ on the LabFloor in Text Representation, and copy the dielectric function value for the xx-component at zero energy. It should be approximately 6.3.

In this tutorial, we will use bulk boron as the energetic reference for the substitutional boron atom. This means that the chemical potential for boron is the total energy per atom of bulk boron. Open the **TotalEnergy** object E_{Total} on the LabFloor in *Text Representation*, copy the energy value and divide it by 12. The result should be approximately -80.0 eV.

The ChargedPointDefect study object

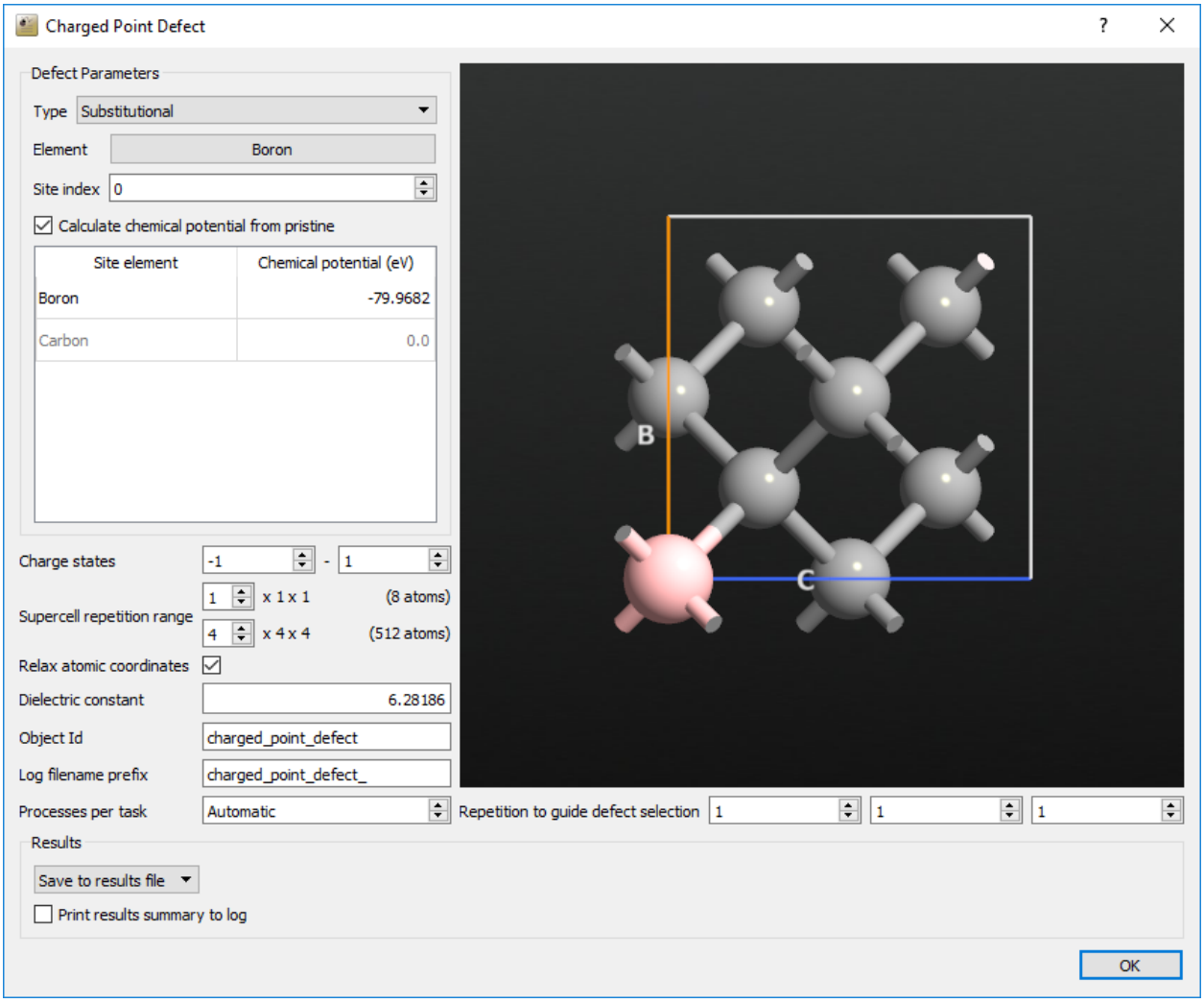
We will now set up the full calculation of the `ChargedPointDefect` study object. Drag the optimized structure onto the **Builder** and send it on to the **Script Generator**. Add a **New Calculator** and a **ChargedPointDefect** study object.

The calculator should be set up with the same parameters as used for the geometry optimization of bulk diamond in the previous section. We are now ready to set up the study object:

- Change **Type** from *Vacancy* to *Substitutional*.
- Change **Element** from *Indium* to *Boron*.
- Type in the desired value of the **chemical potential** for boron, which is *-79.96822167 eV* in this case.
- Change **Charge states** to *-1* as the lower bound and *+1* as the higher bound.
- Change the upper limit of **Supercell repetition range** to *4x4x4*.
- Change the **Dielectric constant** to *6.28186*.

Warning: For substitutional defects, the **ChargedPointDefect** study object automatically assigns the default pseudopotential and basis set to the substitutional element. In this tutorial, this is what we wish to use, and therefore no additional steps are needed. However, if you wish to use a non-default basis set, you should manually add it to the calculator, either in the Script Generator or in the script.

The window should now look like this:



Save the script and run it on your desired cluster. In our case, it took about 13 hours to run on a total of 32 cores. If you encounter any problems, you can compare your script against the one we have provided here: `boron_in_diamond.py`

Warning: From the P-2019.03 release the **ChargedPointDefect** study object was moved to the SMW package. In order to run `boron_in_diamond.py` with this version it is therefore necessary to add

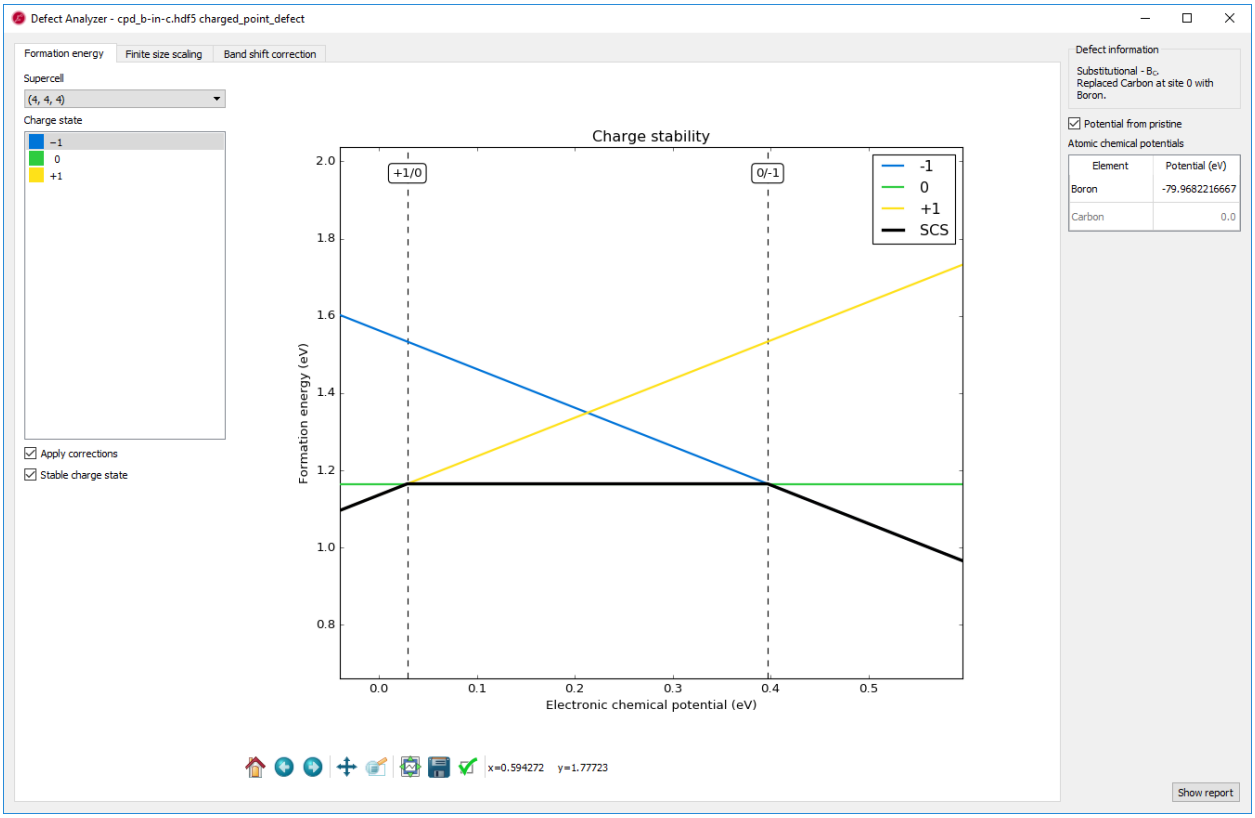
```
from SMW import *
```

at the top of the script.

Tip: The study objects contain automatic restart functionality. If the script does not complete for some reason, simply run it again. It will automatically read the status from the `.hdf5`-file, and set about completing the remaining tasks.

Analyzing the results

We are now ready to analyze the results of our calculations. Select the **ChargedPointDefect** object on the $E_f^q(\Delta\mu_e)$ LabFloor. In the right-hand list, open the **Defect Analyzer**. You should now see a window looking like this:

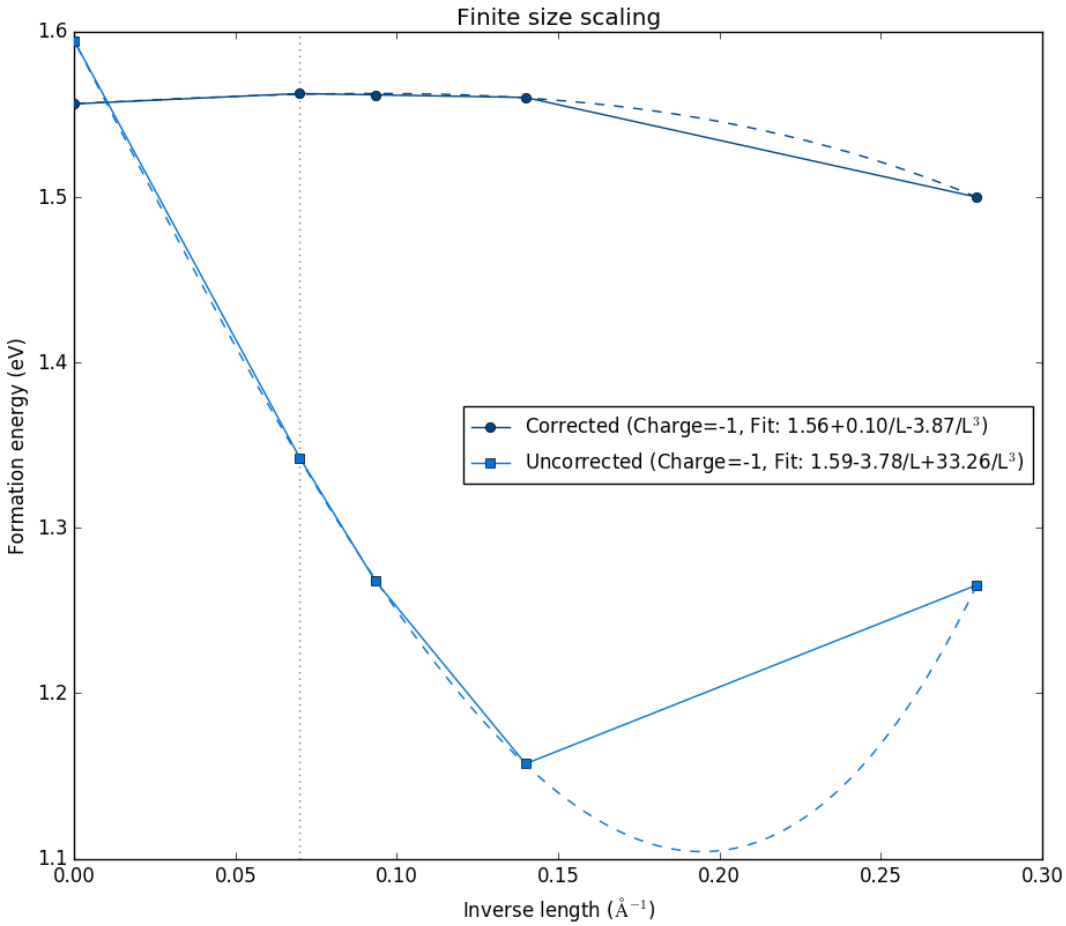


This is the formation energy for each charge state of the defect, as a function of the chemical potential of the electrons, relative to the valence band maximum. The colored lines each represent a charge state, the solid black line shows the most stable charge state at a given electronic chemical potential, and the dotted lines show the transitions between different charge states. In this case, the +1 state is stable only for very low values of the electron chemical potential, while the neutral state is most stable up to about 0.4 eV. If the electronic chemical potential is higher than that, the -1 charge state is most stable.

On the left, you can choose which supercell to base the plot on, including the extrapolation to infinity, and which charge state to highlight. You can also choose whether or not to include the corrections and whether the black line should be shown or not. On the right, you can change the chemical potentials of the elements.

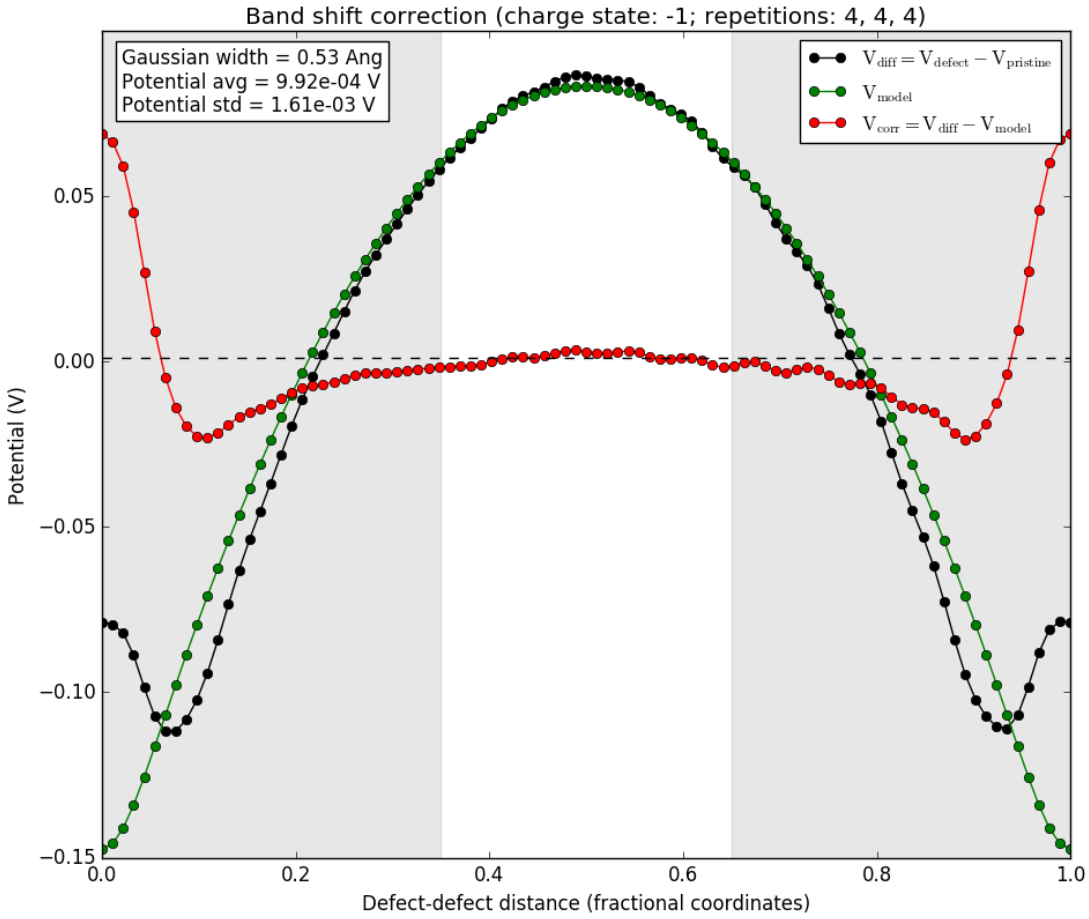
Charge state -1

We will now look more closely at each of the charge states, starting with the -1 state. Click on the tab titled **Finite size scaling**. The main panel in the window should now look like this:



The figure shows the formation energy as a function of the inverse length of the supercell, defined as the the cubic root of the volume. It shows both with the correction terms (dark blue), and without corrections (light blue). In both cases, they are extrapolated to infinite supercell size, with the fits shown as dashed lines. Notice how they both converge to approximately the same formation energy value, and how the formation energy values with the corrections included are already converged to the extrapolated value for the second-smallest 2x2x2 supercell. This shows that this charge state is well-localized within the 2x2x2 supercell, and the correction scheme, which mitigates finite-size effects on the charged defect formation energy, is valid for this particular charged defect state.

Next, click on the tab titled **Band shift correction**. The window should now look like this:



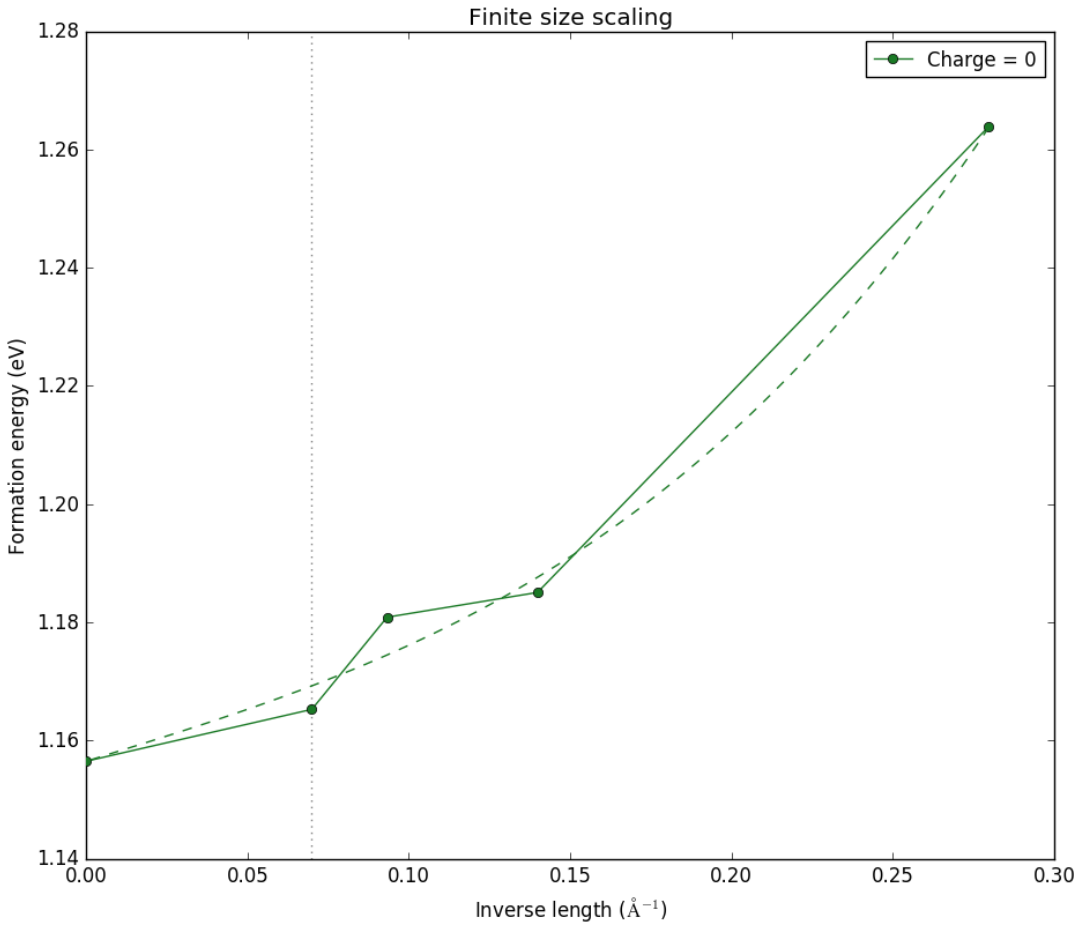
The black line is the difference between the potentials of the supercell with the defect and the pristine supercell without the defect. The green line is the potential from a Gaussian model charge placed at the defect position (the origin of the plot), and the red line is the difference between the two. The light area in the middle is the region where the red line is averaged to produce the value of the band shift term of the correction. The red line should be flat in this region for the correction term to be appropriately calculated. If this is not the case, the assumptions of the correction scheme are violated and it does not apply. In this case, however, the defect charge is well-localized and the red line is flat in the region where the bandshift is evaluated.

You may also select the other supercell sizes, and see how the defect charge is well-behaved, even for the 2x2x2 supercell. However, it is not well-localized for the very smallest 1x1x1 supercell, as expected. This supercell is much too small to give sensible results, even with the correction sceheme, and is included here only for comparison.

This behavior is not surprising, as the boron atom has one less (valence) electron than carbon atoms. Therefore, the -1 charge state leads to the boron atom being electronically equivalent to a carbon atom, stabilizing it as a substitutional defect.

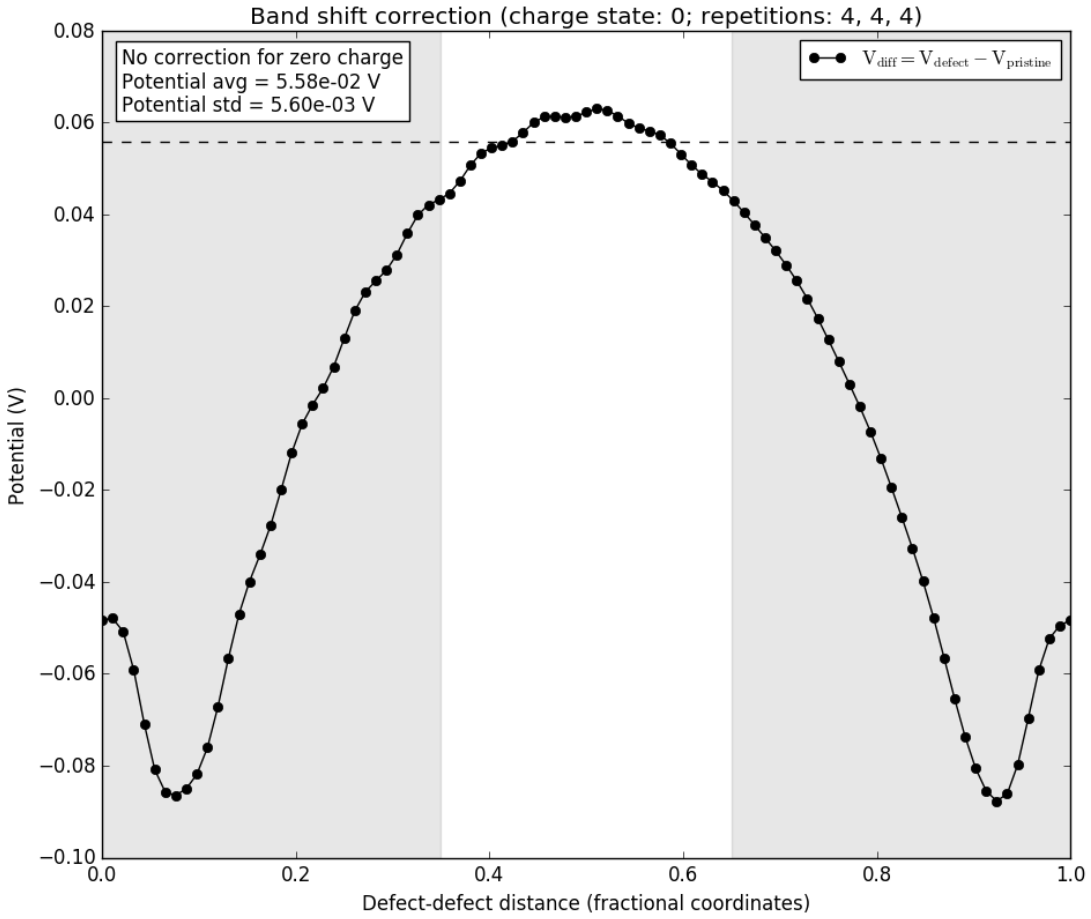
Charge state 0

We now move on to the neutral defect. Go again to the **Finite size scaling** tab, and change the charge state to *0*. Select also the *4x4x4* supercell if you have changed away from that. The window should now look like this:



Note that there is now only one data-set, as the corrections only apply if the system is charged. We see also that the variation in formation energy with size is significantly smaller than before. Here, the total range is about 0.1 eV, compared with 0.4 eV before. This is expected, as the neutral defect has no explicit charge that can suffer from finite-size effects. However, there can still be, e.g., elastic interactions between the defects, and it is therefore expected to see some size dependence, as we do here.

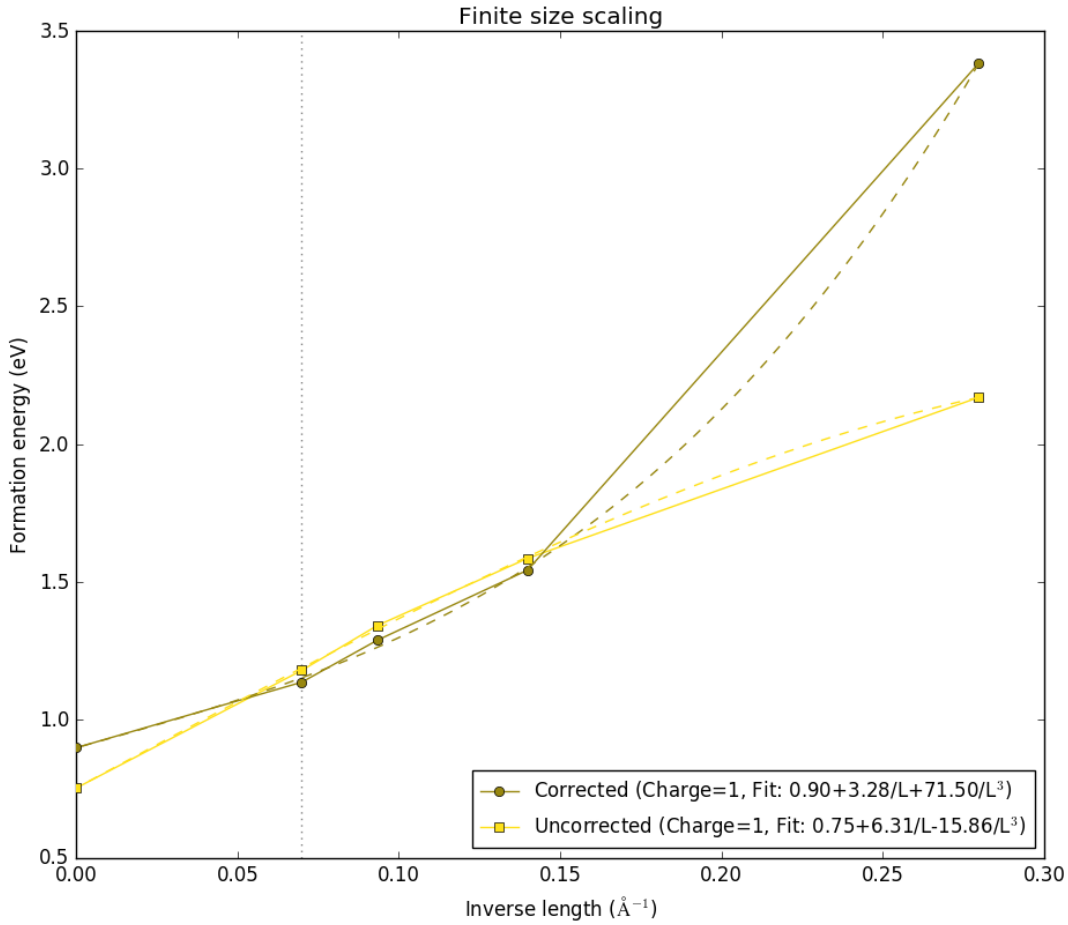
For completeness, we will take a short look at the **Band shift correction** tab. Click it, and the window should now look like this:



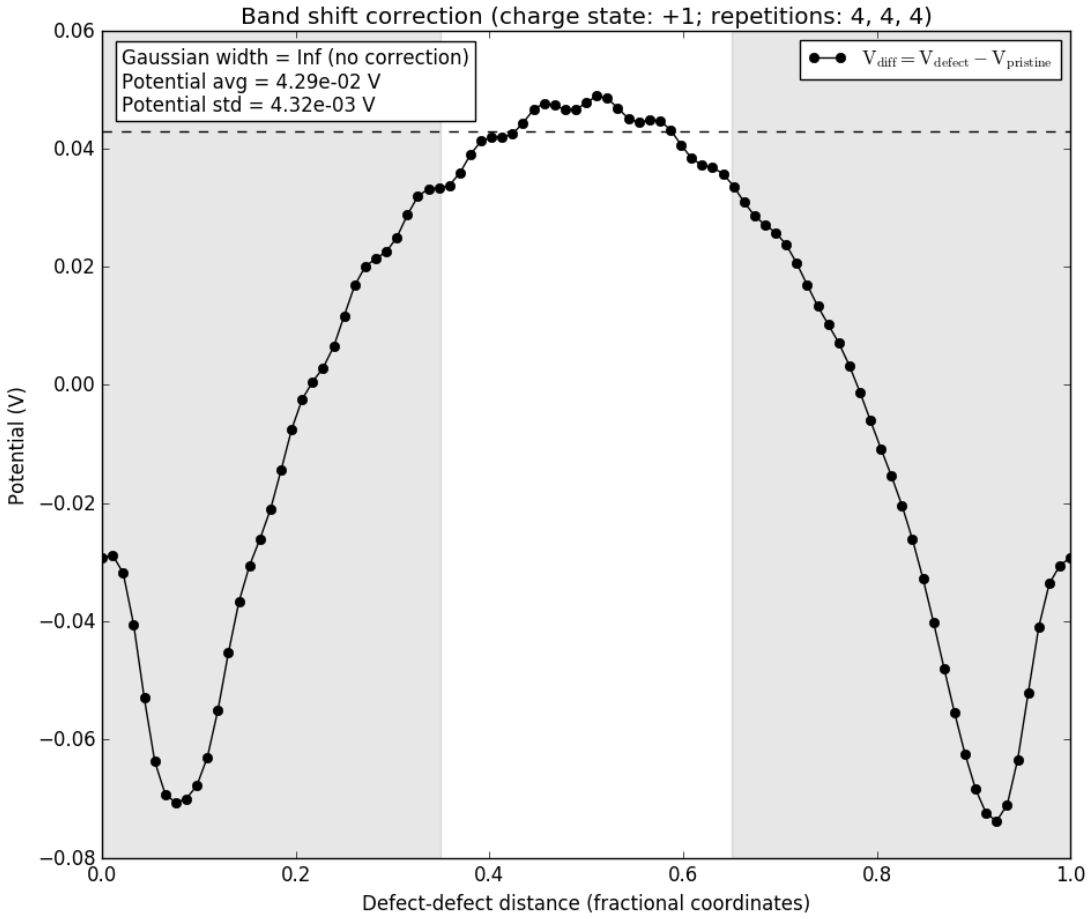
Only the black line is present as there is no charge to model, and hence no difference to investigate. Note also the box in the upper left, which informs that there is no correction for the neutral defect.

Charge state +1

Finally, we turn our attention to the +1 charge state. Select again the **Finite size scaling** tab, and change the charge state to *+1*. Select also the *4x4x4* supercell if you have changed away from that. The window should now look like this:



In this case it is very hard to see at first glance which data-set is the raw data and which includes the corrections, in contrast to what we saw for the -1 charge state. They still seem to converge to approximately the same value, but the corrections do not really add any useful information in this case. This indicates either a defect that cannot be modelled using the correction scheme, or a defect that is simply unphysical. In both cases, it will often be a delocalized charge. In order to look further at this charge state, select the tab **Band shift correction** again. The window should now look like this:



Note the box in the upper left again. We see that the Gaussian width is described as infinite, and no corrections are therefore applied, as they require a finite width of the charge distribution.

In this case, the best approach is to use as large a supercell as possible, removing the need for corrections and the accompanying approximations and assumptions.

Discussion and summary

Boron is a p-type dopant in group-IV semiconductors such as diamond. As mentioned above, this is because boron has one less valence electron than carbon, and so its effect as a substitutional defect in the diamond lattice is to add an effective positive charge (i.e., a hole) to the crystal. This also makes boron an acceptor dopant, as it will attract an electron from the crystal to completely fill its covalent bonds with its neighbours in the lattice.

When the neutral boron dopant captures an electron it becomes ionized. The ionization energy is readily measured experimentally, and corresponds to the stable-charge transition level between the 0 and -1 charge states shown in the main plot of the Defect Analyzer. The experimental value is 0.37 eV[4], in good agreement with our calculated value of 0.4 eV.

The +1 charge state, instead, is not experimentally relevant; this is again confirmed by our results, which show it to be

[4] Klaus Thonke. The boron acceptor in diamond. *Semiconductor Science and Technology*, 18(3):S20, 2003. URL: http://stacks.iop.org/0268-1242/18/i=3/a=303.

almost completely absent as a stable charge state within the band gap, as the transition level between +1 and 0 is close to the valence band maximum.

In this tutorial, we have used the **ChargedPointDefect** study object to set up and analyze a calculation of the formation energies for various charge states of boron substitution in diamond. QuantumATK includes a correction scheme that can mitigate the errors due to periodic interaction between the defects in finite supercells, if the defect is well-localized. In general, and especially if the defect charge is not well-localized, charged defects are best studied with larger supercells, and we have seen how the convergence with respect to size can be studied.

Appendix

The periodic correction scheme

As already mentioned, we use the correction scheme proposed by Freysoldt *et al.*[Page 675, 3] to correct for the electrostatic interaction between the periodic images of the charges and the band shift caused by introduction of the defect. This correction contains two terms:

$$E_{corr} = E_{lat} - q\Delta V_{q/b},$$

where E_{lat} corrects for the interaction between the periodic images of the charge and $q\Delta V_{q/b}$ corrects for the band shift. E_{lat} is calculated by considering the electrostatics of a model charge distribution, which is fitted to the actual calculation by the study object, and comparing the potential in the periodic cell with the potential where the boundary conditions are not periodic:

$$E_{lat} = \int_{\Omega} \left[\frac{1}{2} q(\tilde{V}_q^{lr} - V_q^{lr}) \right] d^3r.$$

V_q^{lr} is the long-range potential of the model charge distribution and $\tilde{V}_q^{lr}$ is the corresponding quantity in the periodic cell and shown in green. $\Delta V_{q/b}$ is calculated from the shift in the electrostatic potential due to introduction of the defect:

$$\Delta V_{q/b} = \tilde{V}_{q/b} - \tilde{V}_q^{lr},$$

where $\tilde{V}_{q/b}$ is the difference between the electrostatic potential in the pristine bulk configuration and with the defect added and shown in black. $\Delta V_{q/b}$ is shown in red, and should be evaluated far from the defect, where it is constant, in order to get the correct offset in the zero-point reference energy . This is shown by the lightly colored region in the plots above.

Note that the obtained difference is defined between two *screened* potentials, see the reference[Page 675, 3]. The dielectric screening is described by the static dielectric constant computed in the preliminary calculations.

References

3.2.17 Introduction

The papers investigate the accuracy of a range of different *DFT: LCAO* electronic-structure methods in calculations for bulk semiconductors. In particular, we apply GGA and MGGA exchange-correlation functionals, as well as the *Pseudopotential Projector-Shift* method. Some comparisons to *HSE* results are also included.

Both structural and electronic quantities are computed and the different methods are benchmarked against each other. Computational settings such as *k*-point sampling and density mesh cutoff energy are also considered.

Note: Thepapers are continously expanded with more data and more semiconductors.

All calculations are done using the newest updates (August 2016) to the *SG15/PseudoDojo* pseudopotentials and basis sets. The applied exchange-correlation functionals usually include PBE, PBEsol, TB09-MGGA, and PBE augmented with the PPS method. Each combination of pseudopotential, basis set, and exchange-correlation functional constitutes a **computational method**.

The following **structural and electronic quantities** are considered:

•Lattice constant(s)

•Elastic constants

•Band structure

•Band gaps

•Effective masses

•Dielectric constant(s)

Extensive **convergence studies** are conducted for the different computational methods. In particular, the convergence of the lattice constant, bulk total energy and band gap is thoroughly studied with respect to k-point sampling and density mesh cutoff energy.

3.2.18 Methods

The *SG15/PseudoDojo* normconserving pseudopotentials were recently implemented in the *DFT: LCAO* calculator engine. This suite of pseudopotentials was generated using the Optimized Norm-Conserving Vanderbilt (ONCV) method, as described in[1] and[2]. More information is available on the SG15-ONCV website.

As explained in the document *Pseudopotentials and basis sets available in QuantumATK*, three different QuantumATK basis sets are available for each element when using the SG15 pseudopotentials, with increasing accuracy; **Medium**, **High**, and **Ultra**. All three derive from the numerical atom-centered basis sets of the FHI-aims package, but have been significantly modified and optimized with respect to computational speed with the ATK-DFT calculator.

The Medium basis set is default for the SG15 pseudopotentials, and should be sufficient for most applications. However, if extreme accuracy is needed, the High and Ultra basis sets add more basis functions at the expense of increased computational load. The Ultra basis set is exceptionally large, and will usually add only little extra accuracy as compared to the High basis set, but at a fairly high extra computational cost. The Medium and High basis sets will in most cases constitute a better trade-off between computational accuracy and speed, so these are the two SG15 basis sets used in the papers.

HSE

The HSE06 screened hybrid density functional[3] is implemented in the FHI-aims package. The Tier 1 basis set is used with both Light and Tight accuracy settings, here denoted l1 and t1, respectively.

[1] D.ăR. Hamann. Optimized norm-conserving Vanderbilt pseudopotentials. *Phys. Rev. B*, 88:085117, Aug 2013. doi:10.1103/PhysRevB.88.085117.

[2] M.ăSchlipf and F.ăGygi. Optimization algorithm for the generation of ONCV pseudopotentials. *Computer Physics Communications*, 196:36–44, 2015. doi:10.1016/j.cpc.2015.05.011.

[3] A.ăV. Krukau, O.ăA. Vydrov, A.ăF. Izmaylov, and G.ăE. Scuseria. Influence of the exchange screening parameter on the performance of screened hybrid functionals. *Journal of Chemical Physics*, 2006. doi:10.1063/1.2404663.

Nomenclature

Each computational method is specified by a combination of exchange-correlation functional, pseudopotential, and basis set. Since this study uses only the SG15 pseudopotentials, this is omitted from the naming of a method. For example, a PBE calculation with the Medium basis set is simply denoted PBE-m. If a calulation with some method was not done at the lattice constant computed with that method, it is indicated in a paranthesis, e.g. PBEsol(PBE)-h for PBEsol-h at the PBE-h lattice constant.

Note: The ATK-DFT density mesh cutoff and *k*-point sampling used in a calculation are not contained in the naming scheme described above. One single choice of mesh cutoff and *k*-point sampling is determined in a convergence study.

Pseudopotential Projector-Shift

The Pseudopotential Projector-Shift (PPS) method for ATK-DFT was introduced with QuantumATK 2017. The method introduces shifts to the angular momentum projector channels used for constructing the pseudopotential, which affects the theoretical predictions obtained with standard exchange-correlation functionals, e.g. band gaps and equilibrium lattice constants. The projector shifts are essentially free parameters, which can be fitted against relevant data and used in ATK-DFT calculations, e.g. to obtain better band gaps with otherwise ordinary PBE.

The projector-shifted pseudopotential, $V_{\mathrm{ps}}(\mathbf{r})$, is written

$$V_{\mathrm{ps}}(\mathbf{r}) = V_{\mathrm{ps}}^{\mathrm{loc}}(\mathbf{r}) + \sum_{\alpha\beta} |\alpha\rangle [V_{\alpha\beta} + U_{\alpha\beta}] \langle\beta|,$$

where $V_{\mathrm{ps}}^{\mathrm{loc}}$ is the local part of the pseudopotential, $V_{\alpha\beta}$ the pseudopotential coupling matrix, and $U_{\alpha\beta}$ the pseudopotential projector-shift matrix. The sum runs over projectors α and β.

All entries in the matrix $U_{\alpha\beta}$ are zero when performing a standard GGA calculation, while the PPS method adds projector shifts along the matrix diagonal. These entries correspond to different angular momentum channels; *s*, *p*, *d*, *f*, etc. Only the *s* and *p* channels are relevant for silicon, which has no *d*-electrons, while both *s*, *p* and *d* are relevant for other elements, e.g. germanium. Choosing the projector shifts to use for a particular exchange-correlation functional is then a matter of fitting the shifts, or manually tuning them.

References

3.2.19 Silicon

Here we investigate computational methods for ATK-DFT calculations for bulk silicon in the diamond crystal structure. We apply the SG15 pseudopotentials and a number of exchange-correlation functionals; PBE, PBEsol, TB09-MGGA, and PBE. A limited number of HSE calculatios have been done for comparison.

The following **quantities** have been calculated for each computational method, both at the experimental silicon lattice constant and at the theoretically predicted ones:

•Elastic constants: Bulk modulus, Poisson ratio, and Youngs modulus.

•Band structure along the X—L Brillouin zone path.

•Direct and indirect band gaps.

•Effective masses in the valence band maximum and conduction band minimum.

•Static dielectric constant.

Convergence studies indicate that a 9x9x9 k-point grid and a 100 Hartree density mesh cutoff yields a well-converged silicon lattice constant, and these settings were used for computing the silicon ground state electronic structure used in all the ATK-DFT analyses listed above.

The projector shifts used for the **pps-PBE** method are 11.23 eV for *s*-orbitals and -1.09 eV for *p*-orbitals. This yields a single computational method with fairly high accuracy for predicting the fundamental band gap and lattice constant for silicon. Note that these PPS parameters should only be used with SG15 pseudopotentials.

This document is organized as follows. The section *Summary* gives a short summary of the main results from this study, while the section *Convergence* briefly presents the convergence studies mentioned above, and the section *Timing* compares CPU timings and scalability of different SG15 basis sets for silicon. Next, the section *Results* contains figures illustrating the performance of the different computational methods in predicting the materials properties listed above. Finally, the *Appendix* gives a template QuantumATK Python script for setting up ATK-DFT calculations for silicon similar to the ones presented here, and also elaborates on how to calculate hole effective masses using ATK-DFT.

Summary

The following list summarizes the main conclusions from this study.

- The PBEsol and pps-PBE methods both yield very accurate silicon **lattice constants**, but it is hard to say which method is most appropriate for computing elastic constants.
- All tested methods yield a qualitatively correct silicon **band structure**, although the position of the conduction band minimum depends a little on the method used.
- The TB09-MGGA, pps-PBE, and HSE methods accurately reproduce the fundamental (indirect) **band gap** of silicon.
- Electron **effective masses** in the silicon comduction band minimum are quite well captured by all tested methods. However, for hole effective masses in the valence band maximum, including spin-orbit interactions in the DFT calculations is crucial.
- As for the silicon static **dielectric constant**, the TB09-MGGA(PBE) and pps-PBE methods appear very accurate.

Convergence

An optimum combination of Monkhorst–Pack *k*-point grid and density mesh cutoff energy must be chosen such as to maximize computational accuracy while minimizing the computational effort. The two figures below illustrate how the calculated silicon lattice constant depends on these two parameters.

We see from Fig. 3.42 that a 9x9x9 *k*-point grid is in general sufficient to get a well-converged silicon lattice constant: The convergence curves are similar across different combinations of exchange-correlation functionals and SG15 basis sets, and with a 9x9x9 grid and denser, the lattice constant is converged to within roughly 10^{-3} % of the most dense k-point grid tested (19x19x19).

Similarly, Fig. 3.43 shows that a density mesh cutoff energy of 100 Hartree is in general sufficient: The calculated silicon lattice constant is converged to within roughly 10^{-3} % of the most dense density mesh tested (200 Ha cutoff energy).

Note: All remaining ATK-DFT calculations in this study of pure silicon therefore use a 100 Hartree mesh cutoff energy and a 9x9x9 Monkhorst–Pack *k*-point grid.

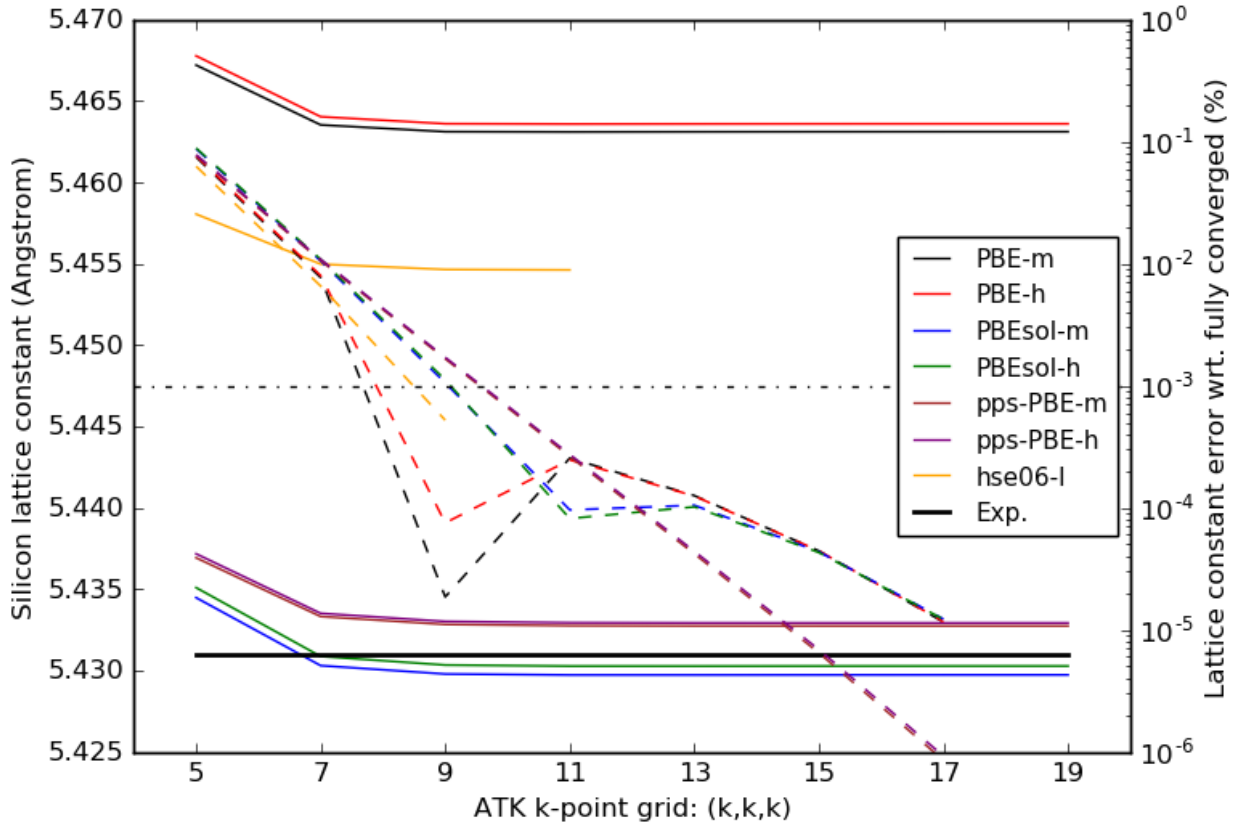


Fig. 3.42: Convergence of the silicon lattice constant with respect to k-point sampling (using Monkhorst-Pack grids). For the QuantumATK calculations the PBE and PBEsol exchange-correlation functionals, and PBE with the pseudopotential projector-shift (pps-PBE) have been used. Both the medium (m) and high (h) SG15 basis sets with 200 Hartree mesh cutoff energy have been used. HSE calculations have been performed using FHI-AIMS and the light basis set. The dashed lines indicate the (absolute) relative difference to the fully converged lattice constant (right-hand y-axis, in percent).

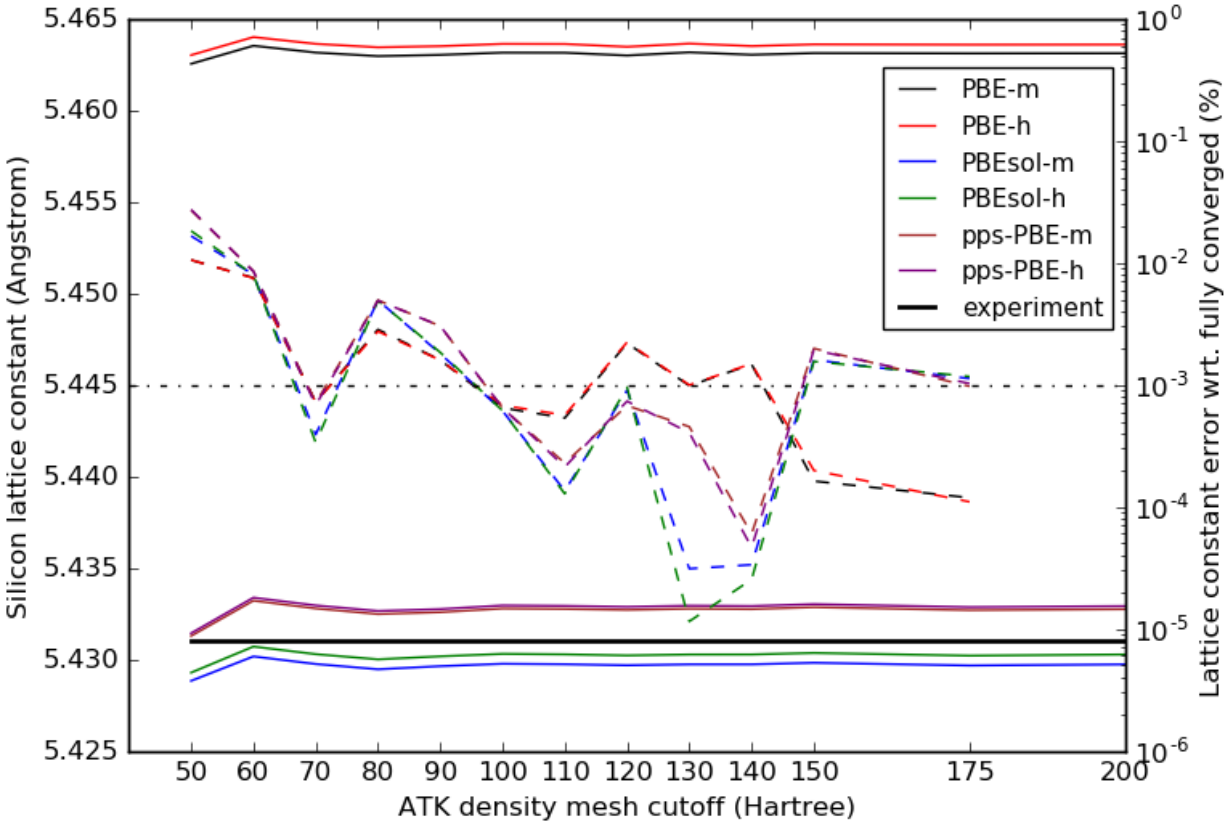


Fig. 3.43: Convergence of the silicon lattice constant with respect to the QuantumATK mesh cutoff energy using the PBE and PBEsol exchange-correlation functionals, and PBE with the pseudopotential projector-shift (pps-PBE), all for both the medium (m) and high (h) SG15 basis sets with a 15x15x15 k-point grid. The dashed lines indicate the (absolute) relative difference to the fully converged lattice constant (right-hand y-axis, in percent).

Timing

The computational cost of a calculation may depend critically on the choice of basis set, particularly for calculations on large systems with many atoms. Fig. 3.44 shown below illustrates this for the present case of silicon and the Medium, High, and Ultra SG15 basis sets.

The silicon primitive cell (containing 2 atoms) was systematically repeated in steps of one along all three lattice vectors, resulting in rapidly increasing systems sizes. PBE calculations were then performed for each system, using -point sampling only, to avoid any influence of k-point sampling on the recorded CPU times.

It is quite clear from the figure that the computational loads of the SG15 Medium and High basis sets for silicon are not much different for relatively small system sizes, but the High basis set becomes significantly more demanding for larger system sizes. For the 432-atom silicon unit cell, the CPU time difference is roughly a factor of 2. Moreover, the Ultra basis set appears in comparison extremely demanding, and will not be used in the remaining calculations in this study.

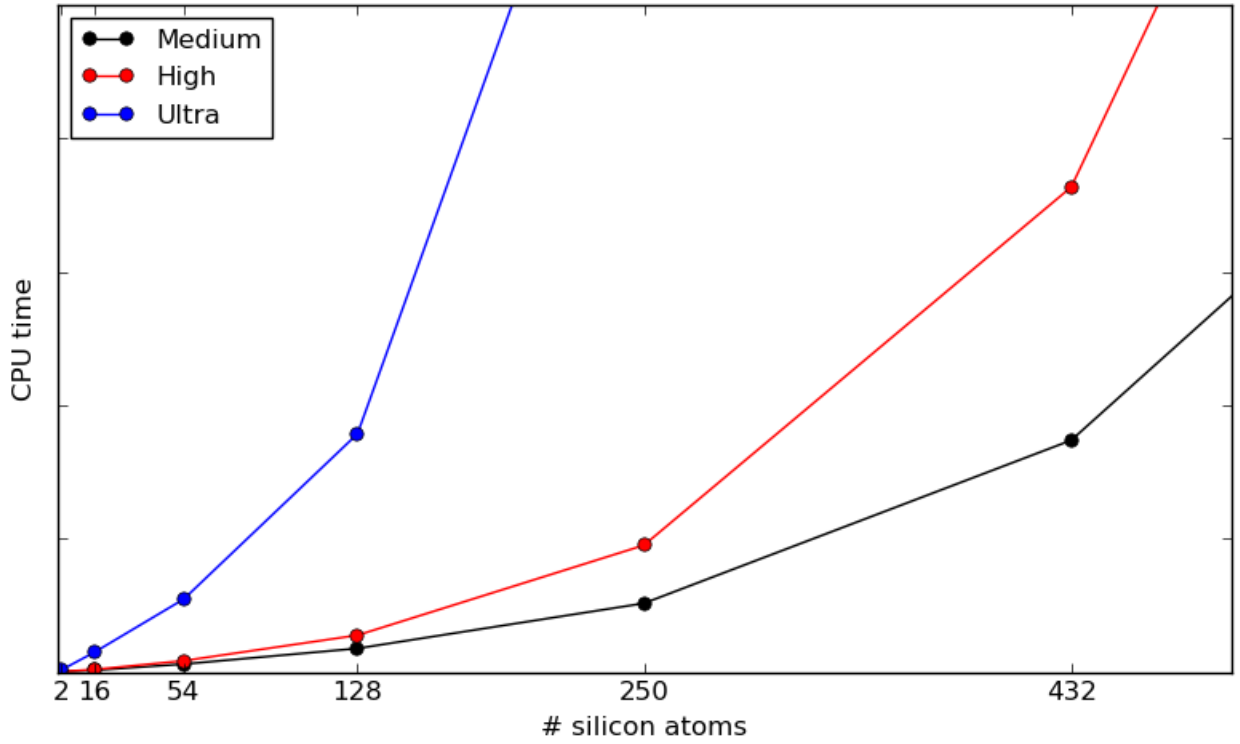


Fig. 3.44: Total CPU time for the first 5 SCF iterations by the ATK-DFT calculator for the Medium (black), High (red), and Ultra (blue) SG15 basis sets, against the number of silicon atoms in the increasingly larger (repeated) silicon unit cell. Only the point is sampled (1x1x1 k-point grid) and the calculations were executed on a single CPU. Note that both figure axes are linear.

Results

This section presents results obtained with the PBE, PBEsol, pps-PBE, and TB09-MGGA methods using Medium and High basis sets with the SG15 pseudopotential for silicon. The list below gives direct links to all subsections:

•*Lattice constant*

•*Elastic constants*

•*Band structure*

•*Band gaps*

•*Effective masses*

•*Dielectric constant*

Lattice constant

The PBEsol lattice constant for silicon is very close to the experimental value, 5.431 Å, but the pps-PBE method is also very accurate. The bare PBE functional (without any pseudopotential projector shifts) is known to overestimate lattice constants in general, but for silicon only by 0.6%.

Elastic constants

Almost all methods tested appear to underestimate the silicon bulk modulus and Youngs modulus, but overestimate the Poisson ratio. The PBE-m method is a curious exception to this rule, but the pps-PBE method appears in general better.

Band structure

All tested methods correctly predict a valence band maximum (VBM) at the Brillouin zone point, and a conduction band minimum (CBM) close to the X point, along the –X path. However, the position of the CBM depends slightly on the method applied, and the band energy in the CBM significantly so (see also section *Band gaps*). On the other hand, the Medium and High basis sets appear to give very similar silicon band structures.

Band gaps

The silicon fundamental band gap is well reproduced by the TB09-MGGA, pps-PBE, and HSE methods, both at experimental and DFT optimized lattice constants, the pps-PBE yielding a particularly accurate band gap. Note, however, that the TB09-MGGA c-parameter was calculated self-consistently from the silicon electronic structure (no fitting), while the pps-PBE projector shifts were essentially fitted to the silicon band gap and lattice constant. On the other hand, the TB09-MGGA functional cannot be used for geometry optimization, which the pps-PBE method is well suited for. Also note that the HSE direct gap, E_{Γ_1}, is very accurate.

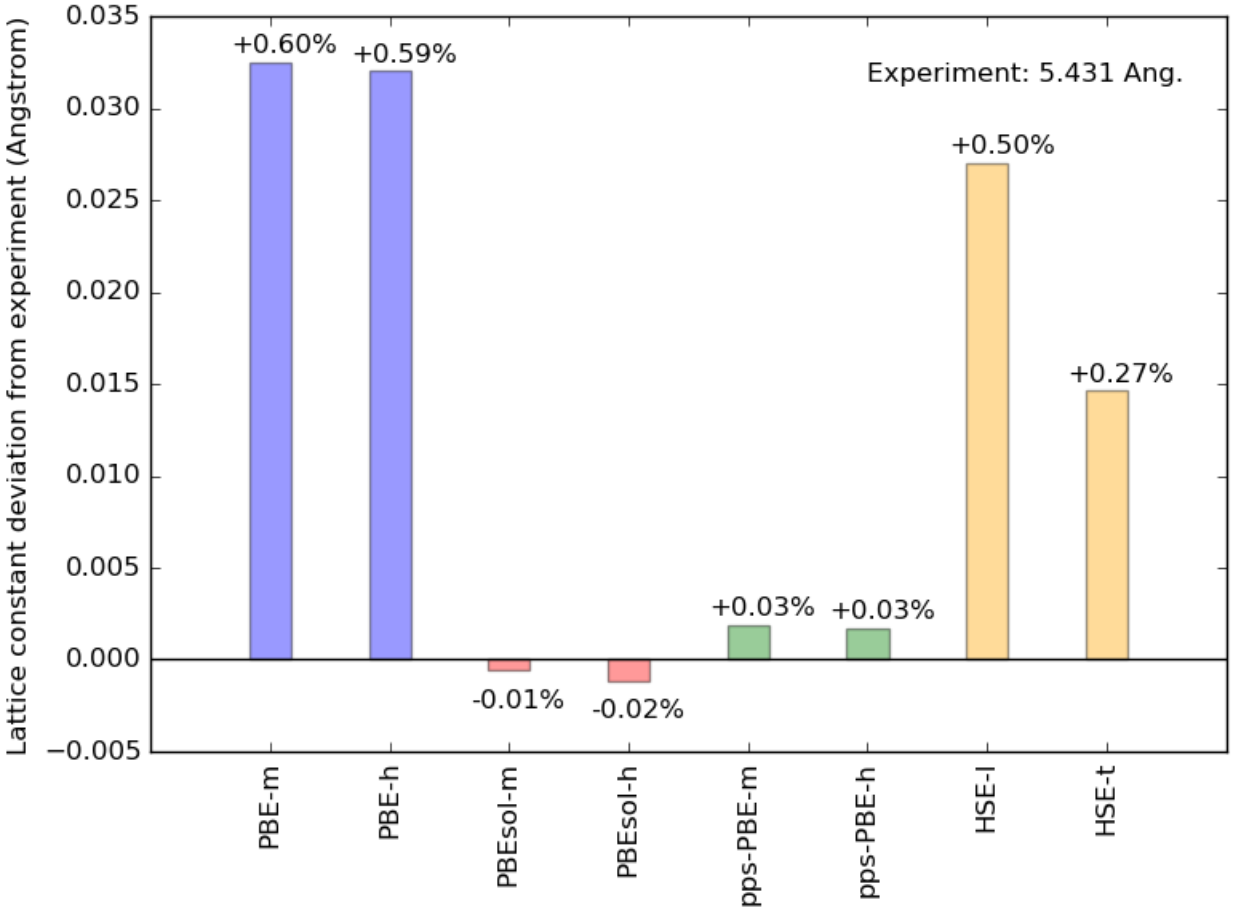


Fig. 3.45: Silicon lattice constant calculated using QuantumATK and the PBE (blue), PBEsol (red) exchange-correlation functionals and the pps-PBE method (green), plotted as deviation from the experimental value. Both medium (m) and high (h) SG15 basis sets are used. HSE calculations have been performed using FHI-AIMS with light (l) and tight (t) basis sets.

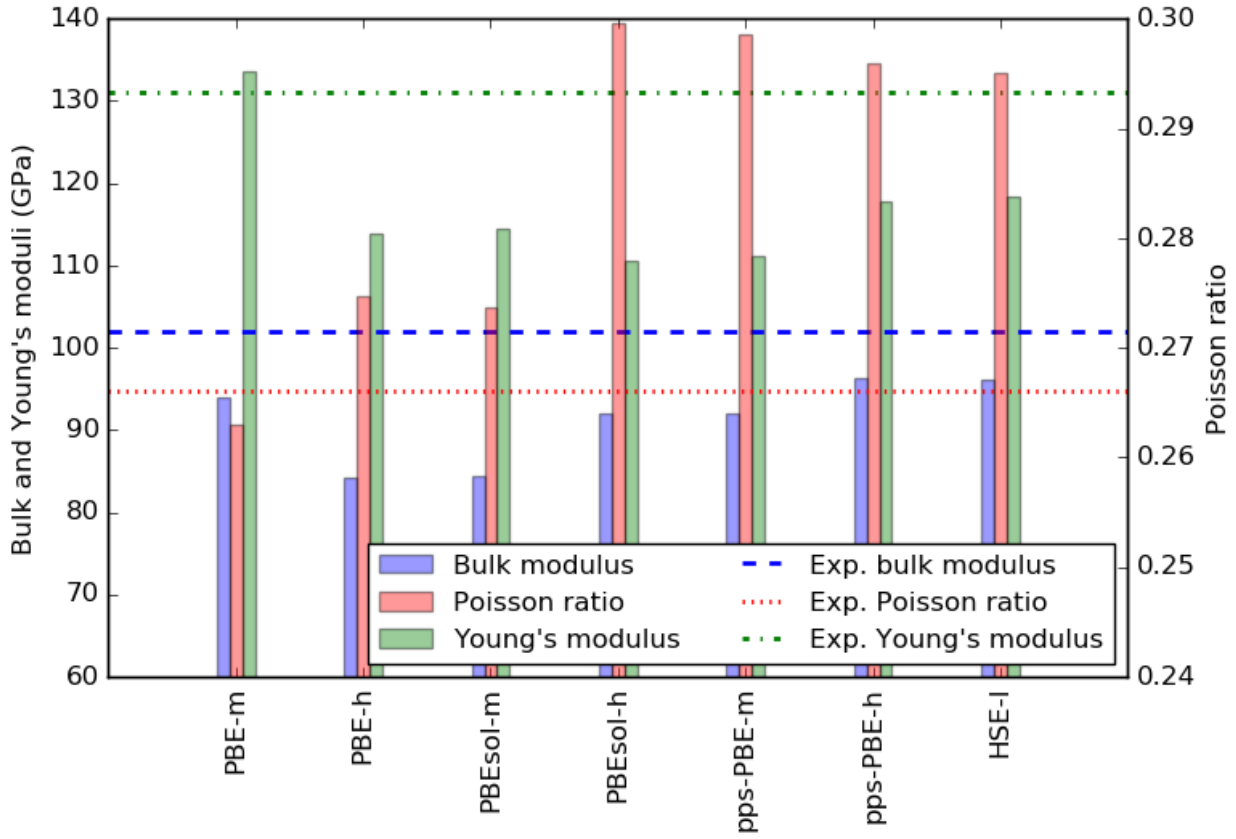


Fig. 3.46: Silicon bulk modulus (blue), Poisson ratio (red), and Youngs modulus (green) calculated using the PBE, PBEsol, pps-PBE, and HSE methods. The horizontal lines indicate experimental values, all from www.crystran.co.uk.

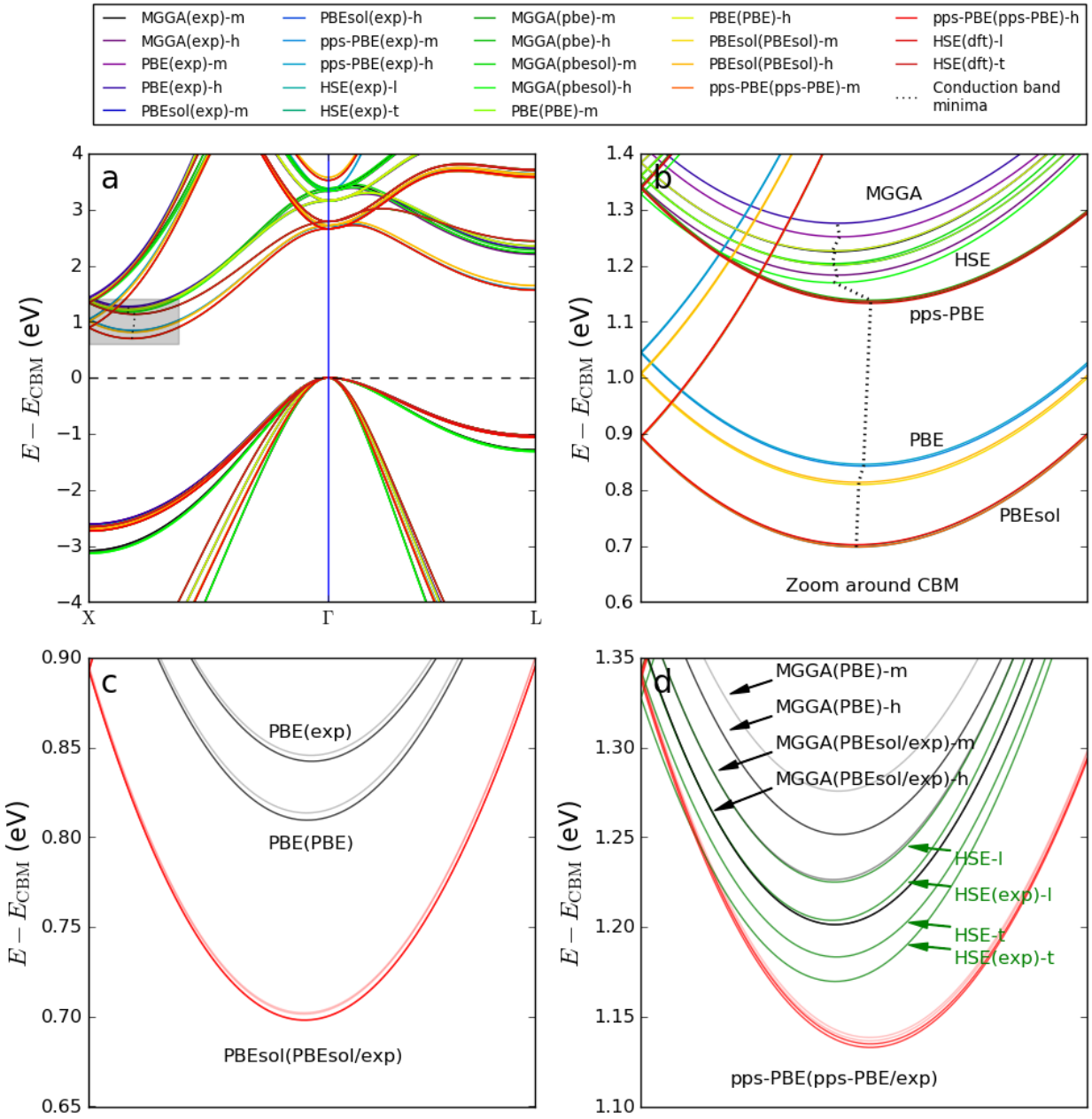


Fig. 3.47: (**a**) Bandstructure of silicon calculated using QuantumATK with the PBE, pbesol and TB09-MGGA exchange-correlation functionals and with the pps-PBE method. The bandstructure has also been calculated using FHI-AIMS and the HSE functional. All the plotted band structures are aligned at the valence band maximum (VBM). Both medium (m) and high (h) SG15 basis sets are used in the QuantumATK calculations. In the FHI-AIMS calcualtions, light (l) and tight (t) basis sets have been used. The name in parenthesis in the labels refers to the lattice constant: exp means lattice constant fixed at the experimental value, 5.431 Angstrom, while PBE, PBEsol, and pps-PBE indicate that the lattice constant was optimized using DFT. The shaded black area indicates the region around the conduction band minimum (CBM) shown in panel **b**. (**b**) Zoom around the CBM of the data shown in panel **a**. The dashed black line marks the position of the calculated CBMs.

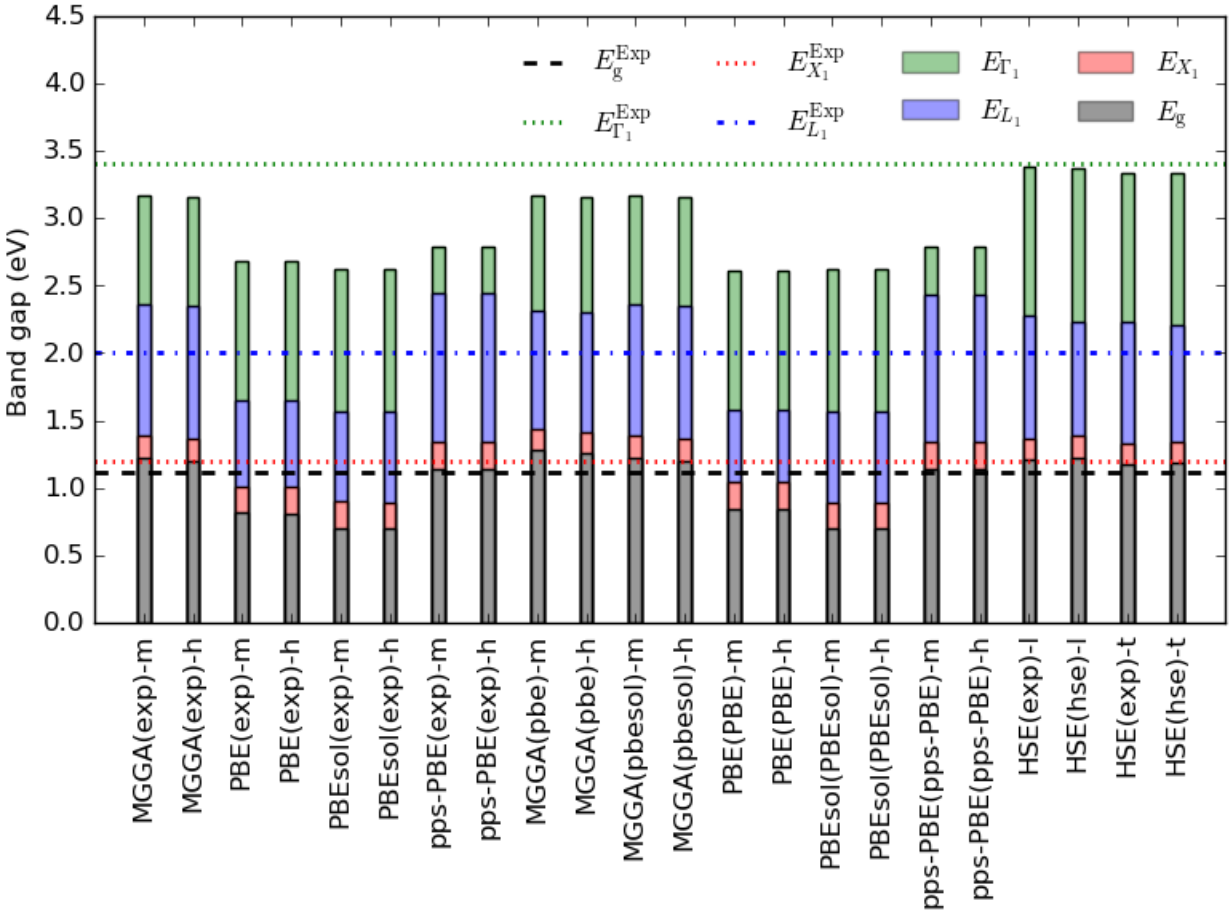


Fig. 3.48: Silicon band gaps; E_g (fundamental gap, black) and energy gaps from the Γ-point valence band maximum to the conduction band minima at X (red), L (blue), and Γ (green), calculated using the PBE, PBEsol, and TB09-MGGA, HSE (using FHI-AIMS) exchange-correlation functionals and with the pps-PBE method. Both medium (m) and high (h) SG15 basis sets are used, while for the HSE calculations the FHI-AIMS light and tight are used. The name in parenthesis in the labels on the x-axis refers to the lattice constant: exp means lattice constant fixed at the experimental value, 5.431 Angstrom, while pbe, pbesol, pps-pbe and hse indicate that the lattice constant was optimized using DFT and the corresponding functional. The reference values E_g^{Exp} (black, dashed), $E_{X_1}^{Exp}$ (red, dotted), $E_{L_1}^{Exp}$ (blue, dash-dotted), and $E_{\Gamma_1}^{Exp}$ (green, dotted) are from http://www.ioffe.ru/SVA/NSM/Semicond/Si/Figs/121.gif.

Effective masses

The silicon effective masses that are calculated with spin-orbit interaction included are in fair agreement with experiment for all the density functionals, basis sets, and pseudopotentials adopted in this study, see the two figures and table in this section. There exists a certain trend that the hole effective mass values tend to be somewhat underestimated as compared to experimental values for the LDA and GGA-PBE density functionls, which underestimate the silicon band gap, whereas the hole effective masses for the TB09-MGGA and pps-PBE density functionals, which give accurate band gap for bulk Si, tend to be overestimated. The electron longitudinal and transversal masses calculated at the X-valley minimum are relatively insensitive to the choice of density functional. Having the spin-orbit interaction included into the effective mass calculation is crucial for reproducing the experimental effective mass values.

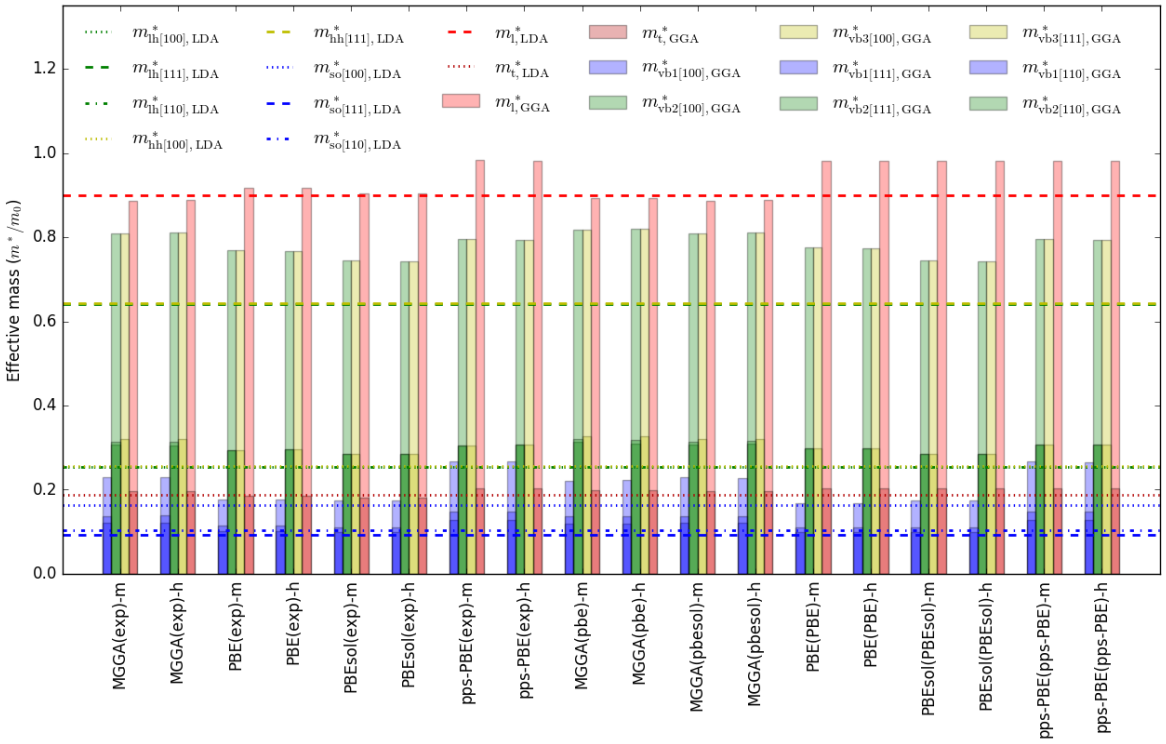


Fig. 3.49: Silicon effective masses calculated with different methods, basis sets, and lattice parameters, without including spin-orbit interaction. The LDA effective masses calculated with the FHI pseudopotential and DoubleZetaPolarized basis set are given with red dashed and red dotted lines for the longitudinal and transversal electron masses, respectively. The yellow, green, and blue lines stand for the heavy, light, and split-off hole effective masses for different crystollagraphic directions: the (100) direction (dotted line), (111) direction (dashed line), and (110) direction (dot-dashed line). The LDA results are taken as a reference for the case of no spin-orbit coupling, since the LDA derived effective masses of Si calculated with spin-orbit coupling are in good agreement with experiment. Note that the same color scheme is chosen for the ATK-DFT calculated effective mass data that are given with color bars.

[1] J.ăL. Janssen, Y.ăGillet, S.ăPonc, A.ăMartin, M.ăTorrent, and X.ăGonze. Precise effective masses from density functional perturbation theory. *Physical Review B*, 205147:1–22, 2016. doi:10.1103/PhysRevB.93.205147.

[2] R.ăN. Dexter, H.ăJ. Zeiger, and B.ăLax. Cyclotron Resonance Experiments in Silicon and Germanium. *Physical Review*, 104(3):637–644, nov 1956. doi:10.1103/PhysRev.104.637.

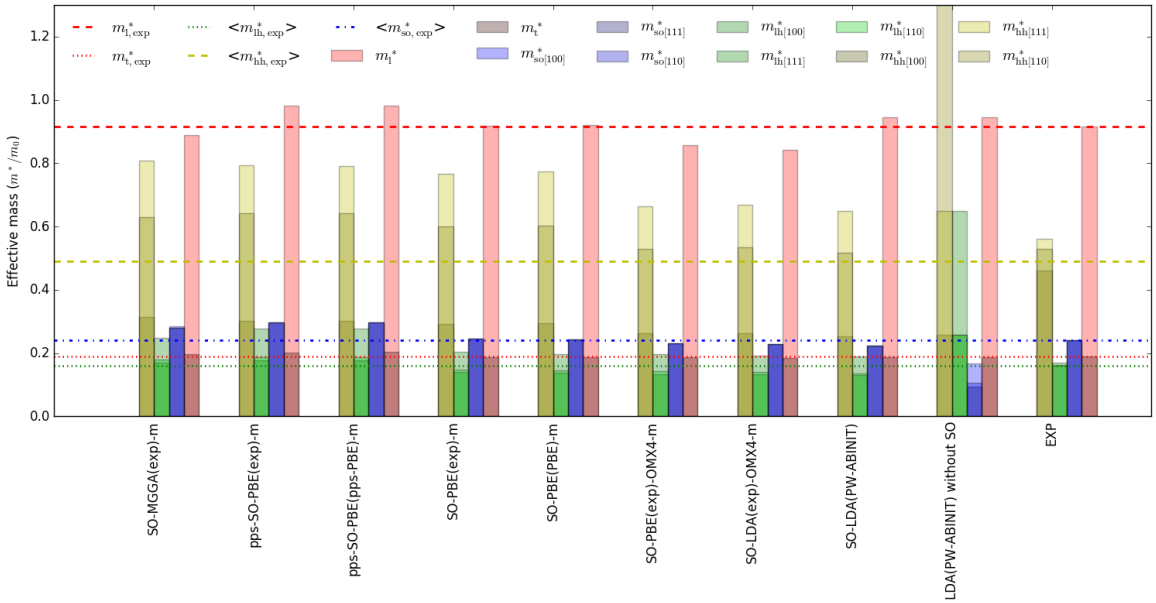


Fig. 3.50: Silicon effective masses calculated with different methods, basis sets, calculations. The experimental heavy, light, and split-off effective masses are averaged over the three crystallographic directions, and are given with yellow dashed, green dotted, and blue dot-dashed line, respectively. The red dashed and dotted lines correspond to the longitudinal and transversal electron masses measured in experiment, respectively. Note that the same color scheme is chosen for the ATK-DFT calculated effective mass data that are given with color bars. For the sake of comparison, the LDA-ABINIT results for effective masses obtained by Janssen *et al.* [Phys. Rev. B **93**, 205147 (2016)] with and without spin-orbit interaction included are added to the figure together the experimental data for different crystallographic directions, see R. N. Dexter and B. Lax, Phys. Rev. **96**, 223 (1954). The experimental light-hole effective mass is from http://www.ioffe.ru/SVA/NSM/Semicond/Si/bandstr.html.

m/m$_e$*	*MGGA(exp) SG15-m*	*pps-PBE(exp) SG15-m*	*pps-PBE SG15-m*	*PBE(exp) SG15-m*	*PBE SG15-m*	*PBE(exp) OMX4-m*	*LDA(exp) OMX4-m*	*LDA (literature)*	*Experiment*
m_l	0.888 (0.887)	0.983	0.983 (0.983)	0.918	0.921 (0.921)	0.856	0.842	0.946 (0.946)	0.9163
m_t	0.197 (0.197)	0.202	0.203 (0.203)	0.187	0.188 (0.188)	0.187	0.184	0.188 (0.188)	0.1905
m_{hh}^{100}	-0.313 (-0.323)	-0.302	-0.302 (-0.306)	-0.291	-0.294 (-0.298)	-0.262	-0.263	-0.254 (-0.258)	-0.46
m_{hh}^{111}	-0.808 (-0.809)	-0.794	-0.791 (-0.796)	-0.767	-0.773 (-0.775)	-0.665	-0.669	-0.648 (-0.650)	-0.57
m_{hh}^{110}	-0.630 (-3.353)	-0.642	-0.642 (-3.988)	-0.601	-0.602 (-3.871)	-0.530	-0.534	-0.517 (-2.703)	-0.53
m_{lh}^{100}	-0.249 (-0.320)	-0.277	-0.277 (-0.306)	-0.203	-0.196 (-0.298)	-0.196	-0.193	-0.188 (-0.258)	-0.171
m_{lh}^{111}	-0.169 (-0.809)	-0.177	-0.177 (-0.796)	-0.141	-0.139 (-0.775)	-0.134	-0.133	-0.130 (-0.650)	-0.157
m_{lh}^{110}	-0.180 (-320)	-0.187	-0.187 (-0.306)	-0.149	-0.146 (-0.298)	-0.142	-0.141	-0.137 (-0.258)	-0.163
m_{so}^{100}	-0.284 (-0.223)	-0.297	-0.297 (-0.267)	-0.245	-0.242 (-0.169)	-0.230	-0.229	-0.223 (-0.167)	-0.24
m_{so}^{111}	-0.280 (-0.121)	-0.297	-0.297 (-0.129)	-0.245	-0.242 (-0.099)	-0.230	-0.229	-0.223 (-0.094)	-0.24
m_{so}^{110}	-0.281 (-0.138)	-0.297	-0.297 (-0.148)	-0.245	-0.242 (-0.111)	-0.230	-0.229	-0.223 (-0.105)	-0.24

Fig. 3.51: Table of silicon effective masses calculated with different computational methods, all including spin-orbit interaction. The literature LDA+SO data were calculated using the ABINIT plane-wave code[1], while the experimental data for the heavy and light hole effective masses are fromPage 698, 2 and from http://www.ioffe.ru/SVA/NSM/Semicond/Si/bandstr.html for split-off masses. For the sake of comparison, effective masses calculated without including spin-orbit interaction are provided in parenthesis.

Dielectric constant

The static ($\omega = 0$) dielectric constant of the silicon bulk is most reliably reproduced by the TB09-MGGA functional at the PBE lattice constant and by the pps-PBE method at the corresponding theoretical lattice constant. The lattice constant used seems in general to significantly affect the calculated dielectric constant; note for example how the PBE dielectric constant changes from 11 to 14 depending on the lattice constant used, but independently of the basis set.

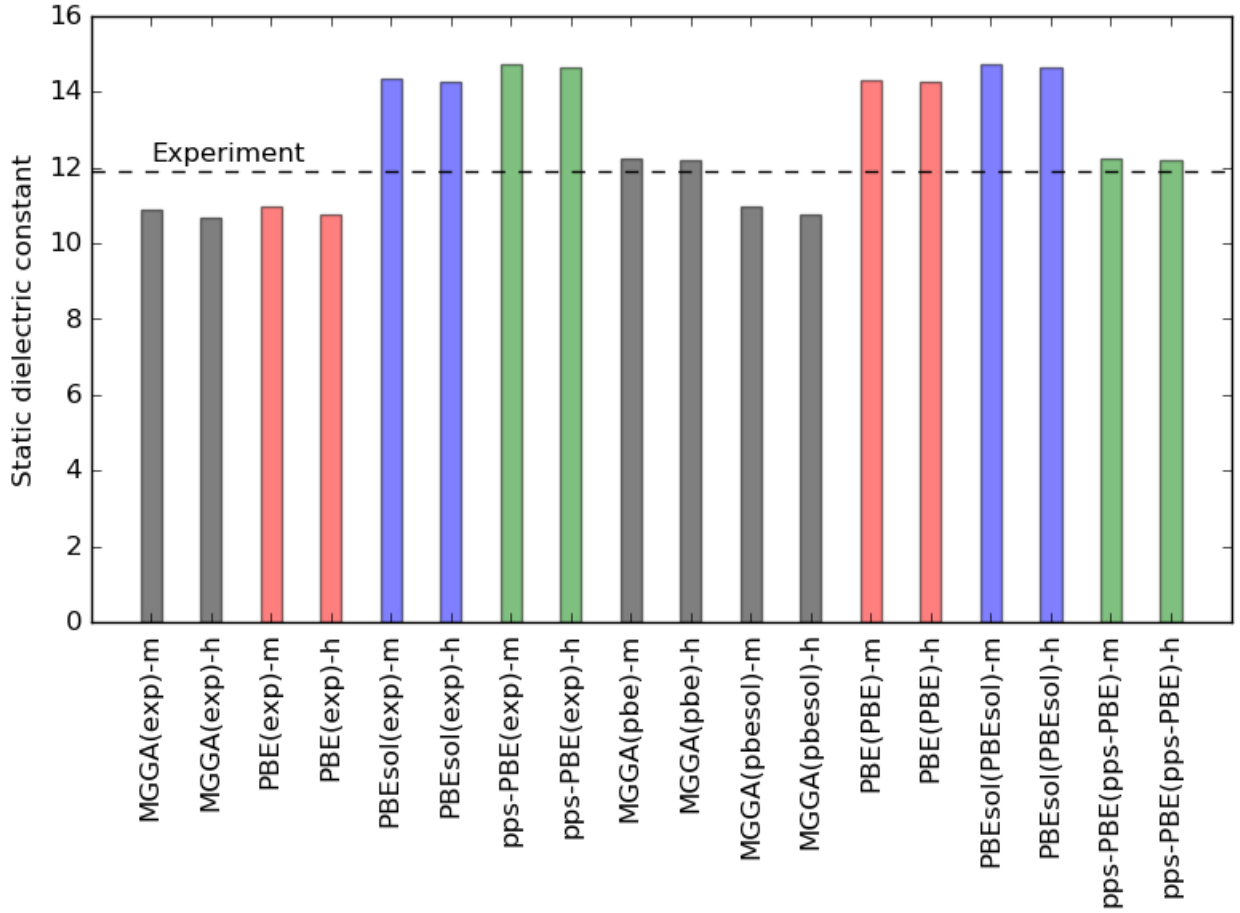


Fig. 3.52: Static dielectric constant calculated using the TB09-MGGA (black), PBE (red) and PBEsol (blue) exchange-correlation functionals and the pps-PBE method (green). Both medium (m) and high (h) SG15 basis sets are used, and the number of included electronic bands were 10 below the Fermi level and 20 above it. The name in parenthesis in the labels on the x-axis refers to the lattice constant: exp means lattice constant fixed at the experimental value, 5.431 Angstrom, while PBE, PBEsol, and pps-PBE indicate that the lattice constant was optimized using DFT. The dashed line indicates the exprimental value, 11.9.

Appendix

The QuantumATK Python script shown below may be used as a template for ATK-DFT calculations for silicon with the SG15 pseudopotential. The script defines the silicon bulk configuration and then sets up the ATK-DFT calculator with PBE exchange-correlation. The script blocked named `Basis Set` shows various options for the SG15 basis set; ordinary PBE and PBE with pseudopotential projector shifts, both with Medium and High basis sets.

The script is available for direct download: `silicon.py`.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.431*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
# Ordinary GGA: Medium or High basis set for SG15.
basis_set = [BasisGGASG15.Silicon_Medium]
#basis_set = [BasisGGASG15.Silicon_High]

# GGA with pseudopotential projector-shift method: Medium or High basis set for SG15.
#projector_shift = PseudoPotentialProjectorShift(s_orbital_shift=11.23*eV,
#                                                p_orbital_shift=-1.09*eV)
#basis_set = [BasisGGASG15.Silicon_Medium(projector_shift=projector_shift)]
#basis_set = [BasisGGASG15.Silicon_High(projector_shift=projector_shift)]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = GGA.PBE

k_point_sampling = MonkhorstPackGrid(
    na=9,
    nb=9,
```

(continues on next page)

(continued from previous page)

```
    nc=9,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=100.0*Hartree,
    )

iteration_control_parameters = IterationControlParameters(
    damping_factor=0.4,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('silicon.nc', bulk_configuration)
```

Calculation of effective masses

The following is a list of important points abouth calculations of masses using ATK-DFT. See also section *Effective masses*.

1. As seen from Fig. 3.49, the SG15_medium basis set is rather accurate for calculating the electron and hole effective masses of Si, as suggested by comparison of effective masses obtained with the SG15 Medium and SG15 High basis sets.
2. The calculated effective masses are not sensitive to the choice of either DFT-derived or experimental lattice parameter value for DFT calculations, see Fig. 3.49 and Fig. 3.50, i.e., the electron and hole effective masses of Si depend weakly on the lattice constant near its experimental value.
3. The ATK-DFT-LCAO calculations of effective masses are in consistency with the corresponding plane-wave pseudopotential calculations done with the ABINIT code, see Janssen *et al.* [Phys. Rev. B 93, 205147 (2016)], which provides a reference for the quality of the LCAO basis set. The agreement does not depend on whether the calculations are done with or without spin-orbit interaction included.
4. In general, the spin-orbit interaction has to be taken into account for hole effective mass calculations to achieve a good agreement with experiment. Otherwise, the effective masses are unphysically anisotropic and are heavily overestimated as seen in the first figure and the table. Unlike the hole effective masses, the longitudinal and transversal electron effective masses are both virtually unaffected by the spin-orbit interaction.
5. The electron, light, and split-off hole effective masses calculated using the LDA and GGA-PBE functionals with the SG15 basis set are closer to the corresponding experimental values, compared to the masses obtained with the OMX basis set. The heavy hole masses seem to be in a better agreement with experiment if obtained with the OMX basis set. Note that the LDA and GGA functional underestimate the indirect band gap of Si by roughly a factor of 2, as compared to experiment.
6. Using the TB09 meta-GGA and pps-GGA-PBE functionals corrects for the band gap value mentioned in #5. At the same time, it results in hole effective mass values that are larger than the LDA and GGA-PBE derived masses, and

are somewhat overestimated compared to the corresponding experimental values. That might be explained by the fact that the TB09 meta-GGA and pps-GGA-PBE calculated band gap increases as compared to the LDA and GGA-PBE functionals, so that it drives the system into more insulating state. That is consistent with the increase of the hole effective masses. We also notice that using the pps-GGA-PBE functional, which has adjustable parameters, allows correcting the lattice constant value of Si (in addition to correcting the band gap) that is otherwise highly overestimated compared to the experimental value if a standard GGA-PBE functional is adopted for volume optimization of the Si unit cell.

8. There exists a subtle issue in hole effective mass calculation for degenerate bands at the G point. The accurate extraction of effective masses with a finite-difference method requires calculating the effective mass for a given crystallographic direction as a function of the k-mesh density by varying delta parameter (which is a k-mesh descretization step) in the EffectiveMass analyzer. The actual effective mass value is then to be taken in a parabolic band region where the corresponding mass values should show virtually no delta dependence, i.e., the effective mass as a function of delta (k-mesh density) reaches a plateau. Note that the default value of 0.001 $Å^{-1}$ for delta is good enough for calculating electron and split-off masses, but it may need to be increased to ~0.01 $Å^{-1}$ for light and heavy hole masses. The TB09-MGGA derived light and heavy hole effective masses were calculated at even higher value of delta ~0.03 $Å^{-1}$.

References

3.2.20 Germanium

Here we investigate computational methods for ATK-DFT calculations for bulk germanium in the diamond crystal structure. We apply the SG15 pseudopotentials and a number of exchange-correlation functionals; PBE, PBEsol, TB09-MGGA, and PBE.

The following **quantities** have been calculated for each computational method, both at the experimental germanium lattice constant and at the theoretically predicted ones:

- Elastic constants: Bulk modulus, Poisson ratio, and Youngs modulus.
- Band structure along the X—L Brillouin zone path.
- Band gaps: –L (fundamental gap), – (direct gap), and the –X and –Δ indirect gaps.
- Electron effective masses in the L point; transverse and longitudinal.
- Static dielectric constant.

Convergence studies indicate that a 100 Hartree mesh cutoff in conjunction with a 8x8x8 *k*-point grid for calculation of elastic properties (e.g. lattice constant) and a 9x9x9 *k*-point grid for electronic properties (e.g. band gap) are suitable for well-converged and computationally efficient ATK-DFT calculations for bulk germanium. These settings were used for computing the quantities listed above.

The projector shifts used for the **pps-PBE** method are 15.0 eV for *s*-orbitals, 0.2 eV for *p*-orbitals, and -2.0 eV for *d*-orbitals. This yields a single computational method with fairly high accuracy for predicting the fundamental band gap and lattice constant for germanium. Note that these PPS parameters should only be used with SG15 pseudopotentials.

This document is organized as follows. The section *Summary* gives a short summary of the main results from this study, while the section *Convergence* briefly presents the convergence studies mentioned above, and the section *Timing* compares CPU timings and scalability of different SG15 basis sets for germanium. Next, the section *Results* contains figures illustrating the performance of the different computational methods in predicting the materials properties listed above. Finally, the *Appendix* gives a template QuantumATK Python script for setting up ATK-DFT calculations for germanium similar to the ones presented here.

Summary

The following list summarizes the main conclusions from this study.

•The PBEsol, pps-PBE, and HSE methods all yield fairly accurate germanium **lattice constants**, but pps-PBE and HSE appear somewhat more accurate than PBEsol for computing elastic properties of germanium.

•GGA methods, such as PBE and PBEsol, tend to yield a qualitatively wrong (semi-metallic) germanium **band structure**. The TB09-MGGA, pps-PBE, and HSE methods do much better, though the lowest TB09-MGGA valence bands are in general shifted upwards as compared to those found with pps-PBE and HSE. Note, however, that the TB09-MGGA c-parameter was computed self-consistently. Manually tuning the c-parameter could probably bring the TB09-MGGA valence bands down in energy.

•The PBE and PBEsol density functionals should not be used for computing germanium **band gaps**. Conversely, the pps-PBE and HSE methods can yield fairly accurate numbers for the fundamental band gap. Tuning the TB09-MGGA c-parameter may possibly bring better TB09-MGGA band gaps, but this was not tested.

•The predicted band gaps appear to depend somewhat on the germanium lattice constant used for the calculation. In general, the **fundamental band gap** increases as the unit cell volume decreases.

•Even though PBE and PBEsol do not predict the correct **band gap ordering** in germanium, they are actually significantly better for computing **effective electron masses** in the germanium L point (the experimental conduction band minimum), than the TB09-MGGA and pps-PBE methods. However, all tested methods overestimate the electron effective masses, especially the fairly small transverse effective mass.

•The PBE and PBEsol functionals should not be used for computing **optical properties** of germanium, such as the static dielectric constant. The vanishing -point direct band gap with GGA methods is most likely the reason for this. The ppsPBE method appears significantly better in this respect.

Convergence

The type of Monkhorst–Pack k-point grid and the density mesh cutoff energy are essential ATK-DFT parameters that affect the computational efficiency and precision: A dense k-point grid and high cutoff energy usually give high precision, but may also be computationally intense. It is therefore important to investigate the trade-off between computational precision and cost. Fig. 3.53 illustrates how the germanium total energy, fundamental band gap, and lattice constant depend on the mesh cutoff and k-point grid.

Note that a standard Monkhorst–Pack k-point grid is -centered for odd numbers of points (e.g. 3x3x3), but does not include the point for even number of points (e.g. 4x4x4). The blue lines in Fig. 3.53 are for k-point grids where even grids are shifted to (such that both 3x3x3 and 4x4x4 grids **do** include), while the red lines are for calculations where odd grids are shifted away from (such that both 3x3x3 and 4x4x4 grids **do not** include).

For the SG15 pseudopotential with both Medium and High basis sets, a 100 Hartree mesh cutoff gives band gaps and lattice constants that are converged to within 10^{-4} eV and roughly 10^{-4} Å, respectively.

It is also clear that an off- k-point grid (non-centered, red) gives faster convergence of the total energy and lattice constant than the k-point grid that includes the point (centered, blue). We see that the non-centered 8x8x8 Monkhorst–Pack grid yields higly converged lattice constants.

On the other hand, the computed band gap converges fastest with the -centered k-point grid (blue). This may be because the valence band maximum is located at the point. The 9x9x9 Monkhorst–Pack grid is -centered by default, and yields highly converged band gaps.

Note: All remaining ATK-DFT calculations in this study therefore use a 100 Hartree mesh cutoff energy and a standard (non-centered) 8x8x8 Monkhorst–Pack k-point grid for elastic properties and the standard 9x9x9 grid for computing electronic properties.

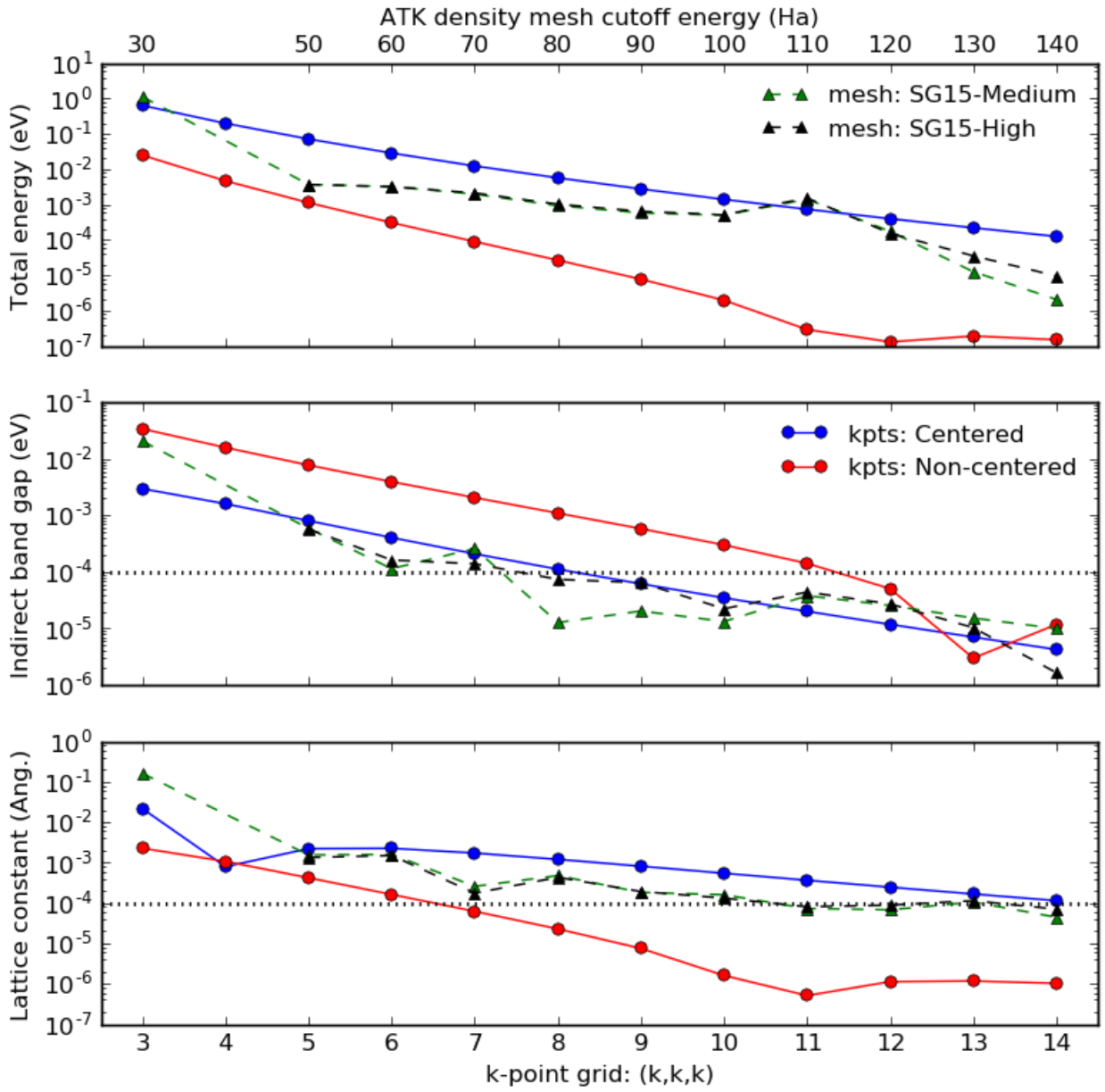


Fig. 3.53: Convergence of the germanium total energy, fundamental (indirect) band gap, and lattice constant with respect to the ATK-DFT density mesh cutoff (top x-axis in all three panels; green and black dashed lines) and k-point grid (bottom x-axis in all three panels; blue and red lines). The centered k-point grids are shifted to the Γ point (only affects even grids, e.g. 4x4x4), while the non-centered k-point grids are shifted by (0.5,0.5,0.5) and therefore do not sample the Γ point (only affects odd grids, e.g. 3x3x3). The horizontal dotted lines indicate tight convergence criteria.

Timing

The computational cost of a calculation may depend critically on the choice of basis set, particularly for calculations on large systems with many atoms. Fig. 3.54 shown below illustrates this for the present case of germanium and the Medium, High, and Ultra SG15 basis sets.

The germanium primitive cell (containing 2 atoms) was systematically repeated in steps of one along all three lattice vectors, resulting in rapidly increasing systems sizes. PBE calculations were then performed for each system, using -point sampling only, to avoid any influence of k-point sampling on the recorded CPU times.

It is quite clear from the figure that the computational loads of the SG15 Medium and High basis sets for germanium are not much different for relatively small system sizes, but the High basis set becomes more demanding for larger system sizes. Moreover, the Ultra basis set appears in comparison extremely demanding, and will not be used in the remaining calculations in this study.

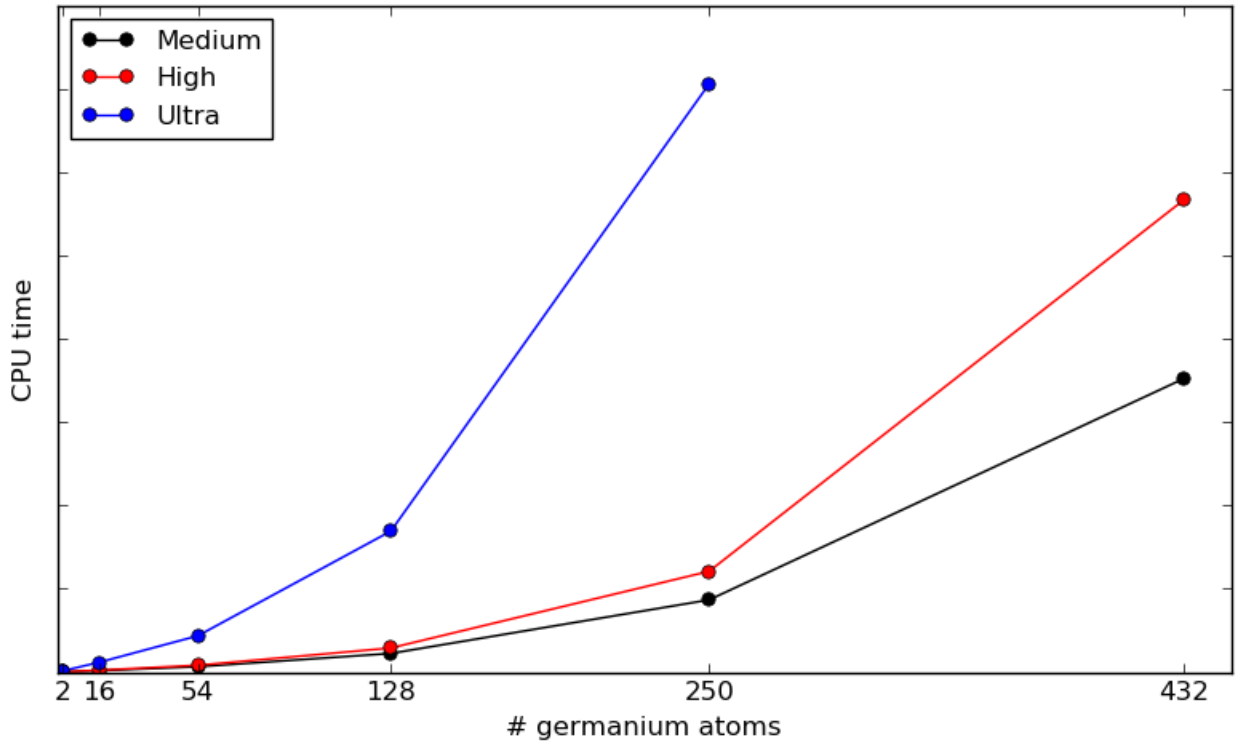


Fig. 3.54: Total CPU time for the first 5 SCF iterations by the ATK-DFT calculator for the Medium (black), High (red), and Ultra (blue) SG15 basis sets, against the number of germanium atoms in the increasingly larger (repeated) bulk unit cell. Only the point is sampled (1x1x1 k-point grid) and the calculations were executed on a single CPU core. Note that both figure axes are linear. For reference, the 432-atom SG15-Medium calculation reach full convergence in 11 SCF steps, which take less than 30 minutes if running in parallel on 16 modern CPU cores.

Results

This section presents ATK-DFT results obtained with the PBE, PBEsol, pps-PBE, and TB09-MGGA methods using Medium and High basis sets with the SG15 pseudopotential for germanium, as well as HSE results obtained with the FHI-aims calculator. The list below gives direct links to all subsections:

•*Lattice constant*

•*Elastic constants*

•*Band structure*

•*Band gaps*

•*Electron effective masses*

•*Dielectric constant*

Lattice constant

The PBEsol lattice constant for germanium is very close to the experimental 300 K value, 5.6574 Å, but the pps-PBE and HSE methods are also quite accurate. The bare PBE functional (without any pseudopotential projector shifts) is known to overestimate lattice constants in general, and for germanium by almost 3%.

Elastic constants

The experimental bulk modulus, Poisson ratio, and Youngs modulus of germanium are well reproduced by the pps-PBE method, but HSE also does well in this respect. PBEsol is in general a bit more off, and pure PBE underestimates all three elastic quantities.

Band structure

The germanium fundamental band gap is fairly small (0.66 eV at 300 K), and occurs between the and L high-symmetry points in the Brillouin zone. The remaining conduction band minima are quite close in energy (see also *Band gaps*). This band structure is hard to correctly reproduce within GGA-DFT. As seen in the figure below, PBE and PBEsol tend to close the direct gap at , and yield a rather small –L band gap. On the other hand, the TB09-MGGA, pps-PBE, and HSE methods correctly yield a significant direct band gap at , and also appear to preserve the correct band gap ordering. The SG15 High basis set (dash-dotted lines) gives in general slightly lower band energies than the Medium basis set does (full lines). A similar trend is observed for the tight and light accuracy settings for the FHI-aims HSE calculations.

Band gaps

As noted just above, the PBE and PBEsol exchange-correlation functionals predict a qualitatively wrong band gap ordering for germanium; the -point direct gap vanishes, and the lowest conduction band at the L point may even cross the Fermi level. No PBE or PBEsol band gaps are therefore included in the figure below.

The germanium band gaps are quite well reproduced by the pps-PBE and HSE methods. Especially the pps-PBE calculation at the pps-PBE lattice constant with the SG15 High basis set appears very accurate. The TB09-MGGA method correctly predicts that the fundamental germanium band gap is the –L gap, but overestimates the band gaps in general. Note, however, that the TB09-MGGA c-parameter was calculated self-consistently from the germanium electronic structure (no fitting), while the pps-PBE projector shifts were essentially fitted to the germanium fundamental

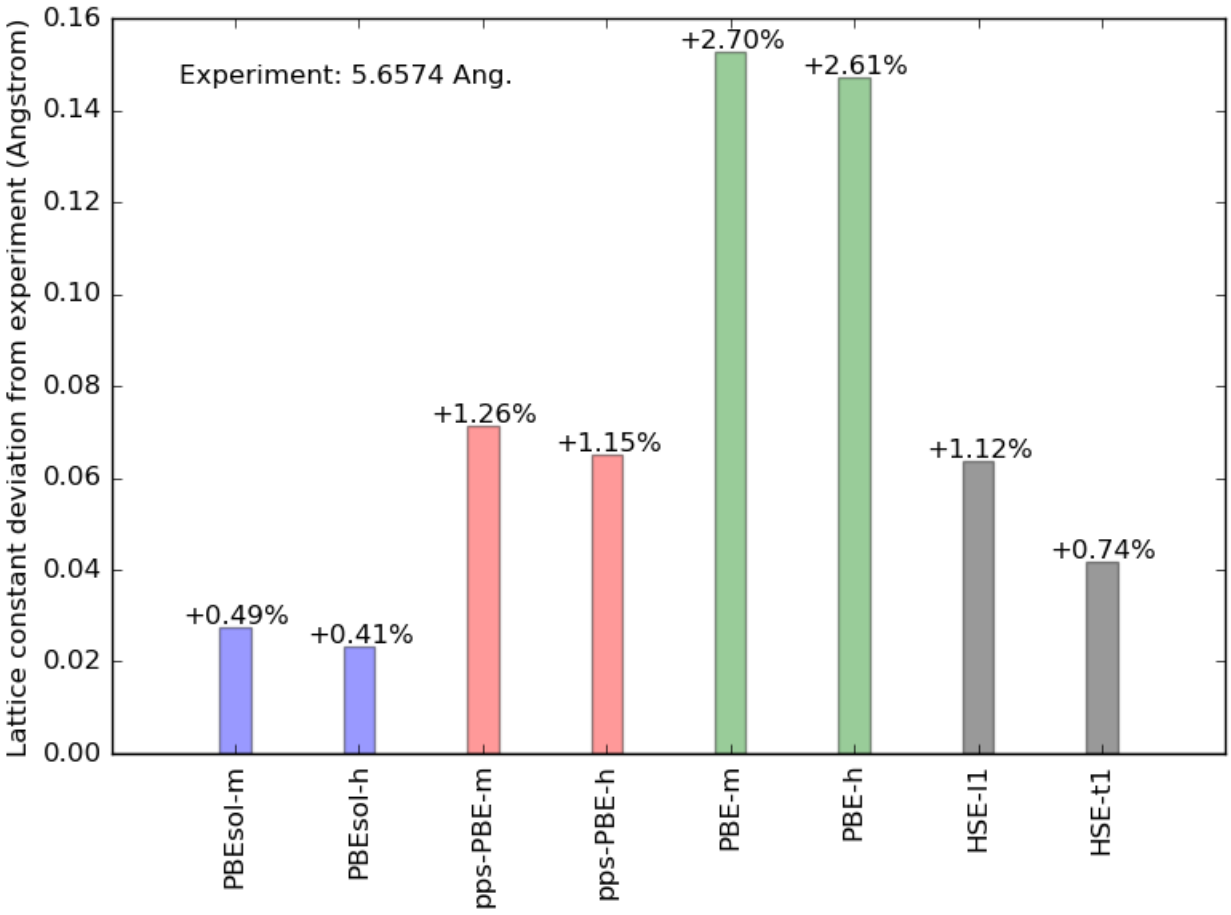


Fig. 3.55: Germanium lattice constant calculated using the PBEsol (blue), pps-PBE (red), PBE (green), and HSE (black) methods, plotted as deviation from the experimental value.

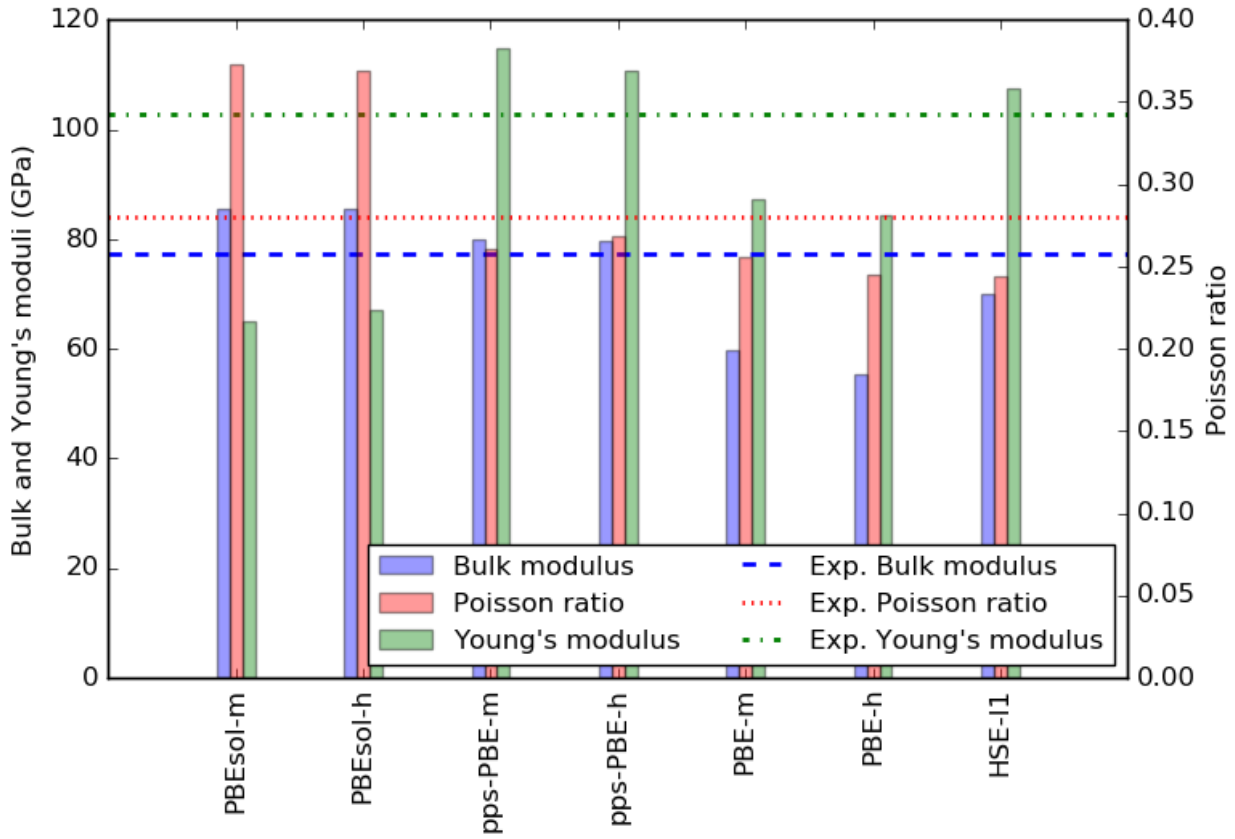


Fig. 3.56: Germanium bulk modulus (blue), Poisson ratio (red), and Youngs modulus (green) calculated using the PBEsol, pps-PBE, PBE, and HSE methods. The blue dashed, red dotted, and green dash-dotted lines indicate the experimental values, all from www.crystran.co.uk.

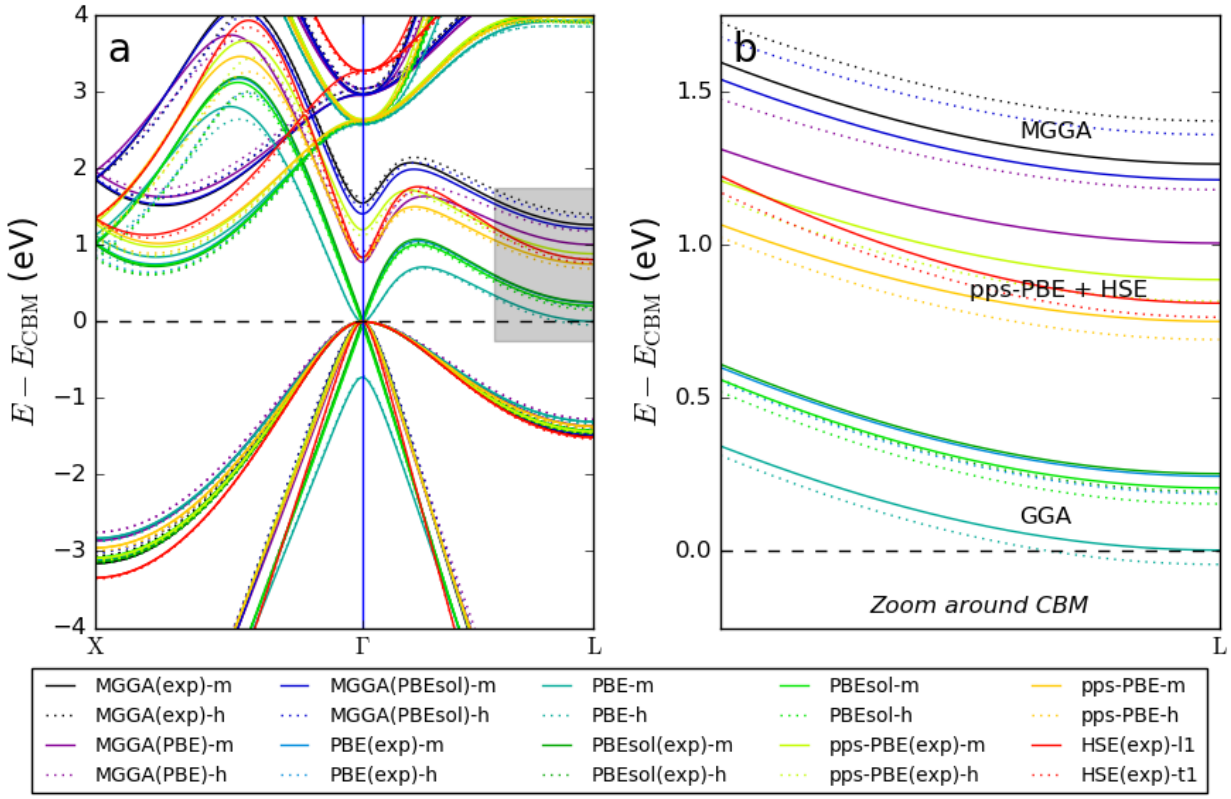


Fig. 3.57: Germanium band structure calculated with the TB09-MGGA, PBE, PBEsol, pps-PBE, and HSE methods. All band structures are aligned at the Γ-point valence band maximum. The shaded area in panel **a** indicates the region around the conduction band minimum (CBM) shown in panel **b**.

gap and lattice constant. On the other hand, the TB09-MGGA functional cannot be used for geometry optimization, which the pps-PBE method is well suited for.

We also observe that the band gaps may depend somewhat on the applied germanium lattice constant. For example, the pps-PBE E_{Γ_1} gap is significantly larger at the experimental lattice constant than at the pps-PBE lattice constant.

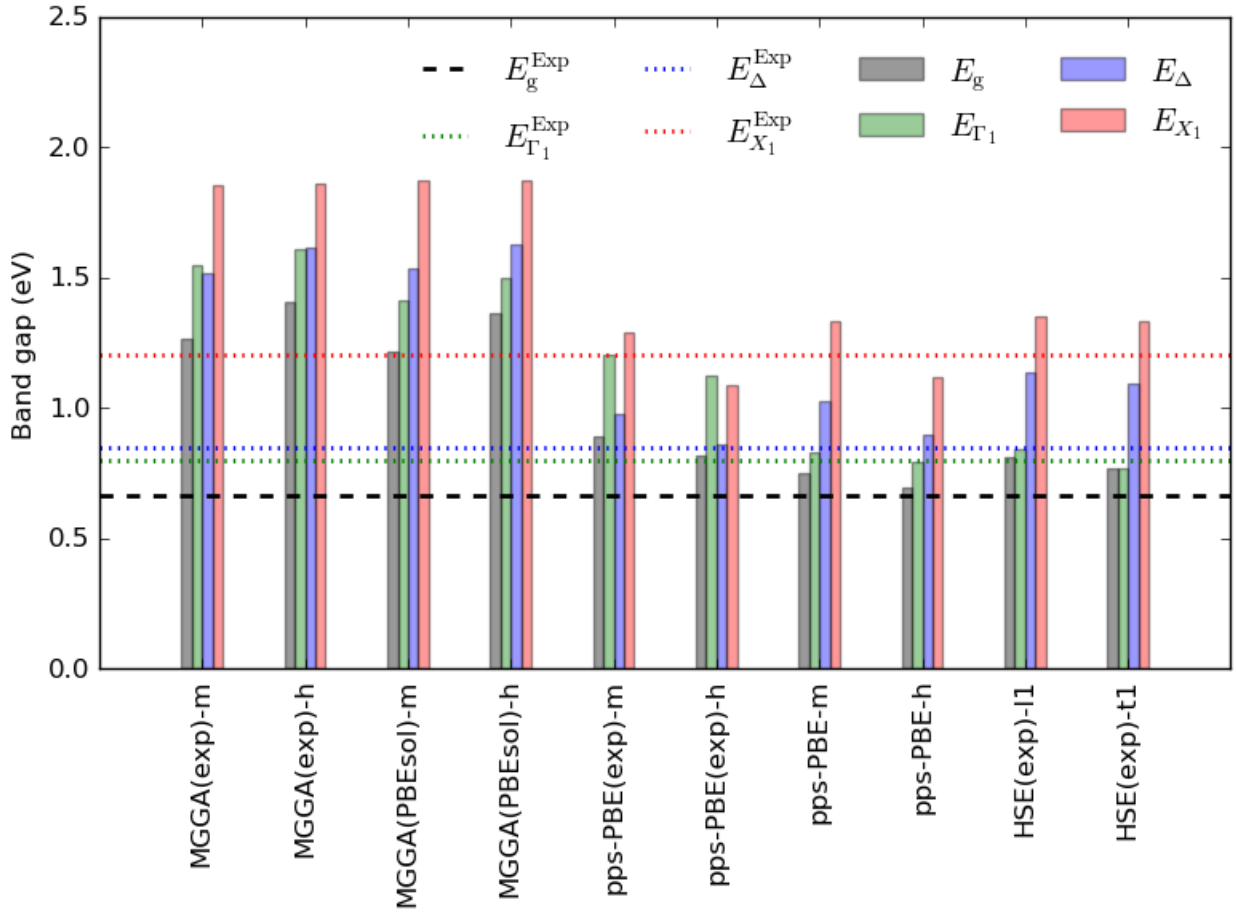


Fig. 3.58: Germanium band gaps; E_g (fundamental gap, black) and energy gaps from the Γ-point valence band maximum to the conduction band minima at X (red), Δ (X-point valley, blue), and Γ (green), calculated using the TB09-MGGA, pps-PBE, and HSE methods. Note that PBE and PBEsol calculations predict a qualitatively wrong band-gap ordering (often semi-metallic at Γ), so results from those methods are not included. TB09-MGGA at the PBE lattice constant yields a direct fundamental band gap, so TB09-MGGA(PBE) results are also not included. Experimental (300 K) values E_g^{Exp} (black, dashed), $E_{X_1}^{Exp}$ (red, dotted), E_Δ^{Exp} (blue, dotted), and $E_{\Gamma_1}^{Exp}$ (green, dotted) from http://www.ioffe.ru/SVA/NSM/Semicond/Ge/Figs/221.gif.

Electron effective masses

The figure below shows the computed transverse (blue; left y-axis) and longitudinal (red; right y-axis) effective masses for electrons in the germanium conduction band minimum at the L point. PBE and PBEsol data are included, even though those methods do not capture that the conduction band minimum is at the L point. Both electron effective masses are in general overestimated by the TB09-MGGA and pps-PBE methods, while PBEsol and PBE do better, especially for the transverse effective masses.

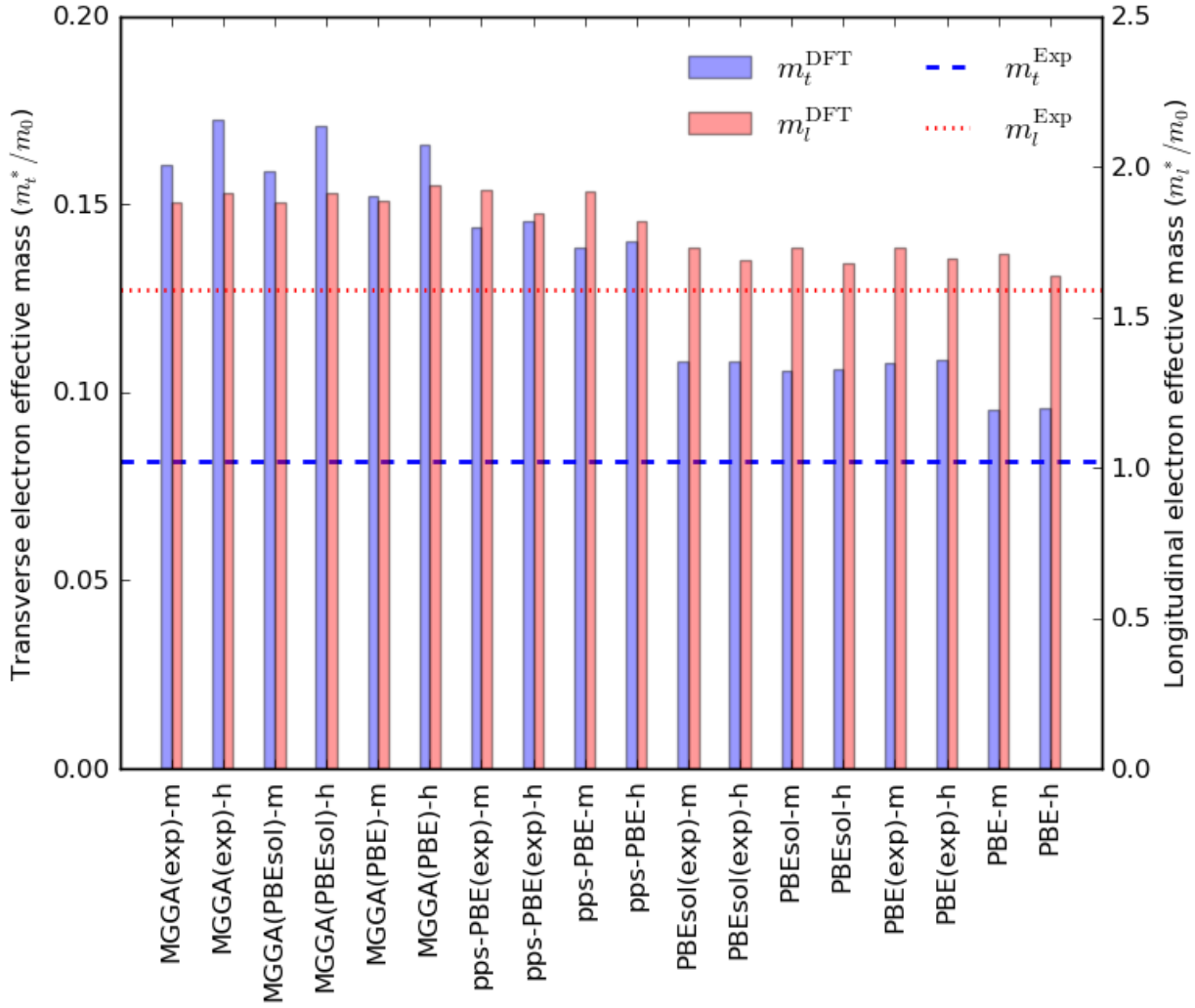


Fig. 3.59: Germanium transverse (blue; left y-axis) and longitudinal (red; right y-axis) electron effective masses calculated in the L point (the experimental conduction band minimum), using the TB09-MGGA, pps-PBE, PBEsol, and PBE methods. The blue dashed and red dotted lines mark the experimental values, adapted from http://www.ioffe.ru/SVA/NSM/Semicond/Ge/bandstr.html.

Dielectric constant

The static ($\omega = 0$) dielectric constant of the germanium bulk is most reliably reproduced by the pps-PBE method: The pps-PBE values in the figure below are in general closer to experiment than the TB09-MGGA values, while the PBEsol and PBE values are far too large. The latter is most likely due to the completely wrong PBEsol and PBE germanium band structure in the region around the Fermi level.

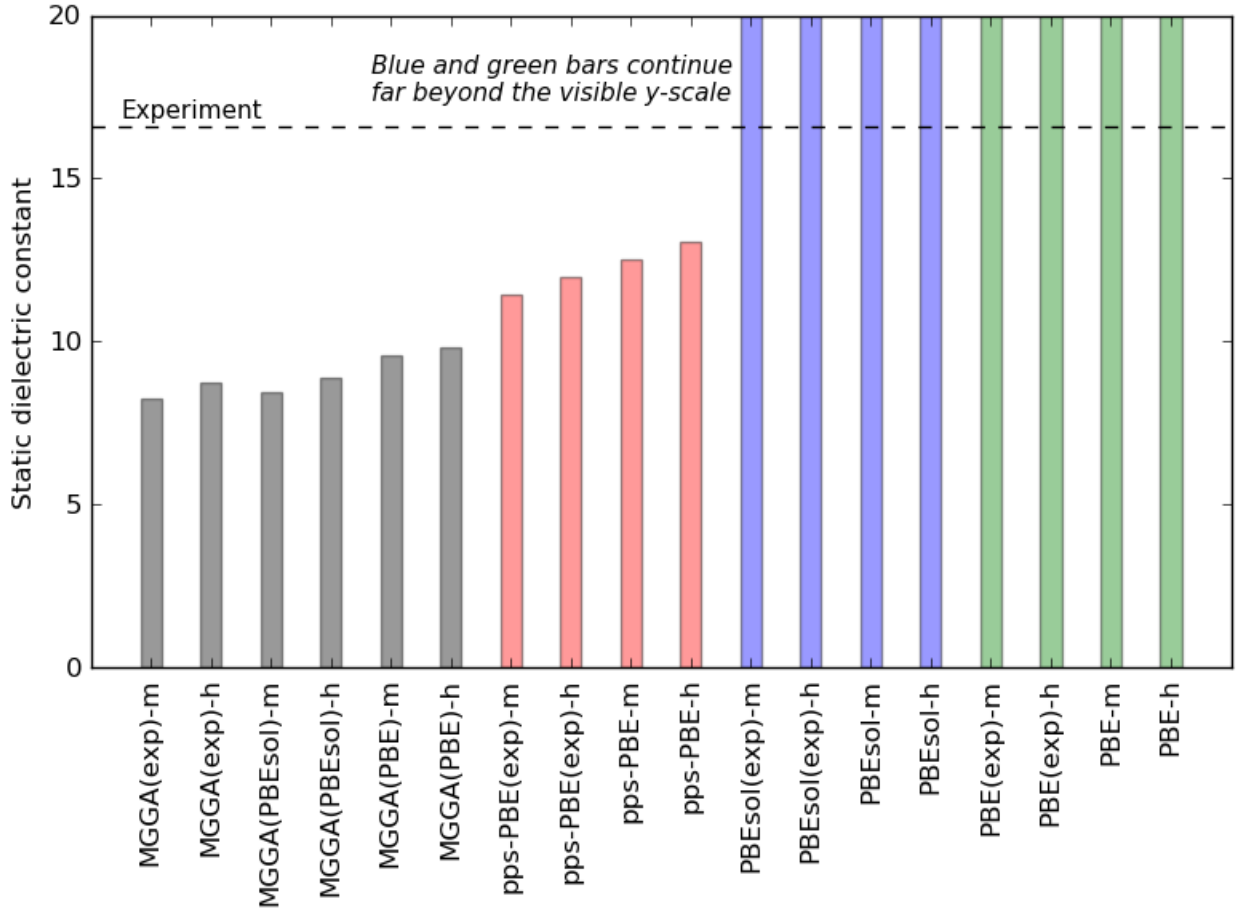


Fig. 3.60: Static dielectric constant of germanium calculated using the TB09-MGGA (black), pps-PBE (red), PBEsol (blue), and PBE (green) methods. The number of electronic bands included in the OpticalSpectrum analysis were 10 below the Fermi level and 20 above it. The dashed line indicates the experimental value, 16.6, from www.crystran.co.uk.

Appendix

The QuantumATK Python script shown below may be used as a template for ATK-DFT calculations for germanium with the SG15 pseudopotential. The script defines the germanium bulk configuration and then sets up the ATK-DFT calculator with PBE exchange-correlation. The script block named `Basis Set` shows various options for the SG15 basis set; ordinary PBE and PBE with pseudopotential projector shifts, both with Medium and High basis sets.

The script is available for direct download: `germanium.py`.

```
# ------------------------------------------------------------
# Bulk Configuration
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.6574*Angstrom)

# Define elements
elements = [Germanium, Germanium]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# -----------------------------------------------------------
# Calculator
# -----------------------------------------------------------
#----------------------------------------
# Basis Set
#----------------------------------------
# Ordinary GGA: Medium or High basis set for SG15.
basis_set = [BasisGGASG15.Germanium_Medium]
#basis_set = [BasisGGASG15.Germanium_High]

# GGA with pseudopotential projector-shift method: Medium or High basis set for SG15.
#projector_shift = PseudoPotentialProjectorShift(s_orbital_shift=15.0*eV,
#                                                p_orbital_shift=0.2*eV,
#                                                d_orbital_shift=-2.0*eV)
#basis_set = [BasisGGASG15.Germanium_Medium(projector_shift=projector_shift)]
#basis_set = [BasisGGASG15.Germanium_High(projector_shift=projector_shift)]

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = GGA.PBE

k_point_sampling = MonkhorstPackGrid(
    na=9,
    nb=9,
    nc=9,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=100.0*Hartree,
    )

iteration_control_parameters = IterationControlParameters(
```

(continues on next page)

(continued from previous page)

```
    damping_factor=0.4,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('Germanium.nc', bulk_configuration)
```

3.2.21 $Si_{0.5}Ge_{0.5}$

Here we investigate computational methods for ATK-DFT calculations for a simple $Si_{0.5}Ge_{0.5}$ alloy in the diamond crystal structure, i.e., with an atom of each element in the basis (in this document also denoted SiGe for brevity). We apply the SG15 pseudopotentials and a number of exchange-correlation functionals; PBE, PBEsol, TB09-MGGA, and PBE with the *Pseudopotential Projector-Shift* method. A limited number of HSE calculatios have been done for comparison. See the section *Introduction* for more information.

The following **quantities** have been calculated for each computational method, both at the experimental $Si_{0.5}Ge_{0.5}$ lattice constant and at the theoretically predicted ones:

•Elastic constants: Bulk modulus, Poisson ratio, and Youngs modulus.

•Band structure along the X—L Brillouin zone path.

•Direct and indirect band gaps.

•Effective masses in the conduction band minimum.

•Static dielectric constant.

Convergence studies indicate that a 8x8x8 k-point grid and a 100 Hartree density mesh cutoff yields well-converged values for the $Si_{0.5}Ge_{0.5}$ lattice constant, total energy and indirect band gap. These settings were used for computing the ground state electronic structure used in all the ATK-DFT analyses listed above.

The projector shifts used for the **pps-PBE** method are 11.23 eV for Si s-orbitals and -1.09 eV for Si p-orbitals, while shifts of 15.0 eV are used for Ge s-orbitals, 0.2 eV for Ge p-orbitals, and -2.0 eV for Ge d-orbitals. Note that these PPS parameters should only be used with SG15 pseudopotentials.

This document is organized as follows. The section *Convergence* briefly presents the convergence studies mentioned above, and the section *Timing* compares CPU timings and scalability of different SG15 basis sets for $Si_{0.5}Ge_{0.5}$. Next, the section *Results* contains figures illustrating the performance of the different computational methods in predicting the materials properties listed above. Finally, the *Appendix* gives a template QuantumATK Python script for setting up ATK-DFT calculations for $Si_{0.5}Ge_{0.5}$ similar to the ones presented here.

Summary

The following list summarizes the main conclusions from this study.

- The PBEsol, and to a lesser degree pps-PBE, methods both yield very accurate $Si_{0.5}Ge_{0.5}$ **lattice constants**, and also give the best results for elastic constants.
- All tested methods yield a qualitatively correct $Si_{0.5}Ge_{0.5}$ **band structure**, but with significant differences between the different methods. The method which is closest to the HSE reference is pps-PBE.
- The pps-PBE method accurately reproduces the fundamental (indirect) **band gap**. There is a large spread in the results for the direct gaps, so care is needed to describe these accurately.
- **Electron effective masses** in the Δ-point CBM are quite well captured by all tested methods.
- Even though PBE and PBEsol do not describe the band gaps as accurately as pps-PBE, they are actually significantly better for the **static dielectric constant**. It is underestimated by the other methods.

Convergence

The type of Monkhorst–Pack k-point grid and the density mesh cutoff energy are essential ATK-DFT parameters that affect the computational efficiency and precision: A dense k-point grid and high cutoff energy usually give high precision, but may also be computationally intense. It is therefore important to investigate the trade-off between computational precision and cost. Fig. 3.61 illustrates how the bulk total energy, fundamental band gap, and lattice constant depend on the mesh cutoff and k-point grid.

Note that a standard Monkhorst–Pack k-point grid is -centered for odd numbers of points (e.g. 3x3x3), but does not include the point for even numbers of points (e.g. 4x4x4). The blue lines in Fig. 3.61 are for k-point grids where even grids are shifted to (such that both 3x3x3 and 4x4x4 grids **do** include), while the red lines are for calculations where odd grids are shifted away from (such that both 3x3x3 and 4x4x4 grids **do not** include).

For the SG15 pseudopotential with both Medium and High basis sets, a 100 Hartree mesh cutoff gives band gaps and lattice constants that are converged to within 10^{-4} eV and roughly 10^{-4} Å, respectively.

It is also clear that an off- k-point grid (non-centered, red) gives faster convergence of the total energy and lattice constant than the k-point grid that includes the point (centered, blue). We see that the non-centered 8x8x8 Monkhorst–Pack grid yields highly converged lattice constants.

Note: All remaining ATK-DFT calculations in this study therefore use a 100 Hartree mesh cutoff energy and a standard (non-centered) 8x8x8 Monkhorst–Pack k-point grid for both elastic properties and electronic properties.

Timing

The computational cost of a calculation may depend critically on the choice of basis set, particularly for calculations on large systems with many atoms. Fig. 3.62 shown below illustrates this for the present case of $Si_{0.5}Ge_{0.5}$ with the Medium, High, and Ultra SG15 basis sets.

The SiGe primitive cell (containing 2 atoms) was systematically repeated in steps of one along all three lattice vectors, resulting in rapidly increasing systems sizes. PBE calculations were then performed for each system, using -point sampling only, to avoid any influence of k-point sampling on the recorded CPU times.

It is quite clear from the figure that the computational loads of the SG15 Medium and High basis sets are not much different for relatively small system sizes, but the High basis set becomes significantly more demanding for larger system sizes. For the 432-atom unit cell, the CPU time difference is roughly a factor of 2. Moreover, the Ultra basis set appears in comparison extremely demanding, and will not be used in the remaining calculations in this study.

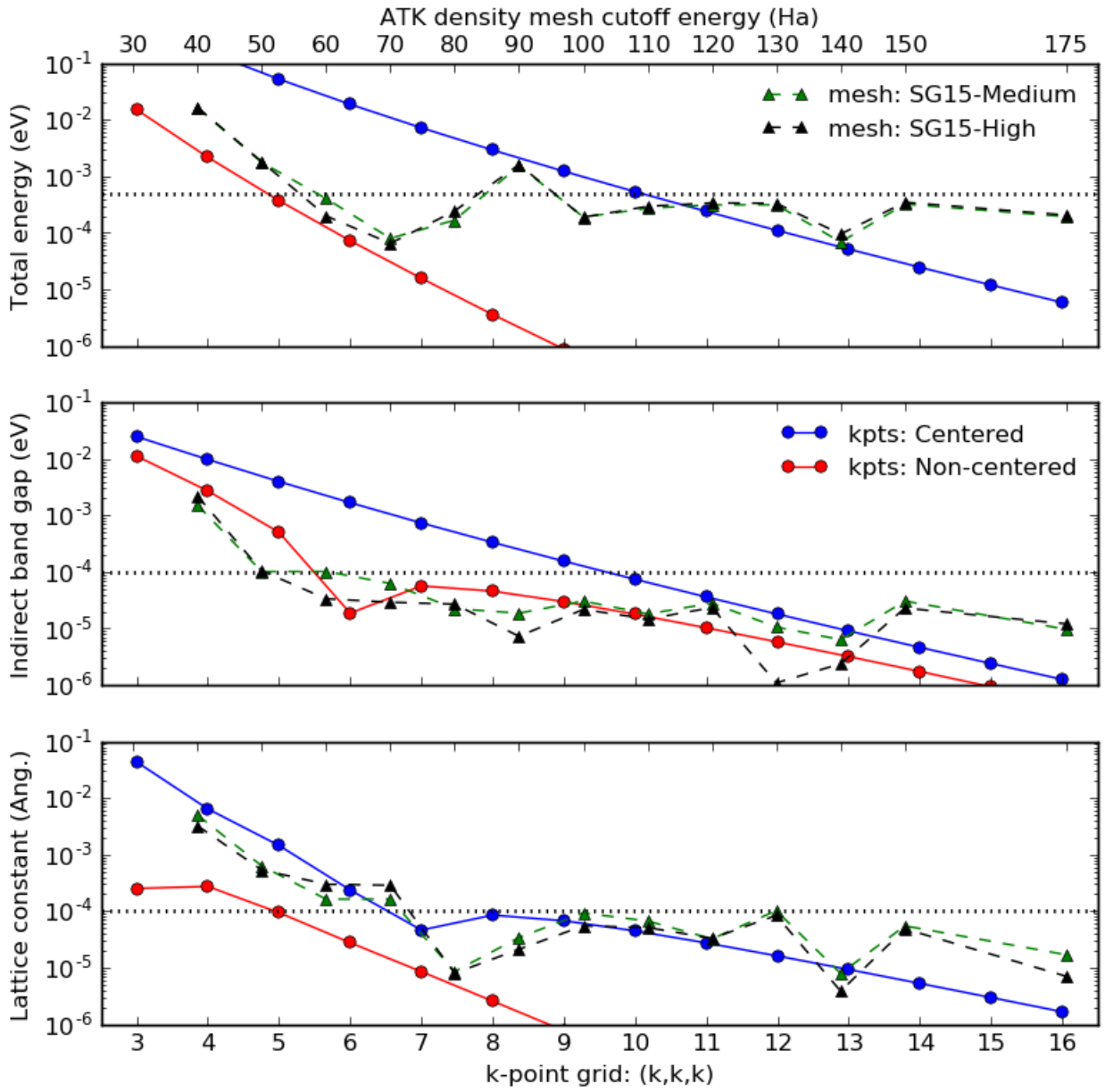


Fig. 3.61: Convergence of the SiGe total energy, fundamental (indirect) band gap, and lattice constant with respect to the ATK-DFT density mesh cutoff (top x-axis in all three panels; green and black dashed lines) and k-point grid (bottom x-axis in all three panels; blue and red lines). The centered k-point grids are shifted to the Γ point (only affects even grids, e.g. 4x4x4), while the non-centered k-point grids are shifted by (0.5,0.5,0.5) and therefore do not sample the Γ point (only affects odd grids, e.g. 3x3x3). The horizontal dotted lines indicate tight convergence criteria.

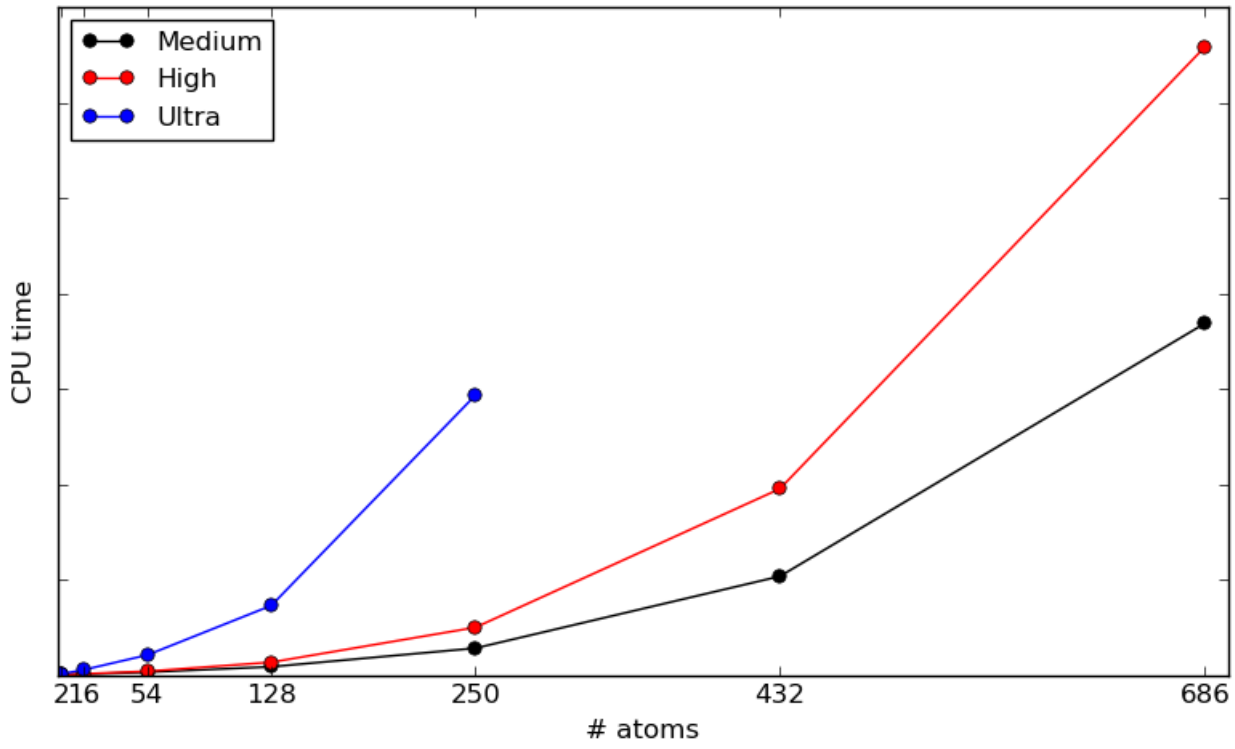


Fig. 3.62: Total CPU time for the first 5 SCF iterations by the ATK-DFT calculator for the Medium (black), High (red), and Ultra (blue) SG15 basis sets, against the number of atoms in the increasingly larger (repeated) SiGe bulk unit cell. Only the point is sampled (1x1x1 k-point grid) and the calculations were executed on a single CPU core. Note that both figure axes are linear.

Results

This section presents results obtained with the PBE, PBEsol, pps-PBE, and TB09-MGGA methods using Medium and High basis sets with the SG15 pseudopotentials for $Si_{0.5}Ge_{0.5}$.

The cited experimental reference values are in general for a 50/50 mix of Si and Ge, but with no local order in the distribution of Si and Ge on the lattice. There may therefore be a slight difference between the experimentally investigated system and the very small model system used here.

The list below gives direct links to all subsections:

•*Lattice constant*

•*Elastic constants*

•*Band structure*

•*Band gaps*

•*Effective masses*

•*Dielectric constant*

Lattice constant

The PBEsol lattice constant for $Si_{0.5}Ge_{0.5}$ is very close to the experimental value, 5.538 Å, but the pps-PBE method is also quite accurate. The bare PBE functional (without any pseudopotential projector shifts) is known to overestimate lattice constants in general, and in this case it overestimates by more than 1%. This is in contrast to pure silicon where PBE only overestimates the lattice constant by 0.6%. HSE performs better than PBE, but slightly worse than pps-PBE. However, HSE is expected to perform much better for the details of the electronic structure.

In all cases, except for PBEsol, we see a significantly smaller error for the better basis set (High or Tight).

Elastic constants

Both PBESol and pps-PBE hit close to the experimental bulk modulus, while regular PBE underestimates it. The same trend is observed for Youngs modulus, but without a reference value for comparison. The Poisson ratio is approximately the same for all methods.

Band structure

All tested methods correctly predict a valence band maximum (VBM) at the Brillouin zone point, and most also predict a conduction band minimum (CBM) close to the X point, along the –X path. However, the position of the CBM depends slightly on the method applied, and the band energy in the CBM significantly so (see also section *Band gaps*). In fact, for PBE the CBM may shift to the L point depending on basis set.

There is also a qualitative difference between HSE and pps-PBE at the X point. HSE has a gap of about 0.3 eV between the two lowest conduction bands, while they (almost) touch for pps-PBE.

Changing the basis set from Medium to High gives a shift of about 0.1 eV at the X point, and quickly decreasing towards . In all cases, the shift brings the result closer to the HSE values. For HSE, there is almost no difference between Light and Tight basis sets.

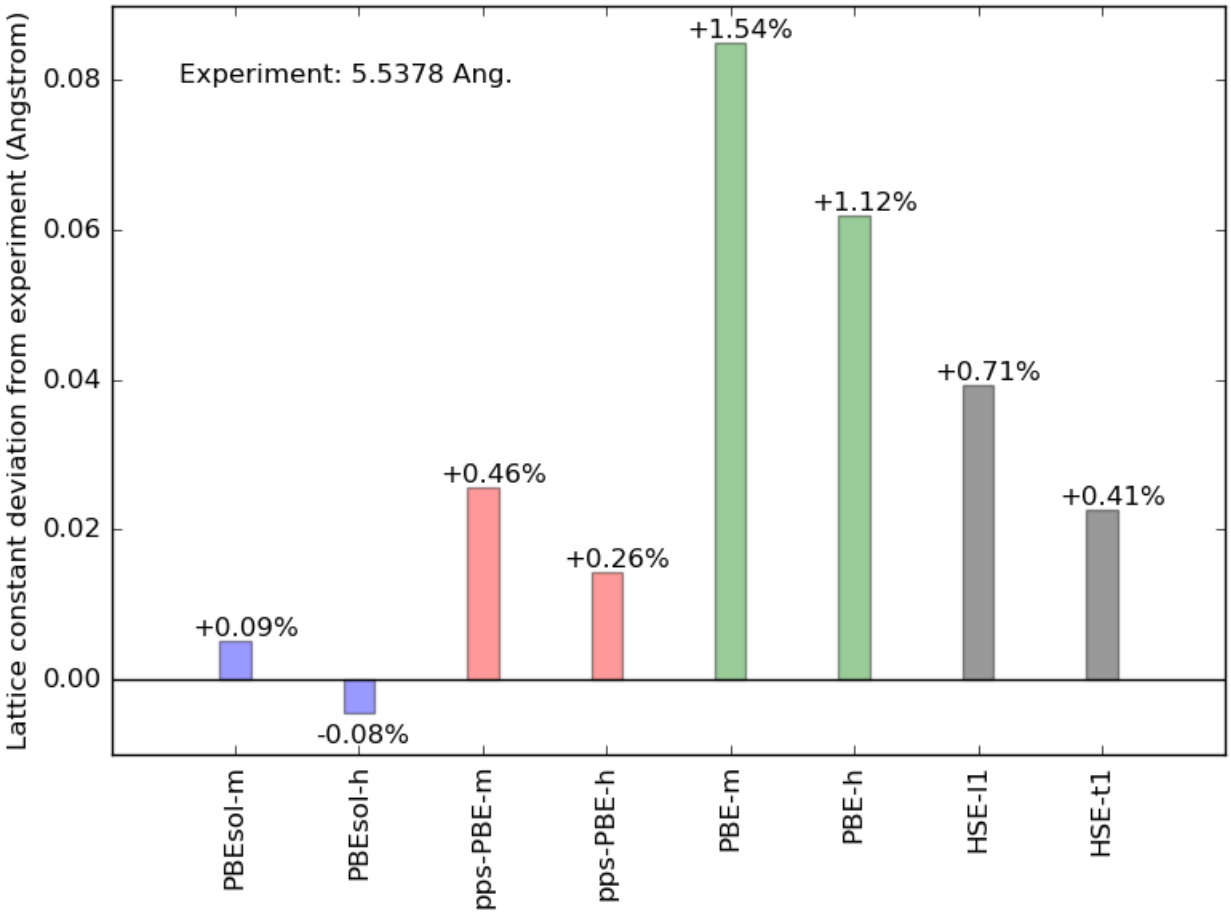


Fig. 3.63: SiGe lattice constant calculated using the PBEsol (blue), pps-PBE (red), PBE (green), and HSE (black) methods, and two different qualities of basis sets, plotted as deviation from the experimental value from the Ioffe website.

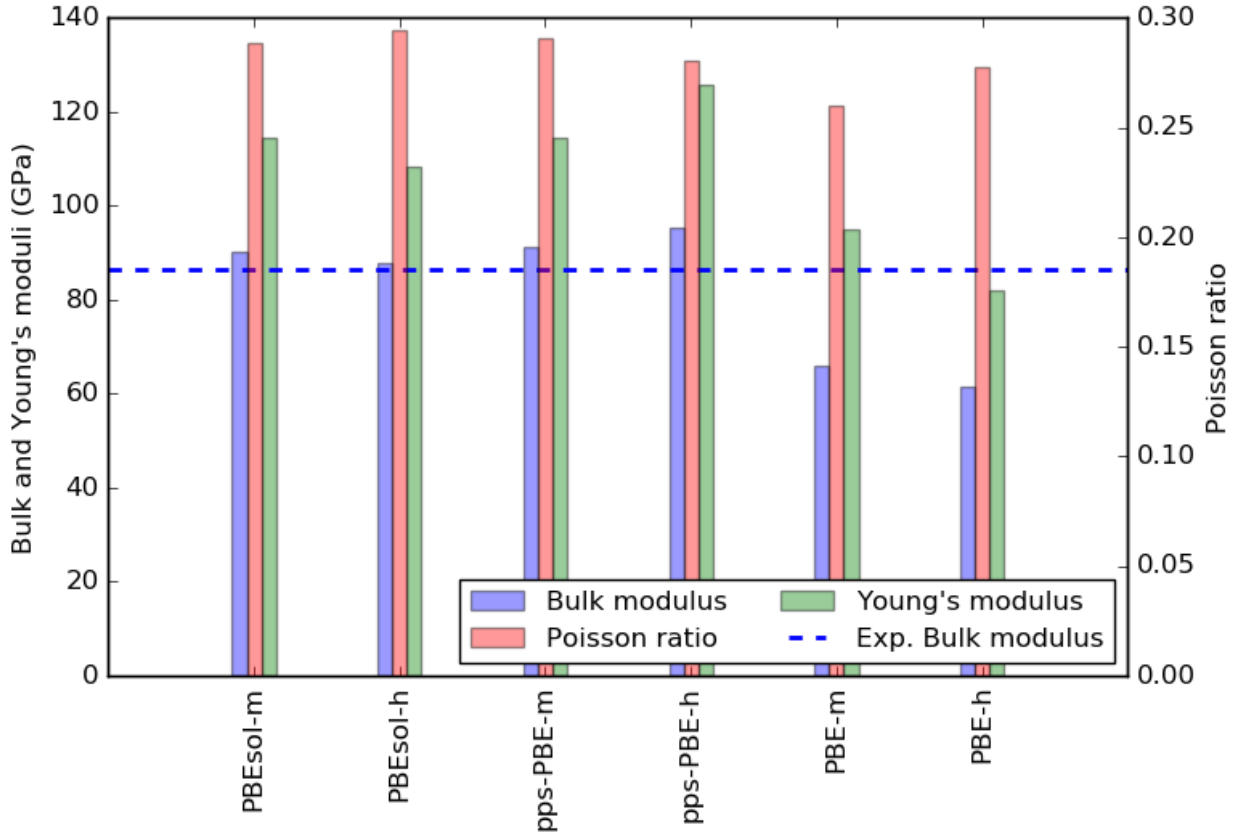


Fig. 3.64: SiGe bulk modulus (blue), Poisson ratio (red), and Youngs modulus (green) calculated using the PBEsol, pps-PBE, and PBE methods, and two different qualities of basis sets. The blue dashed line indicates the experimental bulk modulus, 86.5 GPa, adapted from the Ioffe website.

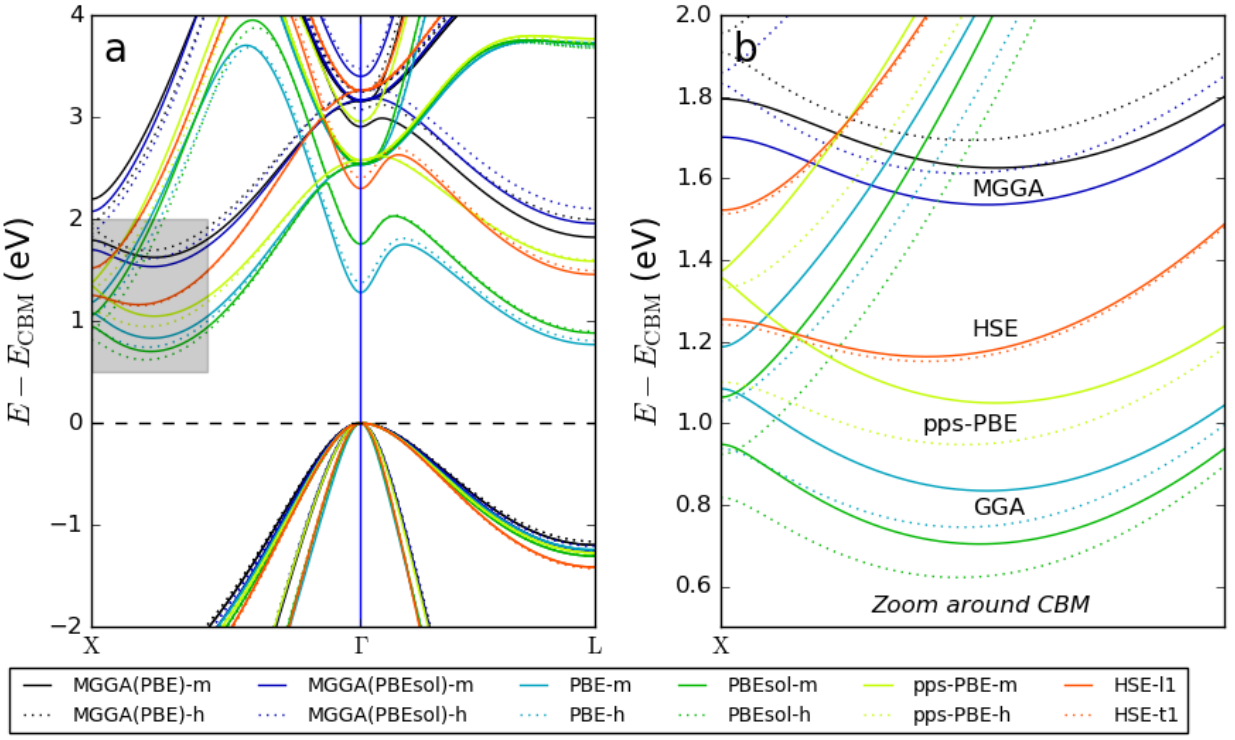


Fig. 3.65: SiGe band structure calculated with the PBE, pps-PBE, PBEsol, TB09-MGGA and HSE exchange-correlation functionals with two different qualities of basis sets. All bandstructures are calculated with the equilibrium structure for that computational model, except for TB09-MGGA, which has been calculated with both the PBE and PBEsol lattice constants.

Band gaps

The SiGe fundamental band gap (E_Δ) is especially well reproduced by the pps-PBE method, but also HSE, regular PBE and PBESol give reasonable agreement with experiment. Note, however, that the TB09-MGGA c-parameter was calculated self-consistently from the electronic structure (no fitting), while the pps-PBE projector shifts were essentially fitted to the silicon and germanium band gaps and lattice constants. On the other hand, the TB09-MGGA functional cannot be used for geometry optimization, which the pps-PBE method is well suited for.

There is much less agreement between the methods when it comes to the direct gaps, with TB09-MGGA consistently giving larger values, PBE and PBESol giving smaller values, and pps-PBE and HSE in between.

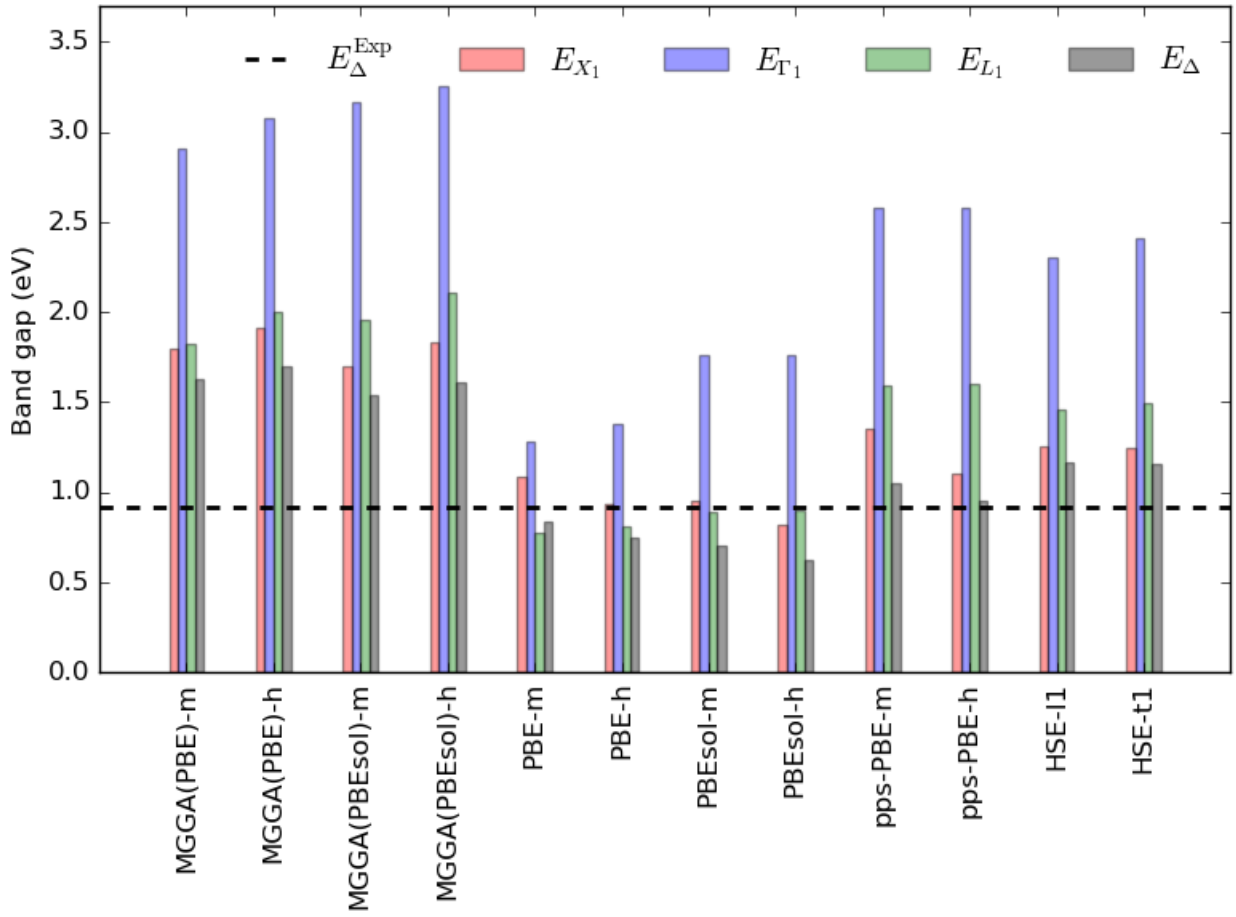


Fig. 3.66: SiGe band gaps, including both the fundamental gap (E_Δ, indirect), and the direct gaps to the X, Γ, and L points. The experimental E_Δ of 0.917 eV is from the Ioffe website, and is indicated by a black dashed line.

We also evaluate the effect of using HSE for band gap calculations. The figure below shows GGA and HSE band gaps of SiGe computed at the GGA lattice constants. The HSE indirect (fundamental) gaps are all roughly the same, while the direct gap at the Γ point does vary with lattice constant. The latter is probably due to strain effects. The PBE and PBEsol gaps are all significantly smaller than the corresponding HSE ones, as expected. On the other hand, the pps-PBE gaps are quite similar to the corresponding HSE gaps.

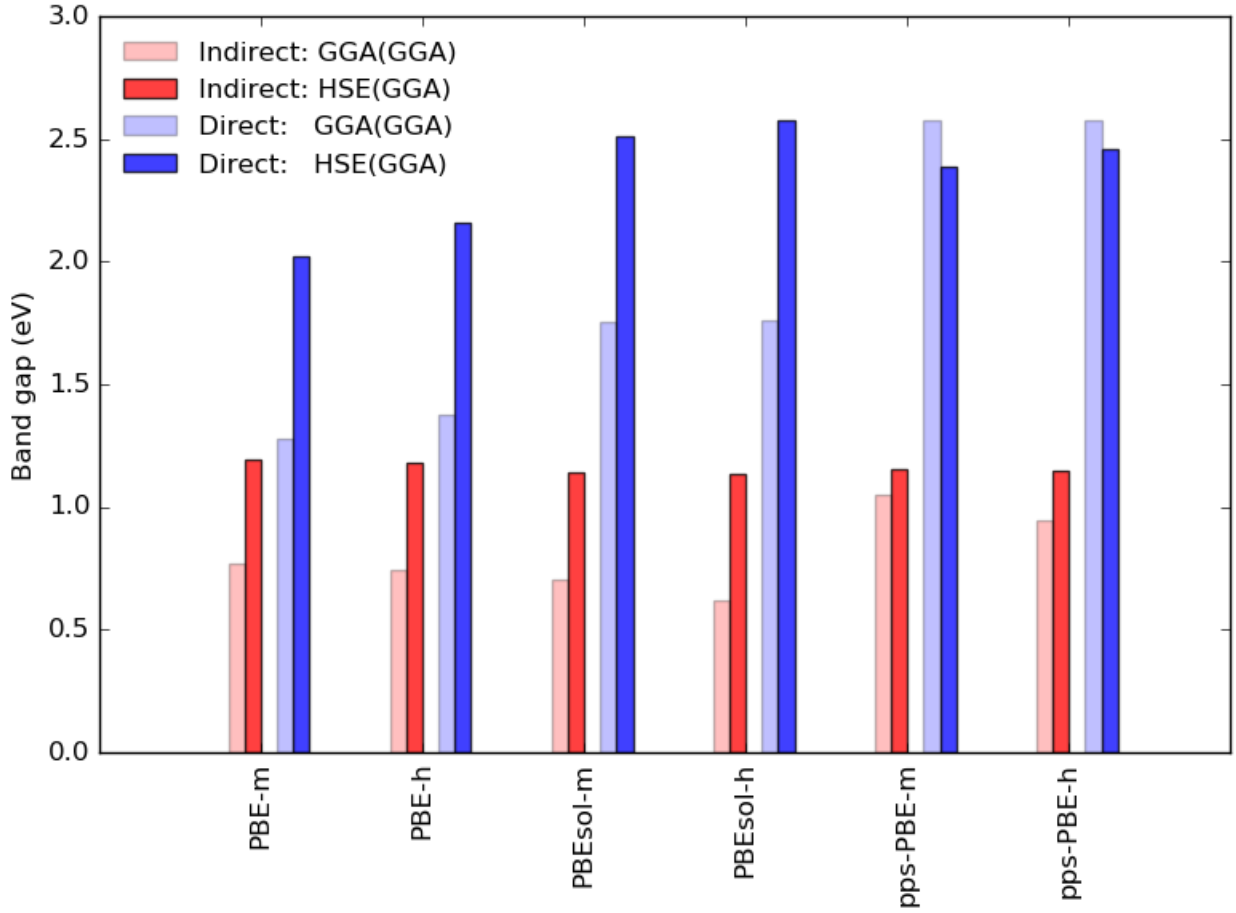


Fig. 3.67: Comparison of GGA and HSE indirect (E_{Δ}) and direct (E_{Γ_1}) band gaps at GGA lattice constants.

Effective masses

The SiGe electron effective masses in the Δ-point CBM are fairly close to experimental values for all applied methods.

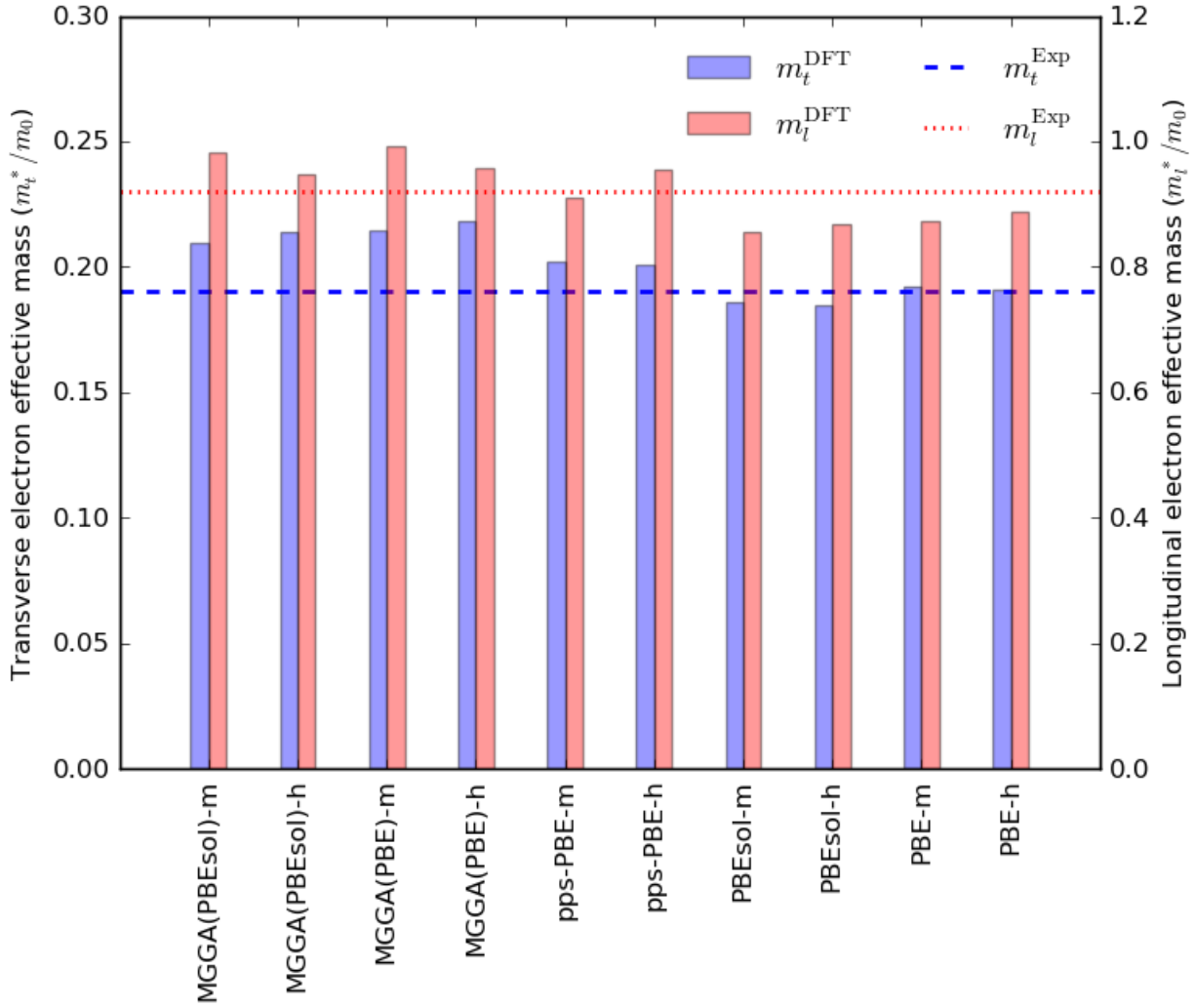


Fig. 3.68: SiGe transverse (blue; left y-axis) and longitudinal (red; right y-axis) electron effective masses calculated in the Δ-point conduction band minimum. Experimental effective masses are from the Ioffe website.

Dielectric constant

The static ($\omega = 0$) dielectric constant of the SiG lattice is most reliably reproduced by the PBE and PBEsol functionals, with the pps-PBE method a bit off, and TB09-MGGA more than 30% off. The latter is probably related to the fact that TB09-MGGA severely overestimates the direct band gap, while PBE and PBEsol are closer to the experimental direct gap. However, the fact pps-PBE gives a very good direct gap, but is somewhat off for the static dielectric constant, may indicate that the overall bandstructure is not as accurately described as the direct gap itself. We see almost no dependence of the dielectric constant on the basis set quality.

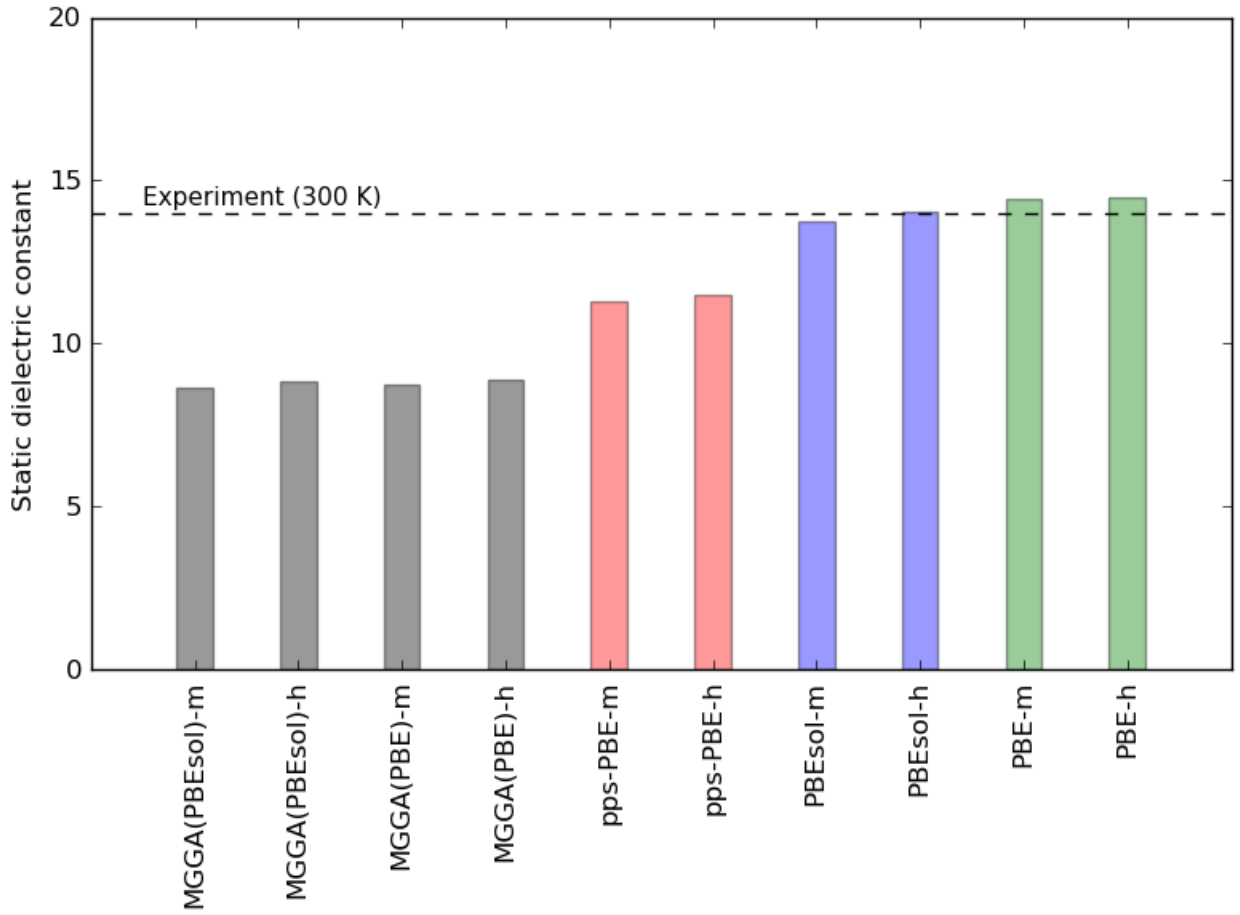


Fig. 3.69: Static dielectric constant of SiGe calculated with PBE, PBEsol, TB09-MGGA and pps-PBE methods, and two qualitites of basis sets. The experimental static dielectric constant of 14.0 is from the Ioffe website.

Appendix

The QuantumATK Python script shown below may be used as a template for ATK-DFT calculations for $Si_{0.5}Ge_{0.5}$ with the SG15 pseudopotential. The script defines the SiGe bulk configuration and then sets up the ATK-DFT calculator with PBE exchange-correlation. The script blocked named `Basis Set` shows various options for the SG15 basis set; ordinary PBE and PBE with pseudopotential projector shifts, both with Medium and High basis sets.

The script is available for direct download: `si50ge50.py`.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.533*Angstrom)

# Define elements
elements = [Silicon, Germanium]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
# Ordinary GGA: Medium or High basis set for SG15.
basis_set = [BasisGGASG15.Silicon_Medium,
             BasisGGASG15.Germanium_Medium]
#basis_set = [BasisGGASG15.Silicon_High,
#             BasisGGASG15.Germanium_High]

# GGA with pseudopotential projector-shift method: Medium or High basis set for SG15.
#shift_si = PseudoPotentialProjectorShift(s_orbital_shift=11.23*eV,
#                                         p_orbital_shift=-1.09*eV)
#shift_ge = PseudoPotentialProjectorShift(s_orbital_shift=15.0*eV,
#                                         p_orbital_shift=0.2*eV,
#                                         d_orbital_shift=-2.0*eV)
#basis_set = [BasisGGASG15.Silicon_Medium(projector_shift=shift_si),
#             BasisGGASG15.Germanium_Medium(projector_shift=shift_ge)]
#basis_set = [BasisGGASG15.Silicon_High(projector_shift=shift_si),
#             BasisGGASG15.Germanium_High(projector_shift=shift_ge)]

#---------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Exchange-Correlation
#---------------------------------------
exchange_correlation = GGA.PBE

k_point_sampling = MonkhorstPackGrid(
    na=8,
    nb=8,
    nc=8,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=100.0*Hartree,
    )

iteration_control_parameters = IterationControlParameters(
    damping_factor=0.5,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('SiGe_5050.nc', bulk_configuration)
```

References

3.3 Batteries & Energy Storage

3.3.1 Li-air battery interface

Version: 2015.1

Li-ion batteries are, at the moment, the most widely used in most most electric vehicles and hybrid electric vehicles. However, there are some disadvantages such as high price, slow charging, and low energy/power density. Li-air batteries are attracting attention due to the higher energy storing capacity and are being seen as possible alternative for Li-ion batteries. Nevertheless, still a lot of research has to be done in order to make Li-air battery competitive. In particular, various complex chemical and electrochemical side-reactions occurs at the interfaces. Li_2O_2/Li_2CO_3 is indeed a very important interface since Li_2CO_3 is formed at the cathode with Li_2O_2 when carbonate-based electrolytes are used. In this tutorial you will use **QuantumATK** to create the Li_2O_2/Li_2CO_3 and use **ATK** study its electronic properties. You will compare the results with Ref.[1].

You will:

[1] YedilfanaăS. Mekonnen, JuanăM. Garcia-Lastra, JensăS. Hummelshøj, Chengjun Jin, and Tejs Vegge. Role of Li2O2-Li2CO3 Interfaces on Charge Transport in Nonaqueous Li-Air Batteries. *The Journal of Physical Chemistry C*, 119(32):18066–18073, 2015. doi:10.1021/acs.jpcc.5b04432.

•Import/create Li_2O_2 and Li_2CO_3 bulk structures.

•Cleave surfaces.

•Create Li_2O_2/Li_2CO_3 interface.

•Create device and optimize geometry.

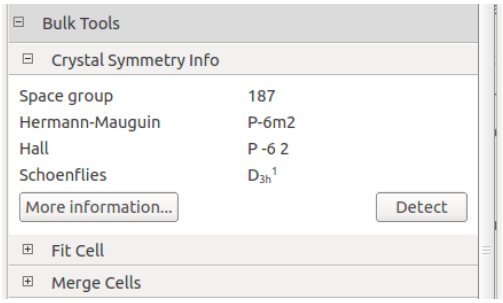
•Calculate electronic properties of the Li_2O_2/Li_2CO_3.

Li_2O_2 bulk and surface structures

Bulk structure

•Open the **Builder** and use *Add* → *From Database* to add the Li_2O_2 structure to the **Stash**.

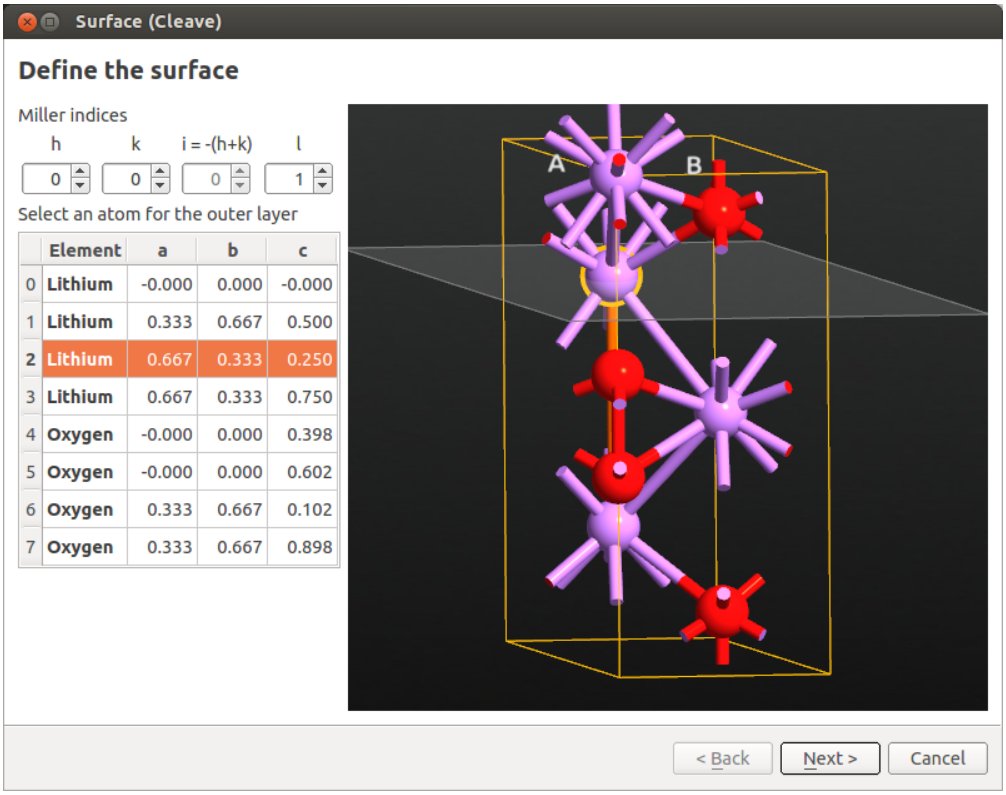
Tip: You can use the *Bulk Tools* → *Crystal Symmetry Info* plugin to get the corresponding space group, for example

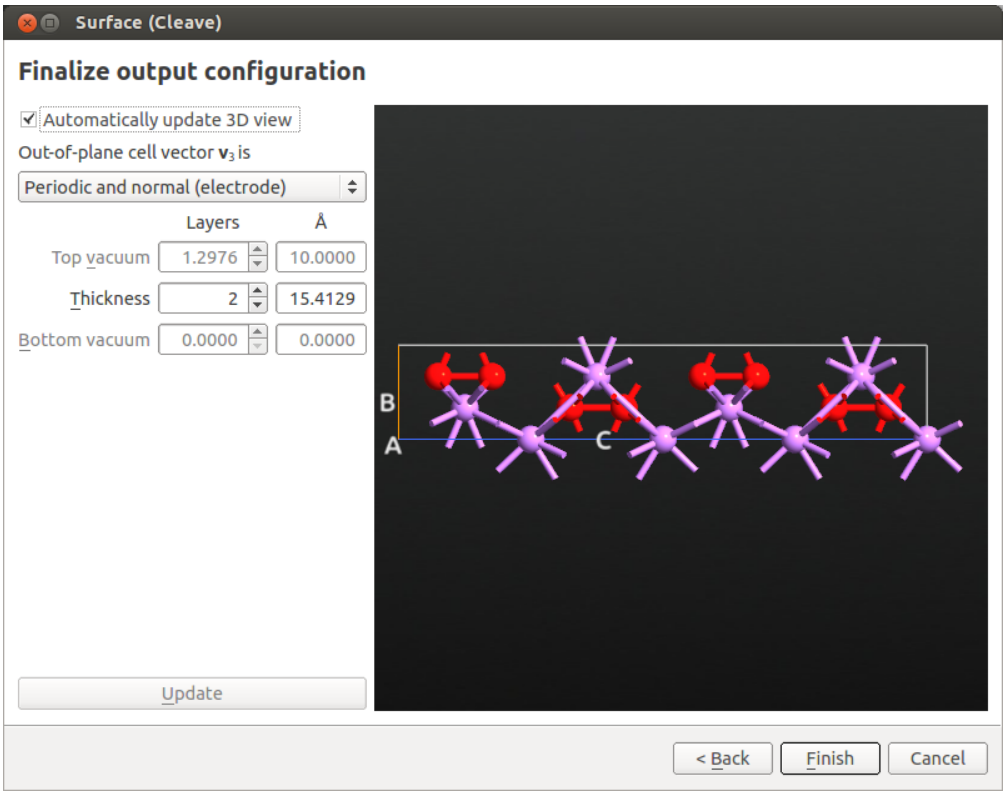


•Optimize the lattice parameters by running the `Li2O2_opt.py` script or use the experimental ones if you wish. In this tutorial you will use the GGA-RBE exchange-correlation potential by following the computational details as in Ref.[Page 729, 1].

Cleave surface

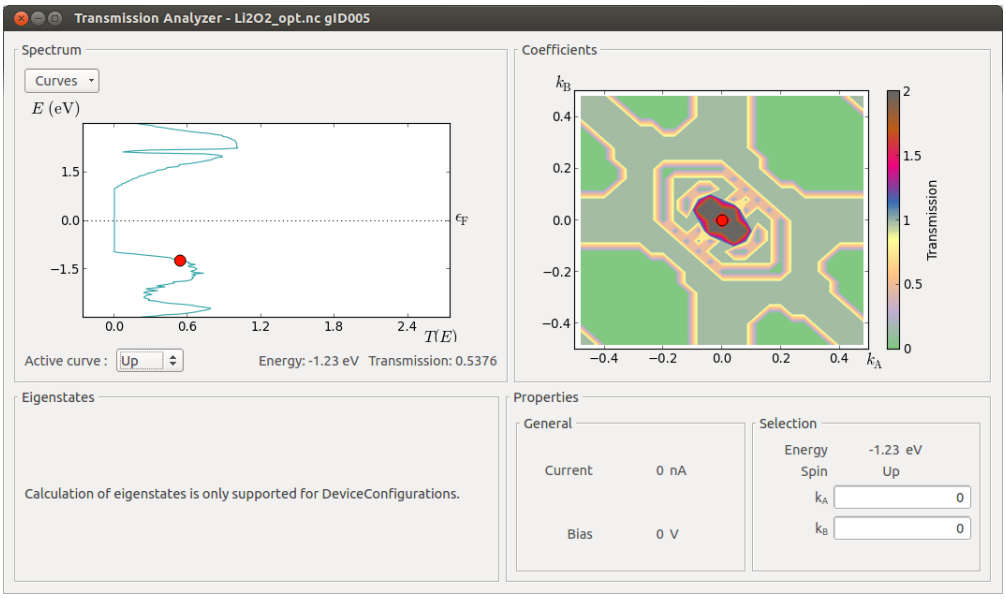
•Use the *Builders* → *Surface(Cleave)* tool to cleave the Li_2O_2 (0001) surface from the optimized Li_2O_2 structure as illustrated in the next figures.


Surface (Cleave)
Define the surface
Miller indices
h
k
i = -(h+k)
l
0
0
0
1
Select an atom for the outer layer
Element a b c
0 Lithium -0.000 0.000 -0.000
1 Lithium 0.333 0.667 0.500
2 Lithium 0.667 0.333 0.250
3 Lithium 0.667 0.333 0.750
4 Oxygen -0.000 0.000 0.398
5 Oxygen -0.000 0.000 0.602
6 Oxygen 0.333 0.667 0.102
7 Oxygen 0.333 0.667 0.898
A
B
< Back
Next >
Cancel


Surface (Cleave)
Finalize output configuration
Automatically update 3D view
Out-of-plane cell vector $\mathbf{v}_3$ is
Periodic and normal (electrode)
Layers
Å
Top vacuum 1.2976 10.0000
Thickness 2 15.4129
Bottom vacuum 0.0000 0.0000
Update
B
A
C
< Back
Finish
Cancel

Electronic structure at zero bias

In the `Li2O2_opt.py` script a few analysis have been added such as the *TransmissionSpectrum* analysis which $T(\mathbf{k}, \epsilon)$ you can find loaded in the **LabFloor** once your simulation is done.



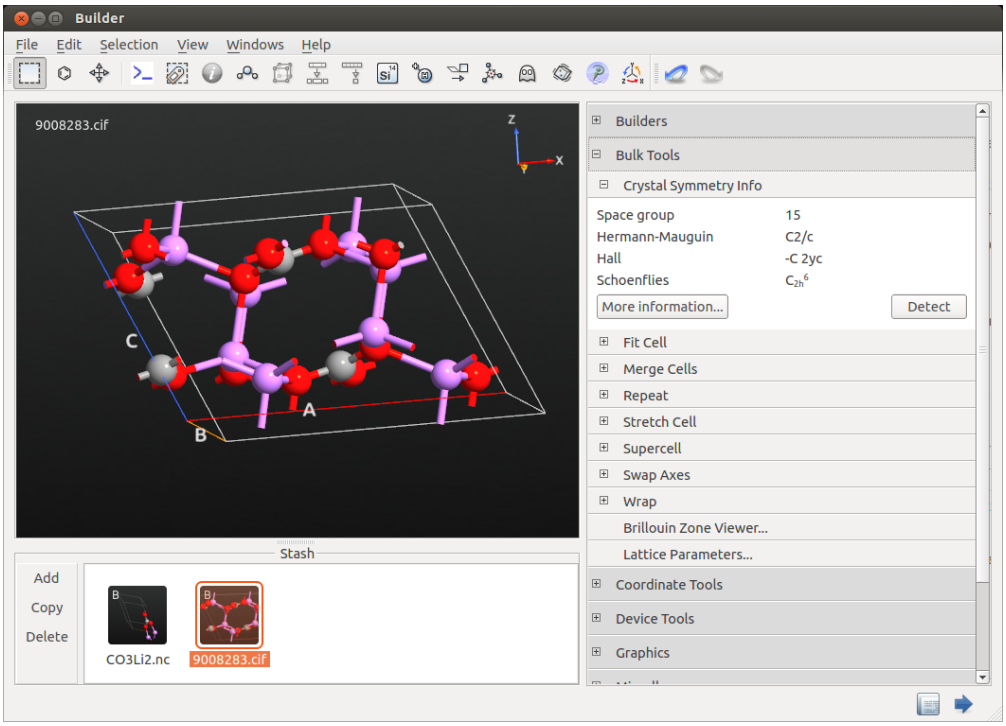
From the figure above, you can see that Li_2O_2 shows a wide band gap and no electronic conduction is expected at low bias voltages. Moreover, as demonstrated in[Page 729, 1], when introducing Li vacancies the electronic conduction decreases even more.

Note: You can follow the transmission_spectrum_perfect_sheets_graphene_mos tutorial for more details and how to set up a transmission spectrum analysis for a perfect periodic bulk system.

Li_2CO_3 bulk and surface structures

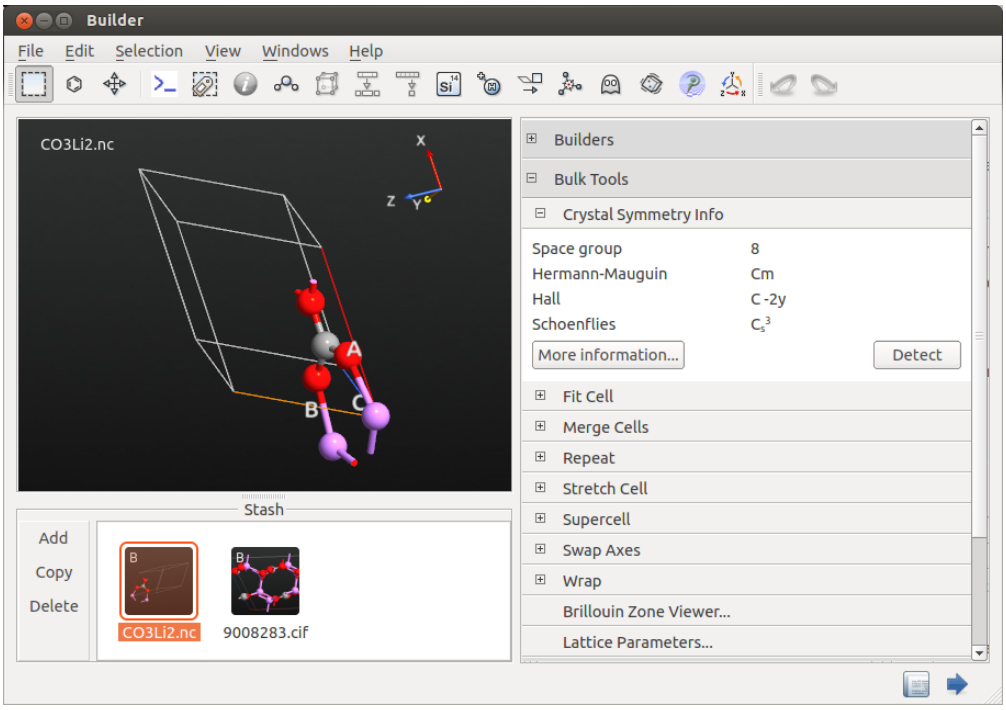
Bulk structure

The Li_2CO_3 bulk structure is not present in the internal database of **QuantumATK**. You need to get the structure file, usually a CIF file, from an external database, e.g. you can use the . Search, download and copy the `9008283.cif` file to your project folder. It will be automatically loaded in the **LabFloor**. For more options see *Import XYZ, CIF, CAR, VASP Files in QuantumATK tutorial*.



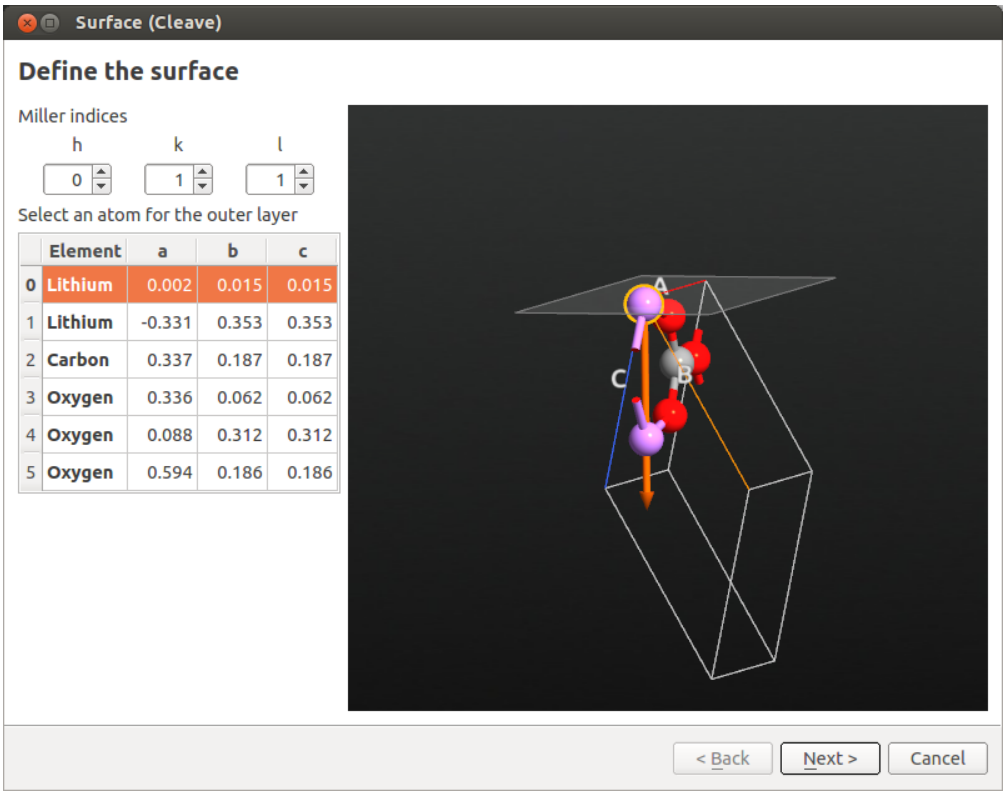
When interfacing the Li_2CO_3 surface with the Li_2O_2 you will need to use a large cell to get a small lattice mismatch. This is because the CO_3 planes in the *C2/c* space group are slightly misaligned from the cell axes. In this tutorial you will use a modified Li_2CO_3 structure with space group *Cm* where the CO_3 planes are perpendicular to the (011) plane. Also, this is the structure used in cite:*Yedilfana2015*.

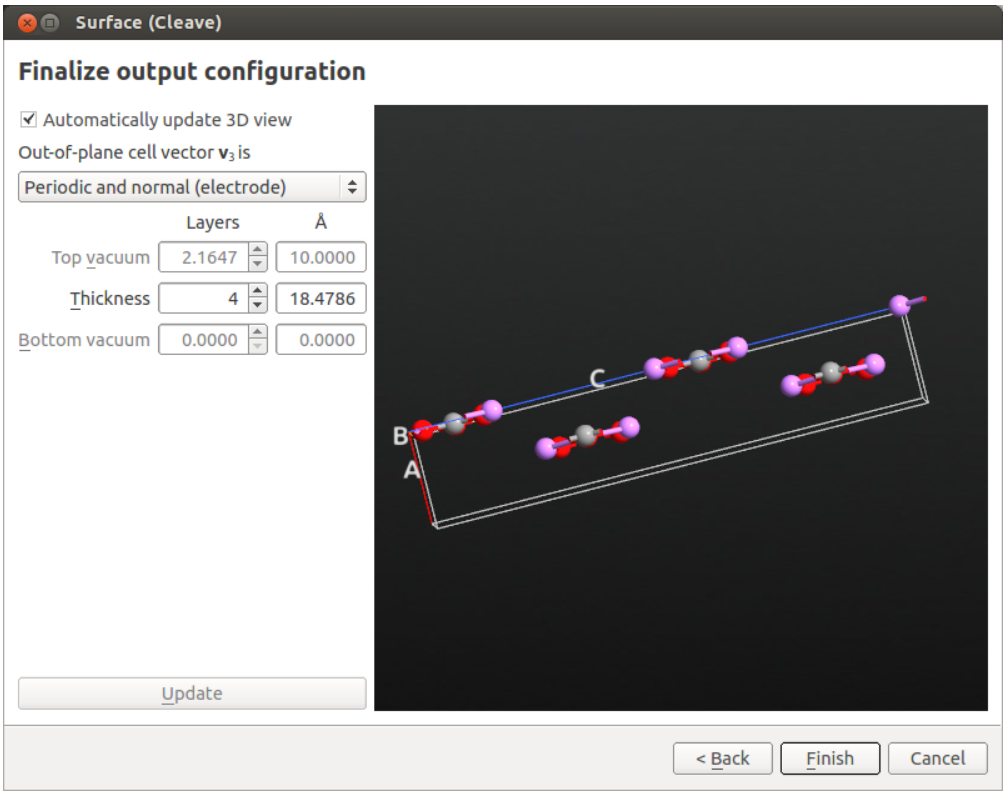
Download the `Li2CO3.py` script and import the structure to the **Builder** to analyze it.



Cleave surface

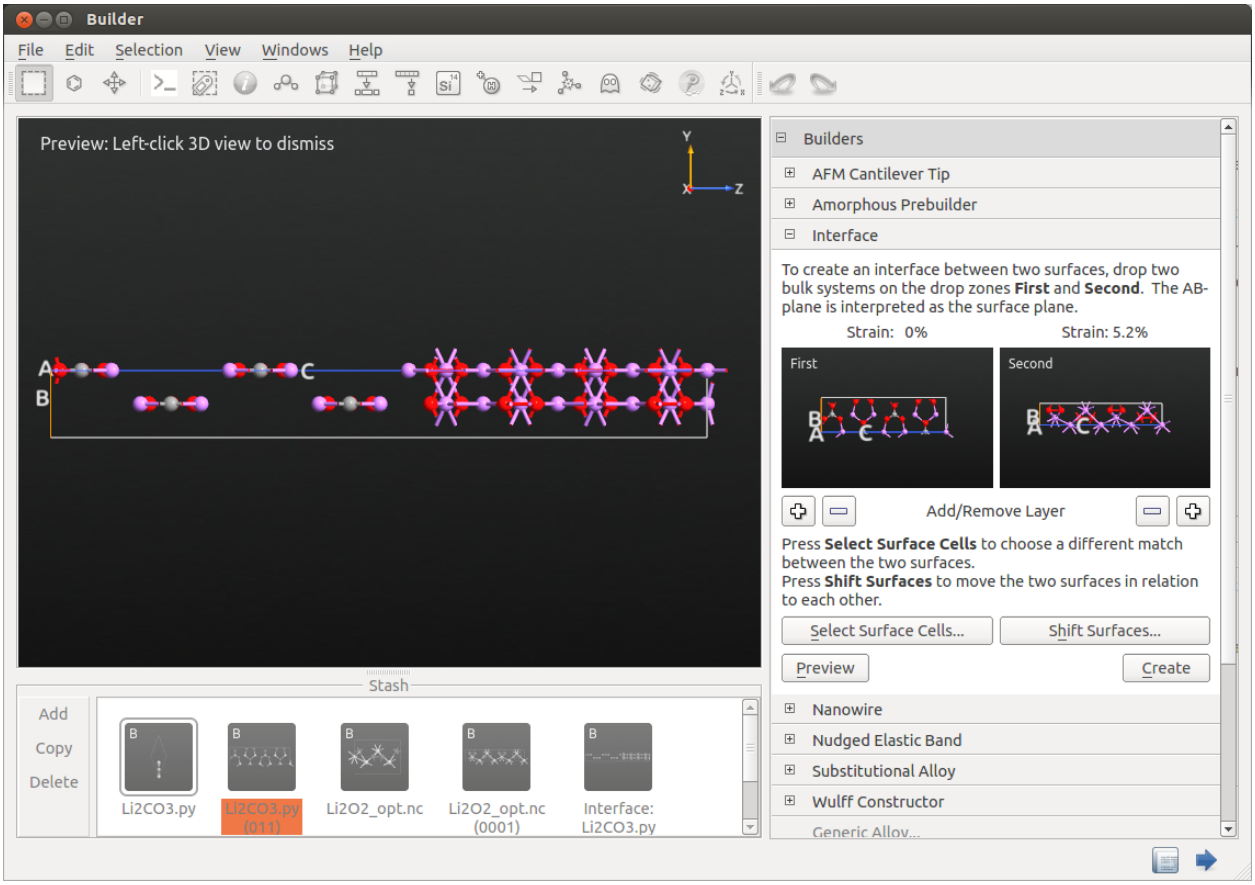
Use the *Builders* → *Surface(Cleave)* tool to cleave the (011) surface as indicated in the figures below.



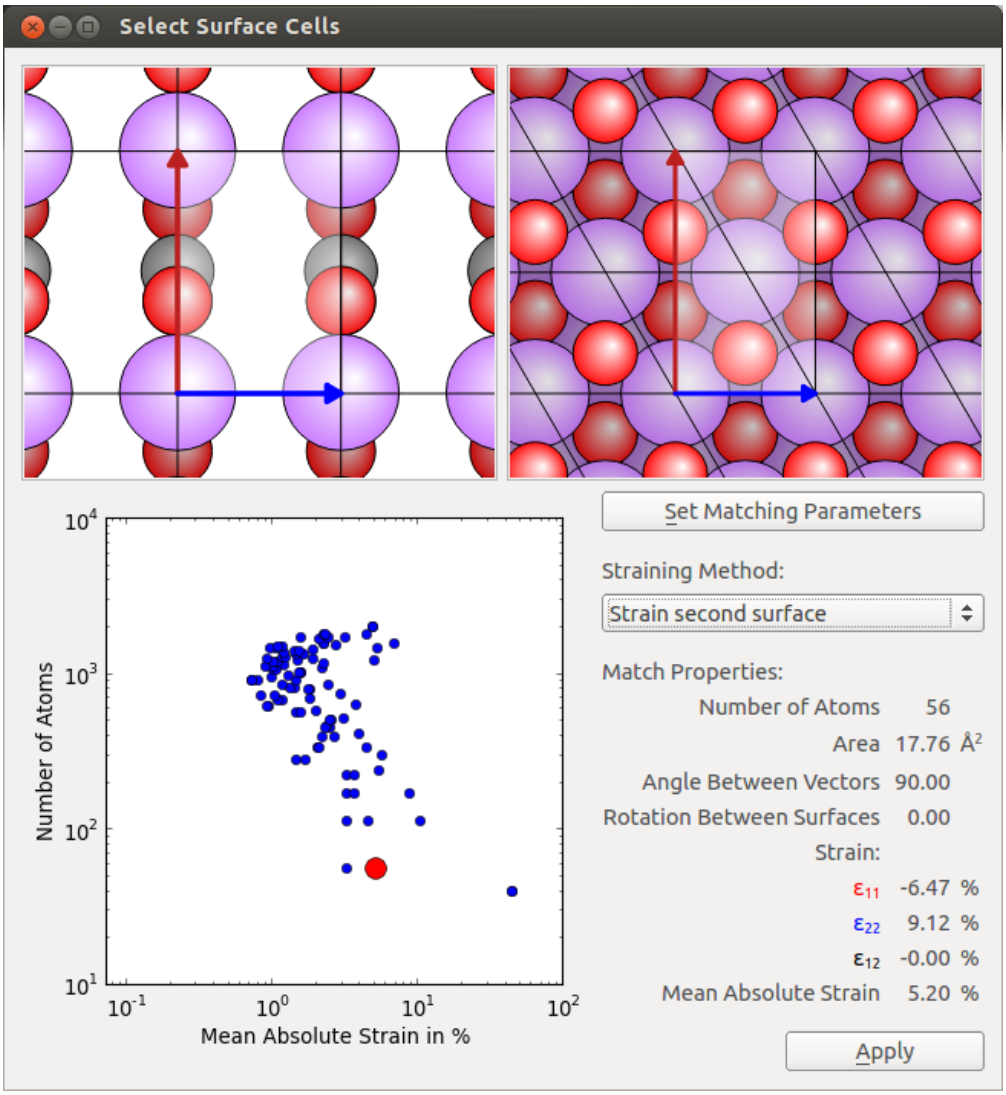


The Li_2O_2/Li_2CO_3 interface

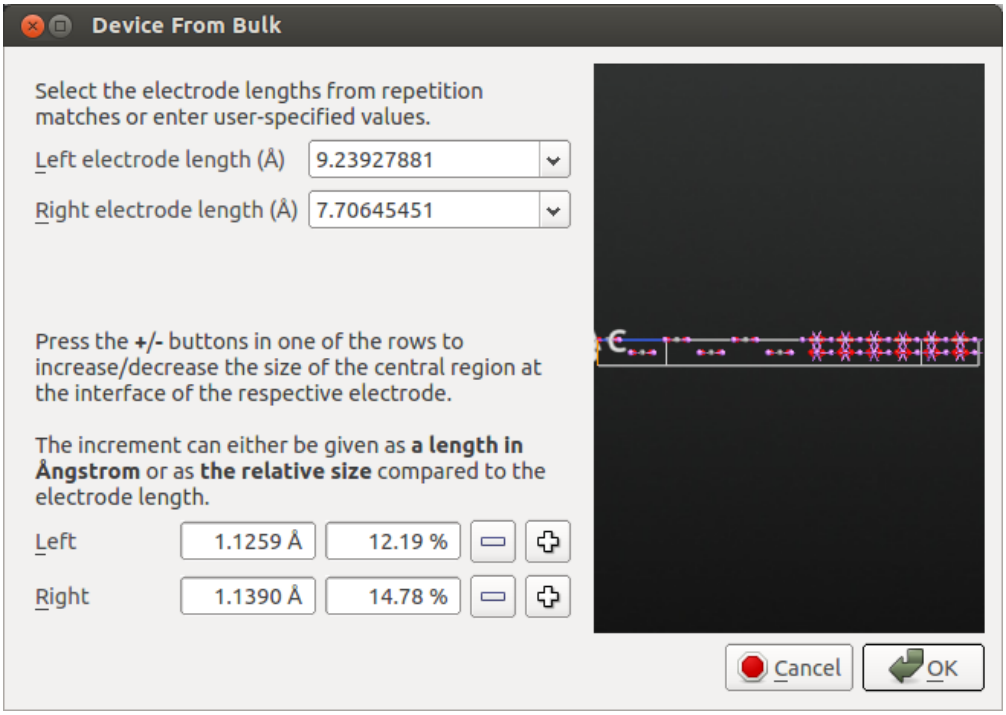
Open the *Builders* → *Interface* tool and drag and drop the Li_2CO_3 surface and the optimized Li_2O_2 surface to the first and second drop zones.



The default results are correct. However, if you want to choose another interface you can click on *Select Surface Cells* button and choose the desired matching parameters.



You can now create the device configuration with the *Device Tools* → *Device From Bulk* plugin. The default electrode lengths are fine.



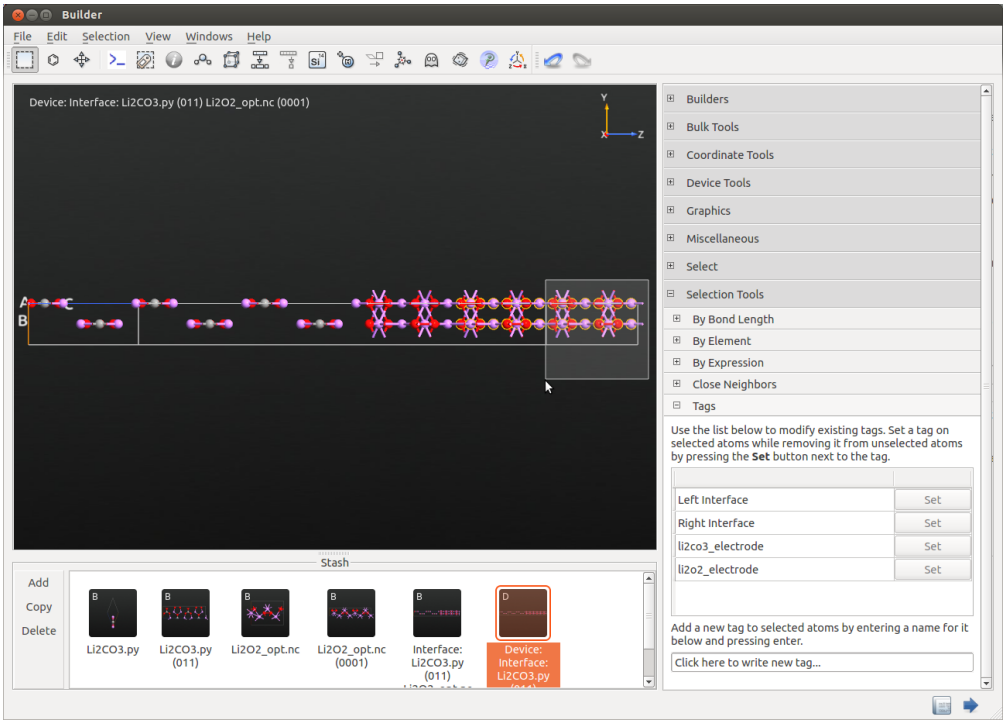
You can download this structure here: `device_configuration_initial.py`. Note that in this final structure the Li_2CO_3 side has been slightly shifted towards the Li_2O_2 side to allow for a proper relaxation

Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

Optimizing the interface

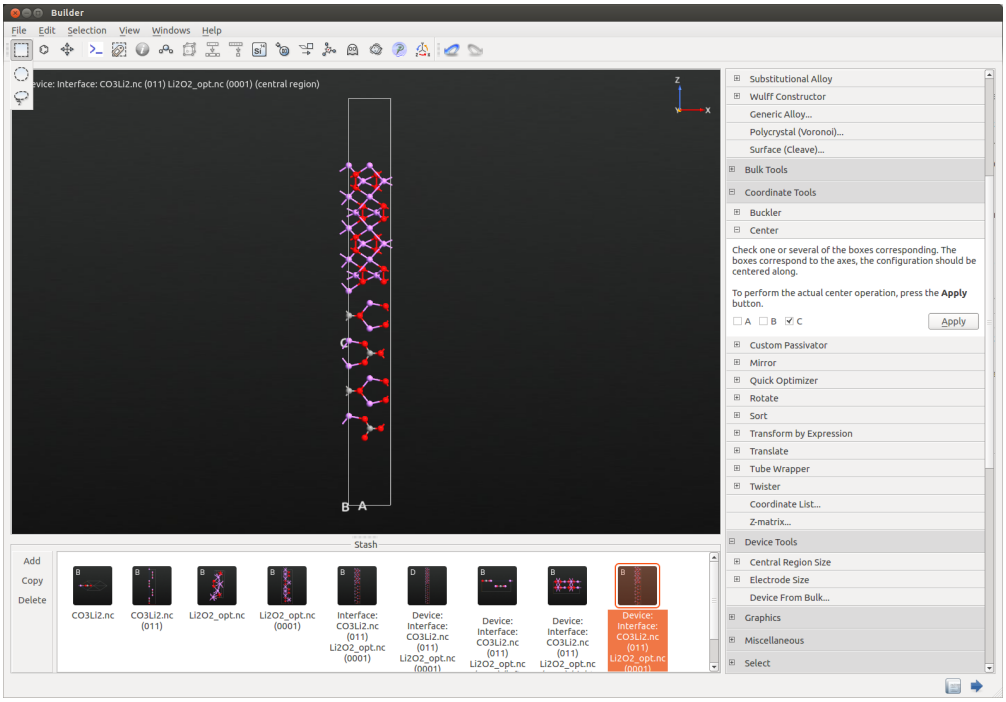
A step-by-step procedure on how to optimize the geometry of a device configuration is described in the *Advanced device relaxation - manual workflow*. The main steps are the following:

- Tag the electrode extension regions by using the *Selection Tools* → *Tags* plugin. You can do that by selecting the atoms in the electrodes, the electrode extension region will be automatically selected.



Hint: Tag also the first layer outside the electrode extension regions which is already part of the central region.By doing so, the *Device From Bulk* plugin will be able to recognize the proper periodicity and recreate the device structure (see below).

•Extract the central region as *BulkConfiguration* by using the split device plugin, increase the C vector and center the configuration.



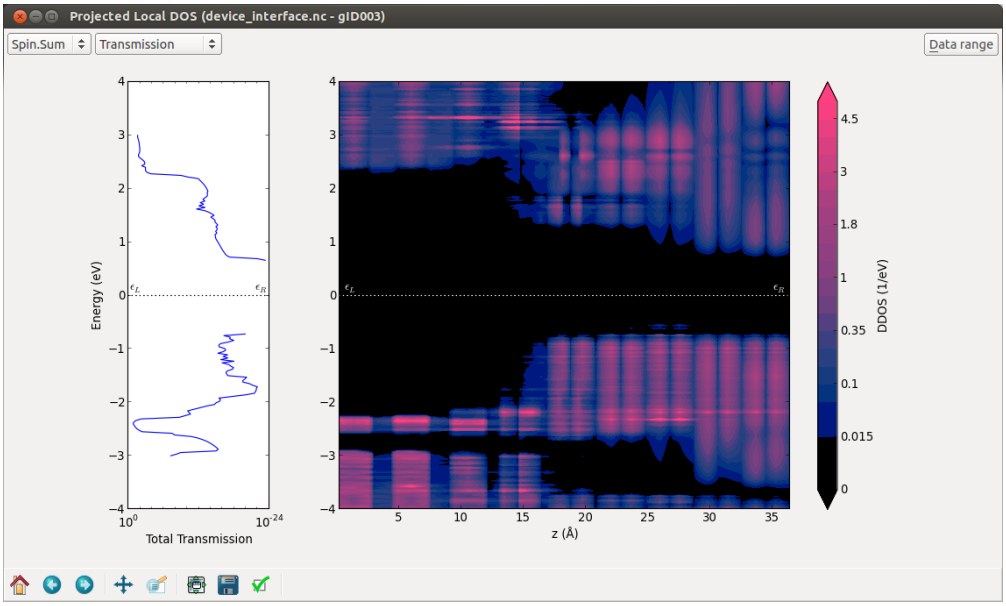
•Set up a geometry optimization calculation with rigid body constrain for the Li_2O_2 tagged region and fixed constrain for the Li_2CO_3 one. You can also download the corresponding `interface_bulk_opt.py`.

•Finally, when the optimization of the interface in a bulk configuration is done, import in the **Builder** the optimized structure and recreate the device as described in the previous section.

Electronic structure

Send the device configuration from the **Stash** to the **Scripter** and set up a new device calculator with the GGA-RPBE exchange-correlation potential and a 7x15x200 k-point sampling. You can leave all other parameters as defaults. Here, you can download the whole script: `device_interface.py`.

You can add various analysis to investigate in more detail the electronic structure and the transport properties. Below, for example, the projected device density of state analysis is shown in the picture. Here you can see that Li_2CO_3, on the left side, has a much higher band gap than Li_2O_2. Electronic conduction is thus expected to be very low, as it is indeed shown in[Page 729, 1].



Hint: You can further investigate the Li diffusion processes in Li_2O_2 and Li_2CO_3 crystals by following the *Li-ion diffusion in LiFePO4 for battery applications*.

References

3.3.2 Li-ion diffusion in $LiFePO_4$ for battery applications

Version: 2016.3

$LiFePO_4$ is among the typical rechargeable Li-ion battery cathode materials with high energy density and low cost, while also being environmentally friendly[1,2]. In this tutorial you will use ATK-DFT to estimate the Li-ion diffusion rates along different crystallographic directions in $LiFePO_4$.

[1] M.ăSaiful Islam and Craig A.ăJ. Fisher. Lithium and sodium battery cathode materials: computational insights into voltage, diffusion and nanostructural properties. *Chem. Soc. Rev.*, 43(1):185, 2014. doi:10.1039/C3CS60199D.

[2] William DavidsonăRichards YanăWang. Design principles for solid-state lithium superionic conductors. *Nature Materials*, 14(10):1026, 2015. doi:10.1038/nmat4369.

In particular, you will:

•Import the $LiFePO_4$ bulk structure.

•Optimize the $LiFePO_4$ lattice parameters.

•Create the $Li_{1-x}FePO_4$ structures.

•Optimize the initial and final configurations.

•Create initial NEB trajectories (IDPP interpolation).

•Optimize the Li diffusion path.

•Calculate the reaction rates using harmonic transition state theory (HTST).

Import $LiFePO_4$ bulk structure

The $LiFePO_4$ structure can be imported from the **Crystallography Open Database** plugin in QuantumATK.

1.Click the **Builder**.

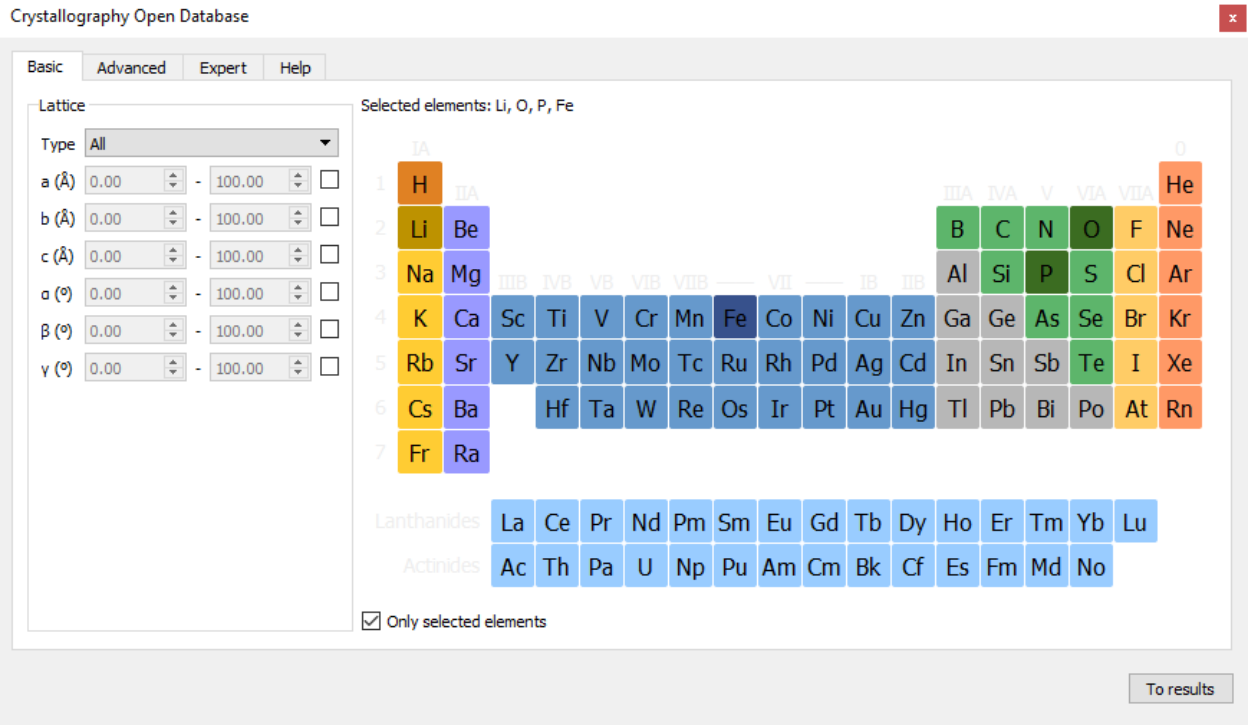
2.Open the *From Plugin* → *Crystallography Open Database*.

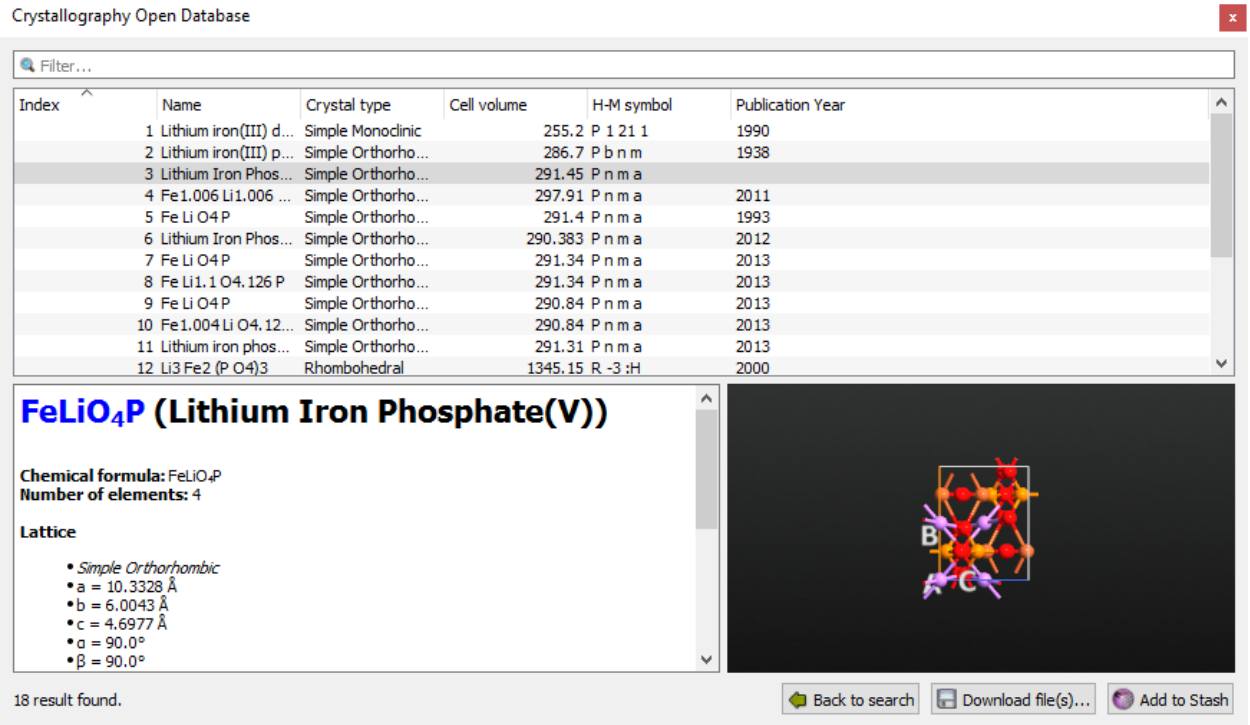
3.Select *Li, Fe, P, O* as elements and click the **To results** button.

4.Choose *Lithium Iron Phosphate(V), Simple Orthorhombic(Pnma)*.

5.Click **Add to stash** button.

It will be automatically loaded in the **Builder** stash.

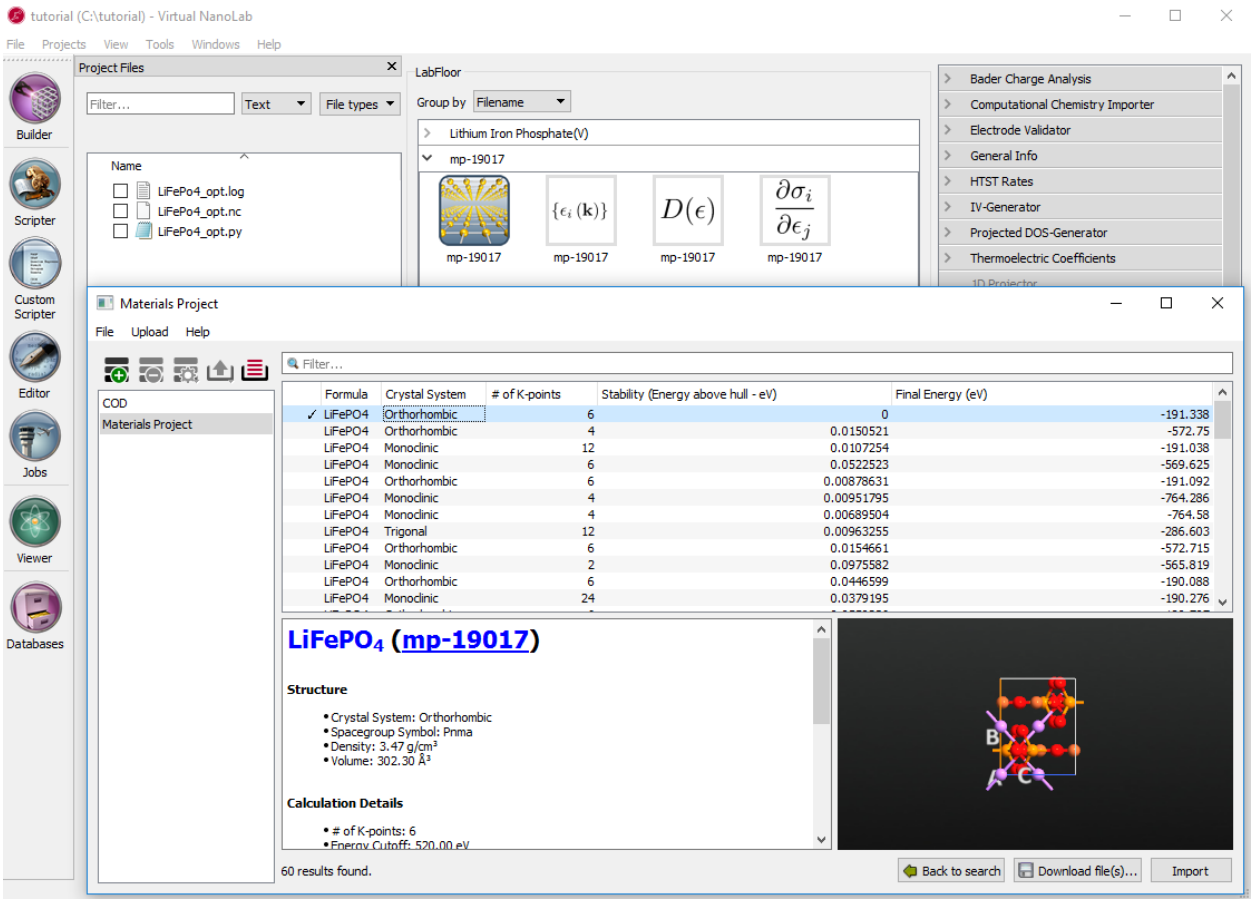




New functionality in QuantumATK 2017:

From the **QuantumATK 2017** version, crystal structures such as $LiFePO_4$ can be imported by the QuantumATK **Databases** tool as follows: Click the **DataBases**. See **COD** (Crystallography Open Database) and **Materials Project**. Both are possible to get the $LiFePO_4$ structure. In the COD case, select *Li, Fe, P, O* as elements and click the **Search** button.

You can also use the **Materials Project** database. Choose the Materials Project. Select elements or type LiFePO4. Select one and click the **import** button, it will be automatically loaded in the **LabFloor** in the same way as the COD. In this case, you can get more calculation information including the structure from the Material Project database.



For more options, see also *Import XYZ, CIF, CAR, VASP files in QuantumATK tutorial*.

Optimize $LiFePO_4$ lattice parameters

You can import the $LiFePO_4$ structure to the **Builder** to have a deeper look into the structure.

Note: In order to better compare with the results of reference[3] you can use the *Bulk Tools* → *Swap Axes* tool to swap **C<->A** and **Z<->X**. However, by doing so, the configuration will lose the information about the symmetry type, in this case *Simple orthorhombic*. Use the *Bulk Tools* → *Lattice Parameters* tool to set the *Lattice type* to *Simple orthorhombic*.

In order to optimize the lattice parameters, send the $LiFePO_4$ structure from the **Stash** to the **Scripter** to set up a new DFT calculation as follows:

1. Add the **New Calculator** block and modify the following parameters:

- SGGA-PBE exchange correlation potential
- 7x5x3 k-point sampling

(We will use the the default FHI pseudopotentials and DZP basis set).

2. Add an **OptimizeGeometry** block and modify:

[3] DhamodaranăSanthanagopalan YuriăJanssen. Reciprocal Salt Flux Growth of LiFePO 4 Single Crystals with Controlled Defect Concentrations. *Chemistry of Materials*, 25(22):4574, 2013. doi:10.1021/cm4027682.

- Max forces 0.02 eV/Å
- Max stress 0.1 GPa
- Remove check from **Constrain cell** to allow for cell optimization
- Check **Save trajectory** and specify the file `LiFePo4_opt.nc`

3. Change default output file name to `LiFePo4_opt.nc`
4. Finally, send the script to the **JobManager** and run the calculation.

It takes about 30 minutes on 8 cores. After finishing the job, open the optimized *BulkConfiguration* with object ID `glD002` with the **Builder**. Check the new optimized lattice parameters which agree to within 0.1 percent with the experimental data.

Create the $Li_{1-x}FePO_4$ structures

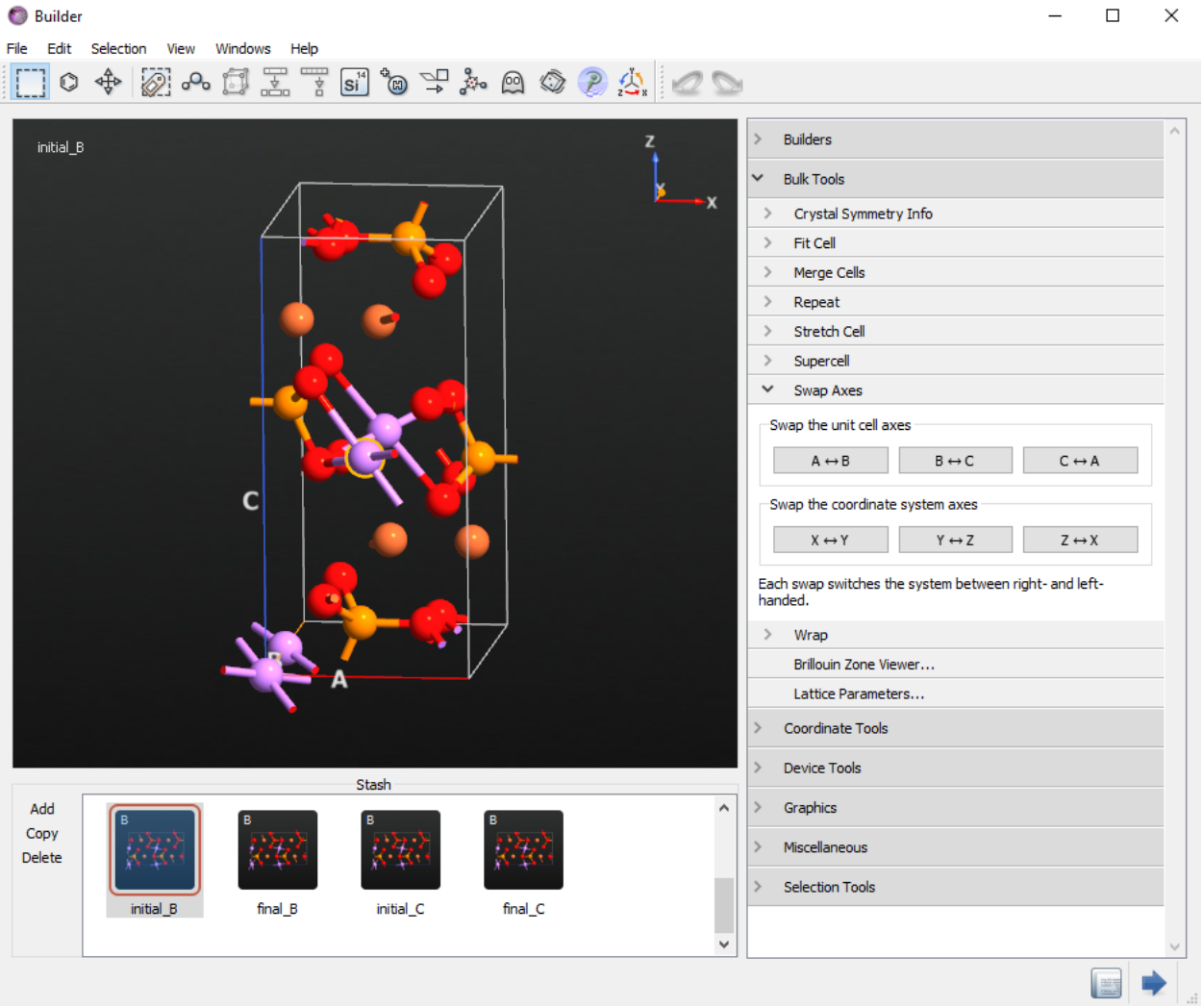
In this tutorial you will study the diffusion of one Li atom for a Li-deficient structure such as $Li_{1-x}FePO_4$. In particular, you will remove one of the four Li atoms from the bulk configuration. Note that the diffusion energy barriers depend on the Li concentration[4].

In the next section you will set up the calculation for the diffusion of a Li atom along two different directions. You thus need to create initial configurations and final configurations for the same Li atom diffusing along the B and C directions.

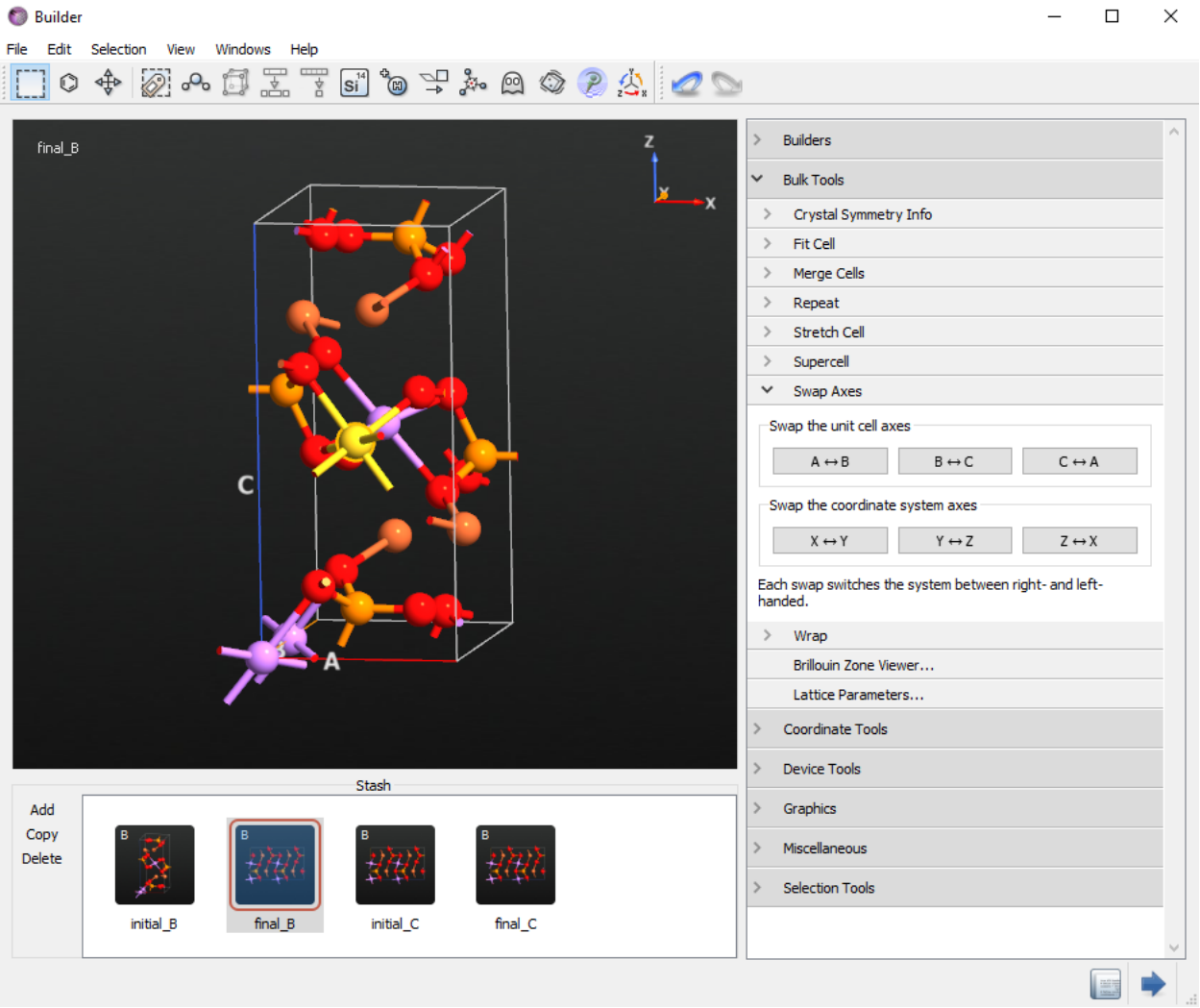
Create initial and final structures for diffusion along B direction

1. Right-click on the configuration on the **Stash** and make three copies such that you have 4 identical configurations. We will make initial and final structures for the B and C directions.
2. From the first configuration, delete one Li atom as in the figure below. This will be your initial structure for the B direction NEB calculations and you can rename it *initial_B*.

[4] Chuying Ouyang, Siqi Shi, Zhaoxiang Wang, Xuejie Huang, and Liquan Chen. First-principles study of Li ion diffusion in LiFePO4. *Physical Review B*, 2004. doi:10.1103/PhysRevB.69.104303.

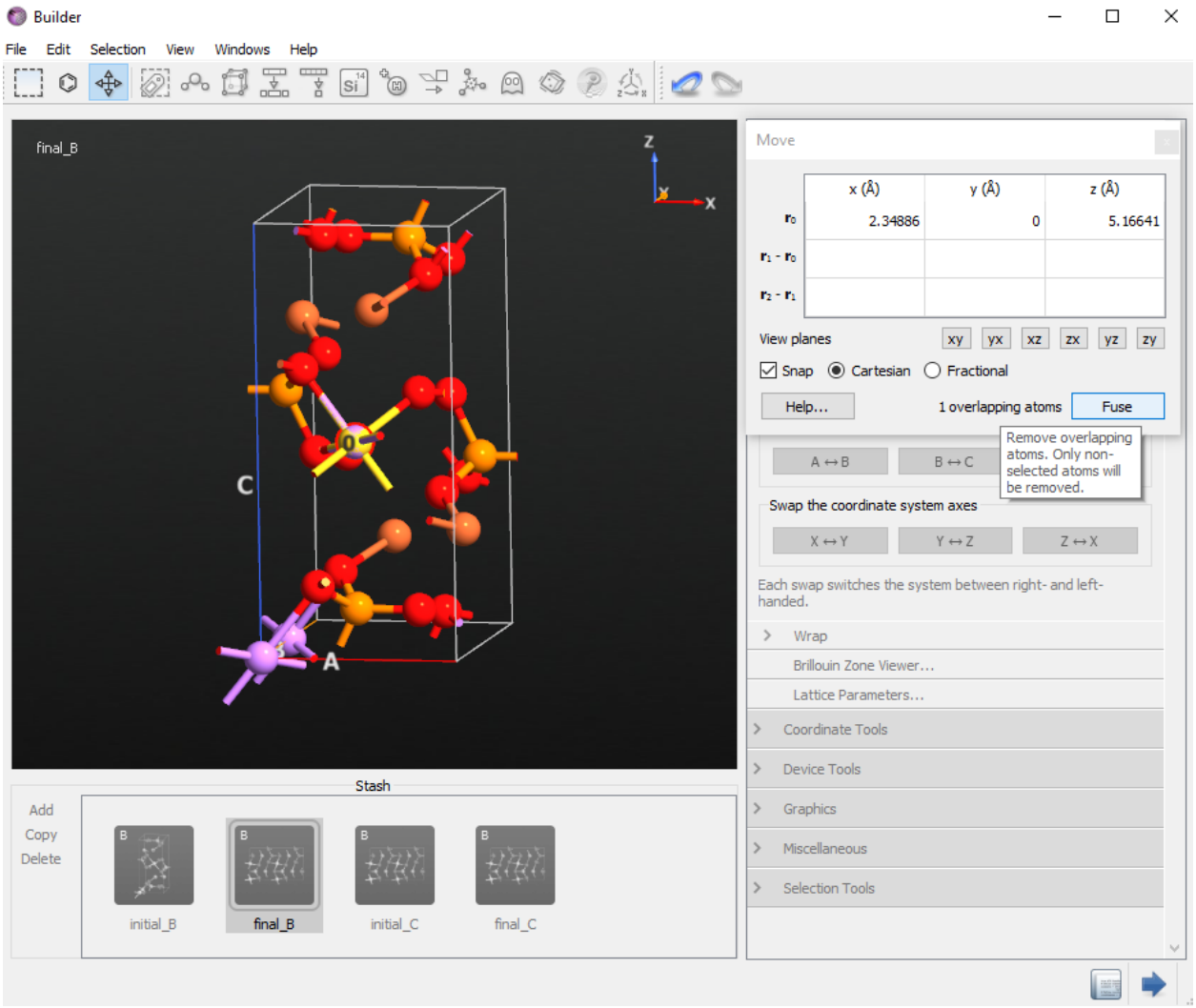


3. Double click the second configuration to make it active. Next rename it *final_B*. Select the same Li atom you just deleted from the initial configuration and open the **Periodic Table** tool. Click on **Gold** (Au) element to change the Li atom to Gold.



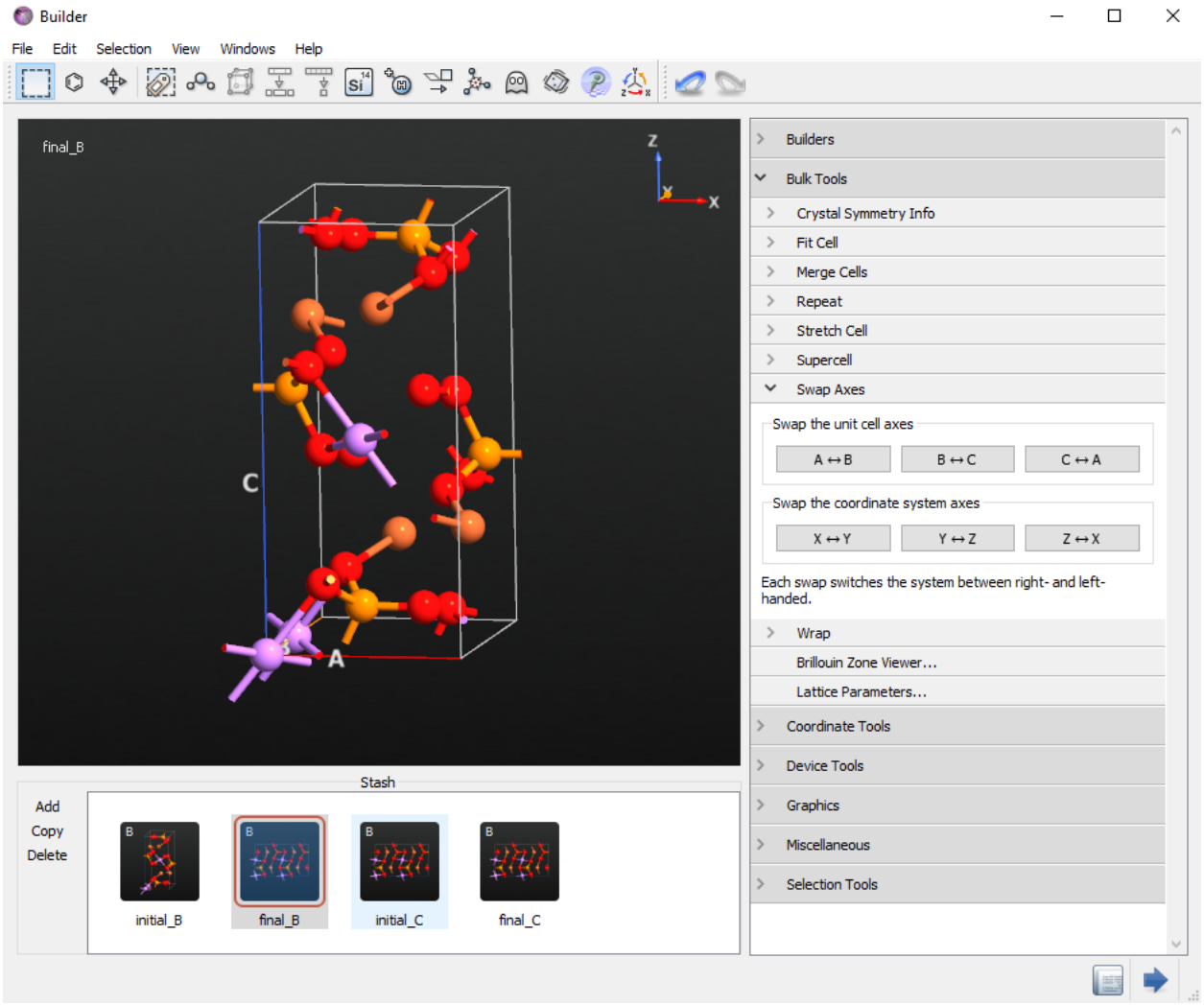
Tip: You are applying this trick to be able to move the Li atom into a specific position. In this case, the position of the Li vacancy defined by the Gold atom. There are indeed several other workarounds to build this final configuration, e.g. by copying the coordinates of the original Li atom you removed. You are welcome to use here the procedure that you find more relevant and intuitive for you.

4. Select Li atom number 7 (the one next to the Gold atom along the B direction) and use the **Move** tool to move it on top of the Gold atom. Be sure that the *Snap* option is enabled.



5. Use the *Fuse* option to remove the Gold atom and your final structure is now ready.

6. Finally, check the Li atom number 6 in the *initial_B* and *final_B* configurations. Li atom number 6 should be moved in the B direction.



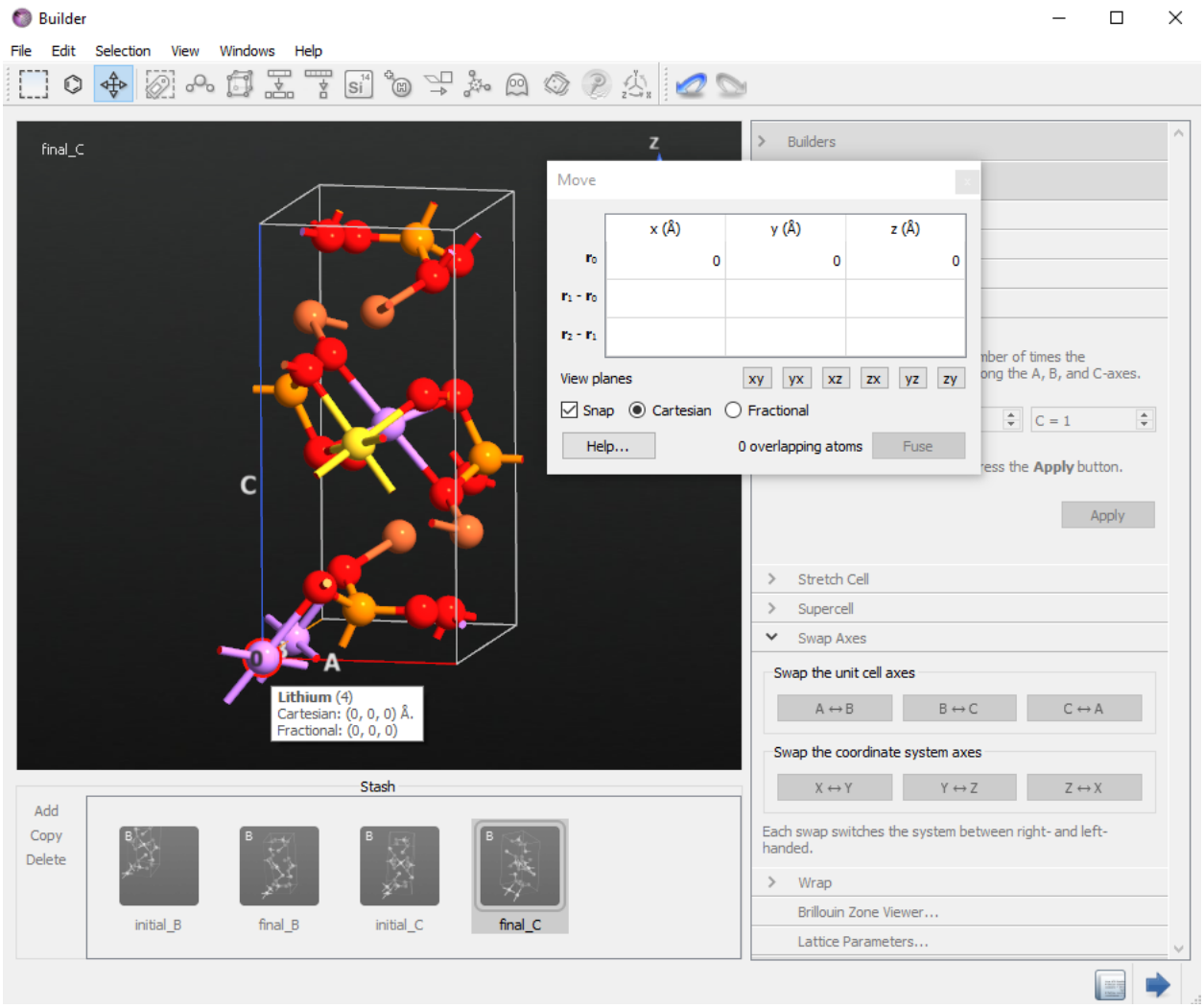
Create initial and final structures for diffusion along C direction

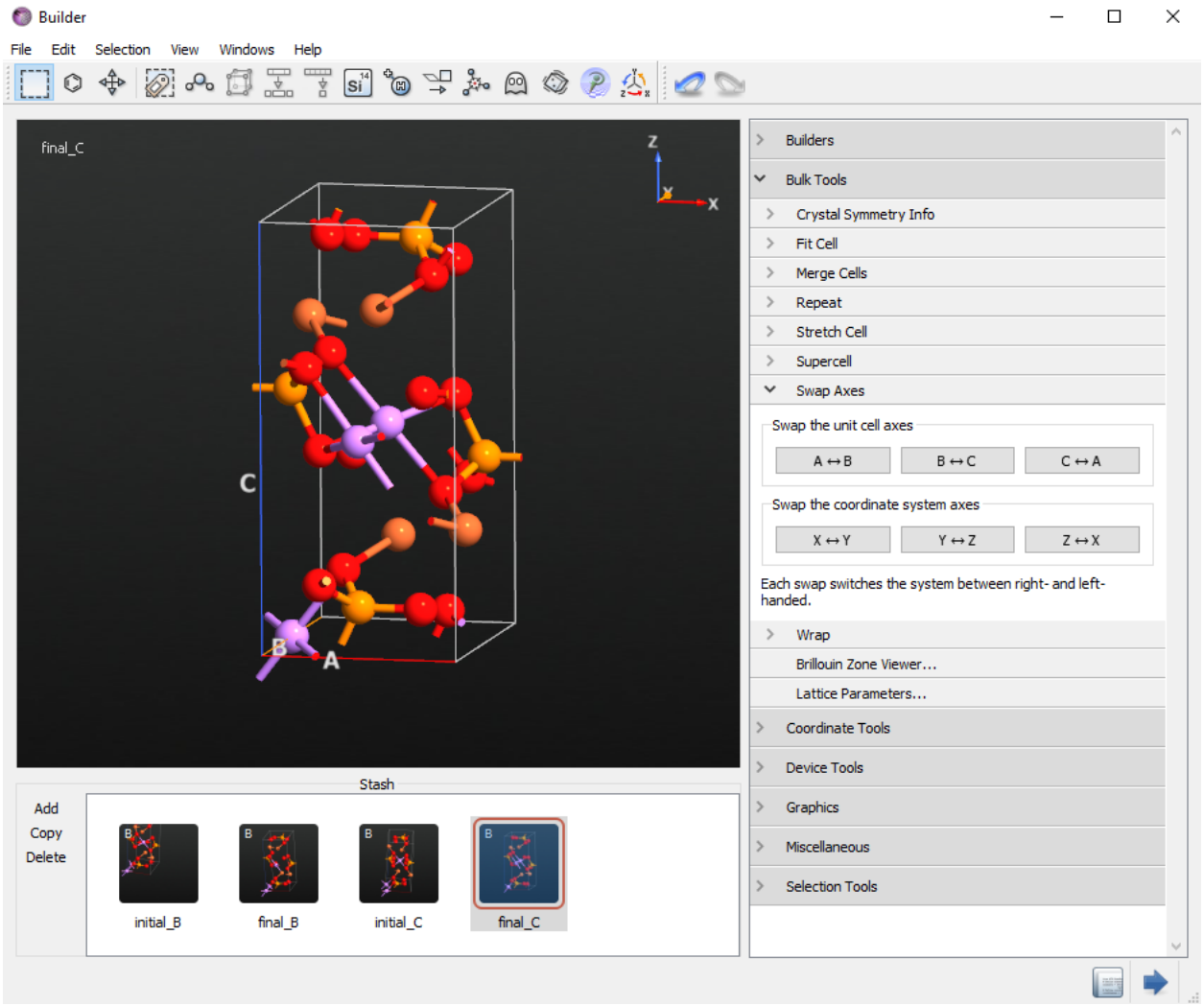
Follow the same procedure as above to create the configuration along the C direction. The initial structure for the C direction is the same as the *intial_B* structure. Rename it *initial_C*. In this case, the final structure moves Li atom number 4 to the removed Li position (substituted Gold) in the *inital_C* structure. Rename the final structure, *final_C*.

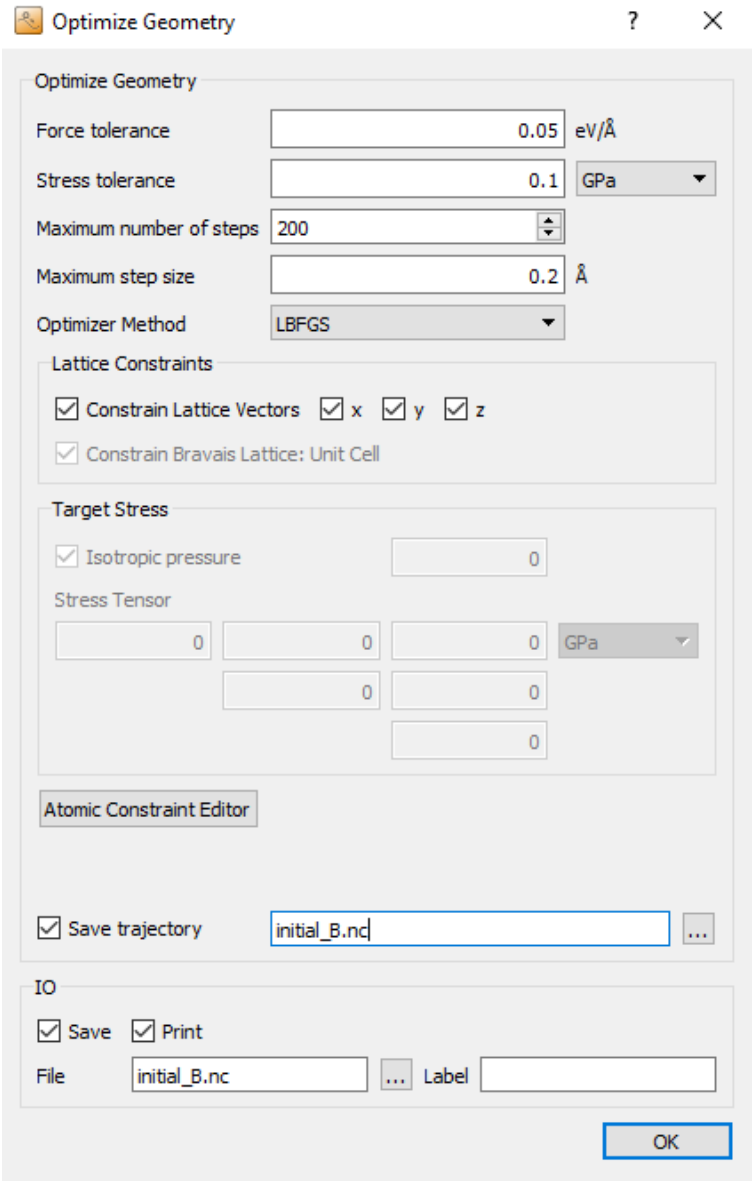
Optimize initial and final configurations

Send each structure, *initial_B*, *final_B*, *initial_C* and *final_C*, to the **Scripter** and set up a geometry optimization calculation with the same parameters as for the $LiFePO_4$ structure. However, in this case you will not optimize the lattice parameters. Check the *Constrain cell* option as indicated below.

Moreover, in this case you can use the unpolarized GGA.PBE functional. This will allow you to run your calculation much faster. Here you can download the four input scripts: `initial_B.py`, `final_B.py`, `initial_C.py` and `final_C.py`.


Builder
File
Edit
Selection
View
Windows
Help
final_C
Z
C
A
Lithium (4)
Cartesian: (0, 0, 0) Å.
Fractional: (0, 0, 0)
Move
x (Å)
y (Å)
z (Å)
r0
r1 - r0
r2 - r1
View planes
xy
yx
xz
zx
yz
zy
Snap
Cartesian
Fractional
Help...
0 overlapping atoms
Fuse
Builders
Apply
Stretch Cell
Supercell
Swap Axes
Swap the unit cell axes
A ↔ B
B ↔ C
C ↔ A
Swap the coordinate system axes
X ↔ Y
Y ↔ Z
Z ↔ X
Each swap switches the system between right- and left-handed.
Wrap
Brillouin Zone Viewer...
Lattice Parameters...
Stash
Add
Copy
Delete
initial_B
final_B
initial_C
final_C


Builder
File
Edit
Selection
View
Windows
Help
final_C
Z
X
C
B
A
Builders
Bulk Tools
Crystal Symmetry Info
Fit Cell
Merge Cells
Repeat
Stretch Cell
Supercell
Swap Axes
Swap the unit cell axes
A ↔ B
B ↔ C
C ↔ A
Swap the coordinate system axes
X ↔ Y
Y ↔ Z
Z ↔ X
Each swap switches the system between right- and left-handed.
Wrap
Brillouin Zone Viewer...
Lattice Parameters...
Coordinate Tools
Device Tools
Graphics
Miscellaneous
Selection Tools
Stash
Add
Copy
Delete
initial_B
final_B
initial_C
final_C

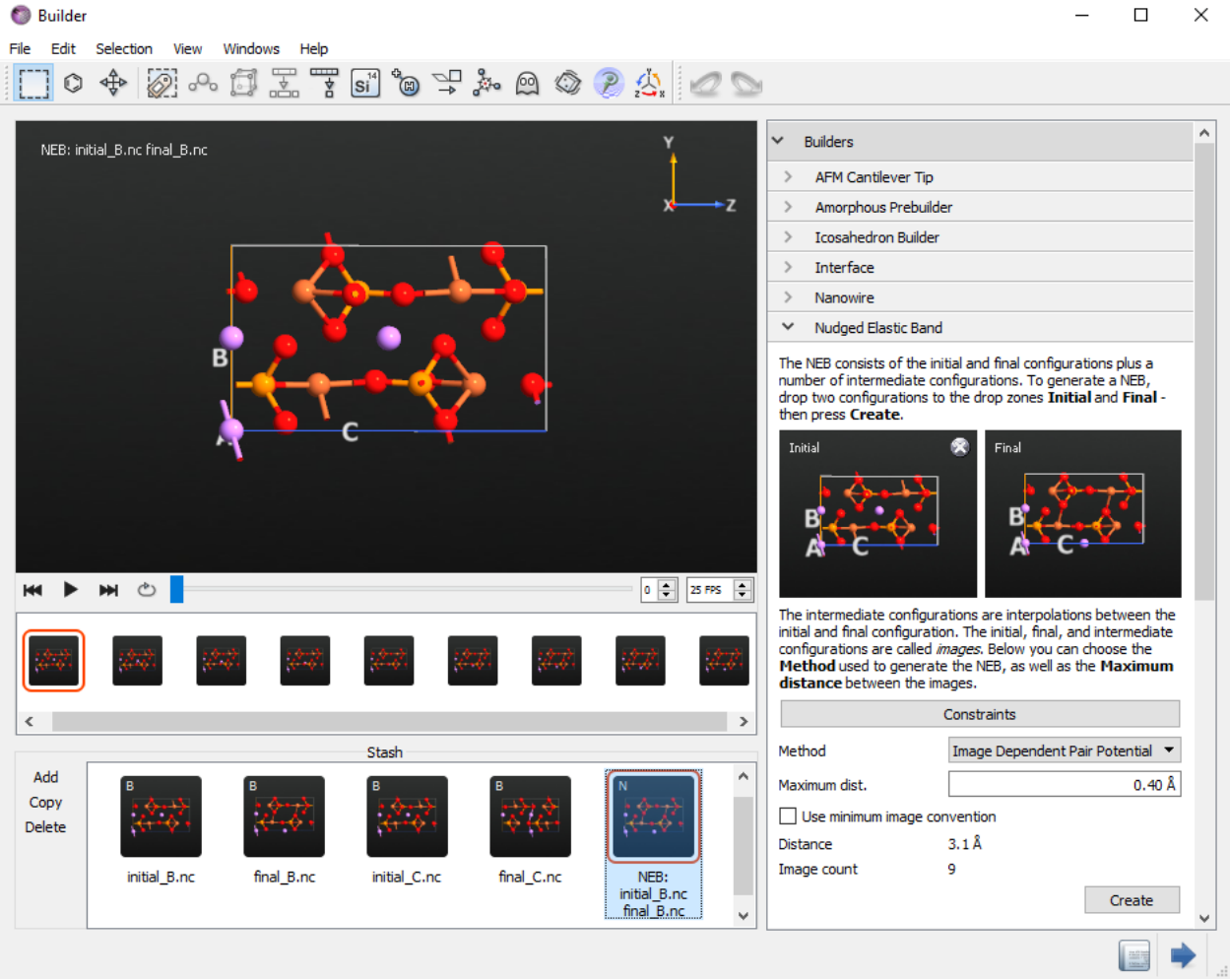

Optimize Geometry
?
Optimize Geometry
Force tolerance
0.05
eV/Å
Stress tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer Method
LBFGS
Lattice Constraints
Constrain Lattice Vectors
x
y
z
Constrain Bravais Lattice: Unit Cell
Target Stress
Isotropic pressure
0
Stress Tensor
0
0
0
GPa
0
0
0
Atomic Constraint Editor
Save trajectory
initial_B.nc
...
IO
Save
Print
File
initial_B.nc
...
Label
OK

Create initial NEB trajectories

The tutorial *Pt diffusion on Pt surfaces using NEB calculations* will teach you how to set up the NEB object. Here we will highlight only the main steps.

1. When the optimization of the end points are done you will find the optimized configurations loaded in the **LabFloor** with object ID equal to *glD002*. Drag and drop these *BulkConfiguration* objects directly into the **Builder**.

2. In the **Builder** open the *Builders* → *Nudged Elastic Band* plugin and drag and drop from the **Stash** the initial and final configurations corresponding to the diffusion of a Li atom along the B direction, `initial_B.nc` and `final_B.nc` configurations.



3. Select the *Image Dependent Pair Potential* interpolation method and use 9 images including the end points.

4. Create the initial Li diffusion path along C direction with the `initial_C.nc` and `final_C.nc` configurations. Use 9 images with 0.8 Å maximum distance including the end points .

Optimize Li diffusion path

You are now ready to set up and optimize the Li diffusion path in $LiFePO_4$.

1. Send the two NEB objects from the **Stash** to the **Scripter**.
2. Add and set up a **New Calculator** with the same settings as for the calculators of the end points.
3. Add a **OptimizeGeometry** block and be sure to select the *Climbing Image Method*.
4. Finally, run the two NEB calculations `neb_B.py` and `neb_C.py`.

Note that as described in the *Pt diffusion on Pt surfaces using NEB calculations*, you can run this calculation in parallel over many MPI processes. In particular, a NEB run is parallelized over the internal images. **ATK** will try to detect the best parallelization strategy and it will write a small report at the beginning of your log file.

In this example, you will immediately see that a load balance problem has been detected. However, 2 out 16 cores will be idle, so it is a rather small loss of efficiency, not greatly reduced as it says in the log file below. It takes about 2 hours in 16 processes. You can also change the number of MPI processes you are using to achieve the best performance.

```
+-------------------------------------------------------------------------------+
| NEB image level parallelism enabled.                                          |
+-------------------------------------------------------------------------------+
| Total number of interior images: 7                                            |
| Total number of processes: 16                                                 |
| Processes per image: 2                                                        |
| Images calculated in parallel: 8                                              |
|                                                                               |
| Process group # 0 will calculate images: 1                                    |
| Process group # 1 will calculate images: 2                                    |
| Process group # 2 will calculate images: 3                                    |
| Process group # 3 will calculate images: 4                                    |
| Process group # 4 will calculate images: 5                                    |
| Process group # 5 will calculate images: 6                                    |
| Process group # 6 will calculate images: 7                                    |
| Process group # 7 will be idle!                                               |
| WARNING: Load balance problem detected.                                       |
|          The number of interior images calculated in parallel is not a        |
|          divisor of the number of interior images. This will greatly          |
|          reduce parallel performance. Either modify the parameter             |
|          "processes_per_image" or the number of MPI processes.                |
|                                                                               |
|          With 2 processes_per_image ideal performance will be obtained       |
|          by using 14 MPI processes.
```

Tip: You can speed up the optimization by first running a simple NEB calculation with a high force tolerance, e.g. 0.08 eV/Å, and without using the *Climbing Image Method*. This will in general be a faster calculation giving you a rough estimate of the energy barrier and the diffusion path. After this step, you can set up a more precise NEB run and also use the climbing method to finalize your results.

Analyze the results

Once the calculation is properly converged you will find the optimized *NudgedElasticBand* object loaded in the **LabFloor**

Tip: Always remember to check the log file at the end of your simulation. In particular, check that convergence is achieved:

```
+-------------------------------------------------------------------------------+
| NEB Optimization using the LBFGS optimizer                                    |
+-------------------------------------------------------------------------------+
| Iteration   Step Length    Max. Force  Max. Energy Image    Max. Energy       |
+-------------------------------------------------------------------------------+
|         0     0.000e+00     1.473e+00                 4       0.925917       |
|         1     4.825e-02     1.362e+00                 4       0.794101       |
...
|        23     1.220e-02     2.772e-01                 4       0.405770       |
|        24     3.731e-03     4.938e-02                 4       0.404796       |
+-------------------------------------------------------------------------------+
| NEB optimization converged after 24 steps.                                    |
+-------------------------------------------------------------------------------+
```

In the two figures below, the optimized Li diffusion process for the Li atom moving along the B and C directions are illustrated by using the **Movie Tool** plugin.

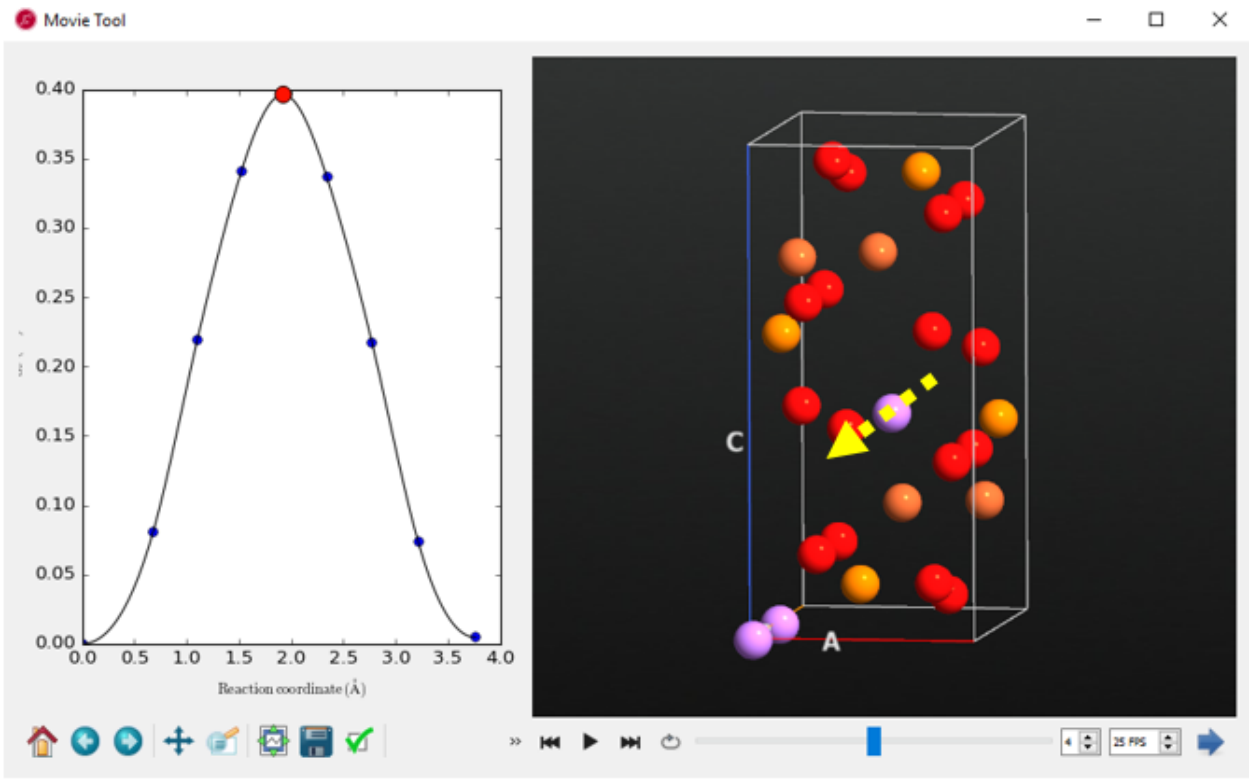
While the Li atom can more easily move along the B direction with 0.39 eV energy barrier, you found a high energy barrier, 2.3 eV for the Li atom moving along the C direction. The results are in agreement with reference[Page 743, 4].

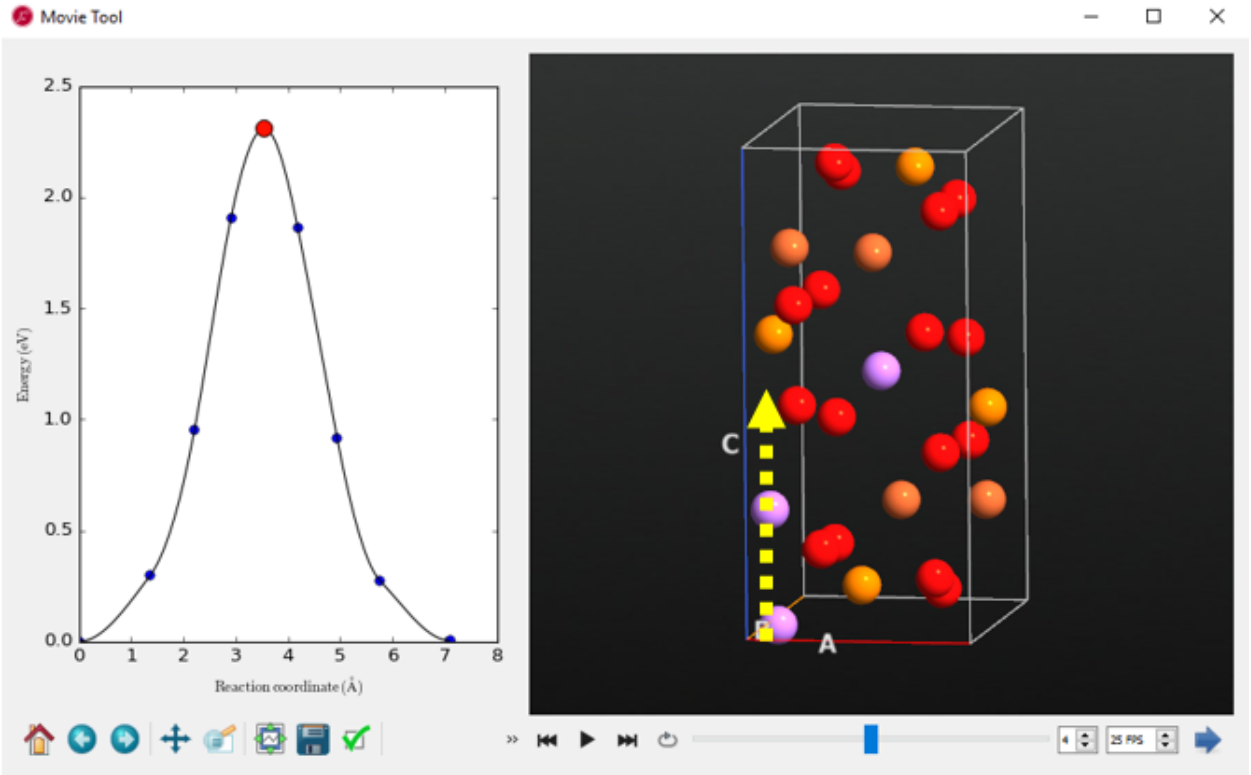
Calculate the reaction rates using harmonic transition state theory

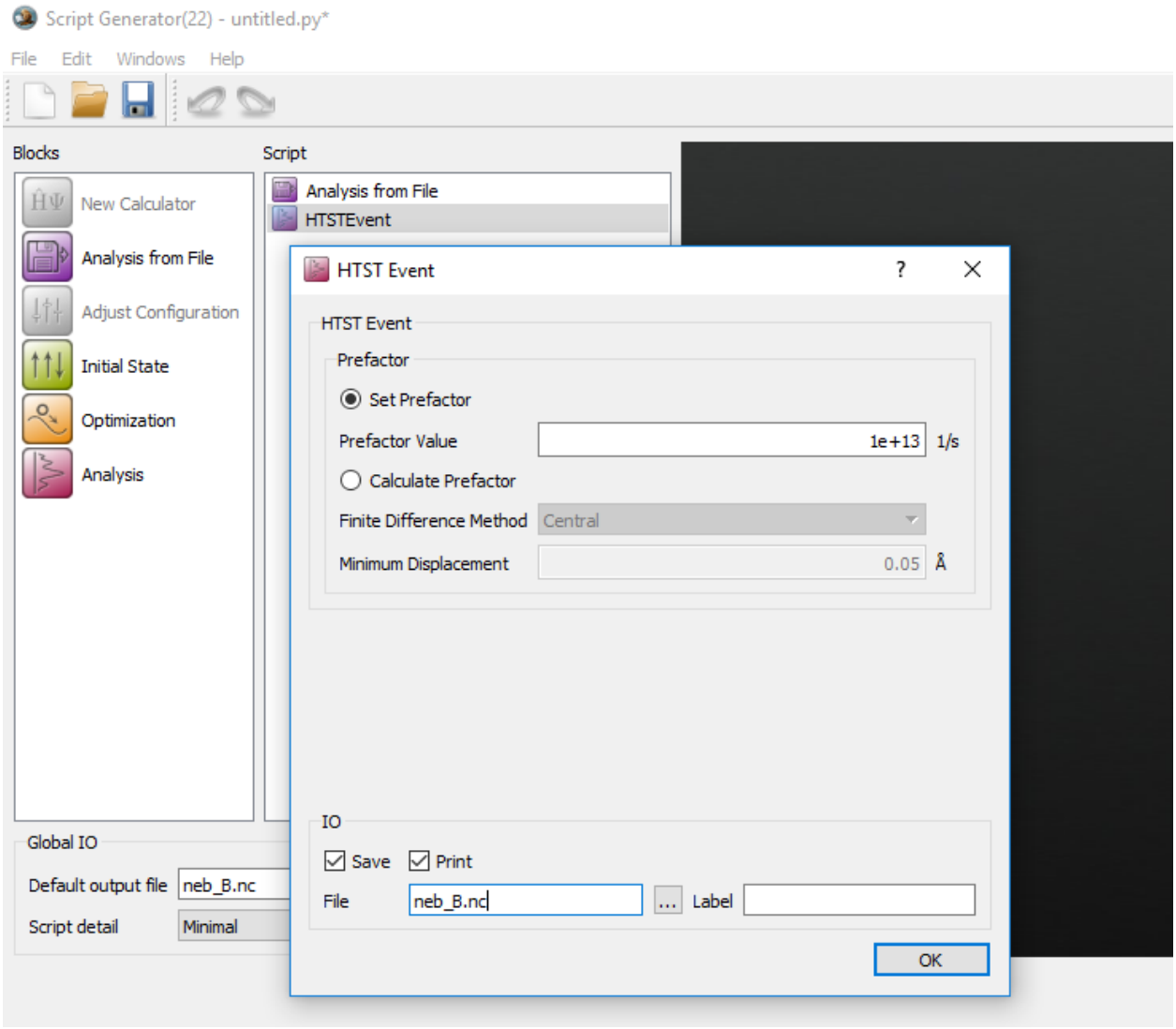
For detailed information about how reaction rates are calculated and the theory behind the harmonic transition state theory please refer to the *Calculating reaction rates using harmonic transition state theory*.

To set up the analysis:

1. Open the **Scripter**.
2. Double click on **Analysis from file** and select the `neb_B.nc` configuration with object ID *glD001*.
3. Add the **HTSTEvent** analysis and set the prefactor to 1e+13 1/s.


Movie Tool
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
Reaction coordinate (Å)
C
A
25 FPS


Movie Tool
2.5
2.0
1.5
1.0
0.5
0.0
Energy (eV)
0 1 2 3 4 5 6 7 8
Reaction coordinate (Å)
C
A
25 FPS



4.Run the analysis which will only take a minute. Here is the input script. `HTST_analysis.py`

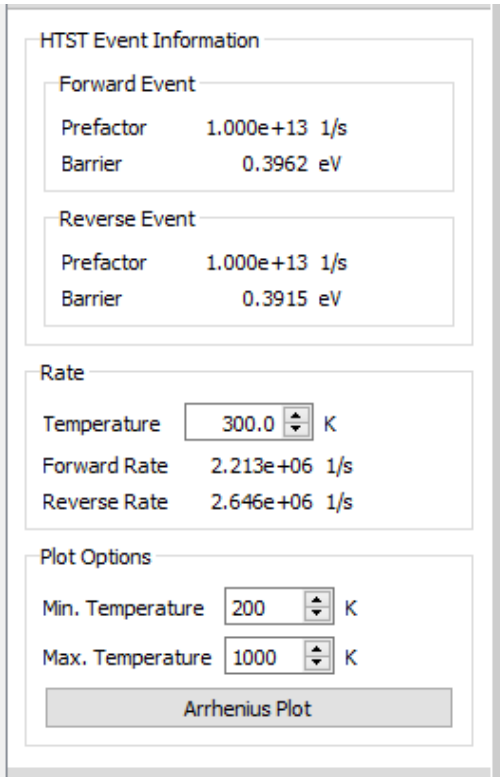
Note: The time required for an HTSTEvent analysis strongly depends on whether or not the prefactor is assumed or calculated. If you need to calculate the prefactor, it can take a long time, depending on the method and system. Therefore, it is usually a very good approximation to assume the prefactor for solid state reactions like bulk diffusion.

In order to visualize the results from the **HTSTEvents** analysis, select the corresponding object from the **LabFloor** and select the *HTST Rates* plugin on the right hand side.

Here, you can calculate the forward and reverse reaction rates for your diffusion mechanism. In this case you will find these two values to be very close to each other because the initial and final states are essentially equivalent.

Here, you can also set the options for an Arrhenius plot:

The table below summarized the results found in this tutorial.


HTST Event Information
Forward Event
Prefactor 1.000e+13 1/s
Barrier 0.3962 eV
Reverse Event
Prefactor 1.000e+13 1/s
Barrier 0.3915 eV
Rate
Temperature 300.0 K
Forward Rate 2.213e+06 1/s
Reverse Rate 2.646e+06 1/s
Plot Options
Min. Temperature 200 K
Max. Temperature 1000 K
Arrhenius Plot

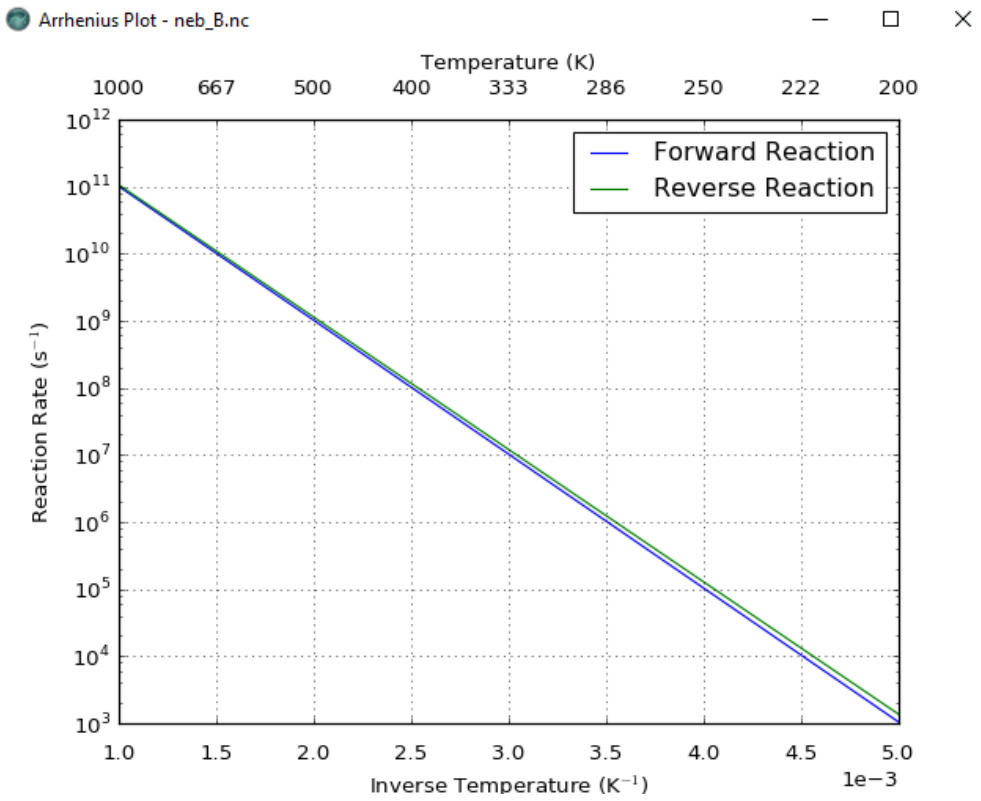

Arrhenius Plot - neb_B.nc
Temperature (K)
1000 667 500 400 333 286 250 222 200
Forward Reaction
Reverse Reaction
Reaction Rate (s⁻¹)
10¹² 10¹¹ 10¹⁰ 10⁹ 10⁸ 10⁷ 10⁶ 10⁵ 10⁴ 10³
1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
Inverse Temperature (K⁻¹)
1e−3

Table 3.1: Table: energy barriers (eV) and reaction rates (s^{-1}) calculated at 300 K for Li diffusion in $Li_{1-x}FePO_4$ along the B and C directions.

Direction	Barrier	k_{HTST}
B	0.39	2.6 x 10^{6}
C	2.30	1.9 x 10^{-26}

References

3.3.3 Open-circuit voltage profile of a Li-S battery: ReaxFF molecular dynamics

Version: 2017.1

Sulfur is a promising cathode material for rechargeable energy storage devices. During battery discharge, lithium ions migrate from the Li-rich anode to the sulfur cathode, and react with the sulfur (lithiation of the cathode):

$$x\mathrm{Li} + \mathrm{S} \rightarrow \mathrm{Li}_x\mathrm{S}$$

The **open-circuit voltage (OCV)** is the electrochemical driving force for this lithiation process.

In this tutorial we apply molecular dynamics (MD) simulations to calculate the OCV profile during cell discharge. We use the **ATK-ForceField** calculator and employ the **ReaxFF** force field developed by Islam an co-workers in Ref.[1]. The **simulated annealing** method is used to simulate amorphous Li_xS compounds for a range of compositions x, and the resulting OCV profile is compared to that in Fig. 3 in Ref.[Page 757, 1].

Note: Fig. 3 in Ref.[1] was made using data from a hybrid grand canonical Monte Carlo/molecular dynamics (GC-MC/MD) method, which is currently not available in QuantumATK. Instead of using the GC-MC to place Li atoms into low energy sites, we use here a simulated annealing method.

Theoretical definition of the open-circuit voltage

Neglecting enthalpic and entropic energy contributions, the OCV is calculated from total-energy differences:

$$\mathrm{OCV} = -\frac{E_{\mathrm{Li}_x\mathrm{S}} - nE_{\mathrm{Li}} - mE_{\mathrm{S}}}{n},$$

where n (m) is the number of lithium (sulfur) atoms in the Li_xS compound (such that $x = n/m$), and E_{Li} and E_{S} are the total energies per atom of the pure Li and S crystals, respectively.

Outline

We first consider $Li_{0.4}S$ and describe all steps needed to compute the open-circuit voltage for partial lithiation of pure S to $Li_{0.4}S$:

- *Amorphous Li0.4S compound*
- *Simulated annealing*
- *Open-circuit voltage*

ATK Python scriting is then used for more automated simulations for a range of Li_xS compositions. The full OCV profile is computed and radial distribution functions for different compositions are investigated:

[1] MdăM. Islam, A.ăOstadhossein, O.ăBorodin, A.ăT. Yeates, W.ăW. Tipton, R.ăG. Hennig, N.ăKumar, and A.ăC.ăT. van Duin. ReaxFF molecular dynamics simulations on lithiated sulfur cathode materials. *Phys. Chem. Chem. Phys.*, 17:3383–3393, 2015. doi:10.1039/C4CP04532G.

- *Full open-circuit voltage profile*
- *Radial distribution functions*

Tip: QuantumATK Python scripts are provided for download throughout this tutorial. You may also download all of them in a single ZIP file: `scripts.zip`.

Note: This is an advanced tutorial – certain basics of operating the **ATK-ForceField** calculator and the **QuantumATK analysis tools** are not explained in great detail. Please see the tutorial *How to Setup Basic Molecular Dynamics Simulations* if needed.

Amorphous $Li_{0.4}S$ compound

The cahtode material is not expected to be in a perfect crystalline phase under battery operating conditions; it is most likely amorphous. QuantumATK offers several different tools for generating amorphous structures, for example the **Amorhpous Prebuilder** and the **PackMol** tool. We use here the latter.

Open QuantumATK and create a new project, preferably in a new directory on your harddrive. The open the

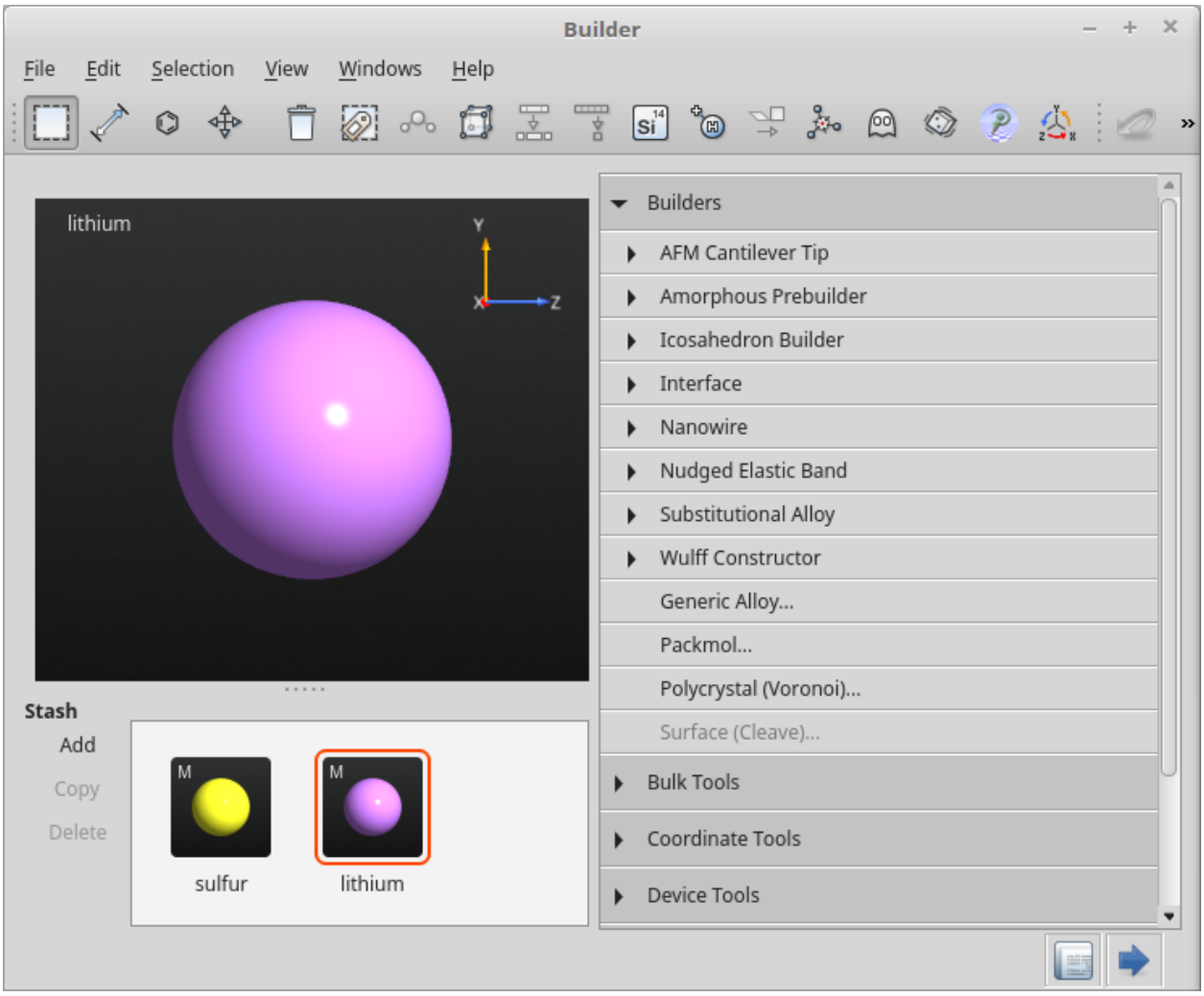
Builder.

PackMol

The initial structure of the amorphous $Li_{0.4}S$ compound is here generated using **PackMol**, which randomly mixes input **molecule** configurations into a reasonable amorphous **bulk** configuration. You therefore need to first add atomic sulfur and atomic lithium to the Builder Stash:

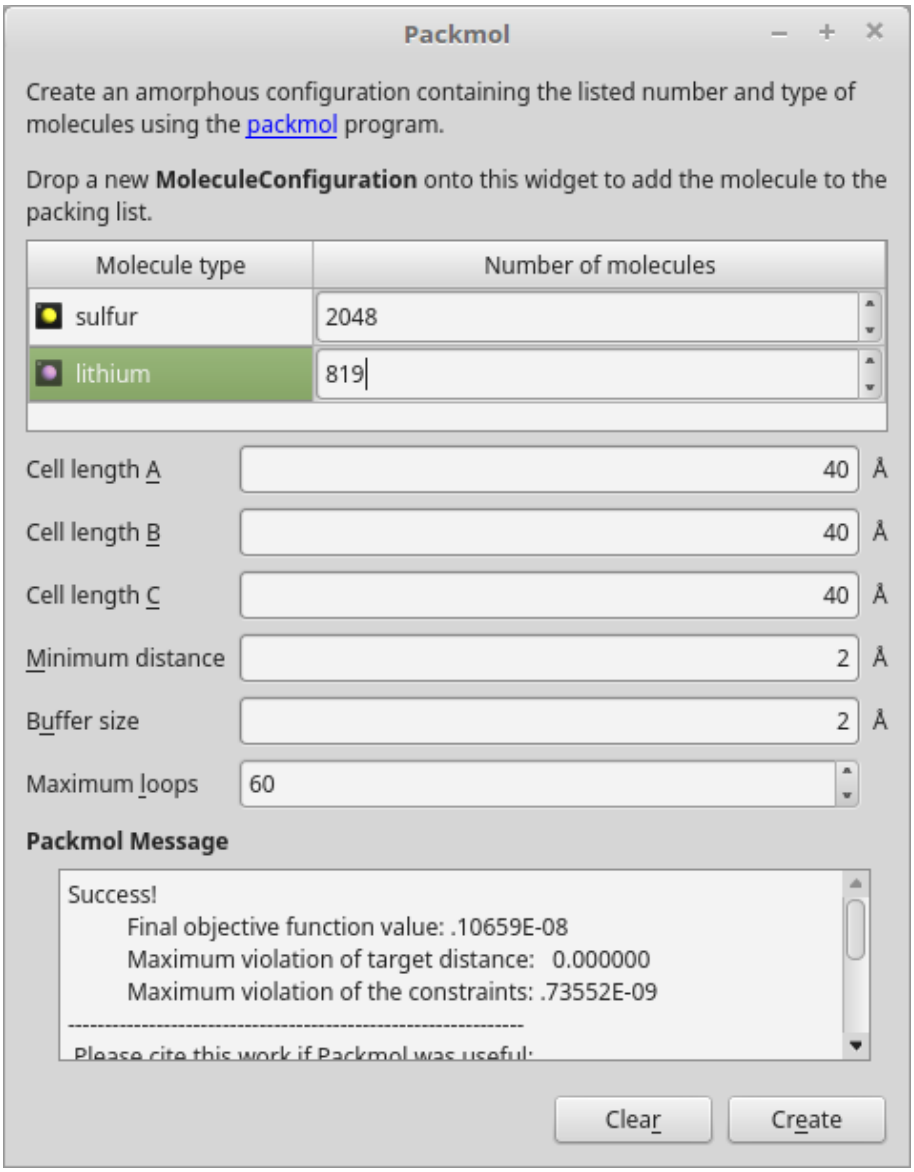
- In the **Stash**, click *Add* → *New Configuration* to add a hydrogen atom in a molecule configuration.
- Select the H atom and use the

tool to turn the H atom into S.
- Repeat the two steps above to create also a lithium atom.
- Rename the Stash items sulfur and lithium, respectively.

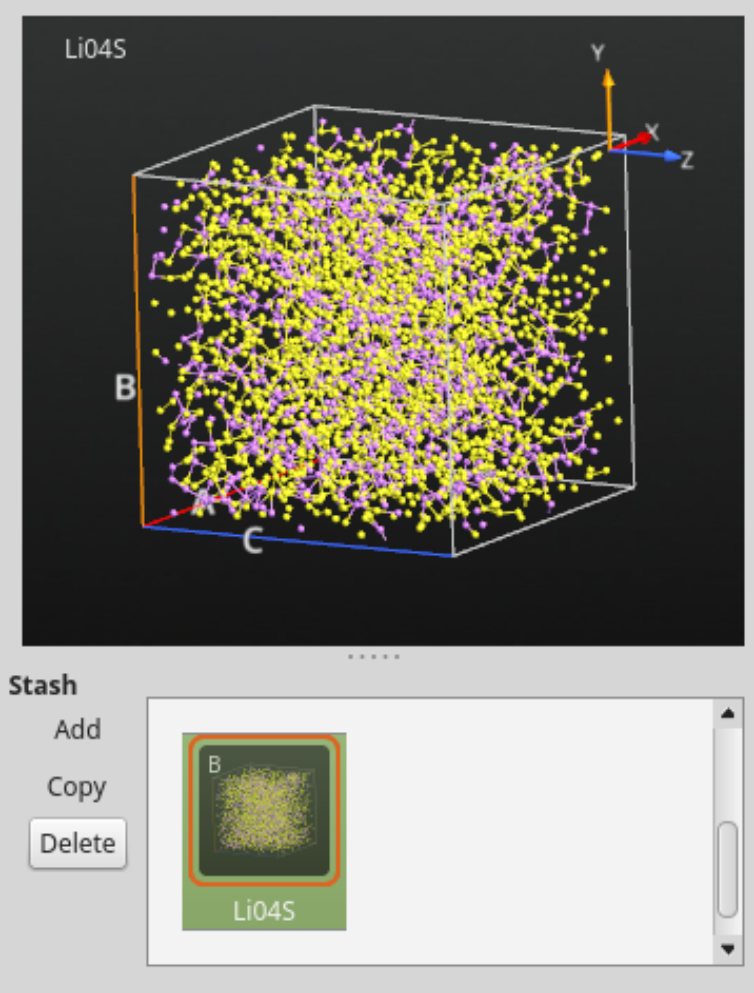


Navigate to the *Builders* → *PackMol* plugin and click it. The **PackMol** widget should now open. Do the following to create a $Li_{0.4}S$ amorphous crystal containing 2048 sulfur atoms.

- Drag-and-drop the sulfur and lithium atoms from the **Stash** onto the **PackMol** widget. The configurations will be added to the Molecule type list.
- Increase the number of sulfur entities to 2048, and the number of lithium entities to 819 (computed from 2048 $\times 0.4 =$ 819).
- Leave all other settings at their defaults, and click **Create**.



The amorphous $Li_{0.4}S$ structure is automatically added to the Stash. Rename it Li04S.



Geometry optimization

The amorphous structure produced by **PackMol** is guarenteed to have some minimal distance between all mixed entities (2 Å by default), but the structure may still be far from equilibrium. It is therefore a good idea to run an optimization of the atomic coordinates and lattice vectors before heating the system to a high temperature using MD.

- Send the Li04S configuration to the **Script Generator** and add the **New Calculator** and **OptimizeGeometry** script blocks.
- Open the calculator block, and select the **ATK-ForceField** calculator. The *ReaxFF_LiS_2005* force field is automatically selected – it was generated in Ref.[1].
- Open the geometry optimization block, and increase the force tolerance to 0.5 eV/Å and increase the maximum number of optimization steps to 1000. Also tick the *Save trajectory* option and enter `Li04S` as the file name for saving the relaxation trajectory.

Save the QuantumATK Python script as `Li04S.py` and run it using either the **Job Manager** or manually in a terminal:

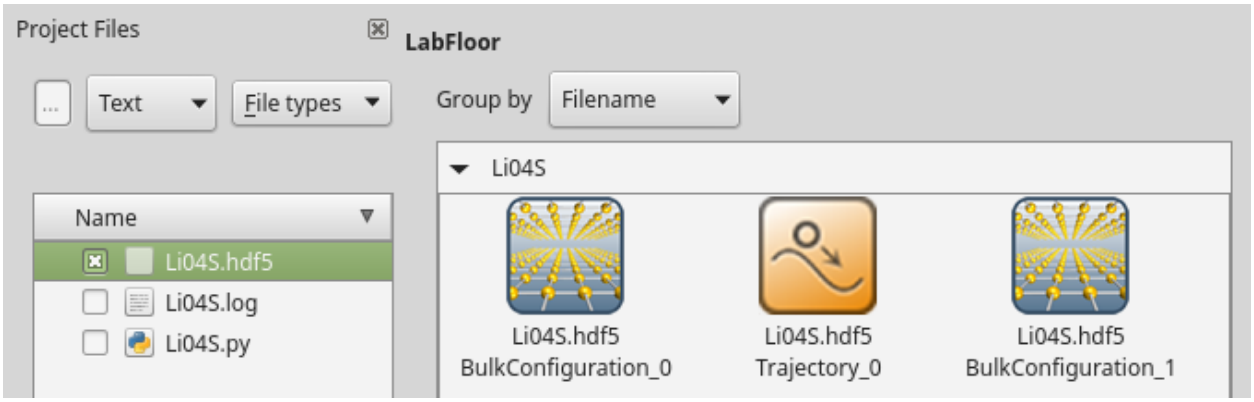
```
atkpython Li04S.py 2>&1 | tee Li04S.log
```

You may also download the script: `Li04S.py`.

Important: Parallelizing ATK-ForceField calculations over more than one physical core is most efficiently done using **OpenMP** threading rather than MPI, see the relevant sections in tutorials on using the Job Manager for *local* and *remote* QuantumATK calculations with threading only.

If running QuantumATK from a terminal, the environment variable `OMP_NUM_THREADS` should either be unset or set to the maximum number of OpenMP threads you intend to use. We recommend to set `MKL_DYNAMIC=TRUE` to allow MKL to dynamically change the number of threads.

The geometry optimization should now have finished, and the output data should be available on the QuantumATK **LabFloor**:



You may use the **Viewer** to visualize the relaxation trajectory; you will see that the unit cell expands a bit upon minimization of atomic forces and stress on the cell.

Simulated annealing

The room temperature structure of the amorphous $Li_{0.4}S$ compound (atomic coordinates and unit cell vectors) is not know *a priori*, so we use the **simulated annealing** method to find it. The basic idea is to heat up the system 1600 K and then slowly cool it down to 300 K using molecular dynamics, all at fixed pressure but allowing the unit cell volume to adapt (NPT dynamics).

The computational workflow is this:

1)Equilibrate the system at 1600 K.

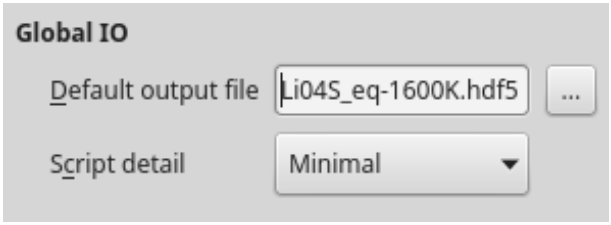
2)Slowly cool down the system from 1600 to 300 K.

3)Equilibrate the system at 300 K.

Equilibration at 1600 K

The bulk configuration labeled *BulkConfiguration_1* on the **LabFloor** (saved in `Li04S.hdf5`) is the optimized one. Select it and drop it on the **Scripter**. The appropriate ATK-ForceField calculator block is automatically added, so add just the **MolecularDynamics** script block.

First, set the default output file name to `Li04S_eq-1600K.hdf5`:



Then open the MD script block and adjust the settings:

•Select the NPT Berendsen MD type.

•Increase the number of MD steps to 50000.

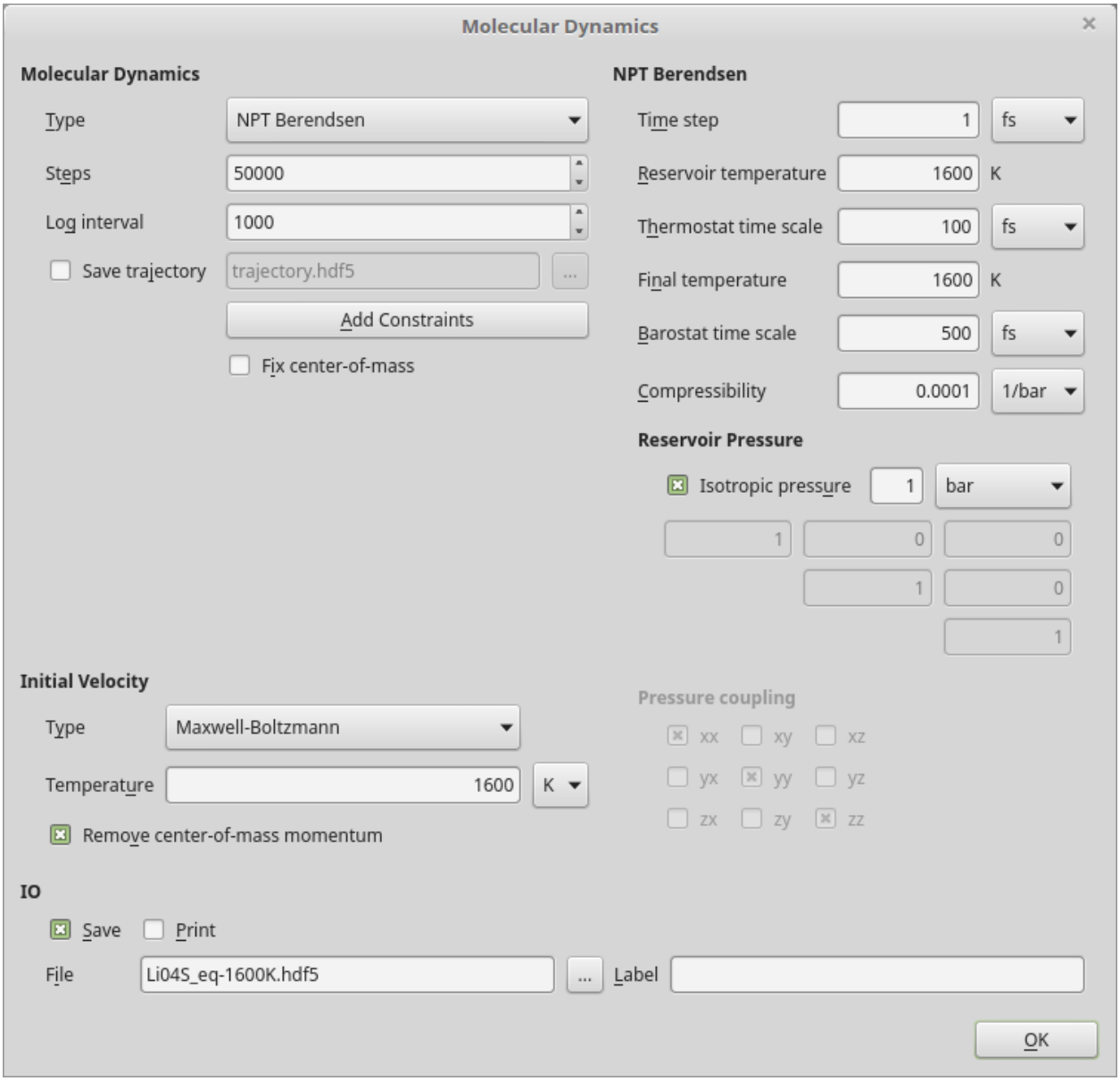
•Increase the log interval to 1000.

•Set the both reservoir temperature and the final temperature to 1600 K.

•Set also the temperature for the Maxwell–Boltzmann distribution of initial particle velocities to 1600 K.

•Optional: Untick the *Save trajectory* option.

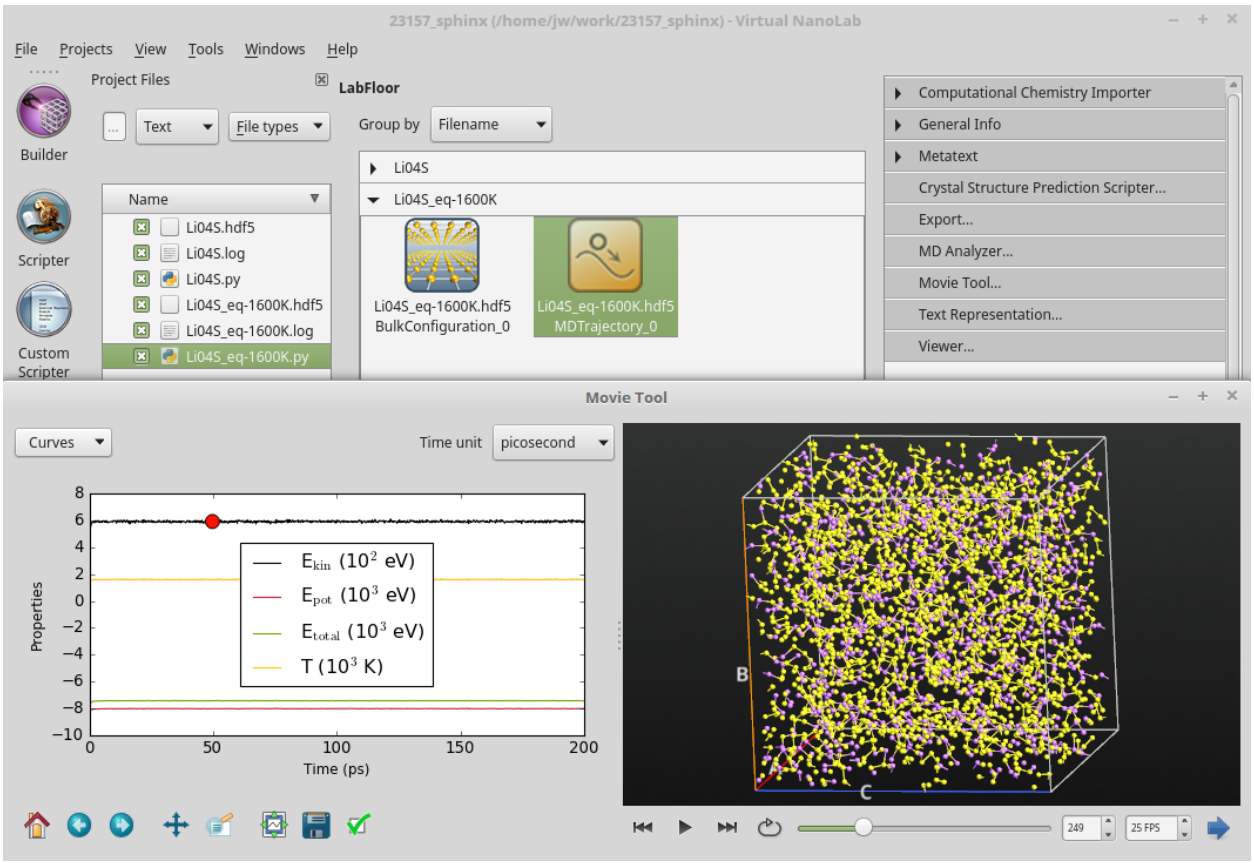
•Click **OK** to save te settings.



Save the script as `Li04S_eq-1600K.py` and run it. Depending on the computational resources (number of OpenMP threads), the job may take up to an hour to finish.

You may also download the QuantumATK Python script: `Li04S_eq-1600K.py`.

Once the calculation is done, the resulting MD trajectory should pop up on the **LabFloor**. Use the **Movie Tool** to visualize the progress of the MD simulation. You can use mouse right-click to enable the *Show bonds* option if needed. Note how the temperature and energies stay roughly constant throughout the simulation, and that the unit cell expands significantly at the beginning, but then stays fairly constant while the atoms move around inside it. The configuration is thereby equilibrated at constant temperature.



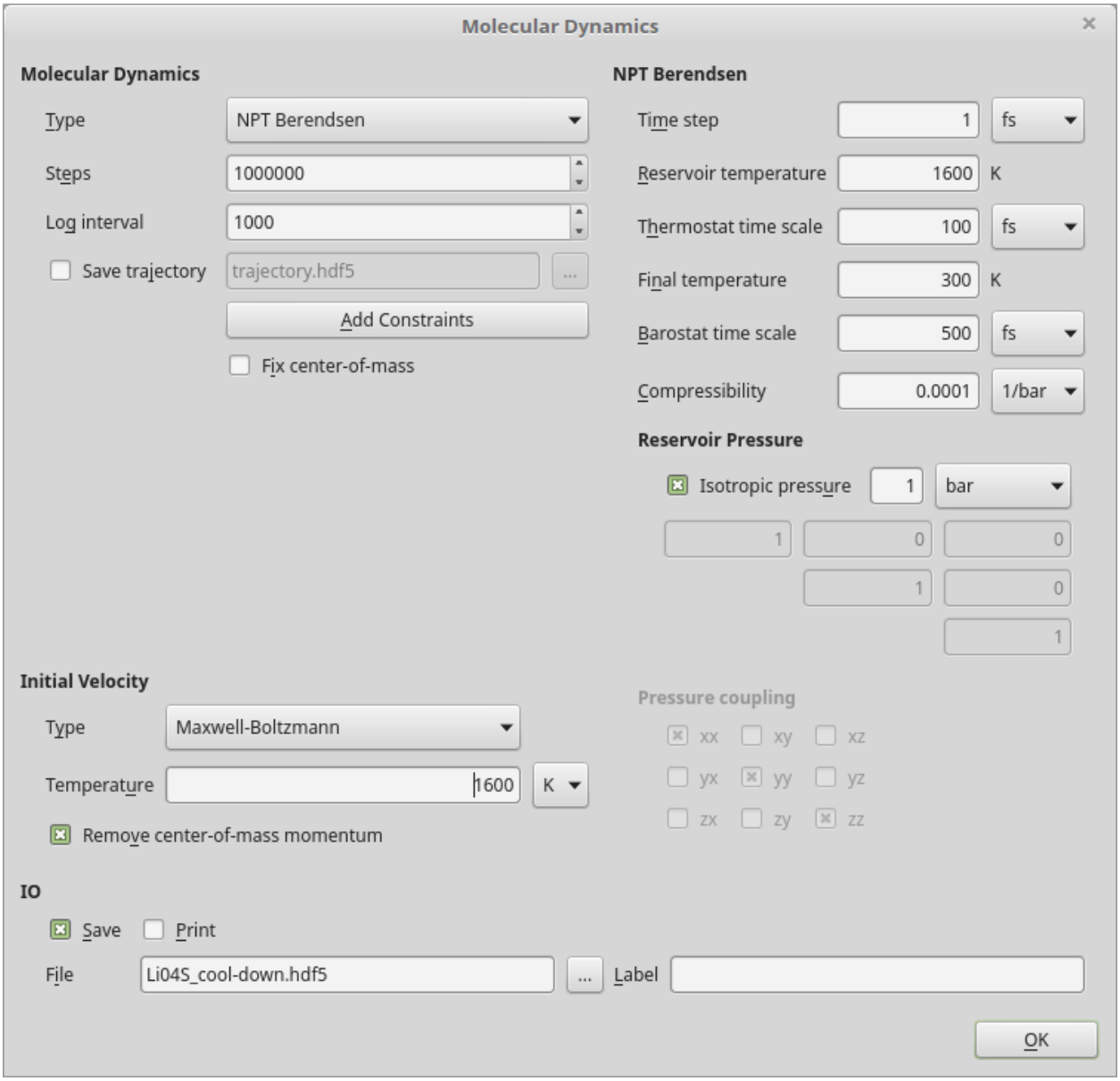
Cool-down from 1600 to 300 K

The $Li_{0.4}S$ configuration is now equilibrated and ready to be cooled down to room temperature. In the **Movie Tool**, use the icon to **transfer the last MD image** in the MD trajectory to the **Scripter**.

Add the **New Calculator** and **MolecularDynamics** script blocks, and select again the ATK-ForceField calculator with the appropriate force field. Then set the default output file name to `Li04S_cool-down.hdf5`, and adjust the settings for the MD block:

- Select the NPT Berendsen MD type.
- Increase the number of MD steps to 1 mio.
- Increase the log interval to 1000.
- Set the reservoir temperature to 1600 K and the final temperature to 300 K.
- Set the temperature for the Maxwell–Boltzmann distribution of initial particle velocities to 1600 K.
- Optional: Untick the *Save trajectory* option.
- Click **OK** to save te settings.

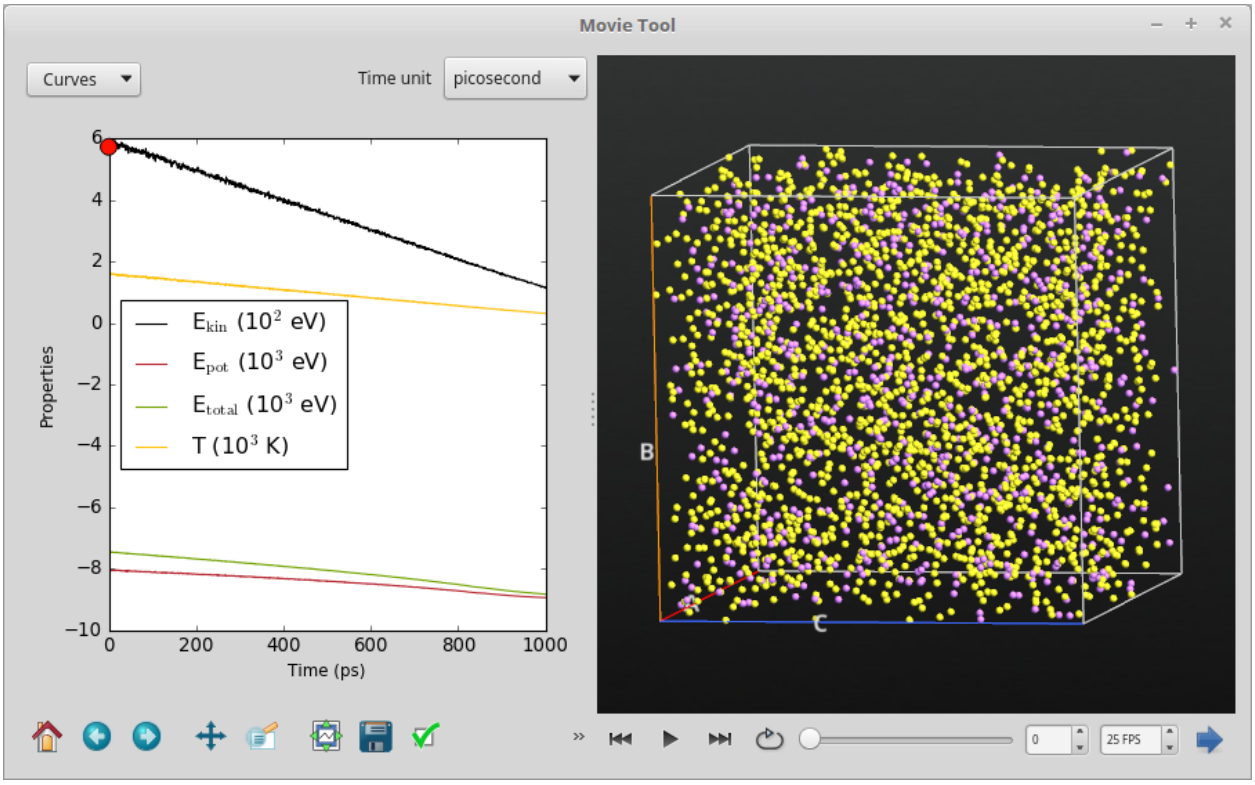
This will result in a **linear temperature ramp** over 1 ns of MD simulation time (10^6 steps $\times 1$ fs/step = 1 ns), and a cooling rate of 1.3 K/ps.



Save the script as `Li04S_cool-down.py` and run it. You may also download the QuantumATK Python script: `Li04S_cool-down.py`.

Warning: The calculation may take a full day if executed on a single core, so it is recommended to use threading when running this job.

Once the calculation is done, the resulting MD trajectory should become available on the **LabFloor**. Use again the **Movie Tool** to visualize the progress of the MD simulation. Notice how the temperature decreases linearly from 1600 K to 300 K, and that the energies of the system decrease as well. Moreover, running the movie from start to end, it is clear that the $Li_{0.4}S$ unit cell shrinks when cooled down, quite as expected.

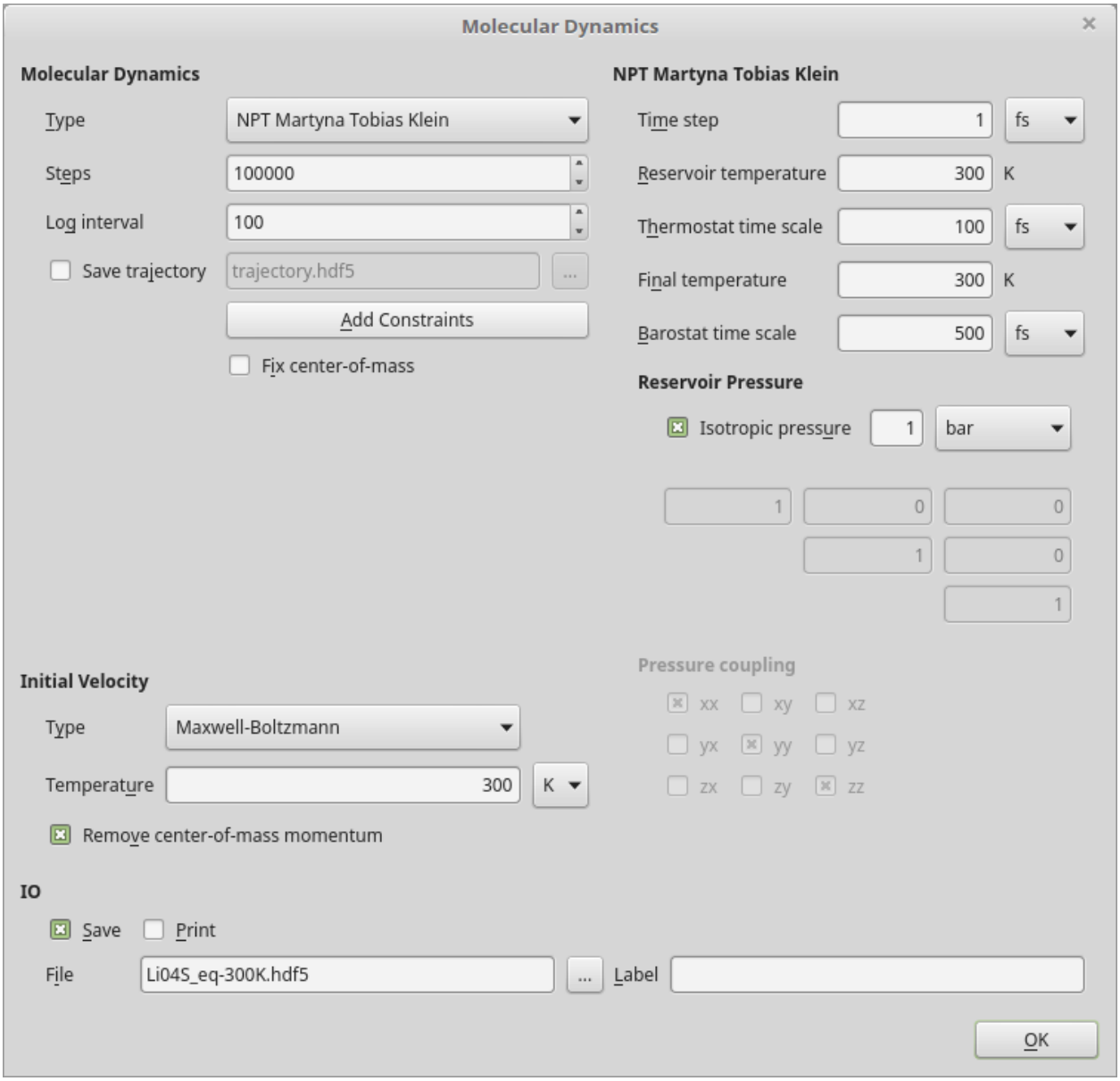


Equilibration at 300 K

Final step is to equilibrate the room temperature amorphous $Li_{0.4}S$ structure. In the **Movie Tool**, use again the icon to transfer the last MD image from the cool-down MD trajectory to the **Scripter**.

Add the **New Calculator** and **MolecularDynamics** script blocks, and select again the ATK-ForceField calculator with the appropriate force field. Then set the default output file name to `Li04S_eq-300K.hdf5` and adjust the settings for the MD block:

- Select the NPT Martyna-Tobias-Klein MD type.
- Increase the number of MD steps to 100000.
- Increase the log interval to 100.
- Set the reservoir temperature to 300 K.
- Set the temperature for the Maxwell–Boltzmann distribution of initial particle velocities to 300 K.
- Optional: Untick the *Save trajectory* option.
- Click **OK** to save te settings.



Save the script as `Li04S_eq-300K.py` and run it. You may also download the QuantumATK Python script: `Li04S_eq-300K.py`.

Once the calculation is done you may use the **Movie Tool** to check that the calculation went as expected (constant temperature, etc.).

The plot below shows the evolution of the total energy per atom and the volume as the MD equilibration progresses. Both energy and volume reaches a stable mean (the black dashed lines), and are thefore well equilibrated after 1000 MD steps. The script `evolution.py` was used to generate the plots.

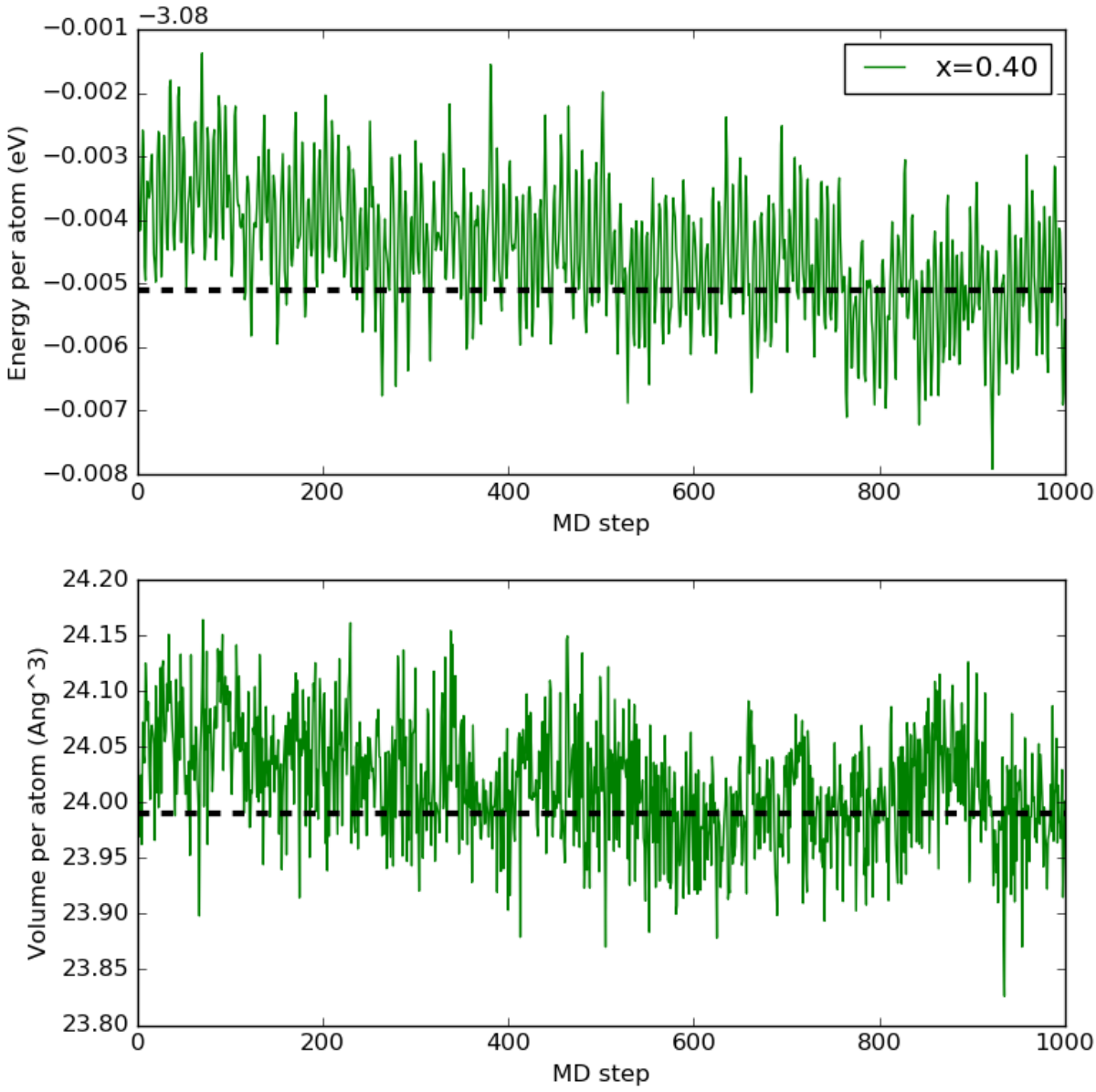


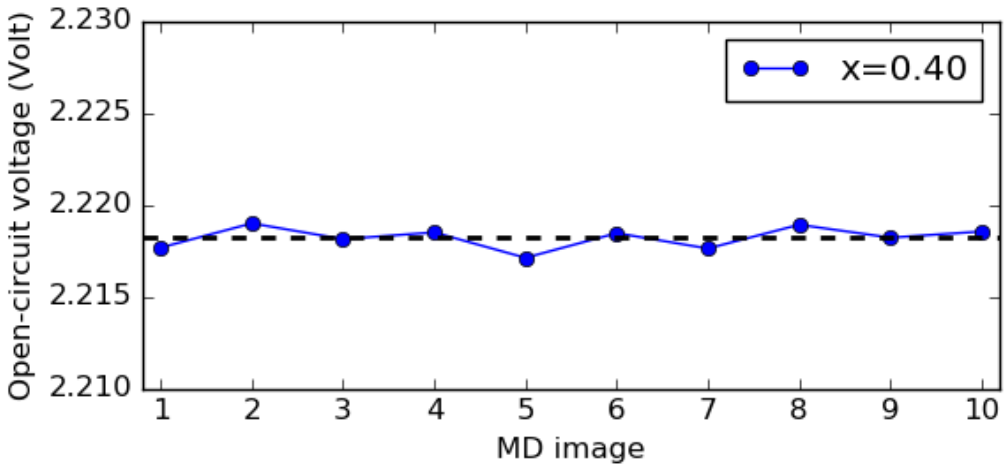
Fig. 3.70: Evolution of the total energy per atom (top) and the volume (bottom) as the MD equilibration progresses. Black dashed lines indicate the average over the last 100 MD images.

Open-circuit voltage

The OCV is calculated from total-energy differences as discussed in the introduction. Each total energy should be for a geometry optimized structure. To check that the calculated OCV does not depend significantly on the chosen equilibration MD image, you should first perform geometry optimization at fixed unit cell volume for several of the last equilibrated images, and compute the corresponding OCV values. Use the script `Li04S_relax.py` to do this.

You also need the total energy per atom of the pure *BCC* Li and *ORTH* S_8 crystals. The scripts `Lithium.py` and `a-Sulfur.py` perform the required geometry optimizations and calculate the total energies. Run both scripts.

Then calculate and plot the OCV for all the 10 selected $Li_{0.4}S$ MD images using the script `ocv.py`. The plot is shown below. The black line indicates the average OCV, and it is clear that even though choosing different MD images from the equilibration MD simulation results in slightly different OCV values, they are very similar. For each Li_xS composition, we shall below use the average OCV over 10 MD images.



Full open-circuit voltage profile

So far, we have obtained the OCV for $Li_{0.4}S$. To plot the full OCV profile for litiation to the sulfur cathode corresponding to the reduction pathway of S_8 to Li_xS, you should repeat the procedure outlined above for a range of Li concentrations. However, that may be a somewhat tedius workflow – some degree of automation would be useful.

ATK Python scripting is ideal for assembling the different MD scripts given above for $Li_{0.4}S$ into **a single script** that will run all calculations for a given lithium concentration. The script `x0.40_full.py` does this for x=0.4. Download it and use it as a template for running all required MD simulations for each composition:

- Use **PackMol** to generate structures for the desired compositions and save each structure in a separate data file named `x#.##.hdf5`, where `#.##` indicates the Li concentration, e.g. `0.40`.
- Run a separate script for each Li_xS composition (copy `x0.40_full.py` and edit it to use the above generated `hdf5` file as input and output data file.)

Note: Calculations for Li-rich compounds may take several days even if executed on 16 cores using OpenMP threading.

The cool-down MD simulation takes by far the largest part of the job wall-time. You may be able to decrease the time-to-result by decreasing the number of MD steps used in the cool-down from 1600 K to 300 K (and thereby increase the cooling rate), but be careful in doing so, since the cooling rate may affect your results.

Also, it turns out that equilibration often converges faster for Li-rich structures than for dilute Li concentrations, so it may be possible to decrease the number of MD steps for those simulations also.

Finally, plot the OCV profile over the Li_xS compositions using the script `ocv_profile.py`.

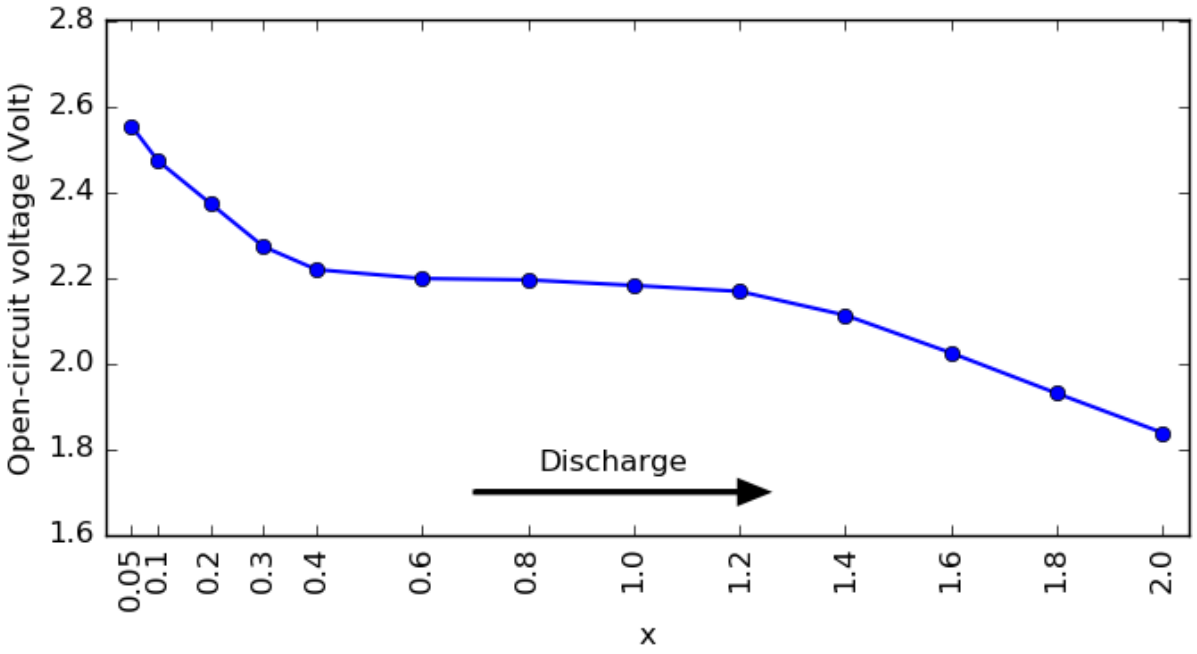


Fig. 3.71: Open circuit voltage profile during lithiation of the sulfur cathode.

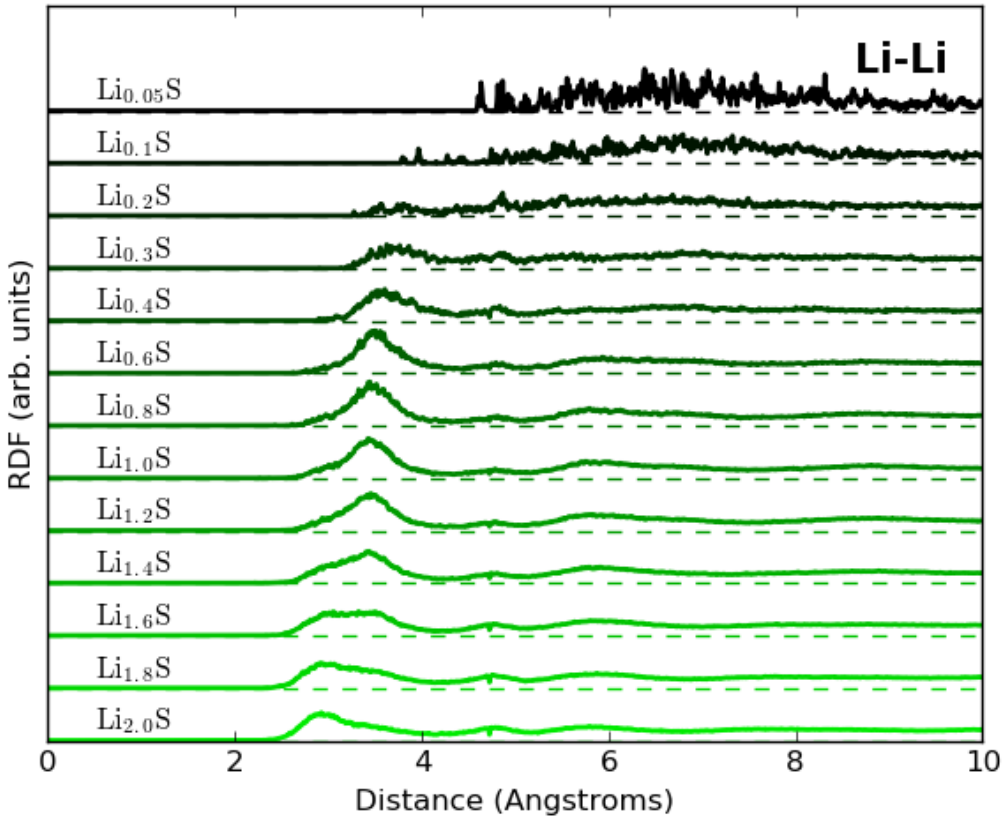
This plot shows the full OCV profile and it is almost same as in Fig. 3 in[Page 757, 1]. Typically, experimental discharge voltage profiles for a Li-S battery can exhibit two or three reduction stages depending on the electrolyte. In this case, ReaxFF molecular dynamics calculations predict an initial drop, a flatter region, and again a voltage drop, all in agreement with experiments.

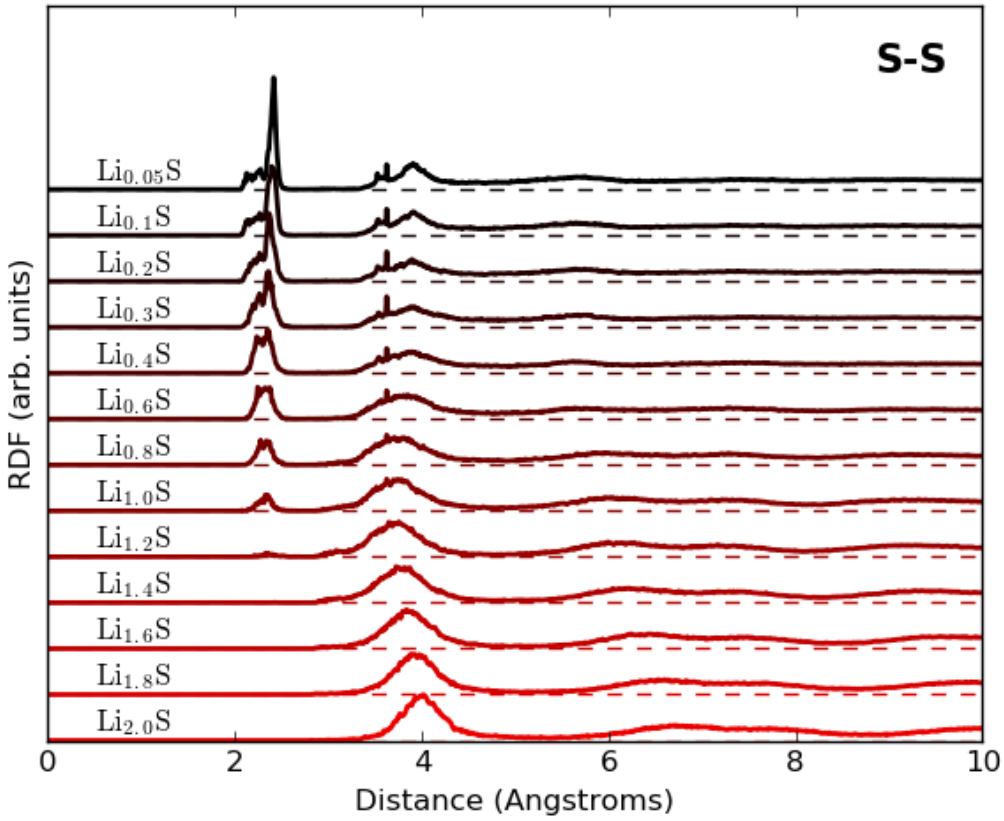
The initial drop may be induced by the formation of high-order polysulfides, *i.e.* sulfur-rich compounds from the S_8 crystal. But it is not stable enough, so it is a fast process. The plateau at ~2.1 V is assigned to a slow process involving the further reduction to low-order polysulfides, *i.e.* Li-rich compounds. Here we only consider the litiated cathode, but in reality, these are very complex processes that may need more detailed understanding depending on the electrolyte composition.

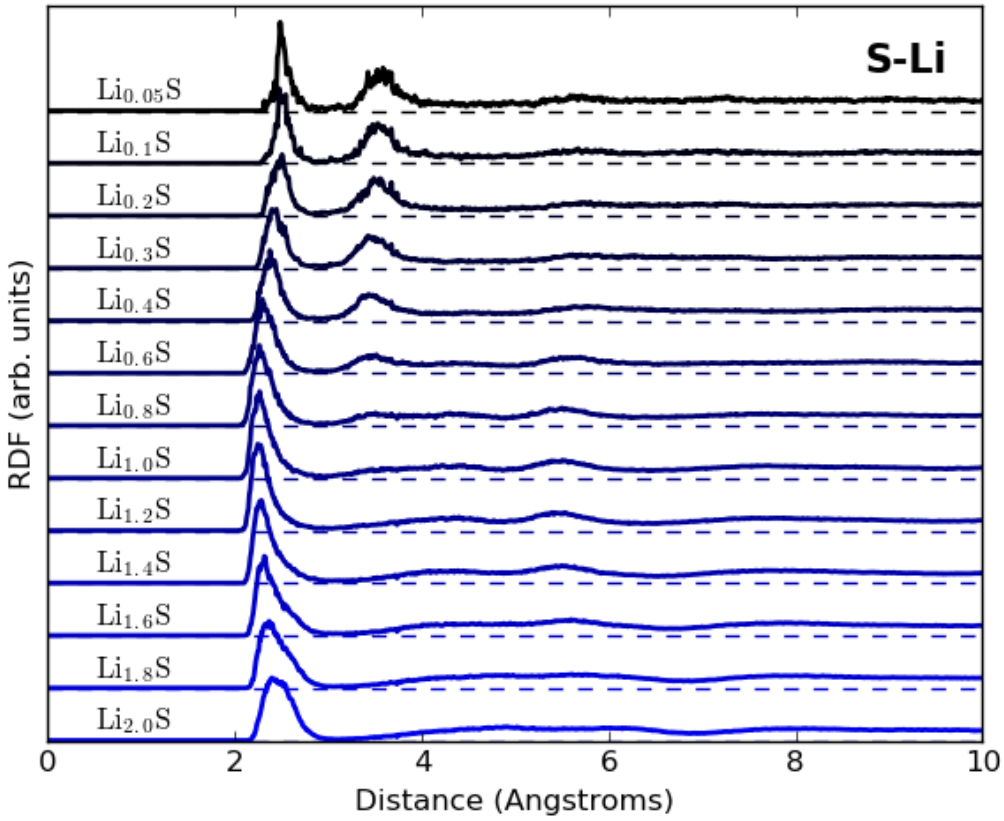
Radial distribution functions

It is also useful to analyze the radial distribution functions (RDFs) for S–S, Li–Li and S–Li in different Li_xS compositions. You may use the following scripts to obtain the plots given below: `rdf_lili.py`, `rdf_ss.py` and `rdf_sli.py`.

The RDFs for Li–Li, S–S and S–Li distances compare well with Fig. 5 in[Page 757, 1], and clearly indicate that 1) Li–Li bonds at ~3 Å distance are formed upon Li uptake during lithiation of the sulfur cathode, 2) short S–S bonds disppear upon Li uptake, and 3) both short and long S–Li distances exist for small Li contents, but the short S–Li bonds (~2.5 Å) dominate at large Li uptake.


Li-Li
$Li_{0.05}S$
$Li_{0.1}S$
$Li_{0.2}S$
$Li_{0.3}S$
$Li_{0.4}S$
$Li_{0.6}S$
$Li_{0.8}S$
$Li_{1.0}S$
$Li_{1.2}S$
$Li_{1.4}S$
$Li_{1.6}S$
$Li_{1.8}S$
$Li_{2.0}S$
RDF (arb. units)
0
2
4
6
8
10
Distance (Angstroms)


S-S
$Li_{0.05}S$
$Li_{0.1}S$
$Li_{0.2}S$
$Li_{0.3}S$
$Li_{0.4}S$
$Li_{0.6}S$
$Li_{0.8}S$
$Li_{1.0}S$
$Li_{1.2}S$
$Li_{1.4}S$
$Li_{1.6}S$
$Li_{1.8}S$
$Li_{2.0}S$
RDF (arb. units)
0
2
4
6
8
10
Distance (Angstroms)


S-Li
$Li_{0.05}S$
$Li_{0.1}S$
$Li_{0.2}S$
$Li_{0.3}S$
$Li_{0.4}S$
$Li_{0.6}S$
$Li_{0.8}S$
$Li_{1.0}S$
$Li_{1.2}S$
$Li_{1.4}S$
$Li_{1.6}S$
$Li_{1.8}S$
$Li_{2.0}S$
RDF (arb. units)
0
2
4
6
8
10
Distance (Angstroms)

References

3.3.4 Photocurrent in a silicon p-n junction

Version: O-2018.06

In this tutorial, you will learn how to calculate the photocurrent through a device under illumination. We consider a 7.6 nm sized silicon p-n junction, and showcase the Photocurrent Analyzer introduced with the QuantumATK O-2018.06 release.

The photocurrent is calculated by adding the electron-photon interaction to the device Hamiltonian, using first-order perturbation theory, as described in[1] and in the ATK Manual: *Photocurrent*.

Device ground state

At first you will perform the self-consistent device calculation. Then, using the ground-state device Hamiltonian, you will calculate and analyze the photocurrent.

You should use the script `silicon_pn_junction.py` to run the ground-state calculation – both the doped device configuration and the appropriate NEGF calculator are defined in the script. Results are saved in `silicon_pn_junction.hdf5`. Simply download the script and run it using the **Job manager** or from a terminal. Once the calculation is done, you can see the file `silicon_pn_junction.hdf5` on the **LabFloor**. The calculation takes *ca.* 20 mins with 4 MPI processes.

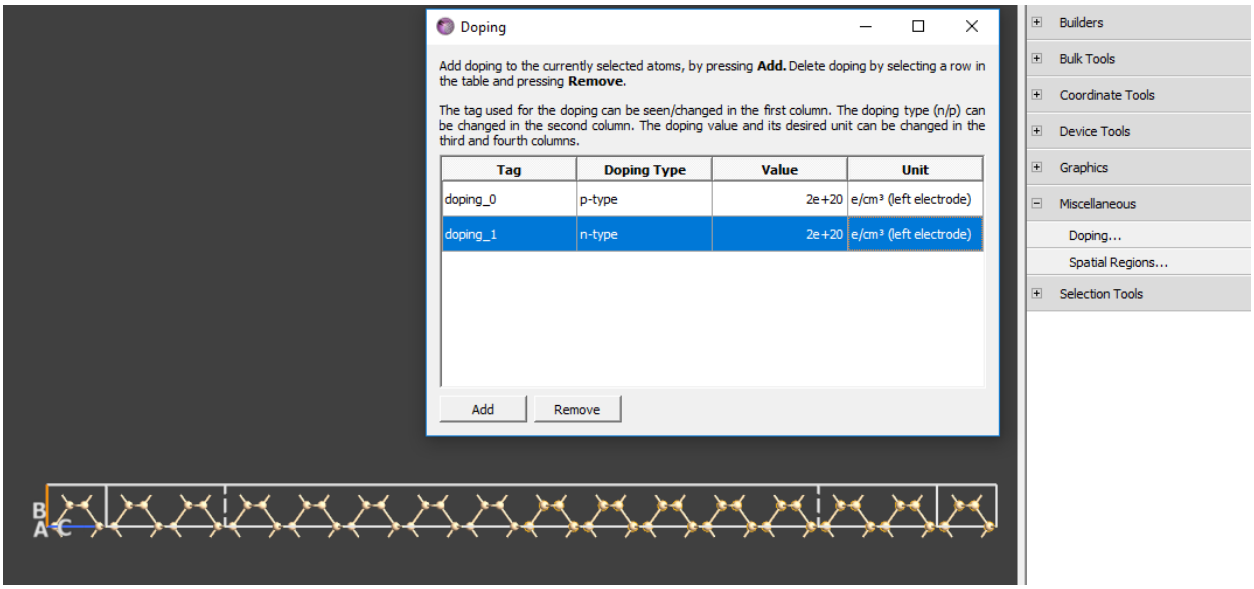
Brief instructions for building the device configuration and setting up the NEGF calculation are also given below.

Silicon p-n junction

The 7.6 nm silicon device should consist of 28 Si layers (76.0284 Å) and 10.8612 Å left and right electrodes. The *p* and *n* doping levels should be both $2*10^{20}$ e/cm^3 for this quite short Si device in order to get a reasonable electronic structure with fairly flat band edges near the electrodes and a small gap from the VBM and CBM to the Fermi level.

If you need more information on how to build a junction, please see the tutorial *Silicon p-n junction*.

[1] Mattias Palsgaard, Troels Markussen, Tue Gunst, Mads Brandbyge, and Kurt Stokbro. Efficient first-principles calculation of phonon assisted photocurrent in large-scale solar cell devices. *arXiv:1801.03683*, 2018. URL: http://arxiv.org/abs/1801.03683.



NEGF calculation

The supplied script `silicon_pn_junction.py` uses a NEGF calculator with the settings given below.

•Basis set: *PseudoDojo* and *Low*.

•Exchange-correlation: *GGA-1/2* and *PBE*.

•k-point sampling: *4 Å* density in the A and B directions.

•Density mesh cutoff: *30 Ha*.

•Default output file: `silicon_pn_junction.hdf5`.

Run the script using the **Job manager** (click the button to start the calculation).

Photocurrent

You are now ready to calculate the photocurrent. Open the **Script Generator** and add a **Analysis from File**.

•Select the file `silicon_pn_junction.hdf5`.

•Make sure the *DeviceConfiguration* object is ticked.

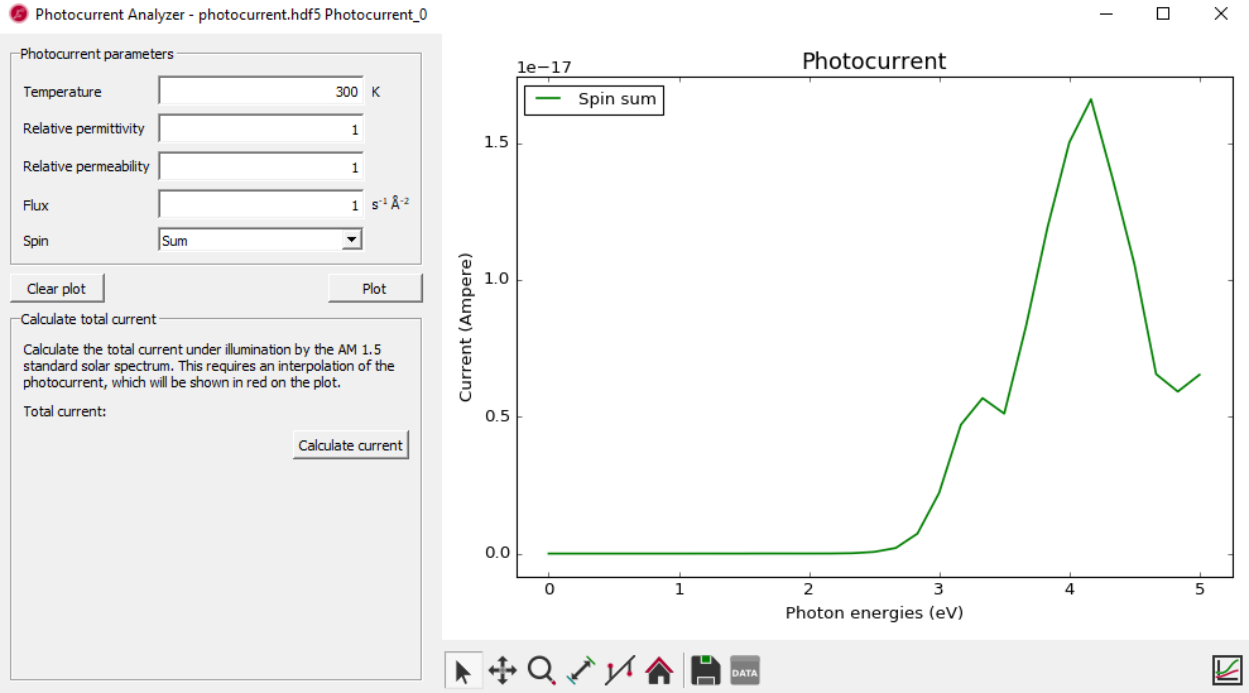
Next, in order to calculate the photocurrent, add the *Analysis* → *Photocurrent* block. Set the photon energy range to 0 – 5 eV with 31 points, and set the k-point sampling to 9 x 9. Keep all other parameters at default values. You can also download the script `photocurrent.py`. Send the script to the **Job manager**, save it as `photocurrent.py`, and click on the button to run the calculation.

Note: The photocurrent calculation takes *ca.* 3.5 hours with 4 MPI processes, so you may consider running the calculation on a remote computing cluster, if possible.

Once the calculation is done, you can see the file `photocurrent.hdf5` in the **LabFloor**. Note that the calculation can be made faster by decreasing the number of photon energies and increasing the `energy_resolution`, leading to fewer electron energy points.

Select the *Photocurrent_0* object of `photocurrent.hdf5` on the **LabFloor**. Open the **Photocurrent Analyzer** in the right-hand panel.

You can easily plot the current *vs.* photon energies by clicking **Plot**. This is shown in the figure below.



You can also calculate the total current based on illumination by the AM1.5 standard solar spectrum obtained from Solar spectra website. Once you click **Calculate current**, the interpolated photocurrent will be indicated by a red line. This interpolated function is multiplied with the flux from the reference spectrum at each photon energy, and the result is then numerically integrated to calculate the total current from the device under the given illumination.

If the photon energies used to calculate the photocurrent do not cover the range of photon energies in the AM1.5 spectrumn, you will see a warning message. In this case, the interpolation will include data outside the data range, so the calculated total current may therefore be inaccurate. We therefore recommend to calculate the total current only from photocurrents covering the full range of the AM1.5 standard solar spectrum, which is 0.3 eV to 4.43 eV. Photocurrents covering a larger range is not a problem, as data outside the range of the reference spectrum simply do not contribute. The figure below shows an example of this.

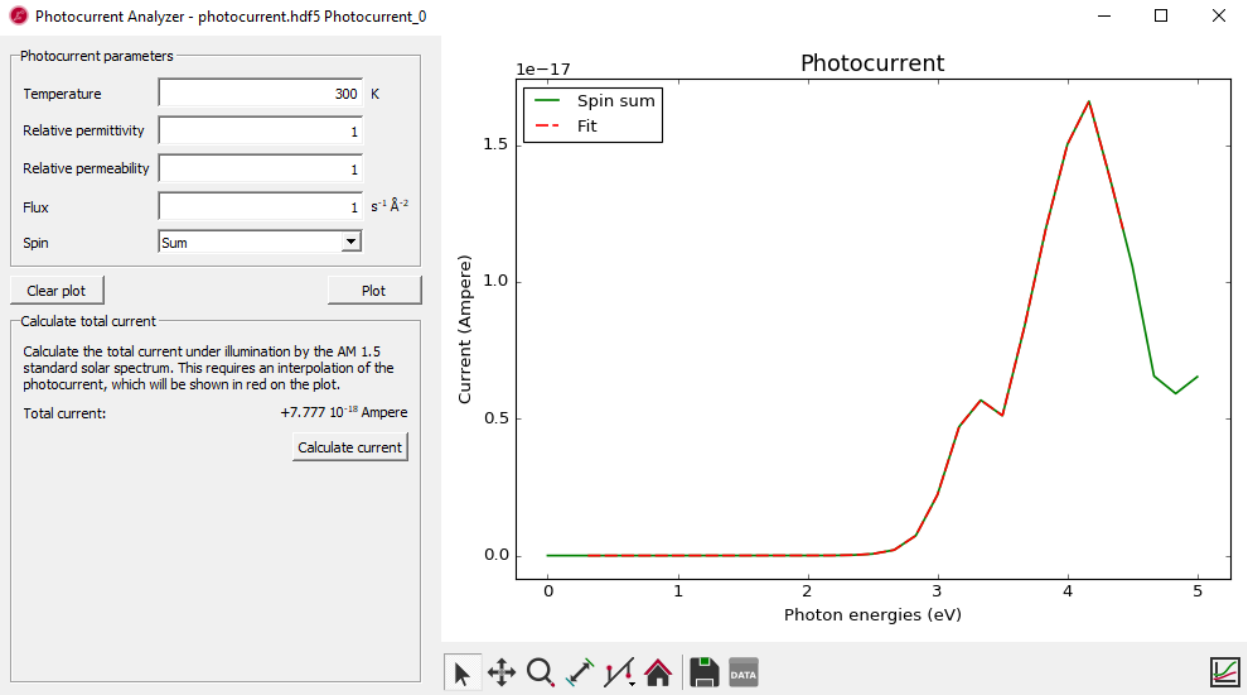
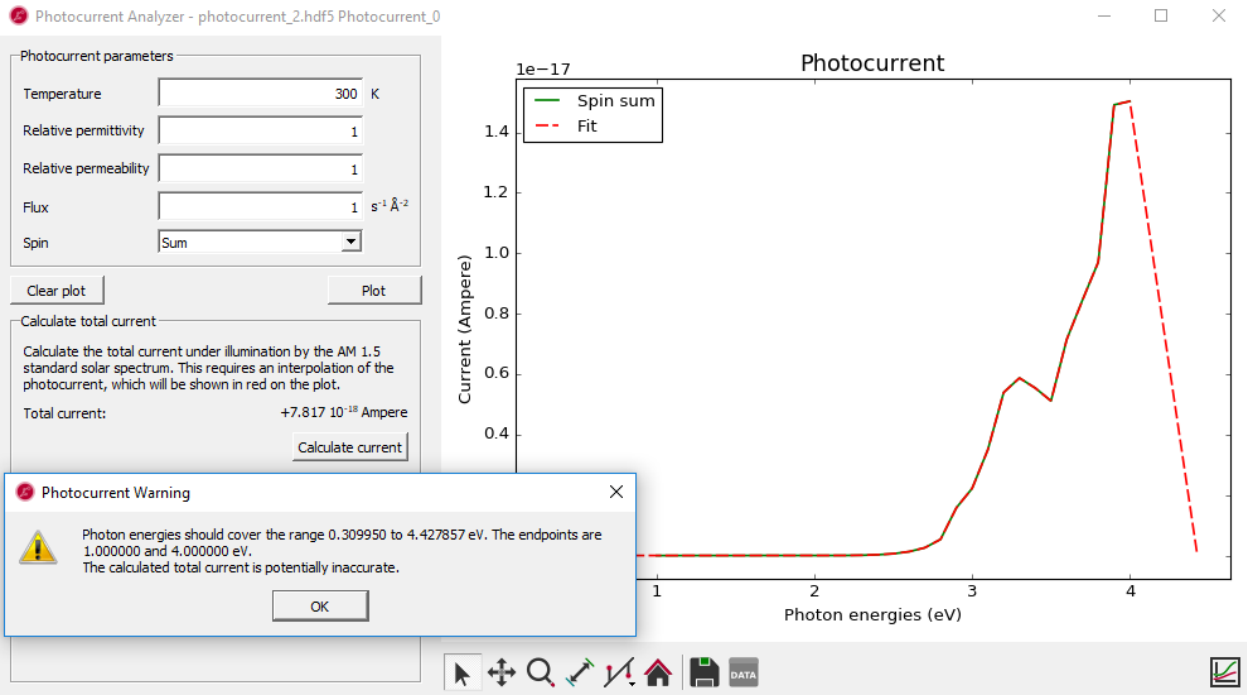


Fig. 3.72: The total current is calcuated as $7.777 \cdot 10^{-18}$ Ampere under the illumination by the AM 1.5 standard solar spectrum.



References

3.4 Complex Interfaces

3.4.1 Building an interface between Ag(100) and Au(111)

QuantumATK: Interface builder Platforms: *In this tutorial you will learn how to use the interface builder. You will build an interface between an Ag(100) and an Au(111) surface. It is assumed that you are familiar with the basic functionalities of QuantumATK.*

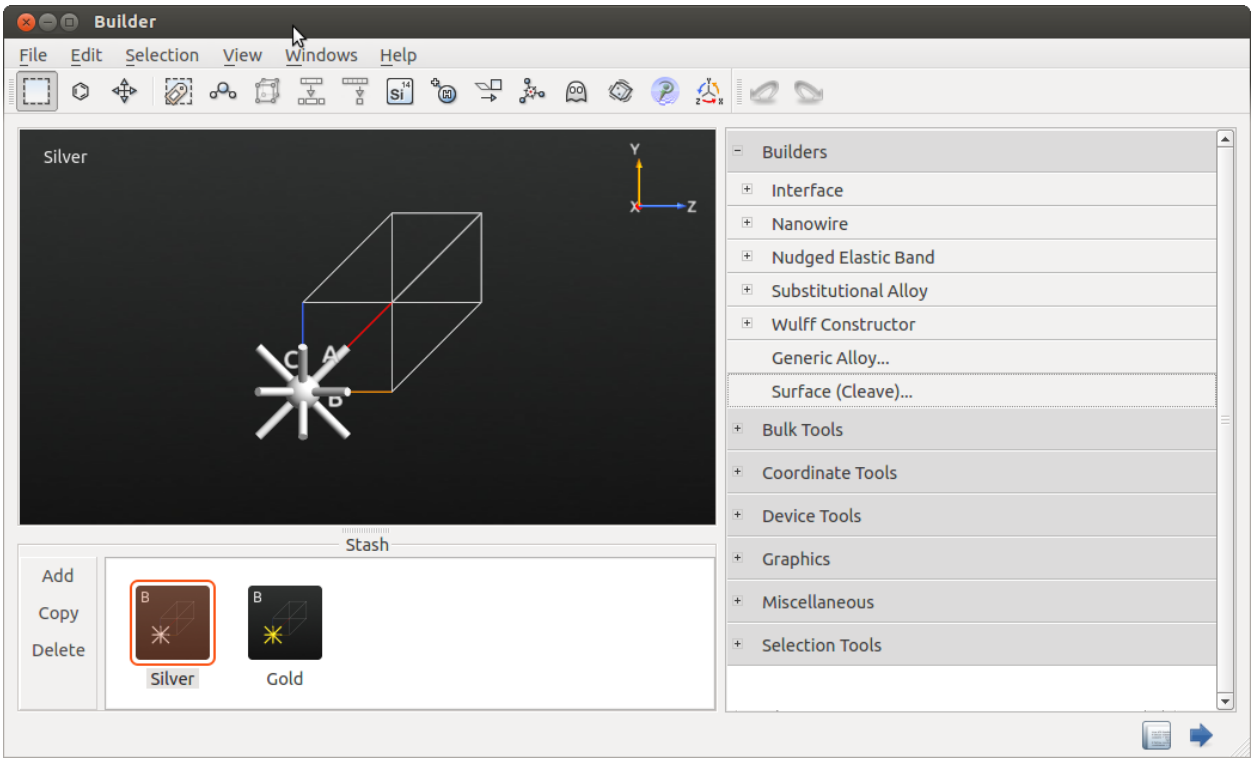
Import silver and gold crystals

Create a new empty project called Ag_Au_interface and open the Builder .

1.Use *Add* → *From database* to import a silver and a gold crystal to the Stash.

2.Double-clisk the Silver Stash item to activate it.

3.Expand the **Builders** group in the right-hand panel bar.



Building the Ag(100) and Au(111) crystals

1.Open the **Surface (Cleave)** tool.

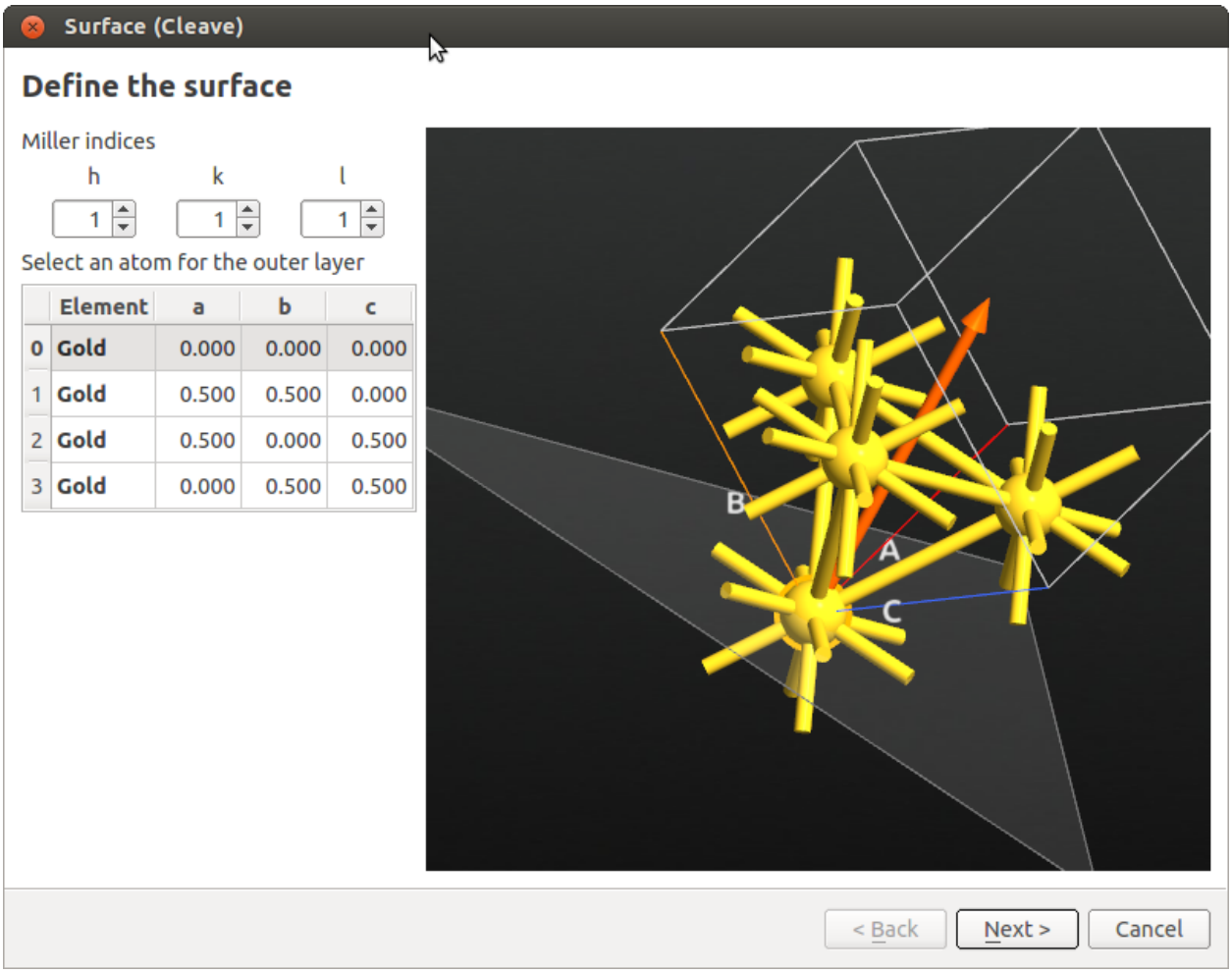
2.Use the default surface direction (100), and click Next.

3.Use the default minimal surface bravais lattice, and click Next.

4.Use the default configuration type, i.e. v3 (C axis) is along z and the structure is periodic.

5.Click **Finish**.

Now do the same for the Gold crystal, but cleave it along the (111) direction.

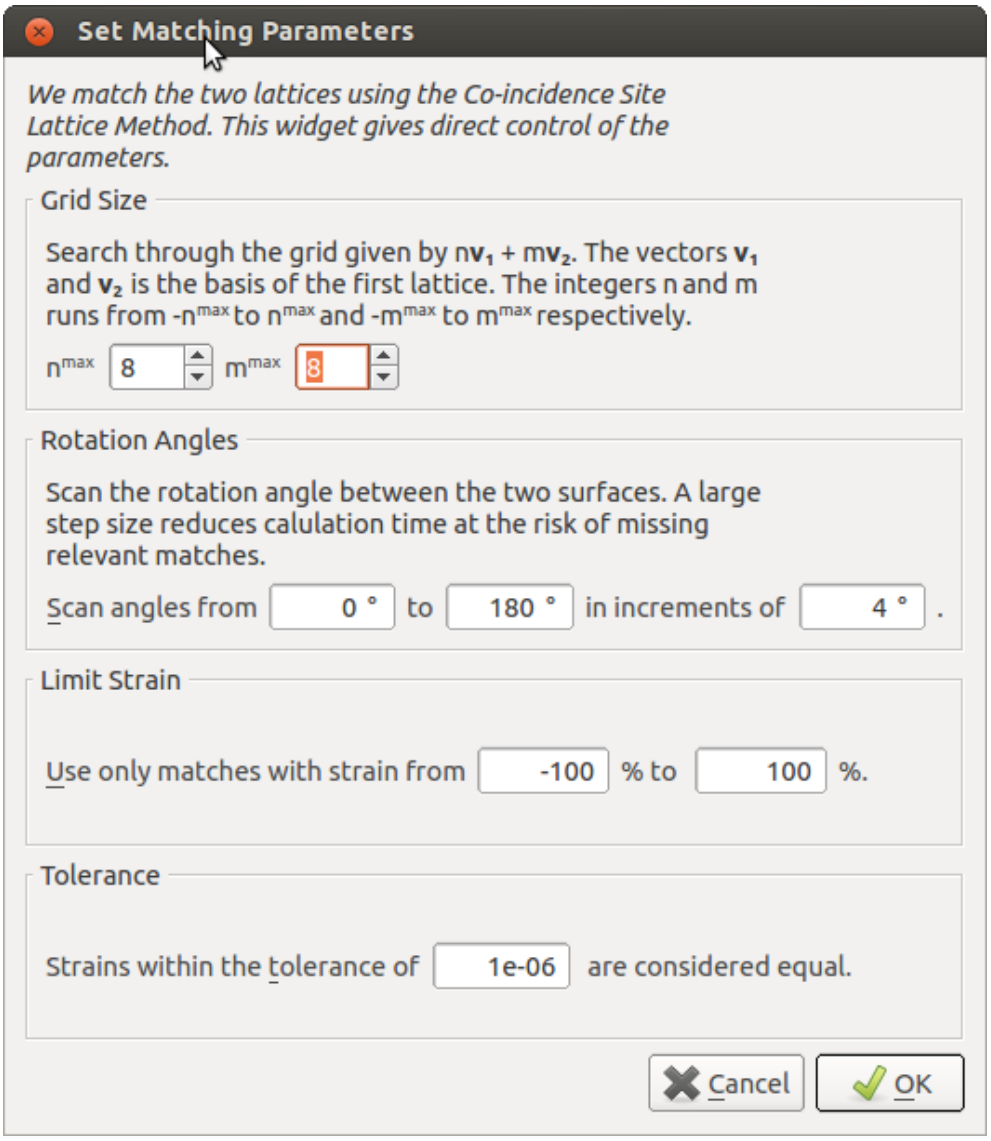


Building the interface

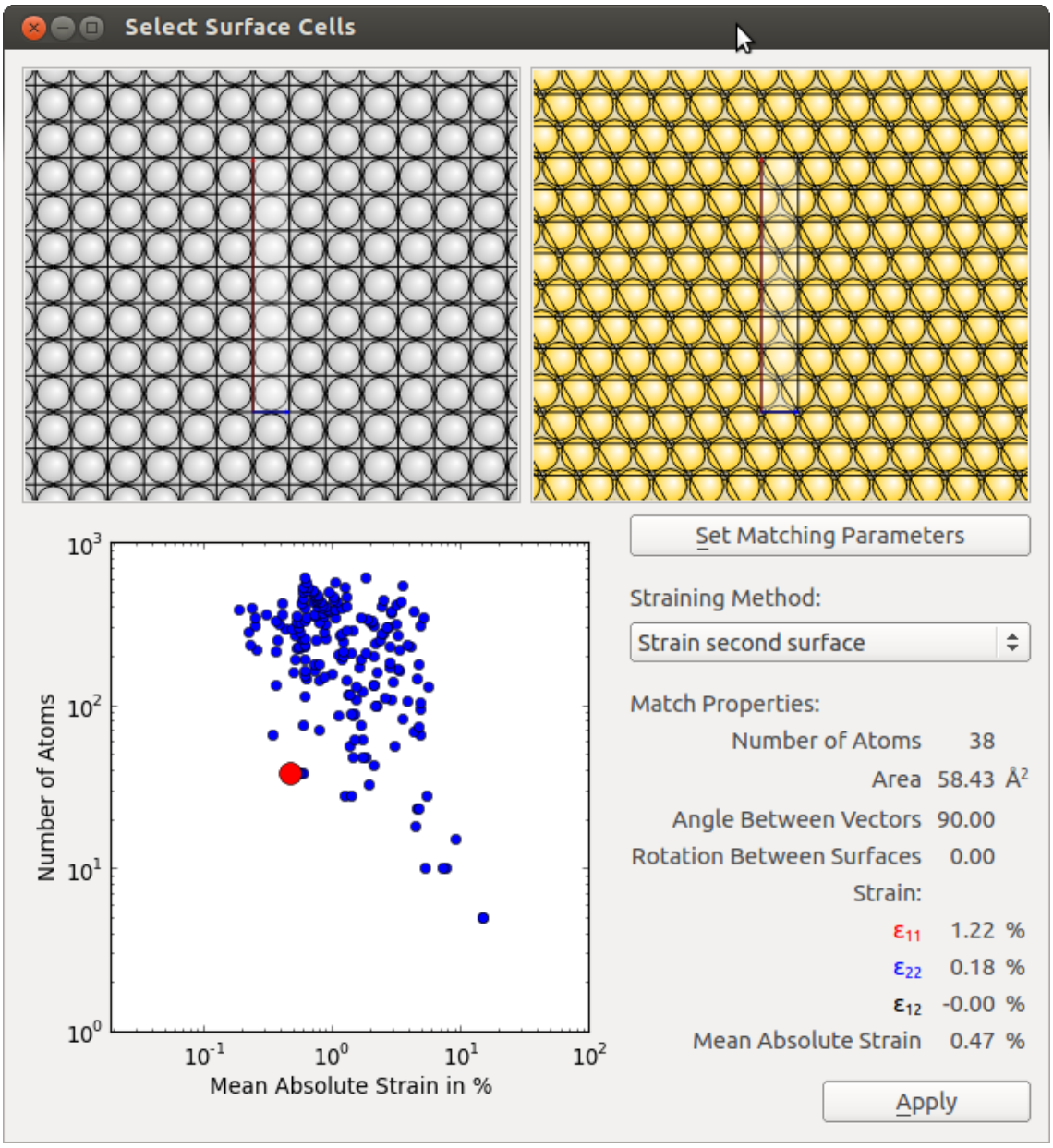
You have now built the Ag(100) and Au(111) supercells. The next step is to combine them into an interface. For this purpose, open the **Interface** plugin from the Builders group in the panel bar. It has two slots. Drop the Ag(100) crystal in the first and the Au(111) crystal in the second slot.

QuantumATK now scans all possible repetitions and rotations of the two surfaces in order to find a common supercell with minimal strain. Open the **Select Surface Cells** menu. The two upper plots show the selected surface cells of the two crystals. The lower left plot shows a number of blue dots, where each dot represents a common supercell for the two crystals.

We will now adjust the algorithm used for matching the two crystals. For this purpose, open the menu called **Set Matching Parameters**. With this menu it is possible to select the number of repetitions included for each cell and which angles are scanned to find a match. Increase n^{max} and m^{max} to 8 in order to search for larger supercells with less stress. Then click OK.



All the matching crystals will now be recalculated. Select the blue point closest to the lower left corner, to obtain a good compromise between a small supercell and a low stress. The supercell has 38 atoms and an average strain of 0.47%. Click Apply.

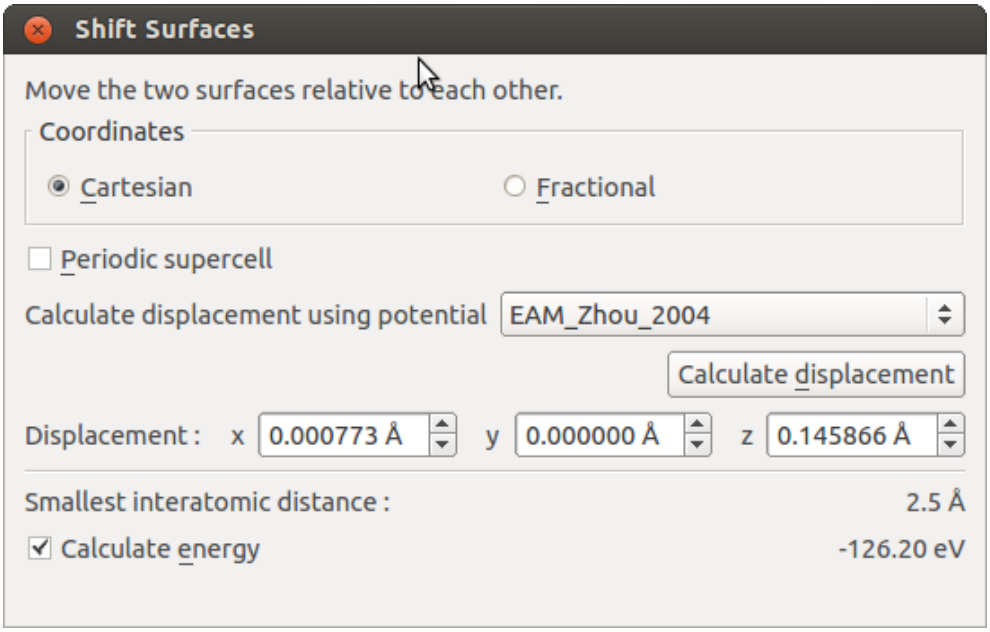


The next step is to define the relative position of the two crystals. For this purpose, open the **Shift Surfaces** menu in the main Interface Builder window. From this menu it is possible to either adjust the relative position of the two crystal manually, or compute the optimal distance using the ATK-ForceField calculator.

1.Select the EAM_Zhou_2004 potential.

2.Check the **Calculate energy** box.

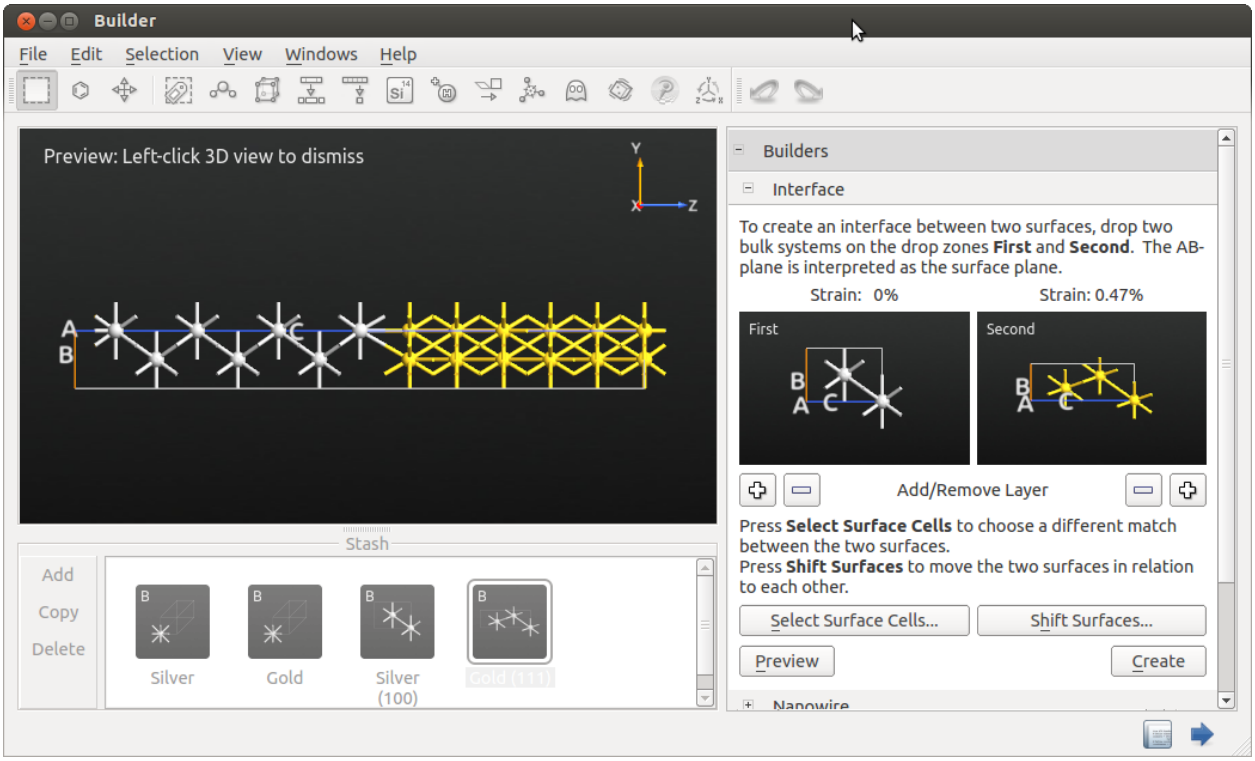
3.Click **Calculate displacement**.



Tip: In this case there are several available potentials. If you are not familiar with any of them, you may just try out different potentials and check if they give consistent results.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

Close the **Shift Surfaces** menu and return to the main Interface window. Now add 5 silver layers and 3 gold layers using the buttons. Check that there are now 7 layers to the left and 6 layers to the right of the interface (Tip: You can use **Ctrl+r** to resize the plot window). Click **Create** to generate the structure.

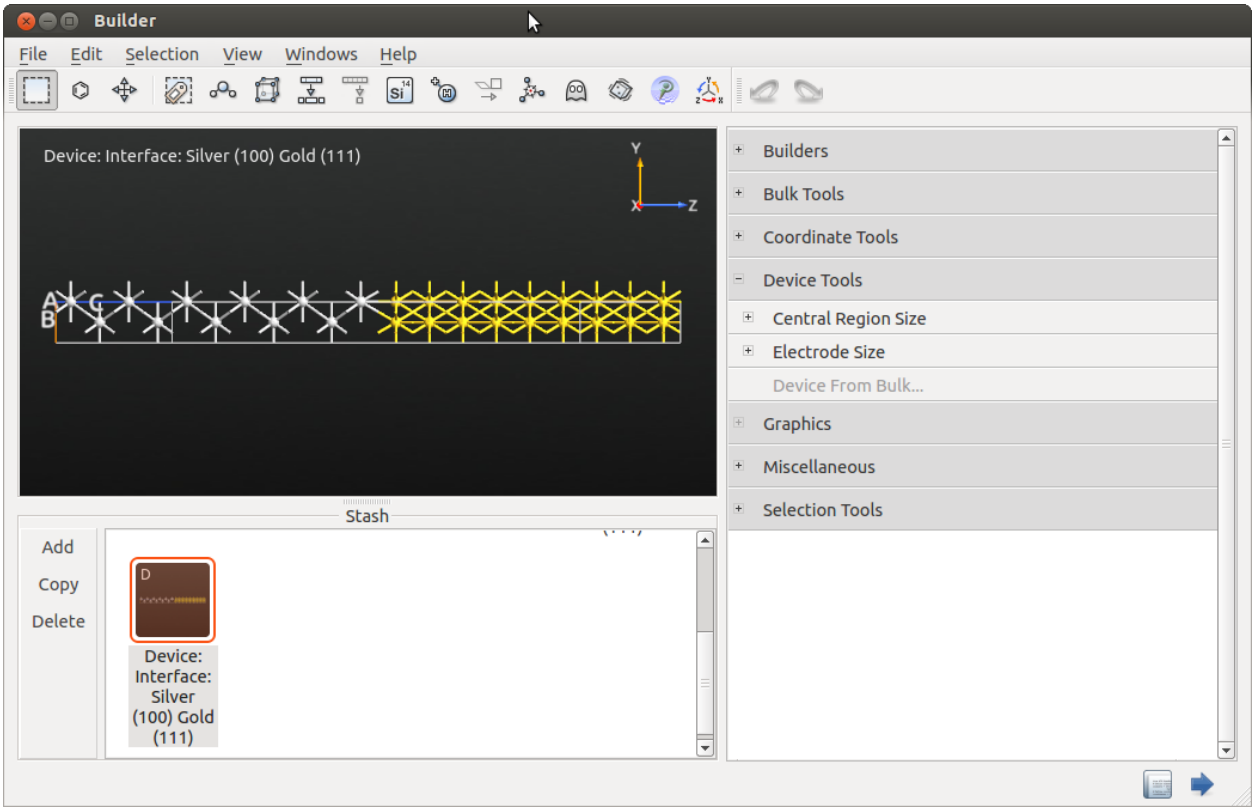


Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

Building the device configuration

The last step is to build a device configuration from the interface structure. Contract the Builders group and expand the **Device Tools** in the Builder panel bar. Select **Device from Bulk**. The tool will try to determine the electrode lengths by looking for repetition matches. Keep the default values and click **OK**. You have now finished the setup.

Save the resulting geometry in the file `Ag_Au_interface.py` by using the **File** menu or by right clicking the Stash item.



3.4.2 Advanced device relaxation - manual workflow

In this tutorial you will learn how to use a combination of Rigid constraints and a simple optimization routine to perform efficient and reliable relaxation of the internal coordinates in a device configuration.

Note: In the O-2018.6 release, we introduced a so-called *Study Object* to automate the workflows described in this tutorial. That is now the recommended way of performing device geometry optimizations in QuantumATK, and if you have access to O-2018.06 or newer, you should go to the corresponding tutorial *Relaxation of devices using the OptimizeDeviceConfiguration study object*. Users of previous versions can still perform the calculations using the more complex workflow, that is described here.

Introduction

Relaxation of the internal coordinates of device configurations can be a tedious and time consuming task. There are two main reasons for this:

The 2-probe configuration consists of three distinct regions: Two electrodes and a central region consisting of electrode extensions and a scattering region. The geometry of each region should be separately optimized in a first-principles manner, but both electrode extensions in the central region must always match the corresponding electrode.

Step 1: Optimize the electrodes. After optimization of the electrodes, it is important to make sure that the electrode extensions in the central region match the optimized electrodes.

Step 2: Optimize the central region. The optimum length of the central region is usually not known *a priori*. In the case of a periodic bulk, one would minimize the stress tensor. However, the stress field is not uniquely defined for a device configuration, so another approach is needed.

In the second step, we must rely on minimizing the device total energy while explicitly preserving the internal coordinates of the electrode extensions. There are at least two different ways to do this using QuantumATK.

BRR method

After building the 2-probe configuration from fully relaxed electrodes, you extract the central region bulk and relax it with Rigid constraints on the atoms belonging to the electrode extensions. The device or interface is then reassembled from the relaxed bulk. This is a computationally efficient approach and will usually capture a large fraction of the required geometry relaxation. We shall refer to this method as Bulk Rigid Relaxation (BRR).

The BBR method does not calculate the total energy using a full device geometry, so the reassembled device may not be exactly in the global minimum-energy geometry. Some types of electronic structure calculations may be sensitive to this difference, others may not.

1D minimization

A somewhat more brute-force approach is to explicitly minimize the device total energy wrt. internal coordinates and the central region length, i.e., do the full 2-probe calculations and vary the central region length. This is a 1D minimization problem with relaxation of the atomic positions at each step, so we shall refer to this method as 1DMIN. We have developed a procedure for doing this as efficiently as possible with QuantumATK, but the computations are in general more demanding than with BRR. It is therefore recommended to use 1DMIN only after pre-relaxing the central region with the BRR method.

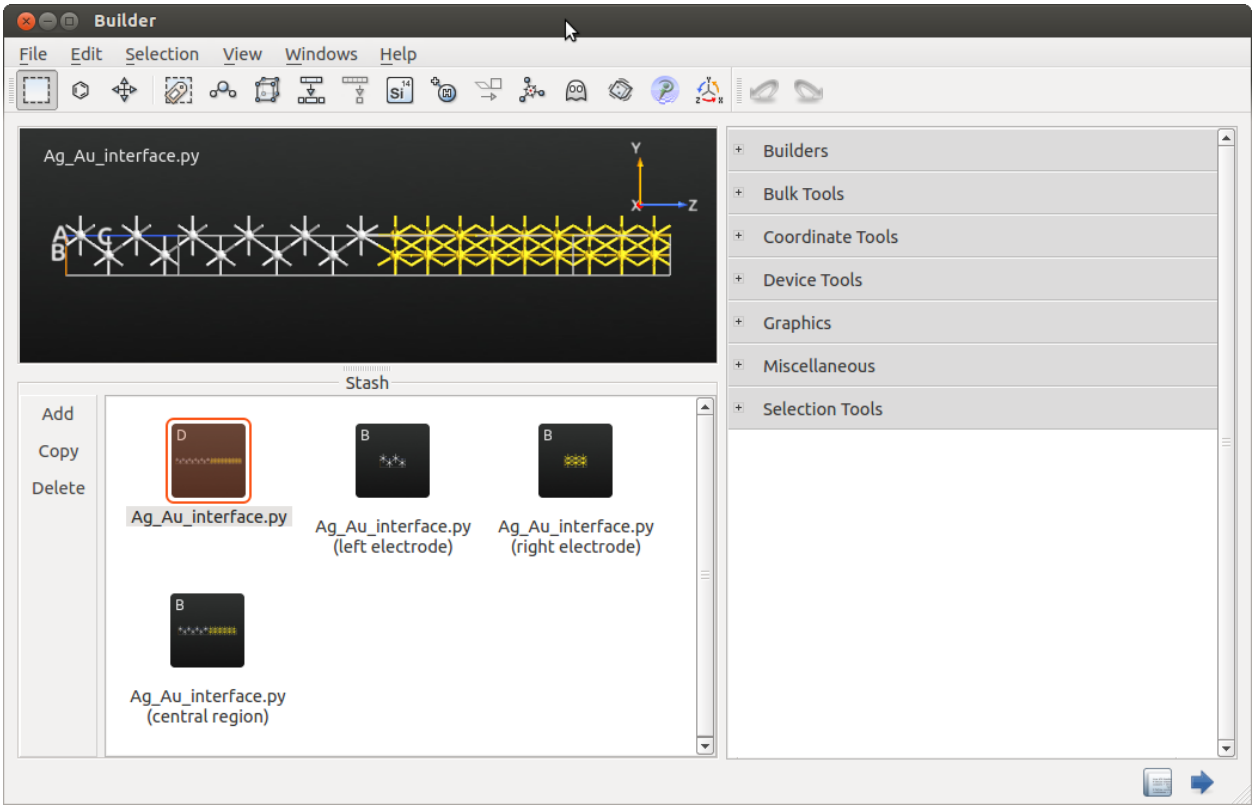
This tutorial introduces the electrode relaxation and the central region relaxation using BRR and 1DMIN methods. The methods are illustrated for relaxing a silver-gold interface using a classical potential. The classical potential is selected for computational convenience, and the same procedure could be used for another relaxation method like DFT. In general we recommend to check ATK-ForceField calculations with DFT calculations.

Preparations

First make a new project called device_relaxation. Our starting point is an unrelaxed 2-probe device configuration. It was created using the QuantumATK Interface Builder follow the tutorial at this link: *Building an interface between Ag(100) and Au(111)*. The geometry can also be downloaded as this file: `Ag_Au_interface.py`.

Save the py-file in your project folder, locate the device configuration on the LabFloor, and drag-and-drop the configuration to the **Builder**.

The 2-probe configuration should look like the one illustrated below. It consists of a left electrode, Ag(100), the central region, and a right electrode Au(111). Use the plugin to split the device into a left electrode, right electrode and central region.



Electrode relaxation

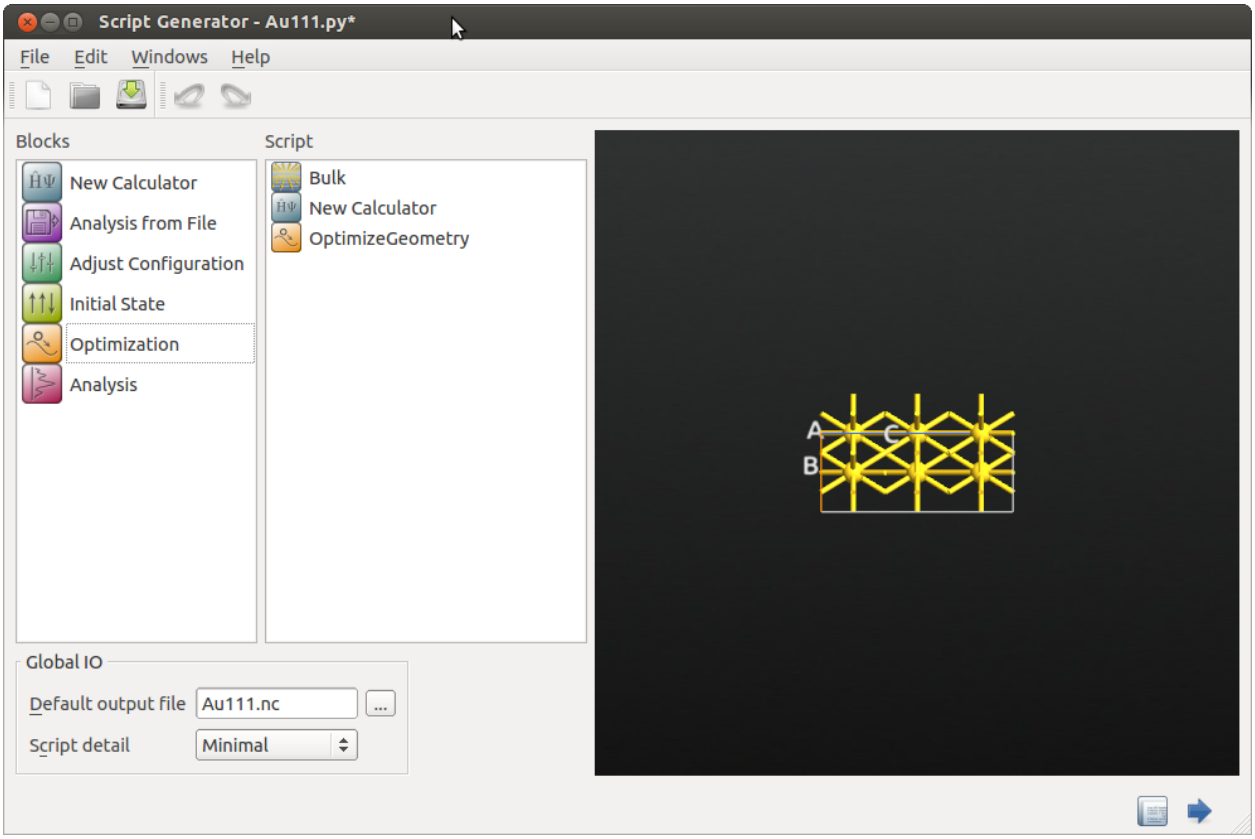
The device geometry was constructed using the interface builder (*Building an interface between Ag(100) and Au(111)*). Prior to building the interface the left and right crystal structures were set up using their experimental geometries. We will assume that the experimental geometry is also the lowest energy structure of the theoretical method we will use for relaxing the device geometry.

In the interface builder we choose Strain second surface upon selecting the surface cell. This means that the Ag(100) surface was kept in its experimental geometry, while the Au(111) surface was strained in the x,y directions upon forming the interface. When the crystal is strained in the x,y directions, it will relax in the z-direction according to the Poisson ratio of the material.

In the following we will apply this relaxation to the Au(111) crystal, by relaxing the stress in the z-direction while keeping the x, y directions fixed. The obtained relaxation of Au(111) will be transferred to the gold atoms in the central region.

Open the builder and change the name of the right electrode to Au111. Double click the geometry to make it active and send it to the **Scripter**. Add the following objects:

- New Calculator
- OptimizeGeometry

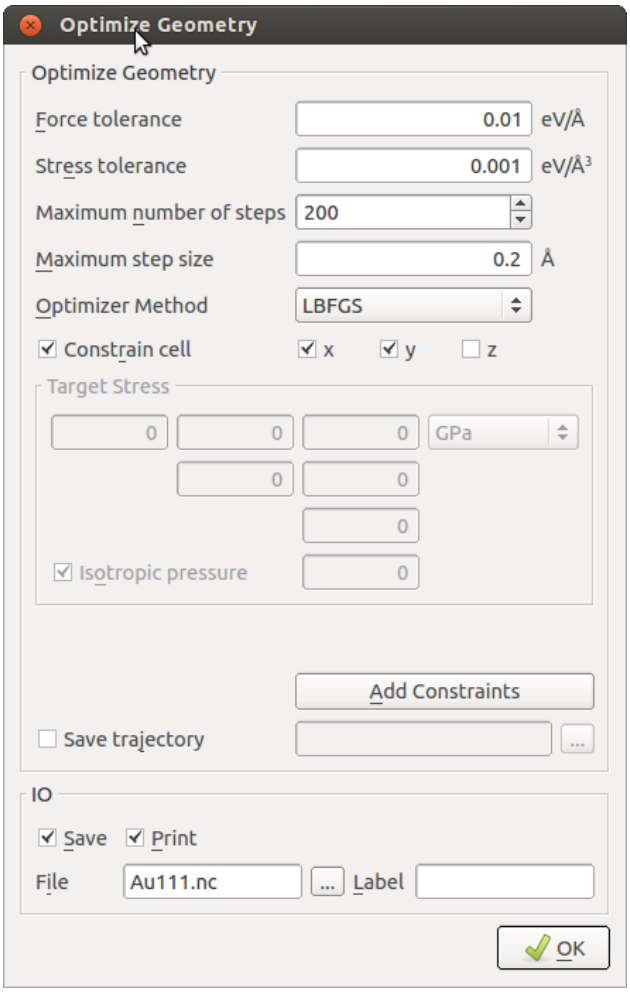


Now open the New Calculator object and select the **ATK-ForceField** EAM_Zhou_2004 potential. Click OK.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

Next open the Optimize Geometry object:

1. Set the force tolerance to 0.01
2. Set the stress tolerance to 0.001
3. Uncheck z in the Constrain cell checkboxes

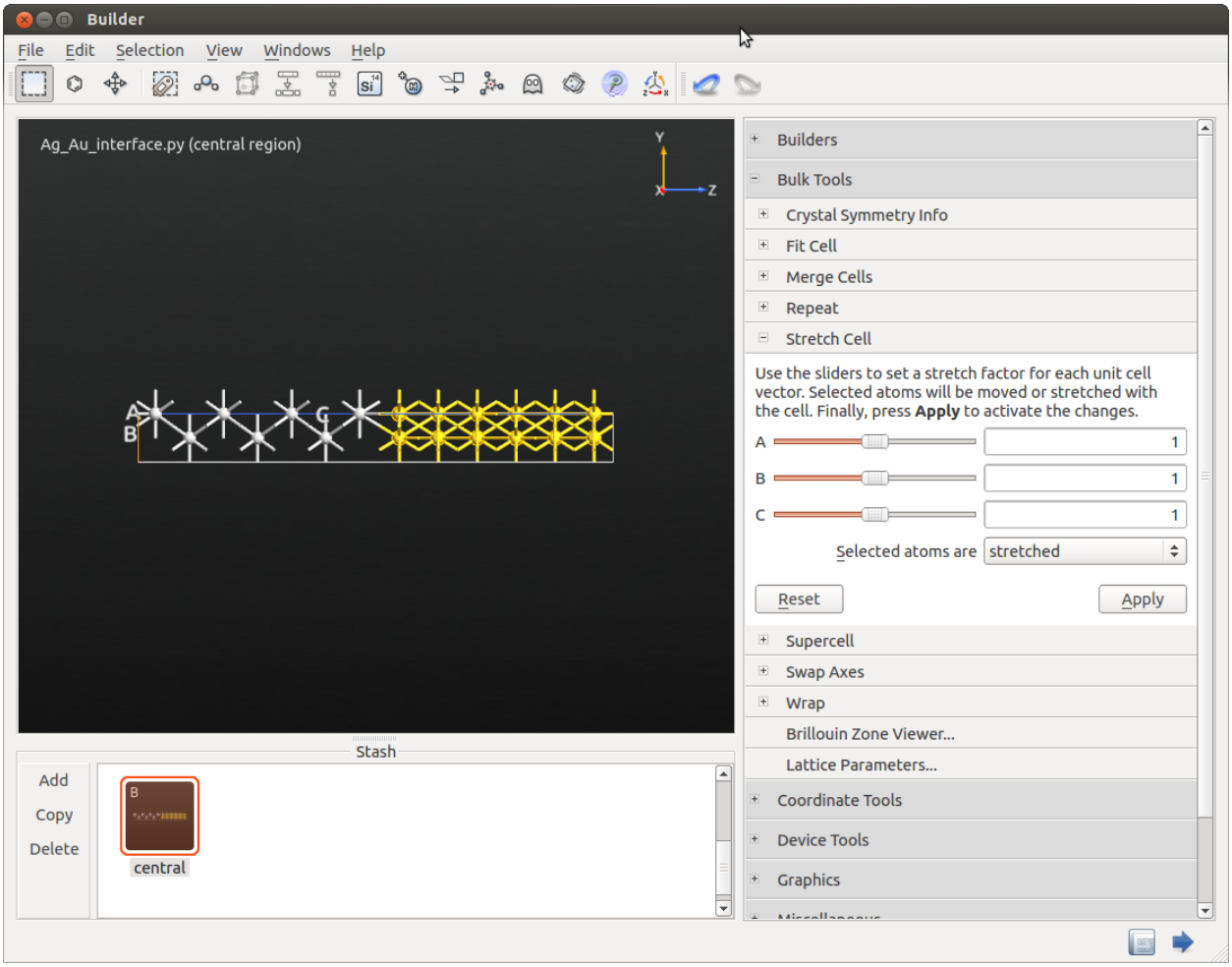


Send the script to the Job Manager and run the relaxation. It should take less than a minute to execute. When finished inspect the log file.

You should now see a new object on the **LabFloor** called Au111.hdf5. It contains two configurations, corresponding to the geometry before and after the relaxation. Select the two geometries and drop them onto the **Viewer**. In the viewer you can hoover the mouse over the unit cell of each configuration and you will notice that the length of the C vector is 7.064 Å before the relaxation and 7.011 after the relaxation. This information could also be obtained from the log file, Au111.log.

The relaxation results in a small compressive strain of 0.75%. To apply this to the central region, open the Builder and double click the central region configuration.

1. Rename the stash item to central.
2. Select all the gold atoms by drawing a rectangle around them.
3. Open the *Bulk Tools* → *Stretch Cell* plugin.
4. Enter 0.9925 along the C axis.
5. Use Selected atoms are *stretched*.
6. Click Apply.



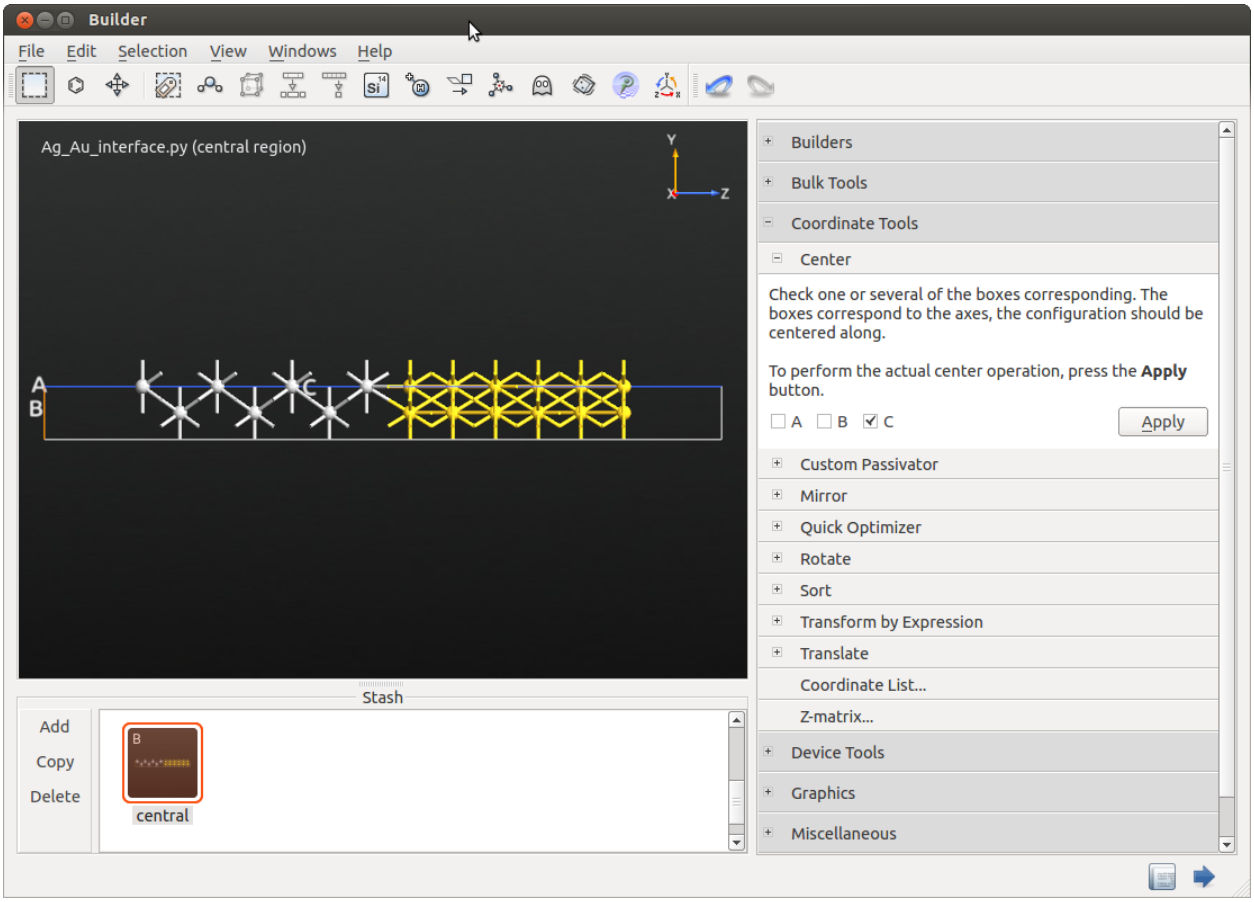
Note: The strain is so tiny that it is not possible to see any visual change in the structure. However, you may open the Lattice Parameters.. plugin and click the and buttons to see how the C cell vector changes when you undo and redo the stretch. Similarly, you can inspect the Coordinate List.

Central region relaxation

You are now going to relax the central region only. We want the electrode extensions to be able to move freely (but rigidly) during geometry relaxation. We therefore add vacuum on both ends:

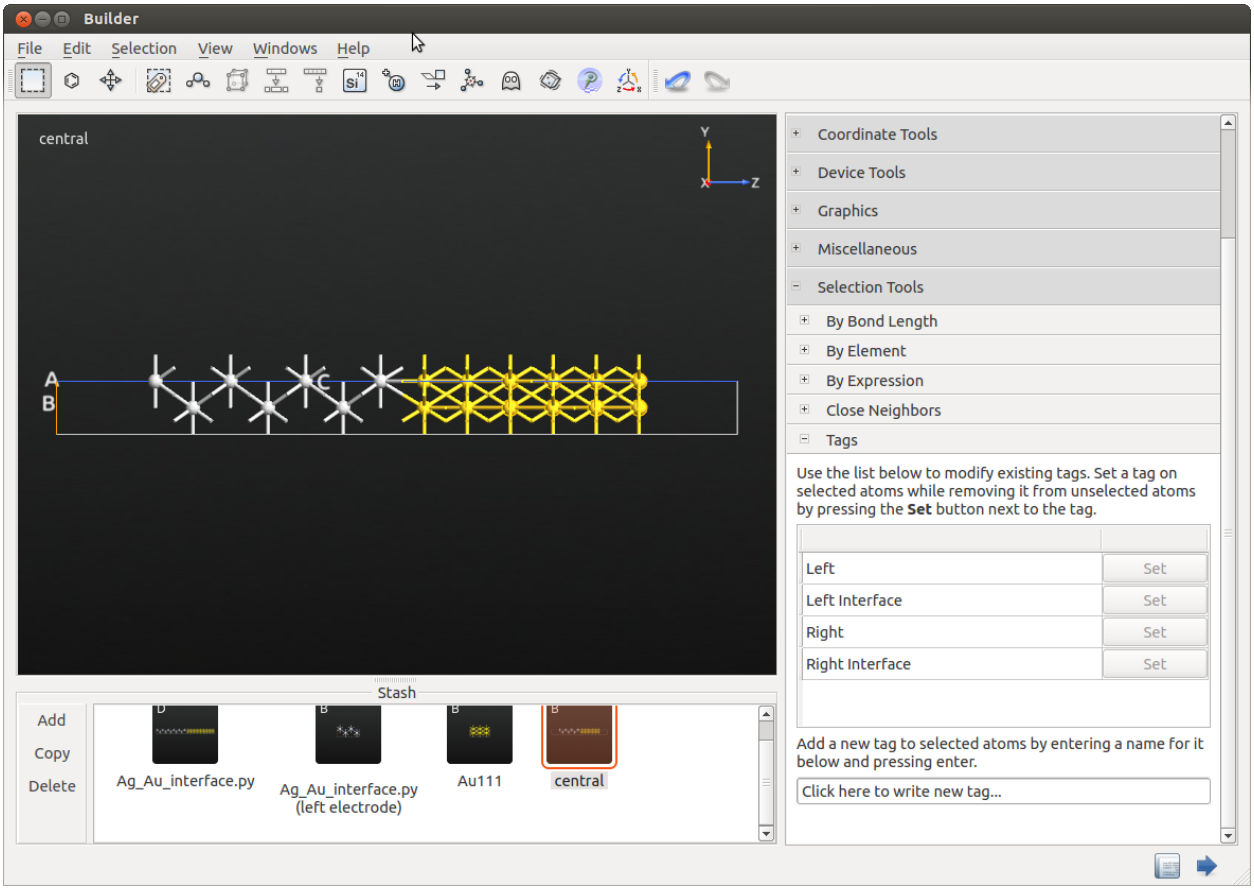
Make sure that no atoms are selected, by clicking the background of the plotting window. In *Bulk Tools* → *Stretch Cell* use the C-vector slider to increase the cell length by 30% or simply type in a multiplication factor of 1.3. The cell is stretched once you click Apply.

Move to *Coordinate Tools* → *Center* and uncheck A and B. Center the central region by clicking Apply.



We want to fix the atoms in the electrode extensions by including an additional layer of the central region. The additional layer will help the *Device from Bulk* tool in QuantumATK to identify the length of the electrode. For this purpose select the 5 outermost silver atoms. Open *Selection Tools* → *Tags*. Enter **Left** in the box named Click here to write a new tag. Press Return.

Next select the 4 outermost gold atoms, and tag them **Right**.



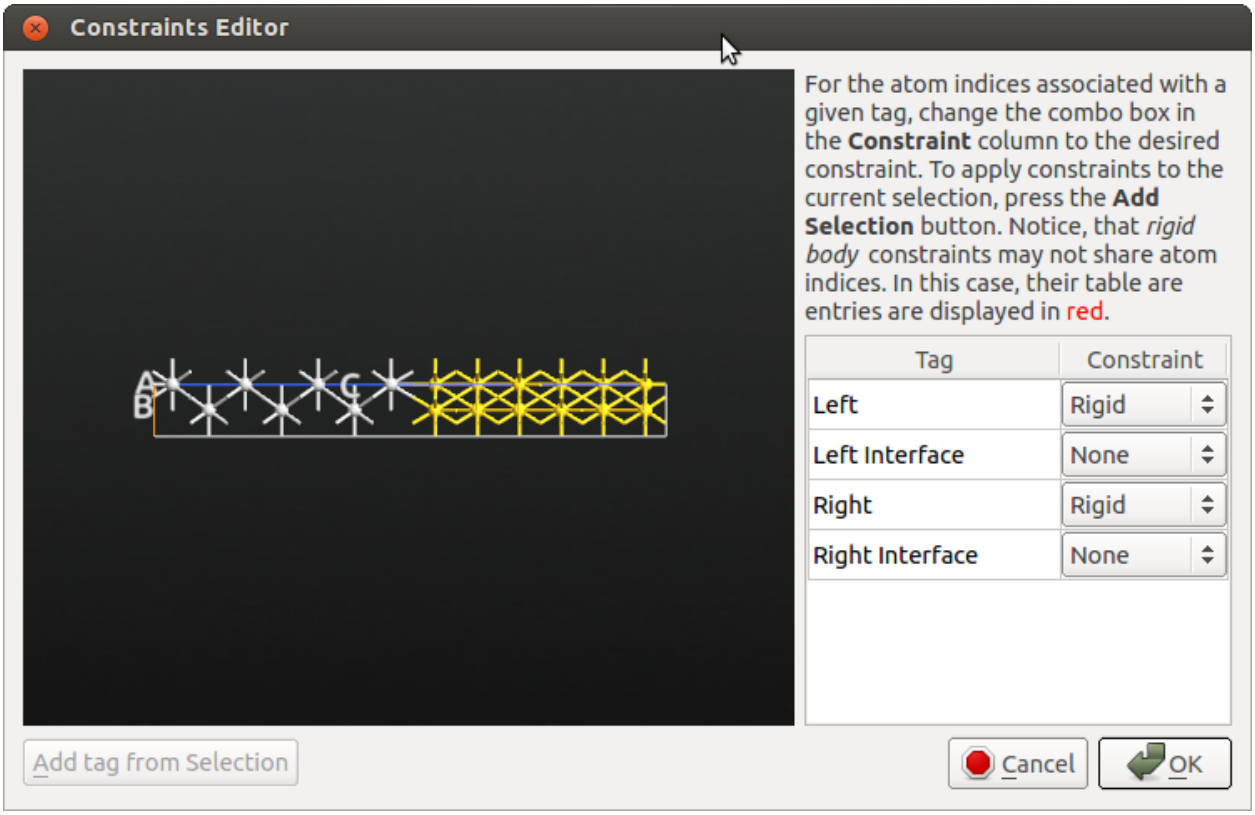
Send the configuration to the **Script Generator** and set up the calculation:

Add a New Calculator and an OptimizeGeometry object.

Choose the ATK-ForceField engine and the EAM_Zhou_2004 potential for these calculations.

Make the following changes to the OptimizeGeometry object:

1. Set the force tolerance to 0.01 eV/Å.
2. Check Save trajectory and set the filename to `central.hdf5`.
3. Click Add Constraints to set up the Rigid constraints on the electrode extensions.
4. Check that the Left and Right tags are assigned to the correct atoms in the configuration. Do this by clicking the tags. Choose Rigid constraints for both tags.
5. Click OK and return to the Scripter.



Note: Choosing instead a Fixed constraint for one of the tagged regions while keeping the other one Rigid, may save BFGS relaxation steps and thereby computational time.

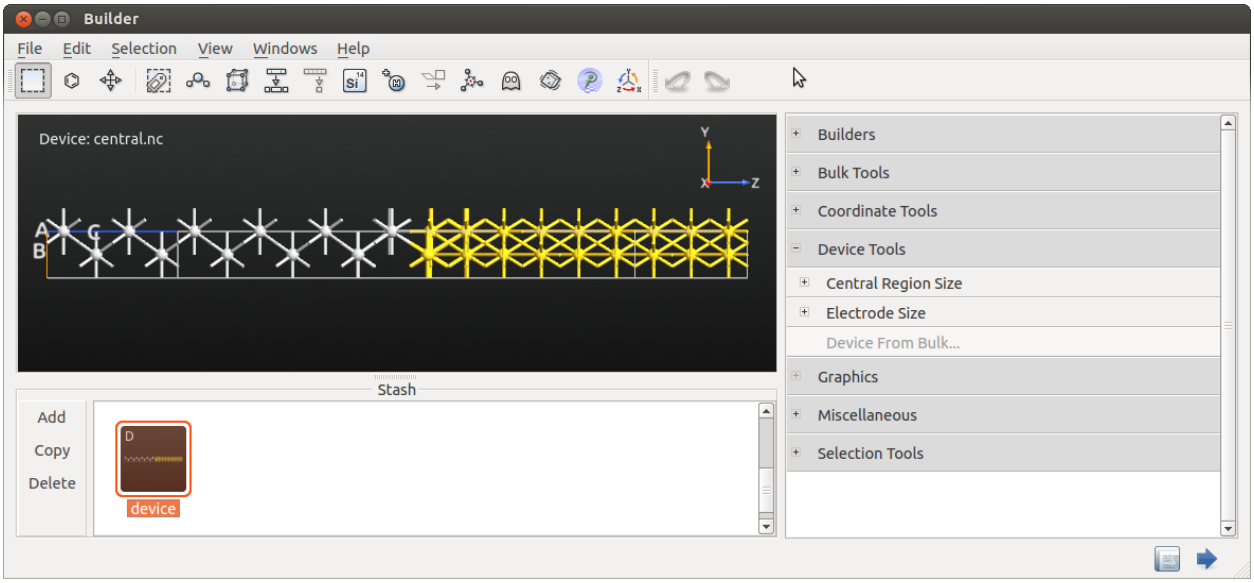
Send the script to the **Job Manager** and run the calculation. It should be very fast. Once finished, the `central.hdf5` output file appears on the **LabFloor**.

You can use the **Viewer** plugin to see the trajectory of the relaxation as illustrated in the animation below.

The relaxed central region is saved in `central.hdf5` as the bulk configuration with id glD002. This is your initial guess for the central region geometry. Drag-and-dro it on to the **Builder**.

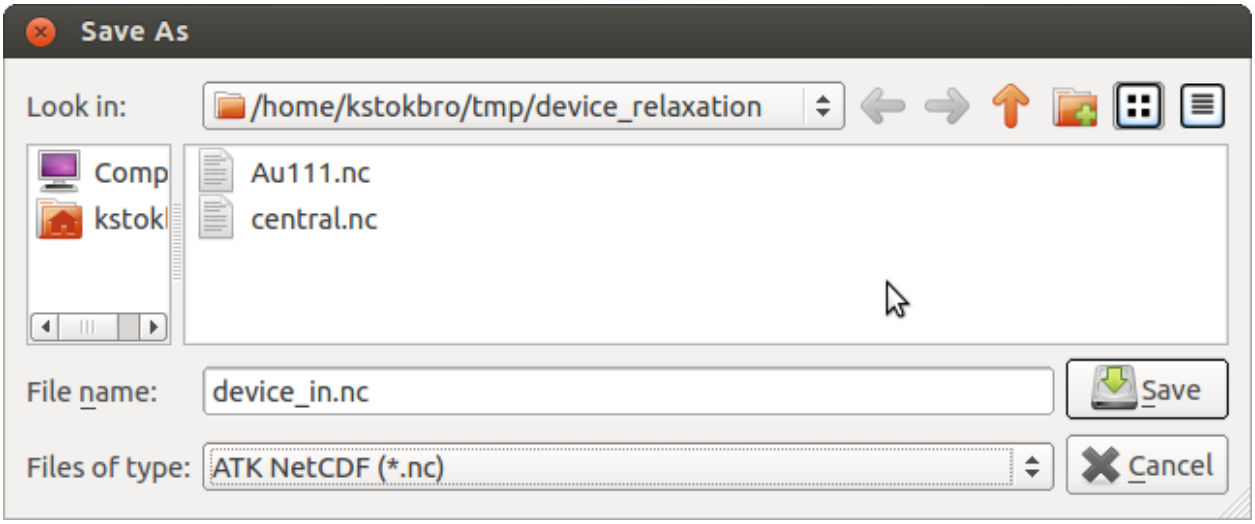
Click *Device Tools* → *Device From Bulk* to turn the `central.hdf5` bulk into a device configuration. You can leave the settings for electrode and central region lengths at default values.

The interface configuration should now look like illustrated below. Rename the configuration to device by using F2 or with a right mouse-click.



1DMIN optimization of the interface using 2-probe calculations

We are now ready to search for the global minimum-energy geometry of the Ag(100)-Au(111) interface. Save the device configuration as a HDF5 file named `device_in.hdf5` by choosing Save as after right-clicking the device Stash item.



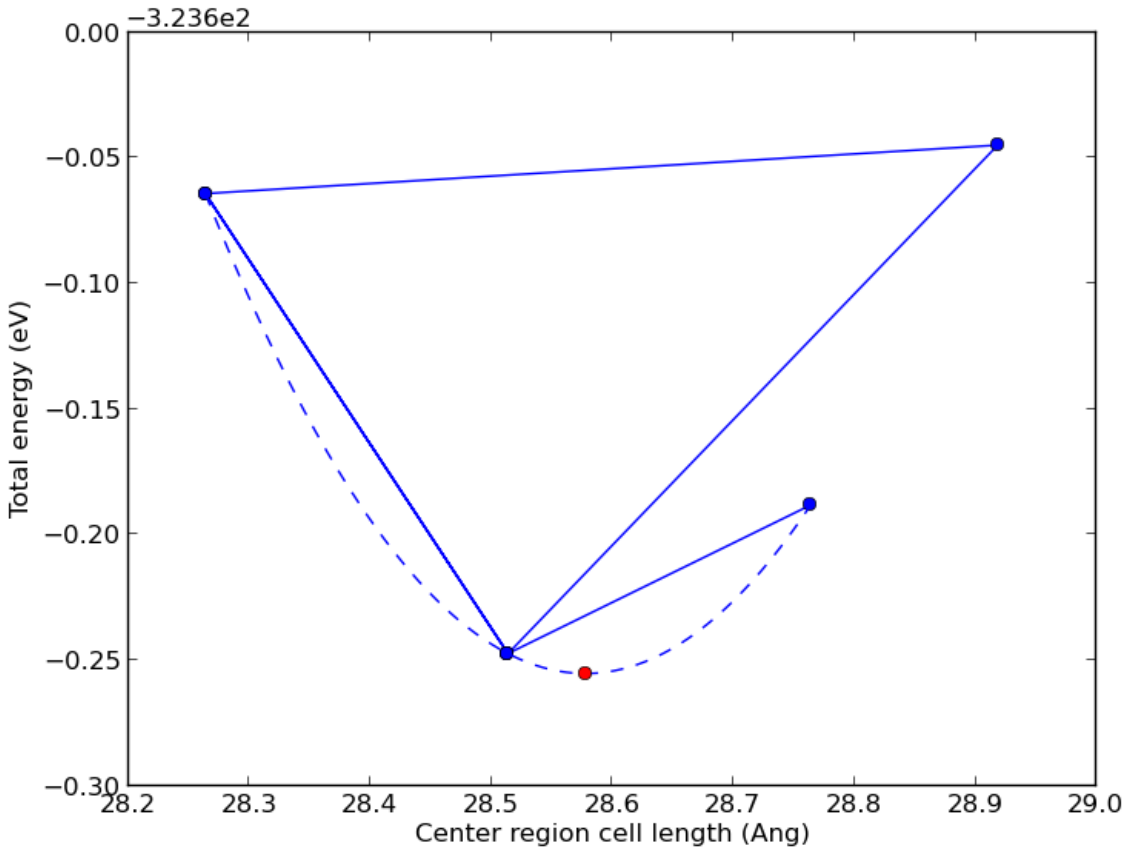
You are now going to apply a simple **SciPy minimization** procedure in search for the central region length that minimizes the interface total energy. The device configuration you just saved in `device_in.hdf5` is the starting point for this.

Two scripts are needed. You can download them here: `device.py` and `optimize.py`. Save both scripts in the folder where `device_in.hdf5` was saved. The first script (`device.py`) reads in the device configuration and sets up and runs the calculation, while the second script (`optimize.py`) contains functions and classes used by `device.py`.

Note: To use the script device.py for other systems you need to edit the script and change the definition of the potential to an appropriate potential for your new system.

Both scripts should now be available in the QuantumATK Project Files list. Drag-and-drop device.py onto **Jobs**. The Job Manager will open. Click to run the script.

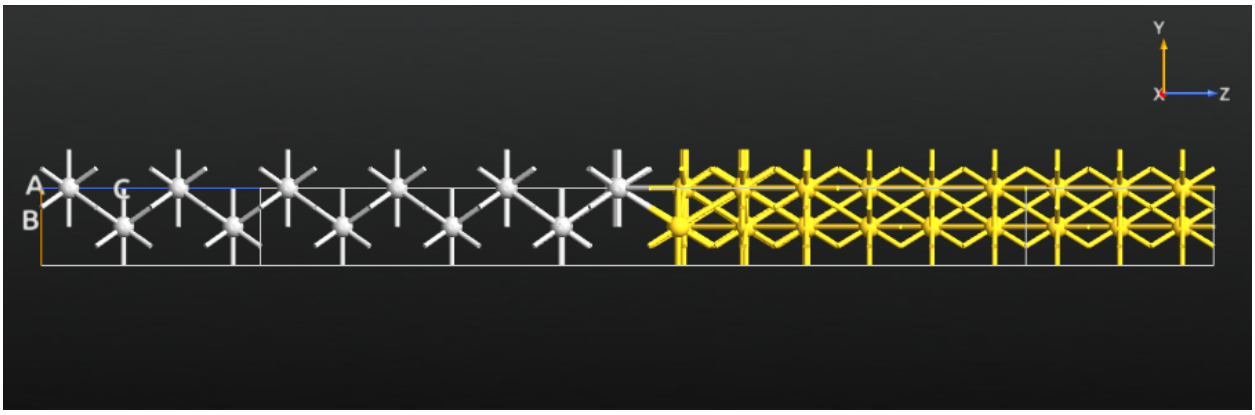
Once finished (its fast!), a window with the figure below will pop up. The blue points are total energies of relaxed interface configurations. The initial cell length is 28.51 Å and the other blue points are trial guesses made by the algorithm. The red point indicates the optimum central region length which is 28.58 Å. The optimization routine simply maps out the illustrated potential-energy surface and finds its minimum!



The job will finish when you close the pop-up window.

The relaxed configurations are saved in the output file `device_out.hdf5` along with the total energies. The configuration corresponding to the red point (the final device or interface geometry) is the latest device configuration (the one with largest id number).

Using the **Viewer** plugin, the minimum-energy interface configuration looks like this:



In this case the change to the structure after the 1DMIN relaxation is minute compared to the BRR relaxation. Often the BRR will be sufficiently accurate and there is no need for the 1DMIN step. The 1DMIN approach is only needed for extreme accuracy.

3.4.3 Relaxation of devices using the OptimizeDeviceConfiguration study object

Version: O-2018.06

This tutorial introduces you to the `OptimizeDeviceConfiguration` study object. For proper optimization of the geometry of a 2-probe device configuration, we recommend the Bulk Rigid Relaxation (BRR) method. However, this can be a complicated task involving several steps. As shown in this tutorial, the `OptimizeDeviceConfiguration` study object automates the BRR method, making it very simple to optimize the geometry of a device.

Note: The **OptimizeDeviceConfiguration** study object was introduced in the O-2018.06 release, and is the recommended way of performing device geometry optimizations in QuantumATK. Users of previous versions can still perform the calculations in a more complex, and manual, workflow, as described here: *Advanced device relaxation - manual workflow*

Introduction

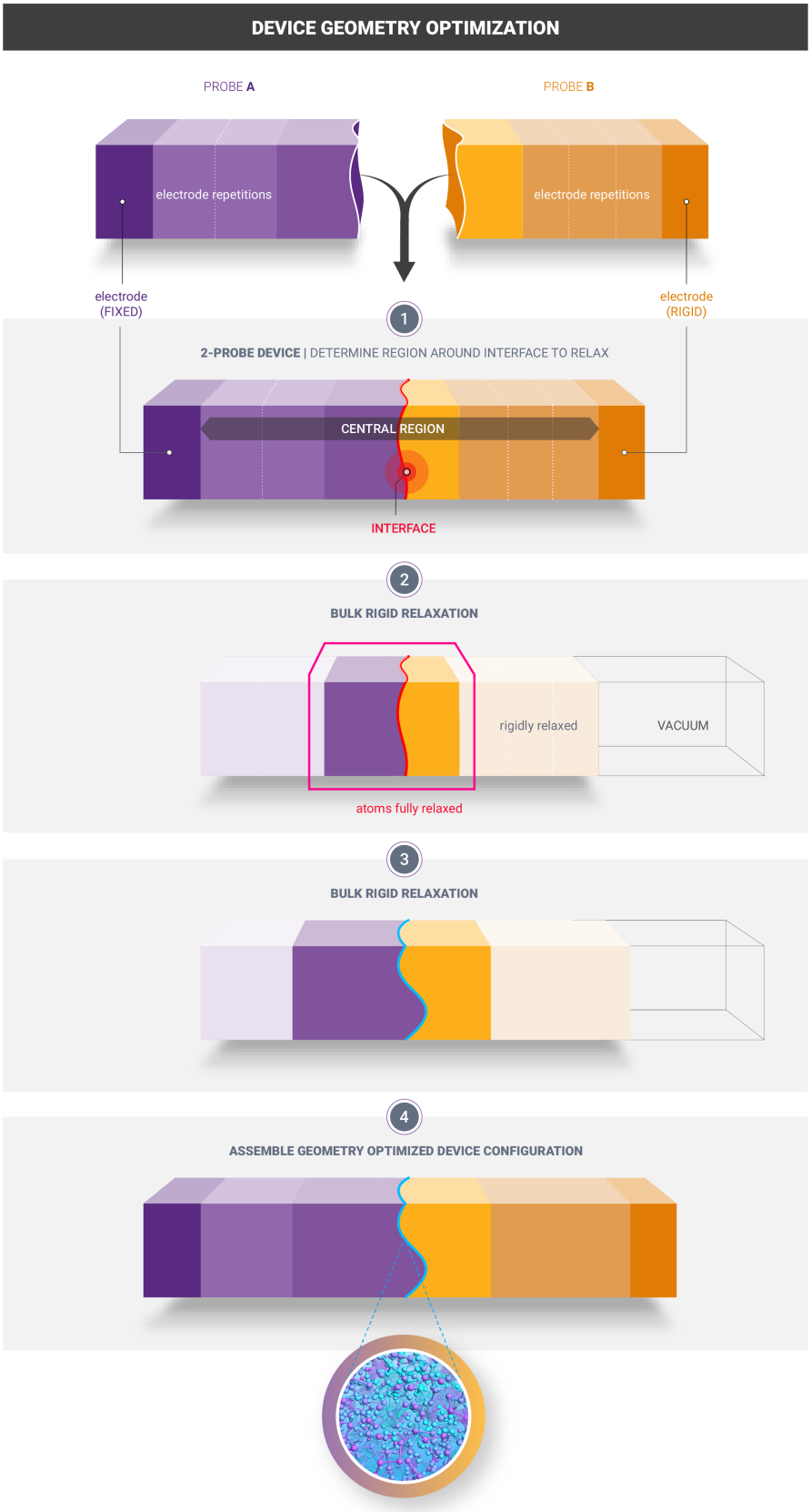
In principle, the complex device structural relaxation can be done manually, as described in the *Advanced device relaxation - manual workflow* tutorial, but the manual BRR procedure requires tedious and time-consuming work. Therefore, we encourage you to use the `OptimizeDeviceConfiguration` study object for relaxing complex device systems. The study object allows for fully automated device relaxation, including automatic identification of the central region atoms that should be allowed to relax. This fully automated procedure is illustrated below, and can be described as follows:

1. **The region of full relaxation is determined.** This happens automatically by first determining which parts of the central region are periodic repetitions of the near-by electrode. The middle point between left and right periodic images is then the center of the region that should be fully optimized, and the region has a user-specified width. Note that the region of full relaxation is not allowed to extend into an electrode extension. If it does, the edge of the region is automatically moved to the border of that electrode extension.

2+3. **Bulk Rigid Relaxation is executed.** The device configuration is converted to a bulk configuration (the electrodes are removed), and the right-hand end of the cell is extended with a vacuum region, in order to allow for contraction

or expansion of the central region during relaxation. Ordinary geometry optimization (force minimization) is then executed while applying a Fixed constraint on the left-hand atoms and a Rigid constraint on the right-hand atoms. See the tutorial *Advanced device relaxation - manual workflow* for more details on the BRR method.

4. **The device configuration is re-assembled.** The vacuum region is removed and the electrodes are attached, resulting in the final geometry optimized device configuration.


DEVICE GEOMETRY OPTIMIZATION
PROBE A
PROBE B
electrode repetitions
electrode repetitions
electrode (FIXED)
electrode (RIGID)
1
2-PROBE DEVICE | DETERMINE REGION AROUND INTERFACE TO RELAX
CENTRAL REGION
INTERFACE
2
BULK RIGID RELAXATION
rigidly relaxed
VACUUM
atoms fully relaxed
3
BULK RIGID RELAXATION
4
ASSEMBLE GEOMETRY OPTIMIZED DEVICE CONFIGURATION

The study object allows you to change certain settings for device relaxation:

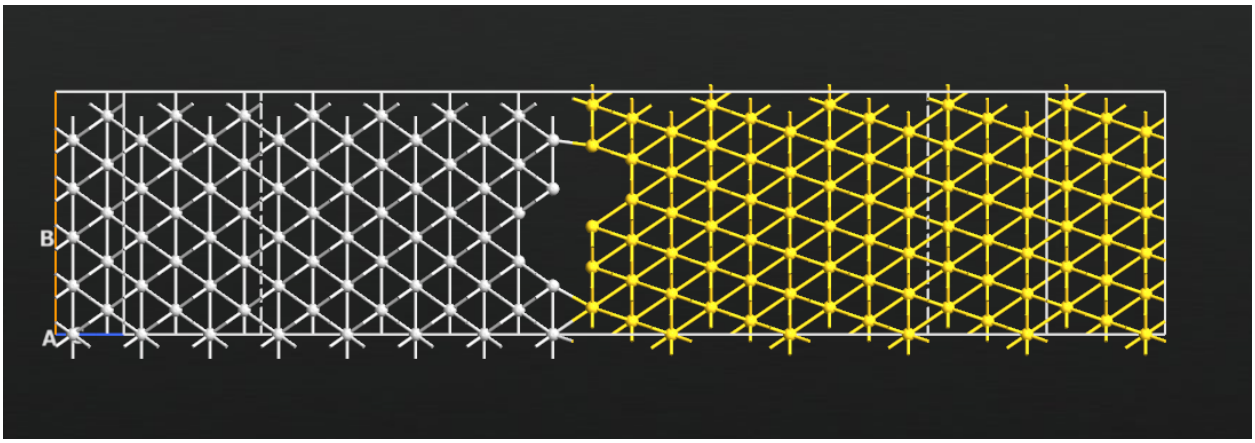
- You may select which part of the central region to fully relax, i.e., you may specify the center and width of the relaxation region.
- The relaxation region may extend to the entire device, except of course the electrodes and their extensions.
- If needed, the left and right surfaces of the central region may be hydrogen-passiovated during steps 2 and 3.
- The usual force convergence criterion for gemoetry optimization may also be tuned.

We will here demonstrate how geometry optimization of a Ag(100)||Au(111) interface can be done using the **OptimizeDeviceConfiguration** study object. For purposes of illustration, a large defect is introduced at the Ag|Au interface in order to enhance the effect of geometry optimization.

Unrelaxed Ag(100)|Ag(111) device

The unrelaxed and defected Ag(100)|Ag(111) device is available as a script: `unrelaxed.py`. Please download it and use it for this tutorial.

The device was built using the **Interface Builder**, as shown in the tutorial *Building an interface between Ag(100) and Au(111)*. The defect at the Ag|Au interface was introduced by simply removing atoms using the **Builder**.



Important: In general, devices should be built from relaxed (unstrained) materials, i.e., the electrodes should be in equilibrium geometries (given the level of theory employed, for example DFT using the PBE exchange-correlation functional, or a particular force field).

This may be crucial, as the theoretically predicted electronic properties of a material will in general depend on the geometry. In particular, the electrode relaxation may have an effect on the band gap and effective masses of the electrode material, which may change IV characteristics of the device. An instructive example of relaxation for strained silicon electrodes is given in *Appendix*.

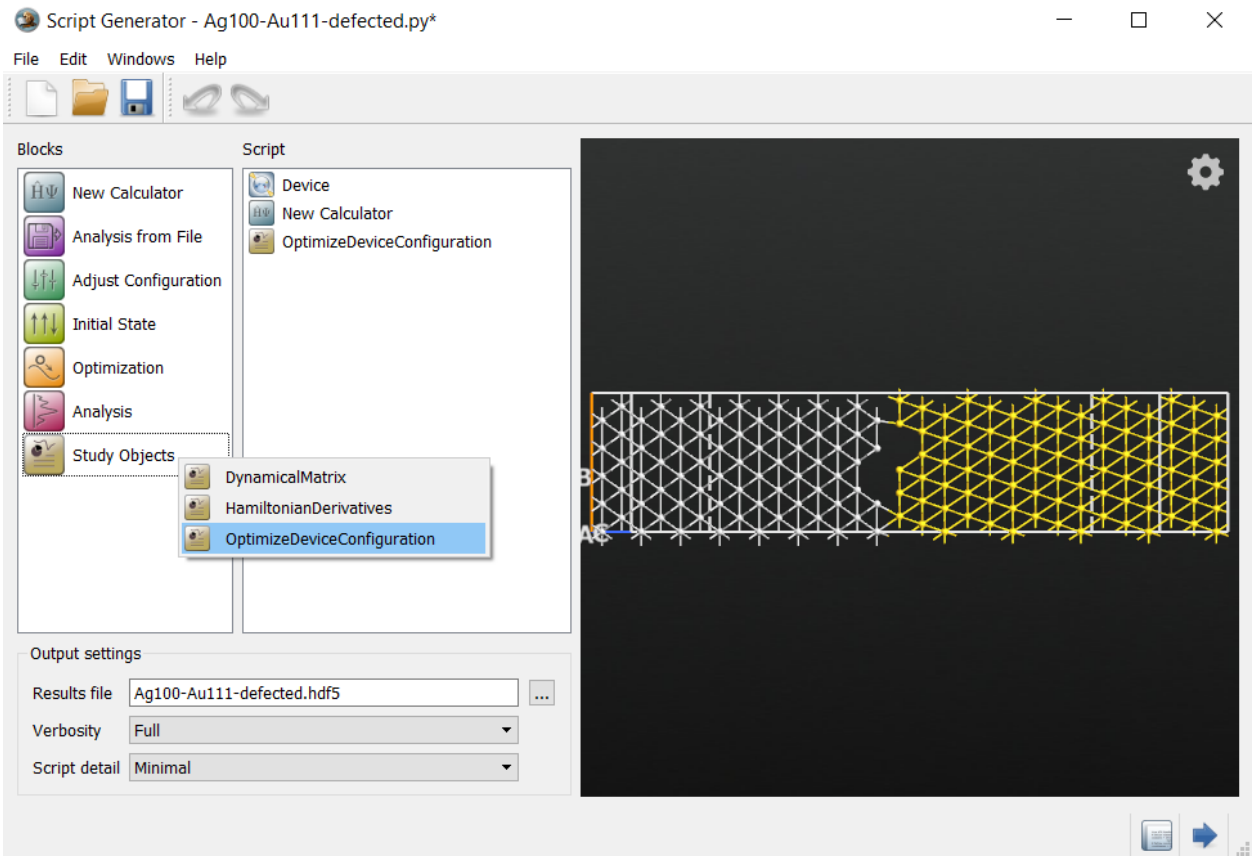
The *OptimizeDeviceConfiguration* study object **does not** relax the electrodes – it is the responsibility of you as a user to make sure the electrode geometries are appropriate before running the device geometry optimization.

Set up and run the device geometry optimization

Send the script `unrelaxed.py` to the **Script Generator**.

You will here relax the device using the ATK-ForceField engine, in order to speed up calculations. In the Script Generator, add the **New Calculator** block, and double-click to open it. Then select the **ATK-ForceField** calculator and choose the **EAM_Zhou_2004** force field.

Next, add the **OptimizeDeviceConfiguration** study object script block, by clicking *Study Objects* → *OptimizeDeviceConfiguration*.

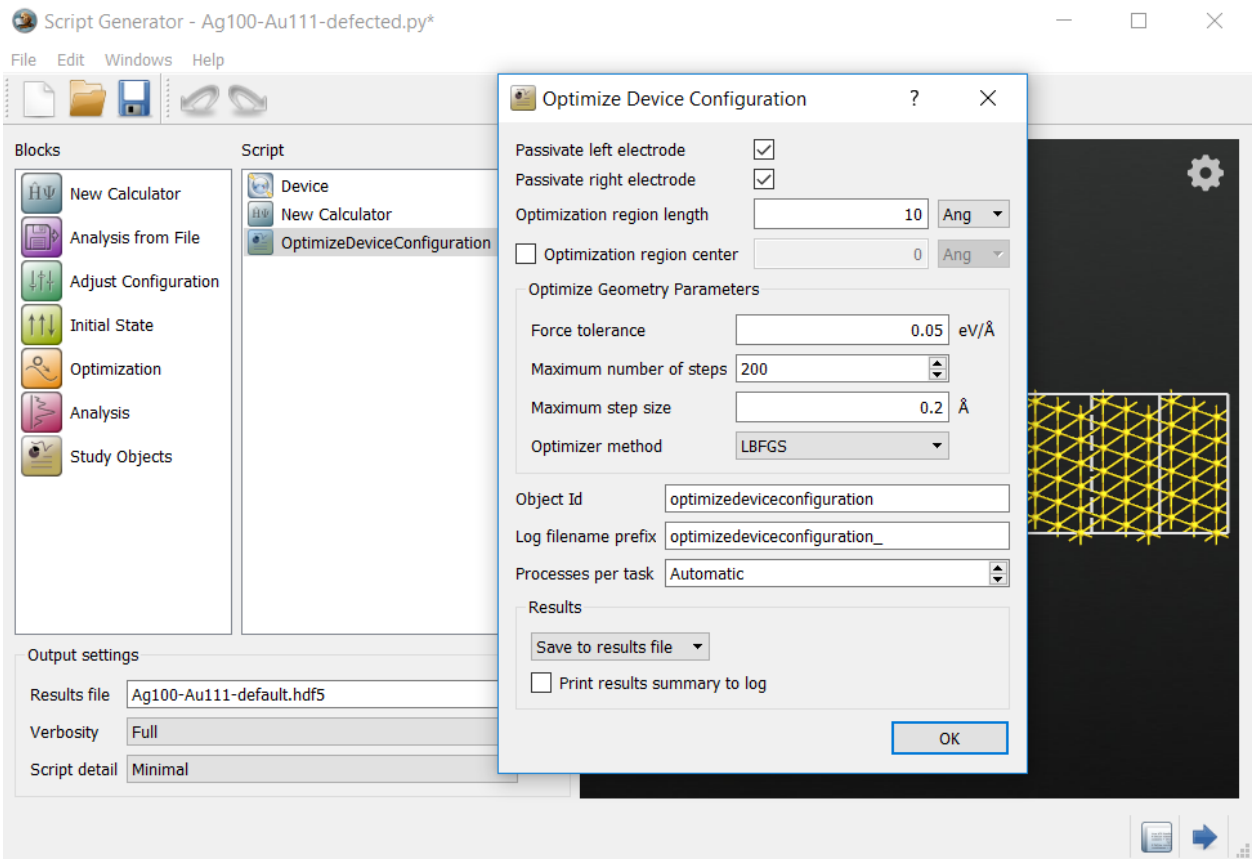


We will run the device geometry optimization using both A) default settings, and B) a custom optimization region.

A) Default settings

Double-click the **OptimizeDeviceConfiguration** block to open it. In principle, none of the settings need to be changed. As you see below, the default width of the optimization region is 10 Å, and the option *Optimization region center* is unticked, so the center is automatically determined, as described above.

Leave the settings at default values, close the widget, and change the name of the output results file to `Ag100-Au111-default.hdf5`, and save the Python script as `Ag100-Au111-default.py`. You may also download the final script: `Ag100-Au111-default.py`.



Note: The options *Passivate left electrode* and *Passivate rigth electrode* are ticked by default, meaning that the `OptimizeDeviceConfiguration` study object will attempt to apply **hydrogen passivation** to the left and right surfaces during the central region bulk relaxation (steps 2+3 in the methodology outlined in the *Introduction*).

This can be of relevance for semiconducting materials, but has no impact on metallic systems such as the Ag(100)|Au(111) interface.

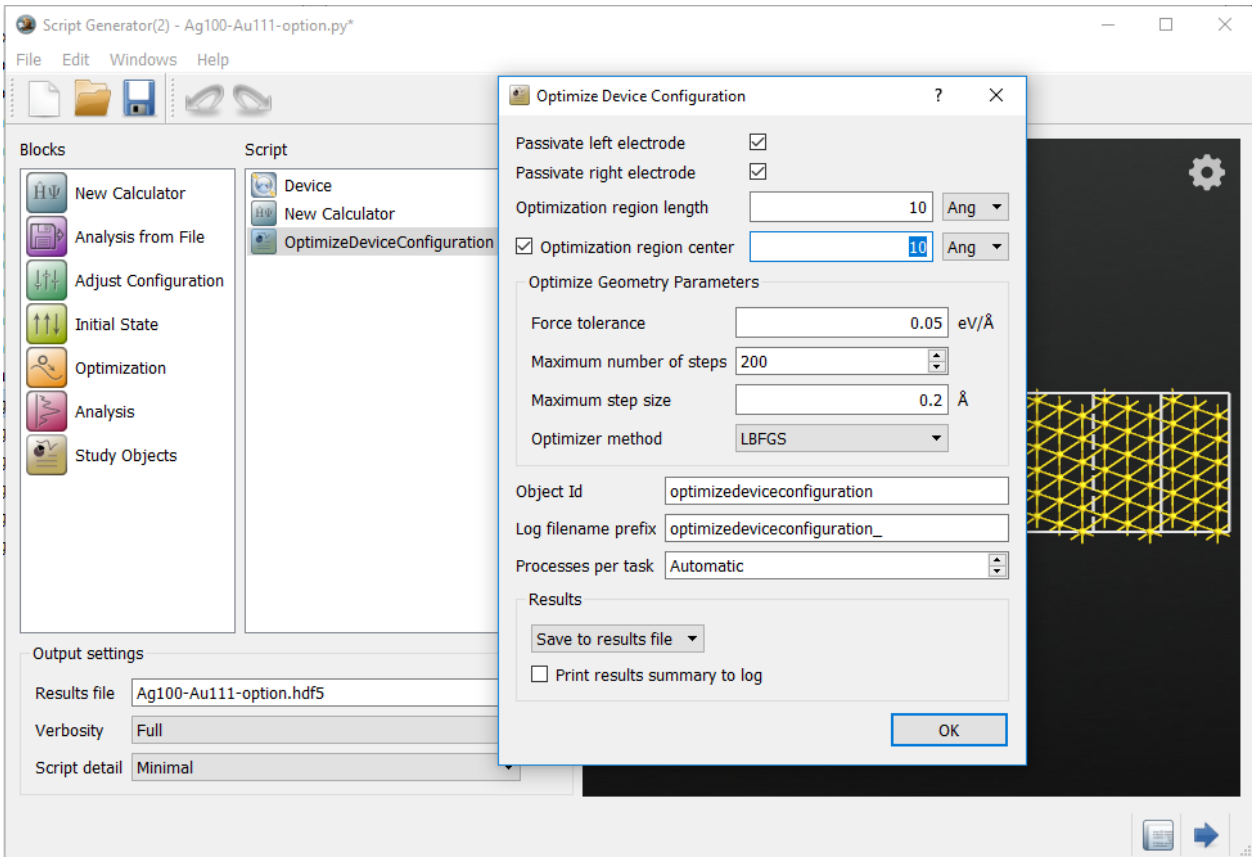
B) Custom optimization region

Open again the **Optimize Device Configuration** widget. For demonstration purposes, you will now specify an optimization region that does not include the Ag|Au interface. The atoms around the interface will therefore not be relaxed.

Tick the option *Optimization region center*, and set the center position to 10 Å, which is measured from the right-hand edge of the left-hand electrode. Keep the optimization region length at 10 Å, such that only atoms within ±5 Å from the center will be allowed to relax.

Note that, since the left electrode extension is of 8.17 Å, i.e., larger than 5 Å, the specified optimization region extends into the electrode extension. This overlap region will not be relaxed because the NEGF method requires that the electrode extension is identical to the electrode.

Finally, close the widget, and change the name of the output results file to `Ag100-Au111-option.hdf5`, and save the Python script as `Ag100-Au111-option.py`. You may also download the final script: `Ag100-Au111-option.py`.



Run the calculations

Run the two **ATK Python** scripts, either using the **Job Manager** or from a terminal:

```
atkpython Ag100-Au111-default.py > Ag100-Au111-default.log
```

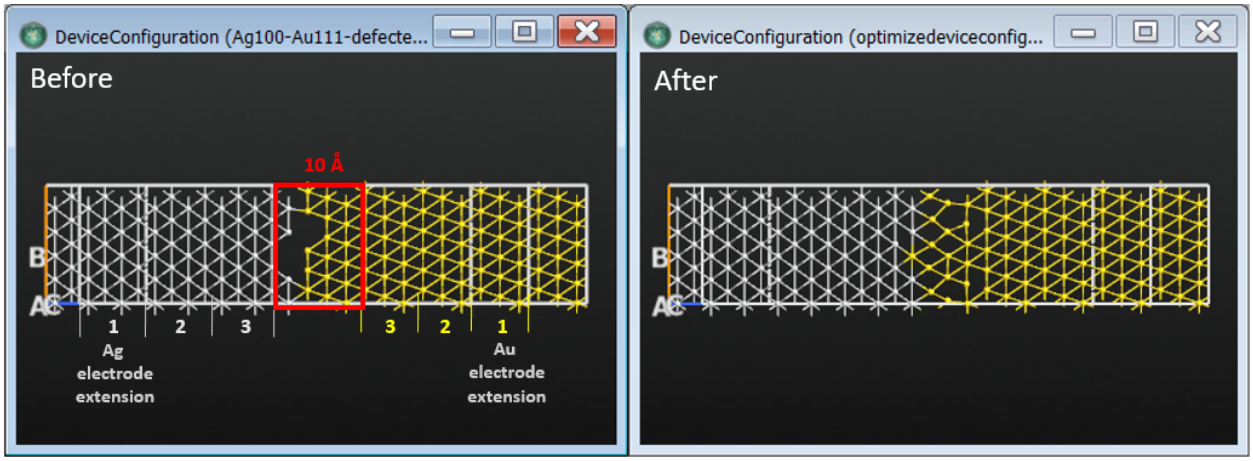
The device geometry optimizations will take only a few seconds to execute.

Relaxed device structures

The **NanoLab LabFloor** should now show the contents of the two saved HDF5 output data files. If not, tick the output files in the left-hand **Project Files** list. Each file contains the *OptimizeDeviceConfiguration* object and the optimized *DeviceConfiguration*.

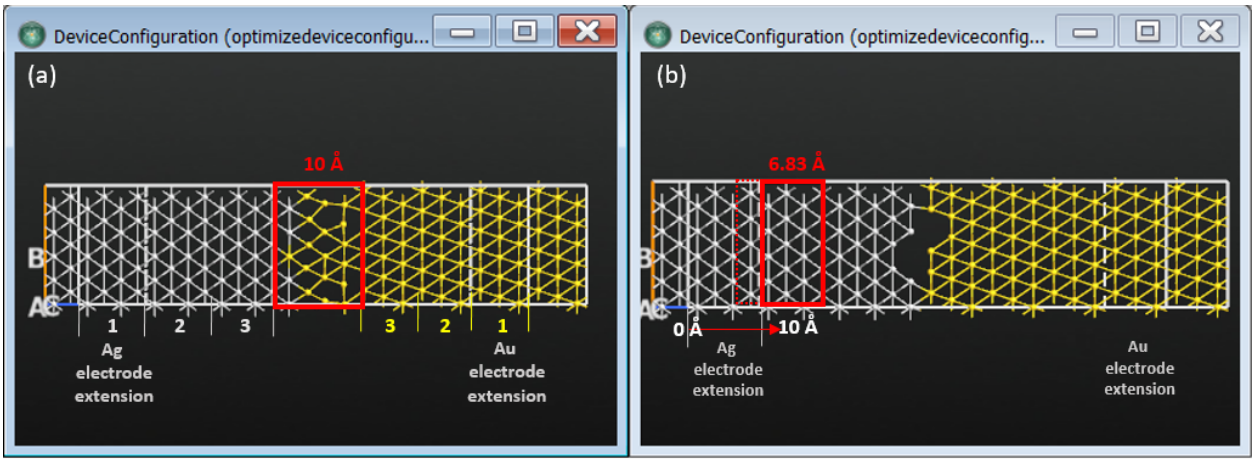
Use the **Viewer** to inspect the optimized devices. You may send the unrelaxed device configuration from the **Builder** to the **Viewer** for easy comparison.

The figure below compares device configurations before and after geometry optimization for **case A** (default settings, `Ag100-Au111-default.hdf5`). You can clearly see the effect of structural relaxation. The periodic repetitions of the left and right electrodes are also indicated, as well as the 10 Å wide region of full atomic relaxation.



The next figure compares the optimized devices from **case A** (default settings) and **case B** (custom optimization region). It is clear that we in case B made a very poor choice of optimization region: Very little relaxation has happened, and the region around the Ag|Au interface is completely unrelaxed. There are two reasons for this:

1. The optimization region **center** is set too close to the left electrode;
2. The optimization region **length** does not cover the interface region with the defects.



The **Text Representation** gives information about the settings used for the OptimizeDeviceConfiguration study object, and a summary of the results. To use it, select one or more OptimizeDeviceConfiguration objects on the **LabFloor**, and click the Text Representation tool in the right-hand plugins bar.

The text representation reports for case A (`Item: 0`) and case B (`Item: 1`) are shown below.

```
# Item:  0
# File:  C:\ATK\OptimizeDeviceConfiguration\Ag100-Au111-default.hdf5
# Title: Ag100-Au111-default.hdf5 - optimizedeviceconfiguration
# Type:  OptimizeDeviceConfiguration
+-------------------------------------------------------------------------------+
| Optimize Device Configuration Report                                          |
+-------------------------------------------------------------------------------+
| Device configuration:                                                         |
+-------------------------------------------------------------------------------+
| Central region (including electrode extensions):                              |
|     Length (before optimization): 54.97 Ang                                   |
|     Length (after optimization): 54.67 Ang                                    |
```

(continues on next page)

(continued from previous page)

```
|     Number of atoms: 135                                                          |
| Electrode extensions:                                                             |
|     Length (left): 8.17 Ang                                                       |
|     Length (right): 7.06 Ang                                                      |
|     Number of atoms (left): 20                                                    |
|     Number of atoms (right): 18                                                   |
+-----------------------------------------------------------------------------------+
| Optimization details:                                                             |
+-----------------------------------------------------------------------------------+
| Optimization region:                                                              |
|     Length: 10.00 Ang                                                             |
|     Left edge position: 24.15 Ang                                                 |
|     Right edge position: 34.15 Ang                                                |
|     Number of atoms: 21                                                           |
| Constrained regions:                                                              |
|     Length (left): 24.15 Ang                                                      |
|     Length (right): 20.82 Ang                                                     |
|     Number of atoms (left): 60                                                    |
|     Number of atoms (right): 54                                                   |
+-----------------------------------------------------------------------------------+

# Item:  1
# File:  C:\ATK\OptimizeDeviceConfiguration\Ag100-Au111-option.hdf5
# Title: Ag100-Au111-option.hdf5 - optimizedeviceconfiguration_option
# Type:  OptimizeDeviceConfiguration
+-----------------------------------------------------------------------------------+
| Optimize Device Configuration Report                                              |
+-----------------------------------------------------------------------------------+
| Device configuration:                                                             |
+-----------------------------------------------------------------------------------+
| Central region (including electrode extensions):                                  |
|     Length (before optimization): 54.97 Ang                                       |
|     Length (after optimization): 54.97 Ang                                        |
|     Number of atoms: 135                                                          |
| Electrode extensions:                                                             |
|     Length (left): 8.17 Ang                                                       |
|     Length (right): 7.06 Ang                                                      |
|     Number of atoms (left): 20                                                    |
|     Number of atoms (right): 18                                                   |
+-----------------------------------------------------------------------------------+
| Optimization details:                                                             |
+-----------------------------------------------------------------------------------+
| Optimization region:                                                              |
|     Length: 6.83 Ang                                                              |
|     Left edge position: 8.17 Ang                                                  |
|     Right edge position: 15.00 Ang                                                |
|     Number of atoms: 15                                                           |
| Constrained regions:                                                              |
|     Length (left): 8.17 Ang                                                       |
|     Length (right): 39.97 Ang                                                     |
|     Number of atoms (left): 20                                                    |
|     Number of atoms (right): 100                                                  |
```

(continues on next page)

(continued from previous page)

```
+-------------------------------------------------------------------------------+
```

Item 0

For case A, you can see the following information in the text representation:

- The central region includes 135 atoms, and initially had a length of 54.97 Å along the transport direction, which decreased slightly to 54.67 Å during relaxation.
- The length of the left (right) electrode extensions are 8.17 Å (7.06 Å), containing 20 (18) atoms each.
- The optimization region length is 10 Å, and the position of the left (right) optimization region edge is at 24.15 Å (34.15 Å), leading to 21 fully relaxed atoms.
- The left and right parts of the device that are constrained during relaxation are 24.15 Å and 20.82 Å long, respectively, and contain 60 and 54 atoms.

Item 1

For case B, we observe that

1. The length of the optimization region is only 6.83 Å, because it has been automatically reduced in order to avoid relaxing the electrode extension.
2. The right-hand edge of the optimization region is at 15 Å, so the Ag|Au interface is not relaxed.
3. Full atomic relaxation has therefore only been done in a region comprising bulk-like Ag, so very little relaxation has actually happened. This is also indicated by that fact that the central region length did not change at all during geometry optimization.

Conclusions

When using the *OptimizeDeviceConfiguration* study object, you may take full control of what is to be the region of full atomic relaxation, by manually setting the center and width of this region. However, it is important to carefully check the result.

Alternatively, you may choose to let the relaxation region extend over the full device, or simply stick to the default 10 Å width and automatic detection of the region center.

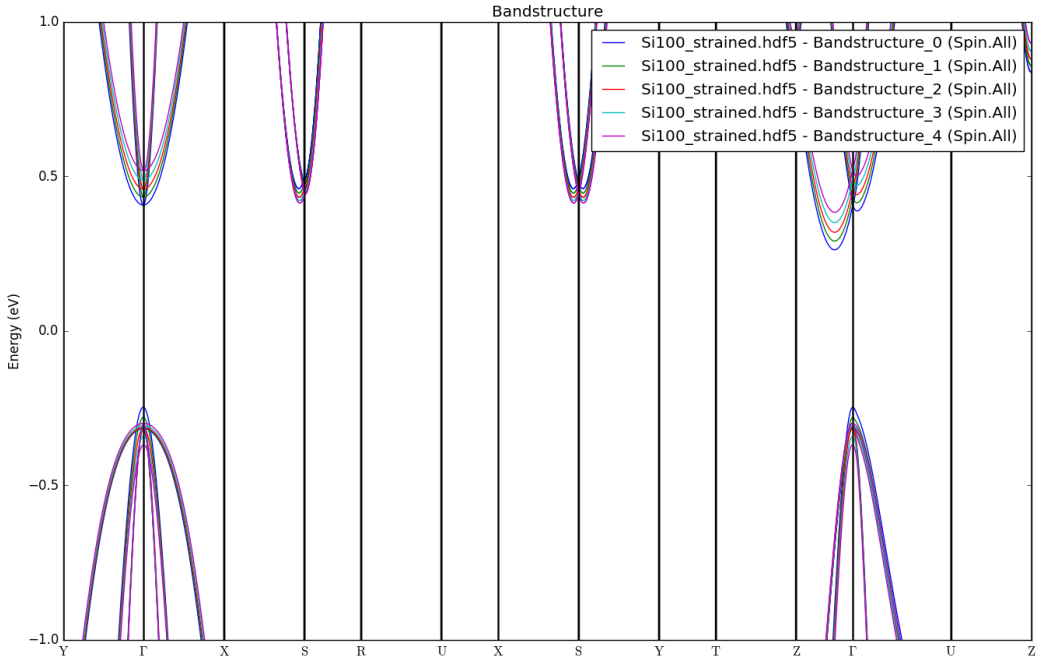
In summary, there are three main modes of operation for the *OptimizeDeviceConfiguration* study object:

- Use default optimization region settings (10 Å width and automatic detection of the region center).
- Manually specify the optimization region center and width.
- Relax all central region atoms (except the electrode extensions) by increasing the optimization region width beyond the central region total length.

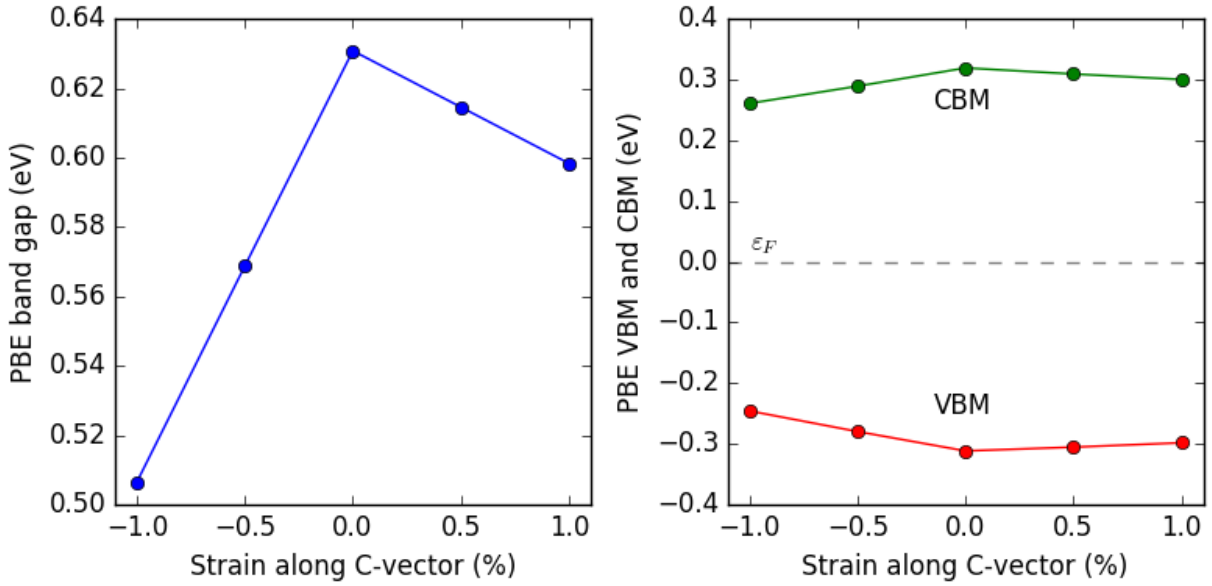
Appendix

As previously mentioned, devices should in general be built from relaxed materials, to avoid effects of stress or strain on the electronic properties. If the starting materials are not relaxed before building the device, geometry optimization of the device may compensate for this by artificially straining parts of the device along the transport direction.

We here show one example of the effect strain on the electronic structure of a semiconductor electrode. The below figure shows the band structure of a Si(100) electrode strained by $\pm 1\%$ along the C-axis, which coincides with the (100) crystallographic direction. By careful inspection one can see that the conduction and valence band energies get slightly shifted.



The figure below shows how the DFT-PBE valence band maximum (VBM) and conduction band minimum (CBM) change upon straining the Si(100) electrode along the C-axis (right-hand panel), and how the silicon fundamental band gap therefore also changes (left-hand panel).



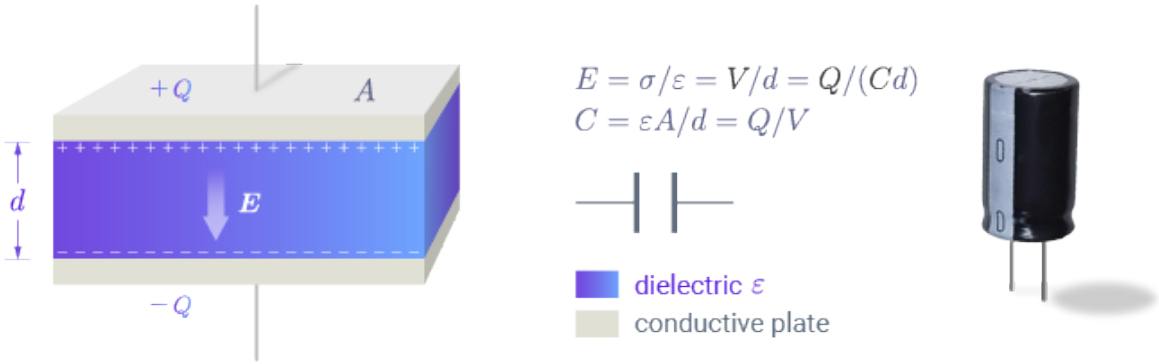
3.4.4 Atomic-scale capacitance

Version: P-2019.03

Downloads & Links

`bias_loop.py` `mulliken-analysis.py` `potential_plot.py` `v-e-plot.py` `v-c-plot.py`

In this tutorial you will perform an atomic-scale simulation of a parallel plate nano-capacitor, which means that the gap between the two parallel metal surfaces is in the nanometer range. This is currently not a realistic device, but the purpose is to show a methodology that can be used to compute the capacitance of other structures, including metal/semiconductor/metal structures.



Attention: Capacitor physics Applying a bias V across the capacitor leads to a build-up of oppositely signed charge Q on the plates, and therefore an electric field E between them. The electric field strength can be written as $E = \sigma/\varepsilon = V/d$, where $\sigma = Q/A$ is the charge density and $\varepsilon = k\varepsilon_0$ is the permittivity of the spacer layer between the plates (vacuum or dielectric). The capacitance can then be calculated from $C = Q/V = Q/Ed = A\varepsilon/d$, where d is the distance between the plates.

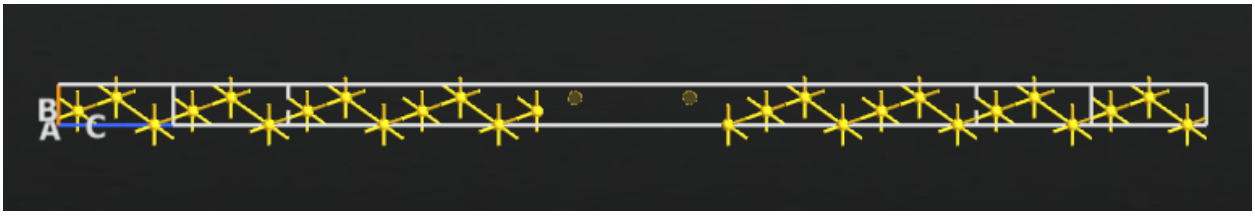
Calculations The equations given above are valid for an ideal macroscopic parallel plate capacitor, but a nano-scale capacitor may in general be quite different. The main method used for computing the capacitance from first principles is to evaluate the electrostatic energy, and from that the capacitance. We will also show a slightly simpler approach based on the Mulliken charge populations of the atoms.

Build the parallel plate capacitor

Open the **Builder** , and do the following:

- Use *Add* → *From Database* to add gold to the Stash.
- Open the *Builders* → *Cleave* plugin, and create a Au(111) electrode by entering the Miller indices as [111].
- Repeat the structure 8 times in the C direction using *Bulk Tools* → *Repeat*.
- Select and delete the two atoms in the middle – they have atom indices 19 and 4.
- Select the surface atoms (indices 10 and 11 after deletion), and turn them into ghost atoms using the toolbar icon .
- Turn the bulk structure into a device configuration via *Device Tools* → *Device From Bulk* and use the default settings.

The result will be the device as shown below.



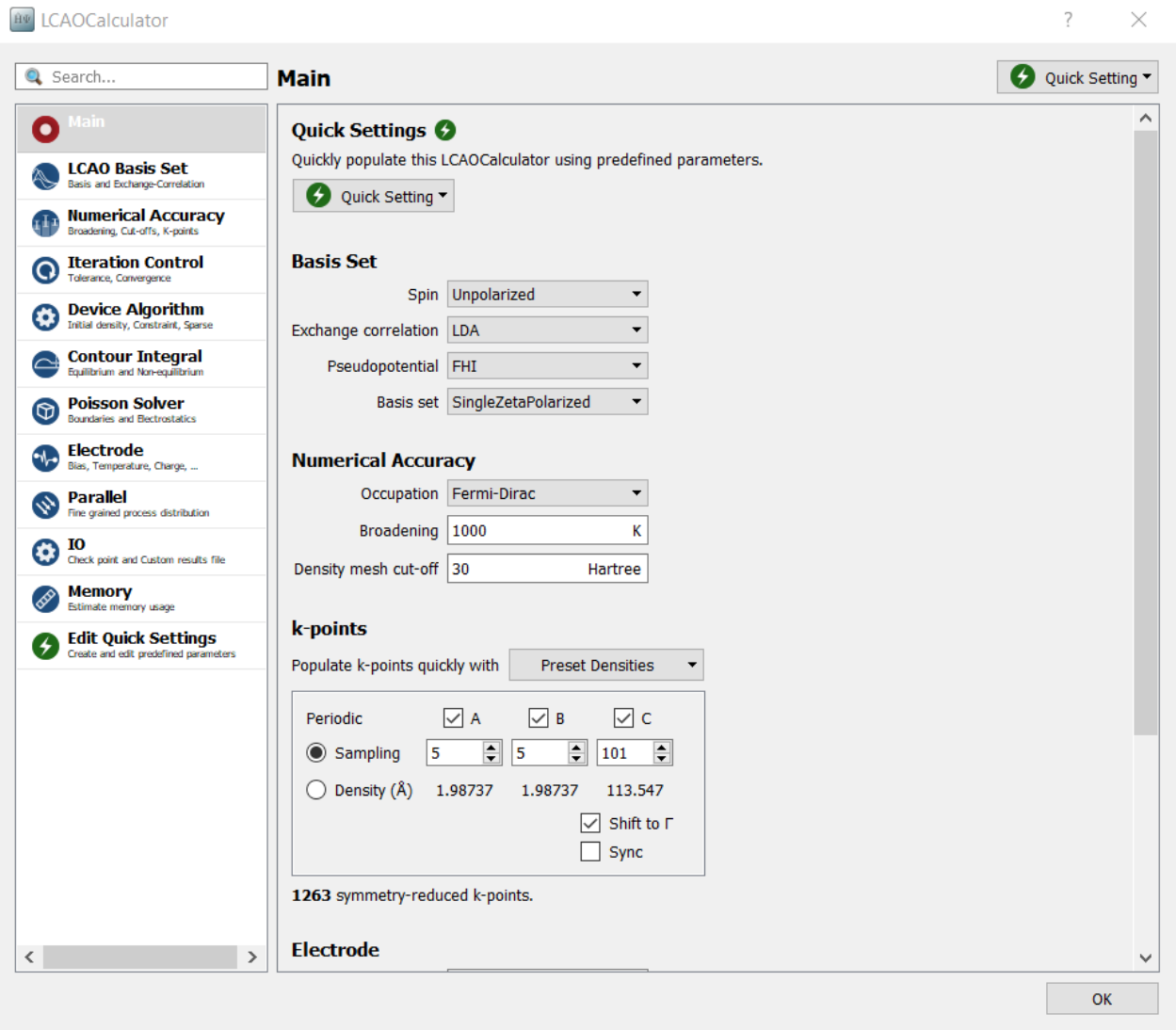
Note: The ghost atoms are used to improve the description of the electron density between the two surfaces. They are simply extra basis sets placed in the empty region. This technique, which is sometimes referred to as a vacuum basis, is also used for calculating the work function of a surface, as explained in a tutorial: *Computing the work function of a metal surface using ghost atoms*.

Calculations

Zero bias

Send the device configuration to the Script Generator by clicking the icon, and set up the script as follows:

- Add an **LCAOCalculator** and use the DFT-LDA engine with a SingleZetaPolarized basis set. Then set a 5x5x101 k-point grid.



•Change the output filename to `au_vacuumgap.hdf5`.

•Save the script as `au_vacuumgap.py`.

Send the script to the **Job Manager** and execute the calculation. It takes 3-4 minutes on a laptop.

Finite bias

To compute the capacitance it is necessary to compute dq/dV, where dq is the induced charge on the surfaces and V the bias voltage. In the bias voltage you will also run the following analysis objects : **MullikenPopulation**, **ElectrostaticDifferencePotential**, and **ElectronDifferenceDensity**.

Use the following script to perform the calculations for bias voltages of 0.25, 0.5, 0.75, and 1.0 V:

```
device_configuration = nlread('au_vacuumgap.hdf5', DeviceConfiguration)[-1]
calculator = device_configuration.calculator()
for voltage in [0.0, 0.25, 0.5, 0.75, 1.0]:
```

(continues on next page)

(continued from previous page)

```
    # Set new calculator with modified electrode voltages on the configuration
    # use the self consistent state of  the old calculation as starting input.
    device_configuration.setCalculator(
          calculator(electrode_voltages=(0.5*voltage*Volt, -0.5*voltage*Volt)),
                     initial_state=device_configuration)
    device_configuration.update()
    nlsave('au_vacuumgap.hdf5', device_configuration, object_id="SCF %s" % voltage)
    # ------------------------------------------------------------
    # Mulliken population
    # ------------------------------------------------------------
    mulliken_population = MullikenPopulation(device_configuration)
    nlsave('au_vacuumgap.hdf5', mulliken_population, object_id="Mulliken %s" % voltage)
    # ------------------------------------------------------------
    # Electrostatic difference potential
    # ------------------------------------------------------------
    electrostatic_difference_potential = ElectrostaticDifferencePotential(device_
↪configuration)
    nlsave('au_vacuumgap.hdf5', electrostatic_difference_potential, object_id="EDP %s" %␣
↪voltage)
    # ------------------------------------------------------------
    # Electron difference density
    # ------------------------------------------------------------
    electron_difference_density = ElectronDifferenceDensity(device_configuration)
    nlsave('au_vacuumgap.hdf5', electron_difference_density, object_id="EDD %s" %␣
↪voltage)
```

You can download the scipt here: `bias_loop.py`. Run it using the Job Manager or execute it from a terminal:

```
$ atkpython bias_loop.py > bias_loop.out
```

It takes about half an hour to finish this calculation on a laptop.

Note: The previously converged calculation is used as an initial guess for the next calculation to speed up the SCF convergence. The `object_id` makes it easier to identify the data in post-SCF analysis.

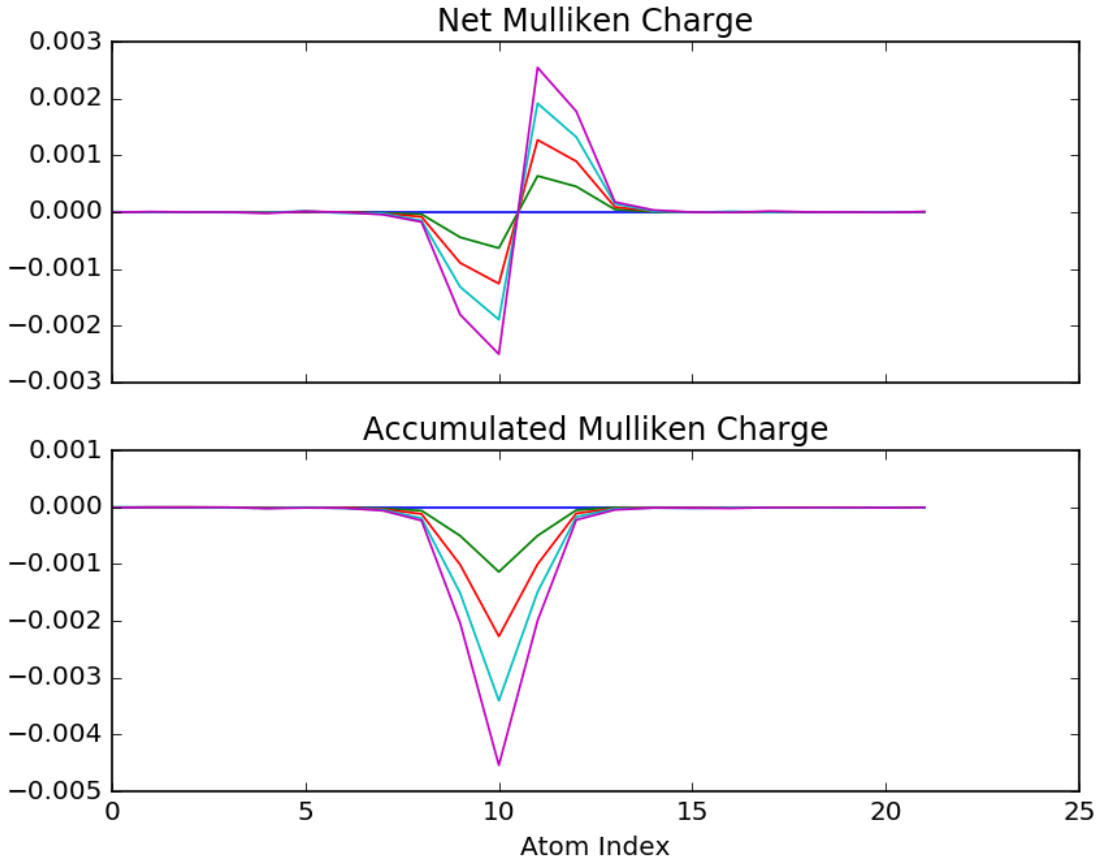
Analysis

We will introduce two different ways to calculate the capacitance:

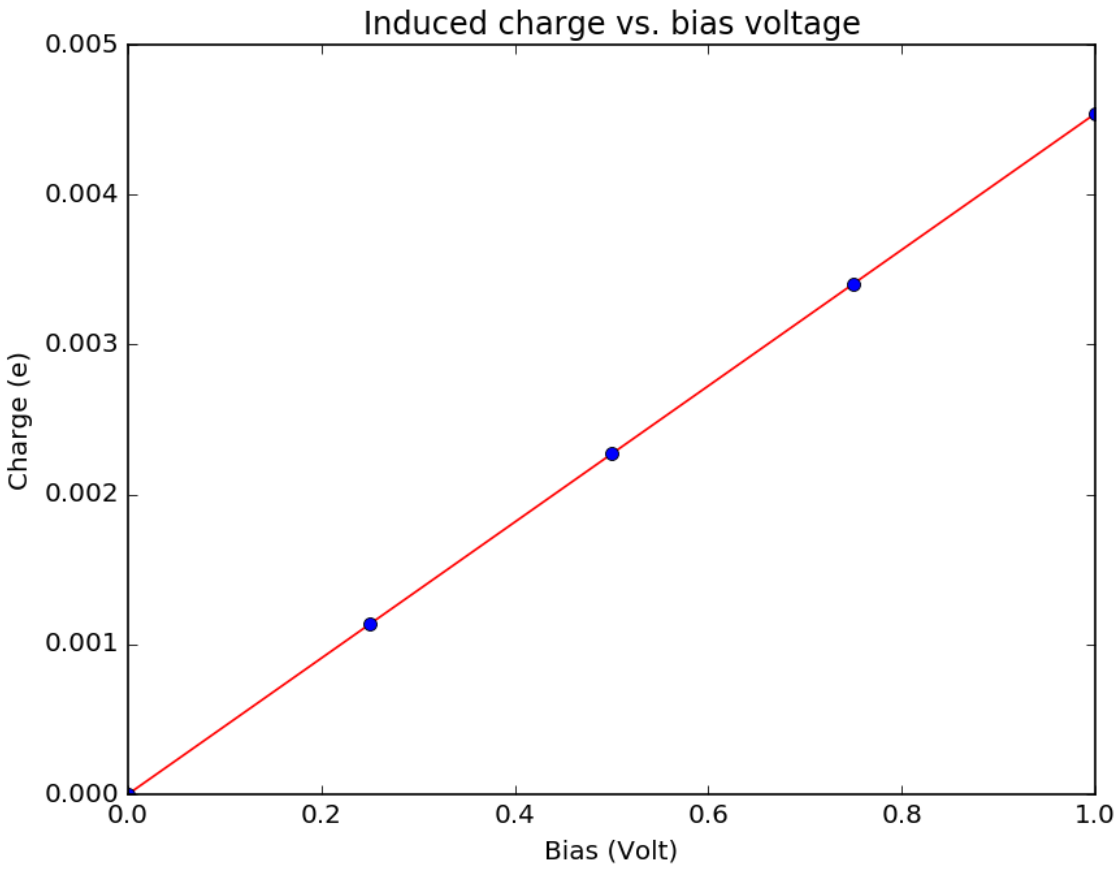
- a simple approach based on the induced charge;
- a more general method based on evaluating the electrostatic energy.

Approach based on the induced charge

The finite bias will build up a net positive/negative charge on the two gold surfaces, as shown in the plots below.



Now, the capacitance can in principle be computed as $C = dq/dV$, where q is the total accumulated charge on one surface as a function of the bias. From the slope of the straight line you can therefore obtain the capacitance of the nano-capacitor:



The plots above, as well as the entire data analysis based on the Mulliken populations, can be achieved by running this script: `mulliken-analysis.py`. It is a long script but a great opportunity to learn about custom data analysis in Python and QuantumATK.

Results

The script reports a capacitance of 7.27*10^{-22} Farad, as computed from fitting a straight line to the data points. To put this number into perspective, consider the capacitance of an ideal parallel plate capacitor, written as $C = \varepsilon A/d$, where $\varepsilon = 1$ for the vacuum permittivity and the area A of each plate can be calculated from the device structure. You can then calculate the parallel plate distance $d = A/C$. This is also done in the script and it reports d=8.77 Å.

Comparing this value with the distance between the surface gold atoms (11.89 Å) the position of the so-called image plane is estimated to be 1.56 Å above the gold surface.

Electrostatic energy analysis

To compute the capacitance using a more general and accurate method, you will need to first calculate the electrostatic energy E as a function of the applied bias V:

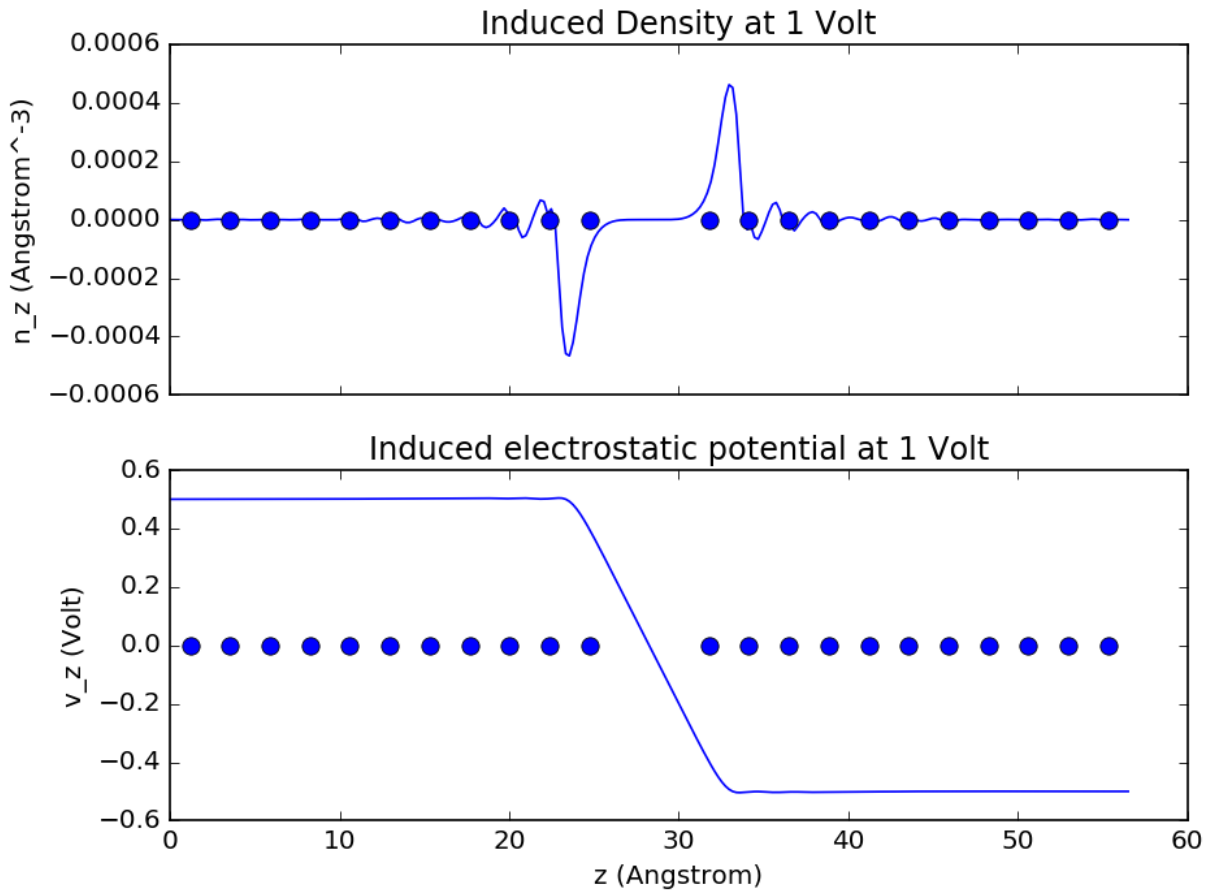
$$E(V) = \frac{1}{2}\int \delta v(\mathbf{r}, V)\delta n(\mathbf{r}, V)d\mathbf{r}$$

where δv is the induced electrostatic potential and δn is the induced density.

The script `potential_plot.py` extracts the potential and density at 1 V bias from the existing HDF5 file and produces the plots of the induced density and electrostatic potential. The script also reports two parameters:

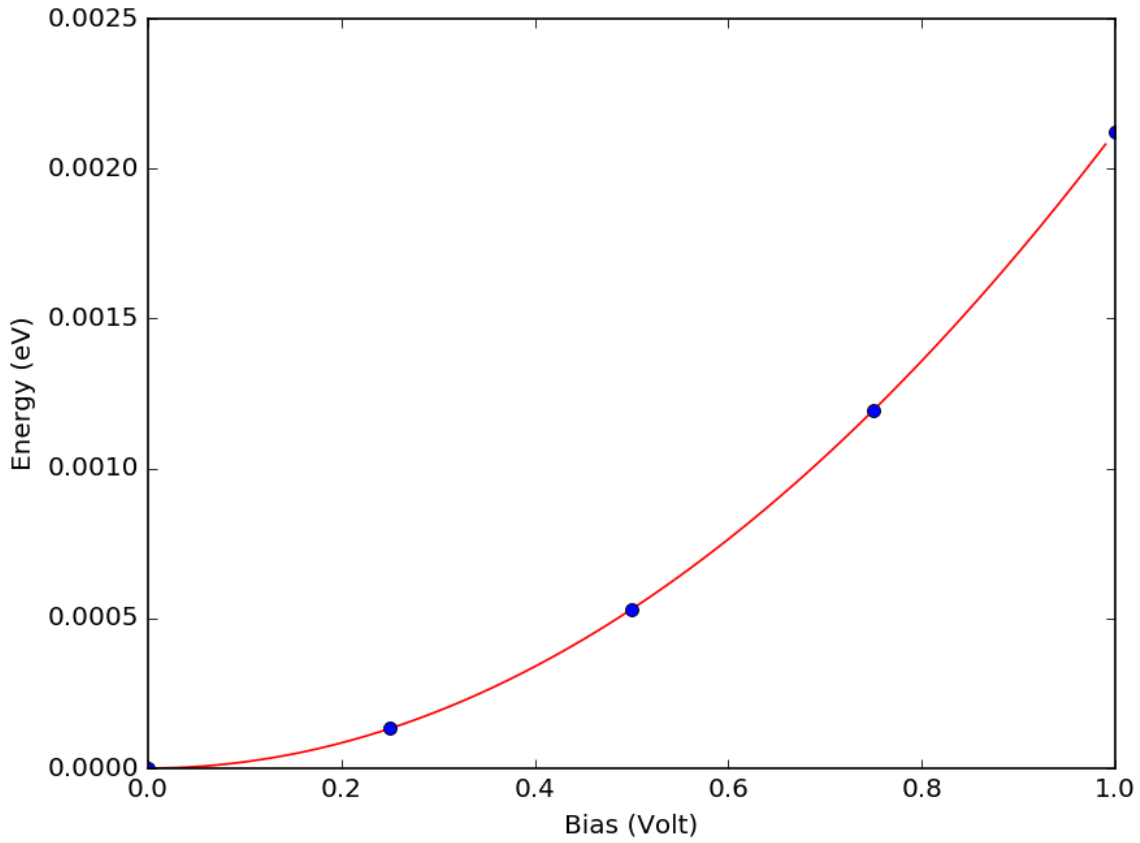
- the electric field in the middle of the vacuum gap E= 0.1127 V/Å;
- the distance between the two image planes d=8.88 Å.

Tip: The position of the image planes is defined as the point where the electric field is zero, that is, where the electrostatic potential becomes flat.



The blue circles correspond to the atom positions. The main charge accumulation takes place in a very narrow region right above the surface, between the outer surface atoms and the ghost atoms.

From this, the capacitance can be extracted by fitting a parabola to the relation $E(V) = CV^2/2$. The script `v-e-plot.py` calculates the electrostatic energy and fits the capacitance as shown below.



Results

The script reports the capacitance C=6.81*10^{-22} Farad, in qualitative agreement with the value obtained from the simple approach using Mulliken charge analysis.

This script also reports d=9.36 Å, corresponding to an image plane at 1.27 Å above the gold surface, in very close accordance with the position of the peak of induced density.

Attention: What is the difference between the two methods? In the Mulliken charge analysis, it is assumed that all electrons on the left surface have the same potential, and similarly for the right surface. When we convolute the electron density and the electrostatic potential we account for the fact that some part of the electron density is located in the region with a finite electric field. The electrostatic energy analysis is therefore more accurate for calculating the capacitance.

The electrostatic energy analysis also has the advantage that it does not require the researcher to separate the atoms as uniquely left and right – this is trivial for the two gold surfaces, but that may not always be the case.

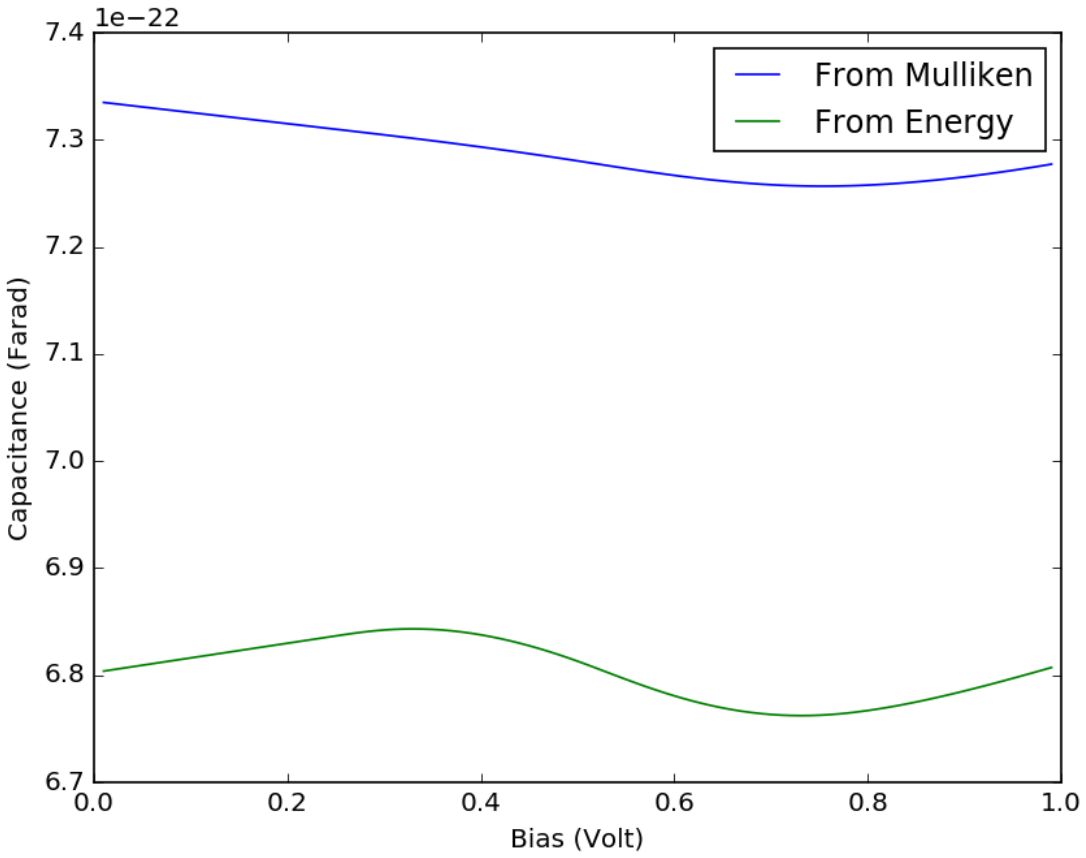
Bias-dependent capacitance

The analysis outlined above is based on the assumption that the capacitance is **independent** of the voltage. However, this is not always the case. A more advanced spline curve fitting allows you to investigate systems whose capacitance does depend on the bias voltage.

Provided that you have already run the scripts `mulliken-analysis.py` and `v-e-plot.py` in the previous section, you can now download and run the script `v-c-plot.py` to perform the fitting.

Note: The data saved by the two scripts are used by the present script to accelerate the analysis.

The computed capacitance is shown below as a function of the bias. As expected, the capacitance changes in this case very little with the bias voltage.



Attention: The present example uses rather few data points to illustrate the method. In general, you should use more than five data points to get an accurate analysis.

Dielectric spacer material

The electric field in the vacuum gap is extremely high (over 900*10^6 V/m at 1 V bias) and would immediately lead to a breakdown discharge in the device. Replacing the vacuum gap between the capacitor plates with a dielectric material can partly solve this.

There are two different ways to do this using QuantumATK:

1. Insert a dielectric material, and explicitly form the two metal–semiconductor–metal interfaces:

- **Spatial region:** Use the QuantumATK Builder to place a dielectric spatial region between the surfaces.

2. Use a cheaper and less rigorous approach, where you simply modify the dielectric constant of the region between the gold surfaces:

- **Implicit solvent method:** Assign a dielectric constant different than 1 to the vacuum region by modifying settings in the Poisson solver.

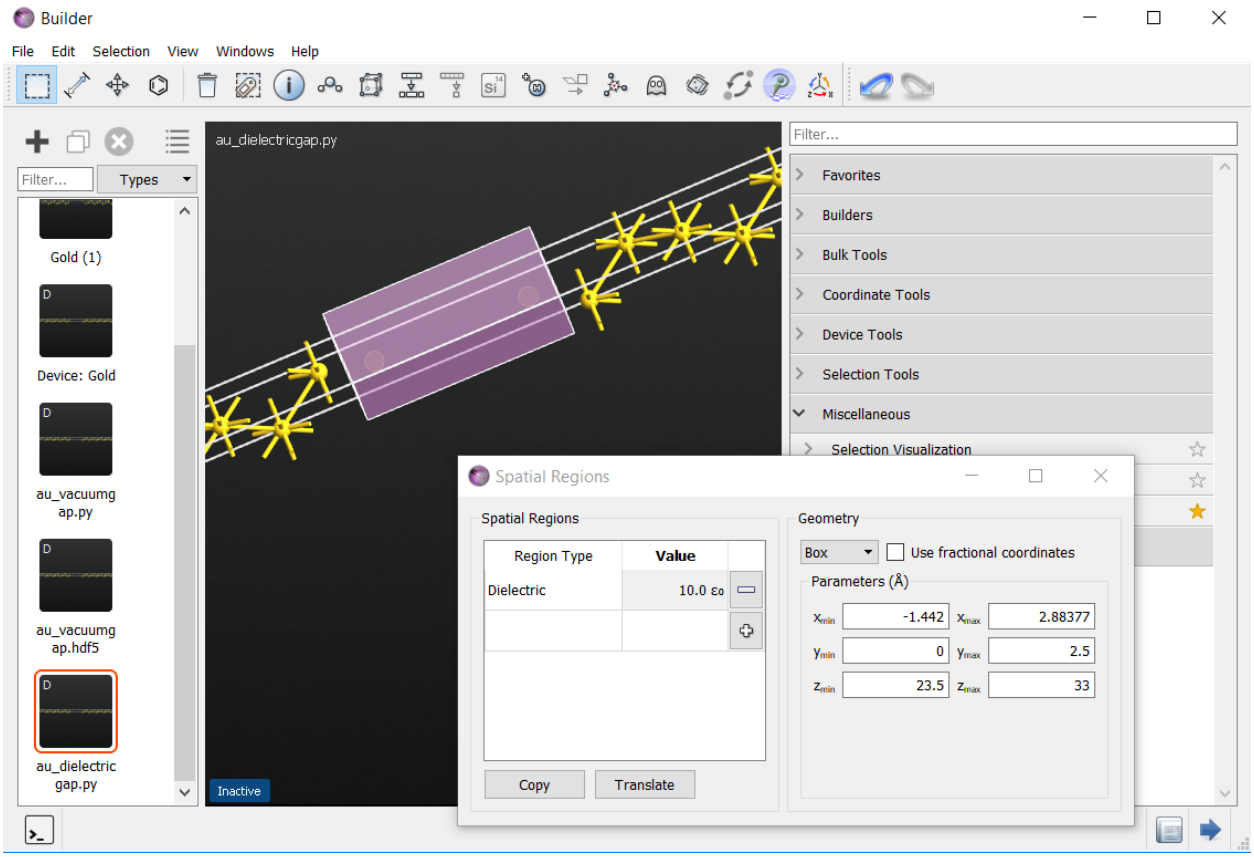
For demonstration purposes, we here go for the first option.

Note: In both cases it is necessary to use the `Multigrid` Poisson solver, but you can still have periodic boundary conditions along A and B.

Dielectric spatial region

Follow the steps that led to calculation of the capacitance from electrostatic energy analysis, but add the dielectric spatial region and change the **Poisson solver** to `Multigrid`.

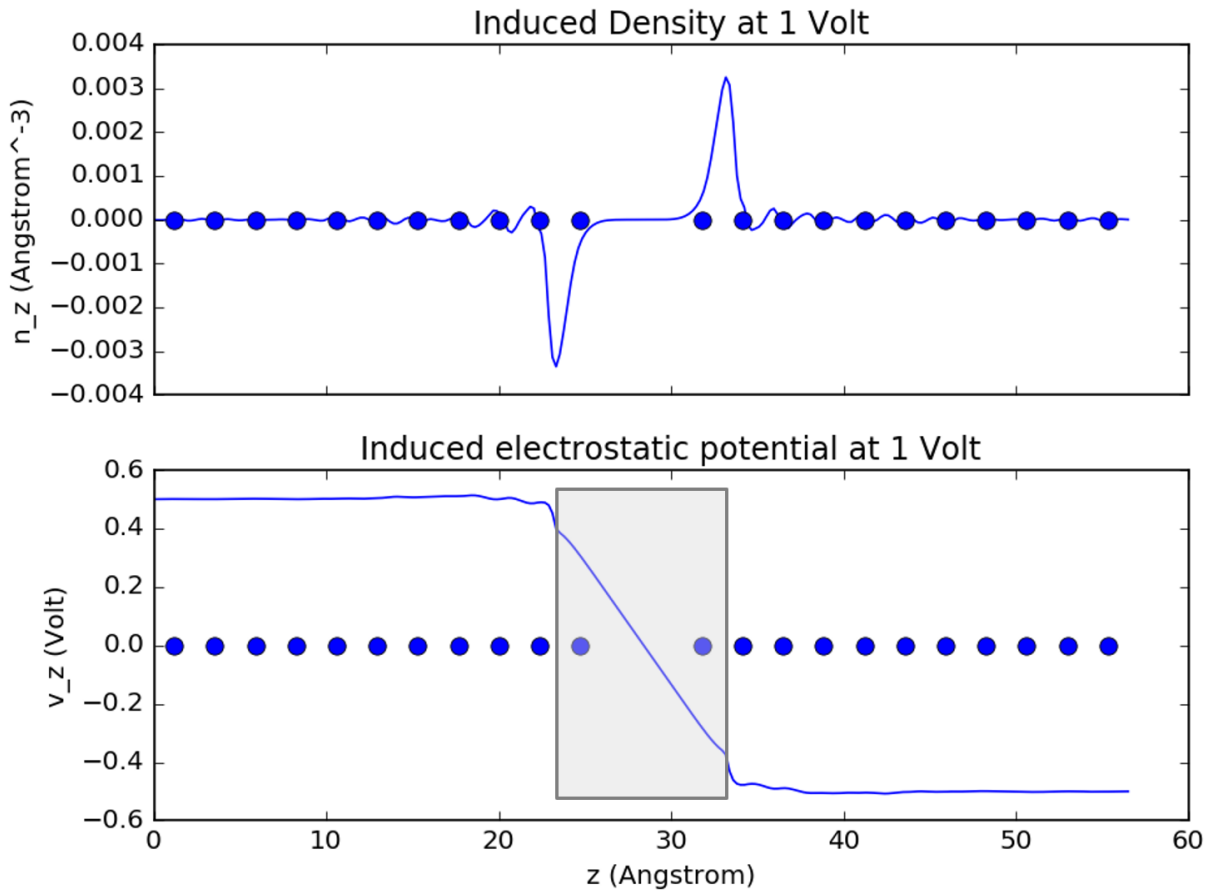
To use a dielectric spatial region, go to the Builder and use the *Miscellaneous* → *Spatial Regions* tool to place between the surfaces a box with a relative dielectric constant of 10:



Note: The dielectric region is a rectangular box, while the unit cell is hexagonal for the present [111] fcc surface. To solve the mismatch, make sure the rectangular box is large enough in the AB plane to extend outside the unit cell. The parts outside the cell will be cut off during the calculation.

Then send the new geometry to the Script Generator and repeat all the steps. Or simply open the structure in the Editor and copy/paste the relevant parts of the scripts you already have, and change the names of the output files.

The results are illustrated below. The position of the dielectric region is indicated by the grey shading. You can clearly see the expected discontinuity of the electric field near the edges of the dielectric region, and you will find that the capacitance has significantly increased.



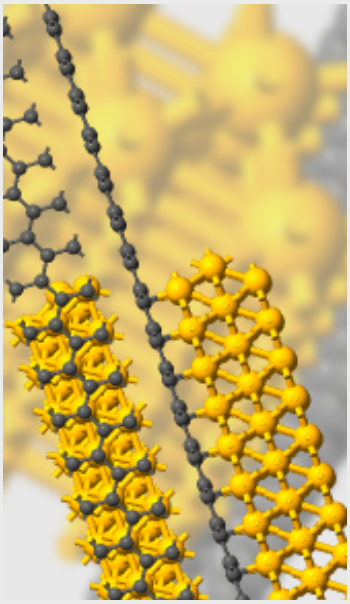
Note:

- The scripts used in the electrostatic energy analysis are self-contained and generic. They can be used to analyse other systems as well, after suitable modification of the file names.
- The scripts used in the Mulliken charge analysis need to be modified before they can be used for other systems, to account for geometry changes.
- These approaches are valid only if the current through the device is very small. For example, the transmission is extremely small (its a capacitor – not a diode!). You can verify that the transmission is extremely small for this large vacuum gap, of the order 10^{-23}.

Tip: What next? Another interesting example of a nano-scale capacitor would be a one formed by *Carbon Nanotube Junctions*.

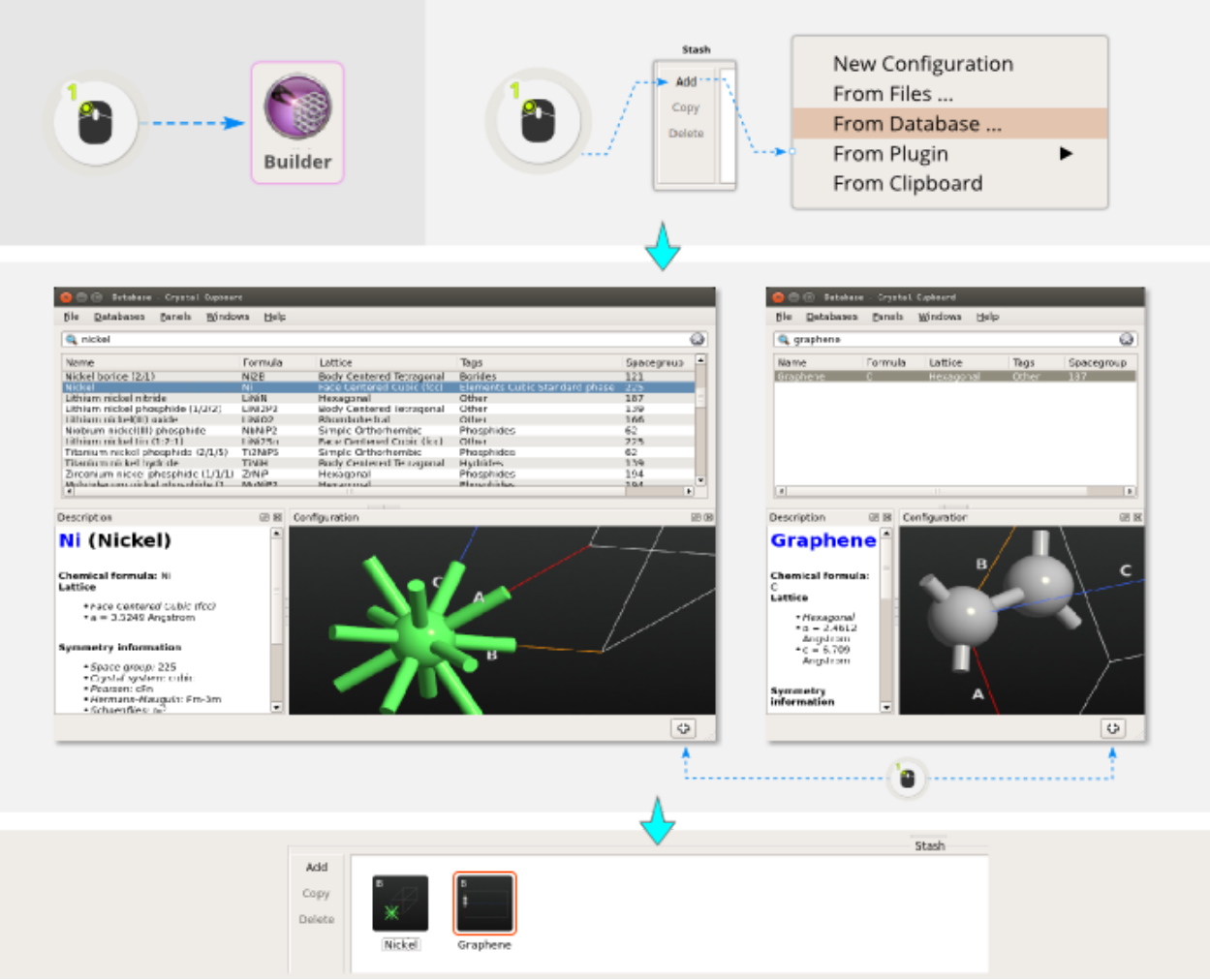
3.4.5 Graphene–Nickel interface

QuantumATK: Interface Platforms: Links: *ATK can easily handle heterogeneous systems, where the two electrodes are not the same, as demonstrated in the above publication. Here you will learn how to build a similar system.*

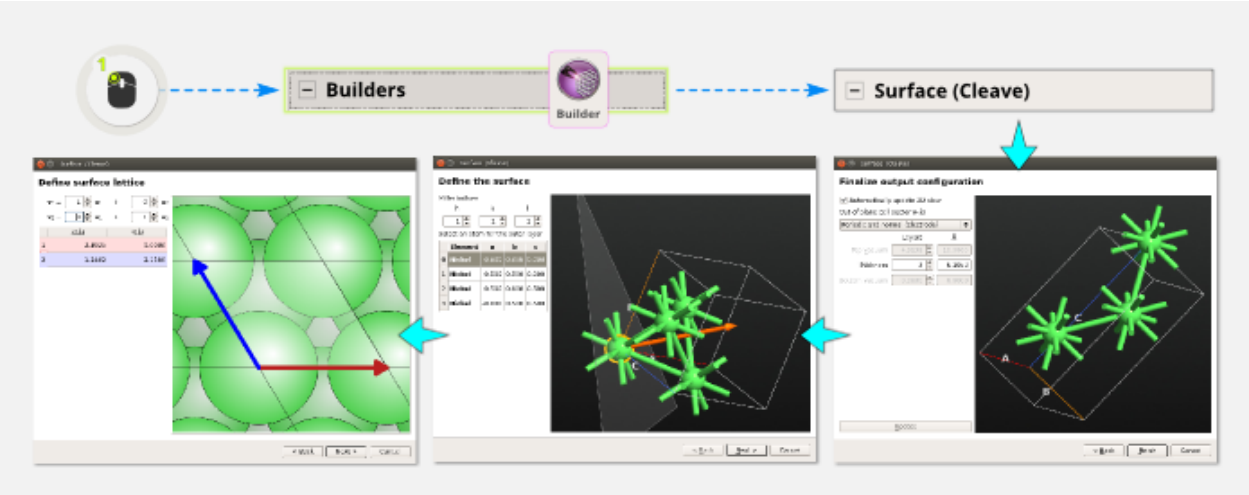


Creating the structure

- **Step 1:** Click *Add* → *Add From Database*. Locate nickel and add it to the Stash. Then also add graphene from the crystal database.



•**Step 2:** Graphene fits very nicely onto Ni(111), so thats the surface well consider. Use the cleave tool *Builders* → *Surface (Cleave)* to cleave the crystal along the (111) plane and make a 3-layer electrode cell.

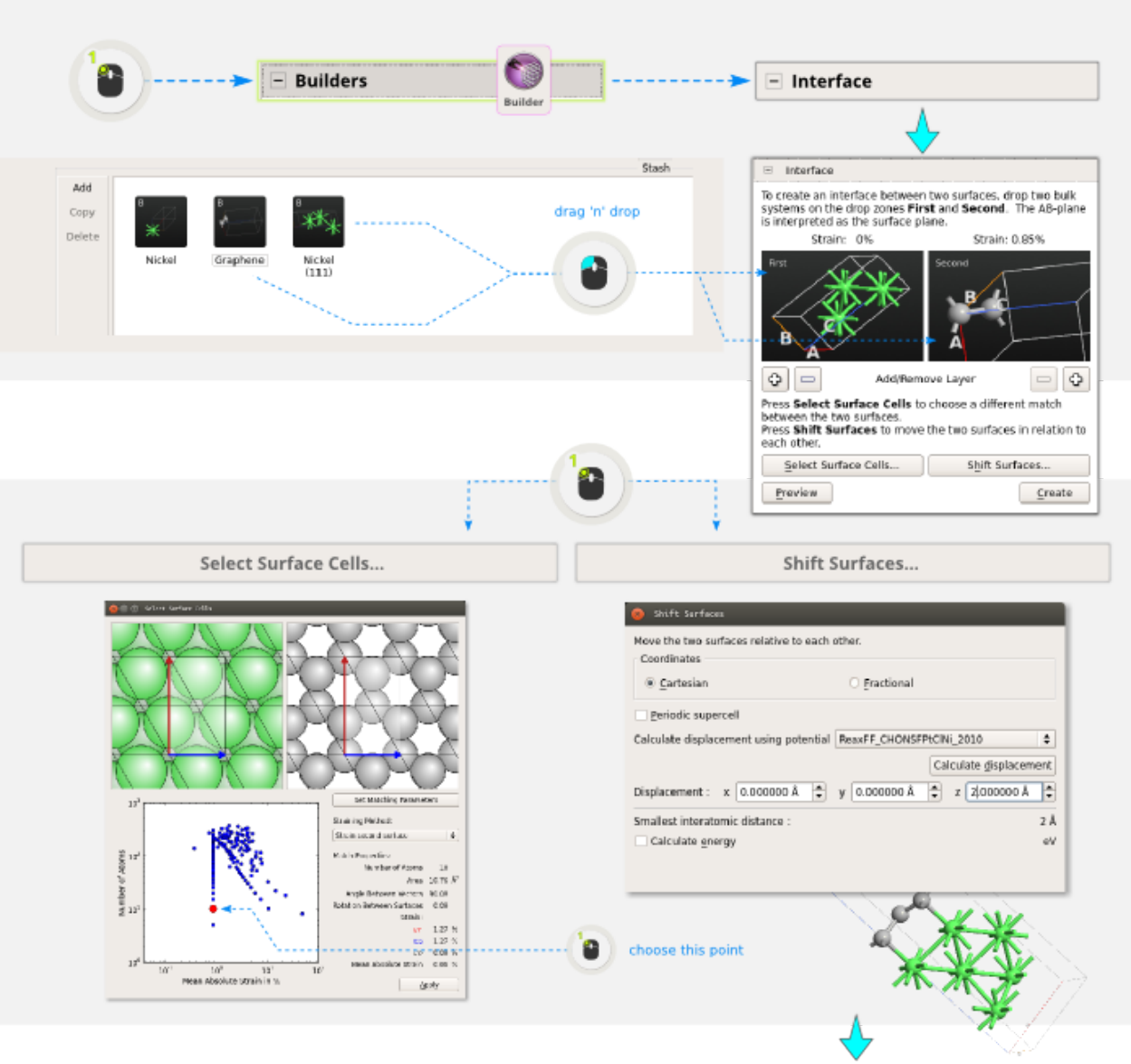


•**Step 3:** Make an interface between the two system by opening *Builders* → *Interface*. Drag the Ni(111) configuration to the first slot and graphene to the second slot.

•**Step 4:** The suggested choice of cell is the minimal commensurate cell. This cell has hexagonal symmetry, but for this

calculation it is more convenient to work with a rectangular cell. Therefore, click **Select Surface Cells**, and pick the cell with a slightly larger number of atoms but the same strain, as shown in the picture below. To choose this cell, click the corresponding point in the plot at the bottom left.

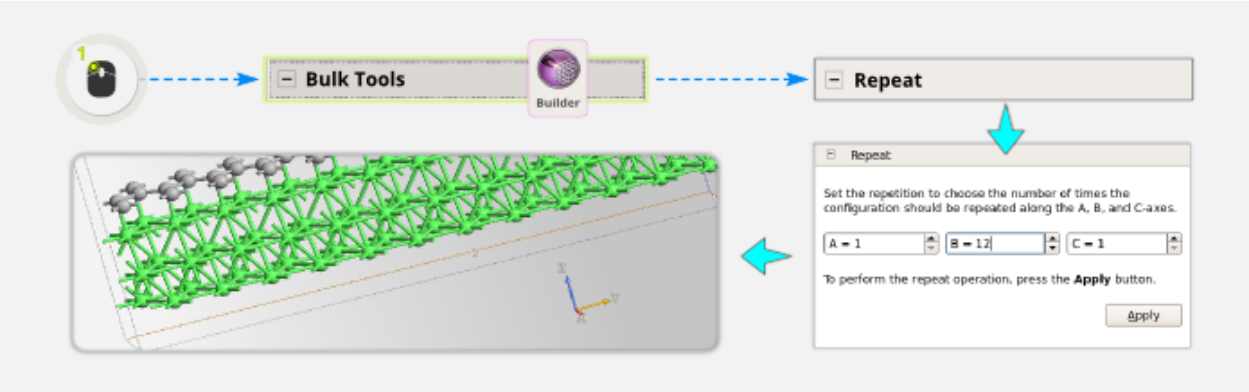
•**Step 5:** Click **Shift Surfaces** and add 2 Angstroms to the Z position.



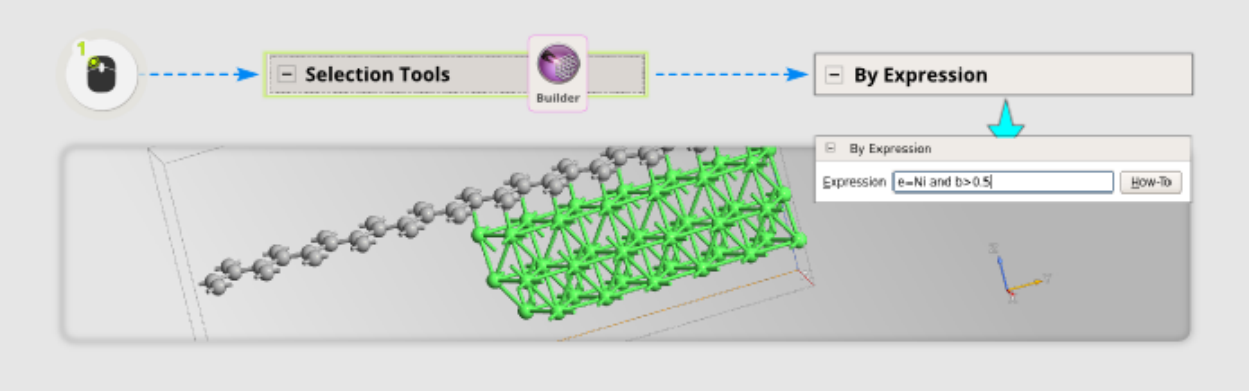
Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

More configurations

•**Step 5:** Use *Bulk Tools* → *Repeat* and repeat the structure 12 times in the **B** direction to make a larger sheet of graphene on an extended Ni(111) surface.



•**Step 6:** Open *Selection Tools* → *By Expression* and enter **e=Ni and b>0.5**. Press Enter or Tab, and then Delete. This cuts away half of the Ni slab.



•**Step 7:** Decide how large a Ni–graphene interface you want, and then use the mouse to select and delete the superfluous carbon atoms.

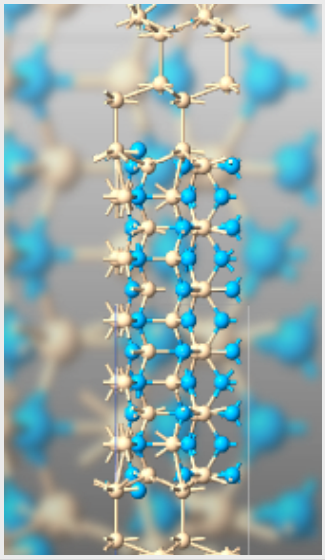
•**Step 8:** If you want to saturate the dangling bonds of the edge carbon atoms (cf. the article – you may or may not want to do this), select the edge atoms and use the passivation tool .

•**Step 9:** The system needs to be re-oriented, since the transport direction is always along Z. Use *Bulk Tools* → *Swap Axes* to swap **B<->C**, and **Y<->Z** (you should normally always swap twice to get a right-handed system).

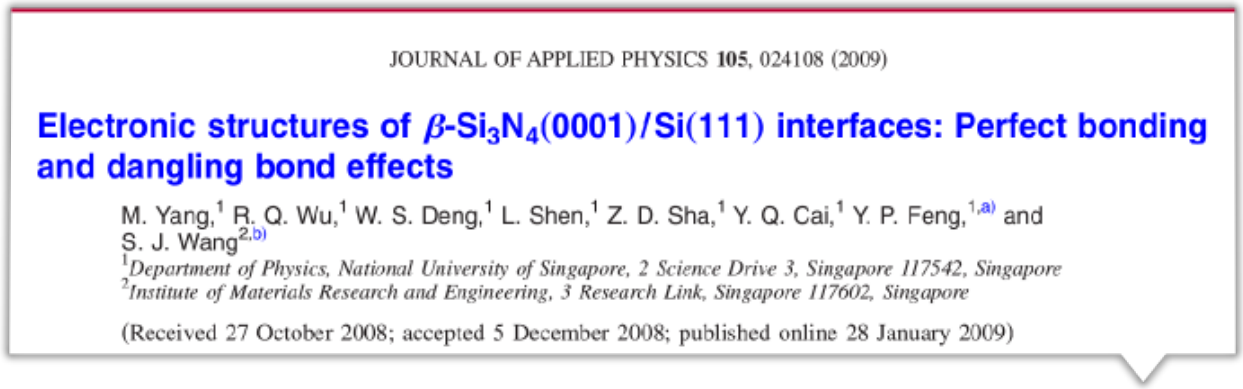
•**Step 10:** Finally, use *Device Tools* → *Device From Bulk* to convert the structure to a device configuration; default option are fine so just click OK.

3.4.6 Building a Si-Si_3N_4 Interface

QuantumATK: Complex Interface Platforms: Links: , `AMS_DATA.cif`, `Si3N4.cif` *In this tutorial you will learn how to build a complex interface using QuantumATK. In particular, we will consider an interface between silicon and silicon nitride.*



Note:



JOURNAL OF APPLIED PHYSICS **105**, 024108 (2009)

Electronic structures of β-Si_3N_4(0001)/Si(111) interfaces: Perfect bonding and dangling bond effects

M. Yang,[1] R. Q. Wu,[1] W. S. Deng,[1] L. Shen,[1] Z. D. Sha,[1] Y. Q. Cai,[1] Y. P. Feng,[1,a)] and S. J. Wang[2,b)]

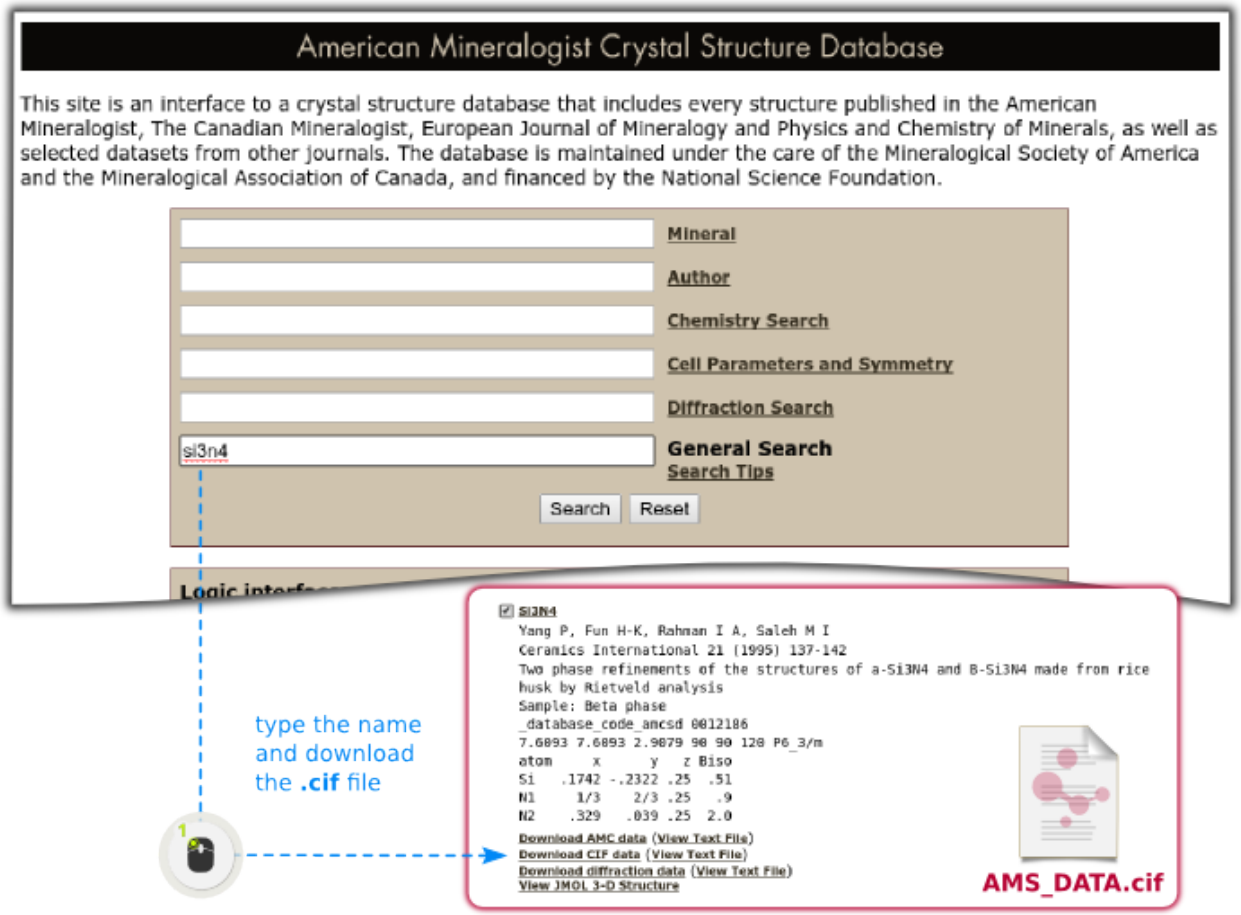
[1]*Department of Physics, National University of Singapore, 2 Science Drive 3, Singapore 117542, Singapore*
[2]*Institute of Materials Research and Engineering, 3 Research Link, Singapore 117602, Singapore*

(Received 27 October 2008; accepted 5 December 2008; published online 28 January 2009)

The -Si_3N_4(0001)/Si(111) interface was as studied in detail in Ref.[1], and we will use the geometry proposed in that work. An earlier model for the crystalline interface of -Si_3N_4(0001)/Si(111) was proposed by , but that model contained dangling bonds, which is not consistent with . The interface you will build here has all bonds saturated.

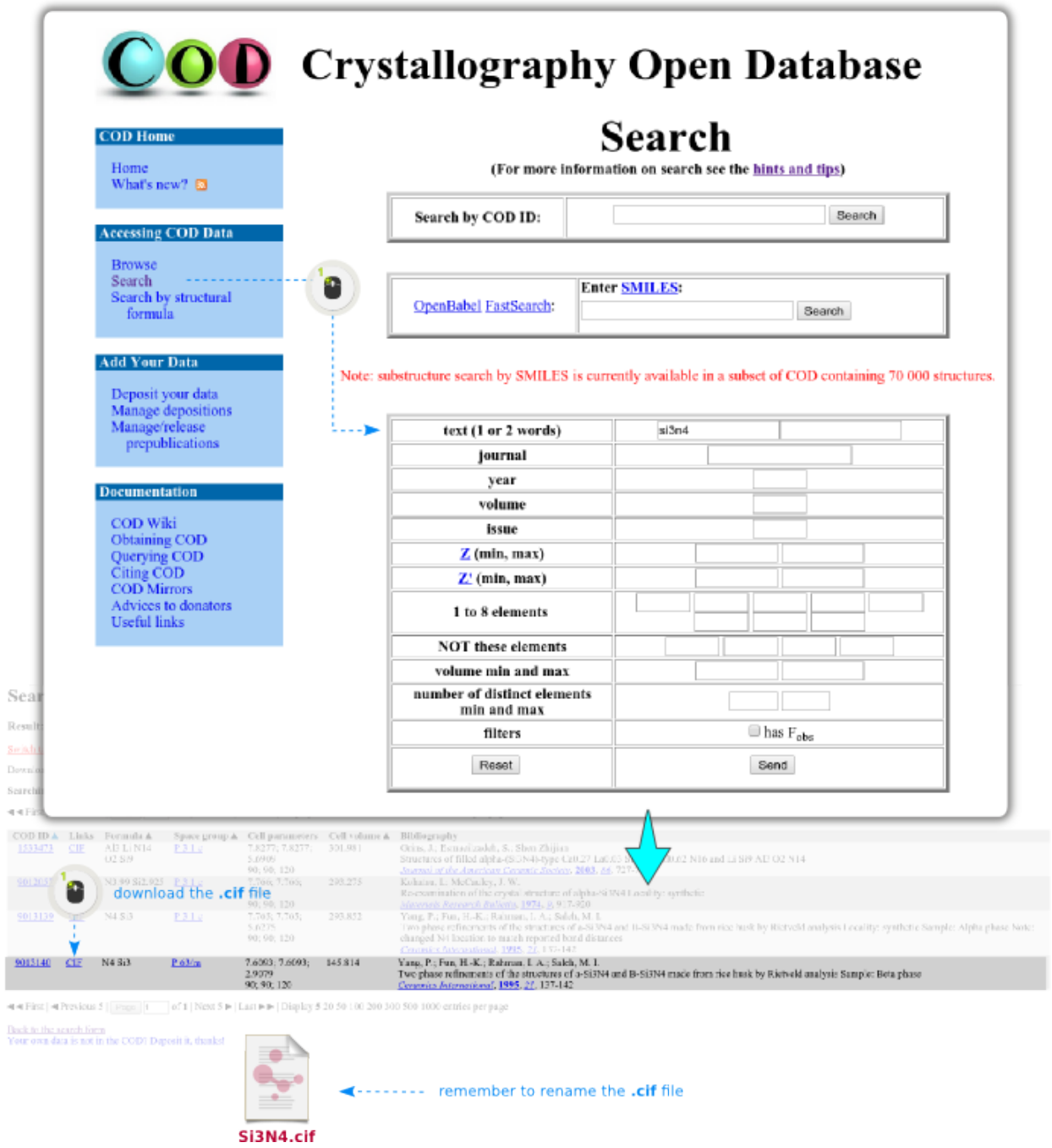
To find an appropriate input structure for -Si3N4, visit the and enter si3n4 in the General Search box. This will return three matches - the one entitled Si3N4, by P. Yang *et al.*, is the -Si3N4 you are looking for (the other two are nierite, -Si3N4). Click Download CIF data and save the CIF file in a convenient location.

[1] M.ăYang, R.ăQ. Wu, W.ăS. Deng, L.ăShen, Z.ăD. Sha, Y.ăQ. Cai, Y.ăP. Feng, and S.ăJ. Wang. Electronic structures of -Si3N4(0001)/Si(111) interfaces: Perfect bonding and dangling bond effects. *Journal of Applied Physics*, 2009. doi:10.1063/1.3072625.



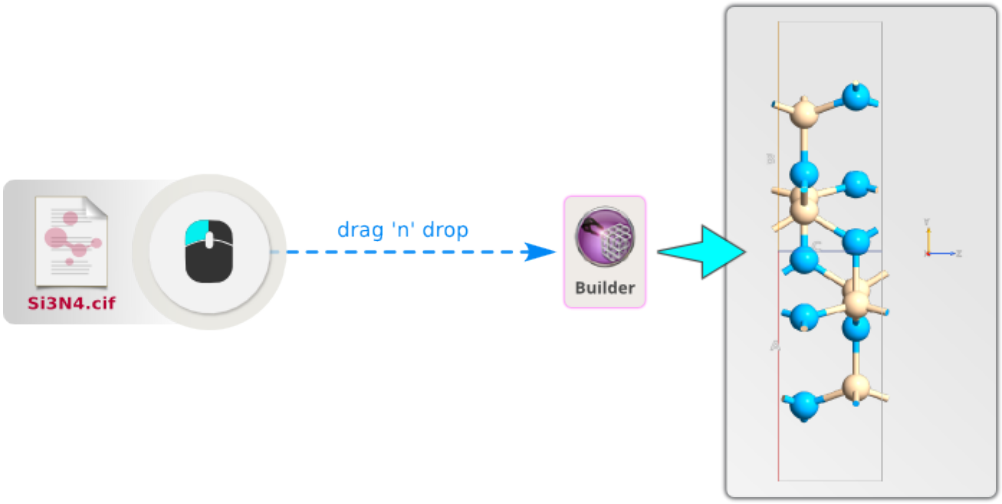
Alternatively, you can use the , and enter si3n4 in the text search field.

In either case, be sure to download the structure with space group P 63/m. Rename the downloaded file to `Si3N4.cif`, to make it easier to identify it.

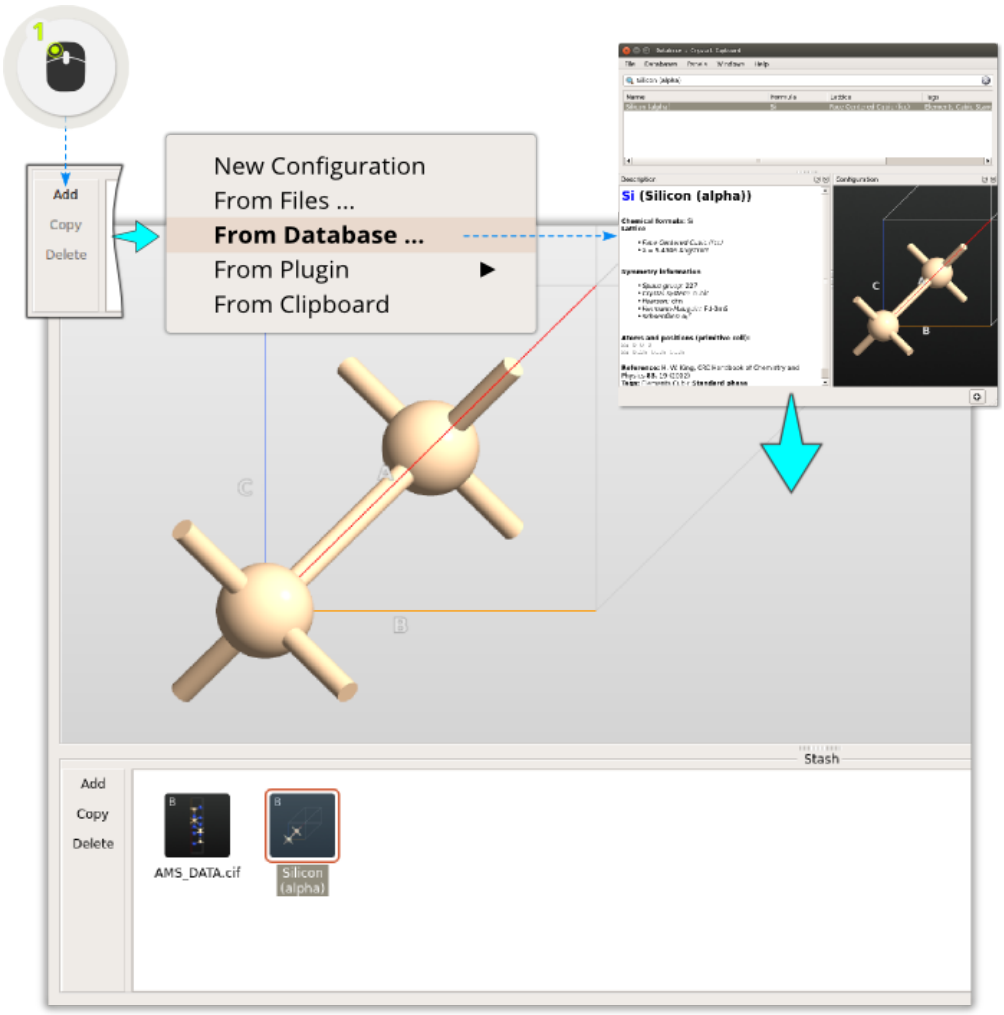

Crystallography Open Database
Search
COD Home
Home
What's new?
Accessing COD Data
Browse
Search
Search by structural formula
Add Your Data
Deposit your data
Manage depositions
Manage/release prepublications
Documentation
COD Wiki
Obtaining COD
Querying COD
Citing COD
COD Mirrors
Advices to donators
Useful links
(For more information on search see the hints and tips)
Search by COD ID:
Search
OpenBabel FastSearch:
Enter SMILES:
Search
Note: substructure search by SMILES is currently available in a subset of COD containing 70 000 structures.
text (1 or 2 words)
si3n4
journal
year
volume
issue
Z (min, max)
Z' (min, max)
1 to 8 elements
NOT these elements
volume min and max
number of distinct elements min and max
filters
has Fobs
Reset
Send
download the .cif file
remember to rename the .cif file
Si3N4.cif

Preparations: Two crystals

•**Step 1:** In QuantumATK, open the Builder and click *Add* → *Add From Files* to import the downloaded CIF file (you can also just drop the file on the Builder). The orientation of the crystal is already appropriate, with the Z axis pointing along 001 (the crystal C axis), as appropriate for the Interface Builder.



•**Step 2:** Next, click *Add* → *Add From Databases* in the Builder, and locate **Silicon (alpha)** by for instance entering silicon in the search filter box. Click the icon in the lower right-hand corner to add the Si crystal to the Builder Stash.

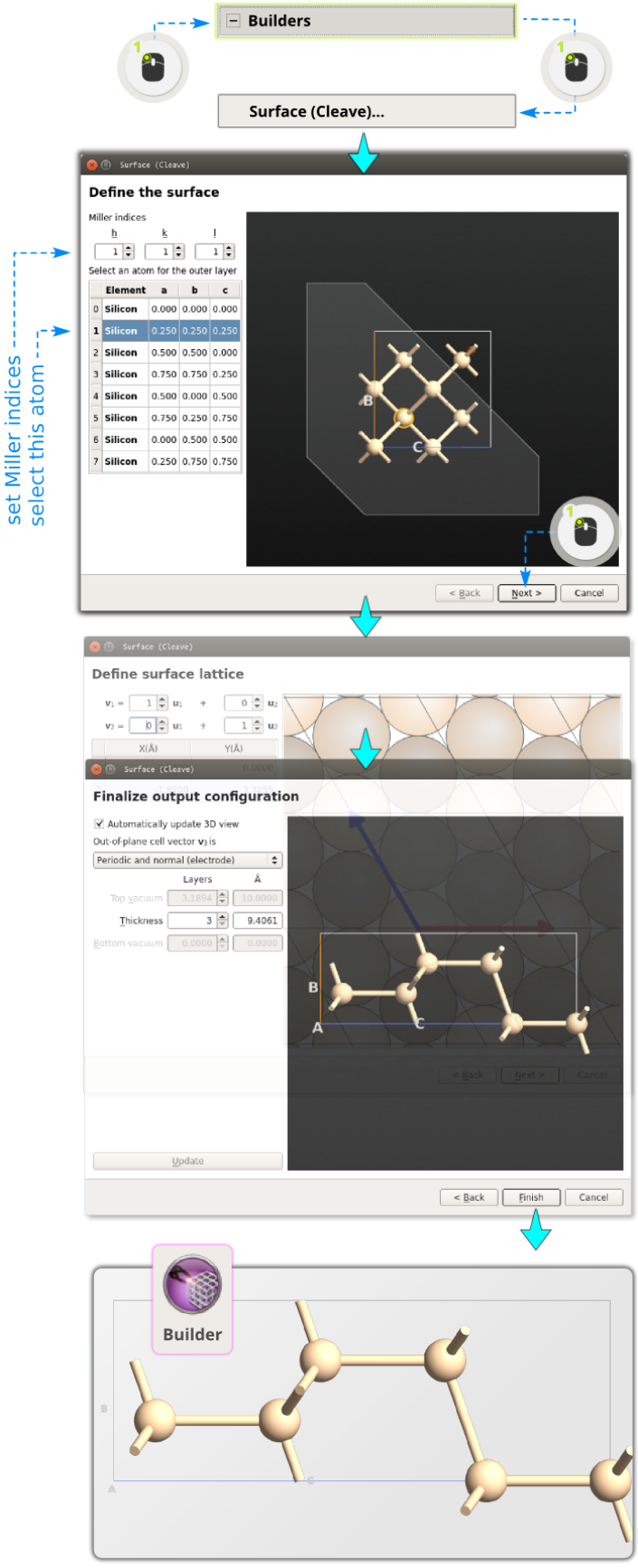


•**Step 3:** The Si crystal needs to be **cleaved** to create a (111) interface:

–Open *Builders* → *Surface (Cleave)* in the right-hand plugin panel. Then enter Miller indices **h=1, k=1, l=1**. This will define the surface C axis, which will be set parallel to Z.

–Additionally, the surface model requires that we have a long bond (along C) at the terminating Si interface, and therefore you should select the atom with index 1 in the list; this atom will define the outermost surface layer.

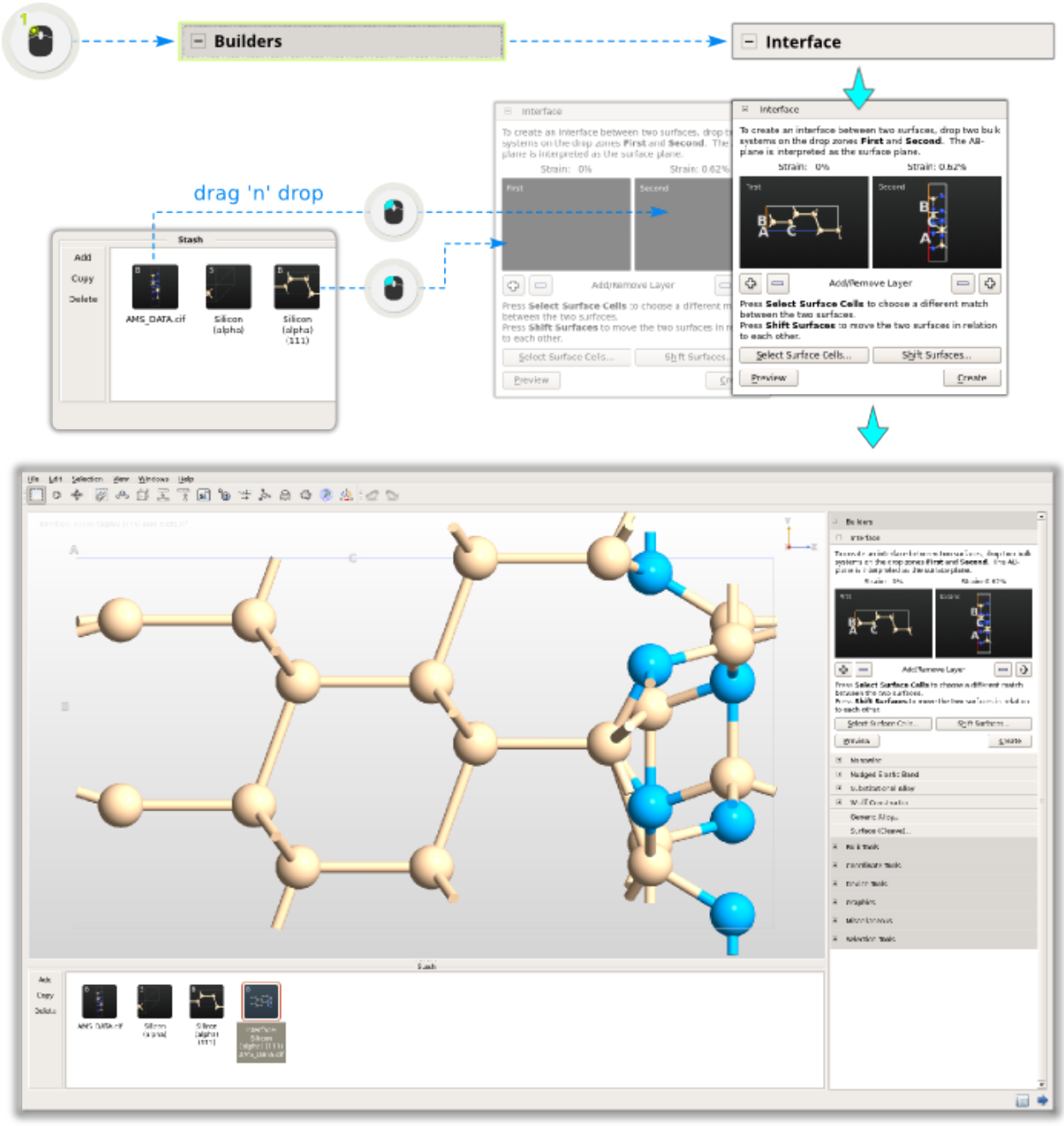
–Click *Next* → *Next* → *Finish* (the additional options in the Cleaver are not relevant for the present case).


Builders
Surface (Cleave)...
Define the surface
Miller indices
h
k
l
Select an atom for the outer layer
Element
a
b
c
Silicon
set Miller indices
select this atom
< Back
Next >
Cancel
Define surface lattice
Finalize output configuration
Automatically update 3D view
Out-of-plane cell vector v3 is
Periodic and normal (electrode)
Layers
Å
Top vacuum
Thickness
Bottom vacuum
Update
Finish
Builder

You now have the two building blocks - its time to put them together.

Building the interface

•**Step 4:** As you will see, the Interface Builder automatically figures out that it needs to repeat the hexagonal Si(111) surface cell 2x2 in the AB plane to fit the hexagonal Si_3N_4 cell. Once it has done so, the strain is extremely small (less than 1%), which of course is why this is an interesting interface. Therefore, no adjustment is needed of the surface cells, but for a general case you can click Select Surface Cells to explore possible choices.

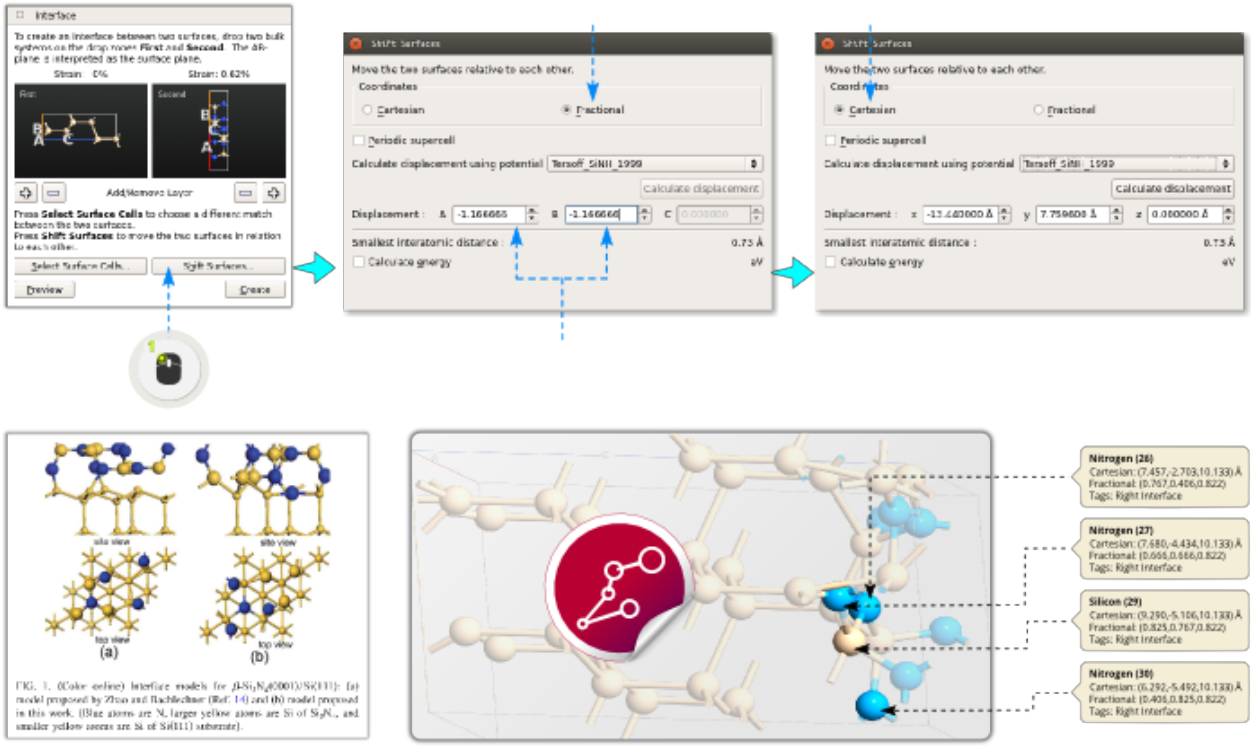


•**Step 5:** The two surfaces do however need to be shifted with respect to each other in the A/B plane. To achieve this, click **Shift Surfaces**, switch to fractional coordinates, and enter -0.16666666 for both A and B. The preview of the model will update right away in the Builder 3D window:

Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

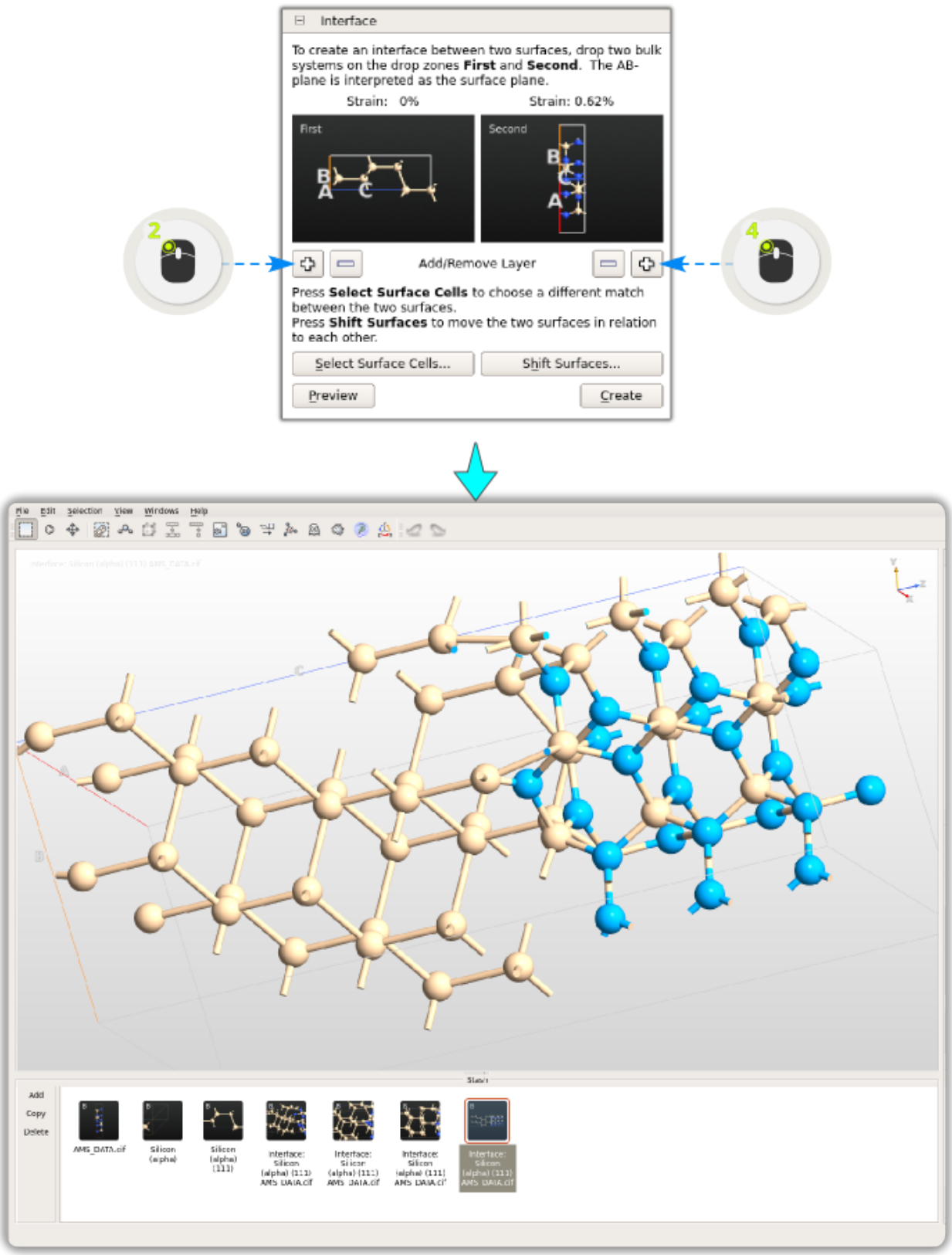
Tip: Looking at the article, you will realize that the Si triangle in the bottom Si_3N_4 layer (the layer closest to the interface, atoms 26, 29, 30 (hold the mouse over the atoms to see the numbers in the tooltip) should be ligned with the last Si(111) atom, sitting on the long bond along C, such that they form a tetrahedron. This arrangement means that the Nth atom (atom number 27) which is currently located in the middle of that triangle, at fractional position (2/3, 2/3), should sit right above that Si atom, which has coordinate (1/2, 1/2). Therefore, a surface shift of (-1/6, -1/6) is needed.

Dont close the Shift Surfaces widget yet. As you can see, the two surfaces are also a bit too close to each other. A suitable shift is 0.7 Angstroms (you need to change back to Cartesian coordinates for this one), which gives similar Si-Si and Si-N bond lengths across the interface as in the two respective bulk materials



Close the Shift Surfaces dialog.

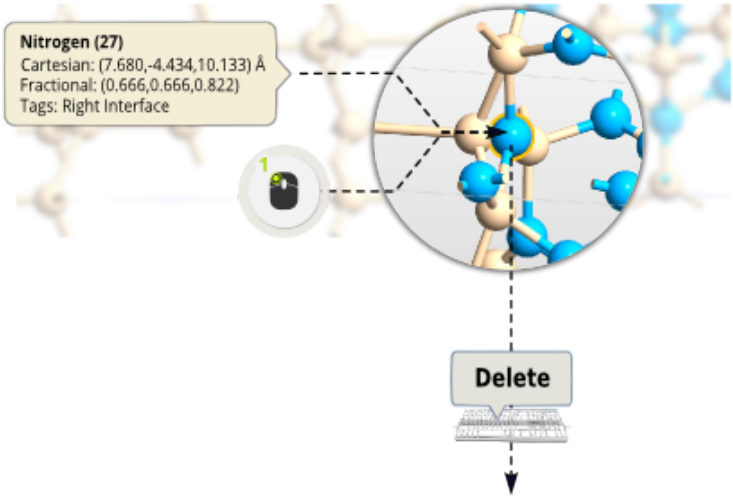
•**Step 6: Add two more layers** of Si(111) by clicking twice for Add/Remove layer under the First window. Similarly, for the Si_3N_4 **add at least four layers** (it must be an even number of times), depending on how thick you wish to make the Si_3N_4.


Interface
To create an interface between two surfaces, drop two bulk systems on the drop zones First and Second. The AB-plane is interpreted as the surface plane.
Strain: 0%
Strain: 0.62%
First
Second
B
A
C
2
4
Add/Remove Layer
Press Select Surface Cells to choose a different match between the two surfaces.
Press Shift Surfaces to move the two surfaces in relation to each other.
Select Surface Cells...
Shift Surfaces...
Preview
Create
File Edit Selection View Windows Help
Add
Copy
Delete

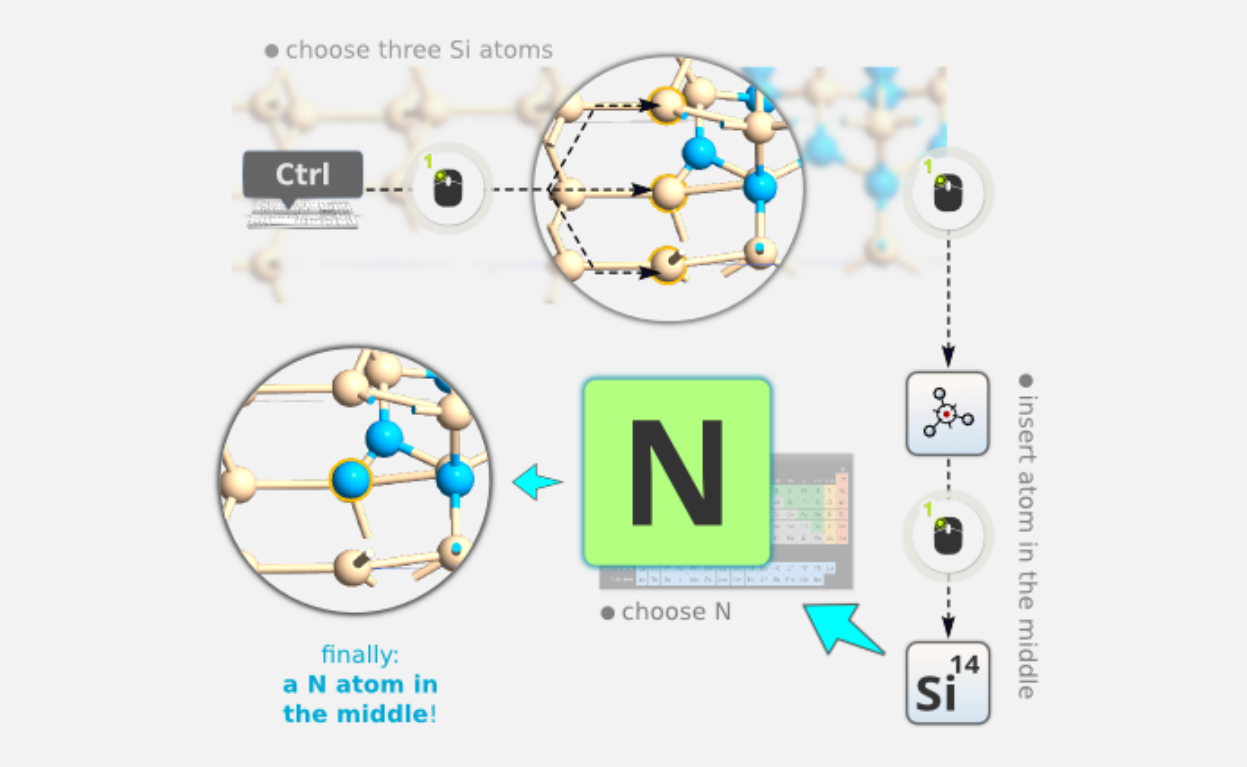
Final adjustment

You are not quite done yet. The crucial observation of Yang **et al.** was that **one N atom in the interface needs to be moved**, in order for all bonds to be saturated. The atom in question is none other than the one you used for the surface shift, sitting in the middle of the Si triangle (atom number 27).

•**Step 7:** Delete this atom by clicking it to select it, and press Del on the keyboard:



•**Step 8:** Insert a **new atom in the middle of the Si triangle in the top layer of the Si (111)**, by selecting the three atoms (click the first one, then hold down Ctrl when clicking the two others; the atoms in questions are numbers 28, 29, 30), and clicking the Centroid plugin in the left-hand side toolbar. The inserted atom has the correct position, but is Si; **change it to N** by using the Periodic Table tool :



At this point you have a single interface between Si(111) and -Si_3N_4(0001), and you may proceed to the next section.

Doubling down: Buried layer model

Tip: One can also take the procedure one step further and create a thin layer of -Si_3N_4(0001) embedded in Si(111), i.e. a double interface model. This is fairly easily acieved by mirroring the single interface.

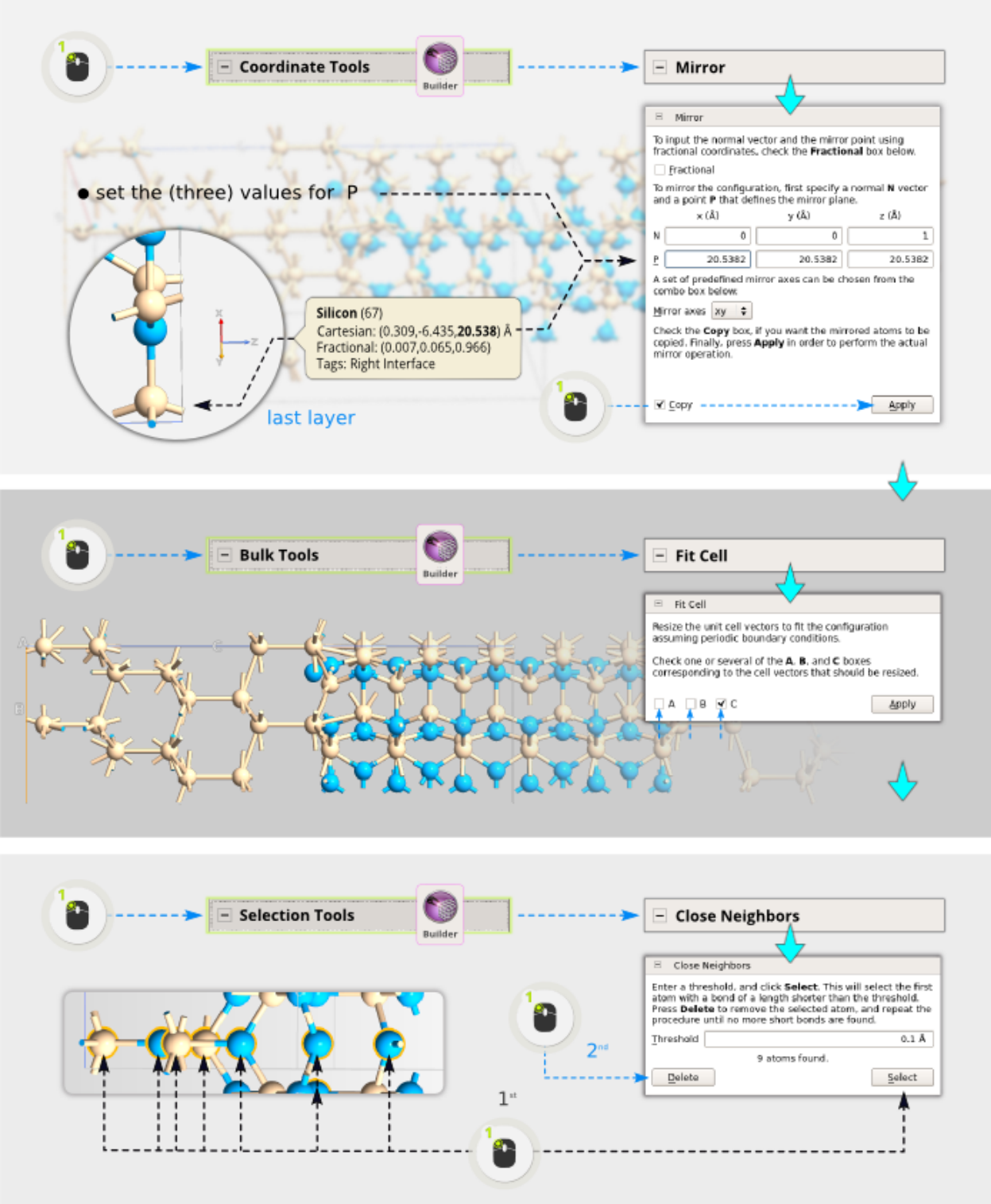
•**Step 9:** Depending on how many Si_3N_4 layers you added, the size of the current model will be different.

1. Place the mouse over the last layer in the C direction and make a note of the Z-coordinate of these atoms. In our example here (for Si_3N_4) the value is 20.5382 Angstroms.

2. Open *Coordinate Tools* → *Mirror*. The Z direction is already selected for the normal N, but to get a correctly mirror-symmetric configuration you should place the point P in the last layer. Therefore, enter the value from the previous step into the third box for P, then tick Copy, and click Apply.

3. Next, adjust the cell by using *Bulk Tools* → *Fit cell*. Before clicking Apply, make sure only C is ticked, you dont want to change anything in A or B.

4. In the reference layer used for the mirror operation, all atoms are now doubled. Therefore, as the final touch, use *Selection Tools* → *Close Neighbors* to select all overlapping atoms and delete them.


Coordinate Tools
Builder
Mirror
Mirror
To input the normal vector and the mirror point using fractional coordinates, check the Fractional box below.
Fractional
To mirror the configuration, first specify a normal N vector and a point P that defines the mirror plane.
x (Å)
y (Å)
z (Å)
N
0
0
1
P
20.5382
20.5382
20.5382
A set of predefined mirror axes can be chosen from the combo box below.
Mirror axes
xy
Check the Copy box, if you want the mirrored atoms to be copied. Finally, press Apply in order to perform the actual mirror operation.
Copy
Apply
set the (three) values for P
Silicon (67)
Cartesian: (0.309,-6.435,20.538) Å
Fractional: (0.007,0.065,0.966)
Tags: Right Interface
last layer
Bulk Tools
Builder
Fit Cell
Fit Cell
Resize the unit cell vectors to fit the configuration assuming periodic boundary conditions.
Check one or several of the A, B, and C boxes corresponding to the cell vectors that should be resized.
A
B
C
Apply
Selection Tools
Builder
Close Neighbors
Close Neighbors
Enter a threshold, and click Select. This will select the first atom with a bond of a length shorter than the threshold. Press Delete to remove the selected atom, and repeat the procedure until no more short bonds are found.
Threshold
0.1 Å
9 atoms found.
Delete
Select
2nd
1st

Interface as a device model

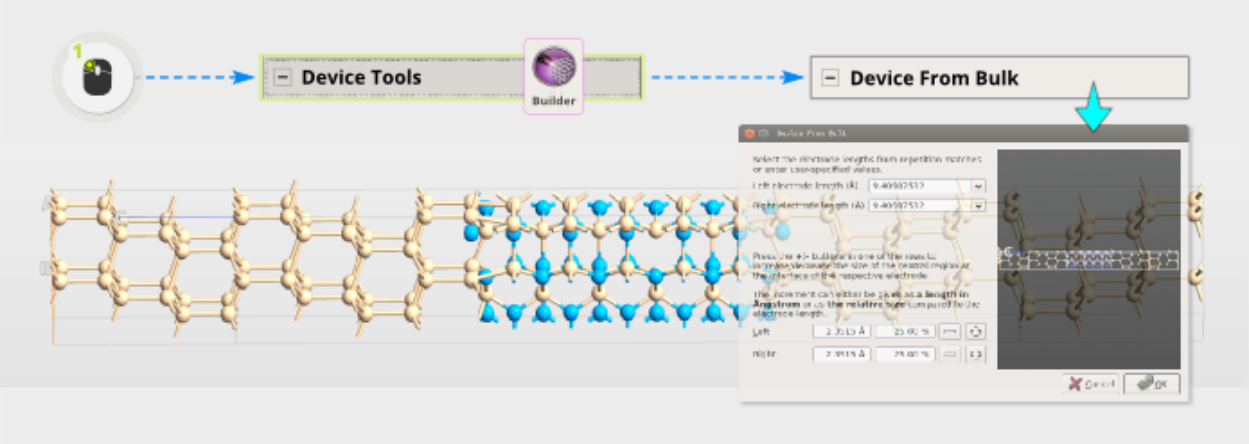
Irrespective of whether you chose the single or double interface model, the geometry is still represented as a periodic structure. In QuantumATK, you can however compute **electronically open systems**, as opposed to the periodic models used in other materials science software packages. This means you can compute the properties of a single interface (or two), rather than a periodically repeated stack of different materials.

Open boundary condition calculations are in QuantumATK carried out using so-called **device configurations**. To convert the current structure into a device:

•Open *Device Tools* → *Device From Bulk*.

•Since you added a few Si (111) layers, the tool can automatically find the electrode lengths, since there is a repetitive pattern.

•Click OK to create the device.



Note: As a final note, although it will not be covered here, it should be pointed out that the geometry needs to be optimized, especially the Si atoms in the interface. If you do that, you will see the formation of fully saturated bonds for all Si atoms. To see it in the unrelaxed structure, send the structure to the Viewer, and increase the range for bonds to be plotted to 1.3 Angstroms, and you will see that all Si atoms have 4 bonds.

References

3.4.7 Determination of low strain interfaces via geometric matching

Version: 2017.1

This tutorial introduces the new **GeneralizedLatticeMatch** method for combining two bulk crystals into an interface.

On many occasions, one is interested in creating a realistic interface between two materials, even without having precise structural informations on the two surfaces that form the interface. The **GeneralizedLatticeMatch** method facilitates this process, by automatically finding all the possible interface supercells between the two crystals based only on their bulk crystalline structure. The method is an optimized version of the algortihm described in[1]. Compared to the lattice-matching method used in the **Interface Builder**, described in *Interface Builder*, the present method is unbiased as it considers all the possible surfaces of the two crystals forming the interface.

[1] Line Jelver, PeterăMahler Larsen, Daniele Stradi, Kurt Stokbro, and KarstenăWedel Jacobsen. Determination of low-strain interfaces via geometric matching. *Phys. Rev. B*, 96:085306, Aug 2017. doi:10.1103/PhysRevB.96.085306.

A number of structural parameters are calculated for each interface, which allows one to easily analyze the matching pattern and choose the most appropriate interface to be used for further studies. The structure of the desired interface can then be easily created using the **Interface Builder** implemented in **QuantumATK**.

Method description

The following describes in the main steps of the algortihm used in the **GeneralizedLatticeMethod** method, which is very similar to that used in the **Interface Builder** (see *Interface Builder*).

1. Initially, the possible surface vectors $[\mathbf{a}_1, \mathbf{a}_2]$ of the first of the two materials forming the interface are constructed, starting from a linear combination of the Bravais vectors of the primitive cell:

$$\mathbf{a}_1 = \sum_{i=1}^{3} c_i \mathbf{u}_i \quad c_i \in \mathbb{Z},$$
$$\mathbf{a}_2 = \sum_{i=1}^{3} c'_i \mathbf{u}_i \quad c'_i \in \mathbb{Z},$$

and by a subsequent projection of the resulting vectors from $\mathbf{R}^3$ to $\mathbf{R}^2$. A similar procedure is applied to construct the surface vectors $[\mathbf{b}_1, \mathbf{b}_2]$ of the second surface. As it will be shown in the Section *Input and output description*, the number of generated surface cells can be limited by specifying:

- The maximum value of the Miller indexes $[h, k, l]$ that define each surface;
- The maximum length of the lattice vectors $[\mathbf{a}_1, \mathbf{a}_2]$ and $[\mathbf{b}_1, \mathbf{b}_2]$.

2. Each couple of surface cells is matched by using the same lattice-match method as used in the **Interface Builder**, and described in the technical note *Interface Builder*.

3. The average strain is then calculated as:

$$\bar{\varepsilon} = \sqrt{\frac{\varepsilon_{11}^2 + \varepsilon_{22}^2 + \varepsilon_{11}\varepsilon_{22} + \varepsilon_{12}^2}{4}}$$

where ϵ_{11}, ϵ_{22} and ϵ_{12} are the components of the 2D strain tensor, as defined in the technical note *Interface Builder*.

Note: Notice that this definition of the average strain differs from that used in the *Interface Builder*. The present definition of strain is more appropriate for the present method as it is an invariant of the strain tensor.

Input and output description

In order to use the GeneralizedLatticeMatch method one can set up a simple script as follows:

```
# Read the BulkConfiguration of the primitive cell of the
# first material
configuration_1 = nlread('InAs.py',BulkConfiguration)[-1]
# Read the BulkConfiguration of the primitive cell of the
# second material
configuration_2 = nlread('Al.py',BulkConfiguration)[-1]

# Run the GeneralizedLatticeMatch method
generalized_lattice_match = GeneralizedLatticeMatch(
                            configuration_1,
                            configuration_2,
                            max_strain=0.02,
                            maximum_miller_index=3,
                            longest_surface_lattice_vector=50*Angstrom,
                            max_surface_area=200.0*Angstrom**2,
                            user_given_miller_index=None
                            )
```

The script reads two input files, each one containing a BulkConfiguration of the bulk primitive cell of one of the two materials that form the interface. Then, it applies the GeneralizedLatticeMatch method on these two structures. A number of **input parameters** can be set to control the precision and the extent of the search for possible interface supercells. The full list of input parameters is:

- `configuration_1`: BulkConfiguration of the bulk primitive cell of the first material.
- `configuration_2`: BulkConfiguration of the bulk primitive cell of the second material.
- `max_strain`: Maximum strain applied to each of the two surfaces.
- `maximum_miller_index`: Maximum value of the Miller indexes $[h, k, l]$ that define each of the two surfaces.
- `longest_surface_lattice_vector`: Maximum length of each surface lattice vector $[\mathbf{a}_1, \mathbf{a}_2]$ and $[\mathbf{b}_1, \mathbf{b}_2]$.
- `max_surface_area`: Maximum value of the surface area of the interface supercell.
- `user_given_miller_index`: Predefined Miller indexes of the surface of `configuration_1`.

Note: When the parameter `user_given_miller_index` is set to a value different than `None`, the search for the possible surfaces is restricted to the second material, whereas the surface of the first material is kept fixed. This option will be used in *Example 2: Lattice match between a bulk system with a predefined surface*.

After the possible matches are calculated, the matching results are printed directly in the QuantumATK log file of the calculation. The matches will be listed in order of increasing strain, together with a number of output parameters that characterize each match.

```
+-----------------------------------------------------------------------------------------
↪+
|    A              B        Strain     Atoms    Area    Aspect    Angle    Rotation    ␣
↪|
+-----------------------------------------------------------------------------------------
↪+
[ 1  0  0] >-< [ 1  0  0]   0.000110      29     72.9      1.0      90.0        0.0
```

(continues on next page)

(continued from previous page)

```
[ 2  2  1] >-< [ 1  0  0]   0.000110      23    163.9      2.2     153.4       26.6
[ 3  3  2] >-< [ 1  1  1]   0.004030      28    170.9      1.2     162.6       60.3
```

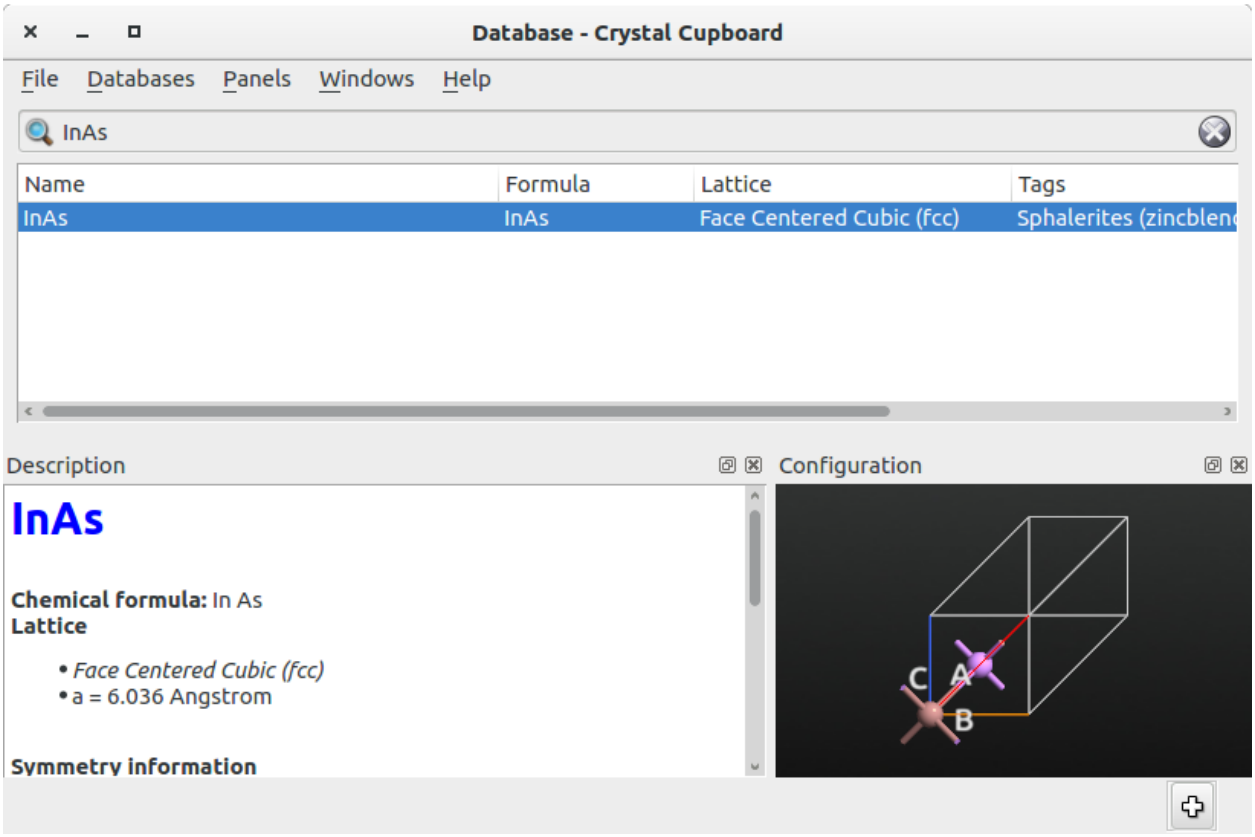
The output parameters are:

- `A`: Miller indexes $[h, k, l]$ of the first surface.
- `B`: Miller indexes $[h, k, l]$ of the second surface.
- `Strain`: Maximum strain of the two surfaces.
- `Atoms`: Total number of atoms in the interface supercell.
- `Area`: Surface area of the interface supercell.
- `Aspect`: Aspect ratio between two surface vectors of the interface supercell. The ratio is calculated by taking the largest vector at the numerator.
- `Angle`: Angle between the two vectors of the interface supercell.
- `Rotation`: Rotation between the two surfaces in the interface supercell.

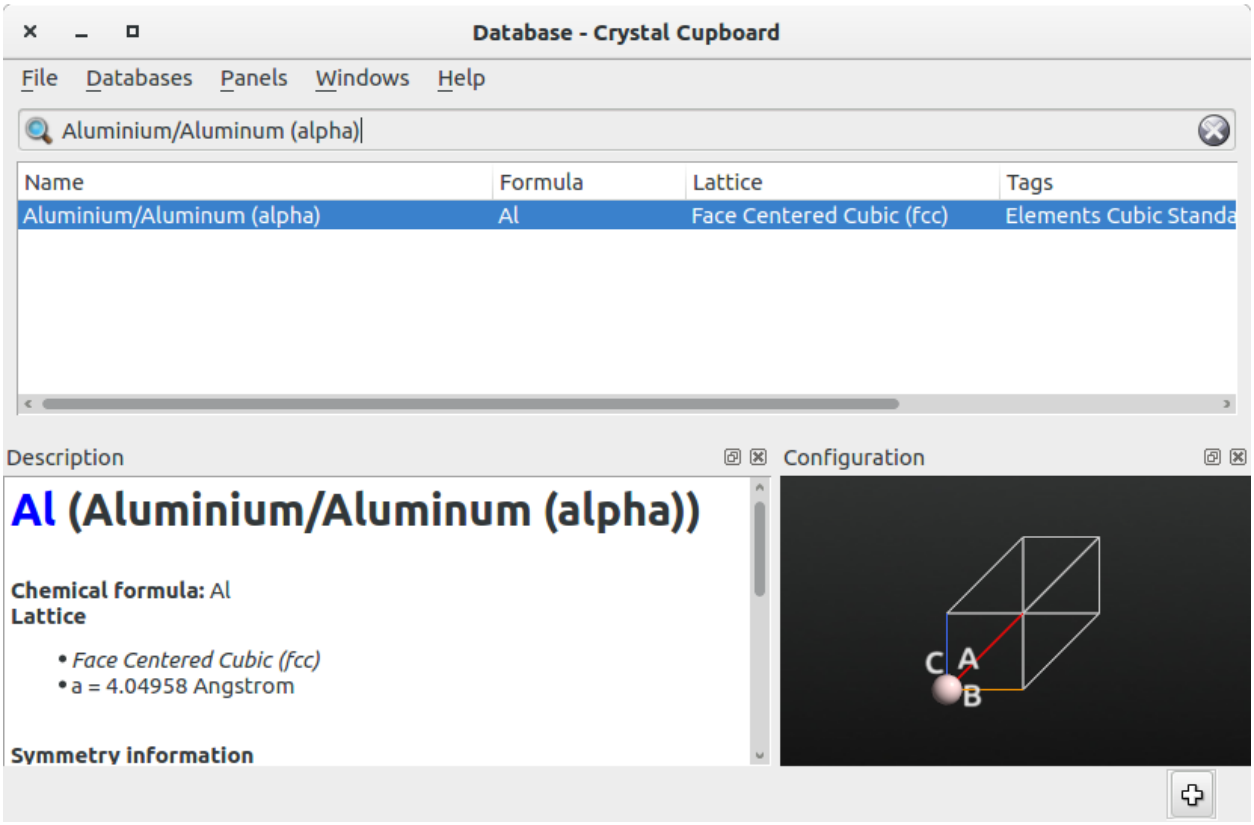
Example 1: Lattice match between two bulk systems

In this first example, you will calculate the possible matches between indium arsenide and aluminum.

To obtain the structure file containing the BulkConfiguration of bulk InAs, open **QuantumATK**, go to the **Builder** and click on *Add* → *From Database*. In the **Database**, select the primitive cell of bulk InAs shown in the figure below, add it to the **Stash** by clicking on the button.



Repeat the same procedure to add to the **Stash** the structure of the primitive cell of bulk Al, which is shown in the figure below.



In the **Stash**, save the two configurations by right-clicking on each structure and selecting *Save as*. Alternatively, you can download the two files from here: `InAs.py`, `Al.py`.

Set up the script for the GeneralizedLatticeMatch method as shown below:

```
# Read the BulkConfiguration of the primitive cell of the
# first material
configuration_1 = nlread('InAs.py',BulkConfiguration)[-1]
# Read the BulkConfiguration of the primitive cell of the
# second material
configuration_2 = nlread('Al.py',BulkConfiguration)[-1]

# Run the GeneralizedLatticeMatch method
generalized_lattice_match = GeneralizedLatticeMatch(
                            configuration_1,
                            configuration_2,
                            max_strain=0.02,
                            maximum_miller_index=3,
                            longest_surface_lattice_vector=50*Angstrom,
                            max_surface_area=200.0*Angstrom**2,
                            user_given_miller_index=None
                            )
```

Alternatively, you can download the script from here: `match_InAs_Al.py`

In this example, the scan will be performed by considering surfaces with a maximum strain of 0.02, with Miller indexes $h, k, l \leq 3$, with a maximum length of the lattice vectors of 50 Å, and with a upper threshold for the surface area of 200 Å^2.

Run the script in the terminal as `atkpython match_InAs_Al.py > match_InAs_Al.log`. The output will look as shown below. There are several possible matches, so for briefness only the first 20 matches are shown.

```
+-------------------------------------------------------------------------------+
|                                                                               |
| Atomistix ToolKit 2017.1 [Build ce08f12]                                      |
|                                                                               |
+-------------------------------------------------------------------------------+

                         |-------------------------------------------------|
Miller planes for A    : ==================================================

                         |-------------------------------------------------|
Miller planes for B    : ==================================================

                         |-------------------------------------------------|
Matching Miller planes : ==================================================
+-------------------------------------------------------------------------------------------
↪+
|    A              B        Strain     Atoms    Area    Aspect    Angle    Rotation    ␣
↪|
+-------------------------------------------------------------------------------------------
↪+
[ 1  0  0] >-< [ 1  0  0]   0.000110      29     72.9      1.0      90.0        0.0
[ 2  2  1] >-< [ 1  0  0]   0.000110      23    163.9      2.2     153.4       26.6
[ 3  3  2] >-< [ 1  1  1]   0.004030      28    170.9      1.2     162.6       60.3
[ 3  1  1] >-< [ 2  1  1]   0.004030      10    120.8      1.7      90.0        0.0
[ 3  1  1] >-< [ 3  1  1]   0.004250      16    120.8      1.7     106.8       33.6
[ 3  3  1] >-< [ 3  3  1]   0.004250      16    158.8      2.2     102.9        0.0
[ 3  3  1] >-< [ 3  3  1]   0.005470      13    158.8      2.2     102.9        0.0
[ 3  1  1] >-< [ 3  1  1]   0.005470      13    120.8      1.7     106.8       33.6
[ 2  1  1] >-< [ 2  1  1]   0.005470      17    178.5      2.4      90.0        0.0
[ 2  1  0] >-< [ 2  1  0]   0.005470      13    162.9      1.2     114.1        0.0
[ 1  1  1] >-< [ 1  1  1]   0.005470      13     63.1      1.0     120.0       60.0
[ 1  1  0] >-< [ 1  1  0]   0.005470      17    103.0      1.4      90.0        0.0
[ 1  0  0] >-< [ 1  0  0]   0.005470      13     72.9      1.0      90.0        0.0
[ 1  0  0] >-< [ 2  2  1]   0.005470       7     72.9      1.0      90.0        0.0
[ 3  2  2] >-< [ 1  1  1]   0.005940      23    150.2      1.2     157.6       60.5
[ 1  0  0] >-< [ 2  1  1]   0.006130      19    182.2      3.9      39.8        0.6
[ 3  1  1] >-< [ 1  1  0]   0.007160      18    151.0      1.7      73.2       22.6
[ 3  1  1] >-< [ 1  0  0]   0.007730      19    120.8      1.7      90.0        0.0
[ 3  2  2] >-< [ 3  1  1]   0.008250      13    150.2      2.3     115.9       35.2
[ 1  0  0] >-< [ 1  1  0]   0.008420      28    200.4      2.2      79.7       38.2
```

Notice how the match with the least strain is that between the $\langle 100 \rangle$ surface of InAs and the $\langle 100 \rangle$ surface of Al.

Example 2: Lattice match between a bulk system with a predefined surface

In some cases, one could be interested in finding the possible matches of a bulk material to a given surface. In this second example, you will investigate how to find the possible matches of bulk aluminium to the InAs(111) surface. InAs nanowires with ⟨111⟩-oriented facets have been synthesized in Ref.[2], and it has been demonstrated experimentally that ⟨111⟩-oriented Al layers can be grown on this surface.

Use the same structure files as in *Example 1: Lattice match between two bulk systems*, and setup the script for the GeneralizedLatticeMatch calculation as follows:

```
# Read the BulkConfiguration of the primitive cell of the
# first material
configuration_1 = nlread('InAs.py',BulkConfiguration)[-1]
# Read the BulkConfiguration of the primitive cell of the
# second material
configuration_2 = nlread('Al.py',BulkConfiguration)[-1]

# Run the GeneralizedLatticeMatch method
generalized_lattice_match = GeneralizedLatticeMatch(
                            configuration_1,
                            configuration_2,
                            max_strain=0.02,
                            maximum_miller_index=3,
                            longest_surface_lattice_vector=50*Angstrom,
                            max_surface_area=200.0*Angstrom**2,
                            user_given_miller_index=(1,1,1)
                            )
```

Notice how the `user_given_miller_index` option, highlighted in yellow, has now been set to `user_given_miller_index=(1,1,1)`. Therefore, in this case only the matches of bulk Al to the InAs(111) surface will be considered. You can also download the script from here: `match_InAs111_Al.py`

Run the script in the terminal as `atkpython match_InAs111_Al.py > match_InAs111_Al.log`. The output will look as shown below. Notice how there are fewer possible matches than those in the output of *Example 1: Lattice match between two bulk systems*. This is because only one surface is now considered for the material denoted `A`.

```
+--------------------------------------------------------------------------------+
|                                                                                |
| Atomistix ToolKit 2017.1 [Build ce08f12]                                       |
|                                                                                |
+--------------------------------------------------------------------------------+

                         |-----------------------------------------------------|
Miller planes for B    : =====================================================

                         |-----------------------------------------------------|
Matching Miller planes : =====================================================
+-------------------------------------------------------------------------------------------
↪+
|    A             B        Strain     Atoms    Area    Aspect    Angle    Rotation    ␣
↪|
+-------------------------------------------------------------------------------------------
```

(continues on next page)

[2] P.ăKrogstrup, N.ăL.ăB. Ziino, W.ăChang, S.ăM. Albrecht, M.ăH. Madsen, E.ăJohnson, J.ăNygård, C.ăM. Marcus, and T.ăS. Jespersen. Epitaxy of semiconductor-superconductor nanowires. *Nature Materials*, 14:400–406, 2015. doi:10.1038/nmat4176.

(continued from previous page)

```
↪+
[ 1  1  1] >-< [ 1  1  1]   0.005470      13     63.1      1.0     120.0       60.0
[ 1  1  1] >-< [ 3  3  1]   0.010100      17    142.0      1.2      46.1       18.4
[ 1  1  1] >-< [ 2  1  0]   0.010710      15    110.4      4.0      30.0        5.9
[ 1  1  1] >-< [ 3  1  1]   0.011070      26    110.4      2.6      19.1       77.9
[ 1  1  1] >-< [ 3  1  1]   0.012770      15    110.4      2.6      19.1       77.9
[ 1  1  1] >-< [ 1  1  0]   0.013260      24    126.2      2.6      40.9       12.1
[ 1  1  1] >-< [ 1  0  0]   0.014880      23    126.2      1.7       8.2        1.1
[ 1  1  1] >-< [ 2  2  1]   0.014880      13    126.2      2.3      10.9       31.1
[ 1  1  1] >-< [ 3  2  1]   0.016000      18    189.3      2.6      26.3       75.4
[ 1  1  1] >-< [ 3  1  2]   0.016000      18    189.3      3.5      19.1       44.6
[ 1  1  1] >-< [ 2  1  0]   0.016150      13    110.4      4.0      30.0        5.9
[ 1  1  1] >-< [ 2  1  1]   0.016420       9     78.9      2.9      30.0       20.8
[ 1  1  1] >-< [ 1  1  0]   0.016510      19    126.2      2.6      40.9       12.1
+-------------------------------------------------------------------------------------
↪+
```

In this case, the match to the InAs(111) surface with the lowest strain is that with the Al(111) surface, which is in agreement with the experimentally measured structure reported in Ref.[Page 842, 2].

References

3.4.8 Modeling metal–semiconductor contacts: The Ag–Si interface

Version: 2016.1

Downloads & Links

`device.py` `c_fit.py` `hdp.py` `hdp_plot.py` `pldos_hdp.py` `IV.py` `IV-n-log.py` `IvsT.py` `arrhenius.py` `spectral_current.py` `barrier_compare.py`

Contacts between metals and doped semiconductors are ubiquitous in modern technology and find applications in several important areas, e.g. photovoltaics. QuantumATK provides an excellent toolbox to study such interfaces, because it correctly accounts for the semiconductor band gap and doping, and describes the interface using the physically correct boundary conditions.

This case study shows how the properties of metal–semiconductor interfaces can be calculated and analyzed using QuantumATK and the NEGF (non-equilibrium Greens function) method. To this end, we will reproduce the main calculations and theoretical analyses reported in the recent publication *General atomistic approach for modeling metal–semiconductor interfaces using density functional theory and non-equilibrium Greens function*[1], which considers the Ag–Si interface. Notice, however, that the methods presented here are completely general and can be applied to any metal–semiconductor interface.

You will be guided through the following steps:

- *Creating the device*
- Calculating and plotting the *projected local density of states*
- Calculating the *ideality factor*
- Estimating and analyzing the *Schottky barrier*

[1] D.ăStradi, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. General atomistic approach for modeling metal-semiconductor interfaces using density functional theory and nonequilibrium Greens function. *Phys. Rev. B*, 93:155302, Apr 2016. doi:10.1103/PhysRevB.93.155302.

•Calculating the *spectral current*

Note: QuantumATK case studies are primarily directed at experienced users of QuantumATK. Instructions are deliberately concise in order to focus mostly on the science.

Warning: If you are running this case study with QuantumATK v2017.2 or higher, be aware that you will have to set explicitly the electronic temperature to $T = 300$ K in all the electronic structure calculations performed for either a `BulkConfiguration` or for a `DeviceConfiguration`, in order to reproduce the results presented in the case study.

Creating the device

Tip: If you wish to skip this part, you can just download the device configuration as an QuantumATK Python script: `device.py`.

The first step is to relax the bulk structures of silver and silicon. Build the structures using the database in the **Builder** and send them both to the **Scripter**, one at a time. In both scripts, add a **New Calculator** block with the following settings:

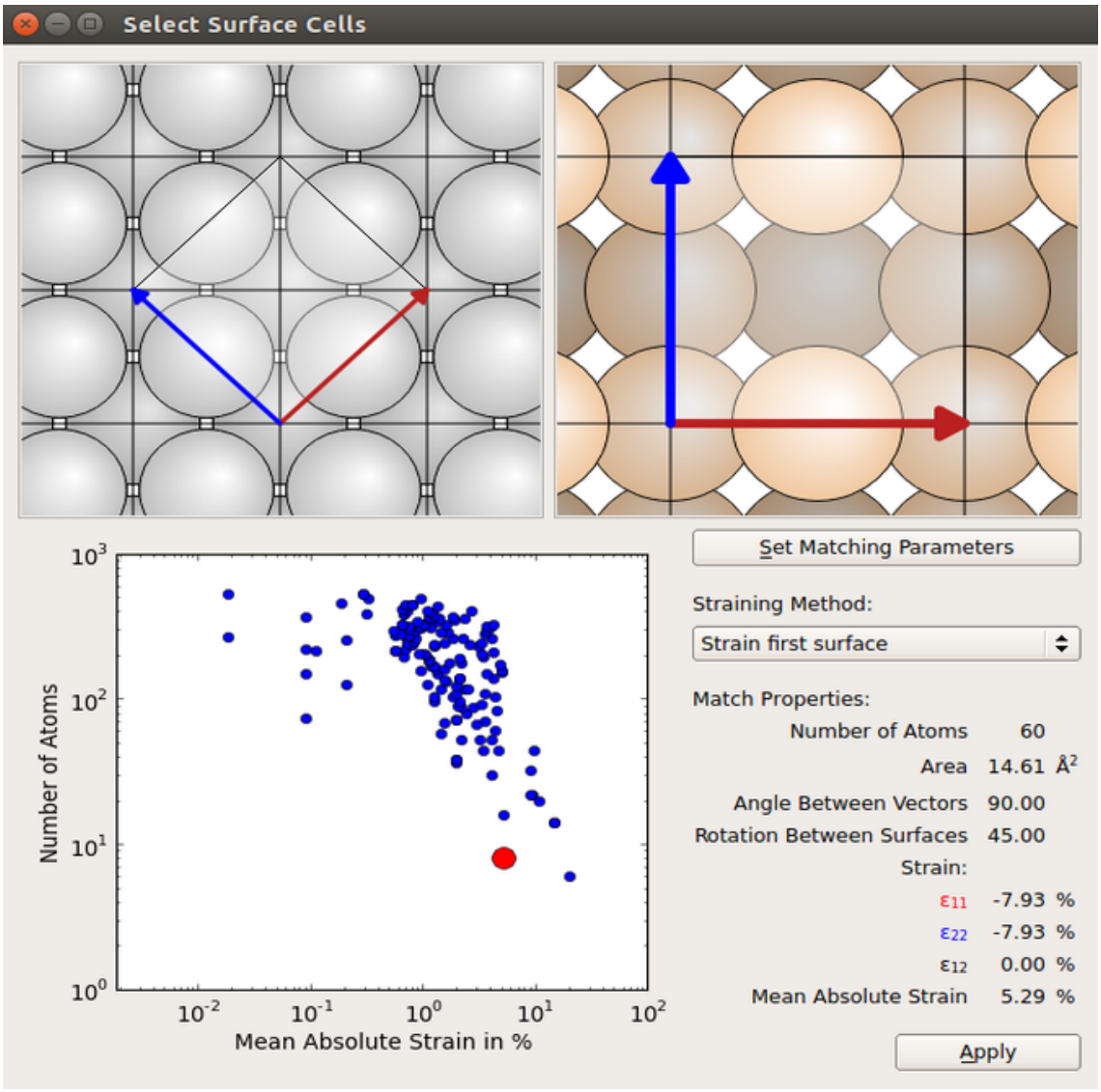
•ATK-DFT

•LDA exchange-correlation functional

•11x11x11 k-point sampling

Also add the **OptimizeGeometry** block, set the force and stress tolerances to 0.01 eV/Å and 0.001 eV/Å^3, respectively, and uncheck the *Constrain cell* box. Run both scripts to obtain the optimized structures. The calculations can easily be run on your local machine.

The Ag and Si (100) surfaces can now by created using the **Surface (Cleave)** tool in the Builder, and from these you can create the interface as explained in the tutorial *Building an interface between Ag(100) and Au(111)*. You should include 6 layers of Ag and 9 layers of Si. Choose *Strain first surface* in the **Select Surface Cells** window under *Straining method* and choose the cell shown in the figure below.

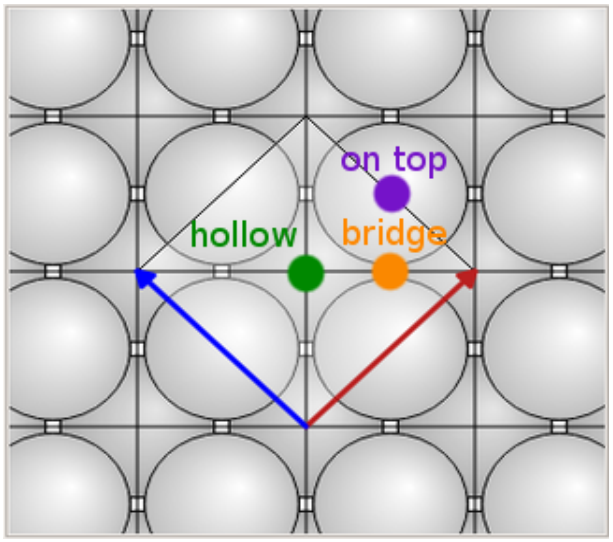


Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

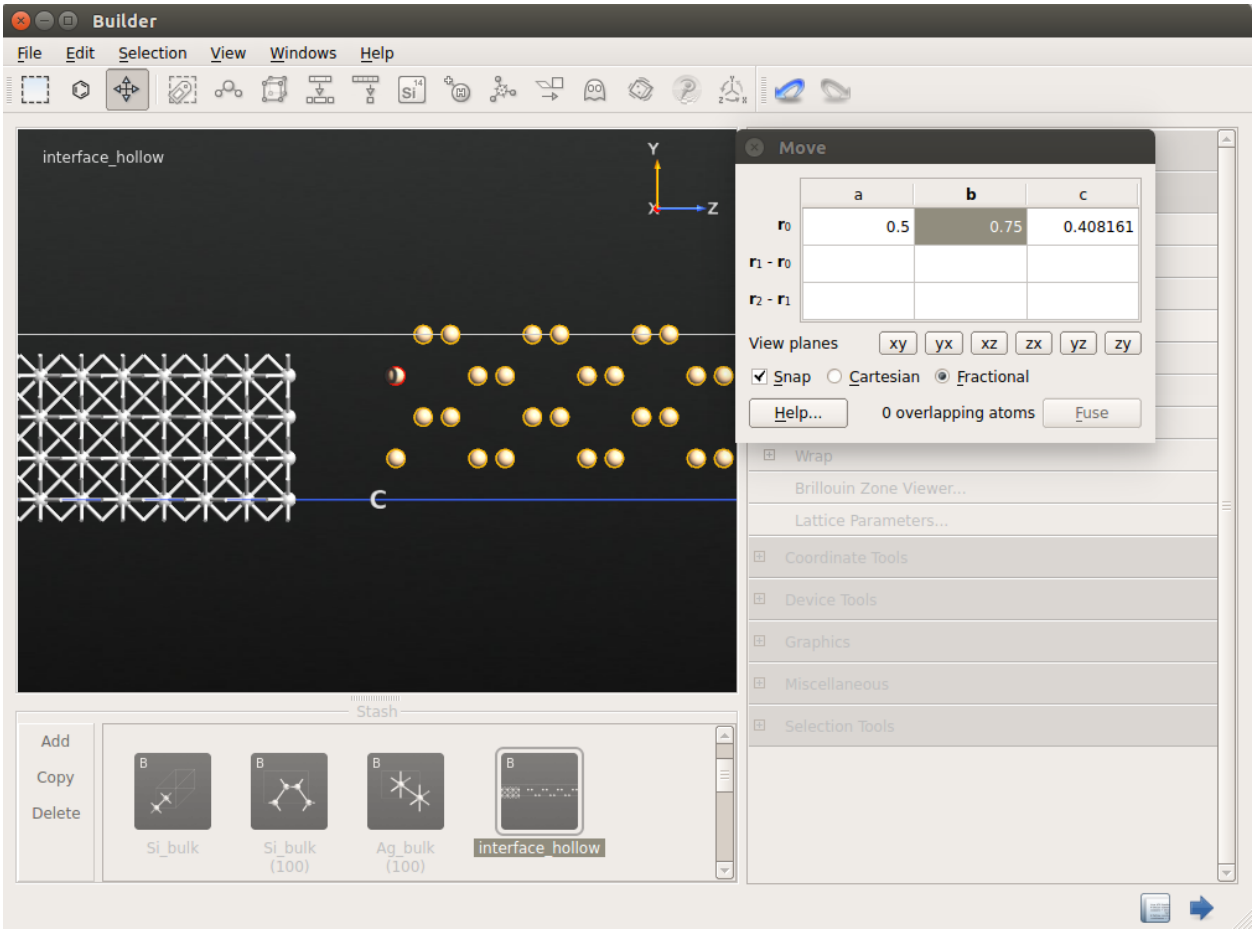
Initial guesses for the interface structure

Use the **Repeat** tool to repeat the interface 2 times along the B-direction. Next, add some vacuum around the configuration by first using the **Stretch Cell** tool to stretch the C-vector by 30%, and then use the **Center** tool to center the atoms along the C-direction only.

At the Ag–Si interface, the Si dangling bonds are initially placed near the hollow site of the Ag(100) surface. In order to find the most stable interface structure, you should investigate the on top and bridge sites as well.



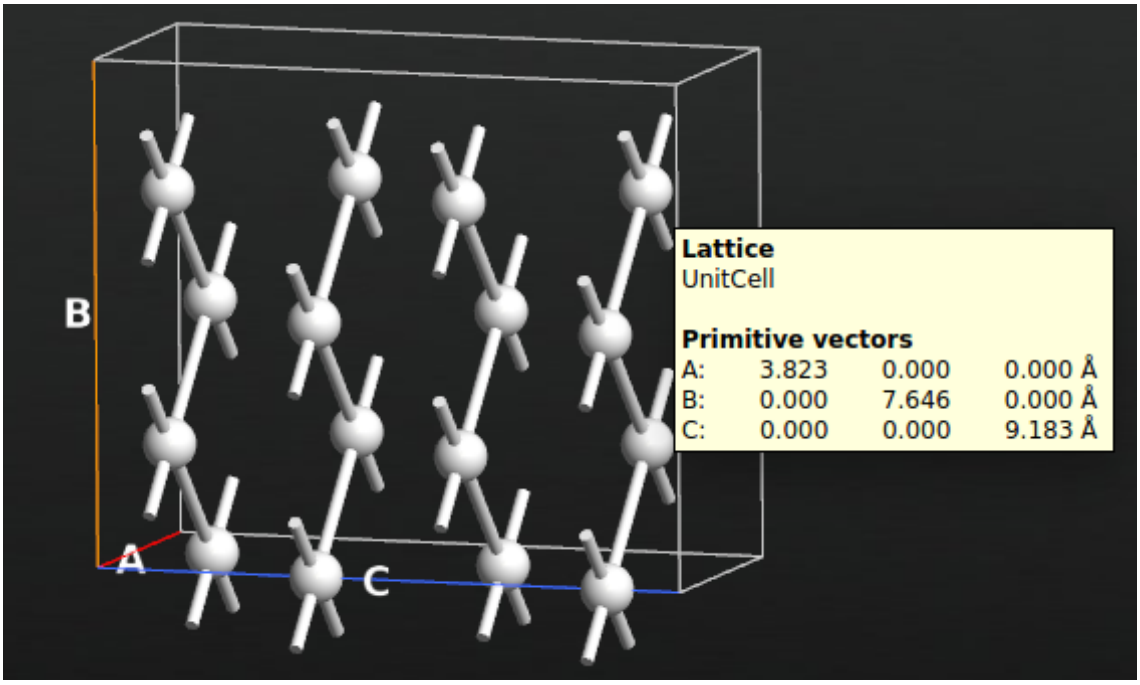
Select all the Si atoms and open the **Move** tool. Choose the topmost of the Si atoms farthest to the left as an anchor by double-clicking it. Then shift the Si(100) surface 4 Å away from the Ag(100) by increasing the c-value in the Move window, and rename the structure `interface_hollow`. Make a copy of the structure and use the **Move** tool again. This time, choose the *Fractional* option and enter a=0.5 and b=0.75 (see image below). Name this structure `interface_ontop`. Finally, make a new copy, set a=0.25 and b=0.625, and name the structure `interface_bridge`.



Note: Repeating the structure is not necessary in the general case. We do it here in order to obtain the structural model considered in[1].

Electrode relaxation

Next, you should relax the Ag electrode in the z-direction. The electrode was strained in the x and y directions when building the interface and should therefore be allowed to relax in the z direction according to the Poisson ratio of the material. You can obtain the electrode by first creating a device configuration (apply the **Device From Bulk** plugin to one of the interfaces created above), and then split it using the icon. Use an ATK-DFT calculator with LDA and a 6x3x1 k-point grid, and use the same force and stress tolerance as for the bulk relaxations above. You will find that the length of the electrode C-vector increases by 9.57%. To be consistent, you should apply the same elongation to the Ag part of the three interface configurations. Use again the **Stretch Cell** tool: Select all the Ag atoms, enter 1.0957 for the stretch along C, and choose *Selected atoms are stretched*.



Central region relaxation

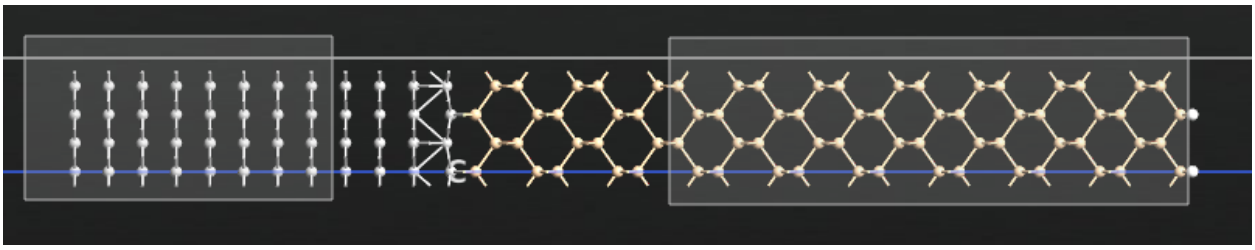
General procedures for relaxing a device configuration is explained in the tutorial *Advanced device relaxation - manual workflow*. Here, you will first relax the three interface structures as bulk configurations (i.e. relax the device central regions) in order to determine which of them is the most stable.

First, add some extra vacuum to all three interface structures by using the **Lattice Parameters** plugin to increase the C-vector length by 30%. Then use the **Center** tool to center the configurations along C only. Finally, use the **Hydrogen passivation** tool to passivate the dangling bonds on the Si(100) surface farthest from the interface. Note that this may add H atoms to the left of the silicon part as well as to the right. Any hydrogen placed in the interface should obviously be removed.

You are now ready to relax each configuration. Use the same calculator settings as for the Ag electrode relaxation, and use the following parameters in the **OptimizeGeometry** block:

- 0.02 eV/Å force tolerance.
- Do not enable stress minimization.
- Use the **Add Constraints** widget to set a *Fixed* constraint for the Ag atoms indicated in the image below, and a **Rigid** constraint for the indicated Si (and H) atoms.

Finally, add also the **TotalEnergy** analysis block to the scripts.



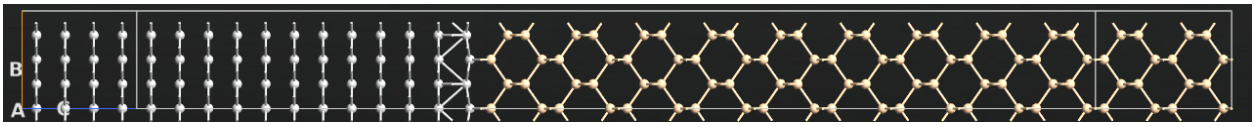
Run the calculations. They are a bit demanding and require roughly 20 hours on a 16-core node. You should find that the structure where the silicon connects to the Ag(100) hollow sites is the most stable (it has the most negative total energy, see below). In the following, only this configuration will be used.

	Total Energy (eV)
hollow	-76536.37
on top	-76535.46
bridge	-76535.40

Device relaxation

Remove the passivating hydrogen atoms from the optimized hollow interface structure, and build a device using the **Device From Bulk** plugin. This final device relaxation should use the same calculator and geometry optimization parameters as for the bulk relaxation above, except that the k-point sampling along C must be increased to 401 (use a 6x3x401 grid).

The calculation takes about 2 hours on 16 cores. If you do not intend to run the calculation, you can download the relaxed device configuration here: `device_hollow.py`.



Fitting the TB09-MGGA c-parameter

We use the TB09 meta-GGA functional[2] to calculate the projected local density of states and transmission spectrum of the device. In order to obtain an accurate description of the Si side of the interface, the TB09 c-parameter is fitted to match the silicon band gap. Details about this procedure can be found in the tutorial *Meta-GGA and 2D confined InAs*. The calculations are quite fast and can be run locally.

Follow these steps:

1. Compute the band structure of the optimized silicon bulk using ATK-DFT with the TB09-MGGA functional and 11x11x11 k-points. You should obtain a band gap of 1.16 eV and a self-consistently determined c-parameter of 1.0841.

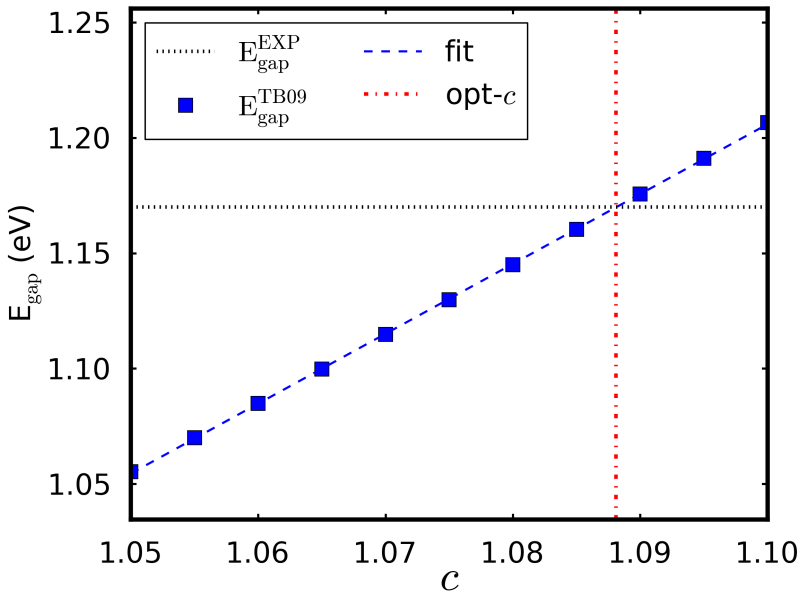
2. Do similar calculations with fixed c-values around 1.0841. This can be done in a single QuantumATK Python script by looping over the different c-values. Use the **Editor** to insert the loop:

```
for tb09 in (1.05,1.055,1.06,1.065,1.07,1.075,1.08,1.085,1.09,1.095,1.1):
    # ------------------------------------------------------------
    # Calculator
    # ------------------------------------------------------------
    #-------------------------------------
    # Exchange-Correlation
    #-------------------------------------
    exchange_correlation = MGGA.TB09LDA(c=tb09)
```

   Remember that the *Bandstructure* block should be included in the loop, just below the *Calculator* section. Use `nlsave('Si_bulk_fitc.nc', bandstructure)` to save all the band structures in a single data file.

3. Fit the c-value to the experimental band gap of 1.17 eV[3] using the script `c_fit.py`. It needs three input parameters; a list of the used c-values (`tb09`), the name of the .nc file containing the bandstructure calculations (`fname`), and the experimental band gap (`SK`).

   The optimal c-value should be around 1.0881.



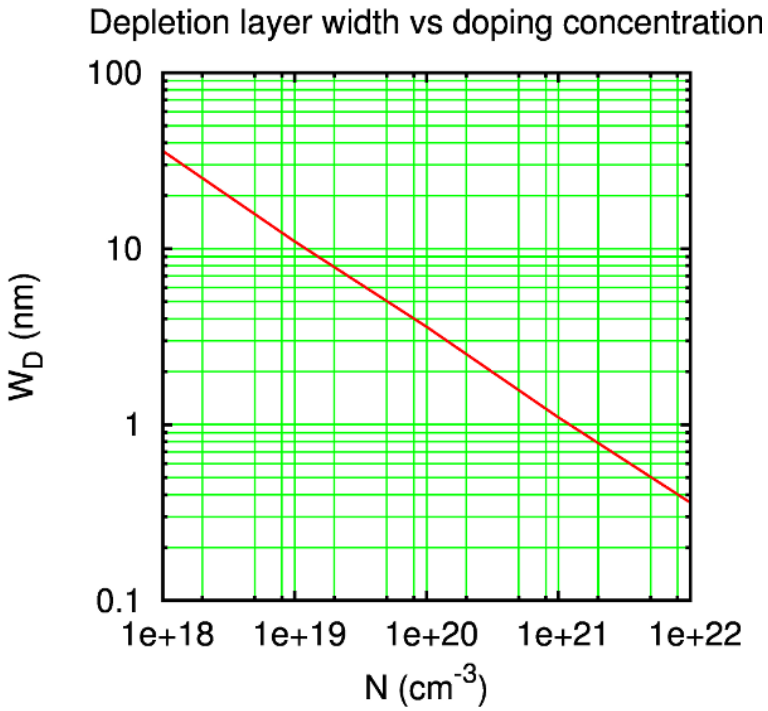
[2] F.ăTran and P.ăBlaha. Accurate band gaps of semiconductors and insulators with a semilocal exchange-correlation potential. *Phys. Rev. Lett.*, 102:226401, 2009. doi:10.1103/PhysRevLett.102.226401.

[3] C.ăKittel. *Introduction to Solid State Physics*. Wiley, 8th edition, 2004.

Silicon doping and depletion layer length

You are now ready to dope the silicon part of the device. Open the relaxed device configuration in the **Builder**, select all the silicon atoms, and use the **Doping** plugin to apply an n-type doping concentration of 10^{19} e/cm^3 to the right electrode.

It is important to consider the length of the depletion layer in your device: The device needs to be longer than the screening length of the semiconductor so that the potential on the silicon side at the boundary with the silicon electrode resembles that of bulk silicon. This can be done by studying the convergence of the Hartree potential with respect to the length of the silicon central region. For a good starting guess, you can consult some typical depletion layer lengths for metal–semiconductor interfaces[4]; a doping of 10^{19} cm^{-3} corresponds to a depletion layer length of around 100 Å.



Tip: You only need to converge the length of the silicon side of the interface, since the metal have a much shorter screening length. The 6 atomic layers of Ag will be sufficient.

Note: In QuantumATK versions 2015 and older, the HartreeDifferencePotential analysis object was called ElectrostaticDifferencePotential.

Increase the length of the silicon part of the central region by using the **Central Region Size** plugin. Use lengths of 80 Å, 100 Å, 120 Å, and 140 Å, and send each device configuration to the **Scripter**. Use a calculator with the same parameters as for the device relaxation, and add the HartreeDifferencePotential block.

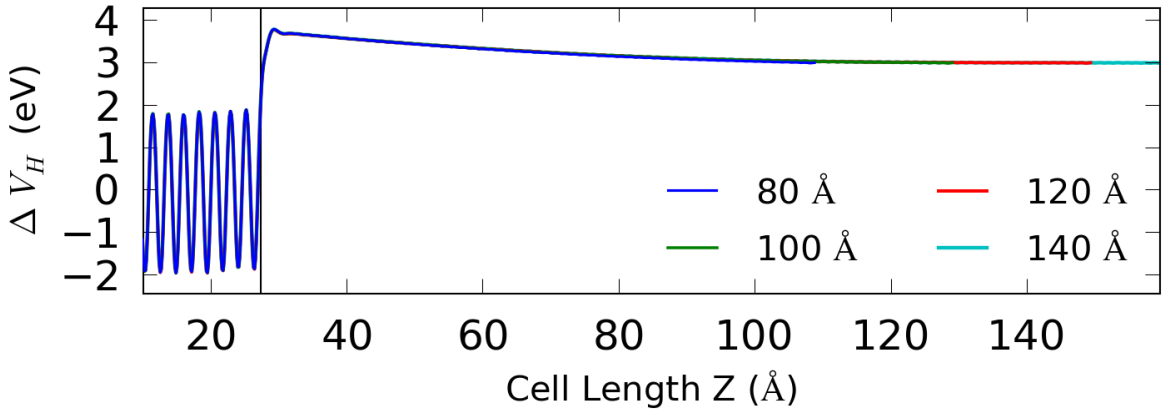
Note that the calculations become increasingly difficult to converge as the length is increased. At 120 Å and more, you will need to adjust the *Iteration Control Settings* in your calculator: In the **Scripter** window, change the **Script detail**

[4] E.ăKasper and D.ăJ. Paul. *Silicon Quantum Integrated Circuits*. Springer-Verlag Berlin Heidelberg, 2005. doi:10.1007/b137494.

to *Show defaults*, and send the script to the **Editor**. In the *Iteration Control Settings* block, change the values of `damping_factor`, `number_of_history_steps`, and `max_steps` used in the device calculation:

```
device_iteration_control_parameters = IterationControlParameters(
damping_factor=0.05,
linear_dependence_threshold=0.0,
algorithm=PulayMixer(),
preconditioner=Preconditioner.Off,
start_mixing_after_step=0,
number_of_history_steps=12,
max_steps=400,
tolerance=0.0001,
mixing_variable=HamiltonianVariable,
)
```

The calculation for the largest device requires around 13 hours on 16 cores. When all calculations have finished and the results appear on the **LabFloor**, you can plot the potentials along the Z-direction using the **1D projector** analysis tool. The potential should be flat in the vicinity of the boundary with the silicon electrode.



The wiggles of the potential can make it hard to see if it has converged to the electrode value. The **macroscopic average** of the potential, $\langle \Delta V_H \rangle$, smoothens out the wiggles. Use the script `hdp.py` to calculate the macroscopic average of the Hartree potential along the z-direction[5]. The script needs to be edited, as it contains some input parameters specific to the interface structure:

gauss_left
Distance along Z (in Å) between two consecutive atomic layers in the left electrode material.

gauss_right
Same as above, but for the right electrode material.

zcoord_int
Z-coordinate of the interface position (in Å) calculated as the midpoint between the last atom in the left-hand material and the first atom in the right-hand material.

fname
Name for the output .dat file containing the averaged potential.

[5] A.ăBaldereschi, S.ăBaroni, and R.ăResta. Band Offsets in Lattice-Matched Heterojunctions: A Model and First-Principles Calculations for GaAs/AlAs. *Phys. Rev. Lett.*, 61:734–737, Aug 1988. doi:10.1103/PhysRevLett.61.734.

Once you have edited these parameters, run the script once for each of the the .nc files containing the HartreeDifferencePotential analysis for the different configurations:

```
atkpython hdp.py device.nc
```

where `device.nc` should be changed to the actual .nc file name.

You can use the script `hdp_plot.py` to compare the resulting .dat output files. Modify the following input parameters:

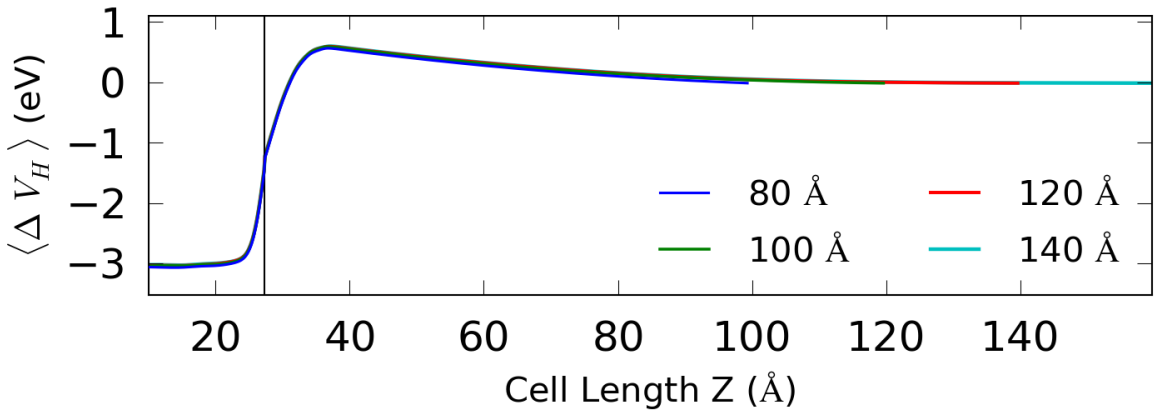
fnames
Names of the .dat files.

names
Lenghts of the silicon central region.

zcoord_int
Same as explained above (input for `hdp.py`).

Edit the script and run it to produce the figure below:

```
atkpython hdp_plot.py
```



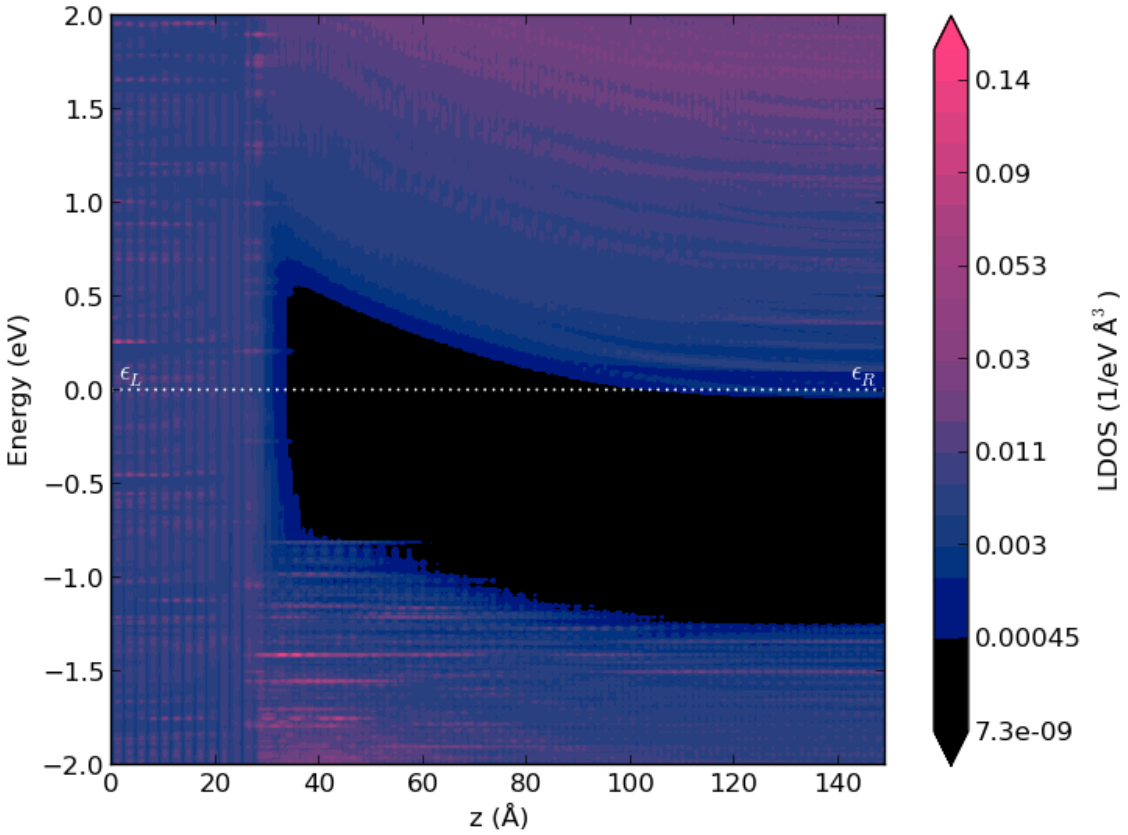
We may conclude from the plot that a silicon central region length of 120 Å should be sufficient. Note that the depletion layer length could change slightly with the bias. You now have the final `device.py`.

Note: The kink at the interface border in the plot above is due to the averaging procedure, where Gaussians of different widths are matched at the Ag–Si border.

Projected local density of states

The projected LDOS (PLDOS) offers a highly useful visualization of the band diagram of the interface. Use the **Analysis from File** block in the Scripter to load the saved device configuration and the converged calculator attached to it. Then add the **ProjectedLocalDensityOfStates** block and edit the settings: Energy range between -2 and +2 eV using 401 energy points and a 18x9 k-point grid. Save the results in the .nc file from which you load the configuration.

The calculation will take around 7 hours on a 16-core node with 64 GB of memory. When the calculation is finished and the results are on the **LabFloor**, you can use the **Projected Local Density of States** analysis plugin to plot the PLDOS along the device transport direction:



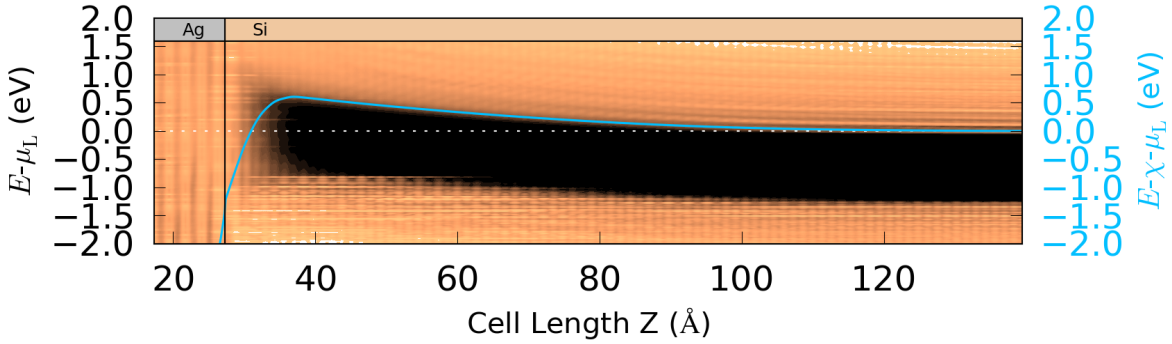
You can also plot the averaged Hartree difference potential along with the PLDOS using the script `pldos_hdp.py`. It requires the .nc file containing both the HartreeDifferencePotential and ProjectedLocalDensityOfStates analysis items, and the same input parameters as `hdp.py`, along with some plotting preferences:

leftname
Name of the left-hand interface material.

rightname
Name of the right-hand interface material.

lenleft
The amount of left material shown in the plot (in Å).

Run the script as `atkpython pldos_hdp.py device.nc` using the generated .nc file.



In this plot, you can easily see the **metal-induced gap states** to the right of the interface border. Moreover, the averaged potential nicely follows the conduction band minimum, both at the interface and far from it. We can therefore estimate the **Schottky barrier**, Φ^{pot}, by calculating the difference between the chemical potential of the left electrode, μ_L, and the maximum of $\langle \Delta V_H \rangle$. The plotting script returns this value together with the plot. You will get a barrier of around 0.606 eV, which is in agreement with experimental results, see[6] and[7]. The script also returns the position of the conduction band minimum with respect to μ_L. You will need this value later on for analyzing the *spectral current*.

Finite-bias calculations

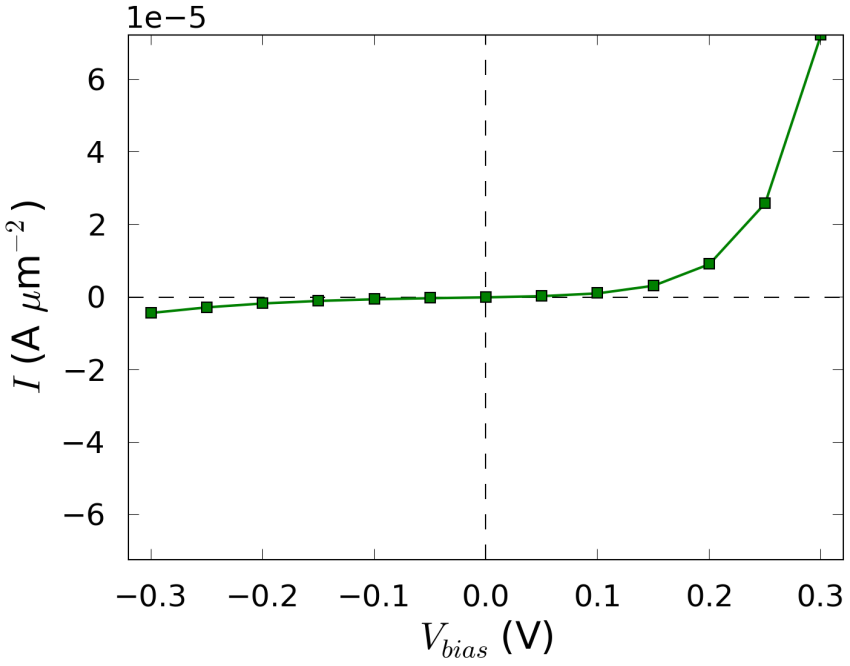
We will investigate the I–V characteristics next. Use the **Analysis from File** functionality and the **IVCurve** analysis block. Choose the following settings for the analysis:

- -0.3 to +0.3 V bias range with 15 points.
- -2 to +2 eV energy range with 401 points.
- 18x9 k-point grid.

The calculation will require around 15 hours on 16 cores. You can use the **IV-Plot** plugin to plot the calculated current vs. bias, or use the script `IV.py` to plot the current density against bias. The script requires a .nc file containing the device configuration and I–V curve items.

[6] R.ăBalsano, A.ăMatsubayashi, and V.ăP. LaBella. Schottky barrier height measurements of Cu/Si(001), Ag/Si(001), and Au/Si(001) interfaces utilizing ballistic electron emission microscopy and ballistic hole emission microscopy. *AIP Advances*, 3:112110, 2013. doi:10.1063/1.4831756.

[7] J.ăJ. Garramone, J.ăR. Abel, I.ăL. Sitnitsky, and V.ăP. LaBella. Hot-electron transport studies of the Ag/Si(001) interface using ballistic electron emission microscopy. *Journal of Vacuum Science & Technology A*, 28:643, 2010. doi:10.1116/1.3397795.



The interface shows Schottky diode-like behavior, with a large increase in the current at forward bias but a rather small increase at reverse bias.

Note: The computed current is significantly lower than the current reported in the publication[Page 843, 1]. See the *Note on the variation of the current* for an explanation.

Ideality factor

Thermionic emission theory describes the I–V characteristics of a Schottky diode as[8]

$$I = I_0 \mathrm{e}^{\frac{qV_{bias}}{nk_BT}} \left(1 - \mathrm{e}^{-\frac{qV_{bias}}{k_BT}}\right),$$

where q is the elementary charge, k_B is the Boltzmann constant, T is the temperature, I_0 is the saturation current, and n is the so-called ideality factor. The ideality factor is a measure of how much the interface resembles an ideal Schottky diode, where a value of n=1 represents the ideal case. Using a log-plot of $I/(1 - \mathrm{e}^{-qV_{bias}/k_BT})$ against V_{bias} you can extract the value of the ideality factor from the slope. Use the script `IV-n-log.py` to create such a plot and calculate the ideality factor. You should edit a few input parameters:

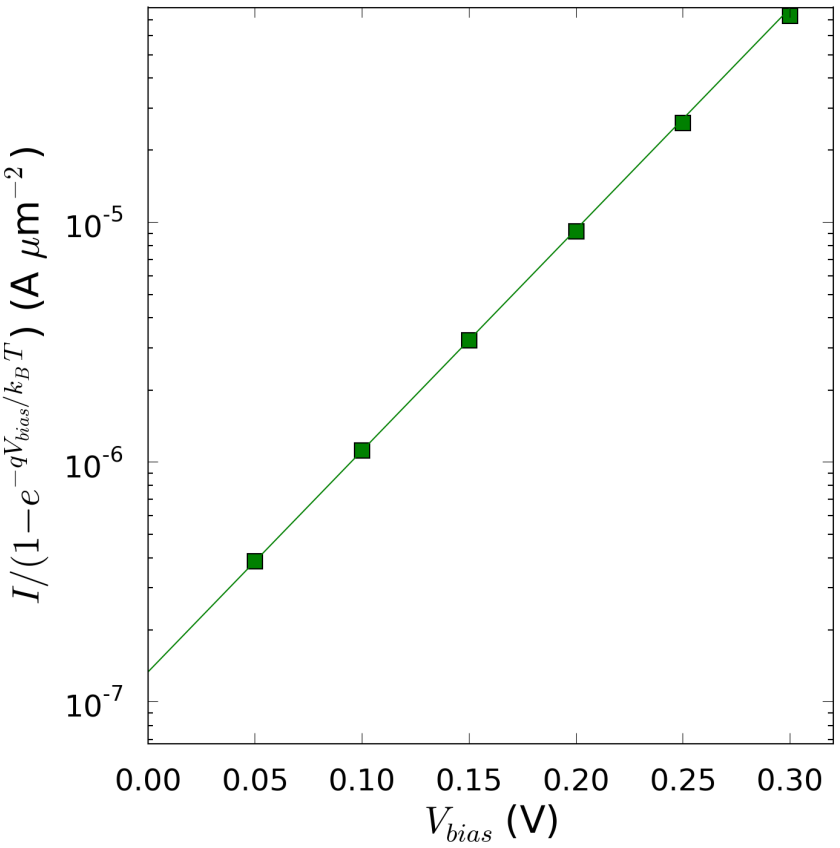
filename
The same .nc file as for `IV.py`.

T
The temperature at which the calculations have been run.

numpoints
The number of bias points to include in the fit used to find the ideality factor.

[8] S.ăM. Sze and K.ăN. Kwok. *Physics of Semiconductor Devices*. Wiley, 3rd edition, 2006.

Running the script, you should get a value around n=1.8208, which means that the system deviates significantly from the ideal Schottky diode behavior.



Schottky barrier

The Activation Energy (AE) method[8] is a widely used experimental procedure for extracting the Schottky barrier. The method extracts the Schottky barrier from an Arrhenius-like plot of $\ln(I/T^2)$ vs. $1/T$,

$$\ln(IT^2) = \ln(AA^*) - \frac{q}{k_B}\left(\Phi^{AE} - \frac{V_{bias}}{n}\right)\frac{1}{T} \Rightarrow$$

$$\phi^{AE}(V_{bias}) = \Phi^{AE} - \frac{V_{bias}}{n} = -\frac{k_B}{q}\frac{\mathrm{d}[\ln(I/T^2)]}{\mathrm{d}(1/T)}.$$

Use the script `IvsT.py` to calculate the transmission at each applied forward bias in a range of temperatures between 250 K and 400 K. The transmission is calculated in a linear-response fashion using the Landauer–Büttiker expression for the current,

$$I = \frac{2q}{h}\int T(E,\mu_L,\mu_R)\left[f\left(\frac{E-\mu_L}{k_BT}\right) - f\left(\frac{E-\mu_R}{k_BT}\right)\right]\mathrm{d}E,$$

where the transmission coefficient $T(E,\mu_L,\mu_R)$ has been evaluated self-consistently at a temperature of 300 K during the the IVCurve calculation.

The script needs three inputs to run:

fname
The .nc file containing the *IVCurve* analysis.

Tmax
Maximum temperature used in the calculations (in K).

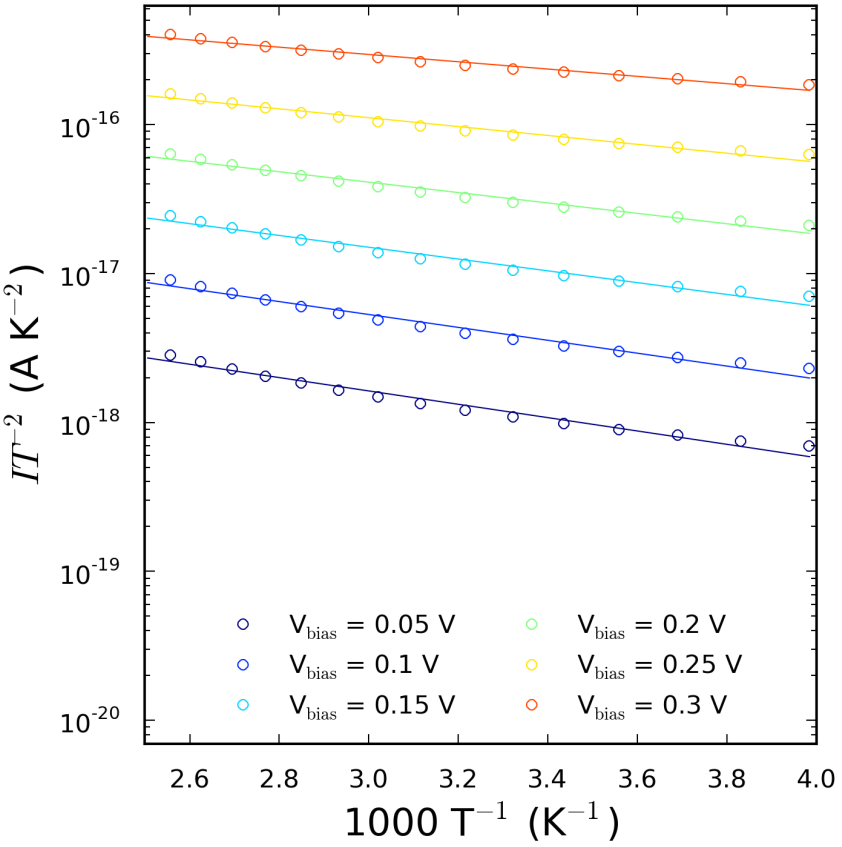
Tmin
Minimum temperature used in the calculations (in K).

Running the script produces a .dat file containing the I–T characteristics for each of the forward bias points. From these files, you can create the simulated Arrhenius plot using `arrhenius.py`. It requires the **Tmax** and **Tmin** inputs, and also the following:

IF
The calculated ideality factor from `IV-n-log.py`.

voltages
Forward biases for which the calculations have been performed (in V).

fname
Name of output file containing the effective barrier height (in eV).



The script produces the plot above, and also returns the Schottky barrier height Φ^{AE} and the effective barrier height $\phi^{AE}(V_{bias})$ for each applied bias (ϕ^{AE} is the barrier from the conduction band minimum of the bulk silicon electrode to the maximum of $\langle V_H \rangle$). The effective barrier height is exported as a .dat file for later analysis. You will find that the Schottky barrier increases with bias voltage. This is unphysical and should not occur, which suggests that the AE method provides only a limited description of this interface.

Spectral current

The final analysis is an investigation of the spectral current,

$$I(E) = \frac{2q}{h} T(E, \mu_L, \mu_R) \left[f\left(\frac{E-\mu_L}{k_B T}\right) - f\left(\frac{E-\mu_R}{k_B T}\right) \right].$$

We will plot the spectral current and the averaged Hartree difference potential together in order to compare the energy of maximum spectral current with the barrier of the Hartree potential.

The starting point is to calculate the Hartree potential at the different biases considered. Do this using **Analysis from File** and load the configurations saved in the file `ivcurve_selfconsistent_configurations.nc`, which was created by the **IVCurve** calculation, and add the **HartreeDifferencePotential** block for each configuration. The calculations should be very fast and can be run locally. You can then reuse `hdp.py` to calculate the average of each of the potentials.

Use the script `spectral_current.py` to plot the averaged potential together with the spectral current. You need to edit a few parameters:

fname
The .nc file containing the IVCurve analysis.

hdp_files
The .dat files containing the averaged potentials.

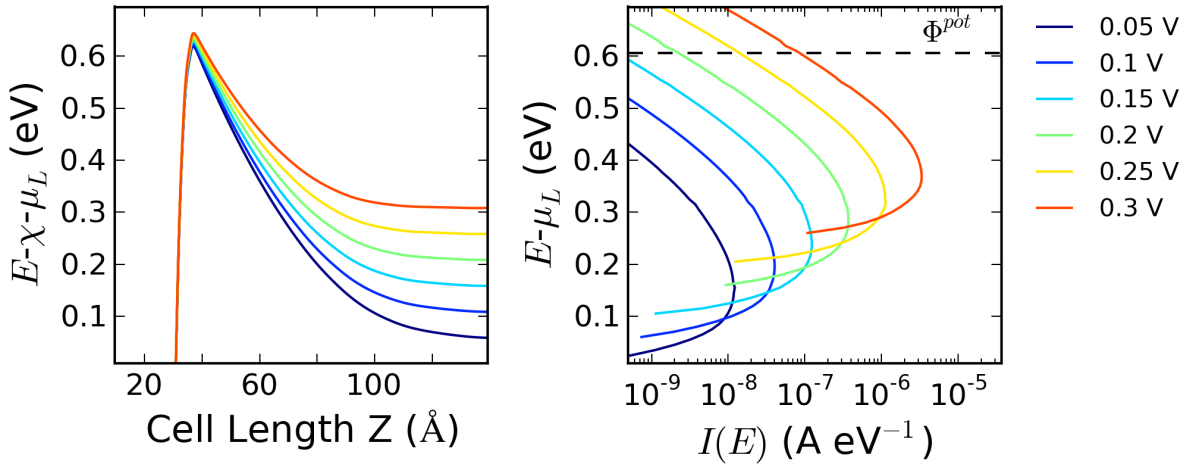
phipot
Schottky barrier calculated with `pldos_hdp.py` at zero bias (in eV).

CB_min
Conduction band minimum of the bulk Si electrode with respect to μ_L, calculated above with `pldos_hdp.py` (in eV).

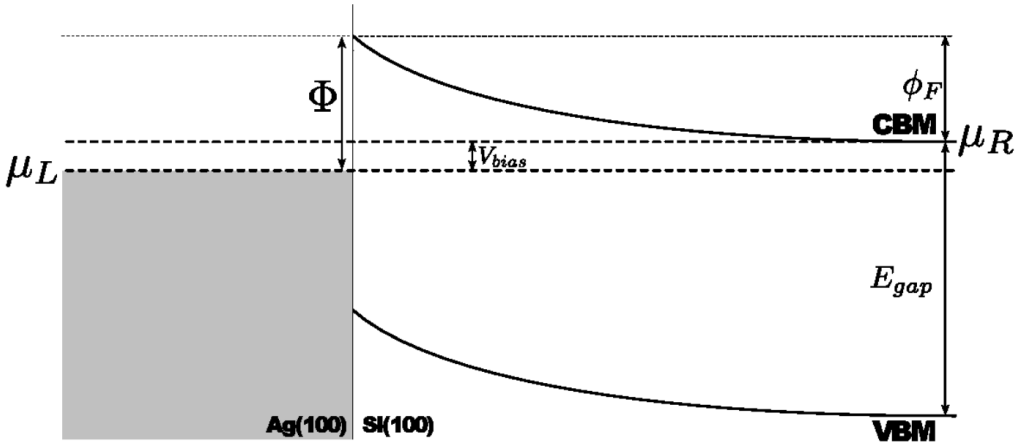
fname2
Name of the output file containing the barriers associated with the thermionic emission process (in eV).

fname3
Name of the output file containing the energy of maximum spectral current (in eV).

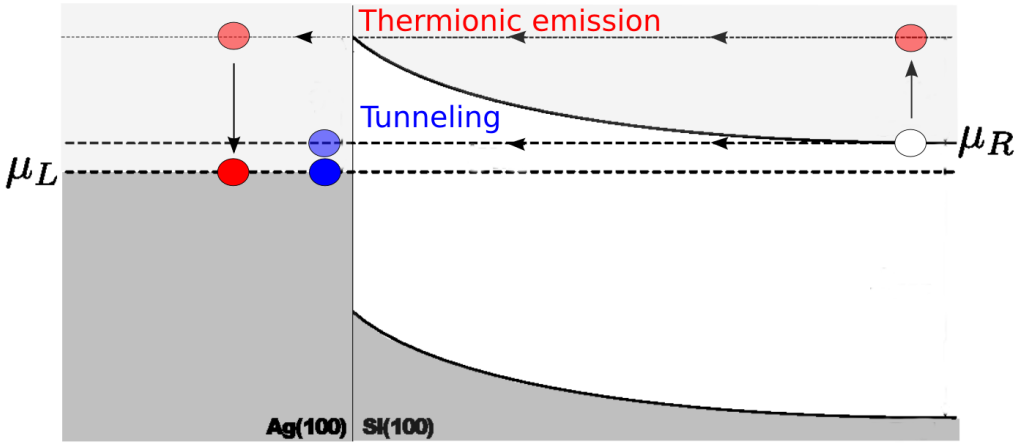
Running the script should produce the figure below.



The script also returns the estimated barrier for thermionic emission, ϕ_F, and the energy of maximum spectral current for each bias as .dat files. The energy axis of the two plots is aligned by plotting the averaged potential relative to the electron affinity of bulk Si (estimated by `pldos_hdp.py`) and μ_L and the spectral current relative to μ_L.



Transmission through the device can occur through two different processes; either by thermionic emission or by tunneling through the Schottky barrier. If the transmission occurs solely by thermionic emission, the spectral current would be zero below Φ^{pot}. This is definitely not the case. In fact, the energy of maximum spectral current is below the thermionic emission barrier for each bias, revealing that the dominant transmission process is tunneling!



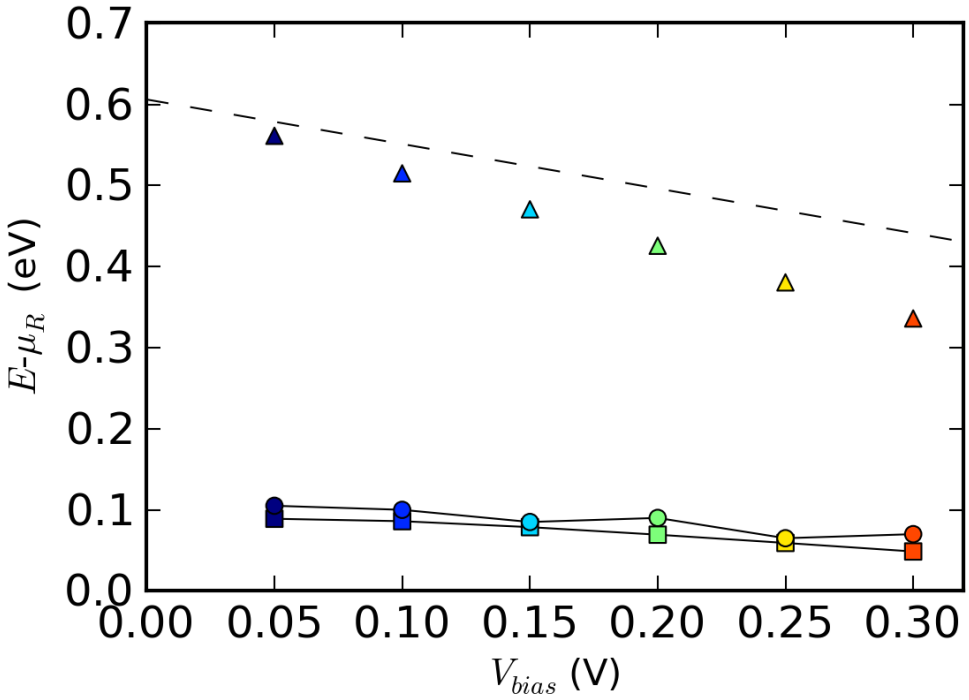
Summing up the results

We compare the finite-bias results by plotting the following:

- The energy of maximum spectral current from `spectral_current.py` (circles).
- The thermionic emission barrier ϕ_F from `spectral_current.py` (triangles).
- The effective AE Schottky barrier $\phi^{AE}(V_{bias})$ from `arrhenius.py` (squares).
- The bias dependence of the AE Schottky barrier predicted by $\phi^{AE}(V_{bias})$ by using $\Phi^{AE} = \Phi^{pot}$ (dashed line).

Create the plot using `barrier_compare.py`. It requires the .dat files created by `arrhenius.py` and

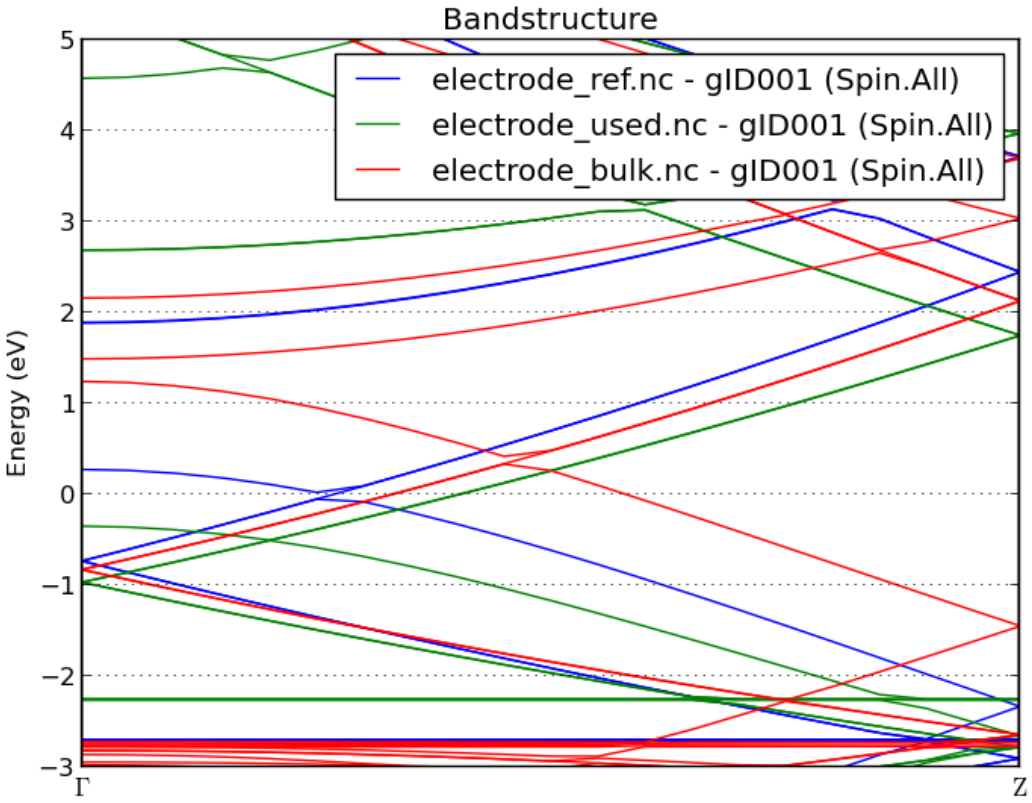
`spectral_current.py` along with the calculated ideality factor from `IV-n-log.py` and the estimated Schottky barrier at zero bias, Φ^{pot}, from `pldos_hdp.py`.



The plot shows that the estimated AE Schottky barrier (squares) lies below the thermionic emission barrier (triangles) for each bias point. This is because the AE method neglects contributions from tunneling and thereby implicitly assumes that the maximum spectral current (circles) will occur at the energy of the barrier height. Furthermore, we see that the analytical bias dependence of the thermionic emission barrier (dashed line) is in good agreement with the calculated barrier heights (triangles).

Note on the variation of the current

The calculated current is two orders of magnitude lower than the current reported in the publication[Page 843, 1]. This is due to the use of two slightly different Ag electrodes: The silver electrode used in the article was not relaxed after matching the two surfaces at the interface – that electrode is therefore compressed as compared to the Ag electrode used in this case study. This has an impact on the electronic structure of the electrode, which can be shown by comparing the electrode bandstructures:



In the figure above, electrode_ref.nc is the electrode used in the article, while electrode_used.nc denotes the electrode used here, and electrode_bulk.nc is a completely unstrained silver electrode. Comparing the first two, we see that the elongation of the electrode generally results in lower energy levels. The electrode used in this case study has fewer k-points (and thereby states) available in the bias window, which results in smaller transmission and thereby a lower current. However, if we compare the first two bandstructures to that of unstrained silver, we see that the main features around the Fermi level are preserved with both the electrode used here and the one used in the publication.

References

3.4.9 Resistivity calculations using the MD-Landauer method

Version: 2017.0

Downloads & Links

`Au100_L3.py` `Au100_L0_T0.py` `Au100_L3_T300.py` `Au100_L4_T300.py` `Au100_L5_T300.py` `analysis.py` `resistivity.py`

Transport calculations with QuantumATK

The bulk resistivity is an intrinsic property quantifying how strongly a material disrupts a flux of electronic current. In pristine metals, the main cause of the resistivity at room temperature is electron-phonon interaction, i.e., electron scattering due to the interaction with the vibrations of the lattice. The higher the temperature, the larger becomes the

vibrational amplitude of the lattice and hence the scattering probability, leading to linear increase of the resisitivity to the temperature.

This case study aims at introducing the **MD-Landauer** method, which is a conceptually simple methodology for calculating the temperature-dependent resistivity based on combined molecular-dynamics (MD) and electronic-transport calculations within the framework of the Landauer formalism (check the tutorial *Transport calculations with QuantumATK*). Despite its simplicity, the MD-Landauer method has several advantages over other more rigorous methods to calculate bulk resistivities (see, e.g., Refs.[1] and[2], and also the tutorial *Phonon-limited mobility in graphene using the Boltzmann transport equation*), as it takes naturally into account the anharmonic effects in describing lattice vibrations, and is applicable to non-crystalline systems as well.

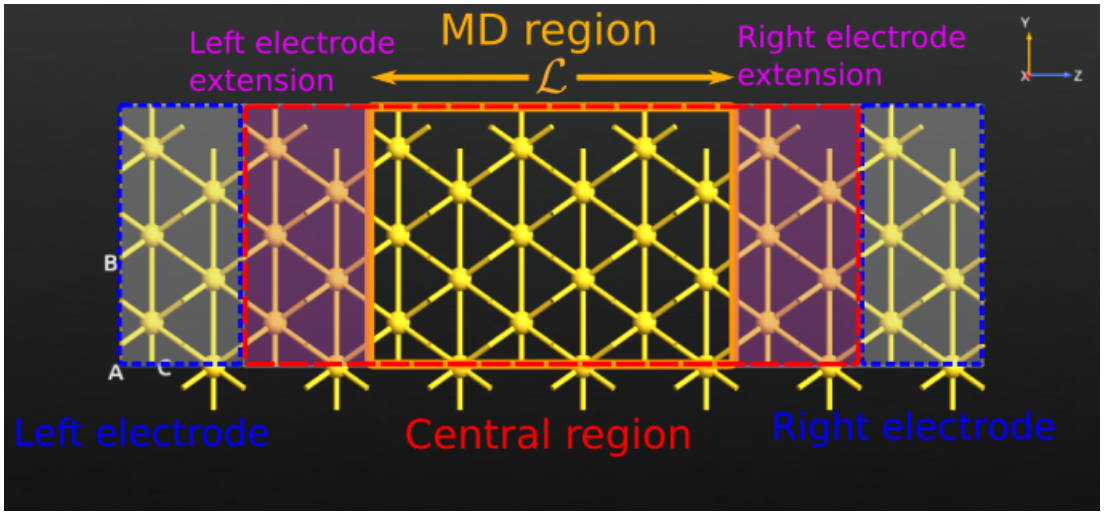
The case study consists of three sections, through which we reproduce some results for bulk gold recently reported in the paper *Electron-phonon scattering from Greens function transport combined with Molecular Dynamics: Applications to mobility predictions*[3]. Section *1. Theory and numerical procedure* provides some theoretical and methodological details. Section *2. Calculation setup* then describes how to setup an MD-Landauer calculation in **QuantumATK**. Finally, Sec. *3. Data analysis* is devoted to explaining how to analyze the data files and compute the resistivity.

Note: The numerical parameters employed in the calculations below have been tuned to achieve a compromise between computational efficiency and accuracy. The results will hence slightly differ from those presented in Ref.[Page 862, 3].

Note: QuantumWise case studies are primarily directed at experienced users of QuantumATK. Instructions are deliberately concise in order to focus mostly on the science.

---

1. Theory and numerical procedure



The figure above displays an illustrative example of a device configuration modeling bulk gold, for which the MD-Landauer calculations are performed. There are two semi-infinite electrodes and, in between, a **central region** which, importantly, consists of three parts: the **MD region** with length $\mathcal{L}$ and the **electrode copy** on its either side. An

[1] M.ăBrandbyge, J.-L. Mozos, P.ăOrdejon, J.ăTaylor, and K.ăStokbro. Density-functional method for nonequilibrium electron transport. *Phys. Rev. B*, 65:165401, Mar 2002. doi:10.1103/PhysRevB.65.165401.

[2] K.ăStokbro, D.ăE. Petersen, S.ăSmidstrup, A.ăBlom, M.ăIpsen, and K.ăKaasbjerg. Semiempirical model for nanoscale device simulations. *Phys. Rev. B*, 82:075420, Aug 2010. doi:10.1103/PhysRevB.82.075420.

[3] T.ăMarkussen, M.ăPalsgaard, D.ăStradi, T.ăGunst, M.ăBrandbyge, and K.ăStokbro. Electron-phonon scattering from Greens function transport combined with Molecular Dynamics: Applications to mobility predictions. *Phys. Rev. B*, 95:245210, June 2017. URL: https://arxiv.org/abs/1701.02883, doi:10.1103/PhysRevB.95.245210.

incoming flux of electrons from one of the electrodes interacts with phonons in the MD region, and is transmitted with a certain probability to the other electrode. Assuming that the electrodes and their replicas always remain equilibrium at temperature T, MD calculations are performed by allowing the atoms to move only in the MD region for a few nanoseconds, starting with random initial velocities based on the Maxwell-Boltzmann distribution. At the end of each MD calculations, the transmission coefficient is computed as

$$T[E;\mathbf{x}(T),\mathcal{L}] = \mathrm{Tr}\big\{G^{\mathrm{r}}[E;\mathbf{x}(T),\mathcal{L}]\Gamma_{\mathrm{L}}(E)G^{\mathrm{a}}[E;\mathbf{x}(T),\mathcal{L}]\Gamma_{\mathrm{R}}(E)\big\},$$

where $G^{\mathrm{r/a[E;\mathbf{x}(T),\mathcal{L}]}}$ represents the retarded/advanced Greens function, and $\Gamma_{\mathrm{L/R}}(E)$ is the width in the left/right electrode (see, e.g., Ref.[4] for details of the derivation). Note that the Greens functions and thus the transmission coefficient are both functions of E and parametrically depend on the set of displacement coordinates of atoms $\mathbf{x}(T)$ as well as $\mathcal{L}$. The Landauer formula then provides the conductance as a function of $\mathcal{L}$ and T:

$$G(\mathcal{L},T) = \frac{2e^2}{h}\int \langle T[E;\mathbf{x}(T),\mathcal{L}]\rangle \left[-\frac{\partial n_{\mathrm{F}}(E,\epsilon_{\mathrm{F}},T)}{\partial E}\right] dE,$$

where $n_{\mathrm{F}}(E,\epsilon_{\mathrm{F}},T)$ represents the Fermi-Dirac distribution function with ϵ_{F} being the Fermi energy, and $\langle\cdots\rangle$ means taking average over a set of MD calculations. The resistance is defined as the inverse of the conductance, i.e.,

$$\begin{aligned} 1/G(\mathcal{L},T) &= \\ R(\mathcal{L},T) &\\ &\equiv \\ R_{\mathrm{c}} + \rho_{\mathrm{1D}}(T)\mathcal{L}, & \end{aligned}$$

where the second line defines the one-dimensional (1D) resistivity $\rho_{\mathrm{1D}}(T)$. The bulk resitivity is then given by

$$\rho_{\mathrm{bulk}}(T) = \rho_{\mathrm{1D}}(T)\cdot A,$$

where A is the cross section of the device normal to the transport direction.

Dividing the procedure into three steps as follows will help clarify how to compute the 1D and bulk resistivies:

- **Calculation of the lattice geometries:**
For a given value of T and a set of values of $\mathcal{L}$, several MD calculations with random initial velocities are performed. For each calculation, the last snapshot of the MD run is taken.

- **Calculation of the resistance:**
For each set of snapshots with a given length $\mathcal{L}$, the averaged transmission coefficient is computed, which is then used to calculate the conductance $G(\mathcal{L},T)$ and the resistance $R(\mathcal{L},T)$.

- **Calculation of the finite-temperature resistivity:**
The 1D and bulk resistivities at the temperature T are calculated.

The rest of this case study details each step as well as the procedure to construct a device configuration of bulk gold.

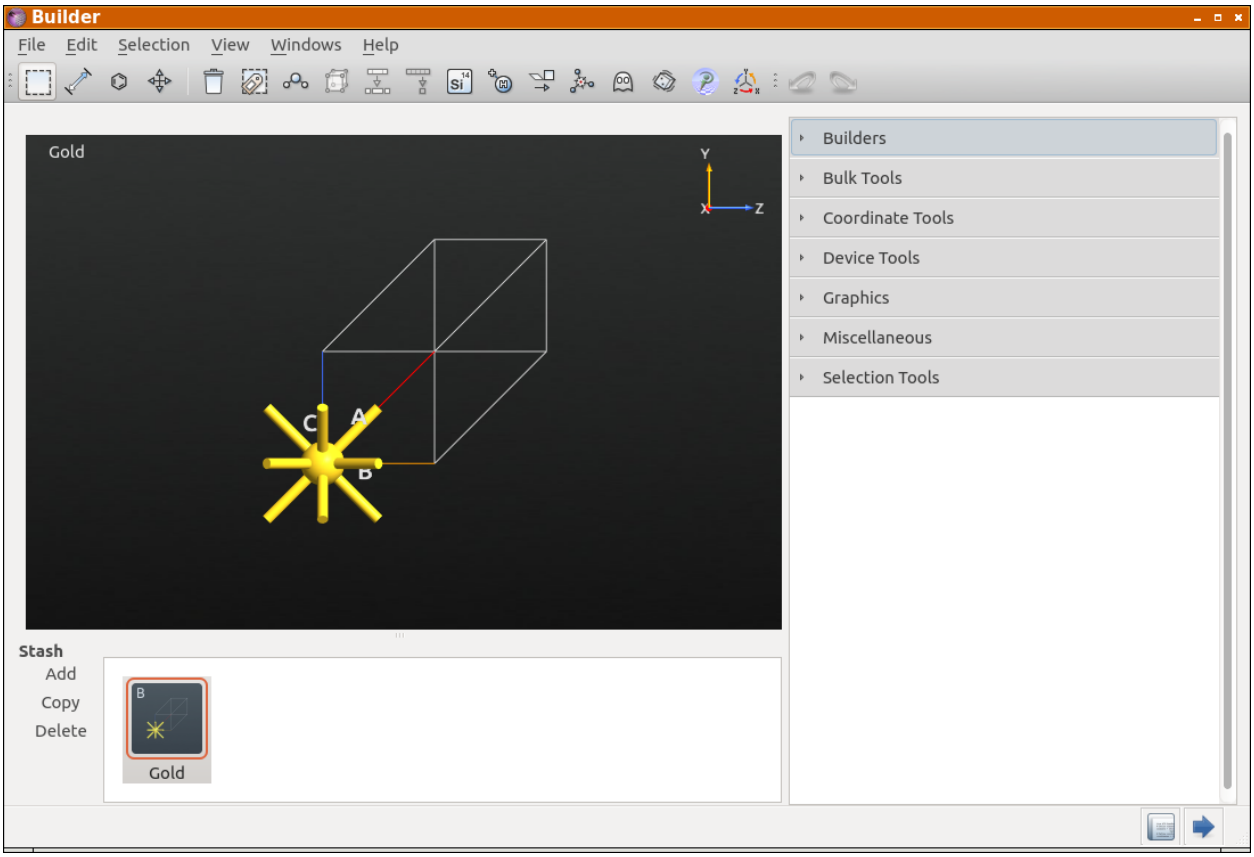
2. Calculation setup

2.1. Construction of the DeviceConfiguration

Tip: This subsection presents a detailed how-to guidline for preparing a device configuation of bulk gold, and finally provides an QuantumATK Python script, `Au100_L3.py`. To save time, one may just download the script and use it.

In the **Builder**, go to *Add* → *From Database* and select *Gold*:

[4] S.ăDatta. *Electronic Transport in Mesoscopic Systems*. Cambridge University Press, 1997. ISBN 0521599431. Cambridge Studies in Semiconductor Physics and Microelectronic Engineering.



Then use the *Builders* → *Surface (Cleave)* tool as follows to create a 3x3 Au[100] bulk supercell with a thickness of 10 layers:

1. Create a [100] surface by setting h=0 and k=l=0.
2. Define the surface lattice vectors by $\mathbf{v}_1 = 3\mathbf{u}_1$ and $\mathbf{v}_2 = 3\mathbf{u}_2$.
3. Select *Periodic and normal (electrode)* and set the thickness to 10 layers.

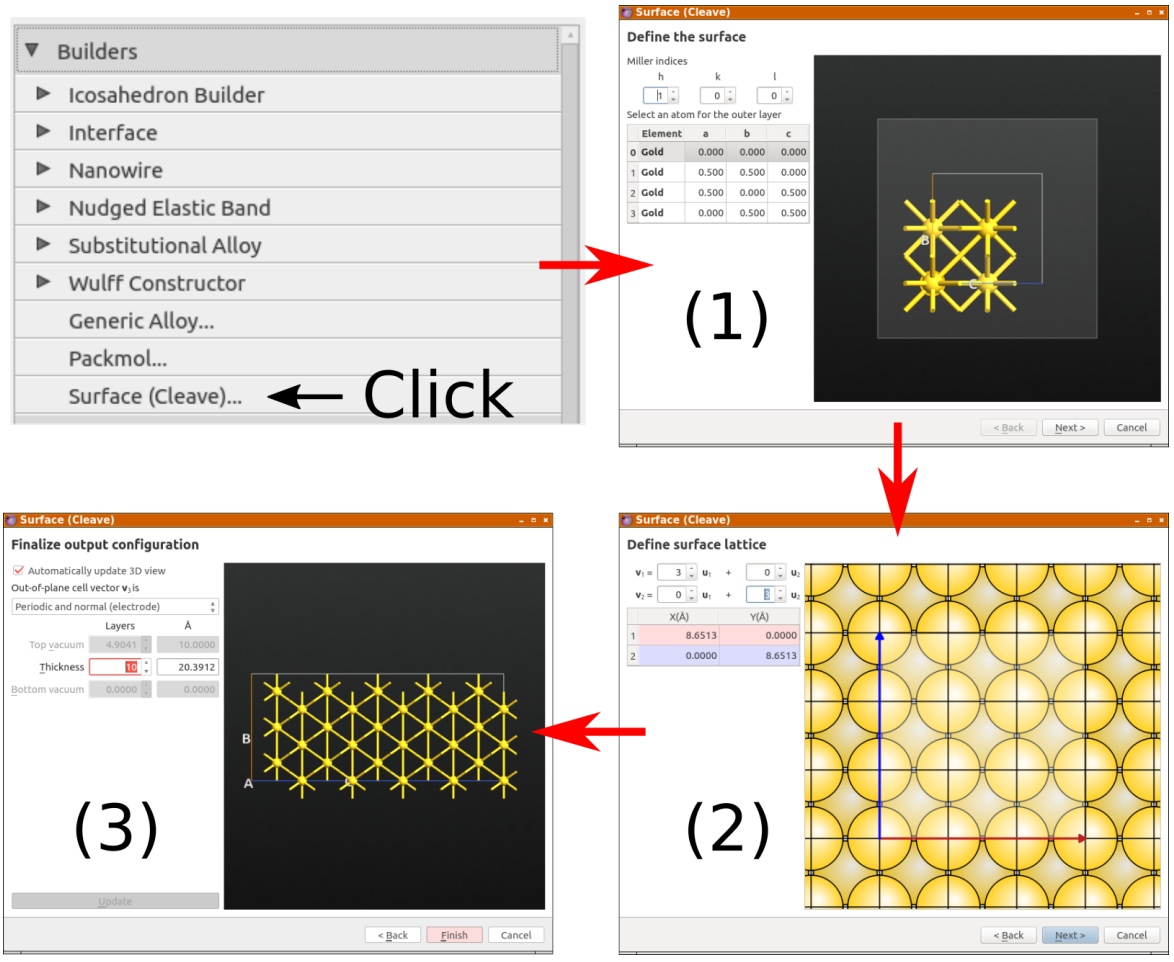
Finally, use the *Device Tools* → *Device from Bulk* plugin to create a DeviceConfiguration. Set the length of both right and left electrodes to be the minimum value 4.07825 Å.

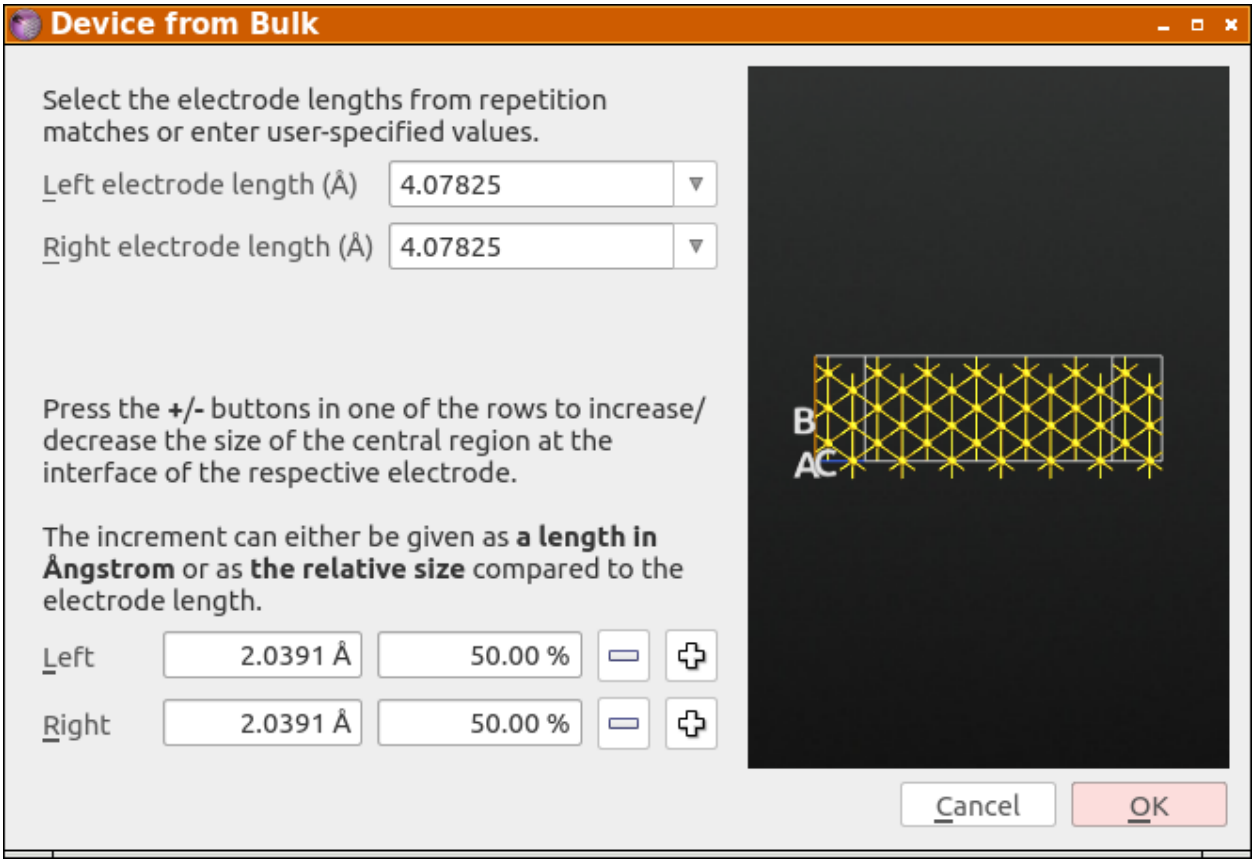
Save the device configuration as *Au100_L3.py*, where *L3* indicates $\mathcal{L} = 3$ in terms of the number of electrode repetitions. Following a similar procedure, one will be able to easily construct the device configurations for $\mathcal{L} = 4$ and 5, and save them with proper file names.

2.2. Setup of the script

Send the device configuration to **Scripter** by clicking , and add four blocks in the order as follows:

- *New Calculator*
- *Optimization* → *MolecularDynamics*
- *New Calculator*


Builders
Icosahedron Builder
Interface
Nanowire
Nudged Elastic Band
Substitutional Alloy
Wulff Constructor
Generic Alloy...
Packmol...
Surface (Cleave)...
Click
Surface (Cleave)
Define the surface
Miller indices
h
k
l
Select an atom for the outer layer
Element
a
b
c
0 Gold 0.000 0.000 0.000
1 Gold 0.500 0.500 0.000
2 Gold 0.500 0.000 0.500
3 Gold 0.000 0.500 0.500
(1)
< Back
Next >
Cancel
Define surface lattice
X(Å)
Y(Å)
1 8.6513 0.0000
2 0.0000 8.6513
(2)
Finalize output configuration
Automatically update 3D view
Out-of-plane cell vector v3 is
Periodic and normal (electrode)
Layers
Å
Top vacuum
Thickness
Bottom vacuum
20.3912
Update
(3)
Finish



- *Analysis* → *TransmissionSpectrum*

In the section **Global IO options**, change the name of the output file to *Au100_L3_T300.nc*. The parameters in each block should then be specified as follow.

MD section

The set of two blocks, and , in the section (1) highlighted in yellow, is meant to account for the MD calculations.

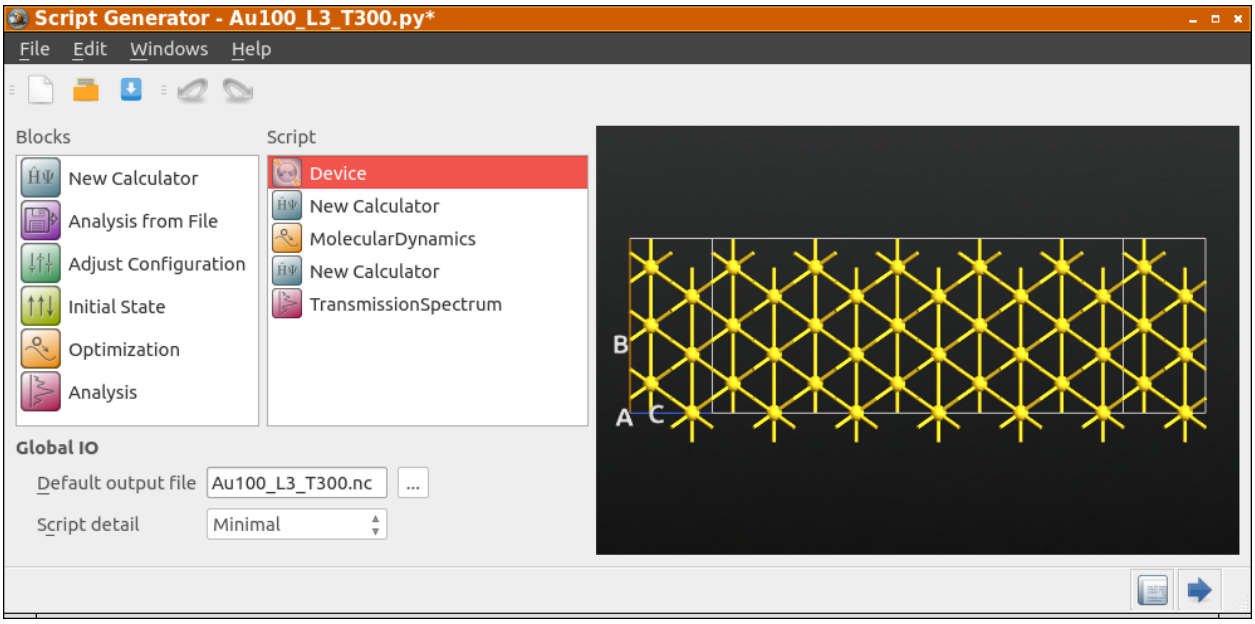
Open the **New Calculator** block, then select ATK-ForceField from the available **Calculators**, and set *EAM_Au_Sheng_2011*[5] in **Parameter set**.

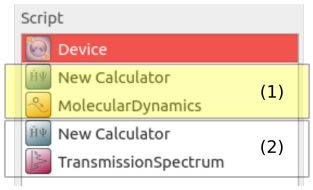
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

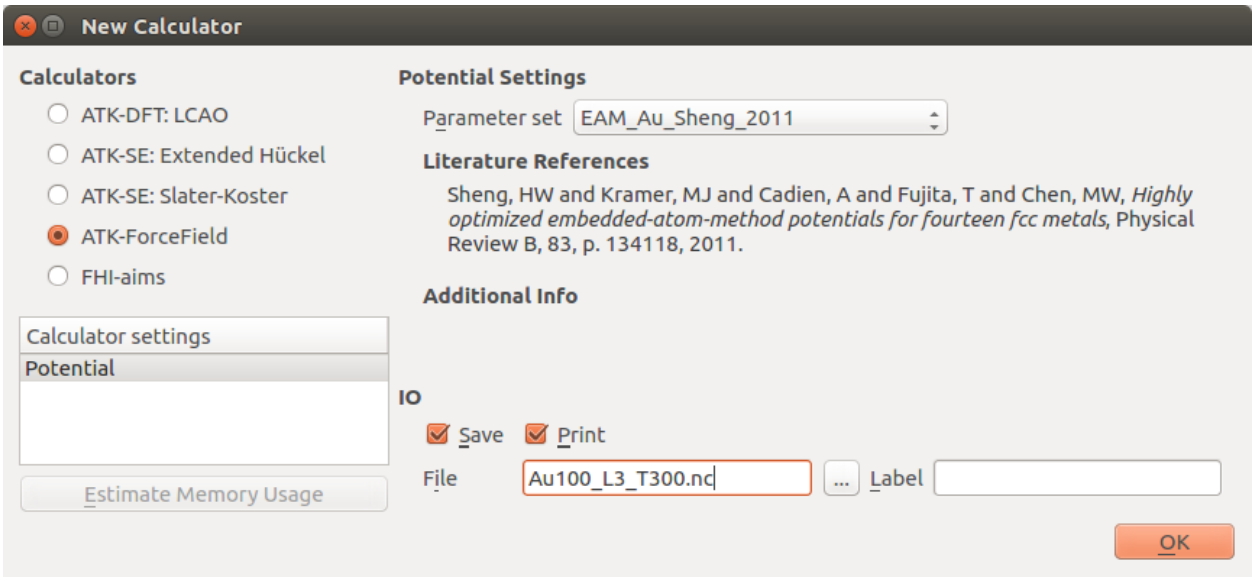
In the **MolecularDynamics** block, choose the Langevin thermostat[6] and set the rest of the MD parameters as shown below.

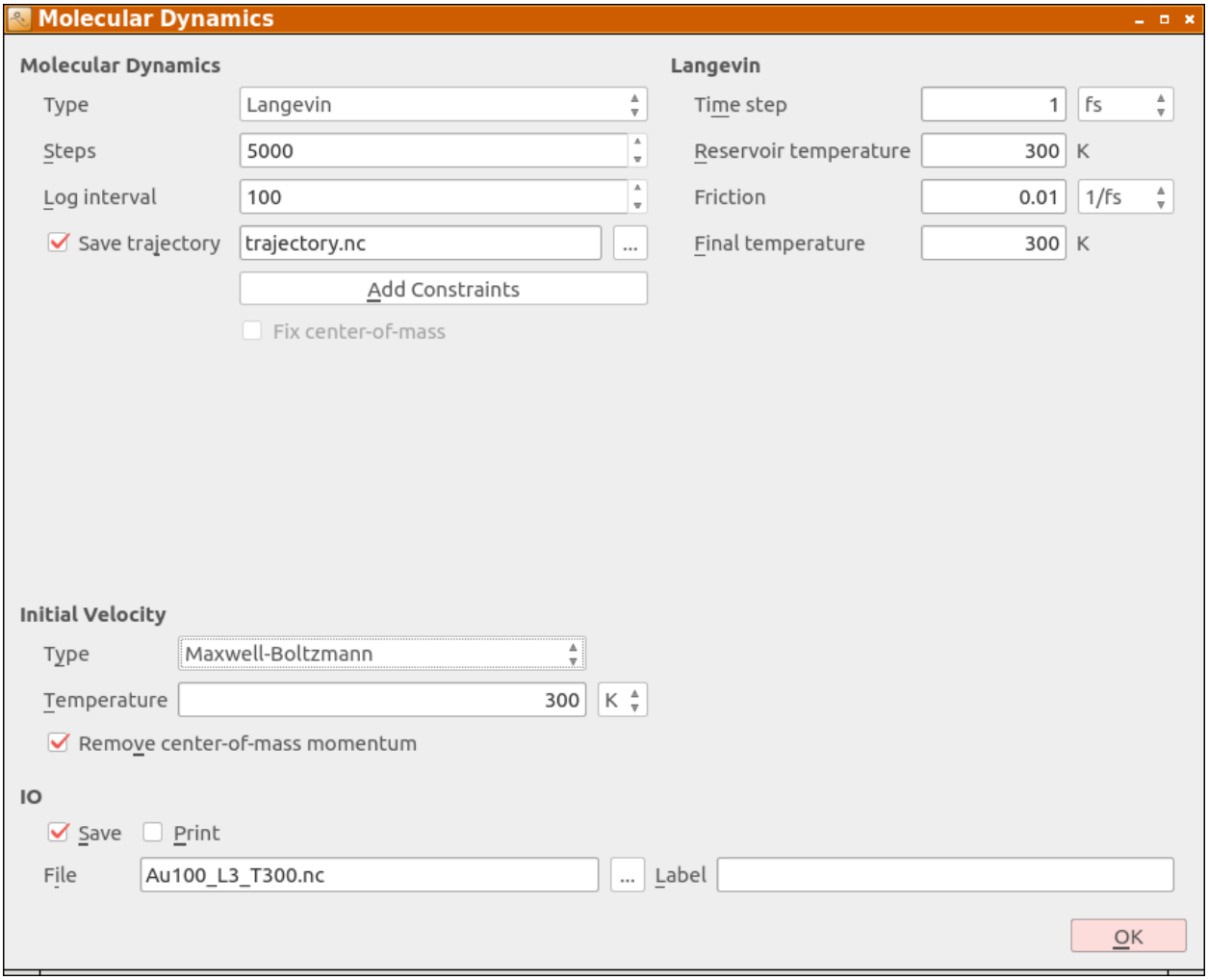
[5] H.ăW. Sheng, M.ăJ. Kramer, A.ăCadien, T.ăFujita, and M.ăW. Chen. Highly optimized embedded-atom-method potentials for fourteen fcc metals. *Phys. Rev. B*, 83:134118, April 2011. doi:10.1103/PhysRevB.83.134118.

[6] N.ăGoga, A.ăJ. Rzepiela, A.ăH. deăVries, S.ăJ. Marrink, and H.ăJ.ăC. Berendsen. Efficient Algorithms for Langevin and DPD Dynamics. *Journal of Chemical Theory and Computation*, 8(10):3637–3649, 2012. PMID: 26593009. doi:10.1021/ct3000876.


Script Generator - Au100_L3_T300.py*
File Edit Windows Help
Blocks
New Calculator
Analysis from File
Adjust Configuration
Initial State
Optimization
Analysis
Script
Device
New Calculator
MolecularDynamics
New Calculator
TransmissionSpectrum
Global IO
Default output file
Au100_L3_T300.nc
Script detail
Minimal
B
A
C


Script
Device
New Calculator
MolecularDynamics
(1)
New Calculator
TransmissionSpectrum
(2)

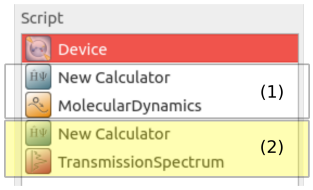

New Calculator
Calculators
ATK-DFT: LCAO
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-ForceField
FHI-aims
Calculator settings
Potential
Estimate Memory Usage
Potential Settings
Parameter set
EAM_Au_Sheng_2011
Literature References
Sheng, HW and Kramer, MJ and Cadien, A and Fujita, T and Chen, MW, Highly optimized embedded-atom-method potentials for fourteen fcc metals, Physical Review B, 83, p. 134118, 2011.
Additional Info
IO
Save
Print
File
Au100_L3_T300.nc
Label
OK


Molecular Dynamics
Molecular Dynamics
Type
Langevin
Steps
5000
Log interval
100
Save trajectory
trajectory.nc
...
Add Constraints
Fix center-of-mass
Langevin
Time step
1
fs
Reservoir temperature
300
K
Friction
0.01
1/fs
Final temperature
300
K
Initial Velocity
Type
Maxwell-Boltzmann
Temperature
300
K
Remove center-of-mass momentum
IO
Save
Print
File
Au100_L3_T300.nc
...
Label
OK

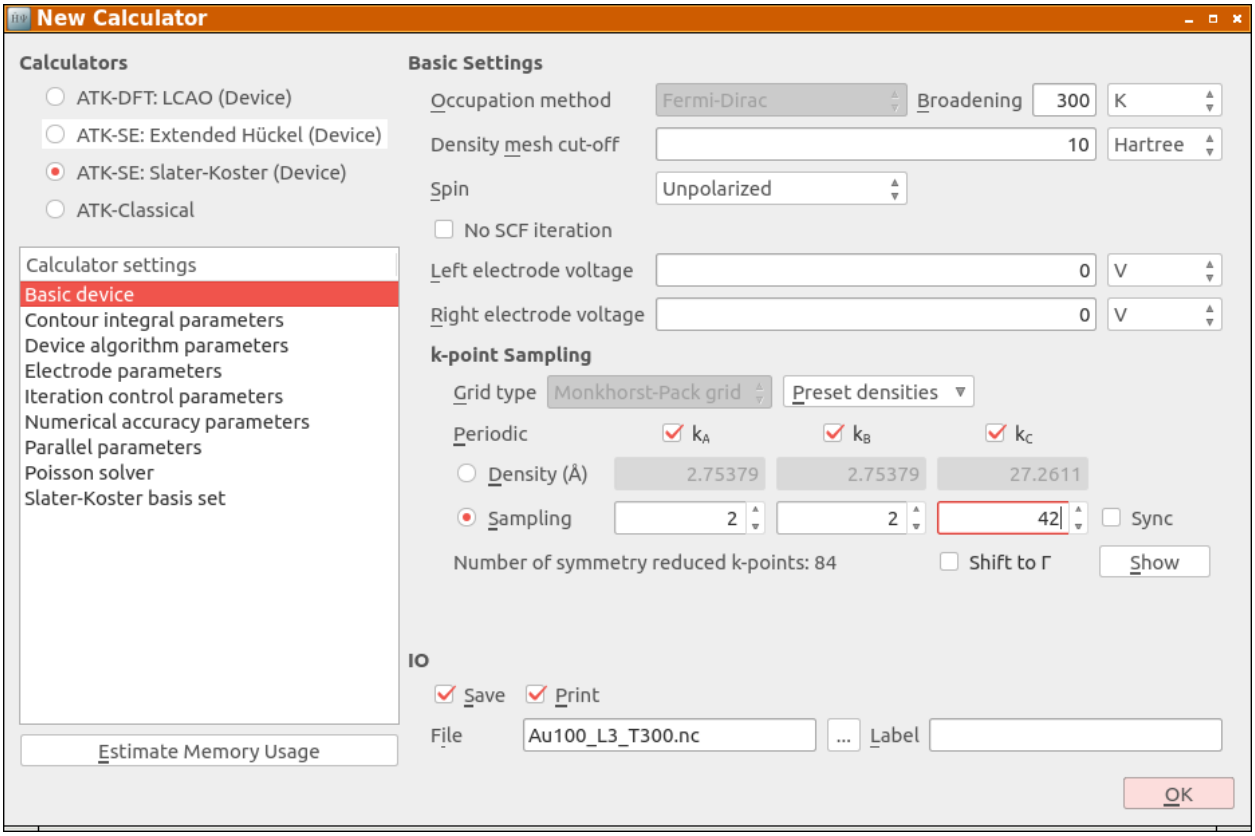
Important: In order to describe a random distribution of initial velocities in the MD simulations, remember to set the initial velocity type to Maxwell-Boltzmann.

Landauer calculation section

The set of two blocks, and , in the section (2) highlighted in yellow is meant to account for the DFTB-based transport calculations based on the Landauer approach.

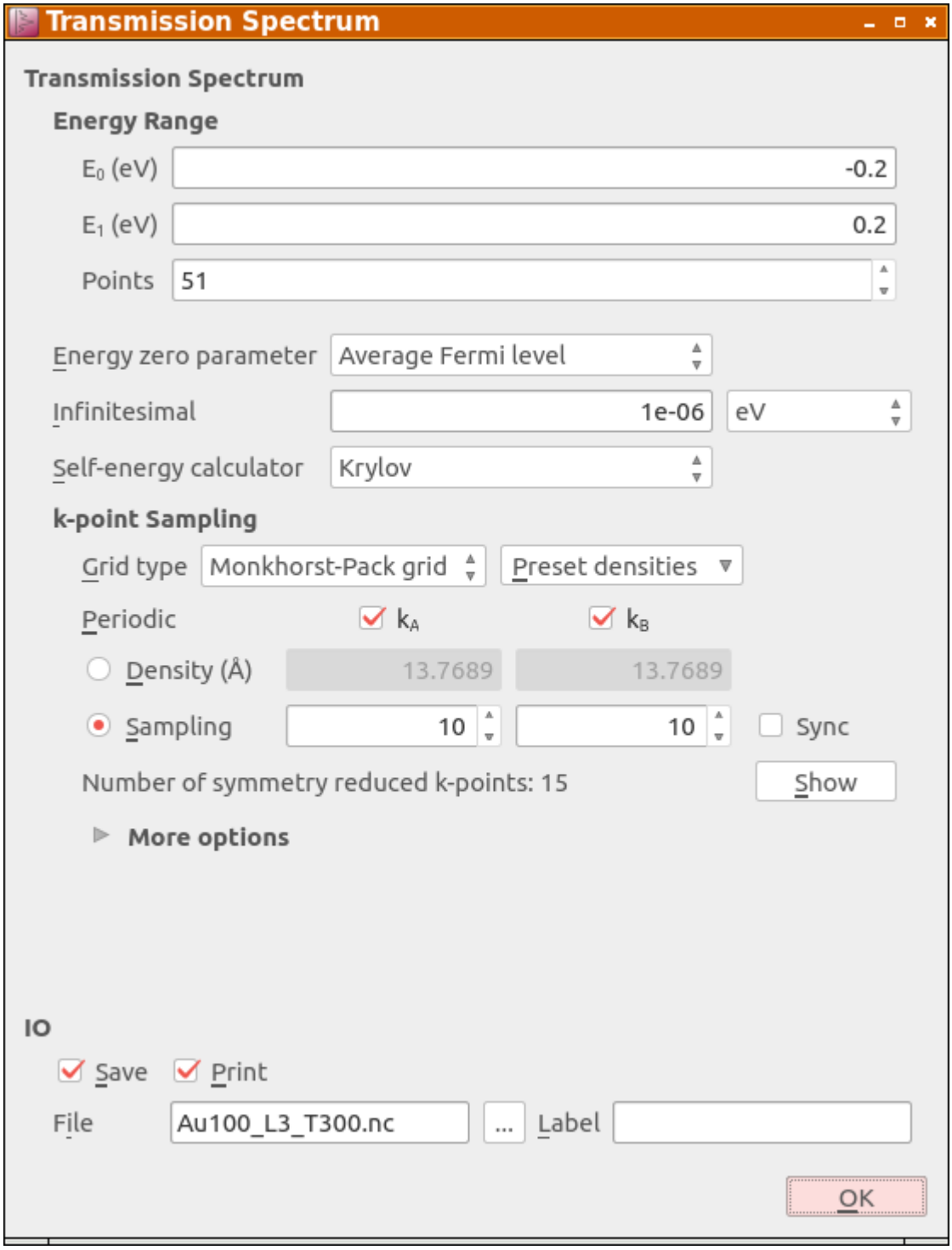


Click on the **New Calculator** block. Select *ATK-SE Slater_Koster (Device)* from the list of available **Calculators** and set up the parameters as shown below.



Important: Remember to untick *No SCF iteration* to carry out the calculations self-consistently.

Click on the **TransmissionSpectrum** object, and set the parameters as shown below.


Transmission Spectrum
Transmission Spectrum
Energy Range
E0 (eV)
-0.2
E1 (eV)
0.2
Points
51
Energy zero parameter
Average Fermi level
Infinitesimal
1e-06
eV
Self-energy calculator
Krylov
k-point Sampling
Grid type
Monkhorst-Pack grid
Preset densities
Periodic
kA
kB
Density (Å)
13.7689
13.7689
Sampling
10
10
Sync
Number of symmetry reduced k-points: 15
Show
More options
IO
Save
Print
File
Au100_L3_T300.nc
...
Label
OK

Note: The choice of the *Krylov* method is of particular importance as it accelerates the convergence in the SCF calculations.

2.3. Edit the python script

Making fully use of GUI supported by **QuantumATK**, the device configuration has been so far constructed with the necessary calculators and analyzer attached. The MD-Landauer method for computing the conductance, resistance, and resistivity, however, requires performing the calculation, for each of given values of $\mathcal{L}$ and T, several times to obtain averaged transmission coefficient $\langle T[E;\mathbf{x}(T),\mathcal{L}]\rangle$. To this end, the python script needs to be slightly modified by hand.

By clicking the icon, send the script to **Editor**. Highlight the whole script and press the `Tab` key on your keyboard to indent every line. Then edit the script as shown below, and save it as *Au100_L3_T300.py* To make sure that there is no mistake in your code, download `Au100_L3_T300.py` and compare the two scripts.

```
# -*- coding: utf-8 -*-

for run in range(0,10):

    # ------------------------------------------------------------
    # Two-probe Configuration
    # ------------------------------------------------------------

    # ------------------------------------------------------------
    # Left Electrode
    # ------------------------------------------------------------

    # Set up lattice
    vector_a = [8.65127469112, 0.0, 0.0]*Angstrom
    vector_b = [0.0, 8.65127469112, 0.0]*Angstrom
    vector_c = [0.0, 0.0, 4.07825]*Angstrom
    left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)
```

Note: Since the MD and Landauer sections are all placed under a loop, this script evaluates the transmission coefficient 10 times starting with different values of initial velocity, and saves each result with a unique ID number (gIDxxx) attached.

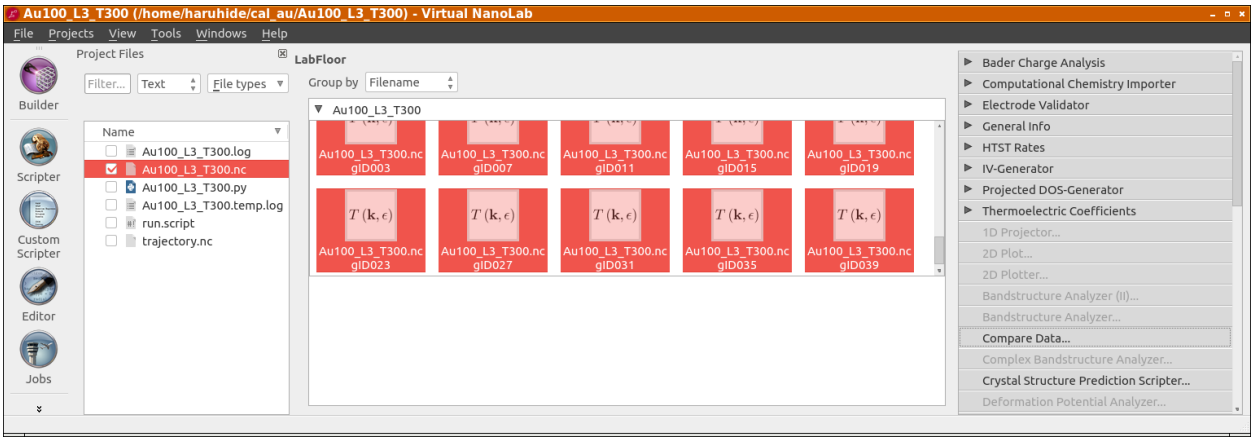
Note: Following exactly the same procedure except employing different values of $\mathcal{L}$, e.g., $\mathcal{L} = 4$ and 5, one can easily prepare the scripts for systematic implementation of the MD-Landauer calculations at T =300 K. To save time, one may download `Au100_L4_T300.py` and `Au100_L5_T300.py`, and use them.

3. Data analysis

Click the icon, send the script to **Job Manager**, and run it. The calculation takes 5-6 hours on 8 cores. The set of transmission coefficients should then be averaged for the analysis. This section hence focuses on how to analyze the results and obtain the 1D and bulk resistivities.

3.1. Transmission functions, conductance, and resistance

QuantumATK provides a useful plugin to make comparison of the transmission functions. Select all the transmission functions $T(\mathbf{k}, \epsilon)$ in **LabFloor** and click the `Compare Data...` button in **Plugin Bar**.



A window will pup up and display all the transmission functions $\{T[E; \mathbf{x}(T), \mathcal{L}]\}_{\mathbf{x}(T)}$ for $T = 300$ K and $\mathcal{L} = 3$.

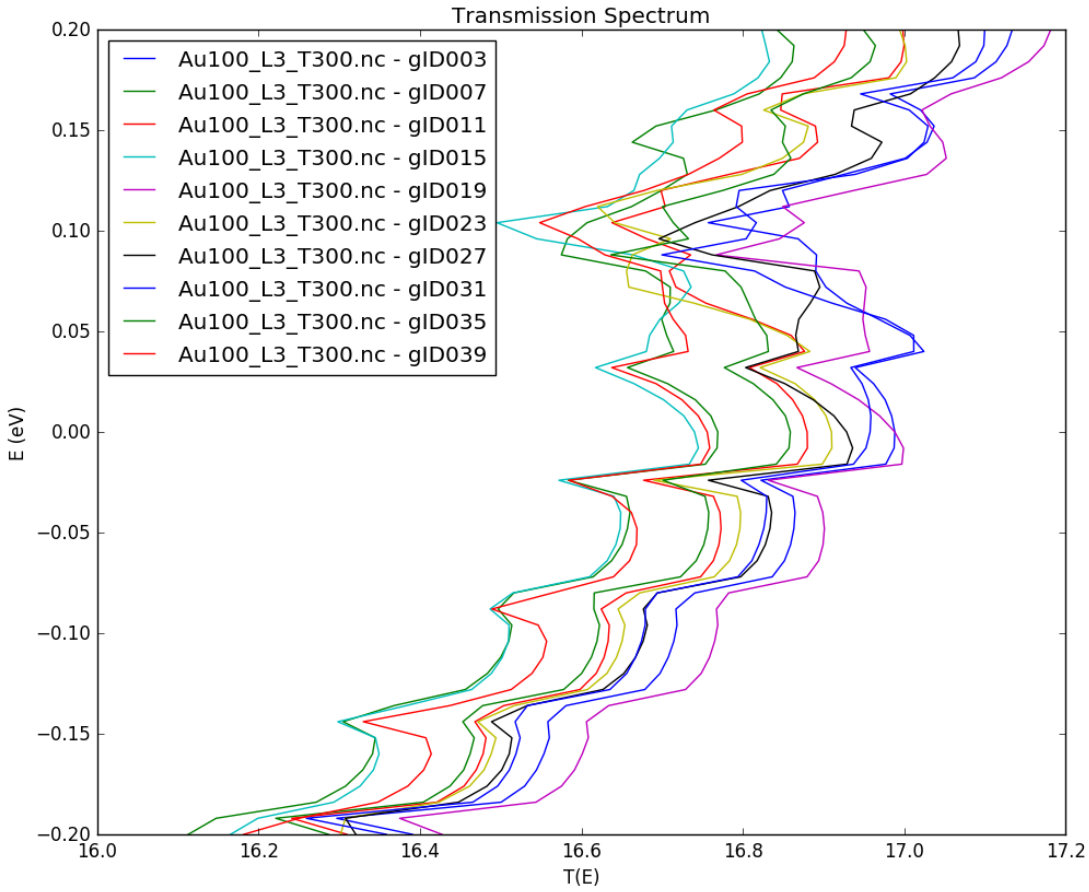
To compute the average of them, download the script `analysis.py` and run it as `atkpython analysis.py` in the same directory of the nc. The analysis will finish in a few seconds, and provide a list of information about the length of the MD-region, and the averaged conductance and resistance:

There also appears a pup-up window, in which black lines show the transmission coefficients as before, and a thick red line represents their average $\langle T[E; \mathbf{x}(T), \mathcal{L}] \rangle$.

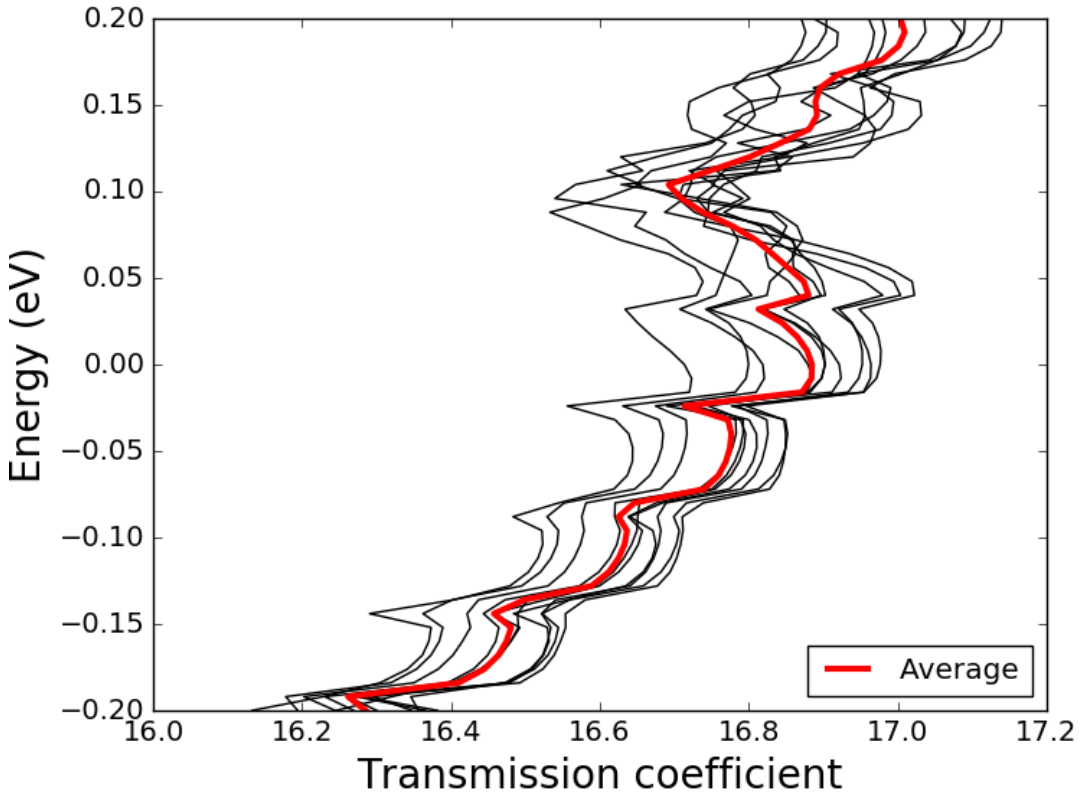
3.2. Resistivity

To compute the resistance at $T = 300$ K also for $\mathcal{L} = 0$, 4, and 5, download the scripts `Au100_L0_T0.py`, `Au100_L4_T300.py`, and `Au100_L5_T300.py`. Run them as `atkpython Au100_Lx_Txxx.py`, and analyze the results. One will then obtain a list of resistances, indicating the linear dependence of the resistance on $\mathcal{L}$:

Length of the MD-region, $\mathcal{L}$		Resistance
(# of electrode repetitions)	(Å)	()
0	0	1482.9
3	12.235	1535.1
4	16.313	1555.2
5	20.391	1571.9




```
LXTerminal
File  Edit  Tabs  Help
Calculating Kinetic Matrix : ================================================

+-------------------------------------------------------------------------+
| MD-Landauer Reprot                                                      |
+-------------------------------------------------------------------------+
| Lenght of the MD region = 12.23475 Ang
| Averaged conductance    = 0.000651443815697 Siemens
| Averaged resistance     = 1535.05179711 Ohm
+-------------------------------------------------------------------------+

Timing:                          Total     Per Step       %

-------------------------------------------------------------------------------

File IO, nlread          :       1.16 s       0.10 s      25.39% |=============|
Loading Modules + MPI    :       1.07 s       1.07 s      23.29% |============|
-------------------------------------------------------------------------------
Total                    :       4.59 s
```



Note: The resistance for $\mathcal{L} = 0$ is computed using the script `Au100_L0_T0.py`, which implements no MD calculation, and because of it, is hence named with *T0*. This script aims at evaluating the contact resistance.

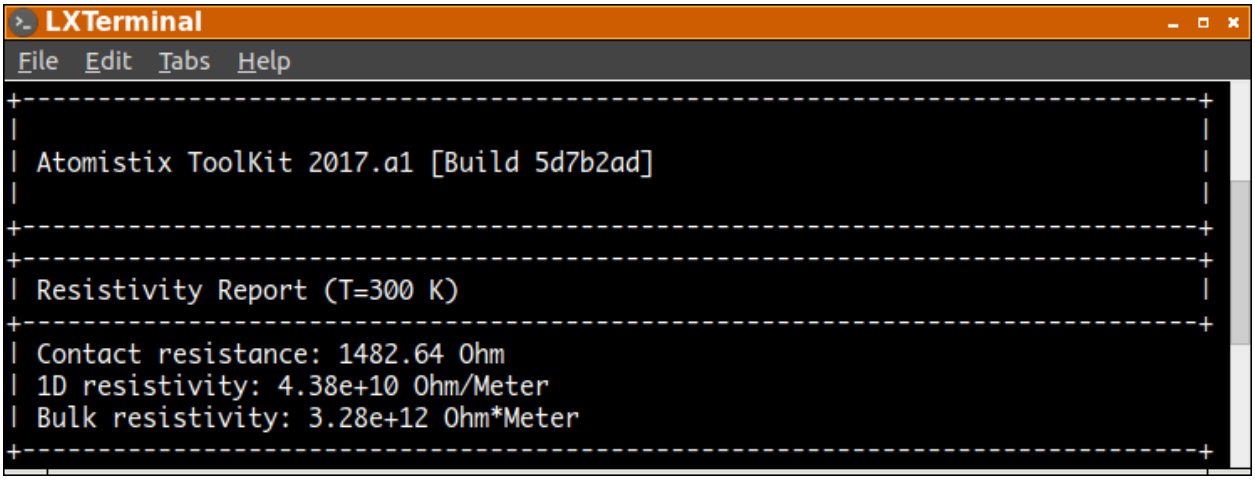
Let us then compute the 1D and bulk resistivities, which requires the value of the cross section. Noting the primitive vectors defined in lines 14 and 15 of *Au100_L3_T300.py*, i.e.,

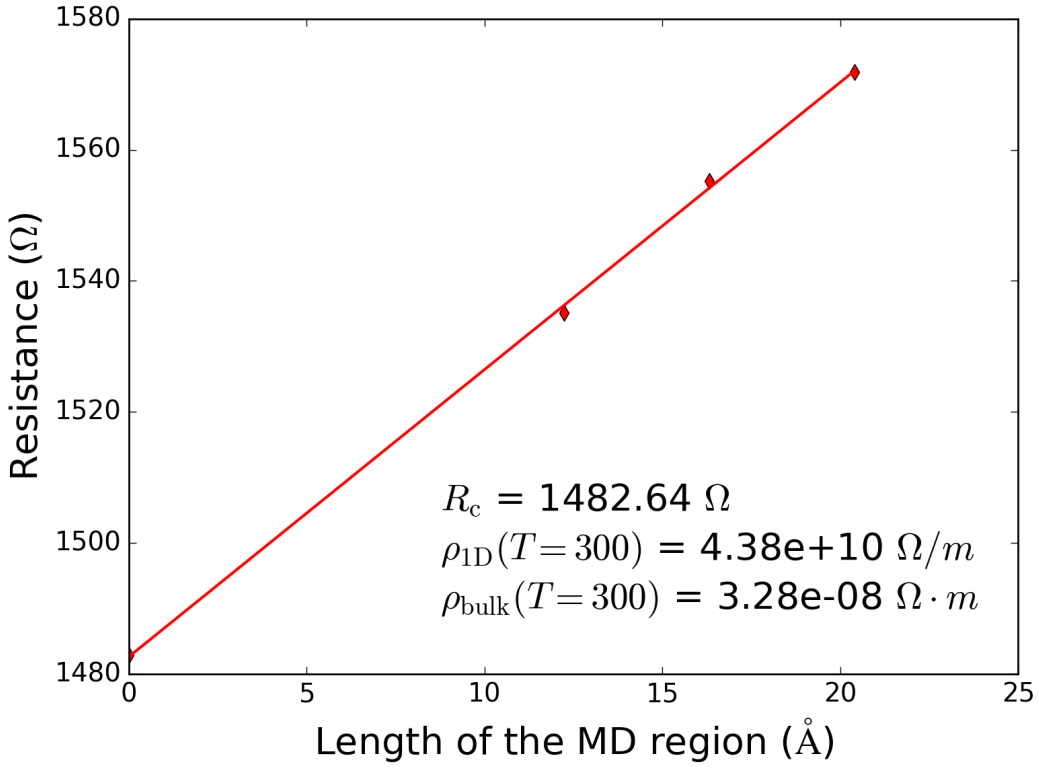
```
14     vector_a = [8.65127469112, 0.0, 0.0]*Angstrom
15     vector_b = [0.0, 8.65127469112, 0.0]*Angstrom
```

the cross section, in this case, is given by $A = \|$vector_a $\|$Œ $\|$vector_b $\|$= 74.84 Å ^{2}. Using this value, and also the values of $\mathcal{L}$ and resistance listed in the table above, one can evaluate the 1D and bulk resistivities at $T = 300$ K. To this end, download `resistivity.py`, and, after entering the values in the table into the **INPUT SECTION** of the script, run it as `atkpython resistivity.py`.

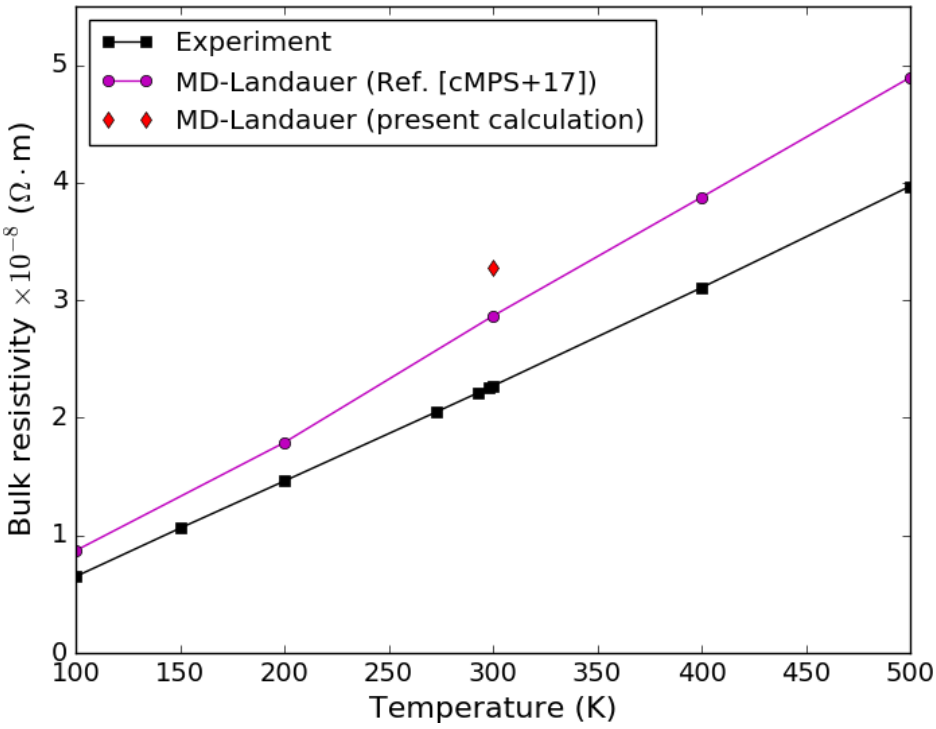
```
6  #-----------------------------------------------------------------------#
7  #                         INPUT SECTION                                 |
8  #-----------------------------------------------------------------------#
9  cs_area = 8.65127469112**2
10 lengths_list = [0,12.235,16.313,20.391]
11 resistances_list = [1482.9,1535.1,1555.2,1571.90]
12 output_filename = 'resistivity.png'
```

The script interpolates the data points by means of the least squares method and provides the values of the contact resistance, as well as the 1D and bulk resistivities at $T = 300$ K:


LXTerminal
File Edit Tabs Help
Atomistix ToolKit 2017.a1 [Build 5d7b2ad]
Resistivity Report (T=300 K)
Contact resistance: 1482.64 Ohm
1D resistivity: 4.38e+10 Ohm/Meter
Bulk resistivity: 3.28e+12 Ohm*Meter


Resistance (Ω)
Length of the MD region (Å)
1580
1560
1540
1520
1500
1480
0
5
10
15
20
25
R_c = 1482.64 Ω
$\rho_{1D}(T=300)$ = 4.38e+10 Ω/m
$\rho_{bulk}(T=300)$ = 3.28e-08 Ω · m

A red diamond in the figure below is the bulk resistivity at $T = 300$ K, and is compared with purple points representing the results of the MD-Landauer calculations shown in Fig. 7(b) of Ref.[7]. Because of the reduced numerical condition employed in this case study, there exists a small difference. The figure also displays black squares representing the experimental result[Page 876, 7], and proves the qualitative agreement with the MD-Landauer result.



References

3.4.10 Electron transport calculations with electron-phonon coupling included via the special thermal displacement method - STD-Landauer

Version: 2017.1

Downloads & Links

Silicon device Inelastic current in a silicon p-n junction Transport calculations with QuantumATK SpecialThermalDisplacement ATK Reference Manual

Phonons have a large influence on the electronic transport properties of semiconductors, but most methods for modeling them are computationally very expensive. In a recent paper; *First-Principles Electron Transport with Phonon Coupling: Large-Scale at Low Cost*[1], Gunst et al. proposed a new method, which is computationally equivalent to traditional 0 K calculations. We call this method **STD-Landauer**. In this case study, we will go through the calculations required to produce one of the figures in the paper, explaining how QuantumATK was used to create this research.

[7] D.ăR. Lide. *Handbook of Chemistry and Physics*. New York: CRC Press, 75th edition, 1997.

[1] Tue Gunst, Troels Markussen, Mattias L.ăN. Palsgaard, Kurt Stokbro, and Mads Brandbyge. First-principles electron transport with phonon coupling: Large scale at low cost. *Phys. Rev. B*, 96(16):161404, oct 2017. arXiv:1706.09290, doi:10.1103/PhysRevB.96.161404.

Specifically, we will re-create figure 2c in the paper[1]. This figure shows the IV characteristics of the short silicon p-n junction, comparing the current with no electron-phonon coupling included, and two different methods for including it. It shows how the electron-phonon coupling effects are very important to include for a correct description of the reverse current.

We will compare the new **STD-Landauer** method to the more well-established Lowest Order Expansion (**LOE**) method, and show that the STD-Landauer can achieve comparable accuracy at much lower computational cost in both time and memory.

Note: QuantumWise case studies are primarily directed at experienced users of QuantumATK. Instructions are deliberately concise in order to focus mostly on the scientific content.

Building the device

We will here use a simple silicon p-n junction, and the workflow for building it is already explained in detail in a tutorial: *Silicon device*. If you do not know how to build such a device, you may go through the steps in the tutorial, applying the modifications mentioned below.

Finding the lattice constant of silicon

As is done in the paper, we will model the structural properties, including phonons, with a classical force field. This means that we first need to find the lattice constant for silicon with the *StillingerWeber_Si_1985* force field. Do a geometry optimization with the following criteria for the forces and stress:

```
max_forces=0.001*eV/Ang,
max_stress=0.01*GPa,
```

This will result in a lattice constant of 5.430952 Å, very close to the database value of 5.4306 Å. Use this optimized bulk configuration as the basis for the following.

Building the pristine device

In this case we go with a device which is actually too short, compared to the screening length at the chosen doping level. However, this choice was made in the paper to allow running of the heavy LOE calculations later on. You should therefore choose 24 layers instead of 52, as specified in the tutorial. You can use the same approach for doping, or alternately you can use the doping widget as described in this tutorial: *NiSi2–Si interface*. However, note that the doping should be n-type on the left and p-type on the right.

Tip: For more details on the implementation of doping in QuantumATK, please consult this note: *Doping methods available in QuantumATK*

Applying the special thermal displacement to the atomic positions

Having built the doped pristine device, we now need to apply the thermal displacements to build the device for calculations with the atomic positions at 300 K. This means that we first need to compute the dynamical matrix, which is independent of the temperature and is used to compute the relevant phonon modes. Afterwards, they are weighted using to the procedure of Zacharias and Giustino[2] and the atoms are displaced accordingly. Use the pristine device, again with the *StillingerWeber_Si_1985* forcefield, and calculate the dynamical matrix. You may follow the procedure in this tutorial: *Inelastic current in a silicon p-n junction*, except for these changes:

•Insert an optimize geometry block with a force tolerance of *0.0001 eV/Ang*

•For the Dynamical matrix, change repetitions to *custom* and *3x3*.

Run the script in a folder called `dyn-mat` to calculate the dynamical matrix; it should take no more than a few minutes. You may also download it here: `dyn-mat.py`. Finally, create the device configuration at 300 K using a simple QuantumATK Python script: `build_std_conf.py`. Note that the QuantumATK method is called `SpecialThermalDisplacement` and you may find the manual page here: *SpecialThermalDisplacement*.

```
# -*- coding: utf-8 -*-
filename = 'device-pristine.hdf5'
filename_dynmat = 'ff-device-si.hdf5'
filesave = 'device-std-300K.hdf5'
device_configuration = nlread(filename, DeviceConfiguration)[-1]
dynamical_matrix = nlread(filename_dynmat,DynamicalMatrix)[-1]
T = 300

device_configuration_STD = SpecialThermalDisplacement(device_configuration, dynamical_
↪matrix=dynamical_matrix, temperature=T*Kelvin)
device_configuration_STD.update()
nlsave(filesave,device_configuration_STD)
```

Run it, and you will have both the pristine and thermally displaced configurations ready for the actual calculations. You may also download the pristine configuration as a script: `device_pristine.py` or an .hdf5 file: `device_pristine.hdf5`. You may also download the thermally displaced configuration here as a script: `device_std.py` or an .hdf5 file: `device_std.hdf5`.

Tip: As you can see here, applying the STD method is very simple, and for the actual calculations, we will not not do it in a separate script.

Calculations

There are several ways in which one may approach these calculations, but here we first do the calculations for zero temperature and bias, and then we use those results as basis for the phonon-including calculations at 300 K, using the special thermal displacement (STD) method, and for finite bias in both cases.

Note:

•Be aware that most of the calculations are impractical to perform on a personal computer and requires some sort of dedicated computational server/supercomputer.

[2] Marios Zacharias and Feliciano Giustino. One-shot calculation of temperature-dependent optical spectra and phonon-induced band-gap renormalization. *Phys. Rev. B*, 94:075125, Aug 2016. doi:10.1103/PhysRevB.94.075125.

•The calculations will generate a lot of files, so we recommend that each type of calculation is done in a separate sub-folder. We provide the names we used and which are assumed in the scripts, but you may of course choose other names and then modify the subsequent scripts accordingly.

Calculations at 0 K and zero bias

First we calculate the PLDOS for the np-junction at zero temperature and bias. Do these calculations in a folder named `0K-zero-bias`. Take the relaxed structure from before, and attach a DeviceLCAOCalculator to it, with the following parameters:

•*k-point sampling:* 9 x 9 x 101.

•*Broadening:* 300 K.

•*Exchange-correlation functional:* LDA.PW.

•*Basis set:* SingleZetaPolarized.

•*Density mesh cutoff:* 75 Hartree

•*Poisson solver:* [Parallel] Conjugate Gradient.

•*Iteration control parameters:*

–Maximum steps: 1000

–Tolerance: 0.0000001

–Number of history steps: 10

Add a **ProjectedLocalDensityOfStates** analysis object, with these parameters.

•*Energy interval:* -1 to 1 eV.

•*Method:* Device DOS.

•*k-point sampling:* 21 x 21.

Finally, you need to make a few manual changes to the *device_iteration_control_parameters* of the full device calculation:

•Remove *tolerance=1e-07,*

•Add *damping_factor=0.01,*

You can also download the script here: `0K-zero-bias.py` Run the script, which should take about 20 minutes on a modern laptop.

Calculations at 300 K and zero bias

For the calculations with the STD configuration, we can do things a little simpler, to ensure that we use the same parameters for the calculator. Do these calculations in a folder named `300K-zero-bias`. Make a copy of the previous script and delete everything related to the structure and calculator, leaving only the PLDOS block. Replace the deleted part with this:

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Two-probe Configuration
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
filename = '../0K-zero-bias.py/0K-zero-bias.py.hdf5'
filename_dynmat = '../dyn-mat/ff-device-si.hdf5'
filesave = '300K-zero-bias.py.hdf5'
device_configuration = nlread(filename, DeviceConfiguration)[-1]
dynamical_matrix = nlread(filename_dynmat,DynamicalMatrix)[-1]
T = 300

device_configuration_STD = SpecialThermalDisplacement(device_configuration, dynamical_
↪matrix=dynamical_matrix, temperature=T*Kelvin)
device_configuration_STD.update()
nlsave(filesave,device_configuration_STD)
```

This will read in the previously calculated Dynamical matrix as well as the configuration, including calculator, from the 0 K and zero bias calculation, and create a new configuration with the thermally displaced atoms and the same calculator as for 0 K. You may download the full script here: `300K-zero-bias.py`

Calculations at finite bias

In this case study, we will first do the self-consistent part of the finite bias calculations, and then afterwards calculate the transmissions. We do this because convergence for the 300 K structure can be a little tricky, and it is easier to fine-tune the convergence for difficult bias-points when doing it in this way.

Tip: In QuantumATK we have implemented an object called *IVCurve*, which automatically calculates the transmission at specified bias values. This is very useful for relatively simple systems and not too high bias values, where there are no problems with the convergence of the SCF loop. In this case, the calculations with the thermally displaced device are more difficult to converge, so we will not use the IVCurve object. For more information on how to use it, see *Transport calculations with QuantumATK*.

The script itself is actually fairly simple, as it simply reads in the zero bias calculation to use as a starting point, and then keeps re-using the previous bias-point as an initial state for the next one. Here we show the script for the un-displaced configuration:

```
# Set bias
# Positive: Reverse and Negative: Forward

pos_bias_list = [0.10, 0.20, 0.30, 0.40, 0.50, 0.55, 0.60, 0.70]*Volt
neg_bias_list = [-0.10, -0.20, -0.30, -0.40, -0.50, -0.60]*Volt

# Read DeviceConfiguration
zero_bias_file = '../0K-zero-bias/0K-zero-bias.py.hdf5'
device_configuration = nlread(zero_bias_file, DeviceConfiguration)[-1]

for bias in pos_bias_list:
    if processIsMaster():
        print "Bias is now: ", bias
    # Get the calculator
    calculator = device_configuration.calculator()

    # Set the bias voltage
    calculator=calculator(electrode_voltages=(bias/2, -bias/2))
```

(continues on next page)

(continued from previous page)

```
    # Attach the calculator and use the old initial state
    device_configuration.setCalculator(
      calculator(),
      initial_state=device_configuration)
    device_configuration.update()
    nlsave('device_bias_%.2f.hdf5' % bias.inUnitsOf(Volt), device_configuration)

# Read DeviceConfiguration
device_configuration = nlread(zero_bias_file, DeviceConfiguration)[-1]

for bias in neg_bias_list:
    if processIsMaster():
        print "Bias is now: ", bias
    # Get the calculator
    calculator = device_configuration.calculator()

    # Set the bias voltage
    calculator=calculator(electrode_voltages=(bias/2, -bias/2))

    # Attach the calculator and use the old initial state
    device_configuration.setCalculator(
      calculator(),
      initial_state=device_configuration)
    device_configuration.update()
    nlsave('device_bias_%.2f.hdf5' % bias.inUnitsOf(Volt), device_configuration)
```

Run these 2 scripts, in folders named `0K-iv` and `300K-iv` respectively. The titles should be self-explanatory:

- `0K-iv-scf.py`
- `300K-iv-scf.py`

They should each take a couple of hours on a 16-core node. Afterwards, we want to calculate the transmissions for each of the configurations and the PLDOS for the most extreme bias points. The first is essentially a loop over the previously calculated files, using the same settings for the Transmission Spectrum as before. This should take a couple of hours on a 16-core node. The PLDOS scripts read in the converged calculation and calculates the PLDOS. They should each take 10-20 minutes using 4 MPI processes on a laptop.

- `0K-iv-transmission.py`
- `300K-iv-transmission.py`
- `pldos-0K-forward.py`
- `pldos-0K-backward.py`
- `pldos-300K-forward.py`
- `pldos-300K-backward.py`

Lowest Order Expansion (LOE) Calculations

Finally, we also need to make the LOE calculations, which takes the inelastic contributions into account through an approximation of the full electron-phonon coupling matrix. This approximation is valid when the Density of States (DOS) varies slowly around the Fermi level. For this, we need to calculate the Hamiltonian Derivatives, which is a fairly heavy calculation. We therefore sample the electronic Brillouin zone in the Γ point only. This is a good approximation in this case, as tunneling is the dominant effect and is in turn dominated by transmission at Γ. Note that this reduces the computational time by a factor of 9 compared to using the same k-point density as in the STD calculations.

Afterwards, we calculate the actual electron-phonon couplings, a calculation which can in general be very time-consuming, due to the many possible combinations of k- and q-points.

Tip: For more information on how to calculate the full electron-phonon coupling, see f.ex. the tutorial: *Phonon-limited mobility in graphene using the Boltzmann transport equation*.

We need to set up a calculator similar to before, with only the k-point sampling changed, followed by a calculation of the Hamiltonian derivatives. Do these calculations in a folder named `loe`. You can use a script similar to before, where you read in the calculator and modify it, or set everything up from scratch in a new script. Here we have chosen to read in the configuration from an existing file, but setting up the calculator anew. The Hamiltonian derivatives should look like this:

```
# ------------------------------------------------------------
# Hamiltonian Derivatives
# ------------------------------------------------------------
hamiltonian_derivatives = HamiltonianDerivatives(
    configuration=device_configuration,
    repetitions=(3, 3, 1),
    atomic_displacement=0.01*Angstrom,
    constraints=[0,1,2,3,44,45,46,47],
    use_equivalent_bulk=False,
    )
nlsave(filesave, hamiltonian_derivatives)
```

You may download the script here: `ham-der.py`. Ensure it points at the right files and run it. In our case, it took about 5 hours on a total of 48 cores with 1 TB of total memory.

Afterwards, we need to run the actual LOE calculations. We use only 1 q-point, in accordance with the paper, which is a good approximation in this case, as the tunneling is also dominated by the Γ-point in the phonon Brillouin zone. This also gives a more direct comparison with the STD method. If the goal is to get as accurate LOE calculations as possible, the number of q-points sampled, should always be converged for the system of interest. You may download the script here: `loe.py`. Make sure it points at the right files, and run it. This should take no more than a few minutes on a laptop.

Tip: For more information on the different parameters of the inelastic transmission spectrum or further discussion of the k- and q-dependence of the transmission in this system, you may consult this tutorial: *Inelastic current in a silicon p-n junction*.

Computational timings

The main advantage of the STD method over the LOE method is computational efficiency, and we will therefore present the timings in a more systematic way. In the table, we show the timings for the full IV curve for each method, normalized to the total number of core-hours on equivalent machines.

Table 3.2: Computational timings in core-hours

	Noninteracting	STD	LOE
Time spent on IV Curve:	28 h	129 h	468 h

We see here that the STD method takes longer than the noninteracting calculation, which is mostly due to convergence being more difficult. In this case study, we have chosen a conservative approach which improves convergence, but at the expense of longer calculations. It is probably possible to find other settings which also converge, and use less time. With the parameters we have chosen for the LOE calculation, almost all the time is spent on calculating the Hamiltonian derivatives, and very little is spent on computing the actual electron-phonon interactions afterwards. However, this is in general not true, and the LOE part itself increases very quickly with the number of k- and q-points included. Additionally, as noted above, we have reduced the calculation time for the hamiltonian derivatives by a factor of 9 by reducing the k-point sampling to just the Γ-point.

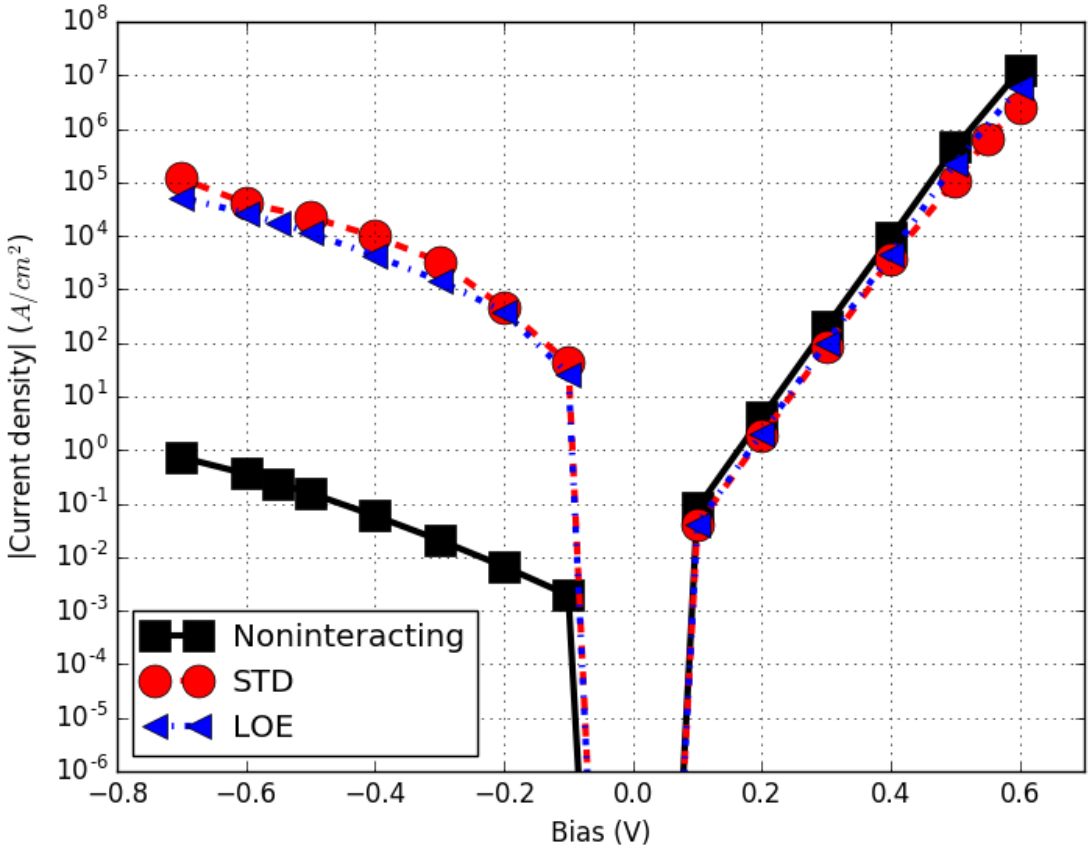
Note: Note that the Hamiltonian derivatives were computed on special nodes with extra memory, which is an added complication of such demanding calculations.

Analysis and discussion

We have now calculated the current as a function of bias using three methods:

- **Noninteracting**: uses a pristine configuration and does not incorporate phonons in any way.
- **SpecialThermalDisplacement (STD)**: Uses a single thermally displaced configuration to account for phonons. This is a new method introduced in the paper this case study is based on.
- *Perturbation Theory (PT)* using the **Lowest Order Expansion (LOE)**: Approximate approach to include the full electron-phonon contribution to the current.

The main point of these calculations is to show that the STD method is as accurate as the LOE method, in this case, but at a much lower computational cost, as there is no need to compute the very expensive dH/dR matrix. We will now plot our data and see how they compare. Download this plotting script, make sure it points at the right files and run it: `plot-iv.py`. You should get a plot looking like this:



We see how inclusion of the electron-phonon coupling (EPC) drastically increases the reverse current. We also see that both the STD and LOE approaches show this increase, and are in clear agreement about the effect and importance of the electron-phonon coupling. Note that for the longer device, which is also treated in the paper[Page 876, 1], the results are also in good agreement with experiments. The increase in current reduces the $I_{\mathrm{ON}}/I_{\mathrm{OFF}}$ figure-of-merit to be on the order of 1 when going below a bias of ± 0.5 V. This means that the current will be essentially the same for both forwards and backwards bias, and the device no longer acts as a diode at low bias. See the table for details:

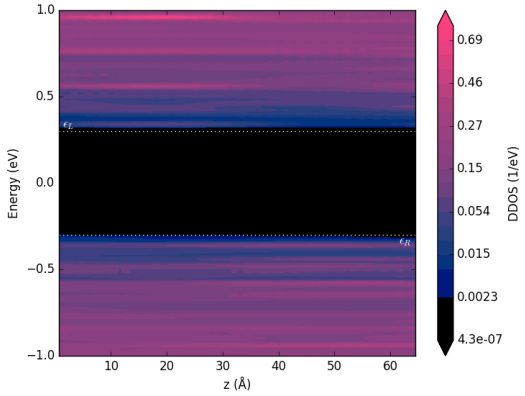
Table 3.3: $I_{\mathrm{ON}}/I_{\mathrm{OFF}}$ figure-of-merit.

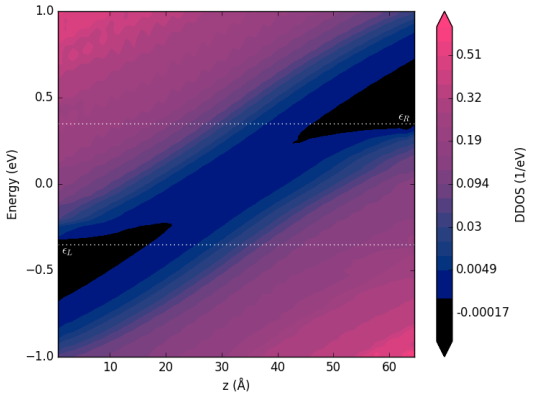
Bias [V]	Noninteracting	STD	LOE
± 0.5	$8 \cdot 10^6$	5	20
± 0.6	$7 \cdot 10^7$	58	200

Note: The $I_{\mathrm{ON}}/I_{\mathrm{OFF}}$ figure-of-merit is a basic property of diodes and describes how effective the diode is at suppressing current at reverse bias. Specifically, it is the current under forward bias divided by the absolute value of the current under a reverse bias of equal magnitude. So if it is 10 at 0.5 V, it means that the current at 0.5 V forward bias is 10 times larger than the reverse current under a reverse bias of -0.5 V.

In order to understand why phonons are very important under reverse bias, and negligible under forward bias, we need to consider the two cases in more detail. Specifically, the mechanism of transport differs between forward and reverse bias. Under forward bias, the bands are essentially flat and carriers can pass unhindered from one electrode to the other. Under reverse bias, there are no states around the Fermi level in the central region and carriers can only travel between

the electrodes by tunneling. This can also be seen from a comparison of the PLDOS at forward and reverse bias, see figures below. We show the PLDOS without electron-phonon coupling included, to make the features more clear. The left/right PLDOS is at forward/reverse bias.





To a first approximation, the band-to-band tunneling probability decreases when $|k|$ increases, and increases when the carrier energy increases. Therefore, the tunneling probability will be higher if the carrier can increase its energy without increasing the momentum. This leads us to the role of phonons. This explains why we can use just the Γ-point for both k- and q-points, as the interaction gives maximum tunneling probability there. The fact that the transmission peaks at the Γ-point, is shown in more detail in the tutorial: *Inelastic current in a silicon p-n junction*.

The STD method compared to the LOE method

In this case study, we have used two quite different methods to account for the electron-phonon interaction. The LOE method uses perturbation theory and Greens functions to calculate the transitions between states with varying k- and q-values. The STD method simply displaces all the atoms according to a weighted average of the phonons, so that the correct thermal average of the current is obtained. It does not model the electron-phonon interactions explicitly. Beforehand, there was no guarantee that the agreement would be as good as shown here, and this will not be true in all cases. Especially if the relevant processes require momentum transfer, the STD method should not be expected to give as good results as the LOE method. However, as the two methods rely on different approximations and assumptions, it is also no guarantee that the LOE method will always give better results than the STD method. Additionally, the STD method is much faster and less memory-demanding.

References

3.5 1D Nanostructures

3.5.1 Transport in graphene nanoribbons

Version: 2016.0

Research on fundamental aspects and device applications of graphene is a very active field. This tutorial illustrates how QuantumATK can be utilized to study various applications related to graphene nanoribbons, ranging from a simple, infinite sheet of graphene to more complex junctions.

The main purpose of the tutorial is to demonstrate the basic work-flow in QuantumATK, how to build systems with nanoribbons and compute fundamental quantities such as band structures and transmission spectra using QuantumATK.

Introduction

The geometry of graphene is simple and regular, and the infinite, planar structure can easily be created, either by hand or by taking a single layer from the crystal structure of graphite.

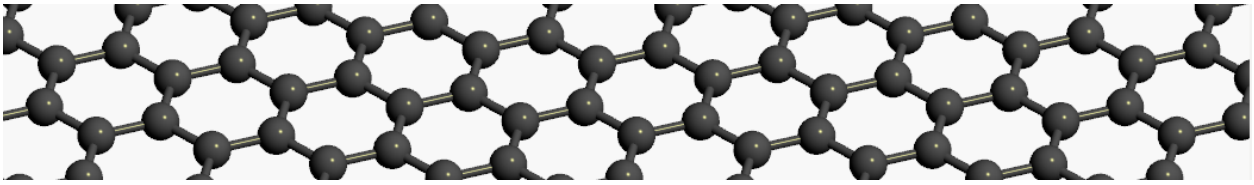


Fig. 3.73: A graphene sheet.

However, to create a device-like structure, the infinite sheet must be cut into a suitable shape. A common shape for electronics applications is a so-called graphene nanoribbon (GNR). These can be quite cumbersome to set up in an effective manner, not least considering the hydrogen passivation of the edges, which is needed in finite structures in order to satisfy all carbon valence electron bonds.

QuantumATK has a specific graphene nanoribbon Builder tool, and you will in this tutorial learn how to use it, and how to customize the structures it produces to make more advanced shapes of graphene. This will allow you to

1. calculate the band structure of ideal, infinite graphene nanoribbons, of either armchair or zigzag type, as a function of the ribbon width;
2. introduce doping or defects, on the edge or elsewhere, and study the influence on the transmission spectrum;
3. use basic ribbons as building blocks for your own device ideas, such as e.g. Z-shaped nanoribbon junctions.

This tutorial will show examples of each type of system.

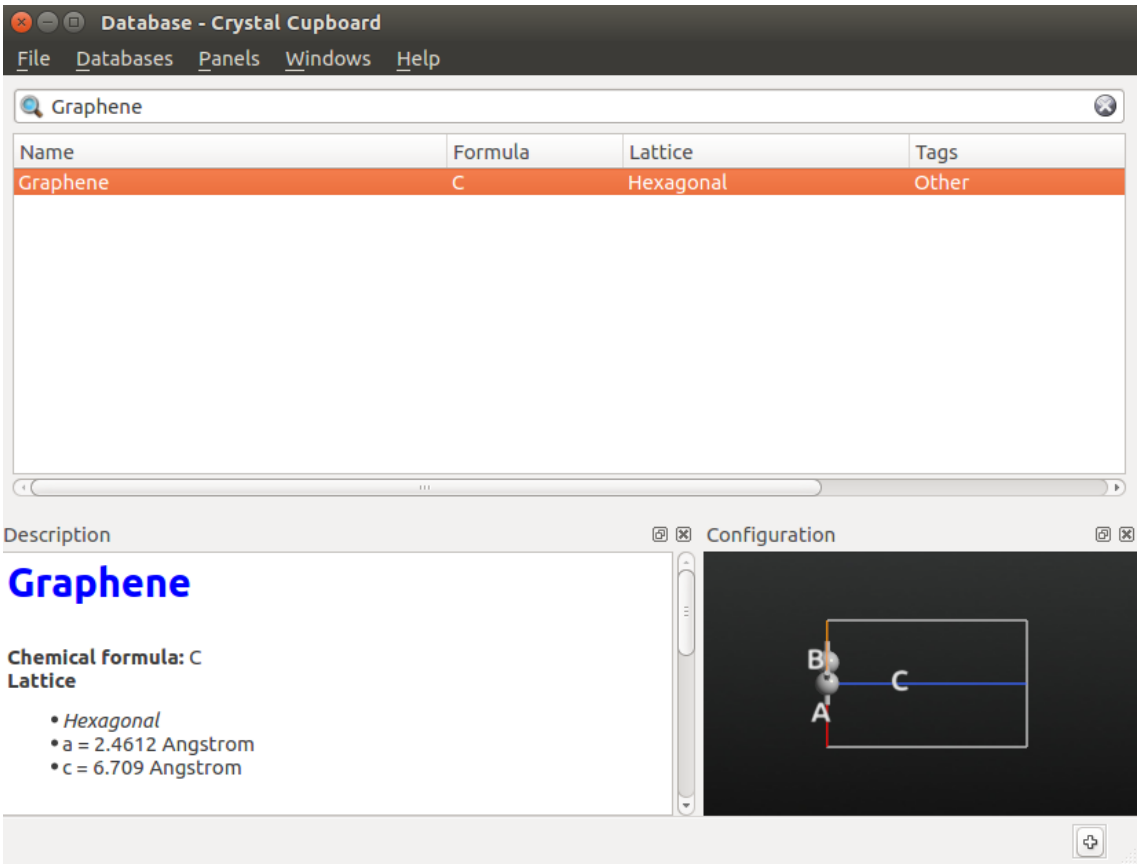
Band structure of 2D graphene

Before starting with the nanoribbons, you will do a quick introductory calculation of the band structure of an infinite 2D sheet of graphene. This will also serve as a way to refresh the basic concepts and work flow in QuantumATK.

Constructing the graphene

Start QuantumATK, create a new project and give it a name, then select it and click **Open**. Launch the builder by clicking the **Builder** icon on the tool bar.

Graphene, and many other crystals, molecules and fullerenes, are available in the built-in database in QuantumATK. In the Builder, click *Add* → *From Database* and locate Graphene (while you are typing, you will see that graphite is included in the database too).



Double-click the line to add the structure to the Stash, or click the in the lower right-hand corner.

Now send the structure to the **Script Generator** by clicking the Send To icon in the lower right-hand corner of the window.

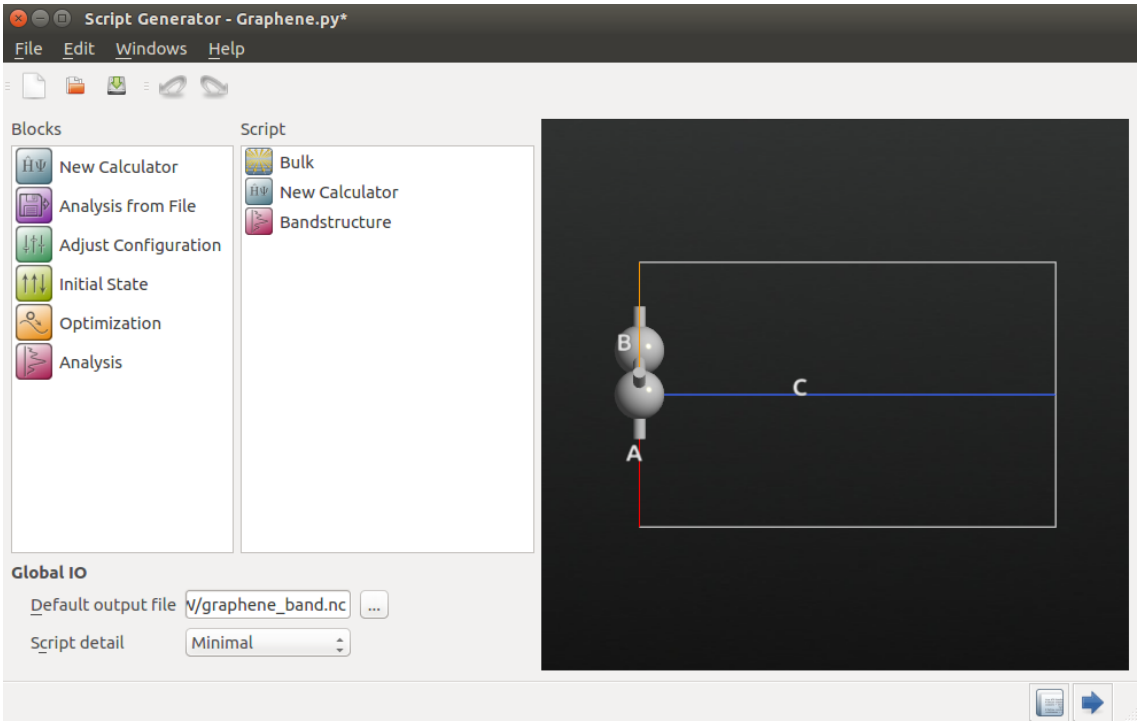
Band structure: k-point sampling

You will now set up the band structure calculation. It is instructional for this particular system to compare the results of a DFT calculation with the semi-empirical method; as you will see, they are very similar.

The first step is to use the Extended Hückel method in the ATK-SE calculator to determine the required k-point sampling. Using this faster method saves time compared to doing the same investigation with DFT. The results should, for this aspect, be very similar.

In the **Script Generator**, change the default output file to `graphene_band.hdf5` and add the following blocks to the script:

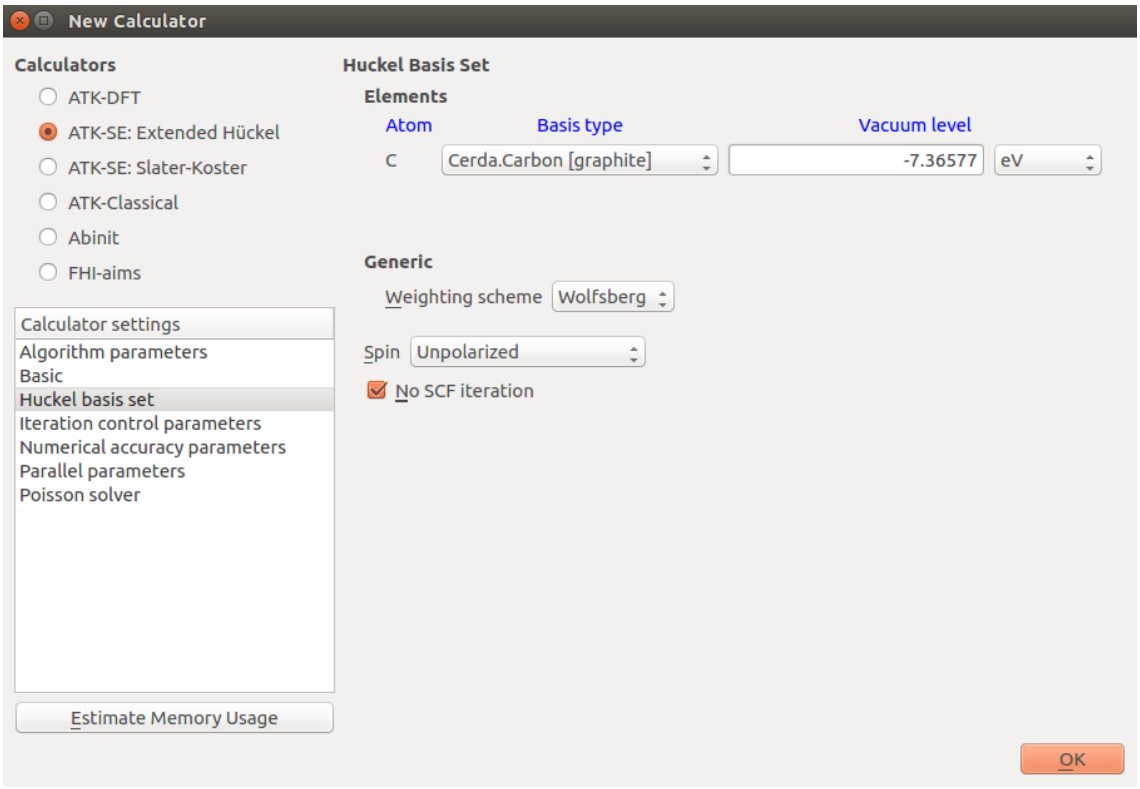
- **New Calculator**
- **Bandstructure**



You should the default parameters, except the for the Hückel basis set parameters: Open the **New Calculator** by double-clicking it.

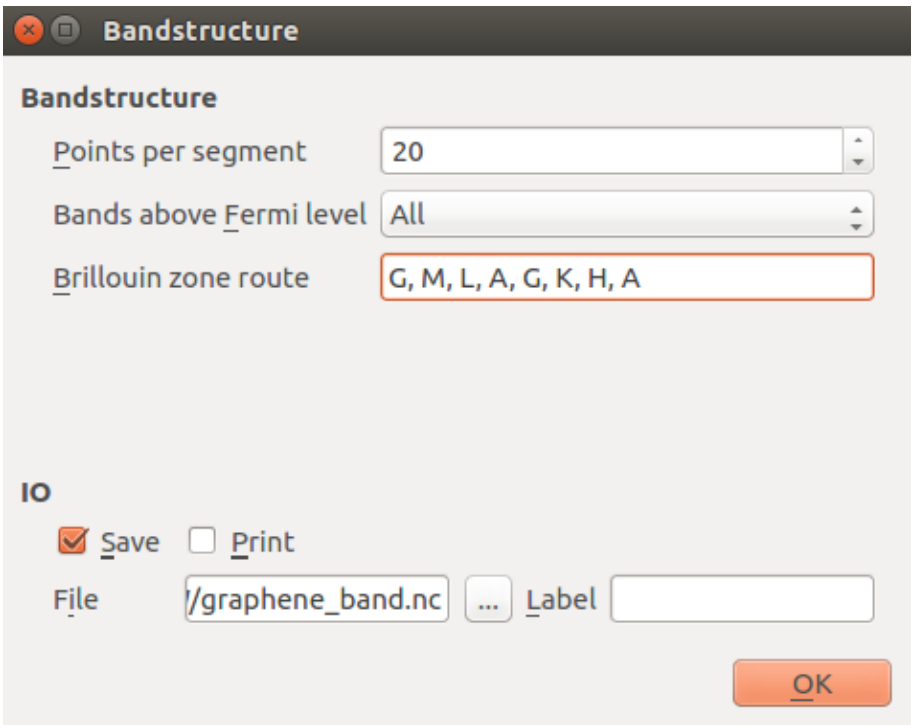
•Select the ATK-SE: Extended Hückel calculator.

•Under **Huckel basis set**, select the Cerda.Carbon [graphite] basis set.



Make sure to use this basis set whenever studying graphene (or carbon nanotubes) with the extended Hückel method. Close the dialogue by clicking **OK** in the lower right-hand corner.

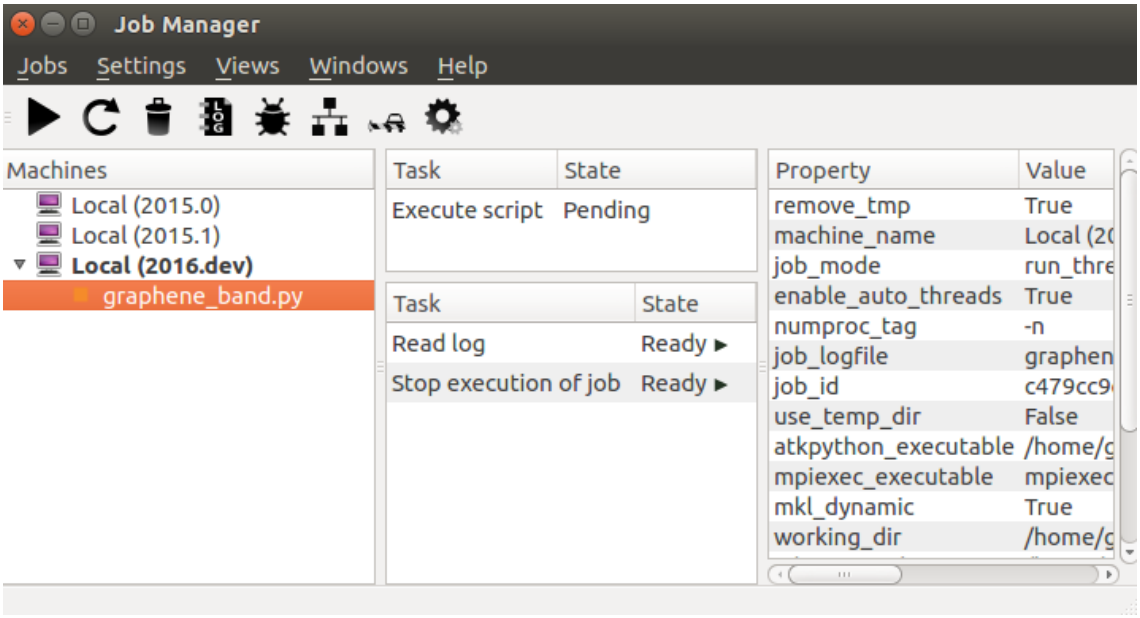
Next, open the **Bandstructure** block. A pre-defined route in the Brillouin zone is set up, corresponding to a three-dimensional hexagonal Bravais lattice. Leave it like that for now; this point will be discussed below.



Note: The different components of the script inherit the global output file name. It is possible to specify a unique file name for each analysis block, but usually it makes most sense to collect all results in the same file.

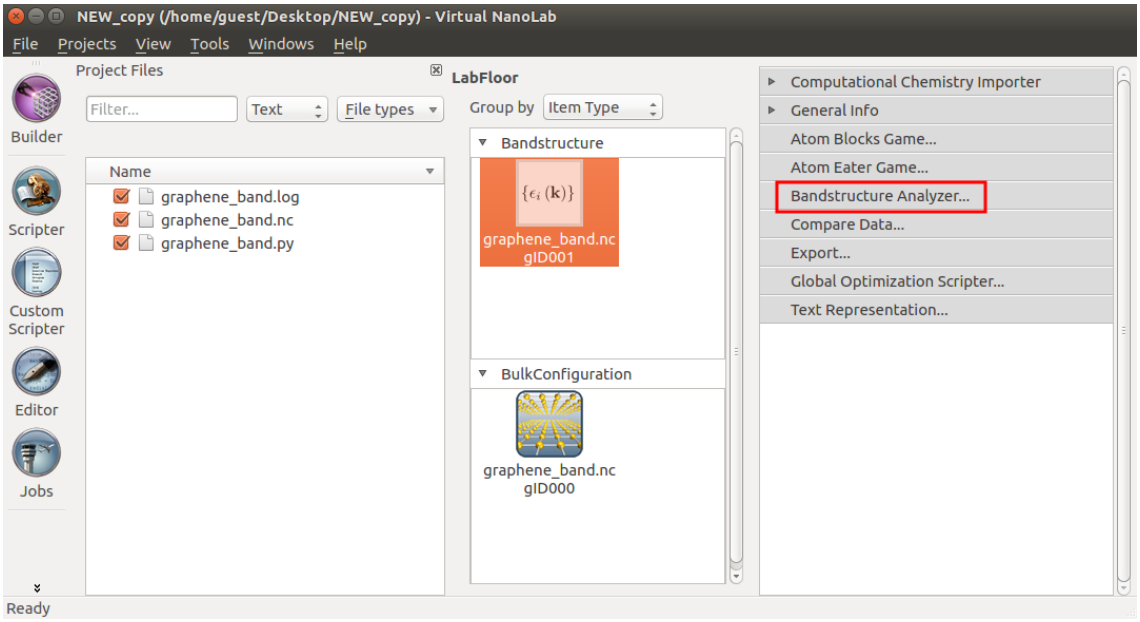
You are now ready to run the calculation: Send the script to the **Job Manager** by clicking the Send To button and by selecting Job Manager in the menu that shows up.

In the Job Manager, click to run the job. The calculation takes only seconds to complete.

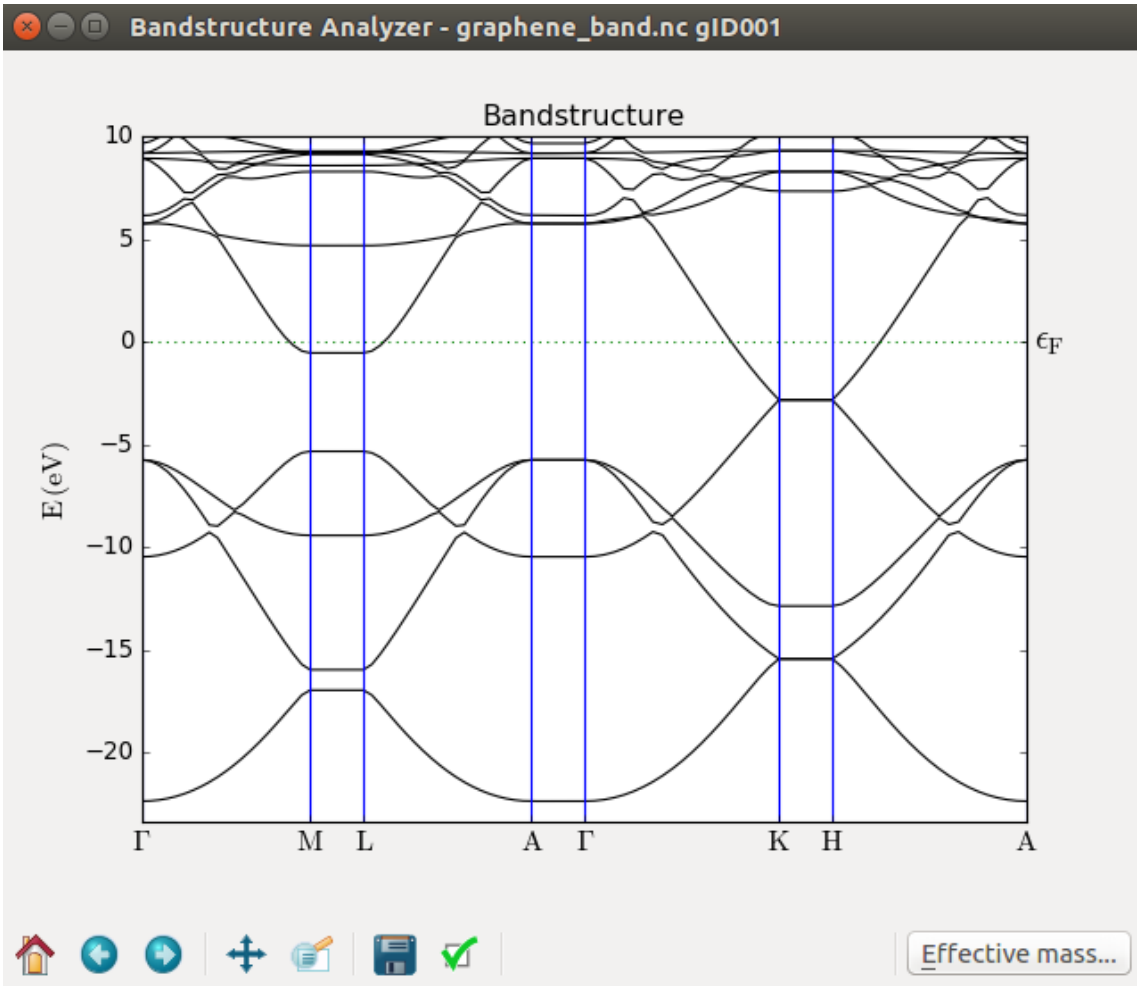


Analysis of the results

Once the calculation is finished, the `graphene_band.hdf5` file should be visible in the Project Files list in QuantumATK. On the **LabFloor**, select Item Type from the drop-down menu as shown in the below figure. Select the Bandstructure item, and click the **Bandstructure Analyzer** button in the right-hand plugins panel.

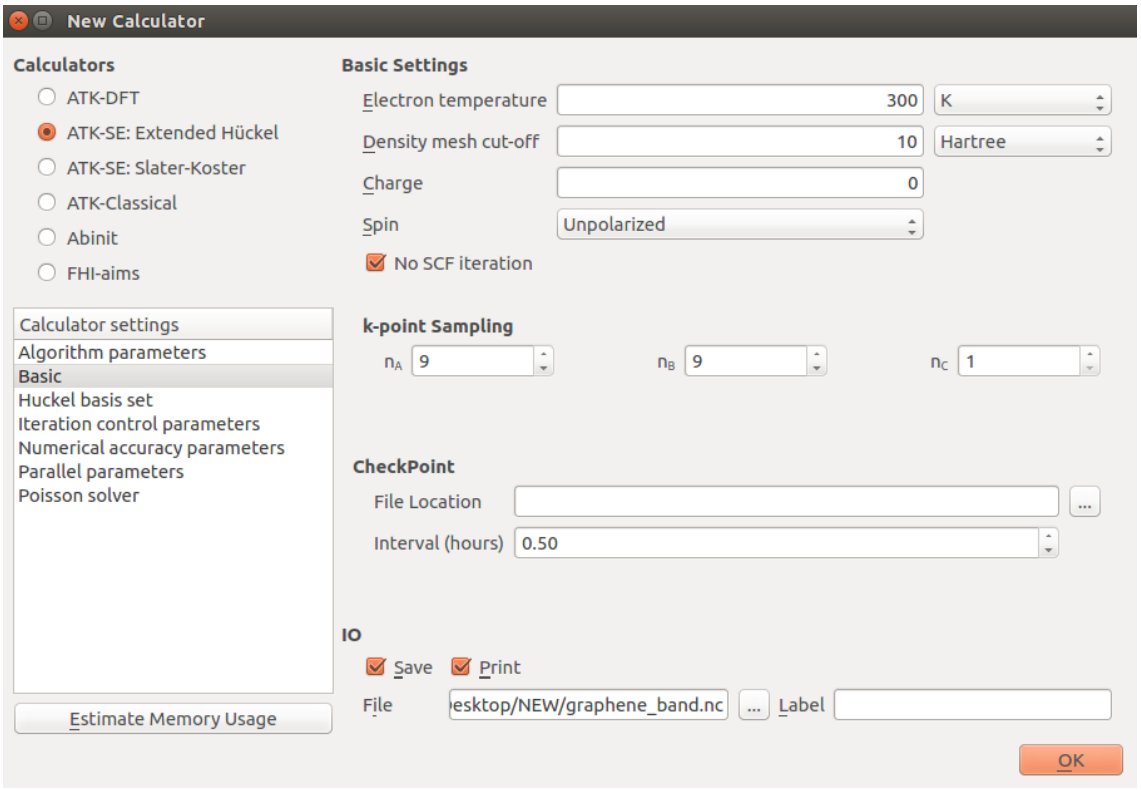


The calculated band structure is visualized. However, on close inspection, you can see that is does not appear to be correct, since the Brillouin zone K point is supposed to coincide with the Fermi level.



The cause of the poor band structure is insufficient k-point sampling; the default is to have just one point in each direction, and this fails to properly describe the graphene electronic structure.

Therefore, open the minimized **Script Generator** window again, open the Calculator block, and enter 9x9x1 k-points under Basic settings.



You should also adjust the symmetry points used for the band structure calculation to avoid the flat segments, which correspond to paths out of the planar graphene Brillouin zone. The only relevant k-points in a 2D hexagonal lattice are G=(0,0,0), K=(1/3,1/3,0), and M=(0,1/2,0), in units of the reciprocal lattice vectors.

This can easily be done in the Scripter (by double-clicking the Bandstructure block and editing the path), but to demonstrate how easy it is to use a script produced by the Script Generator as a template, you will now learn to do that by hand instead!

Send the script to the **Editor** by clicking the icon in the Scripter. Locate the line with the symmetry points, and edit it as shown below.

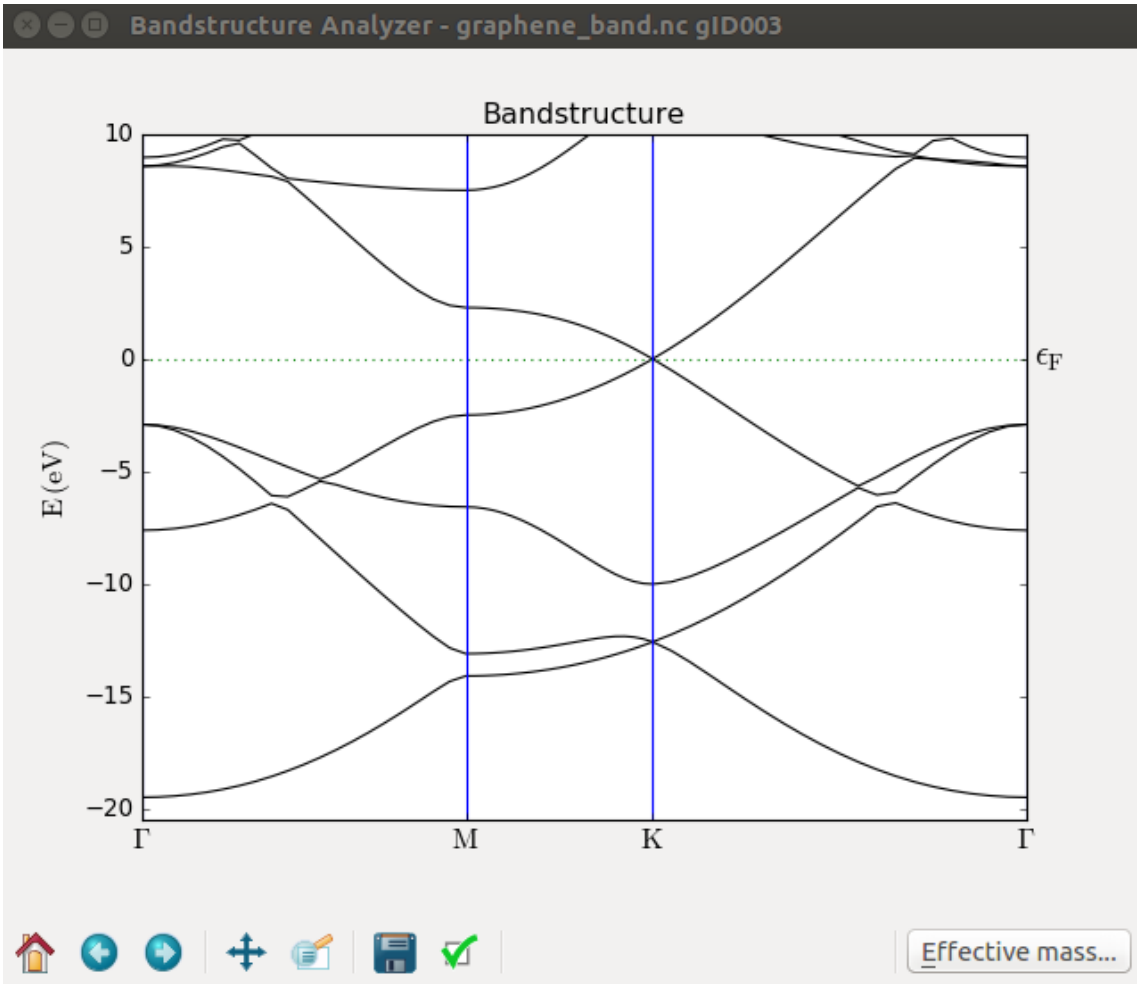
```
bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('graphene_band.nc', bulk_configuration)

# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'M', 'K', 'G'],
```

(continues on next page)

(continued from previous page)

```
    points_per_segment=20,
    bands_above_fermi_level=All
    )
nlsave('graphene_band.nc', bandstructure)
```

Run the calculation by sending the script from the **Editor** to the **Job Manager**. When the calculation finishes, a new Bandstructure object will be located on the **LabFloor**. Select it and plot it as before. The Fermi level is now exactly at the K point, as illustrated below.

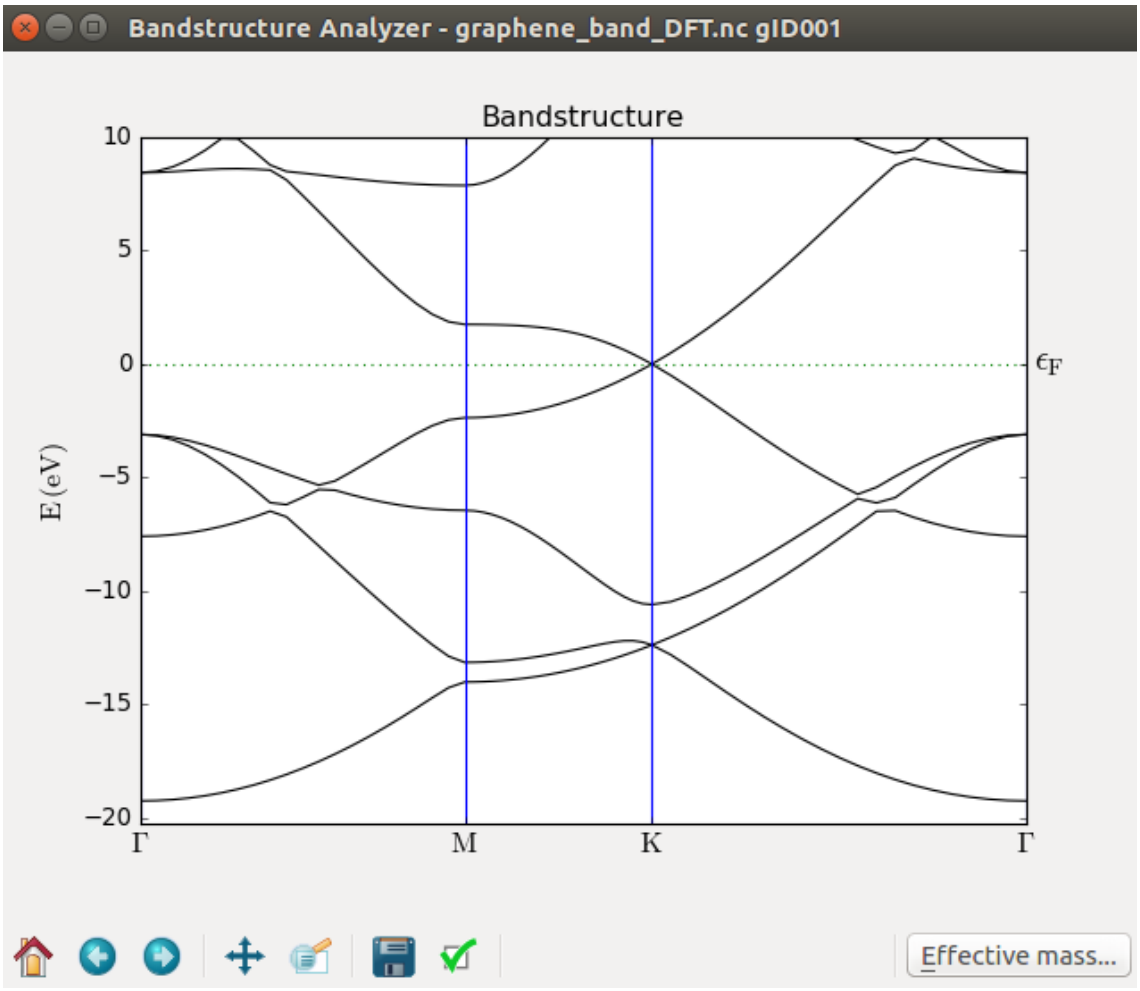


Calculate the band structure using DFT

You now know the required k-point sampling, and you can use this also for the DFT calculation. To switch method, return to the **Script Generator**, open Calculator block, and select the **ATK-DFT** calculator. Set the k-point sampling to 9x9x1, but leave all other parameters at their default values.

Tip: In case you closed the Script Generator window, you can reuse the geometry from the saved HDF5 file by dropping one of the bulk configuration inside it on the Scripter icon on the toolbar.

Keep again the same name for the output HDF5 file, and run the calculation. This will add yet another Bandstructure item to the **LabFloor**, which can be plotted like before. On inspection, you will find that the results are very similar to the Hückel band structure. The Hückel method therefore has good accuracy for this system and will be used for the further examples, mainly to save time; the interested reader is encouraged to verify that the results obtained with DFT are very similar.



Note: As it turns out, getting an accurate alignment of the K point to the Fermi level in graphene is not just a simple matter of having a lot of k-points. The results for 12x12x1 or 13x13x1 k-points are, for instance, considerably worse than those for 10x10x1. There is a general trend that more k-points are better, but on the other hand, 3x3x1 points are better than almost all other values up to 20x20x1. This behavior can be understood by studying how the k-points in the Monkhorst–Pack grids are distributed around K=(1/3,1/3,0); the ideal situation is to have several points close to or at the Fermi level, although the latter is less favorable than having two points close by.

Band structure of an armchair ribbon

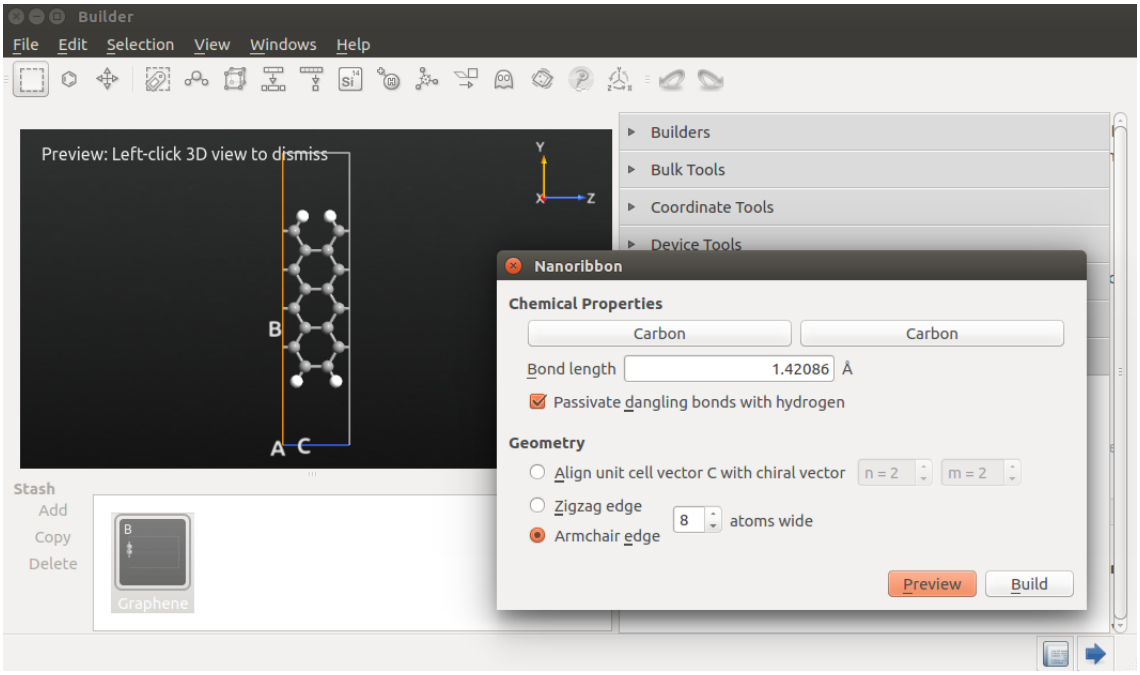
Having investigated the band structure of pure graphene, the focus is now on graphene nanoribbons (GNR), which are finite, narrow strips of graphene cut out from the infinite 2D sheet. There are two primary ways to cut out such a ribbon – these two structures are known as armchair or zigzag ribbons, a nomenclature that refers to the shape of the edges. Notably, an armchair ribbon is an unrolled zigzag nanotube! Armchair ribbons are predicted to be semiconducting, and you will now compute the band structure of such a ribbon to confirm this.

Building the archair ribbon structure

Generating nanoribbons with QuantumATK could not be easier – there is a **Builder** plugin specifically designed for this task. Launch the Builder and click *Add* → *From Plugin* → *Nanoribbon*. This launches the Graphene Ribbon tool.

In the widget that opens, you can design many different types of nanoribbons, not just graphene but also B-N, etc. By selecting a linear combination nA+mB of the two unit cell vectors A and B in the graphene plane, ribbons of any chirality can be constructed (the chirality label comes from the fact that these structure correspond to an unrolled (n,m) nanotube). For simplicty, you can also directly choose to build armchair or zigzag ribbons, and specify their width (which must be even for zigzag ribbons).

Select an armchair ribbon of width 8 (carbon atoms), and click **Builder** to add it to the Stash.

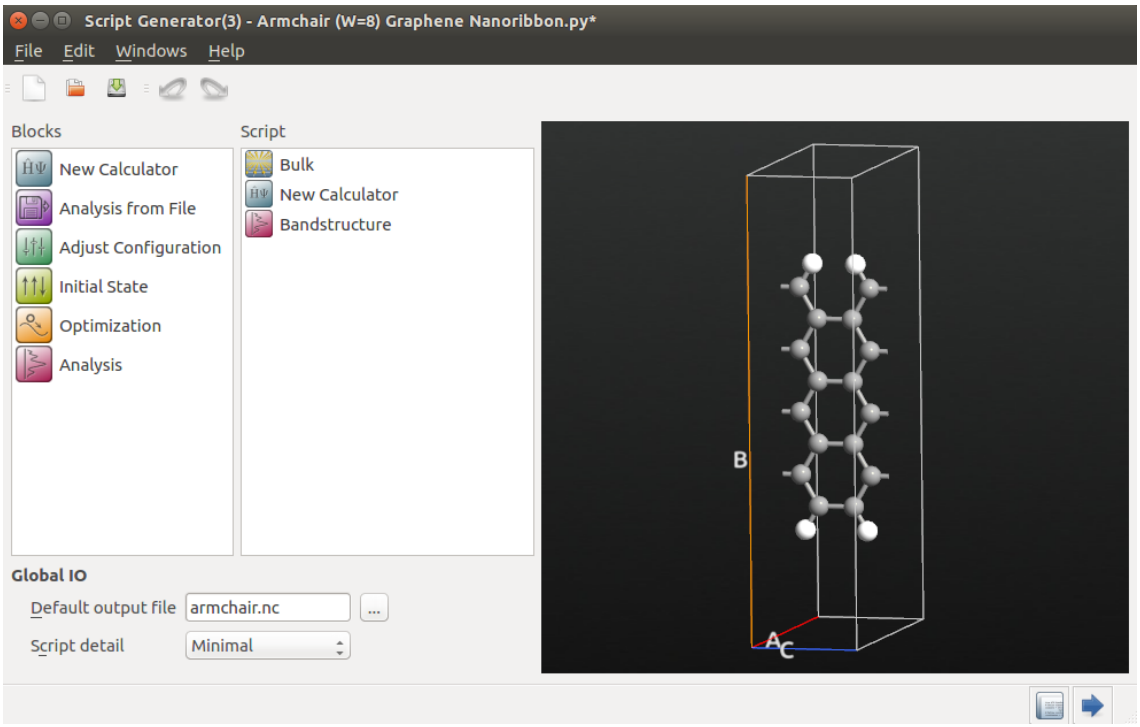


Send the structure to the **Script Generator**.

Band structure

Like before we will use the Extended Hückel method for the calculations. Change the default output filename to `armchair.hdf5`, and add the following two blocks to the script:

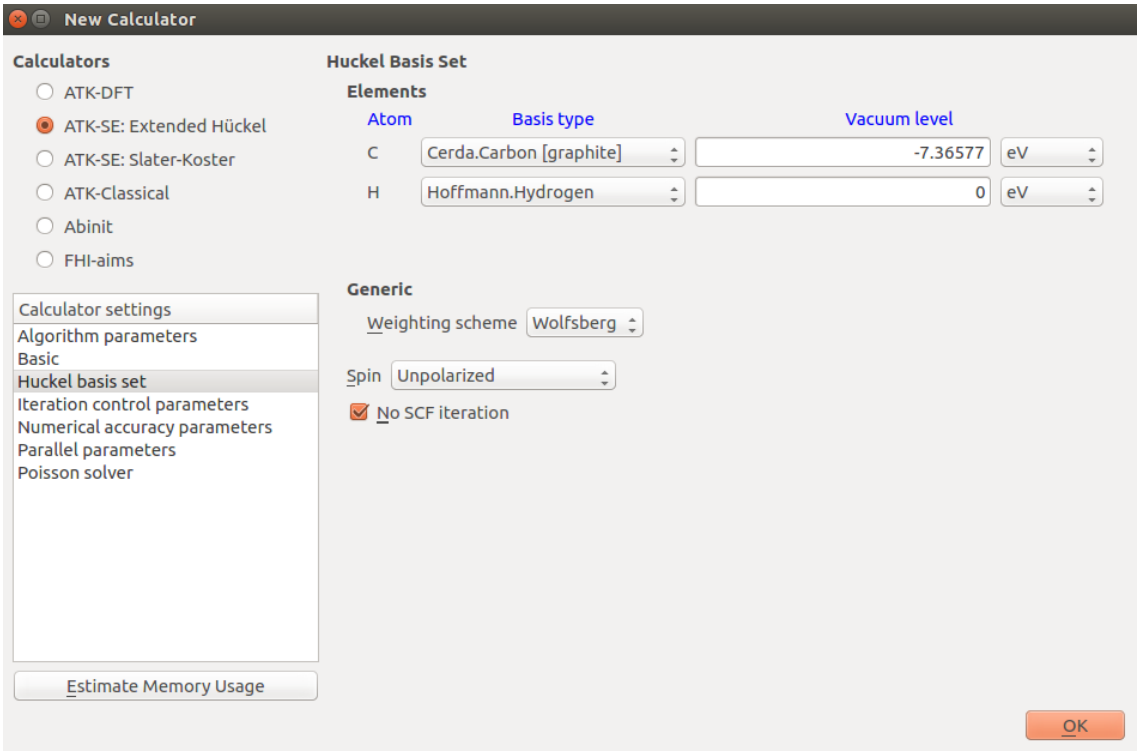
- **New Calculator**
- **Bandstructure**



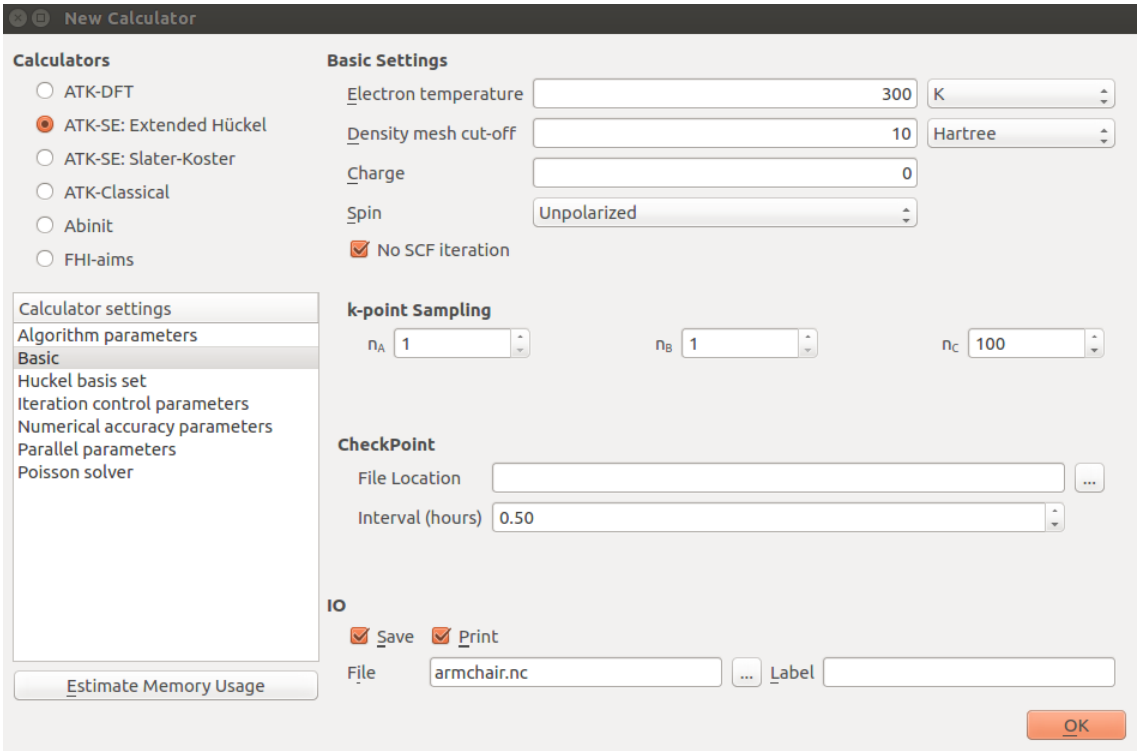
Open the Calculator block to edit it:

•Select the ATK-SE: Extended Hückel calculator.

•Select the Cerda.Carbon [graphite] basis set for carbon. The default Hoffmann basis set should work well for the hydrogen.

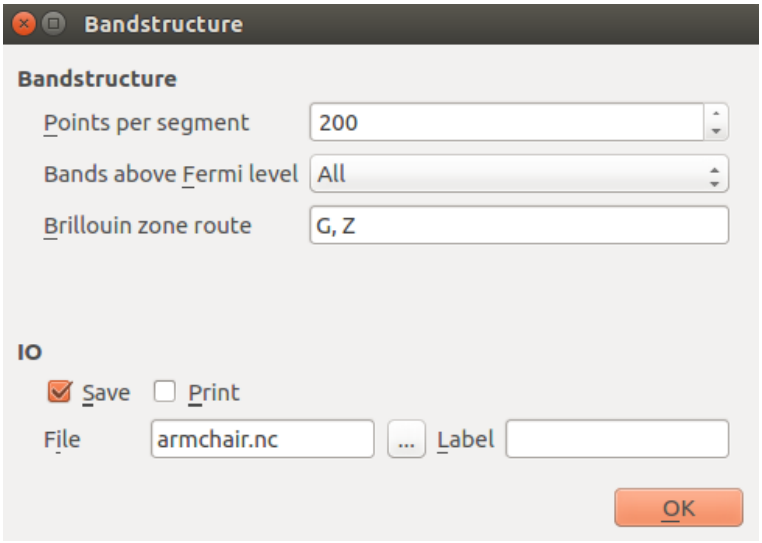


In this case you need only 1 k-point in the directions perpendicular to the direction of the ribbon, since the system is finite in these direction, while the system is periodic in the C direction. A large number of k-points is needed in this direction, since the unit cell is quite short and graphene has a rather peculiar point-like Fermi surface as we saw above; 100 points should be sufficient:



The default suggested route in the Brillouin zone from G=(0,0,0) to Z=(0,0,1/2) for the band structure is indeed the

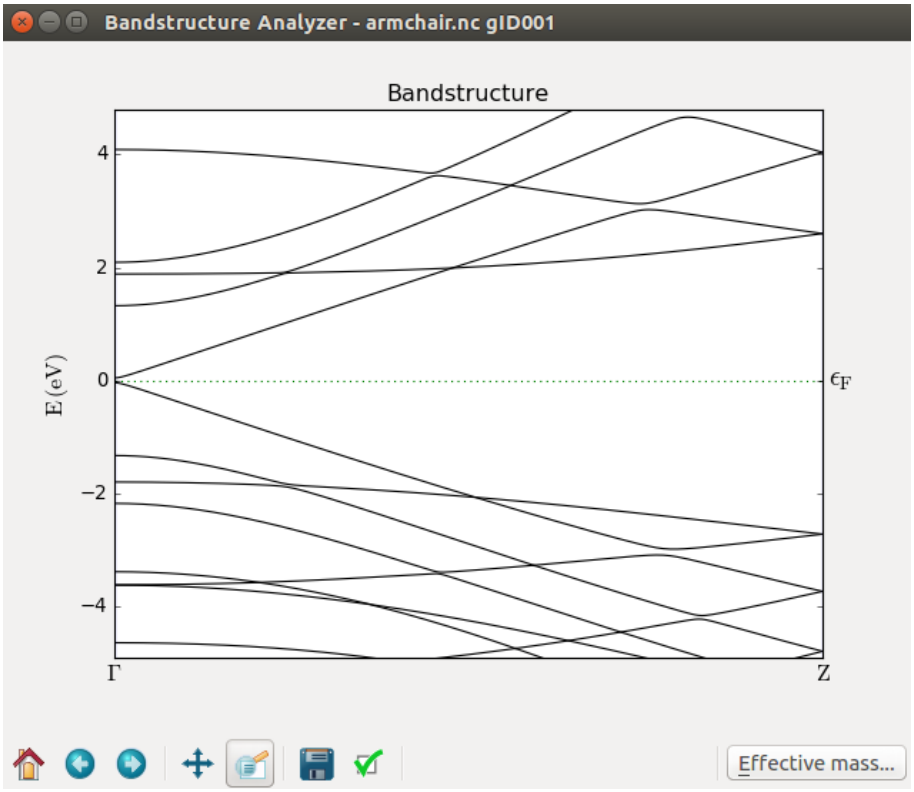
appropriate one, so you just need to increase the number of points to 200 to obtain nicely smooth curves:



Run the calculation using the **Job Manager**.

Analyzing the results

Return to the main QuantumATK window and use the **Bandstructure Analyzer** to plot the Bandstructure item from the file `armchair.hdf5`. The computed band structure is shown in the figure below. By zooming in a bit you should see a small band gap band gap at the point. The armchair nanoribbon is therefore semiconducting.



As an extension of this exercise, you are recommended to redo the calculations for different types of ribbons (armchair/zigzag) and for different widths. The band gap is strongly dependent on both factors!

In the next section, you will investigate a zigzag ribbon, which is metallic rather than semiconducting.

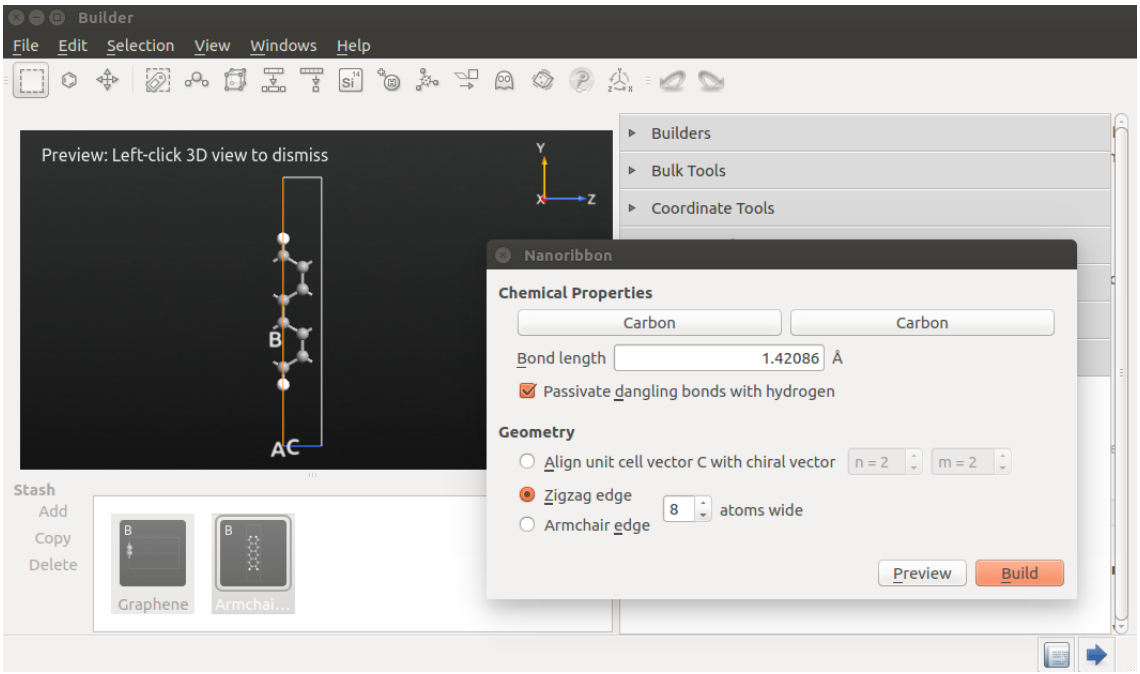
Transport properties of a zigzag nanoribbon

In this section you will do some transport calculations. One of the most basic transport system that can be made with graphene is a perfect, infinite zigzag nanoribbon. Such ribbons are metallic and display no elastic scattering, i.e. perfect conductivity. In the second part you will investigate what happens when scattering is introduced by distorting the edges of the ribbon.

Pristine ribbon

In a pristine device system there is no scattering from the central region and, as illustrated in the following, you can obtain the transmission spectrum from the properties of the electrode.

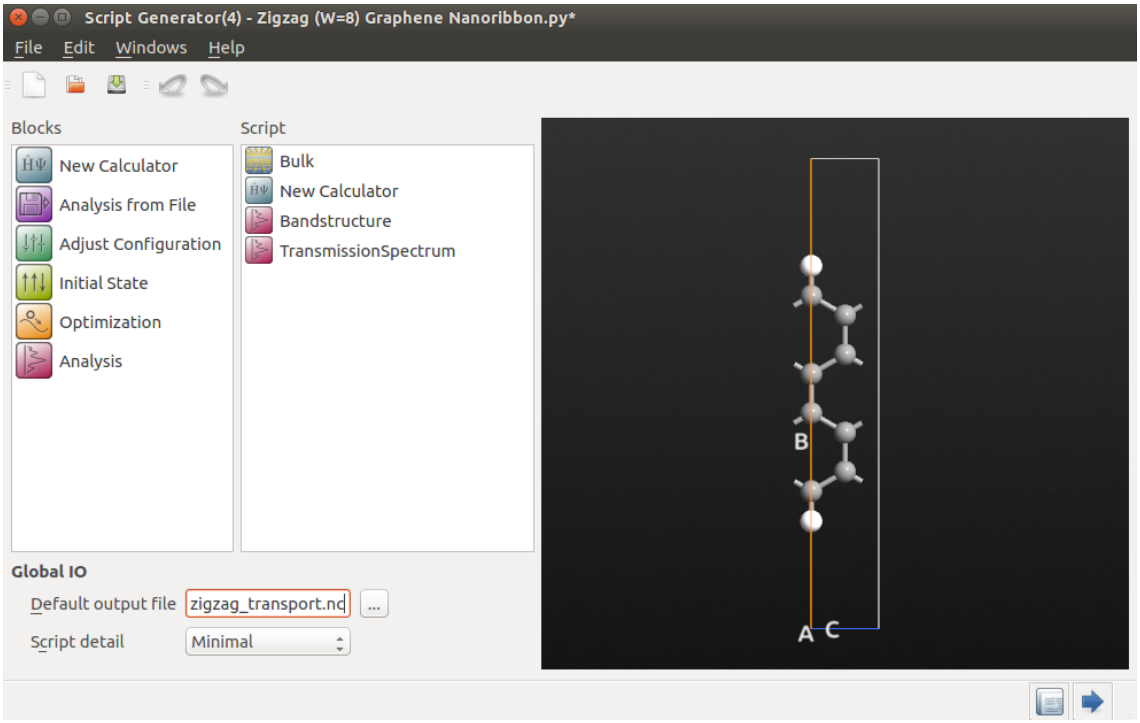
Return to the **Builder** and open the Nanoribbon widget from *Add* → *From Plugin* → *Nanoribbon*. Build a zigzag nanoribbon with a width of 8 carbon atoms.



Transfer the structure to the **Script Generator** and add the following blocks to the script.

- **New Calculator**
- **Bandstructure**
- **TransmissionSpectrum**

Change the default output file to `zigzag_transport.hdf5`.



Edit the Calculator settings:

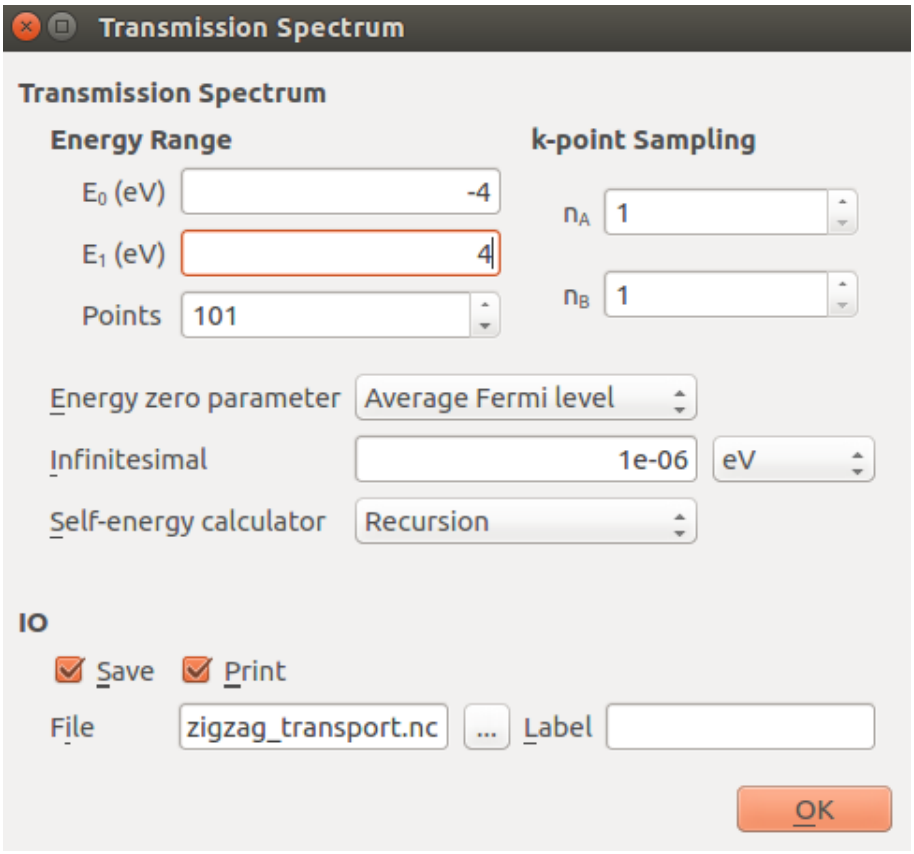
•Select the **Extended Hückel** calculator

•Set the k-point sampling to 1x1x100.

•Change the Hückel basis set to Cerda.Carbon [graphite] for C.

Next, open the the Bandstructure block and set points per segment to 200.

Finally, open the TransmissionSpectrum block and set the energy range from 4 eV to +4 eV.

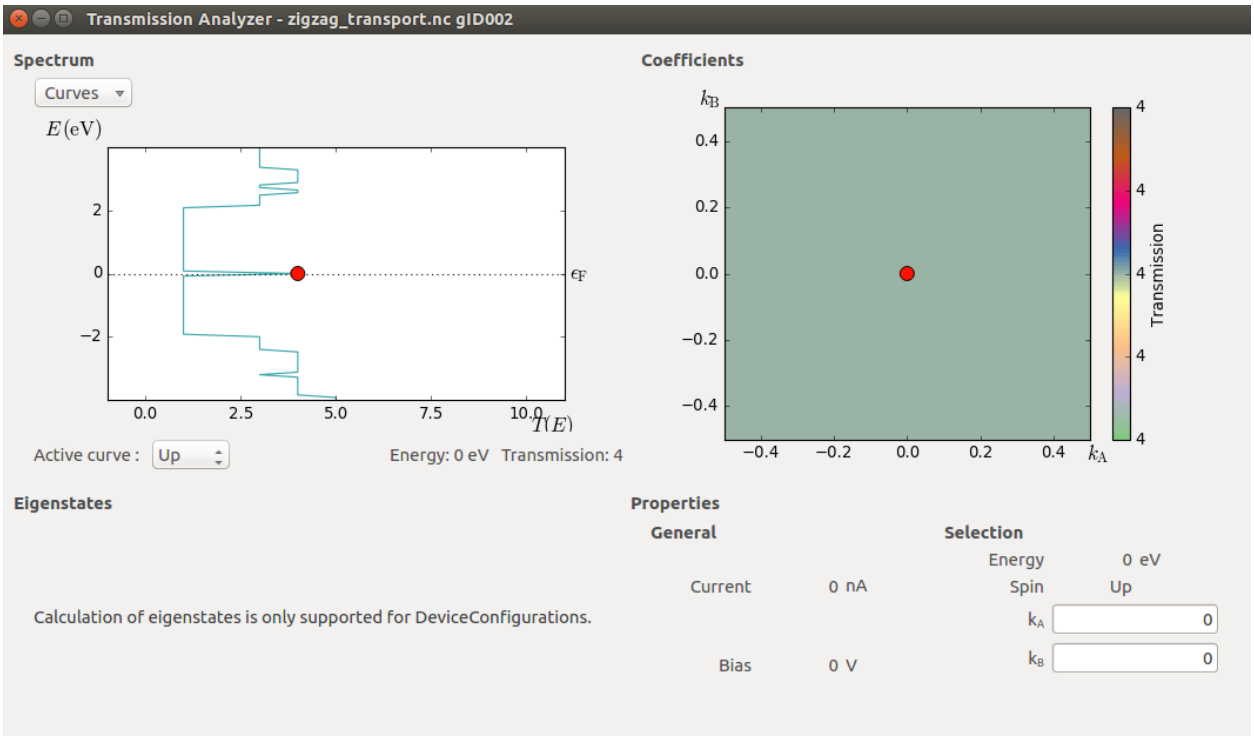


Note: The odd number of energy points ensures that there will be an energy point at the Fermi level.

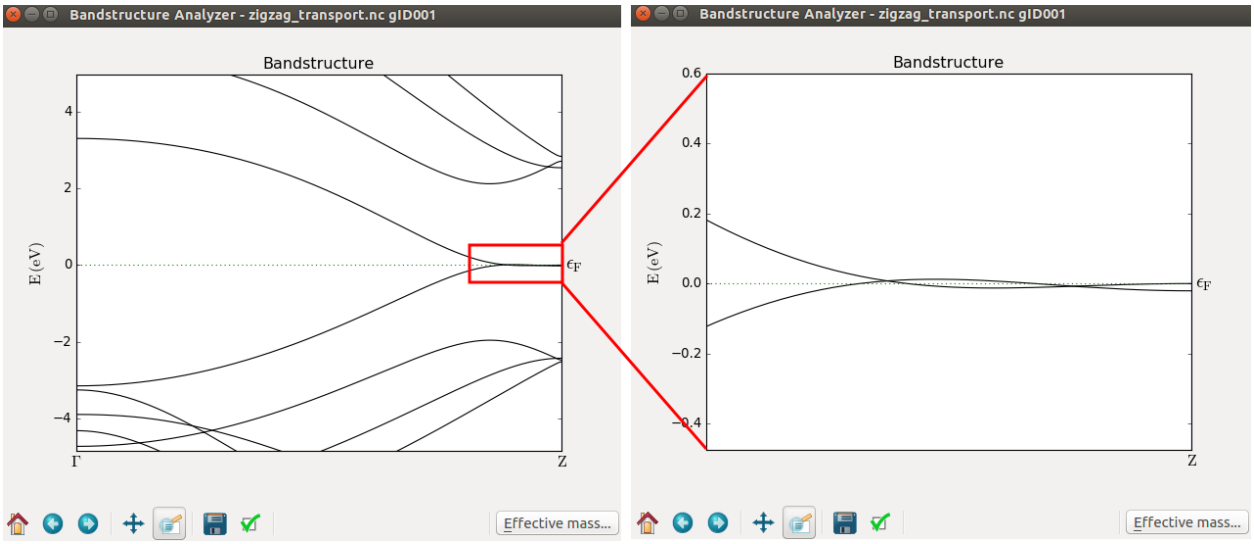
Run the script using the **Job Manager**. It should take only a few minutes to finish.

Analyzing the results

Once the calculation finishes, locate the **TransmissionSpectrum** item on the LabFloor, and open the **Transmission Analyzer** plugin. As expected for a perfect 1D system, the transmission exhibits a sequence of steps with integer transmission.



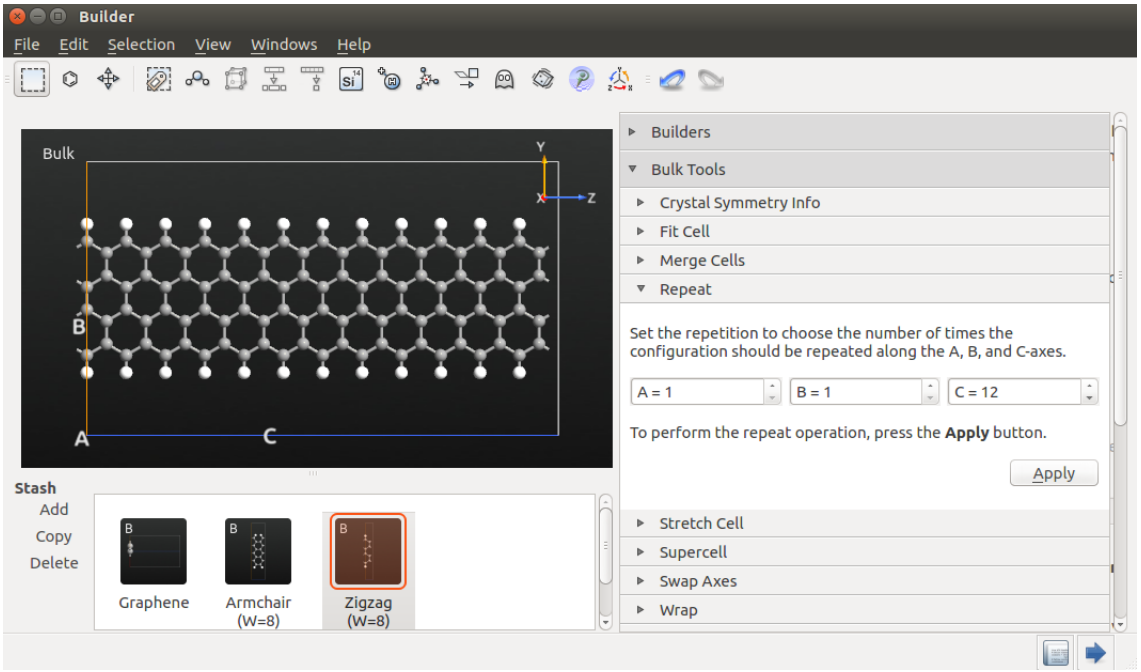
The most prominent feature of the plot is the enhanced transmission around the Fermi level. This is due to a peculiarity in the band structure of the zigzag ribbon (see the band structure below), and interestingly enough, the enhancement is retained even if the ribbon is distorted, at least in the way that will be investigated in the next example.



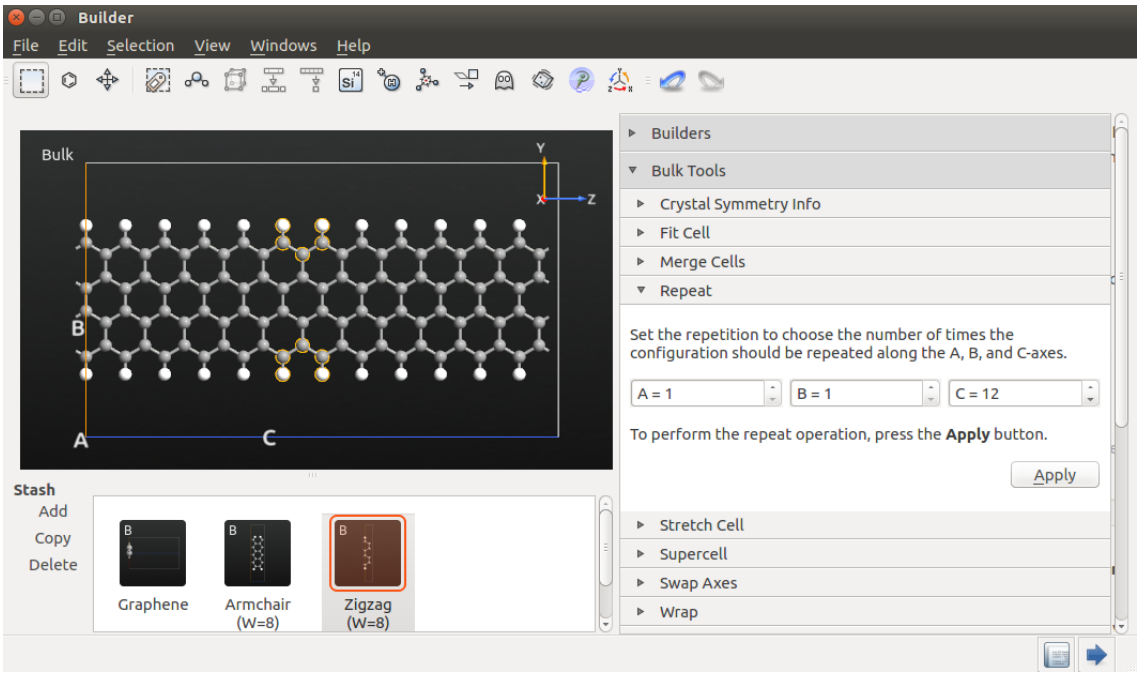
Creating a distortion in the ribbon

So far you have basically just confirmed that armchair ribbons are semiconducting and zigzag ribbons are metallic (at least as long as electron spin is not considered). It is now time to do something a little bit more interesting. The next system you will study is a distorted ribbon; more specifically you will make a constriction in the perfect zigzag ribbon, and see how this influences the transport properties.

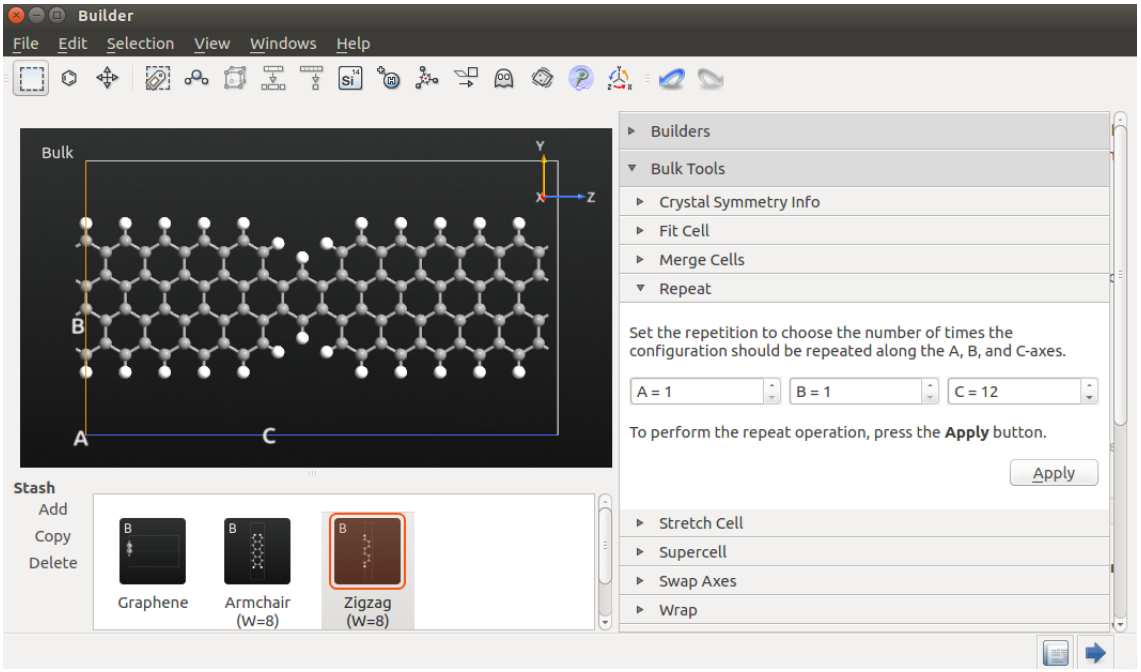
Return to the **Builder** and open the Repeat tool (available from Bulk Tools in the plugin panel). Set C=12 and click **Apply**. Then press `Ctrl`+R to reset the 3D view.



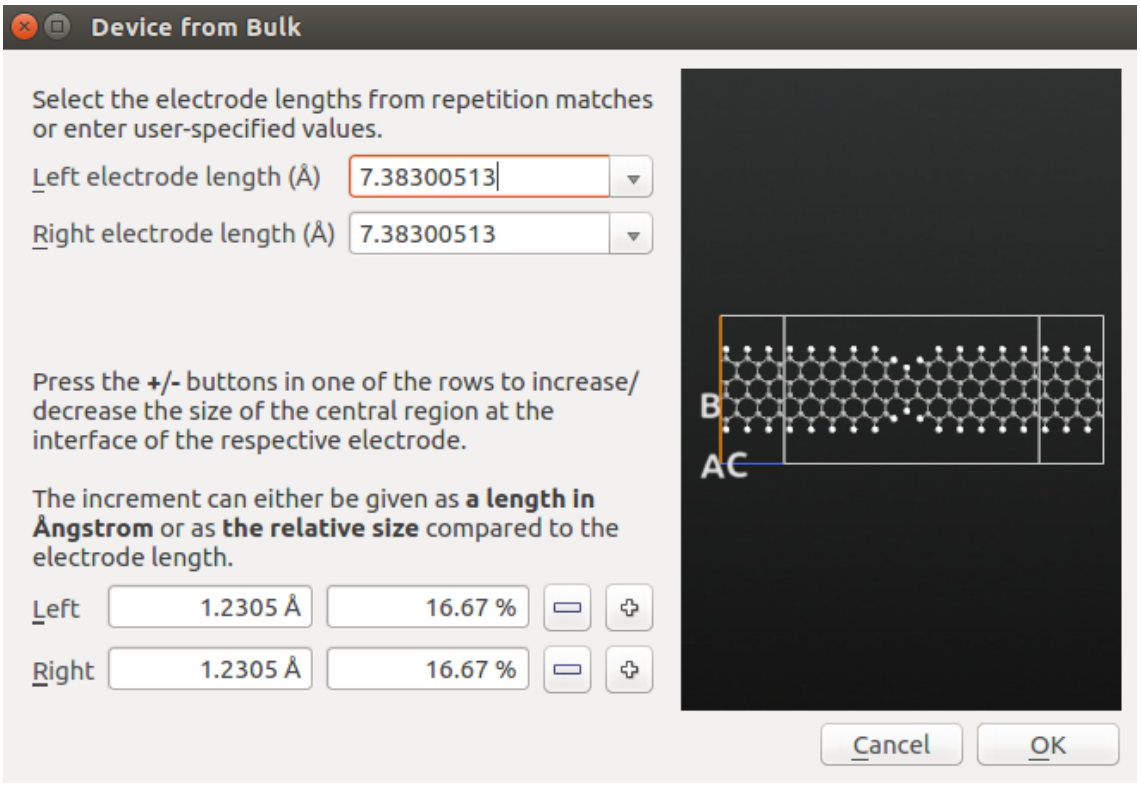
Select the atoms indicated in the figure below by drawing a rectangle with the mouse, while holding down `Ctrl`.



Delete the atoms by pressing the `Delete` key. The dangling bonds that have now been created should be saturated by adding hydrogen atoms. To do this, click the Passivate button on the top toolbar.



Finally, convert the periodic structure to a device geometry by using the *Device Tools* → *Device From Bulk* tool.



The builder will detect any pattern of periodicity close to the edges of the structure, and suggest these as electrodes. You may see these suggestions in the dropdown combo boxes, as indicated in the figure above.

For the current structure, several periodicity patterns can be found, corresponding to a certain number of repetitions of the zigzag unit cell. The suggested value (7.383 Å), corresponds to 3 repetitions, and this is a suitable length. Thus, no further adjustments are needed. Click **OK**, to build the device geometry for the distorted zigzag nanoribbon.

Transport properties of the distorted ribbon

In the following you will learn how to perform the calculation of the zero-bias conductance of the distorted ribbon using open boundary conditions, first non-self consistently and then self-consistently.

Note: Calculations at finite bias must always be performed fully self-consistently, since the non-self-consistent model do not produce a correct voltage drop across the device.

Non-self-consistent calculation

Send the device geometry to the **Script Generator**, and set the default output file name to `zigzag_transport.hdf5` (i.e. the same file name as used for the bulk transmission calculation above, so that the new calculations are saved in that same file).

Then add the **New Calculator** and **TransmissionSpectrum** script blocks.

Set up the Calculator block to use the **Extended Hückel (Device)** calculator with the Cerda.Carbon[graphite] Hückel basis set for carbon. Set also the energy range in the TransmissionSpectrum block to 4 eV to +4 eV.

Run the script using the **Job Manager**. It takes a minute or so, but while it is running you can already proceed and prepare the next step.

Self-consistent calculation

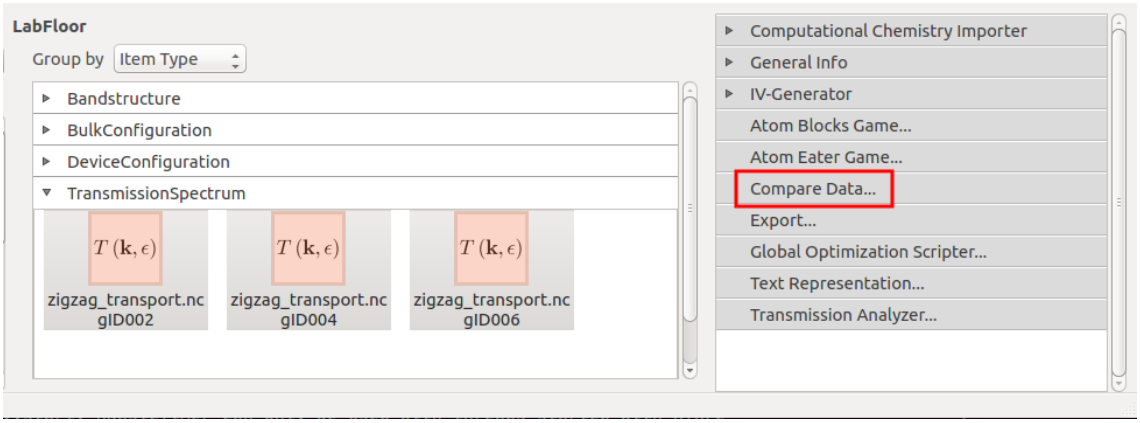
You should also perform a self-consistent calculation with open boundary conditions. This requires only a single change to the Calculator block:

- Remove the tick mark from **No SCF iteration** in the Iteration Control Parameters calculator settings.
- Send the script to the **Job Manager** and run the job.

Run the script using the **Job Manager**. This time the job will take somewhat longer (around 10 minutes).

Comparing the transmission spectra

Let us now compare the computed transmission spectra of the perfect ribbon and the distorted ones. On the **LabFloor**, select **Item Type** in the Group by menu. Then select all three the transmissionspectrum objects and click the **Compare Data** plugin.



Comparing the results you see that the strong transmission at the Fermi level in the perfect ribbon is also present in the distorted ribbon. Apart from this, the transmission spectra for the distorted GRN are strongly suppressed due to scattering by the distortion.

3.5.2 Transmission spectrum of a spin-polarized atomic chain

Version: 2016.3

In this tutorial, you will learn how to build an atomic-chain device, and calculate the spin-polarized transmission spectrum for two cases where the electrodes have parallel and anti-parallel spin-polarizations.

You will use a chain of carbon atoms to make the calculations run fast. Carbon is not naturally magnetic, but when the atom spacing in the chain is large enough – but not too large – the electronic ground state is actually spin-polarized. This system is of course highly artificial, and only chosen for the purpose of illustrating the methodology.

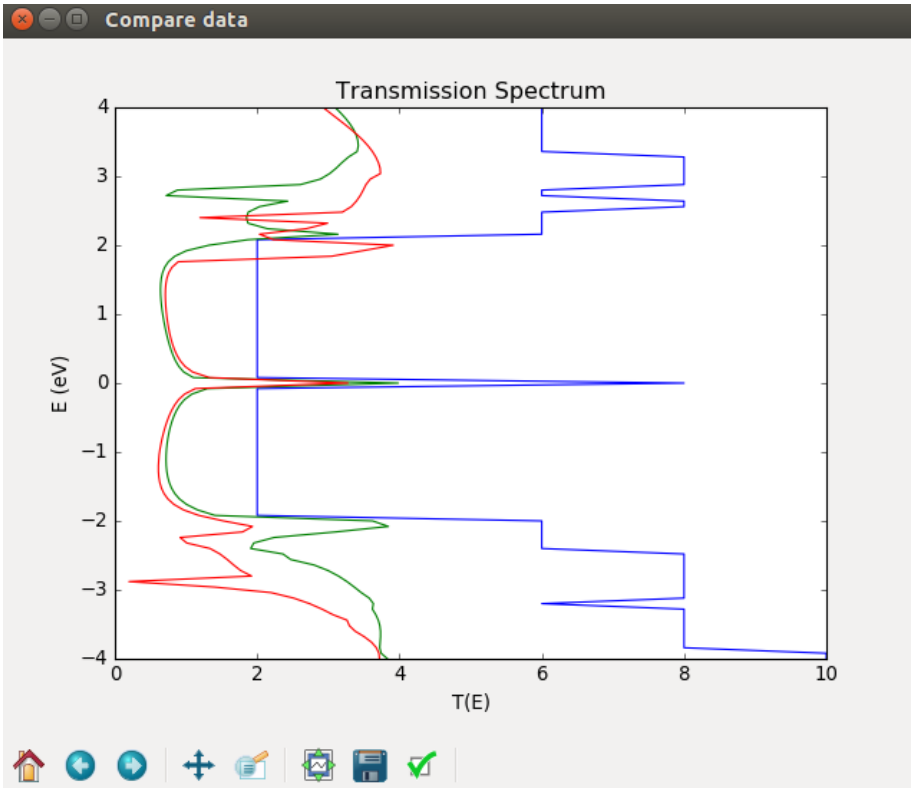


Fig. 3.74: Comparison of the transmission spectra of a perfect ribbon (blue) and a distorted ribbon calculated non-selfconsistently (green) and selfconsistently (red).

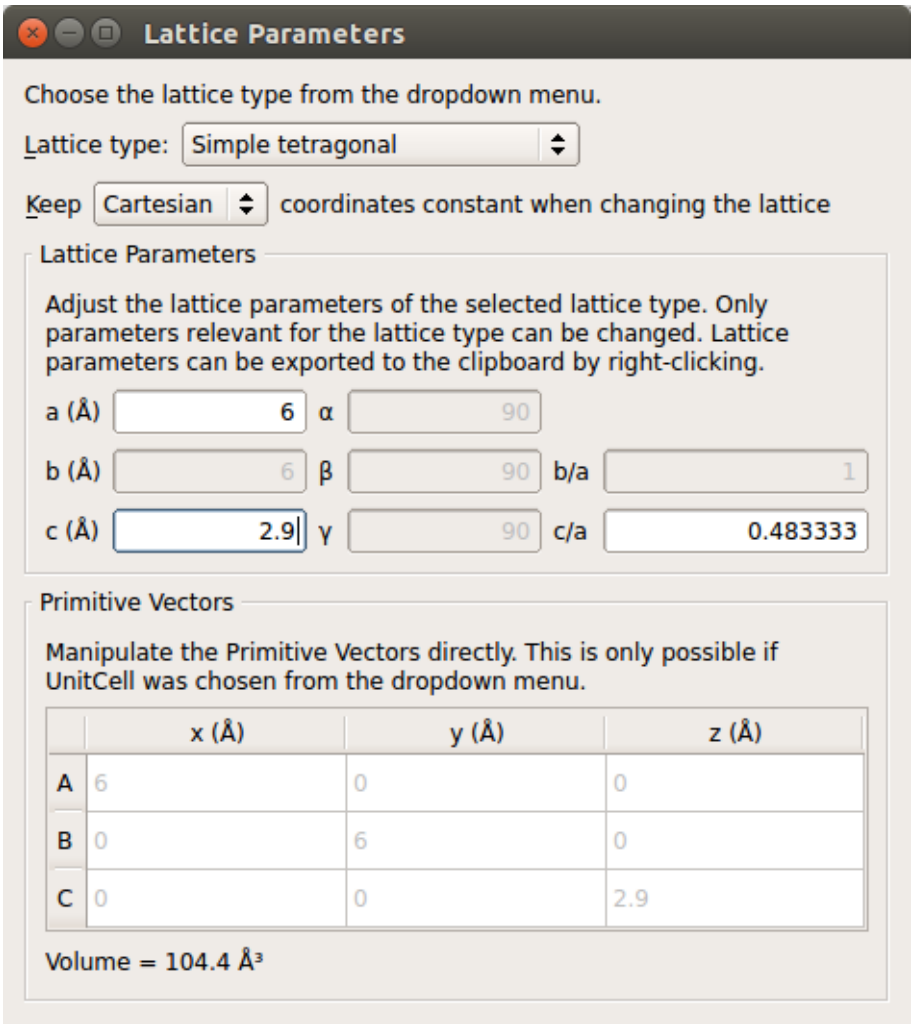
Building the 1D carbon chain

1. Open the QuantumATK **Builder**.

2. In the **Stash**, click *Add* → *New Configuration*. This creates a hydrogen atom.

3. Select the hydrogen atom in the 3D view and use the **Periodic Table** tool to convert the atom to carbon.

4. Open the *Bulk Tools* → *Lattice Parameters* tool and change the lattice type to **Simple tetragonal**. Then set the lattice constants to a = 6 Å and c = 2.9 Å.

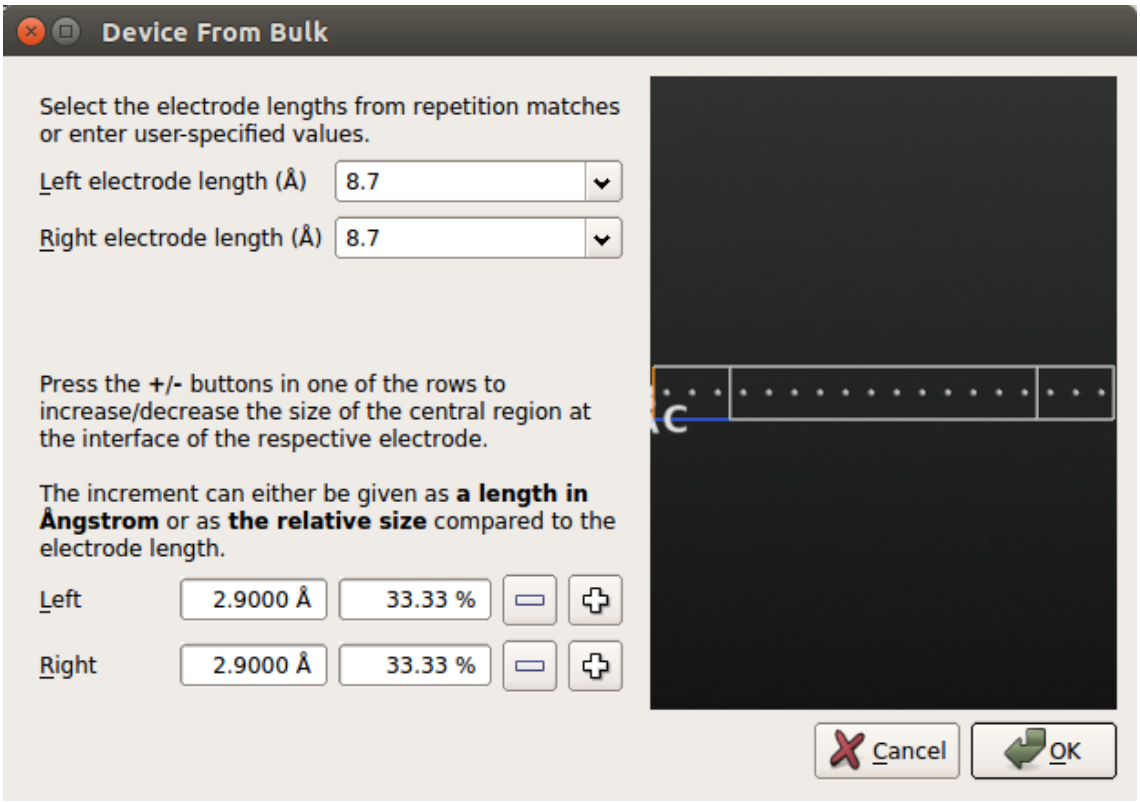


Tip: Periodic boundary conditions will be employed along the A and B directions, so we use the large a = 6 Å unit cell vector to minimize electrostatic interactions between repeated images of the chain along A and B.

5.Click *Coordinate Tools* → *Center* to center the atom in the unit cell.

6.Use the *Bulk Tools* → *Repeat* plugin to repeat the system 12 times in the C direction.

7.Use the *Device Tools* → *Device from Bulk* plugin to create the carbon-chain device. The automatically suggested electrode length of 8.7 Å is adequate, so just click *OK*.



Spin-parallel transmission spectrum

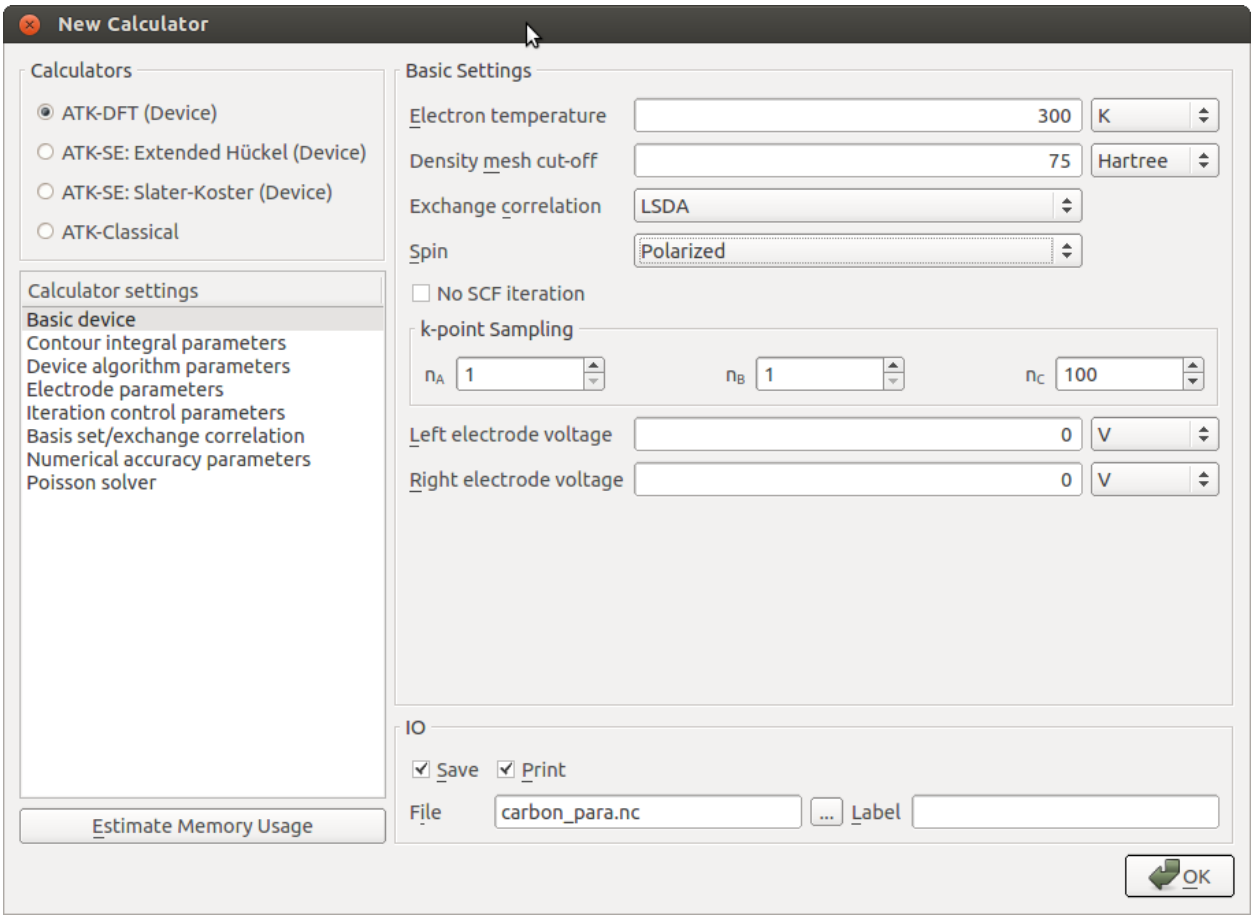
You will now set up a NEGF-DFT calculation with parallel spins throughout the device, and compute the electronic transmission spectrum for this spin configuration.

1.Send the device configuration to the **Script Generator** by using the **Send to** icon in the lower-right corner of the Builder.

2.Set the **Default output file** to `carbon_para.nc`.

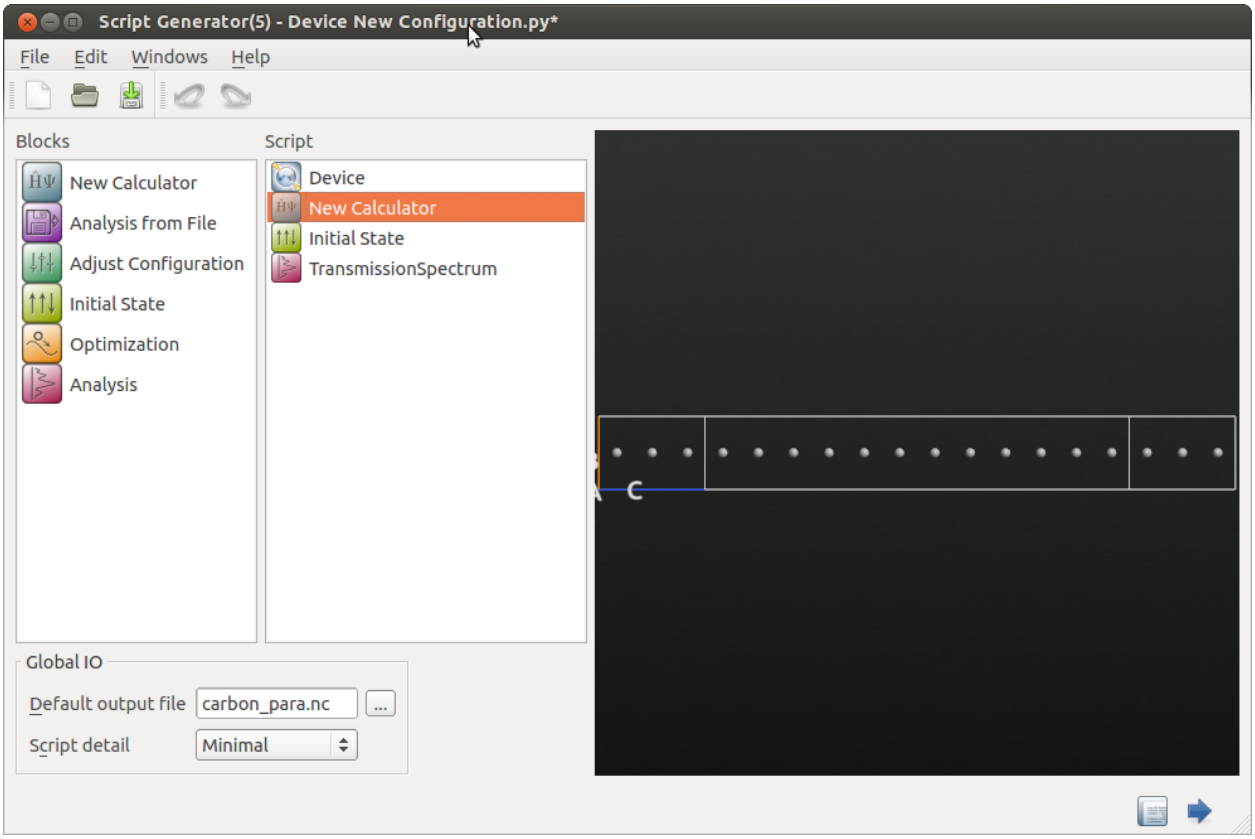
3.Add a New Calculator block to the script, and double-click it to open it.

4.Set the *Spin* to **Polarized**. The exchange–correlation functional will automatically switch to **LSDA**.



5. Add an Initial State block, which defines the initial spin populations. Open it and select **User spin** (all atoms now start out at maximum spin-up polarization).

6. Add a *Analysis* → *TransmissionSpectrum* block. Default parameters are fine for this 1D carbon chain.



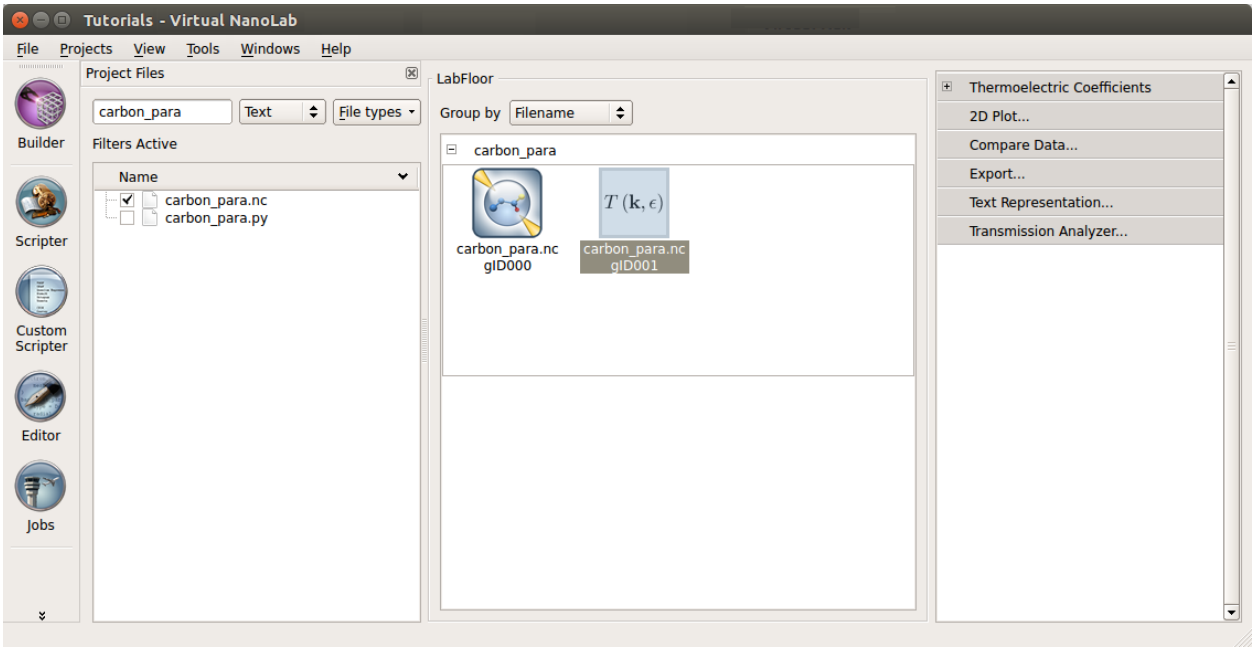
7. The QuantumATK Python script is now ready. Send it to the **Job Manager**, save the script as `carbon_para.py`, and run the calculation – it should only take a few minutes to run.

Note: Do not close the Script Generator window – you will need it later on.

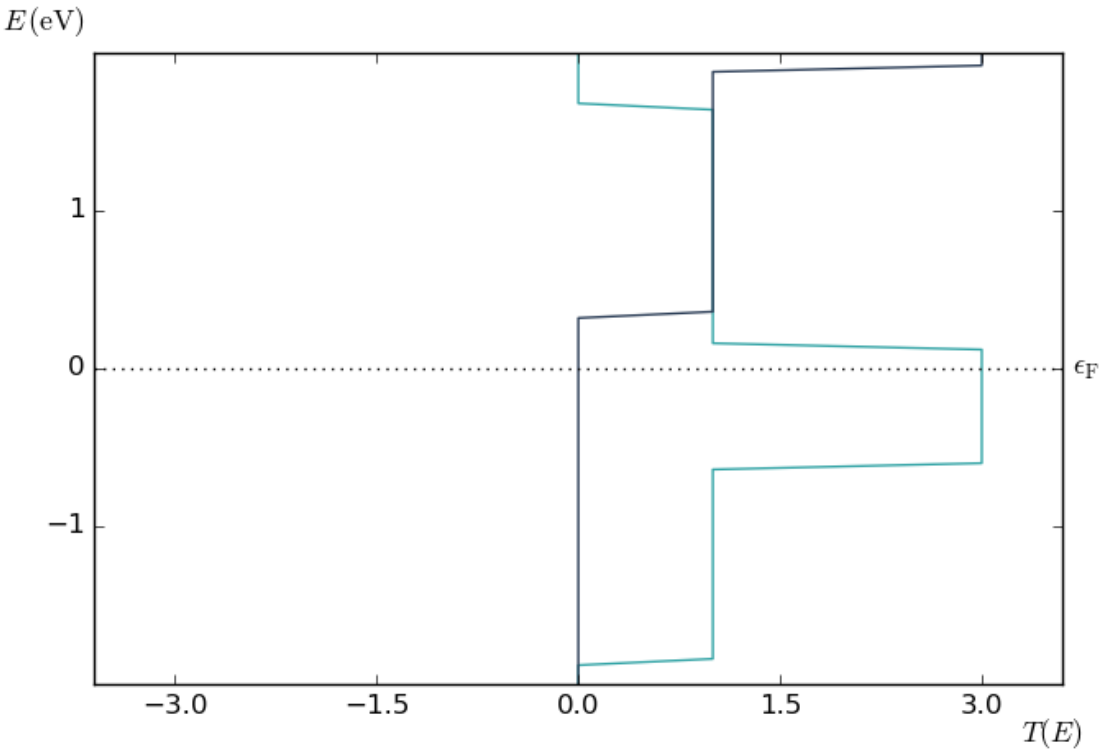
Tip: By inspecting the QuantumATK log file, note that the populations for the up/down (*DM[U]*/*DM[D]*) electronic states are nearly the same on all atoms, as expected in an ideal system. The small differences are primarily due to the fairly low default density mesh cut-off.

Analysis

The calculated **TransmissionSpectrum** analysis object should now be available on the QuantumATK LabFloor.



Select the TransmissionSpectrum and use the **Transmission Analyzer** to plot the spin-parallel transmission spectrum.



Both the up (light blue) and down (dark blue) spin components of the transmission spectrum are plotted, and they are significantly different: The spin-down transmission is zero at the Fermi level, where the (unitless) spin-up transmission is 3.

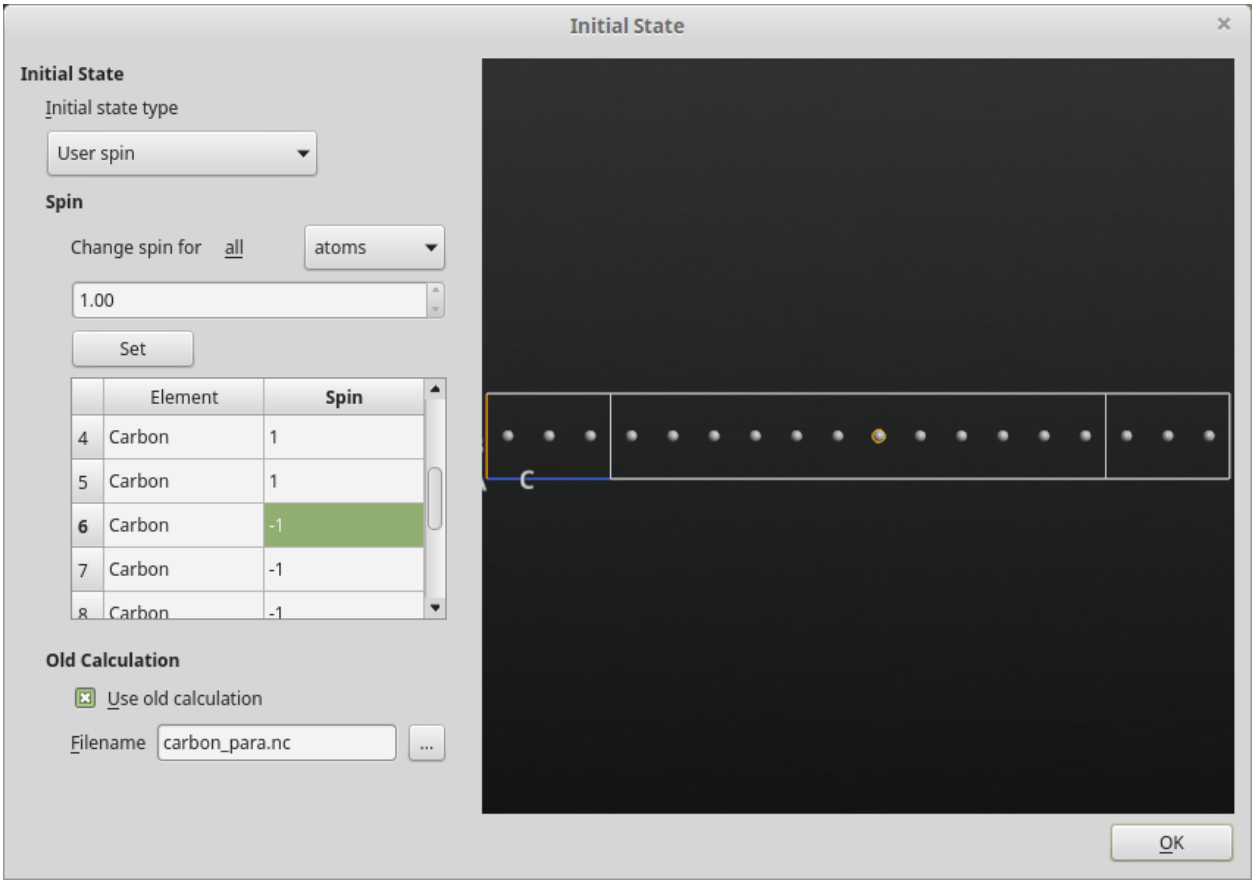
Spin anti-parallel transmission spectrum

You will now do the spin anti-parallel calculation and compare the transmission spectrum to that of the spin-parallel case. The already computed spin-parallel ground state will be used as an initial guess for the anti-parallel calculation.

Return to the **Script Generator** window and modify the settings:

1. Change the **Default output file** to `carbon_anti.nc`.
2. Double-click the InitialState block and modify the following parameters:

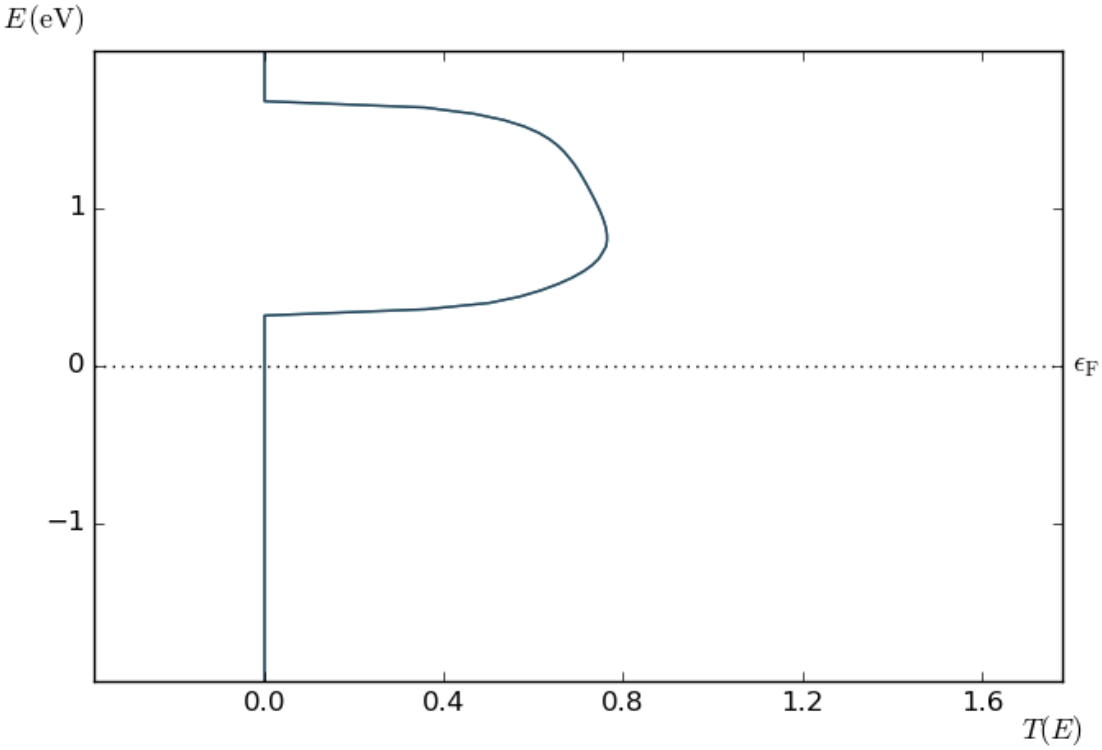
- Set the *initial state type* to **User spin**
- Set the spin on atoms with index 6 to 11 to -1.
- Tick **Use old calculation** and set the *Filename*, from where to read the spin-parallel ground state, to `carbon_para.nc`.



3. Thats it! Save the script as `carbon_anti.py` and run it using the **Job Manager**.

Analysis

When the calculation finishes, use again the **Transmission Analyzer** to plot the transmission spectrum contained in `carbon_anti.nc`. The two spin components have, in this case, identical transmission spectra, as expected from the symmetry of the device.



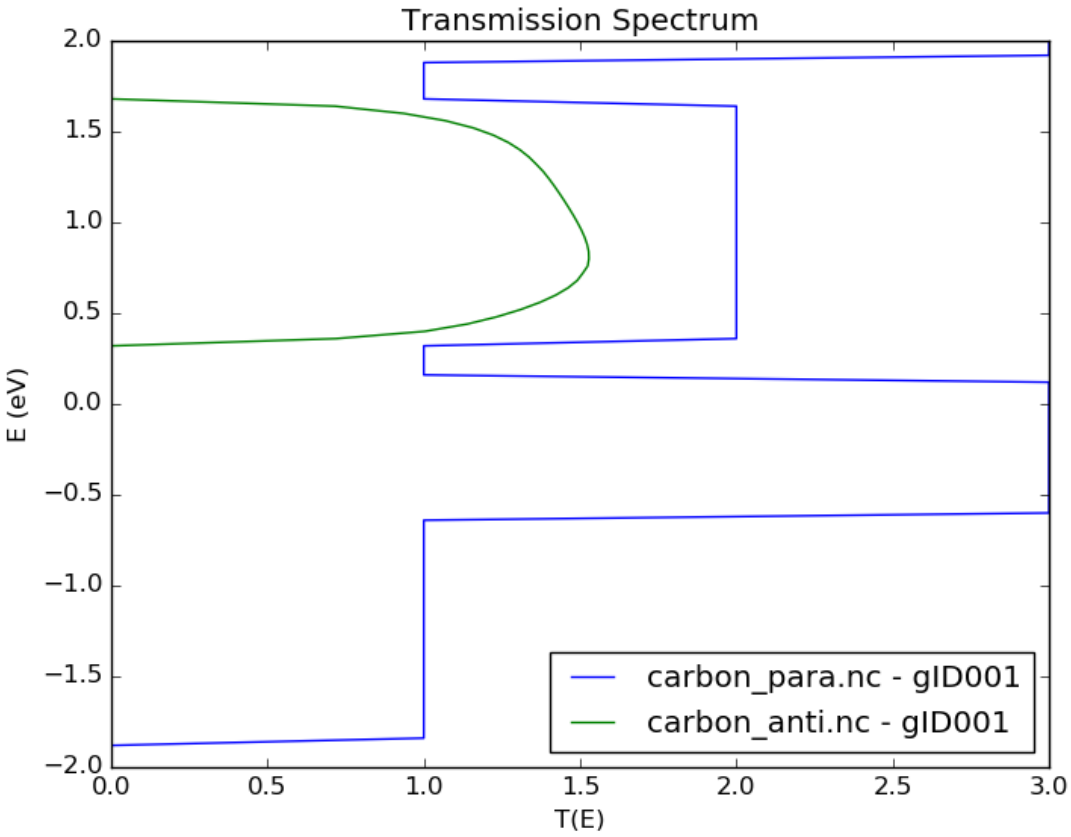
Tip: If you inspect the QuantumATK log file from the anti-parallel calculation, you will see that the spin is indeed flipped in the right part of the device, where the spin-down density matrices, *DM[D]*, have higher occupations than the spin-up ones, *DM[U]*:

```
+-------------------------------------------------------------------------------+
| Density Matrix Report                          DM[U]     DM[D]      DD        |
+-------------------------------------------------------------------------------+
|   0   C   [   3.000 ,   3.000 ,   1.450 ]    2.99639   1.00028  -0.00333     |
|   1   C   [   3.000 ,   3.000 ,   4.350 ]    3.00324   0.99997   0.00321     |
|   2   C   [   3.000 ,   3.000 ,   7.250 ]    2.99536   1.00105  -0.00359     |
|   3   C   [   3.000 ,   3.000 ,  10.150 ]    3.00271   1.00275   0.00546     |
|   4   C   [   3.000 ,   3.000 ,  13.050 ]    2.96897   1.01659  -0.01444     |
|   5   C   [   3.000 ,   3.000 ,  15.950 ]    2.92841   1.08262   0.01103     |
|   6   C   [   3.000 ,   3.000 ,  18.850 ]    1.08262   2.92840   0.01101     |
|   7   C   [   3.000 ,   3.000 ,  21.750 ]    1.01660   2.96897  -0.01444     |
|   8   C   [   3.000 ,   3.000 ,  24.650 ]    1.00275   3.00271   0.00546     |
|   9   C   [   3.000 ,   3.000 ,  27.550 ]    1.00105   2.99535  -0.00359     |
|  10   C   [   3.000 ,   3.000 ,  30.450 ]    0.99997   3.00325   0.00322     |
|  11   C   [   3.000 ,   3.000 ,  33.350 ]    1.00027   2.99629  -0.00344     |
+-------------------------------------------------------------------------------+
```

Regarding the use of the spin-parallel calculation as initial state, QuantumATK will actually flip the density matrix if the

initial spin (which is set in the InitialState block) is less than zero. Also, the spins in both electrodes are automatically set to match the spin configuration in the corresponding electrode extensions.

Finally, mark both the parallel and anti-parallel transmission spectra on the LabFloor, and use the **Compare Data** plugin to directly compare the total transmission in both cases. The transmission in the spin-parallel state (blue) is significantly larger than that of the anti-parallel state (green) over most of the sampled energy window around the Fermi level.

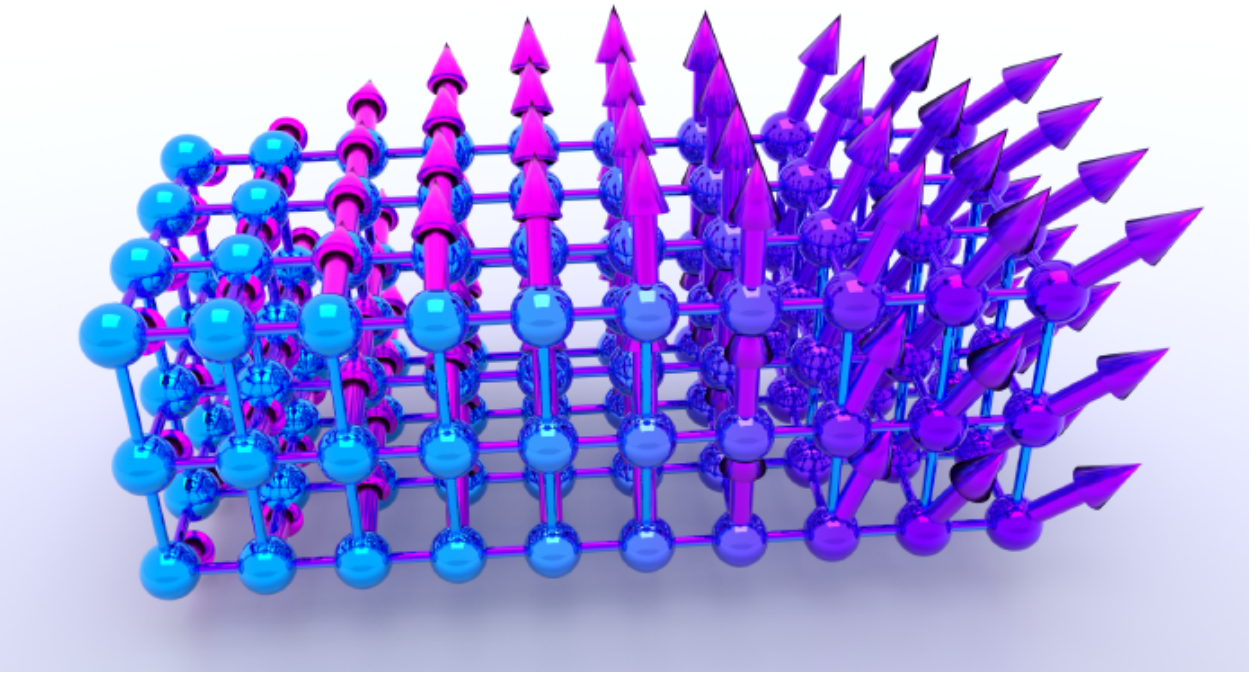


3.5.3 Introduction to noncollinear spin

Version: 2016.3

In standard (collinear) spin-polarized calculations, the spin quantum number (up or down) is added to the electronic states. In contrast, **noncollinear spin** allows the electronic spin to point in any direction. This introduces a few more concepts – and possibilities! – which may be somewhat unfamiliar. This tutorial therefore provides a simple introduction to **ATK-DFT** calculations with noncollinear spin densities.

As briefly stated above, **noncollinear magnetism** refers to situations where the spin direction depends on position in such a way that there is no particular direction in which all the spins are (anti)parallel. Noncollinear spins are quite ubiquitous in nature, and include systems with spin spirals (e.g. chromium) and helicoids, canted spins (e.g. manganites), and most commonly domain walls in ferromagnetic materials. **ATK** allows you to study systems with noncollinear spins from first principles, but it is technically and conceptually quite different from the familiar case of collinear spin.



From collinear to noncollinear spin

It is important to realize that the familiar concept of spin as being either *up* or *down* – and all derived quantities also being labeled by this quantum number – does not work in a noncollinear DFT calculation. Instead, the eigenstate of an atom is a **spinor** with a certain mixing of both spin up and down channels, and many quantities – like the electron transmission spectrum – become a 2x2 matrix rather than two separate numbers (spin up and down transmission).

Another important aspect of noncollinear calculations in practice is that they require in general more CPU time and memory than the corresponding spin-polarized or unpolarized calculation. SCF convergence may also be harder to achieve, since the electronic states have more degrees of freedom.

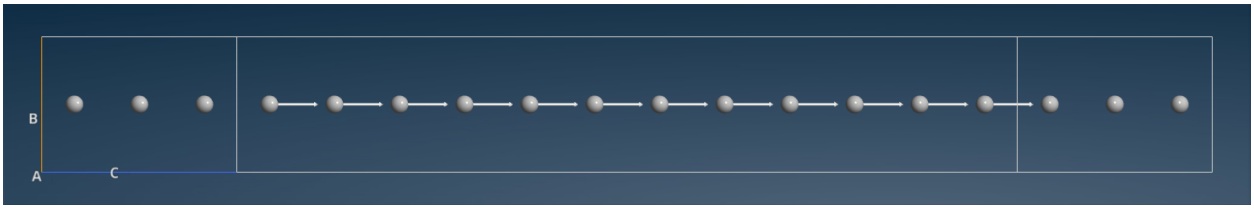
Two key features have therefore been implemented in QuantumATK to improve the SCF convergence rate for non-collinear calculations:

1. use of a collinear spin-polarized calculation as starting point;
2. a special density mixing scheme which diagonalizes the density matrix before mixing it.

Using these techniques, the required number of iterations to reach the selfconsistent noncollinear ground state can be reduced substantially.

Getting started

As mentioned above, the recommended approach for noncollinear calculations is to use a collinear spin-polarized calculation as the initial state. This tutorial therefore takes as starting point the collinear spin-parallel ground state of a simple carbon-chain device obtained in the tutorial *Transmission spectrum of a spin-polarized atomic chain*.



Start by opening **QuantumATK** and create a new project. If you have not already completed the aforementioned tutorial, use a script to calculate the required spin-parallel ground state: `carbon_para.py`. The calculation takes less than 5 minutes and saves the result in the file `carbon_para.nc`.

You will now use the selfconsistent calculation stored in this file as the starting point for a noncollinear calculation of the same linear 1D chain of carbon atoms. However, instead of just considering the parallel and anti-parallel (left electrode *up*, right electrode *down*) spin configurations, you will consider any angle of spin rotation between the two electrodes.

Spin rotation of 120ř

Open the QuantumATK **Editor** (or your own favorite editor) and copy/paste the following lines of *ATK Python* code into it:

```
# Read in the collinear calculation
device_configuration = nlread('carbon_para.nc', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )

# Get the calculator and modify it for noncollinear LDA
calculator = device_configuration.calculator()
calculator = calculator(
    exchange_correlation=NCLDA.PZ,
    iteration_control_parameters=iteration_control_parameters
    )

# Define the spin rotation
theta = 120*Degrees
left_spins   = [(i, 1, 0*Degrees, 0*Degrees) for i in range(3)]
center_spins = [(i+3, 1, theta*i/5, 0*Degrees) for i in range(6)]
right_spins  = [(i+9, 1, theta, 0*Degrees) for i in range(3)]
spin_list    = left_spins + center_spins + right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

# Setup the initial state as a rotated collinear state
device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
    initial_state=device_configuration
    )

# Calculate and save
```

(continues on next page)

(continued from previous page)

```
device_configuration.update()
nlsave('carbon_nc120_ncmix.nc', device_configuration)
```

The spin setup corresponds to the spin polarization of the atoms in the left electrode pointing along the transport axis C, while in the right electrode the polarization is rotated 120 degrees (polar angle in a coordinate system where the XY plane is the equator). In the central region, the angle is interpolated between these two values. Note that this is just the **initial spin configuration** – the actual spin polarization vectors will be computed selfconsistently and may therefore change (you can see the result below).

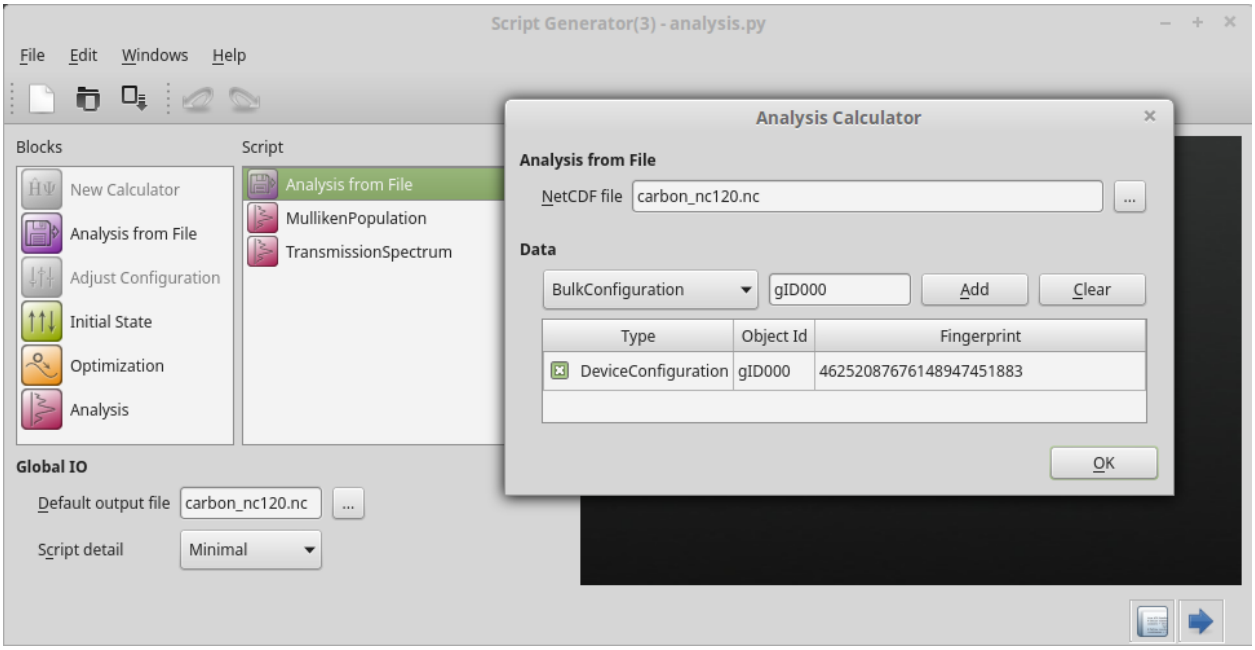
Note: The initial electrode spins are automatically identical to the initial spins on the atoms in the electrode extensions in the central region, in this case the 3 first and last atoms.

Save the script as `carbon_nc120.py`, and then run it – it should take a few minutes only. Remember that you can use the **Job Manager** for this.

Analysis

The NEGF calculation is done, and its time to do some analysis. Use the Script Generator to set up the post-SCF analysis calculations:

1. Open the **Script Generator**.
2. Double-click the **Analysis from File** block to insert it into the Script panel. Then double-click the inserted block and select the file `carbon_nc120.nc`, which was generated in the previous section.
3. Add a **MullikenPopulation** analysis block.
4. Add a **TransmissionSpectrum** analysis block (use default parameters).
5. Set the output file to `carbon_nc120.nc`.
6. Run the script using the **Job Manager**.



The analysis script should look roughly like this:

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Analysis from File
# ------------------------------------------------------------
path = u'carbon_nc120.nc'
configuration = nlread(path, object_id='gID000')[0]

# ------------------------------------------------------------
# Mulliken Population
# ------------------------------------------------------------
mulliken_population = MullikenPopulation(configuration)
nlsave('carbon_nc120.nc', mulliken_population)
nlprint(mulliken_population)

# ------------------------------------------------------------
# Transmission Spectrum
# ------------------------------------------------------------
kpoint_grid = MonkhorstPackGrid()

transmission_spectrum = TransmissionSpectrum(
    configuration=configuration,
    energies=numpy.linspace(-2,2,101)*eV,
    kpoints=kpoint_grid,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )
nlsave('carbon_nc120.nc', transmission_spectrum)
nlprint(transmission_spectrum)
```

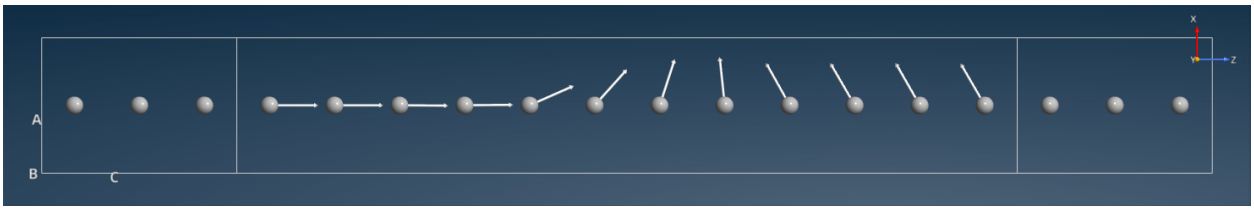
Mulliken populations

The Mulliken populations is now available as an analysis object on the QuantumATK LabFloor, and can be inspected using the **Text Representation** tool in the right-hand panel bar.

Text Representation - carbon_nc120.nc gID001

```
+-------------------------------------------------------------------------------+
|                                                                               |
| Noncollinear Mulliken Population                                              |
|                                                                               |
+-------------------------------------------------------------------------------+
|                                                                               |
| Element     Up        Down       Theta (Degrees)   Phi (Degrees)             |
|                                                                               |
|   0   C     2.9999    1.0000          0.0607           0.0000                 |
|   1   C     3.0000    1.0001          0.0369           0.0000                 |
|   2   C     2.9990    1.0005          0.3963         180.0000                 |
|   3   C     2.9966    1.0034          0.5032           0.0000                 |
|   4   C     2.9917    1.0086         23.6348           0.0000                 |
|   5   C     2.9913    1.0087         47.8371           0.0000                 |
|   6   C     2.9913    1.0087         71.8317           0.0000                 |
|   7   C     2.9925    1.0079         96.0673           0.0000                 |
|   8   C     2.9961    1.0039        119.4304           0.0000                 |
|   9   C     2.9996    0.9999        120.3376           0.0000                 |
|  10   C     3.0000    1.0002        119.9731           0.0000                 |
|  11   C     2.9881    1.0117        120.3336           0.0000                 |
|                                                                               |
+-------------------------------------------------------------------------------+
|                                                                               |
| Mulliken Population Report in Local Atomic Spin-Directions                    |
|                                                                               |
| ----------------------------------------------------------------------------- |
```

You immediately notice the difference to the collinear case: Now the Mulliken population on each atom is described by 4 variables; *Up*, *Down*, *Theta* and *Phi*. Note that *Phi=180* is equivalent to *Phi=0*. The sum of the up and down populations corresponds, as usual, to the total Mulliken charge (the number of electrons), and their difference – combined with the two angles – forms a spin polarization vector, which can be visualized in the QuantumATK **Viewer**:

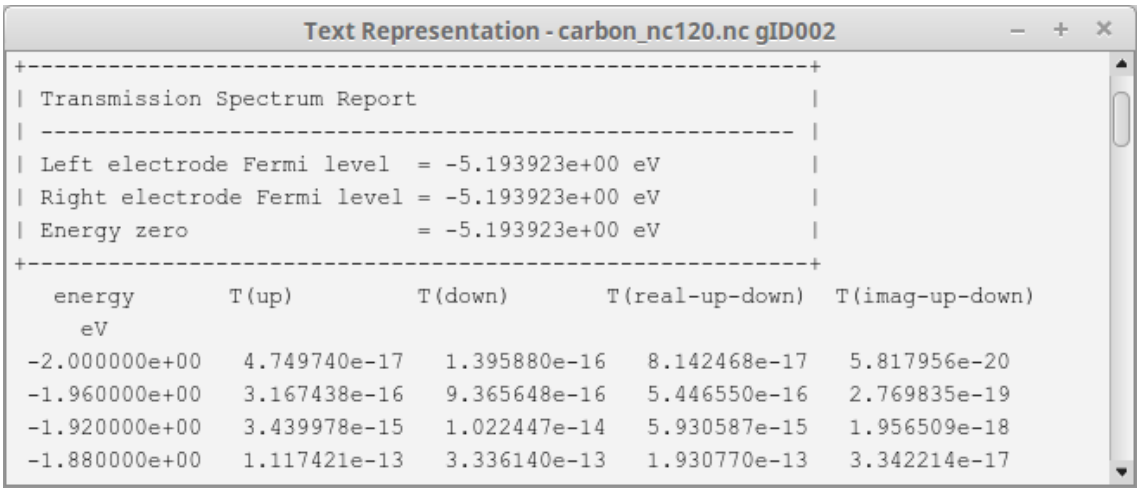


It is clear that the spin polarization direction changes smoothly between the two values in the electrodes.

Tip: The Viewer **view plane** used in the image is ZX. Use the *Camera* settings menu to select a view plane different from the default ZY.

Transmission spectrum

The *TransmissionSpectrum* analysis object is also available on the LabFloor. Use the **Text Representation** tool to inspect it – you again see that it has 4 components; *up*, *down*, *real-up-down* and *imag-up-down*:



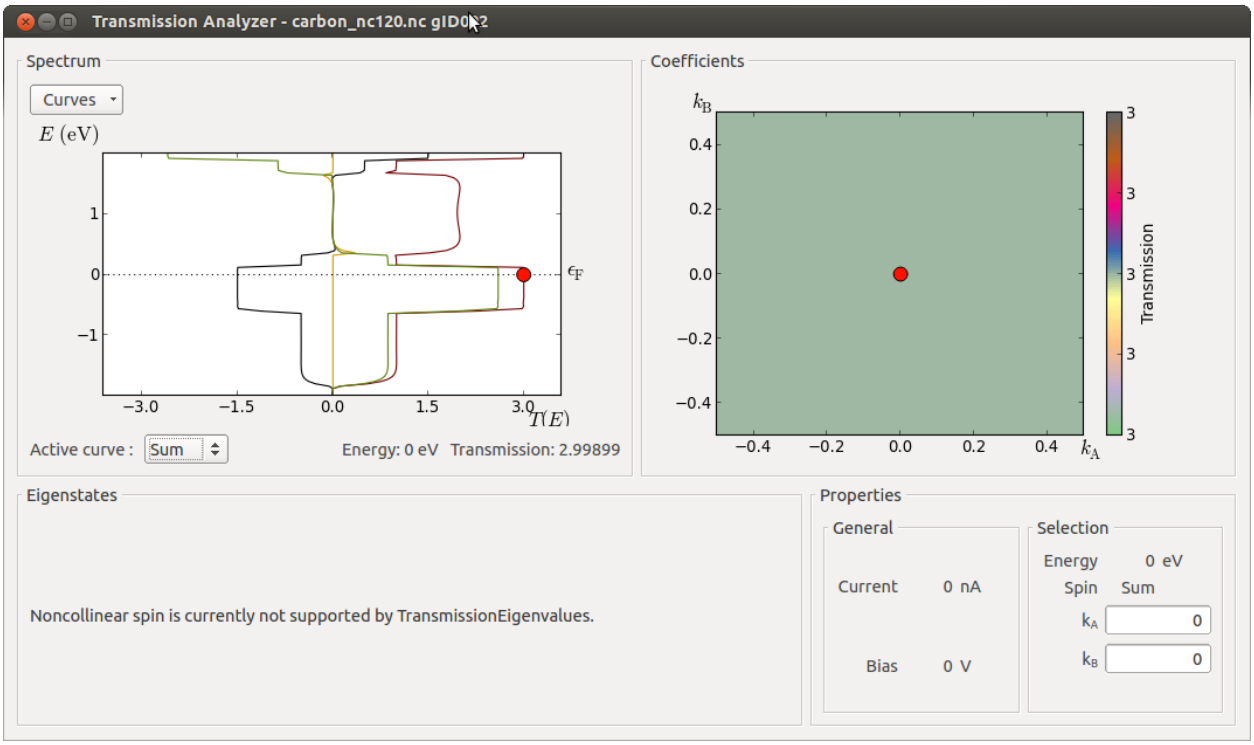
The last spin component (*imag-up-down*) is very small in this simple system, but will in general be important in cases where the spins have other directions.

Note: Any quantitty calculated using noncollinear spin is represented as a 2x2 matrix (a spinor) with different mixing of *up* and *down* components: $\uparrow\uparrow$, $\downarrow\downarrow$, $\uparrow\downarrow$, and $\downarrow\uparrow$.

As explained in the QuantumATK Manual section *Spin*, a range of different **spin projections** may be derived from these 4 basic components, including *Sum*, *X*, *Y*, and *Z*:

$$
\begin{aligned}
T(\text{Spin.Sum}) &= T(\uparrow\uparrow) + T(\downarrow\downarrow) \\
T(\text{Spin.X}) &= 2 \cdot \text{Re}(T(\uparrow\downarrow)) \\
T(\text{Spin.Y}) &= 2 \cdot \text{Im}(T(\uparrow\downarrow)) \\
T(\text{Spin.Z}) &= T(\uparrow\uparrow) - T(\downarrow\downarrow)
\end{aligned}
$$

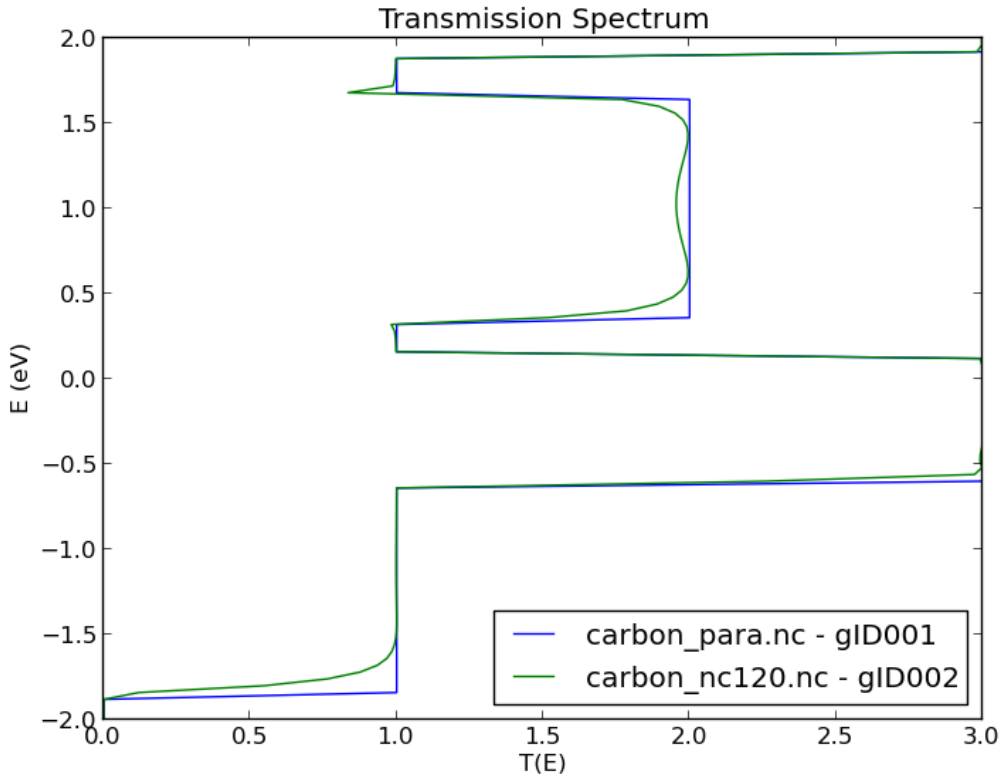
Use the **Transmission Analyzer** plugin to plot the transmission spectrum. The transmission components *Sum*, *X*, *Y*, and *Z* are by default included in the graph:



Tip: The drop-down menu *Curves* at the top-left of the window lets you choose which spin projection to include, while the lower Active curve option is for choosing which k-point resolved spectrum to show in the right-hand Coefficients plot (for this 1D system there is only one transmission coefficient, at (k_A, k_B)=(0,0), so the plot is rather uninteresting). This can also be chosen by clicking the corresponding transmission curve.

The *Spin.Sum* transmission spectrum is very similar to the parallel collinear case, and very different from the anti-parallel collinear case. This may surprise, since the spin configuration in the electrodes are more anti-parallel than parallel. However, due to the noncollinear degrees of freedom, the electron can propagate in a **helical state** when moving from left to right, and this allows for a high transmission. In fact, if you set the rotation angle to 180 degrees, corresponding to anti-parallel alignment of the electrodes, the result will be almost the same.

The figure below compares the transmission spectra for the collinear spin-parallel state and the noncollinear state, using the **Compare Data** plugin.



Spin-orbit interactions

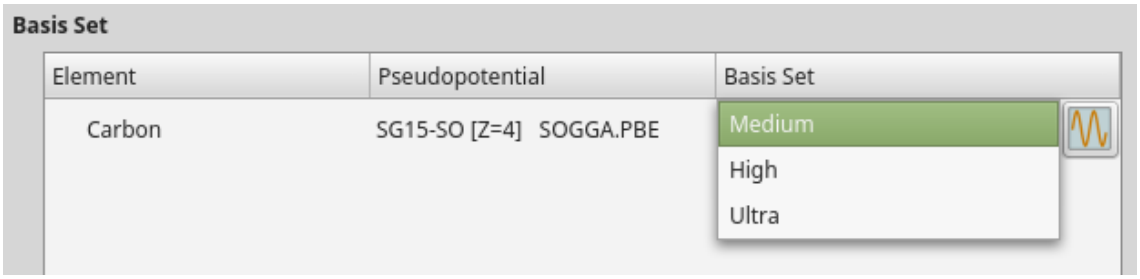
Spin-orbit coupling (SOC) is most often neglected in electronic structure calculations, but it can actually be included in a noncollinear calculation, provided that suitable pseudopotentials are used. You can find more details in the tutorial *Spin-orbit splitting of semiconductor band structures*.

The carbon chain considered here has a very small SOC, so results with spin-orbit interactions included in the electronic structure method will hardly be different from those obtained above. Even so, if you wish to include SOC in calculations similar to the ones outlined in the section *Spin rotation of 120ř*, simply change the exchange–correlation method and pseudopotentials used in the ATK-DFT calculator:

- use spin-orbit GGA (SOGGA) exchange–correlation and SG15-SO pseudopotentials,
- or use spin-orbit LDA (SOLDA) exchange–correlation and OMX pseudopotentials.

Spin-orbit GGA

When setting up the initial collinear calculation, saved as `carbon_para.nc`, choose SGGA exchange-corerlation instead of LSDA, and navigate to the *Basis set/exchange correlation* calculator settings and select the **SG15-SO** type pseudopotential. You will usually have the option to choose between three different basis set sizes:



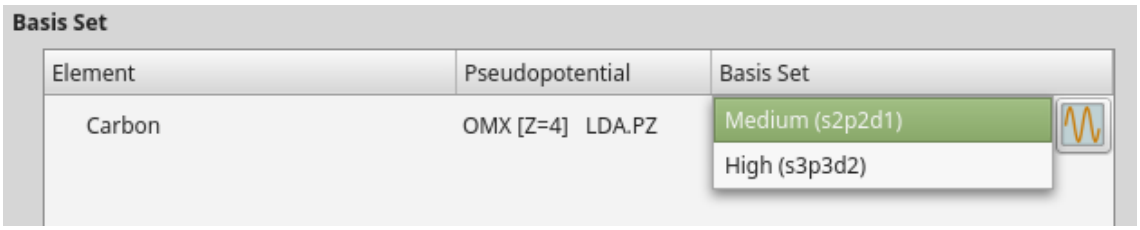
Note: The density mesh cutoff should not be smaller than 100 Ha when using SG15 pseudopotentials.

Run the spin-polarized GGA calculation, which creates `carbon_para.nc`, and then use `SOGGA.PBE` exchange-correlation when setting up the calculator with spin-orbit coupling:

```
# Read in the collinear calculation
device_configuration = nlread('carbon_para.nc', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )

# Get the calculator and modify it for spin-orbit GGA:
calculator = device_configuration.calculator()
calculator = calculator(
    exchange_correlation=SOGGA.PBE,
    iteration_control_parameters=iteration_control_parameters
    )
```

Spin-orbit LDA

When setting up the initial collinear calculation, saved as `carbon_para.nc`, navigate to the *Basis set/exchange correlation* calculator settings and select the **OMX** type pseudopotential. You will usually have the option to choose between different basis set sizes:



Note: The OMX potentials are in general fairly hard, so they often require a larger mesh cutoff than e.g. the FHI potentials, usually at least 150 Ha.

Run the spin-polarized LDA calculation, which creates `carbon_para.nc`, and then use `SOLDA.PZ` exchange-correlation when setting up the calculator with spin-orbit coupling:

```
# Read in the collinear calculation
device_configuration = nlread('carbon_para.nc', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )

# Get the calculator and modify it for spin-orbit GGA:
calculator = device_configuration.calculator()
calculator = calculator(
    exchange_correlation=SOLDA.PZ,
    iteration_control_parameters=iteration_control_parameters
    )
```

Tip: Whats next? You should consider the tutorial *Noncollinear calculations for metallic nanowires*.

Noncollinear spin is essential when computing the spin transfer torque, see for example the tutorials *Spin Transfer Torque* and *Spin transport in magnetic tunnel junctions*.

The tutorials *Spin-orbit splitting of semiconductor band structures* and *Relativistic effects in bulk gold* give more details of spin-orbit calculations with ATK-DFT.

3.5.4 Carbon Nanotube Junctions

QuantumATK: CNT Junctions Platforms: Links: *3D View*, *Exploring Graphene* *An important feature of the NEGF transport model implemented in QuantumATK is its ability to treat heterogeneous systems, meaning device geometries where the two electrodes are not the same. This makes it possible to study a wide range of important systems, such as the contact resistance between two materials (for instance a semiconductor/metal interface) or the capacitance of junctions between nanotubes of different chirality. In this tutorial you will learn how to set up the geometry of such a junction.*

NEGF

You may find a short introduction to the **Non-Equilibrium Greens Function** method here: arXiv.

The steps involved in the Builder are rather simple, but for this type of geometry it is important to study how the transmission and other properties change depending on how much the two tubes overlap. Therefore, you will also learn how to vary this distance, by adding a few lines of Python code to the script.

Note:

•CNTs: Types and Properties

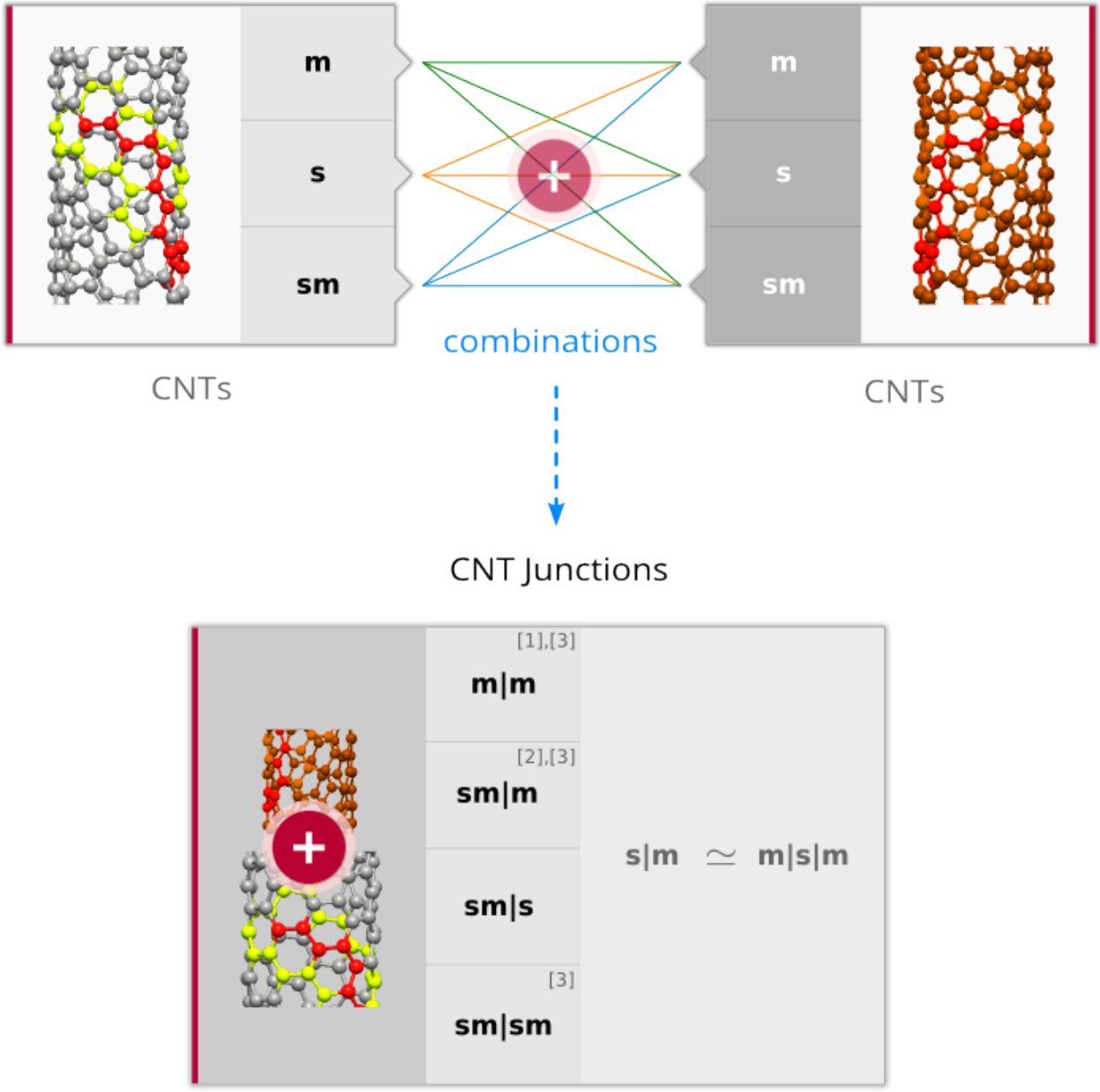
The atomic structure of a CNT is described by a pair of integers called chiral indices $(n, m) \in \mathbf{Z}^2$. If $m = 0$ then the CNT has a zigzag structure. Other special values of these indices describe CNTs with special properties as follows:

1.If $n = m$ then the CNT is **metallic (m)** (armchair structure).

2.If $n \neq m$ then the CNT is **semiconducting (s)** (chiral structure).

3.If $n - m \in 3\mathbf{Z}^*$ then the CNT is **semimetallic (sm)** (chiral structure).

•Possible Combinations for CNT Junctions



Note 1: Here we denote the junction with a vertical bar: |. **Note 2:** Junctions between metallic and semiconducting would be interesting too, although probably in that case the relevant geometry is m|s|m, rather than m|s.

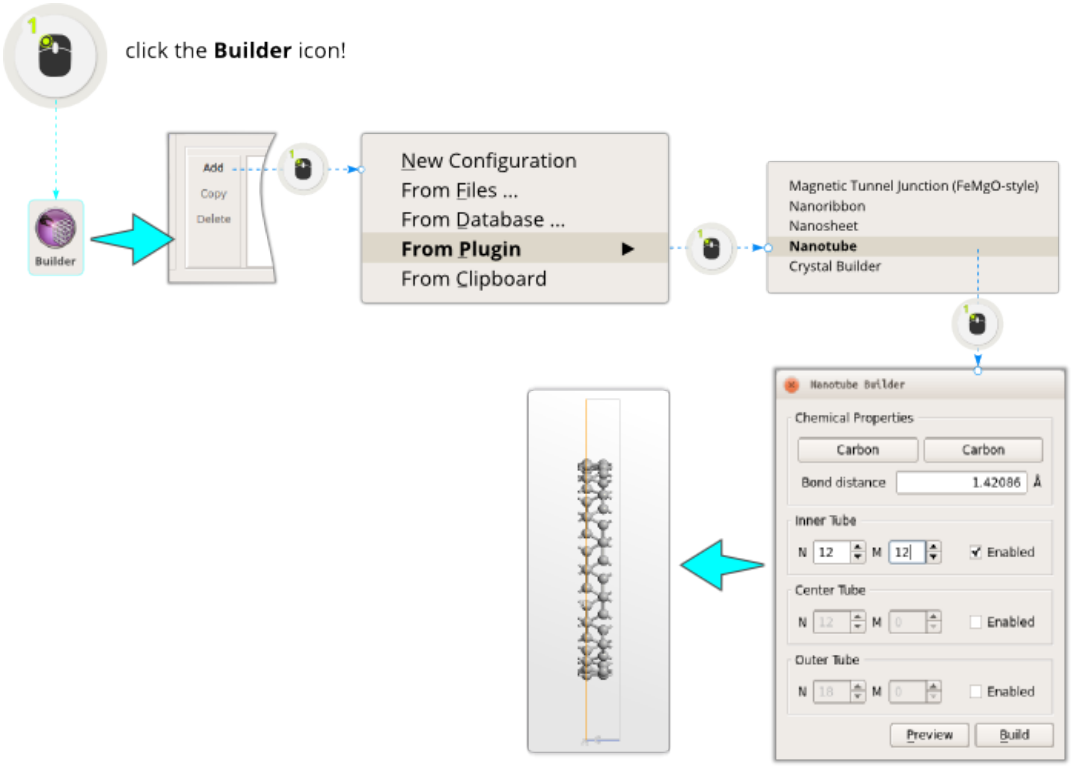
Setting up the geometry

First, you need to set up the reference geometry where the two tubes are just in contact with each other. It is reasonable to define this as the case where the atoms on the edges of the two tubes have the same Z coordinate.

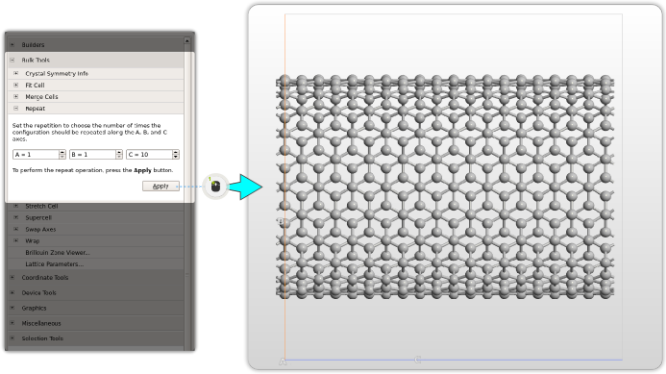
CNT Junction Example: m|m with (12,12)|(5,5)

The instructions below will be using the m|m case of a (12,12)|(5,5) junction, but the steps involved are identical for any combination of chiralities.

•**Step 1:** Create a (12,12) metallic CNT using the **Nanotube** plugin following the steps bellow.



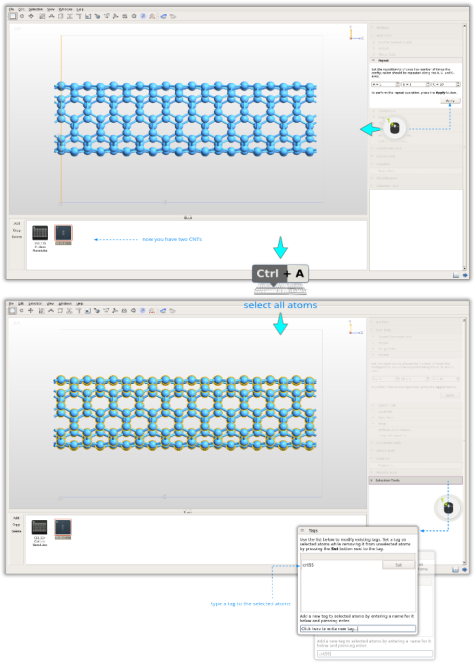
•**Step 2:** The CNT is quite thin and its better to increase its length. Lets repeat the structure 10 times along the C (tube) axis using the *Boolk Tools* → *Repeat* tool and choosing C=10:



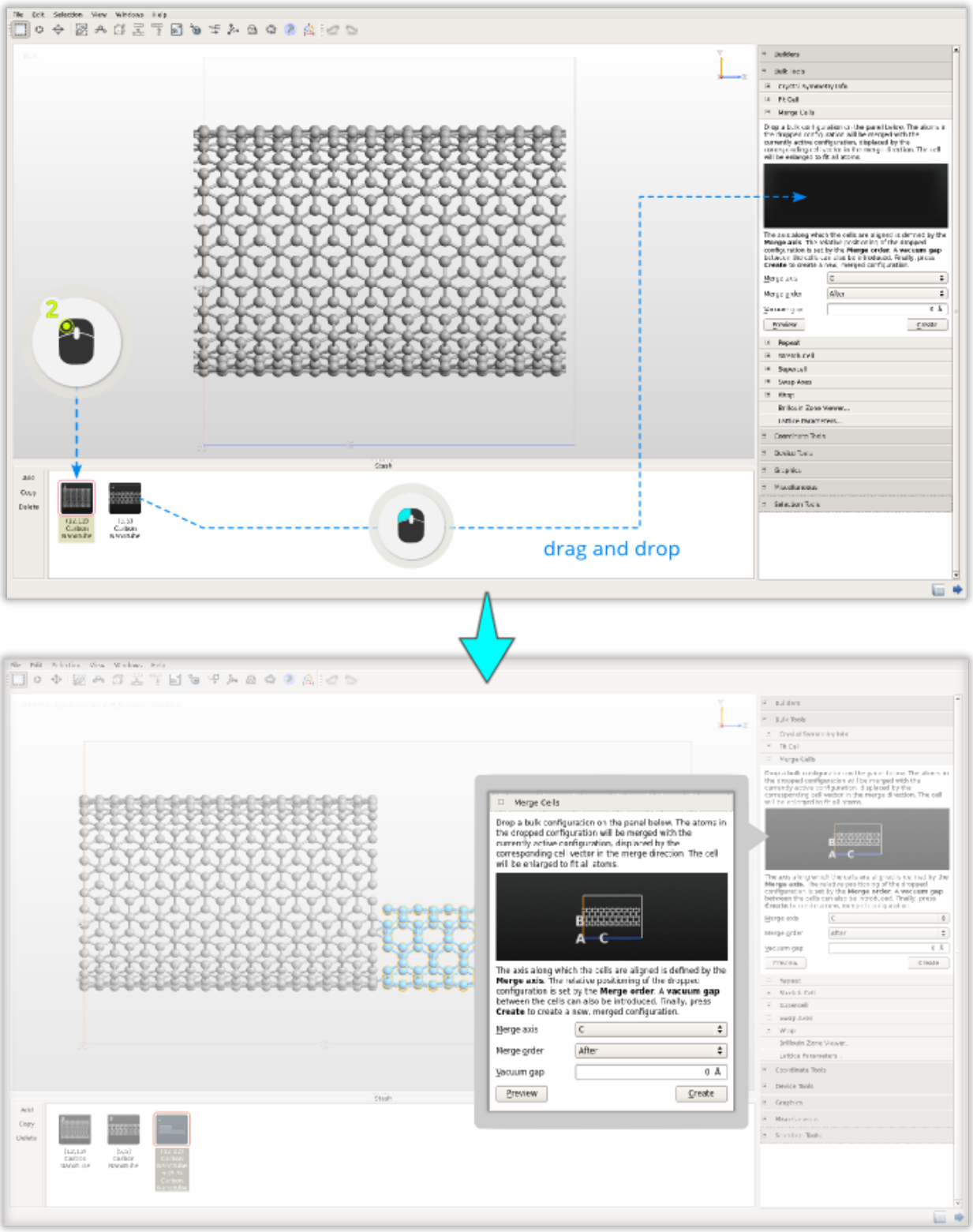
•**Step 3:** In order to create a second CNT with (n,m)=(5,5) just repeat the previous steps and type N=5 and M=5 in the Nanotube plugin. Also we choose to have the same length as the first CNT and thus we repeat the structure ten times.

In the end you will have two CNTs of equal length (one with (12,12) and another one with (5,5) chiral indices) on the Builder **Stash**.

Tip: **Tagging:** In order to later on be able to identify the two tubes, add a tag to all atoms in the (5,5) tube by first selecting all atoms (Ctrl+A) and then open *Select* → *Tags*. Add the tag **cnt55** to all atoms in the (5,5) tube.



- **Step 4:** Double click to the (12,12) structure on the Stash in order to activate it.
- **Step 5:** There is a specific tool in the Builder for *gluing* two bulk cells together: *Bulk Tools* → *Merge Cells*. This plugin takes a bulk configuration and attaches it to the active configuration, on a specified side of the existing unit cell. Drag the (5,5) tube onto the 3D window in the plugin panel. The default options are correct - the (5,5) tube should be added after the (12,12) tube in the C direction. Click **Create**.



•**Step 6:** The (5,5) tube needs to be centered in the z=0 plane (xy-plane), since its original unit cell was smaller than that of the (12,12) tube. Therefore, select all atoms in the (5,5) tube by opening the **Tag Select** tool , select the tag **cnt55**, then OK. Then open *Coordinate Tools* → *Center* and center the tube in the A,B direction - that is, untick C, and

click **Apply**.

•**Step 7:** The system is no longer periodic in any direction, so remove the unit cell by clicking the **Molecule button** .

An interesting quantity to study for a junction like this is how the conductance changes as a function of the amount of overlap - or distance - between the tubes. To have a clear definition of the distance d between the tubes, its important to establish a reference point which we define as d=0. The most convenient definition is the case when the edges of the tubes overlap, and the next steps will align the tubes into this configuration.

•**Step 8:** Activate the **Move tool** and select an atom on the right-most edge of the (12,12) tube (largest Z coordinate).

This selects all atoms in the (12,12) tube - or, more specifically, all atoms connected by bonds to the clicked atom - and the clicked atom becomes an anchor atom, as indicated by the label 0 and the red selection marker.

Set the Z coordinate of this atom to 0 Angstroms. Press Enter. Exit the Move tool by closing the Move widget, and deselect all atoms by clicking the black background (or by pressing Esc). Repeat these steps for the (5,5) tube, but this time select an atom on the left-most edge (smallest Z coordinate).

•**Step 9:** In order to create a device we need a unit cell, so add back the unit cell (it was primarily removed to make it easier to align the two tubes) by clicking the **Bulk icon** . Finally, open *Device Tools* → *Device From Bulk* and click OK.

Changing the distance between the tubes: Manual way

You now have the geometry for the d=0 case, i.e. the geometry where the distance between the tubes is defined as zero. To change the overlap distance d between the tubes, you could now in principle carry out the following steps:

•Make a copy of the geometry on the Stash.

•Convert back to a molecular geometry ( ).

•Select all atoms in the (5,5) tube ( ).

•Open *Coordinate Tools* → *Translate* and move the (5,5) tube the desired distance d.

•Convert back to a bulk configuration ( ) and convert to a device.

Repeat this procedure for all values of d you want to study Now, that becomes rather tiresome, if you want to scan several distances. So instead, you can make a small Python function that performs these tasks automatically. This will be described in the next section.

Using a Python function for changing the overlap distance

Save a copy of the d=0 geometry as a Python script, for safety. Then, send the structure from the Builder to the Editor, by using the **Send To** icon ( ) in the lower right-hand corner of the Builder. Scroll to the bottom of the script, and insert the following lines of code at the bottom of the script:

```
# ------------------------------------------------------------------------------
def moveTube(device_configuration, d=0.0):
    # d is assumed to be in Angstrom
    # Extract the central region from the device configuration
    central_region = device_configuration.centralRegion()

    # Number of atoms
    N = len(central_region.cartesianCoordinates())
    # Which atoms are in the right tube? Those with tag="2"
    right_tube = device_configuration.indicesFromTags("2")

    # Set a Z shift of d for these atoms, and zero for others
    delta = numpy.zeros(N)
    delta.put(indices=right_tube, values=d)
    Delta = numpy.array([numpy.zeros(N),numpy.zeros(N),delta]).transpose()
    # Shift coordinates!
    new_coordinates = central_region.cartesianCoordinates() + Delta*Angstrom
    # Extend (or contract) the unit cell by the same amount
    va, vb, vc = central_region.bravaisLattice().primitiveVectors()
    vc = vc + [0.0, 0.0, d]*Angstrom
    new_cell = UnitCell(va,vb,vc)
    # Rebuild central region (transfer ghost atoms too)
    new_central_region = BulkConfiguration(
        bravais_lattice=new_cell,
        elements=central_region.elements(),
        cartesian_coordinates=new_coordinates,
        ghost_atoms = central_region.ghostAtoms(),
    )

    # Transfer the tags to the new configuration
    for tag in central_region.tags():
        new_central_region.addTags(tag, central_region.indicesFromTags(tag))

    # "Device from bulk" the scripting way
    from NL.CommonConcepts.Configurations.DeviceFromBulk import deviceFromBulk
    left_electrode, right_electrode = device_configuration.electrodes()
    left_electrode_length = left_electrode.bravaisLattice().primitiveVectors()[2,2]
    right_electrode_length = left_electrode.bravaisLattice().primitiveVectors()[2,2]
    new_device = deviceFromBulk(left_electrode_length, right_electrode_length,
                                new_central_region
                )

    return new_device
# ------------------------------------------------------------------------------
d = -5.0
device_configuration = moveTube(device_configuration, d)
```

If you want to understand how this code works, you can study the comments which provide quite detailed explanations. Otherwise, all you really need to know is that by changing the value of d (which is assumed to be given in Angstrom) on the second last line, the right tube is shifted in (negative d) or out (positive d) of the left tube.

Tip: To see the result of the operation, send the script back to the Builder, again using the button. This will add no less than four configurations to the Stash, since the electrodes and central region are also recognized as valid configurations present in the script. If you instead send the script to the **Script Generator**, only the device configuration is transfered.

Comments

•The 10x repetition in the C direction gives a rather large system. This is necessary to be able to set the tube distance d to a large negative value, but also increases the computation time considerably. If you limit the values of d you can reduce the number of repetitions to perhaps 6 or 7, either when building the tubes, or afterwards, by using *Device Tools → Central Region Size*. Just take care so the tubes dont get too close to the opposite edges of the cell, and make sure to inspect visually the geometry of the largest value for d you choose to calculate.

•One can wonder what happens when the tubes are rotated with respect to each other, however the effect is probably small unless perhaps the two tubes have comparable radii such that there is a strong overlap of the pi-orbitals of the two tubes.

•An interesting quantity to compute for systems like this would be the capacitance (see[1] and[2]). This can quite easily be done following a similar procedure as described in the tutorial about a *Atomic-scale capacitance*

References

3.5.5 Capping a carbon nanotube

In this tutorial you will learn how to create a capped nanotube. Specifically you will:

•Create a nanotube.

•Cut a C_{60} fullerene in half - this will be the cap.

•Attach the cap to the nanotube.

•Clean up a bit.

In this tutorial we will build a (5,5) nanotube capped by a C_{60} hemisphere. There are, however, multiple other combinations of nanotubes and fullerenes that match, but such capped tubes can also be constructed using the approach presented here.

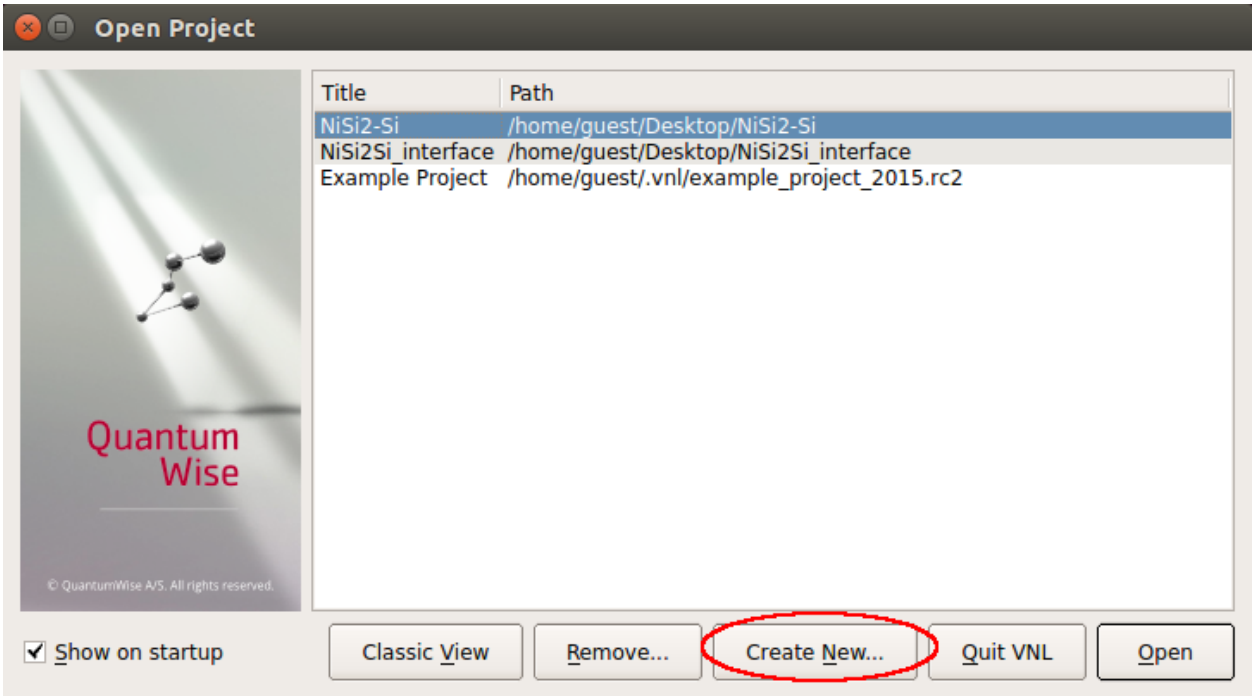
Please see also the video tutorial below. Many more are available on the QuantumATK YouTube channel.

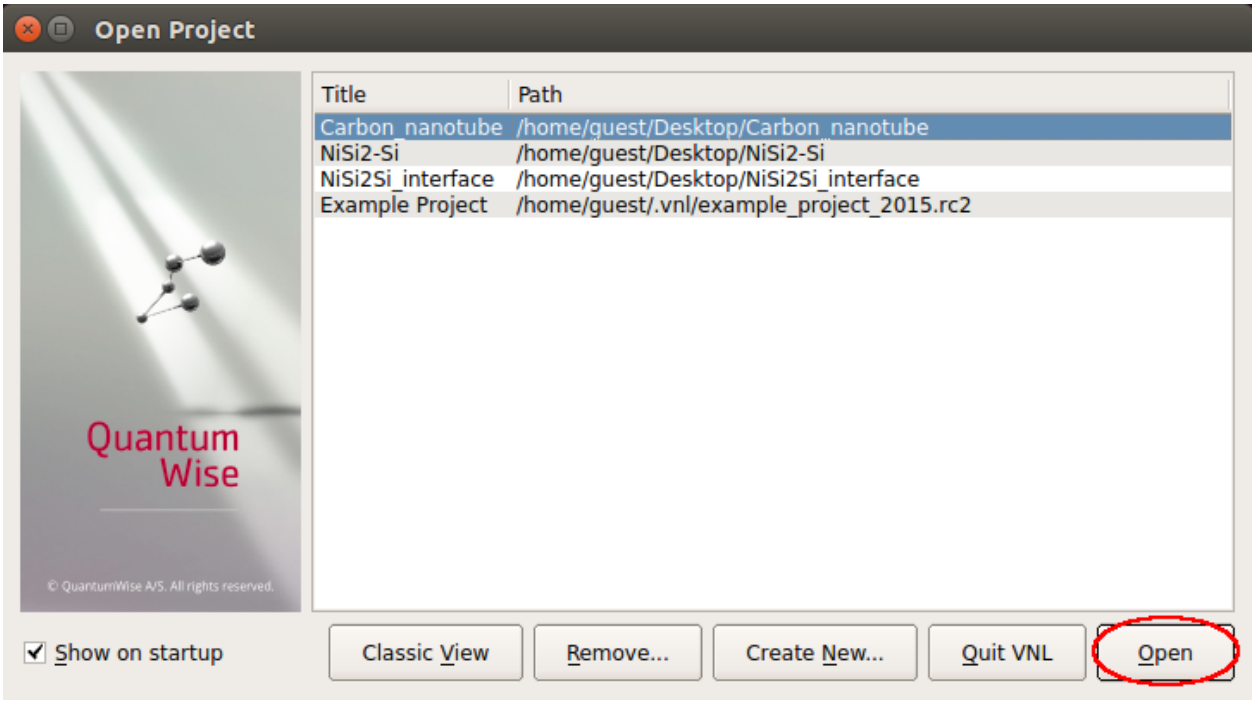
[1] Pawel Pomorski, Christopher Roland, Hong Guo, and Jian Wang. First-principles investigation of carbon nanotube capacitance. *Phys. Rev. B*, 67:161404, Apr 2003. doi:10.1103/PhysRevB.67.161404.

[2] Pawel Pomorski, Lars Pastewka, Christopher Roland, Hong Guo, and Jian Wang. Capacitance, induced charges, and bound states of biased carbon nanotube systems. *Phys. Rev. B*, 69:115418, Mar 2004. doi:10.1103/PhysRevB.69.115418.

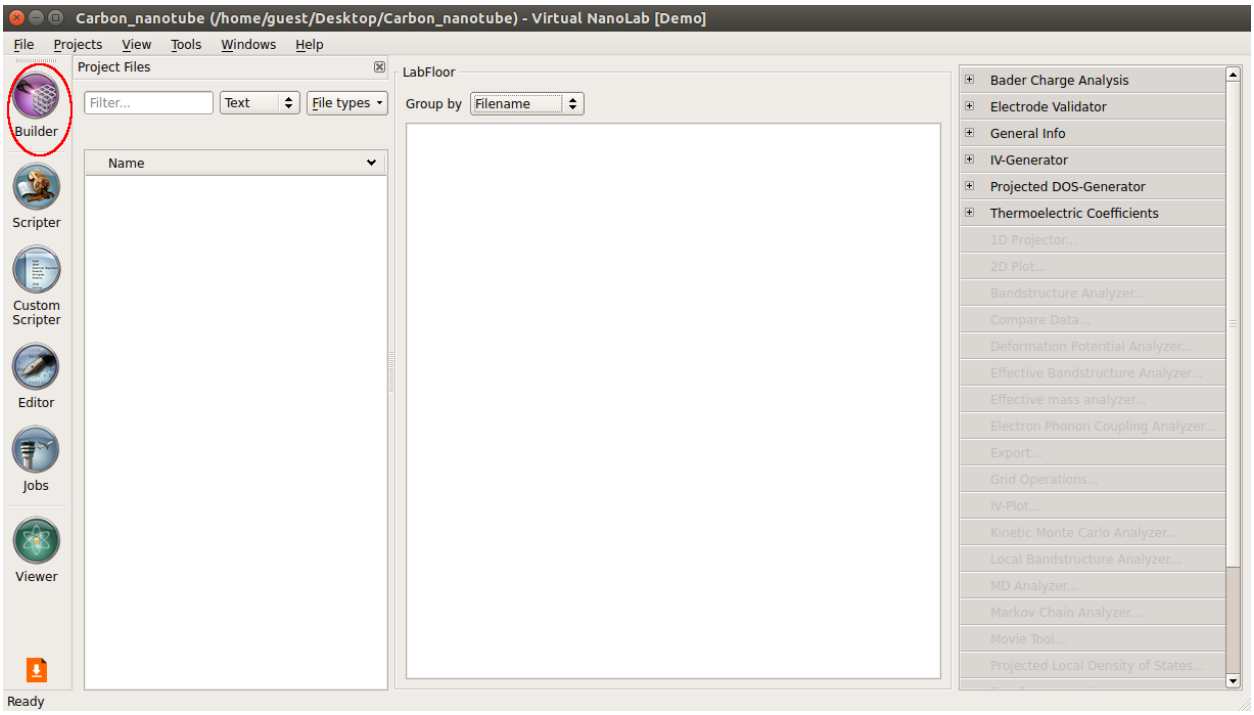
Build an extended (5,5) carbon nanotube

1. Open **QuantumATK** and create a new project by clicking **Create New**. Give the project a *Title* (here: Carbon_nanotube), select the folder where data will be stored, then click **OK** and **Open** to start the project.

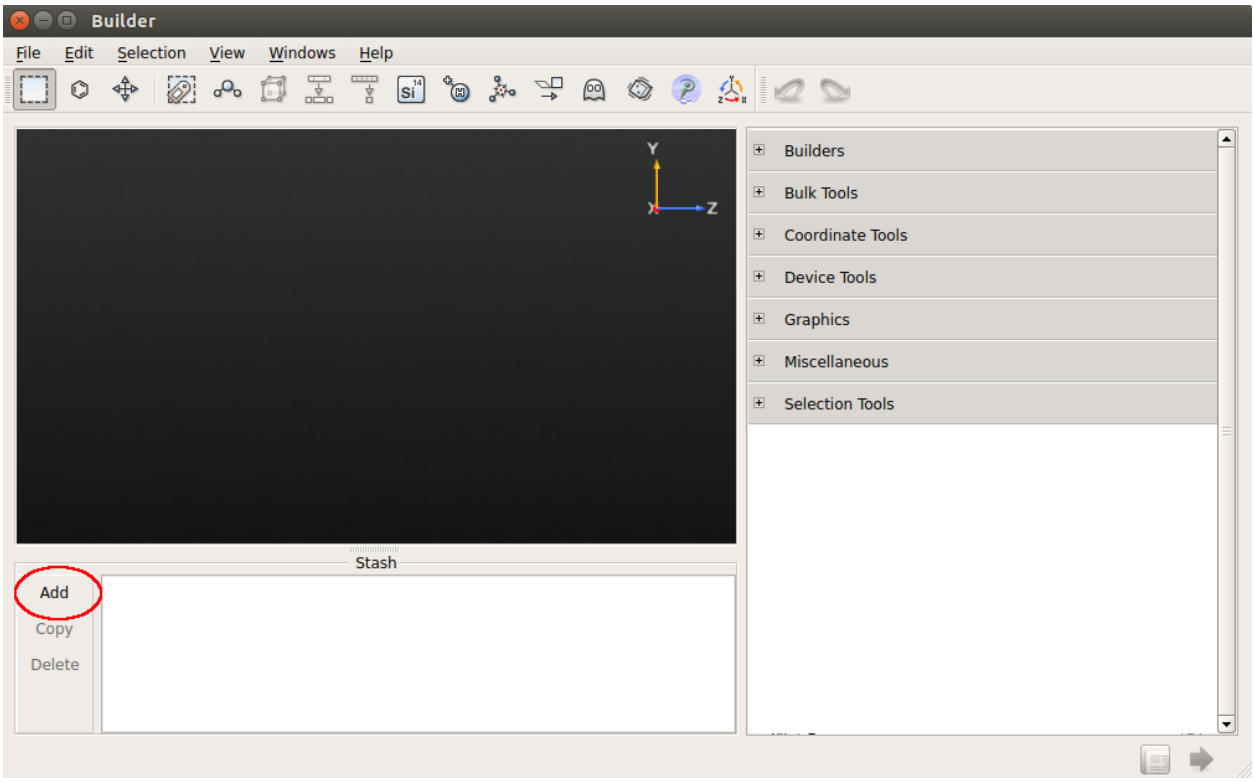


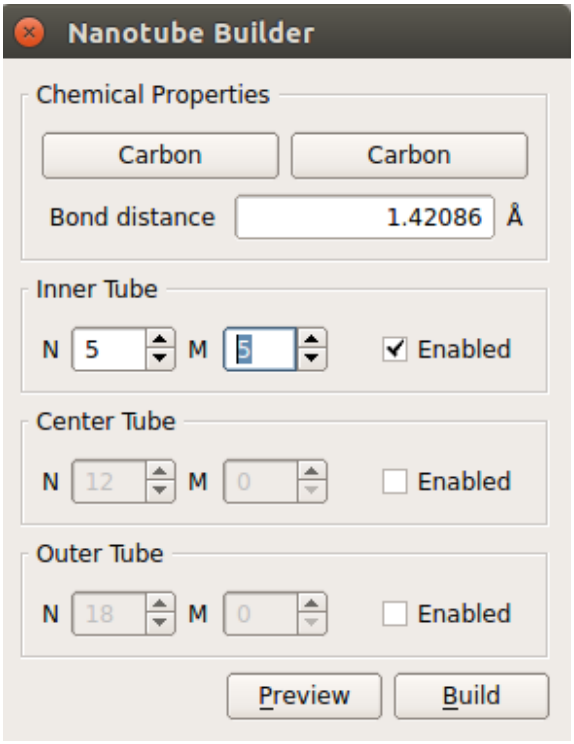


2.In the **QuantumATK** main window, click on the icon to open the **Builder**.

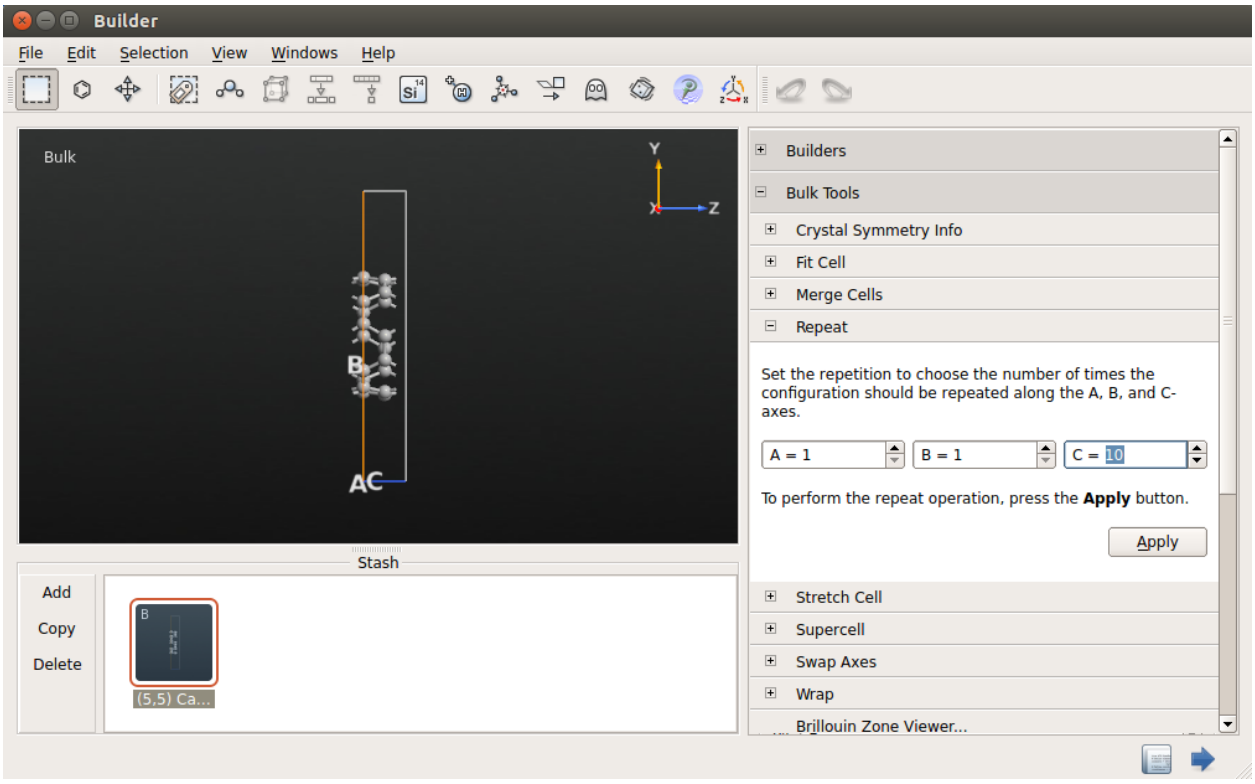


3.Go to the **Stash**, click *Add* → *From Plugin* → *Nanotube*, and the **Nanotube Builder** window will come up. Here, set the parameters as shown in the figure bellow (M = N= 5), and press *Build*.

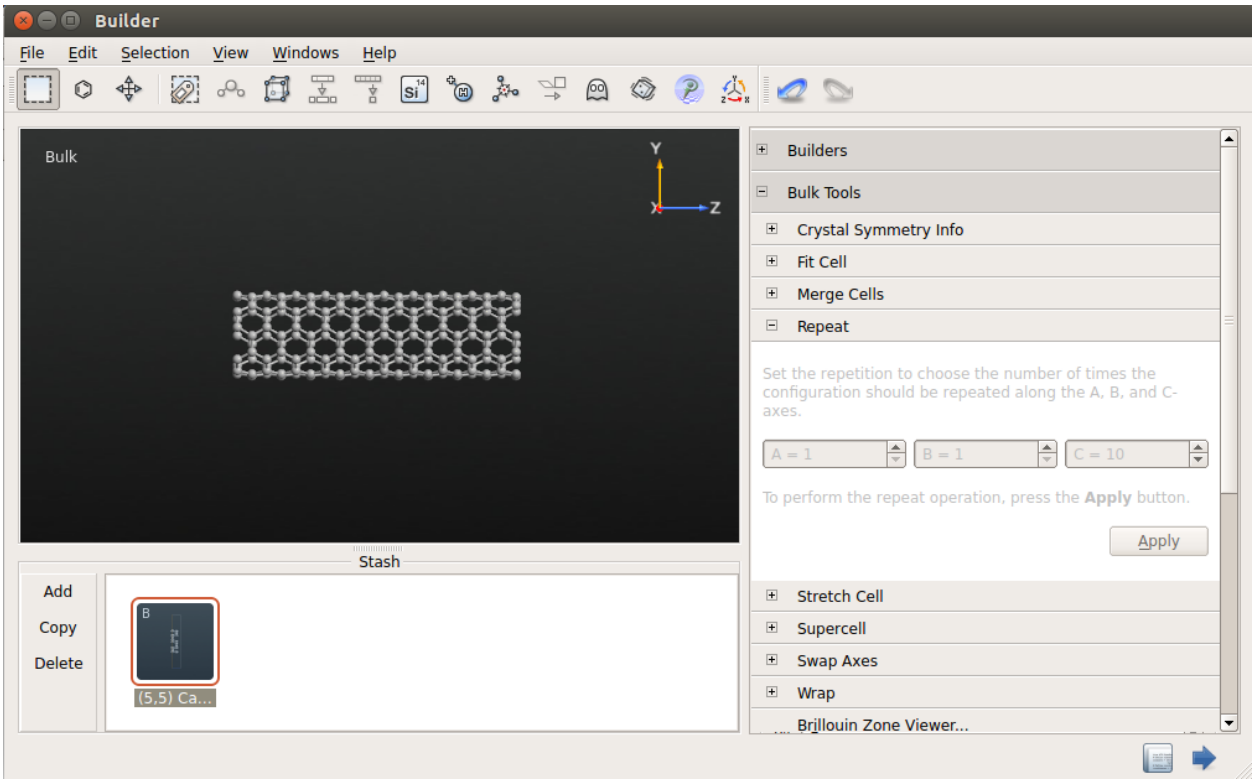




4.In the **Builder**, go to *Bulk Tools* → *Repeat* to extend the structure 10 times in the C-direction, as shown in the figure that follows, and Press *Apply* and after `Crtl` + `R` to see the whole structure properly.



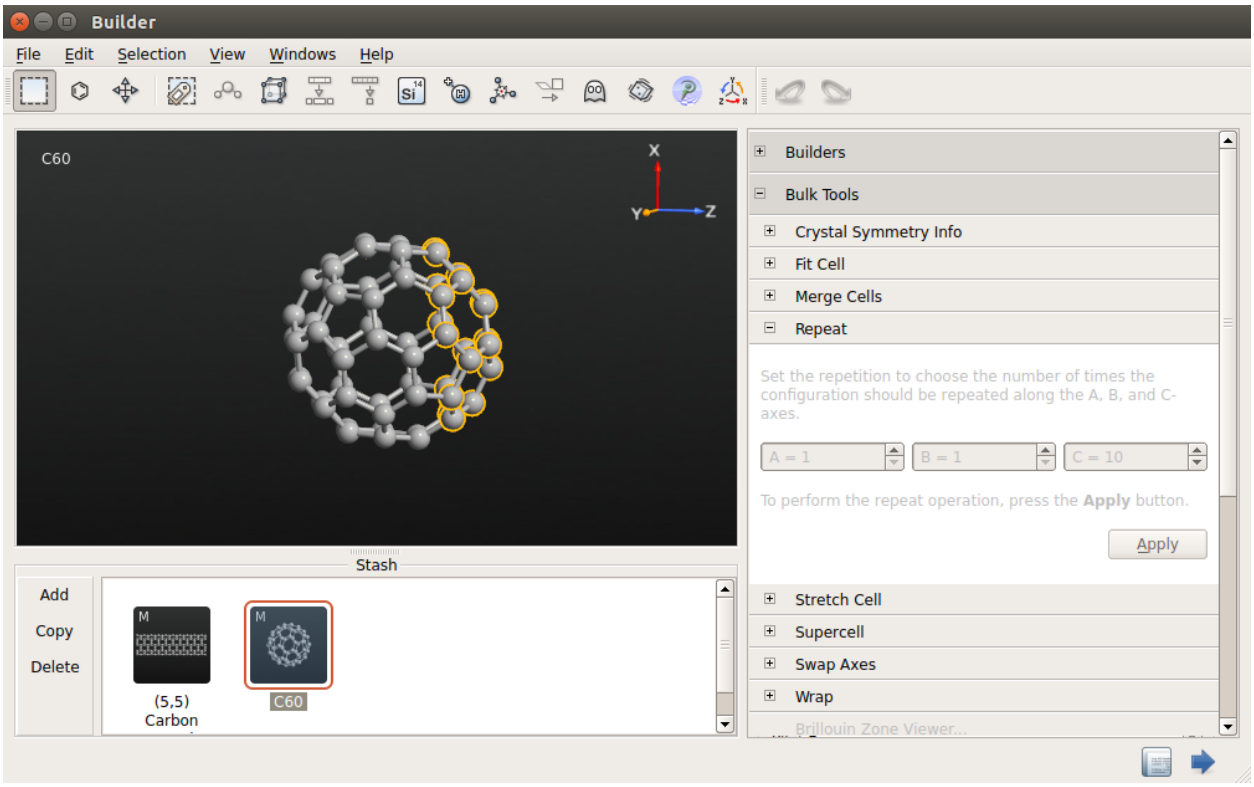
5.Remove the unit cell by clicking the button. The capped tube is anyway not periodic, so we build it as a molecule.



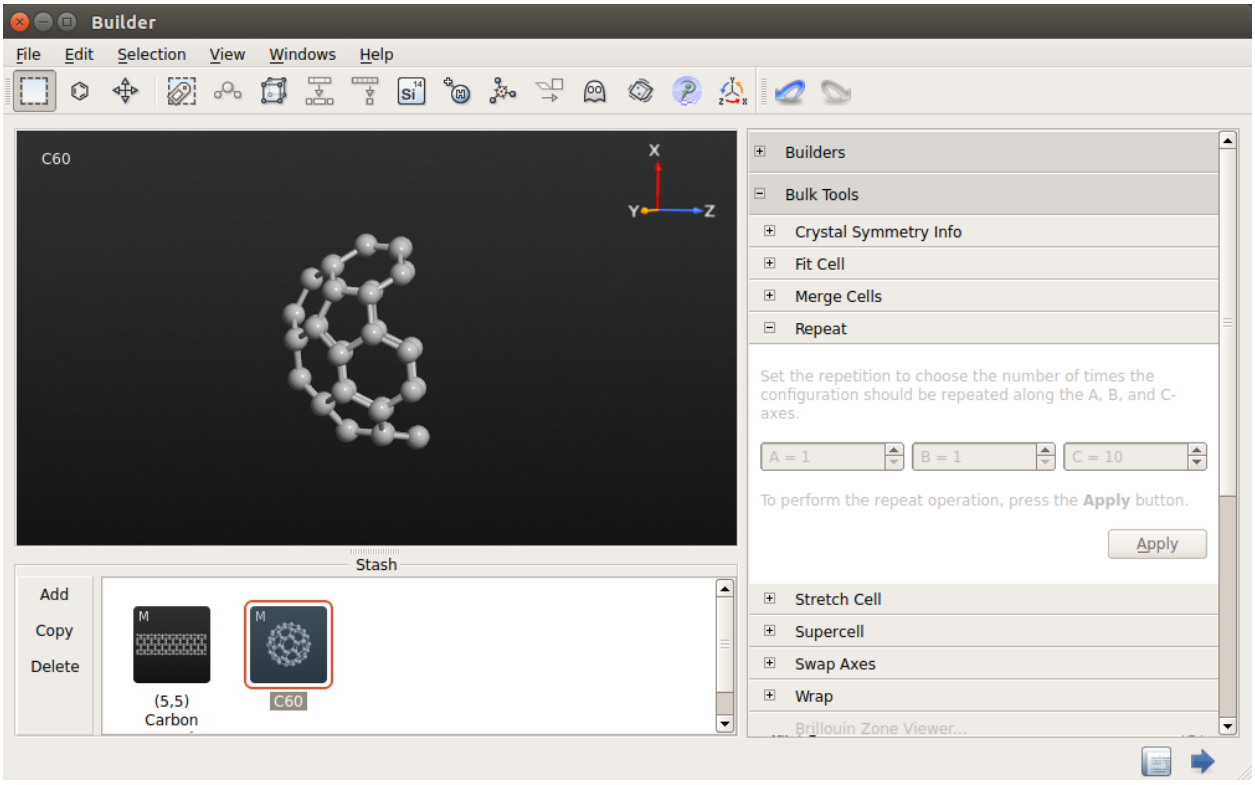
Cut the fullerene in half

1.In order to insert the C_{60} molecule in the **Stash** click on *Add* → *Add From Database*, and choose *Databases* → *Fullerenes* from the menu. Locate C_{60} in the list. Press the icon, or double-click its line in the list, to send it to the **Stash**.

2.In the C_{60}-CNT (5,5) system the tip of the cap is a pentagon. Imagine now a coordinate system in the screen plane where east is right, north up, south down, and west left. Then rotate the fullerene so that the western face of the system is a pentagon, while keeping the Z axis pointing to the east and the X axis roughly to the north.

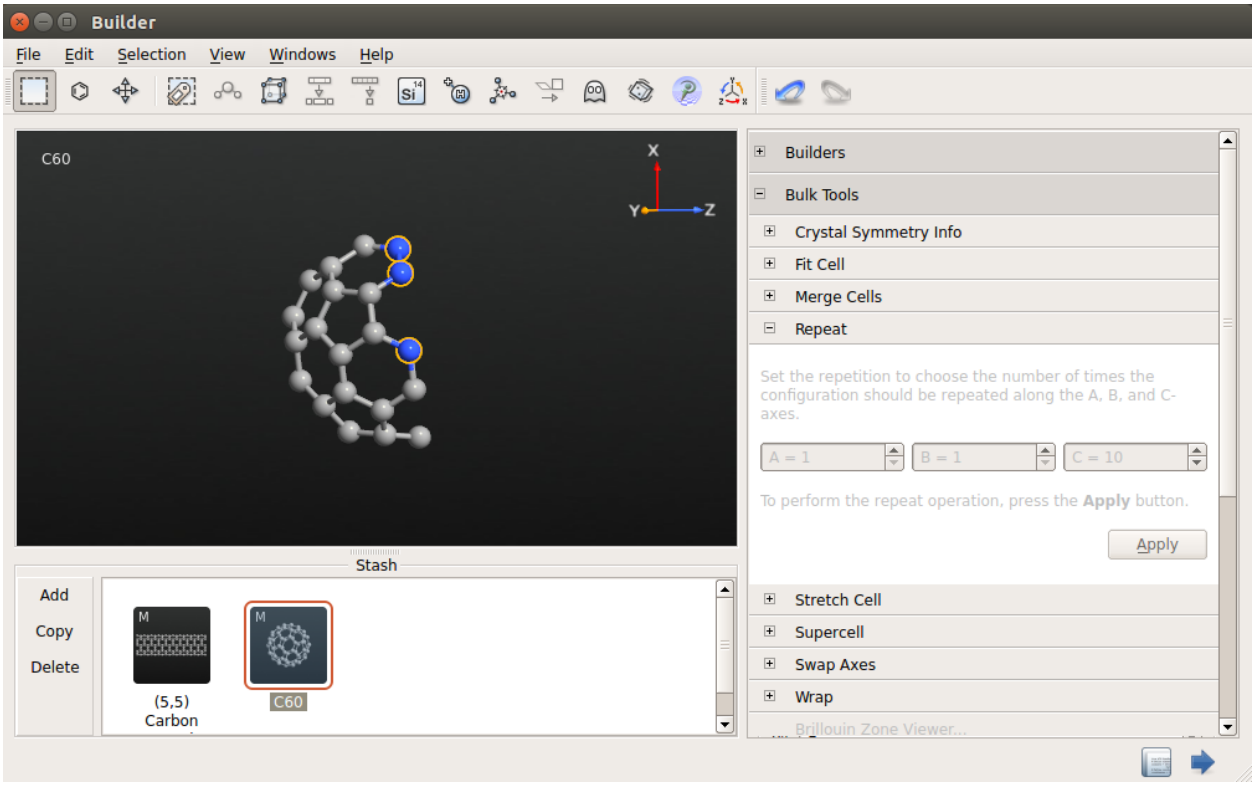


3. The cap is made from 30 carbon atoms, corresponding to half the C_{60}. If, however, you cut the molecule in half, it becomes very difficult to attach it to the nanotube. The trick is instead to leave one extra layer of atoms, which match almost perfectly the edge of the nanotube. Therefore, select the atoms as shown in the picture above, and delete them.



4. The edge of the cap consists of pairs of atoms - just like the edge of the tube. Thus, select 3 atoms as shown in the picture below - two in a pair and one more in the neighboring pair - and change them to nitrogen using the button.

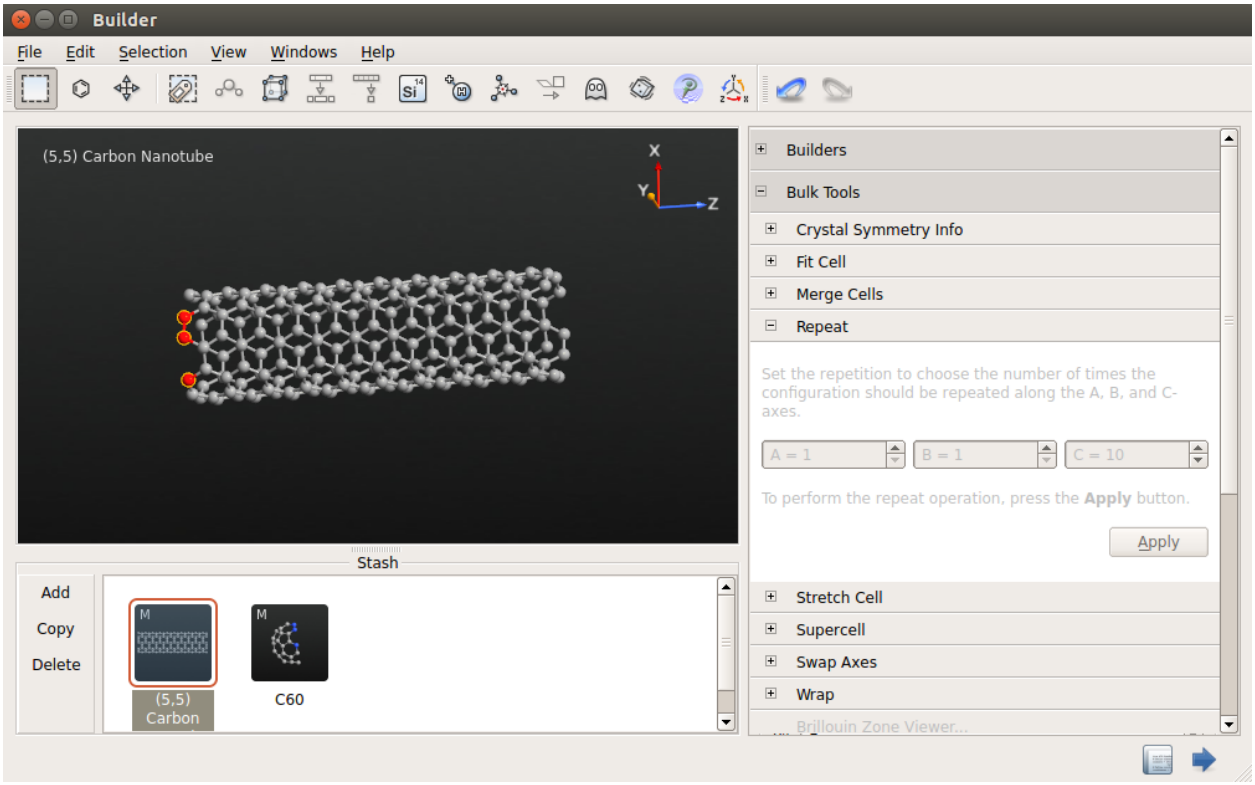
   This is just to make it easier to identify them in the next step.



Capping the tube

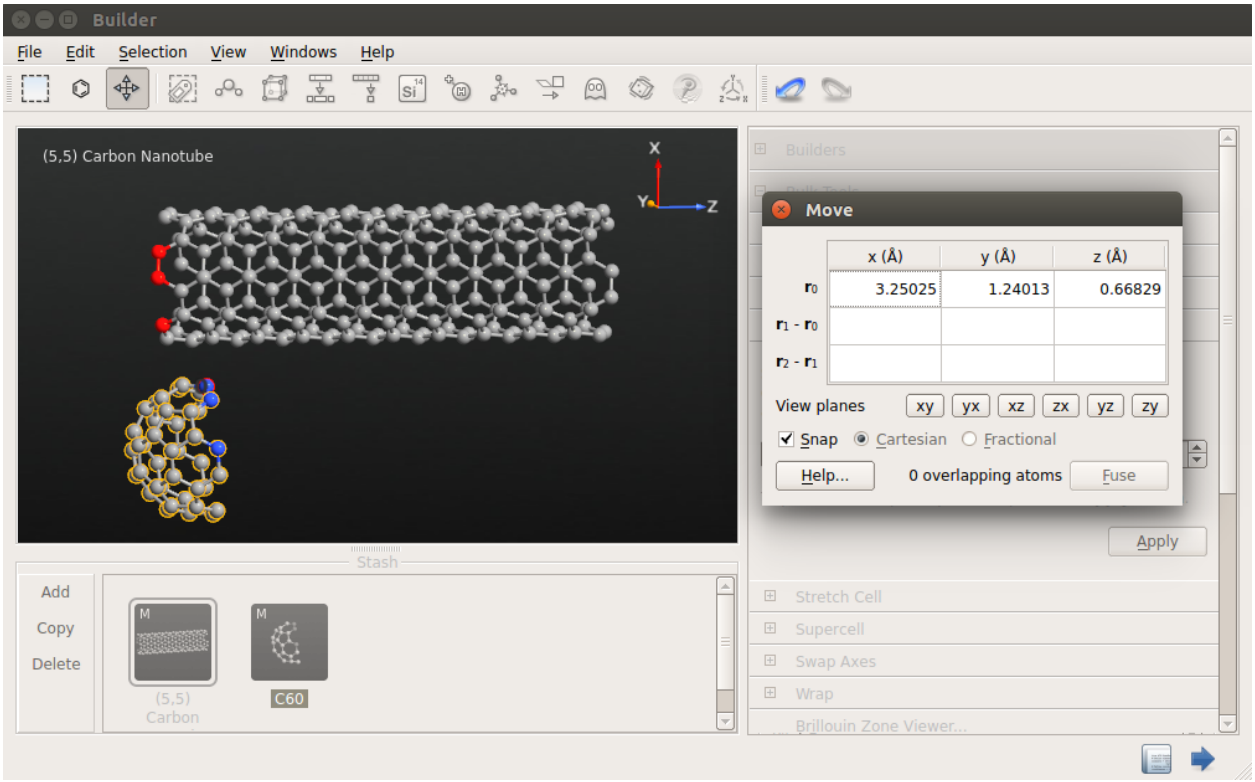
1. In the **Stash**, double-click on the *(5,5) Carbon Nanotube* object.

2. Select three equivalent atoms in the tube, as you did in the cap. By equivalent it means that the edge of the tube looks just like the edge of the cap, with pairs of atoms, and the goal is to match up these pairs. Change these three atoms to oxygen, again to make them easy to identify.



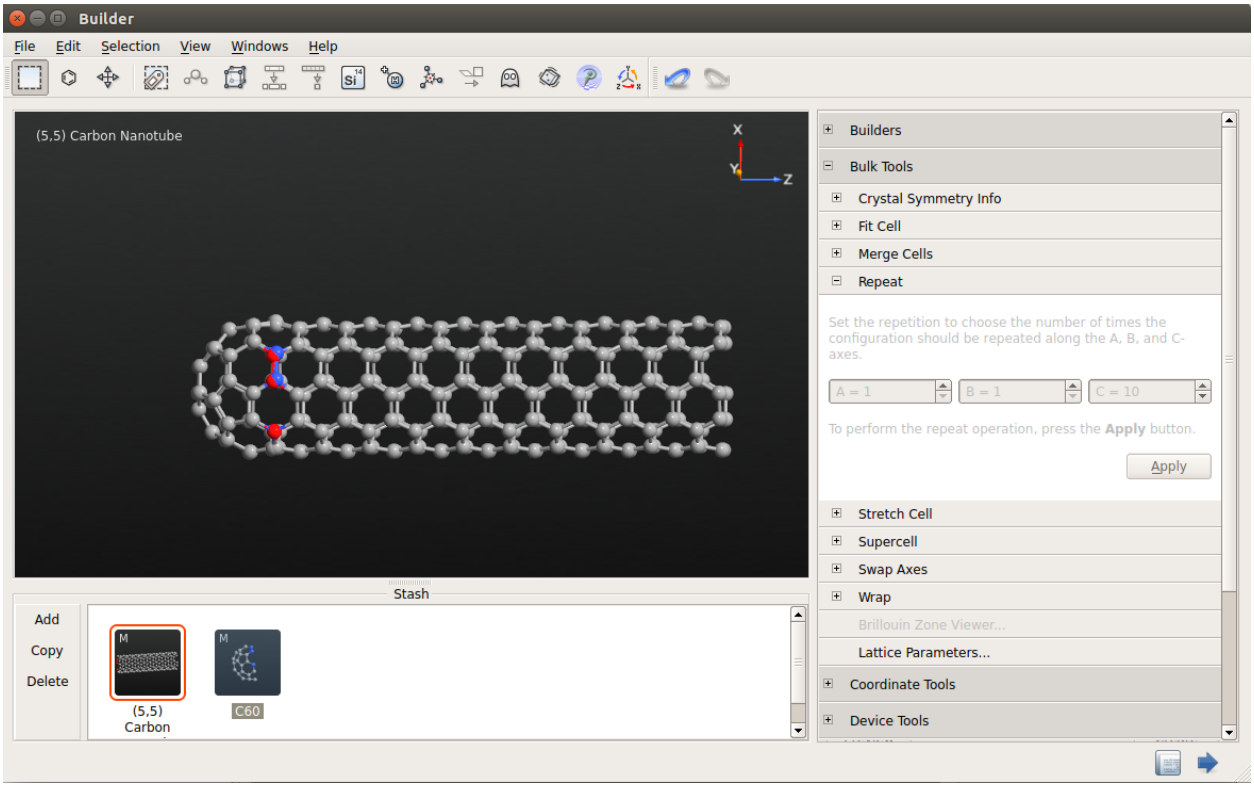
3. Drag and drop the C_{60} object from the **Stash** to the **3D window** to import the cap. The next few steps may differ slightly in precise detail from the images, depending on which precise atoms you changed to nitrogen.

4. When the cap was imported, the **Move** tool was automatically activated. Select one of the nitrogen atoms (as shown in the picture, the first atom in the pair which is not next to the lone nitrogen atom) - its selection marker turns red and the atom is labeled 0. Then drag it so that it ends up on top of the equivalent oxygen atom in the tube.



5. Then select the second nitrogen atom in the pair. When you now drag this atom (red selection marker, and labeled 1), the cap will not move, it will rotate around the bond between atoms 0 and 1. Rotate the cap in place by making atom 1 overlap the next oxygen atom in the pair on the tube edge. You may need to rotate the view to get a clear aim.

6. Finally, select the third nitrogen atom, and move it on top of the third oxygen atom (dihedral rotation). The final structure is shown below.

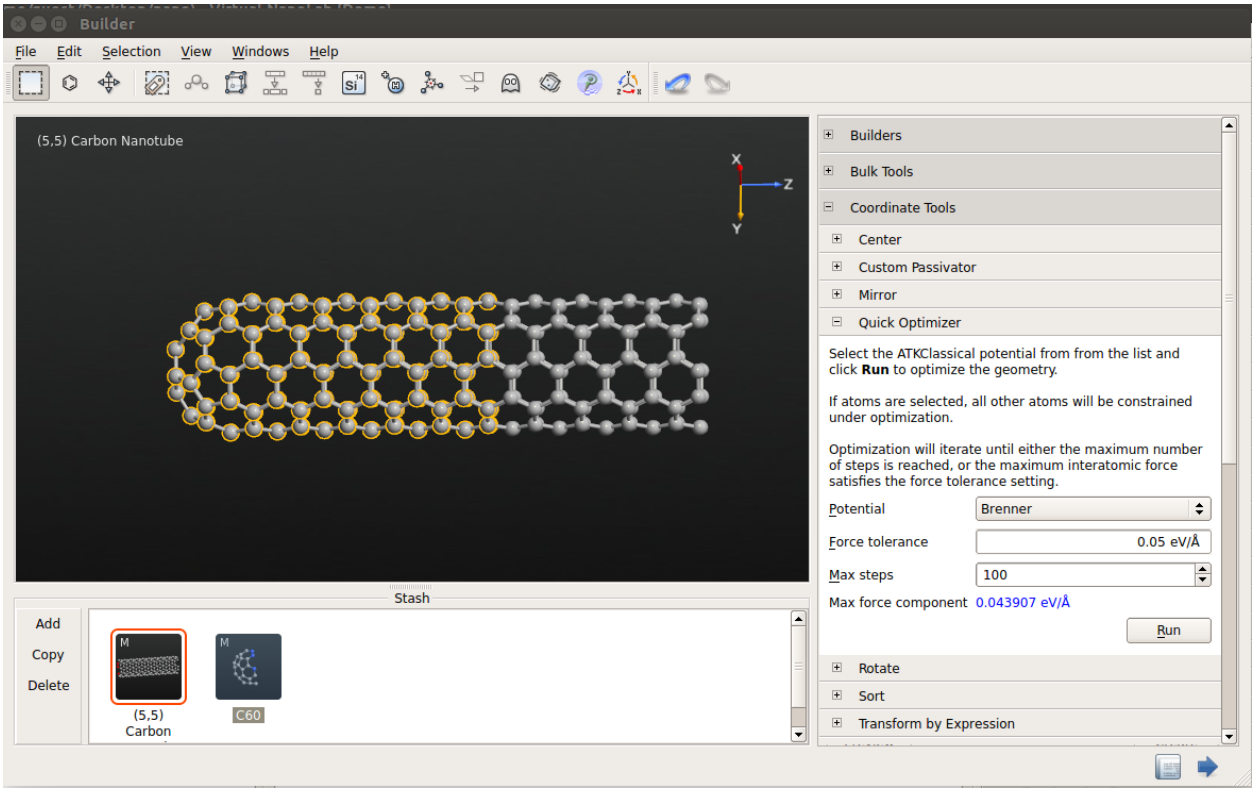


In case you did not succeed (sometimes the cap becomes inverted), undo the last two steps, unselect all anchor atoms by clicking the background, select only one atom and move the cap out a bit from the tube, and try again from 4. It can also be a good idea to pre-align the cap with the tube - see the video above for details!

Finalizing the geometry

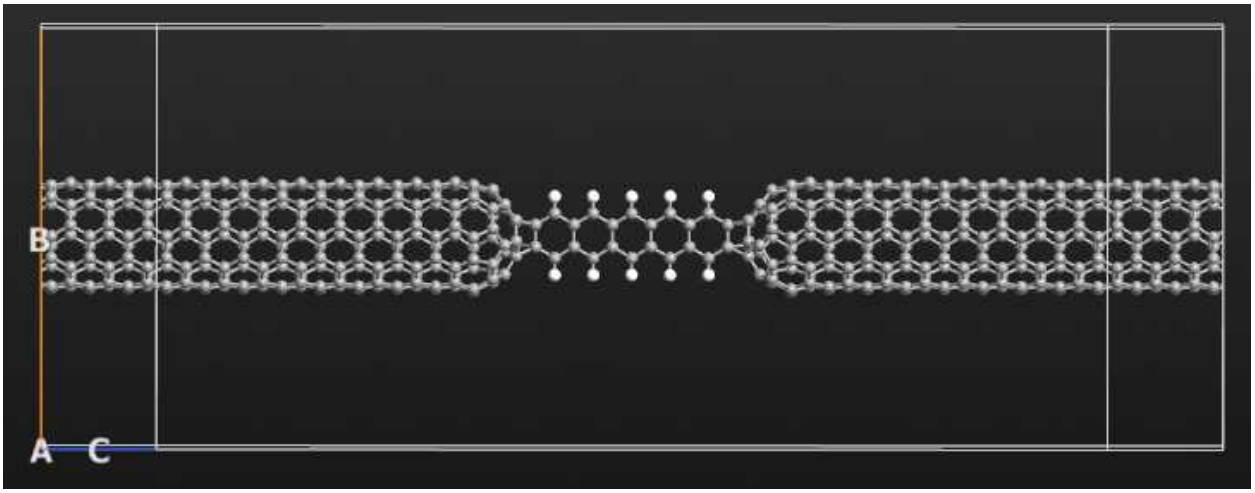
At this point there are overlapping atoms - in fact we utilized this to construct the geometry. You could use Fuse in the **Move** tool, but there are atoms which are only almost overlapping, which are not detected.

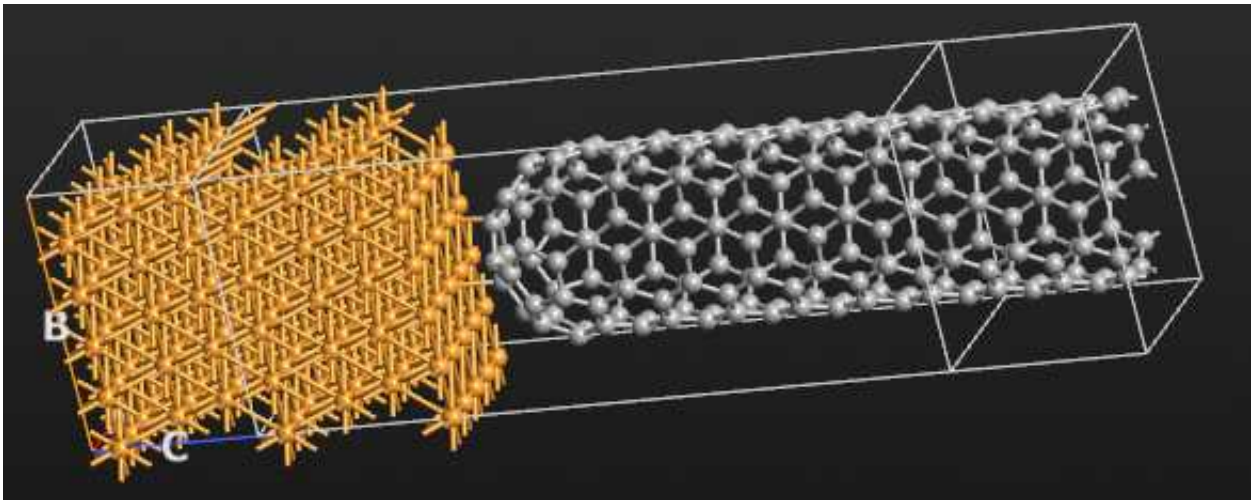
1. Instead, close the **Move** tool, and open *Selection tools* → *Close Neighbors*, enter a distance of 0.7 Å and press *Select* (10 atoms should be selected) and then click `Delete`.
2. Change all atoms back to carbon.
3. You need to optimize the cap, since the radius of the fullerene is slightly different from the naontube. You will however at the same time want to keep the other end of the nanotube as it is (perfectly periodic) so that we can later extend it, if necessary. Therefore, first select the cap and a few more layers of atoms in nanotube (about half of the tube should be fine). Then, open *Coordinate Tools* → *Quick Optimizer*, set the maximum number of iterations to 100, and click *Run*. The tool uses the *ATKClassical* potentials, so the optimization only takes a few seconds.



Connection to other structures

Building this structure just by itself is not very meaningful - you will want to connect it to something, like a surface or molecule. This will be covered by other tutorials, where you will learn how to build structures like those in the pictures below.

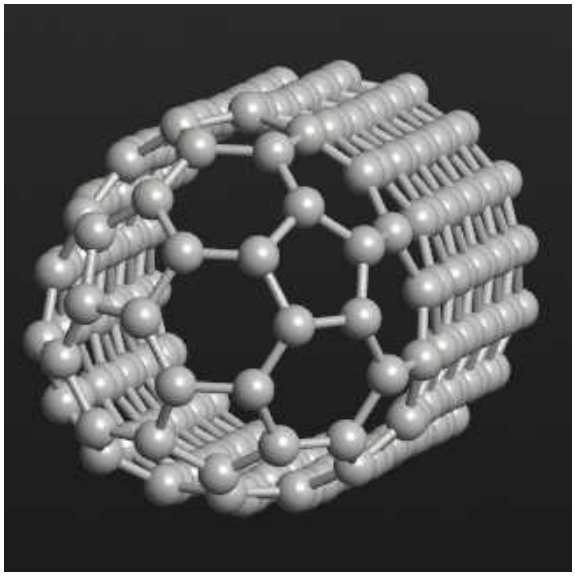




Below is selection of other capped nanotubes that can be built with a very similar approach. These system are slightly more difficult to build, and require some hand tuning.

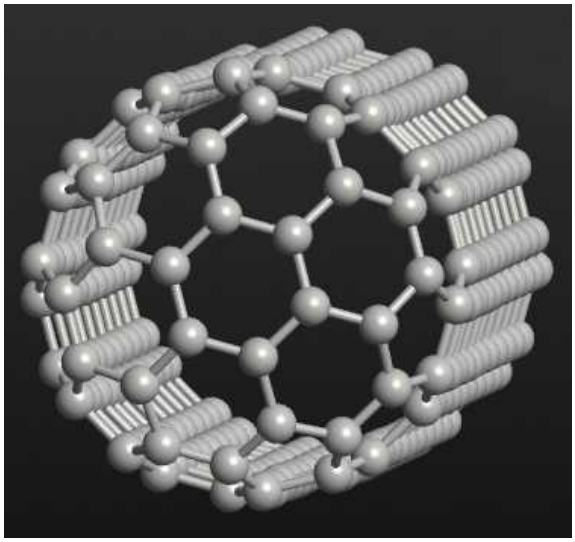
(9,0) CNT + C_{60}

The cap tip is a hexagon and the edge of the tube is zigzag.



(6,6) CNT + fullerene

The cap tip is made from fullerene No. 134 (C_{3v} symmetry) from the QuantumATK database.



3.5.6 Simple carbon nanotube device

In this tutorial you will build a single-wall carbon nanotube (CNT), relax it along the tube length, and then turn it into a device configuration suitable for nanoscale transport studies.

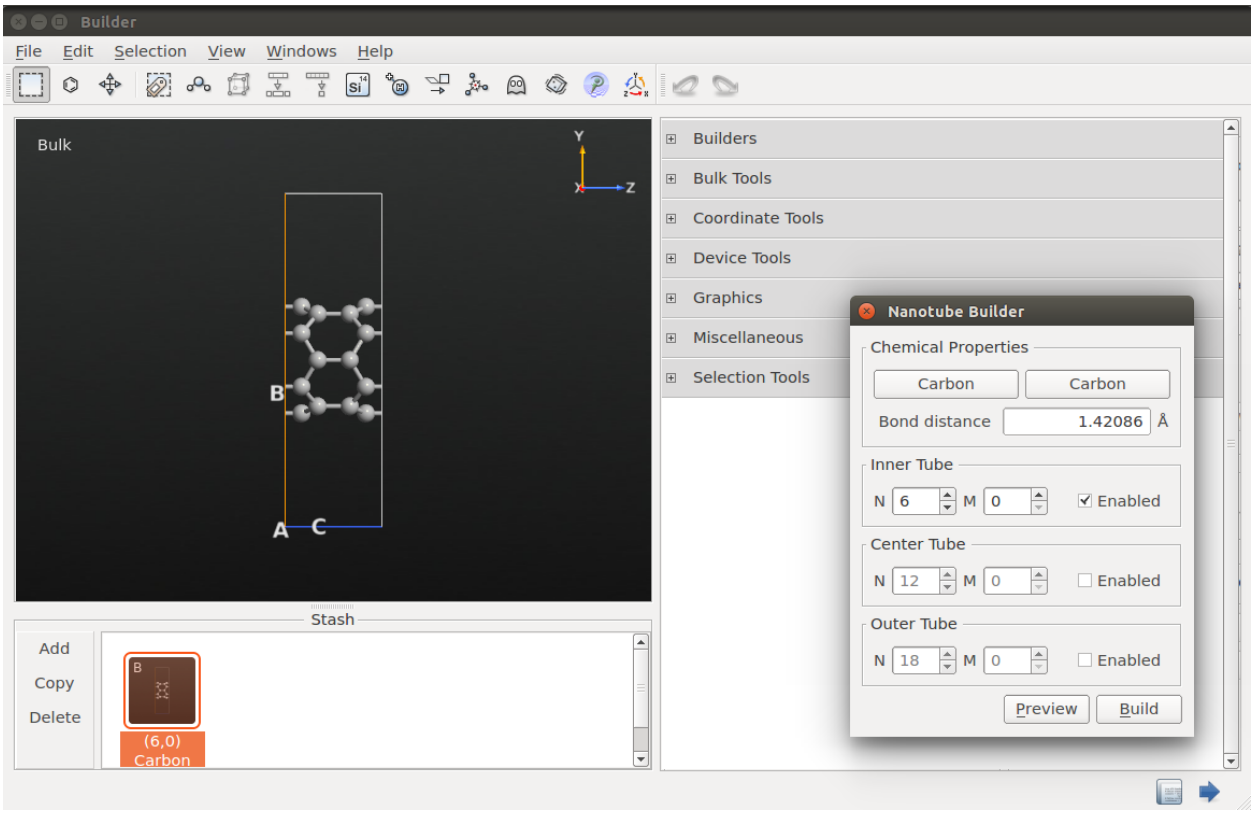
Attention: This is a two-step procedure:

1. Build and geometry optimize a single CNT unit cell.
2. Elongate the relaxed CNT and turn it into a device configuration.

Note: Geometry relaxing the CNT is particularly important for phonon calculations: Negative phonon frequencies will appear in the phonon band structure if the configuration is not properly relaxed.

Build and geometry optimize a short CNT

Open the QuantumATK **Builder** , and use *Add* → *From Plugin* → *Nanotube* to create a (6,0) carbon nanotube.

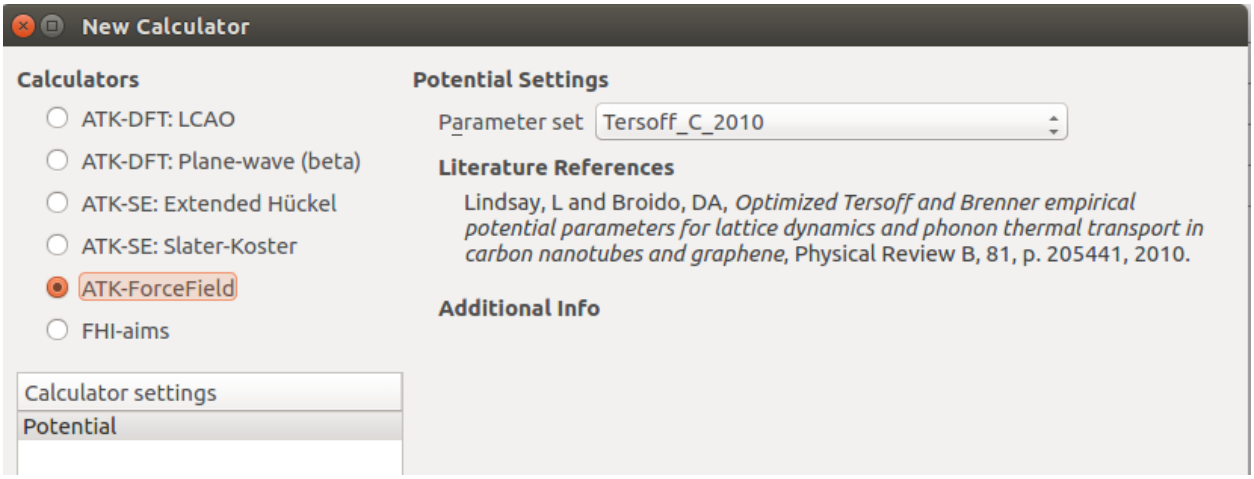


Then send the configuration to the **Script Generator** by using the botton. In the Scripter, add the following blocks:

- New Calculator.
- *Optimization* → *OptimizeGeometry*.

Change the output file to `cnt.hdf5` and double-click the New Calculator block to edit the settings:

•Select the ATK-ForceField calculator and the Tersoff_C_2010 potential.



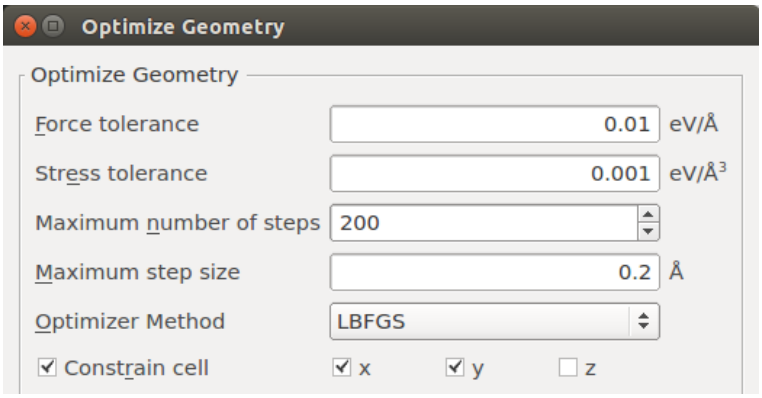
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

Then open the OptimizeGeometry block and edit it:

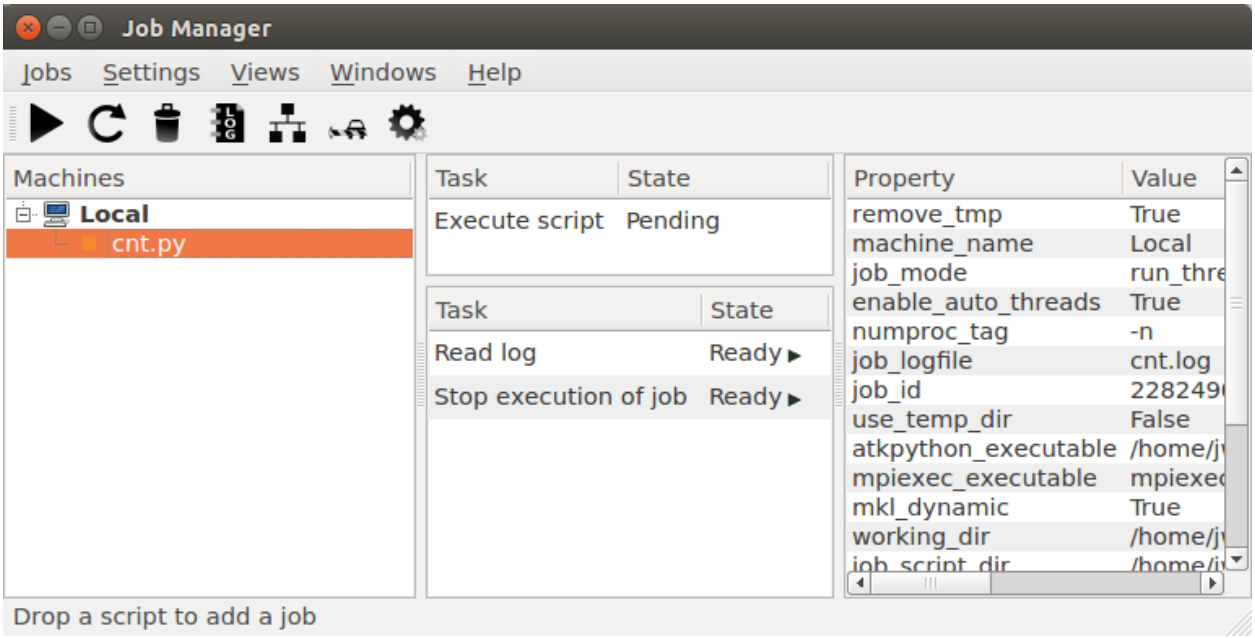
•Set the maximum force to 0.01 eV/Å.

•Set the maximum stress to 0.001 eV/Å^3.

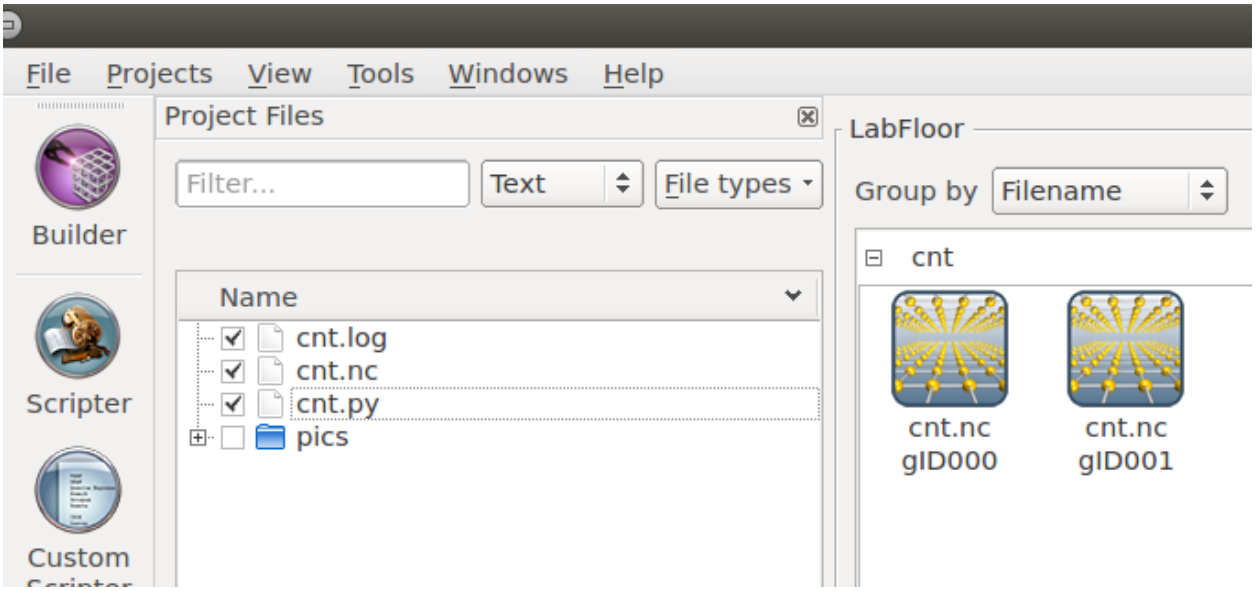
•Remove the checkmark for z under Constrain cell to allow relaxation of the cell along the periodic direction.



Send the script to the **Job Manager**, save the script as `cnt.py`, and run the calculation by clicking. The calculation uses a classical potential, so it should be very fast.



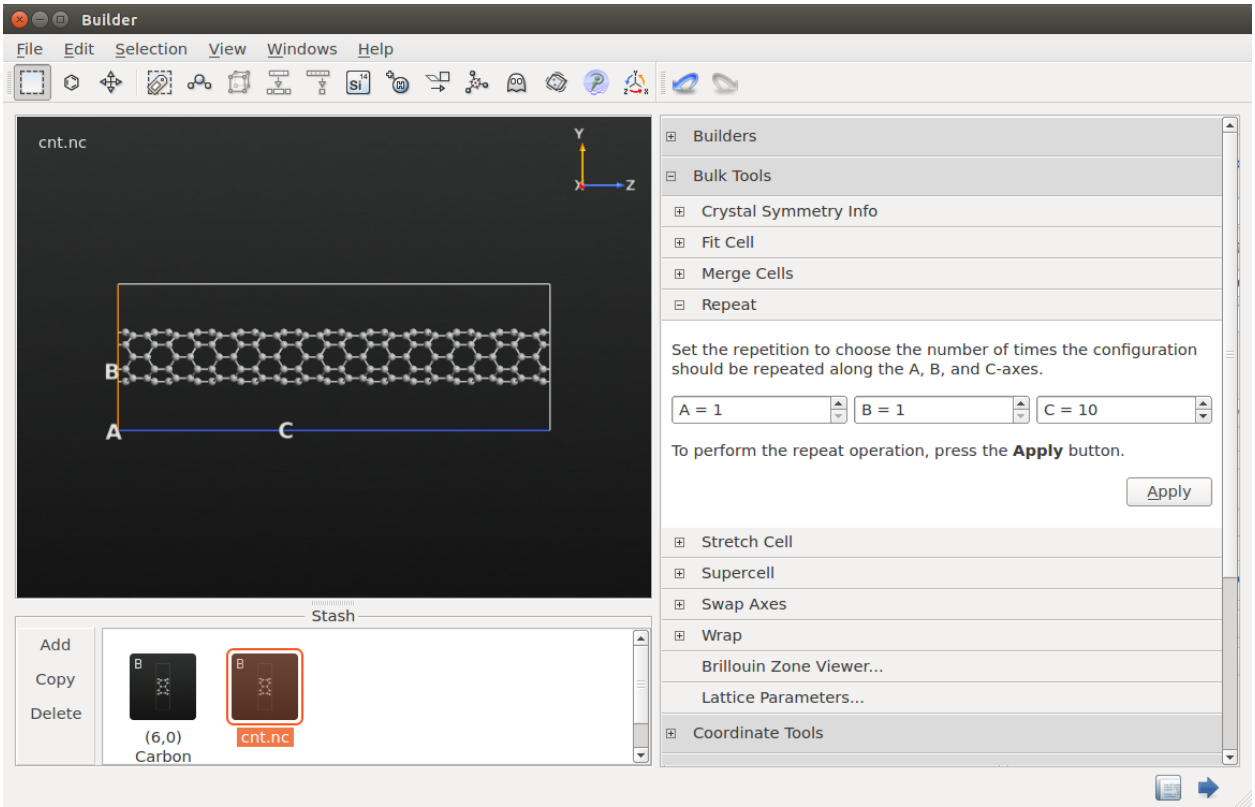
Once finished, you should find the output `cnt.hdf5` on the QuantumATK **LabFloor**. It contains two objects of the type `bulk_configuration` - the initial (`glD000`) and the optimized (`glD001`) CNT structures. You can click an object to highligt it and then use the **Viewer** to visualize it.



CNT device configuration

Drag and drop the optimized CNT onto the **Builder** . Then do the following to make it longer:

- In the right-hand side panel, open the *Bulk Tools* → *Repeat* plugin.
- Enter a repetition of 10 along C (the tube axis) and click **Apply** to create a longer CNT.



You are finally ready to convert the configuration to a device and save it:

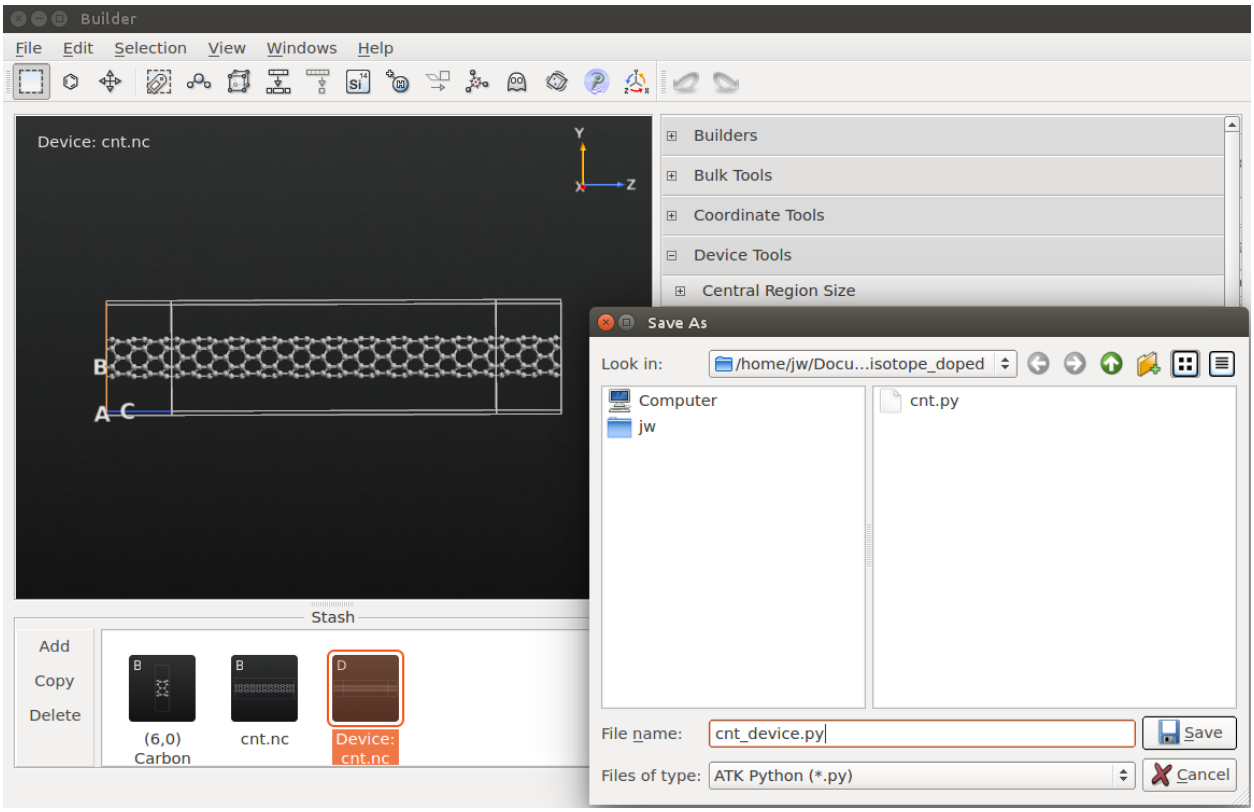
•Open the *Device Tools* → *Device From Bulk* plugin.

•Leave the electrode lengths at default.

•Click **OK** to create the device.

•Highlight the device configuration, right-click it, and choose **Save as** in the drop-down menu. You can now save the configuration in either a HDF5 file (`cnt_device.hdf5`) or as a Python script (`cnt_device.py`).

•Note that the CNT device consists of a **central scattering region** sandwiched between left and right **electrodes**.



Attention: The instructions above result in a CNT device with a central region that is roughly 43 Å long. This is a somewhat arbitrary choice and it may well be that specific applications require a shorter or a longer central region!

The central region and electrode lengths are easily modified:

•Highlight the device Stash item.

•Use the *Device Tools* → *Central Region Size* plugin to change the central region length along the C axis.

•Similarly, the *Device Tools* → *Electrode Size* plugin lets you change the electrode lengths.

Tip: What next? You have created a carbon nanotube device configuration. You can now use it in scientific studies of electron and phonon transport in CNTs.

Thermoelectrics For example, you can check out the tutorial *Thermoelectric effects in a CNT with isotope doping*, where phonon and electron transmission spectra, thermal conductance, and the *ZT* product is calculated for CNTs with and without isotope impurities.

Scripts For your reference, you can download the CNT device configuration created in this tutorial here: `cnt_device.py`.

3.5.7 Thermoelectric effects in a CNT with isotope doping

Although the mass of the nucleus has almost no influence on the electronic properties of an atom (at least in the Born-Oppenheimer approximation), different isotopes will have different thermal properties. The consequences of isotope doping are very important for the case of 1D nanostructures and nanotubes in particular[1],[2],[3]. It is therefore interesting to study how the thermal properties of a CNT (carbon nanotube) change when the isotope distribution in one part of the tube is changed[4].

In this tutorial, you will study the effect of doping a carbon nanotube with a ring of ^{14}C atoms. Specifically, you will calculate the phonon and electron transmission spectra, the thermal conductance, and the *ZT* product. You will use the **ATK-ForceField** engine for phonon calculations, while electronic transmission is calculated using **ATK-SE**.

Attention: This tutorial assumes that you have already finished the tutorial *Simple carbon nanotube device*, where the CNT device configuration used in this tutorial is created. If not, please go through that tutorial first, or simply download the configuration here: `cnt_device.py`.

CNT device with tags for ^{14}C doping

Drag and drop the undoped CNT device configuration (`cnt_device.py`) onto the **Builder** . Then, assign a tag to the ^{12}C atoms that you will convert to ^{14}C:

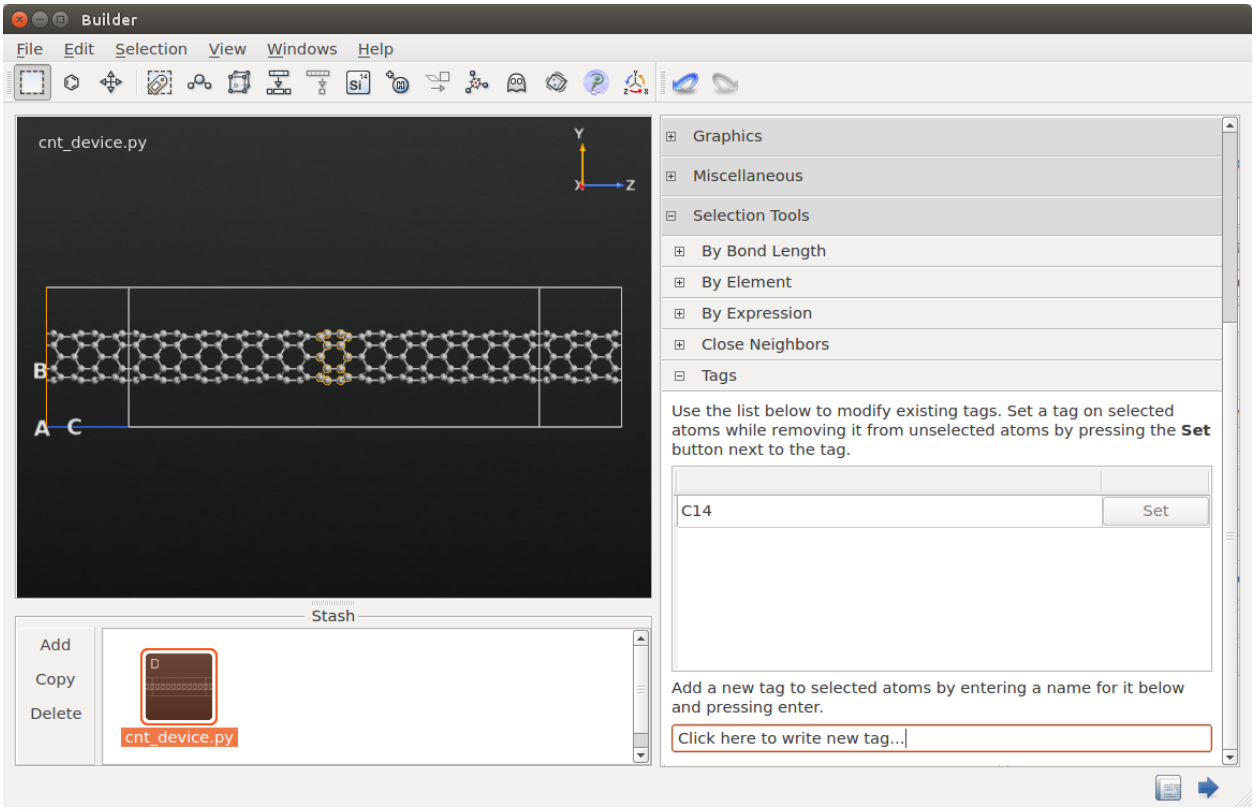
- Select a ring of C atoms in the middle of the structure as shown below.
- Open the *Selection Tools* → *Tags* plugin, and assign the tag C14 to these atoms.

[1] Yong-qiang Cheng, Shu-yun Zhou, and Bang-fen Zhu. Isotope effect on phonon spectra in single-walled carbon nanotubes. *Phys. Rev. B*, 72:035410, Jul 2005. doi:10.1103/PhysRevB.72.035410.

[2] Gang Zhang and Baowen Li. Thermal conductivity of nanotubes revisited: Effects of chirality, isotope impurity, tube length, and temperature. *The Journal of Chemical Physics*, 2005. doi:10.1063/1.2036967.

[3] C.ăW. Chang, A.ăM. Fennimore, A.ăAfanasiev, D.ăOkawa, T.ăIkuno, H.ăGarcia, Deyu Li, A.ăMajumdar, and A.ăZettl. Isotope Effect on the Thermal Conductivity of Boron Nitride Nanotubes. *Phys. Rev. Lett.*, 97:085901, Aug 2006. doi:10.1103/PhysRevLett.97.085901.

[4] Jin-Wu Jiang, Jian-Sheng Wang, and Baowen Li. A nonequilibrium Greens function study of thermoelectric properties in single-walled carbon nanotubes. *Journal of Applied Physics*, 2011. doi:10.1063/1.3531573.



Phonon transmission

First set up and run the calculation of the phonon transmission spectrum for the undoped CNT. Transfer the configuration to the **Script Generator** and add the following blocks:

- New Calculator
- *Analysis* → *PhononTransmissionSpectrum*

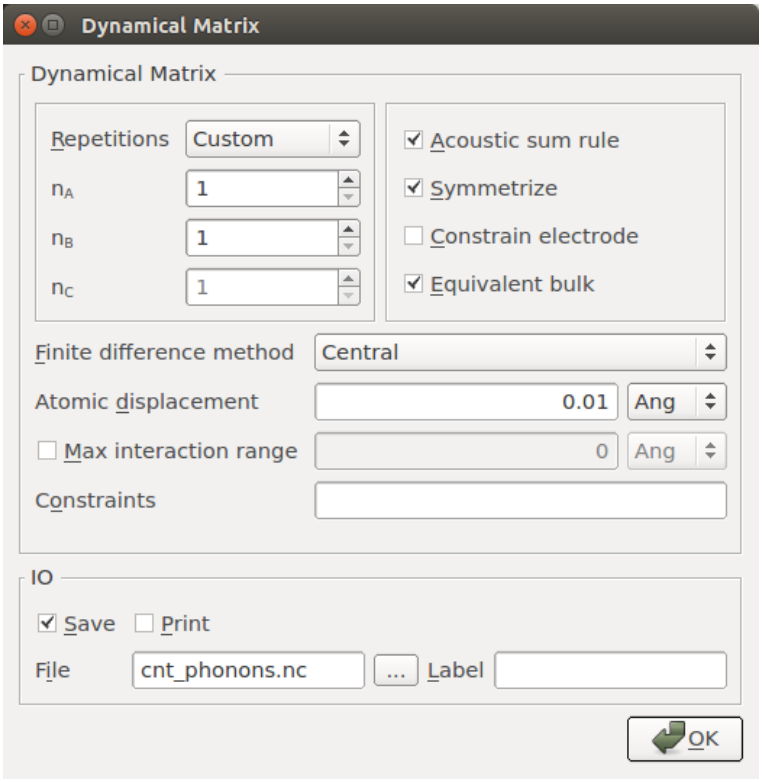
Note that a DynamicalMatrix object was automatically added as well. Change the output file to `cnt_phonons.hdf5` and edit the settings in all three blocks:

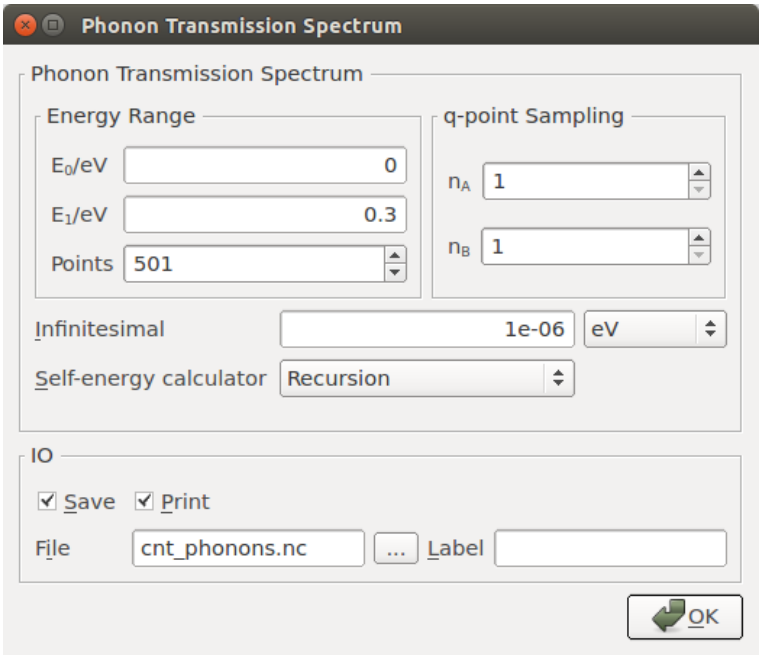
•Calculator: Select the ATK-ForceField calculator and the Tersoff_C_2010 potential.

•DynamicalMatrix: Set Repetition to Custom and leave the repetitions along A, B, and C at the default value 1.

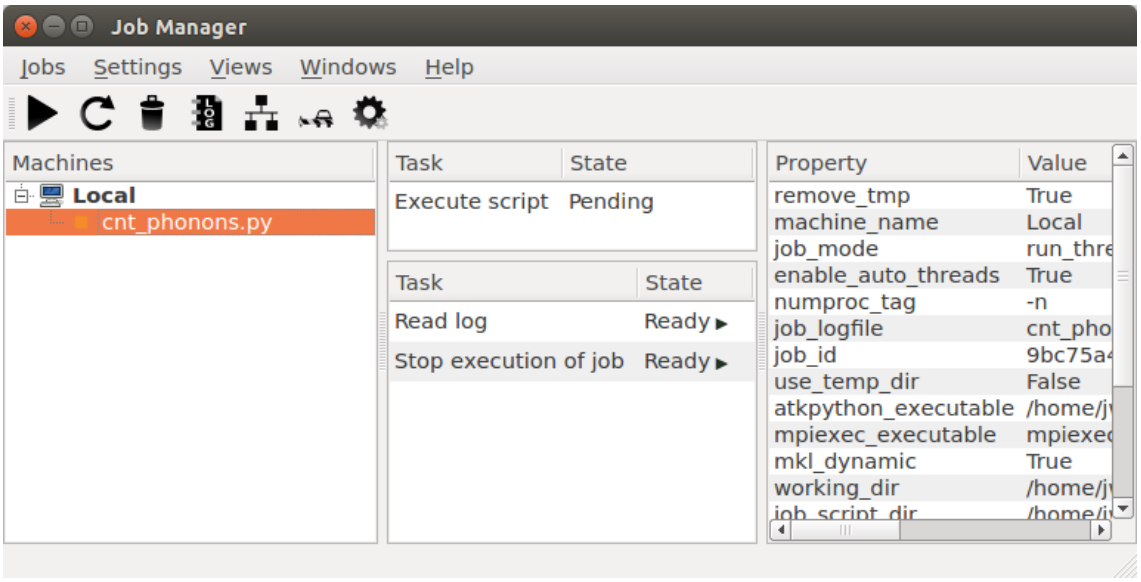
•PhononTransmissionSpectrum: Set the energy range to 0-0.3 eV and the number of points to 501. Leave the q-point sampling at defaults.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.





Run the calculation by sending it to the **Job Manager** . When prompted, save the script as `cnt_phonons.py`, select the default Local machine, and start the script by clicking .



Adding ^{14}C impurity atoms

Once the calculation for the undoped CNT has finished, it is time to introduce the ^{14}C dopants. The predefined elements in QuantumATK are isotope averages, so the mass of carbon in QuantumATK is 12.0107 amu. In order to create ^{14}C, all you need to do is a bit of simple Python scripting: *Implement a new element by writing a derived Python class with a different mass, and add it to the list of available elements*. Do it like this:

- Open your Python script (`cnt_phonons.py`) in the **Editor** , e.g. using drag and drop.
- Find the place where the `device_configuration` is defined (lines 447-450). Insert the following lines of code immediately after:

```
# Define a new class where the C atom has mass 14.00
class C14(Carbon):
    @staticmethod
    def atomicMass():
        return 14.00*Units.atomic_mass_unit

PeriodicTable.ALL_ELEMENTS.append(C14)

# Replace all atoms tagged "C14" with carbon-14
for index in device_configuration.indicesFromTags("C14"):
    device_configuration.elements()[index] = C14
```

- Change the output filename to `cntc14_phonons.hdf5` everywhere in the script (3 places).
- Save the script as `cntc14_phonons.py` and execute it using the **Job Manager**.

Transmission spectra

The QuantumATK **LabFloor** should now contain the calculated results for the doped and undoped CNT:

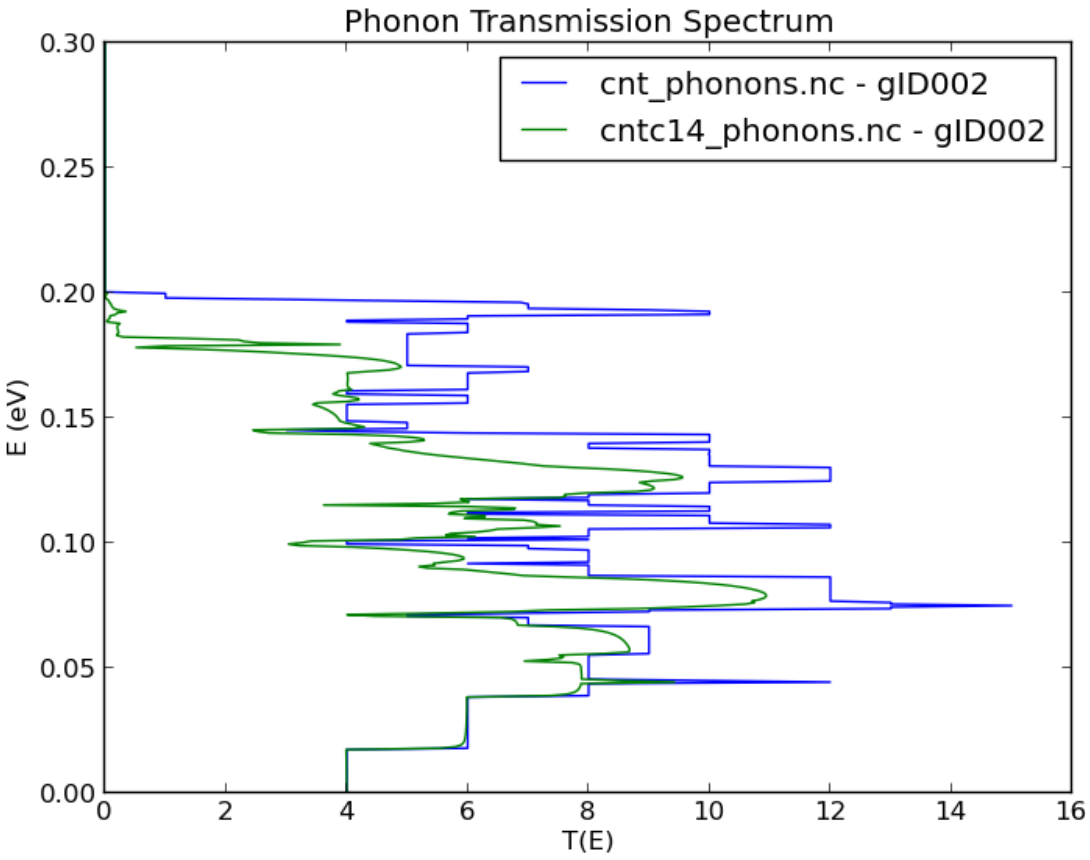
- DynamicalMatrix

$$D_{ij}$$

- PhononTransmissionSpectrum

$$T(\mathbf{q}, \omega)$$

Highlight the transmission spectra and use the **Transmission Analyzer** plugin to display the results. If you select both of them at the same time (holding down Ctrl), you can plot both in the same figure by using the **Compare Data** plugin.



The two phonon transmission spectra in the above figure (blue: undoped CNT, green: isotope doped CNT) are significantly different. The difference is caused by phonon scattering by the ring of ^{14}C impurities. You can see that at low energy (long wavelength) the spectrum is unchanged, while the main difference is at higher energies (shorter wavelength).

Thermal conductance

You can also calculate the thermal conductance of phonons, κ_{ph}, using the **Thermoelectric Coefficients** plugin, situated in the right-hand side of the LabFloor. Select again the PhononTransmissionSpectrum elements and click calculate on the Thermoelectric Coefficients plugin. As expected, κ_{ph} of the isotopically doped CNT is lower than for the CNT with a natural isotope distribution. The difference is small at 300 K, but it increases with temperature. The same effect was reported in[Page 949, 4], where the CNT was also doped by ^{14}C, but in a cluster arrangement.

It is interesting to construct a plot similar to Figure 11(b) in[Page 949, 4]. You can do this by applying a bit of Python scripting. Save the Python code below as `plot_thermal_conductance.py` or simply download the script: `plot_thermal_conductance.py`. If you wish to understand how the script works, try to have a look at the entry about the PhononTransmissionSpectrum object in the *PhononTransmissionSpectrum*.

```
# Read the phonon transmission spectrum from file cnt_phonons.hdf5
phonon_transmission_spectrum = nlread("cnt_phonons.hdf5",PhononTransmissionSpectrum)[0]
# Define a temperature range
temp = numpy.linspace(10,2000,21)
# Calculate the phonon thermal conductance at each temperature
conductances = []
for t in temp:
    conductance = phonon_transmission_spectrum.thermalConductance(
        phonon_temperature=[t]*Kelvin)
    conductance = conductance*10**9
    conductances.append(conductance)

# Now read the phonon transmission spectrum from file cntc14_phonons.hdf5 ...
phonon_transmission_spectrum_c14 = nlread("cntc14_phonons.hdf5",
↪PhononTransmissionSpectrum)[0]
# ... and calculate the relative phonon thermal conductance
conductances_c14 = []
for t in temp:
    conductance_c14 = phonon_transmission_spectrum_c14.thermalConductance(
        phonon_temperature=[t]*Kelvin)
    conductance_c14 = conductance_c14*10**9
    conductances_c14.append(conductance_c14)

# Plot the phonon thermal conductances as function of temperature
import pylab
pylab.figure()
pylab.plot(temp,conductances,label='CNT',linewidth=2.0)
pylab.plot(temp,conductances_c14,label='14C-CNT',linewidth=2.0)
pylab.xlabel("Temperature (K)")
pylab.ylabel("Phonon Thermal Conductance (nW/K)")
pylab.legend(loc="lower right")
pylab.show()
```

Run the script in the directory where your `.hdf5` files are stored:

```
$ atkpython plot_thermal_conductance.py
```

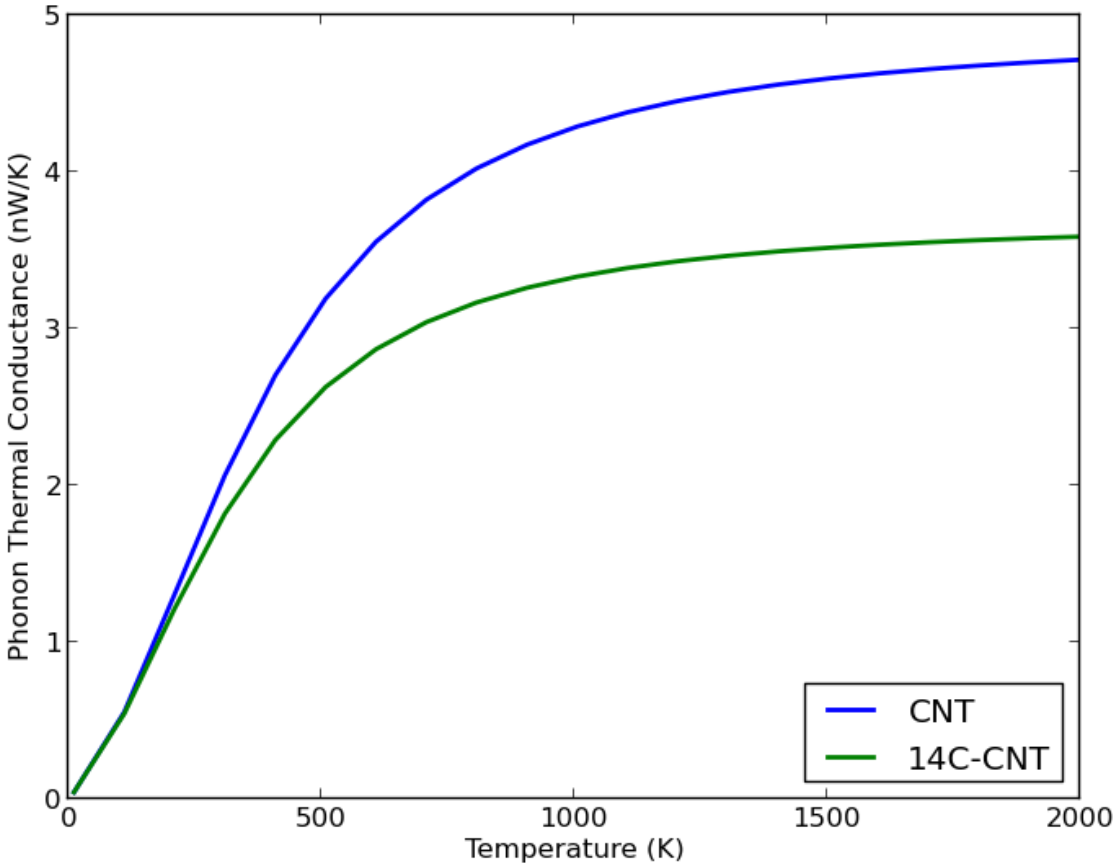


Fig. 3.75: Phonon thermal conductance of a pure CNT and of an ^{14}C isotope doped CNT. The thermal conductance levels out with increasing temperature. The ^{14}C doping clearly inhibits the CNT thermal conductance.

Electron transmission

The thermoelectric figure of merit, *ZT*, has both phononic and electronic contributions. Therefore, even though isotopic doping does not affect the electron transport properties, it is still relevant to calculate the electronic transmission spectrum for the CNT.

From the **Builder** send again your CNT device configuration to the **Script Generator** and add two blocks:

- New Calculator
- *Analysis* → *TransmissionSpectrum*

Change the output file to `cnt_electrons.hdf5` and edit the calculation parameters:

•Calculator: Select the ATK-SE: Slater-Koster calculator and choose the Hancock.C ppPi basis set.

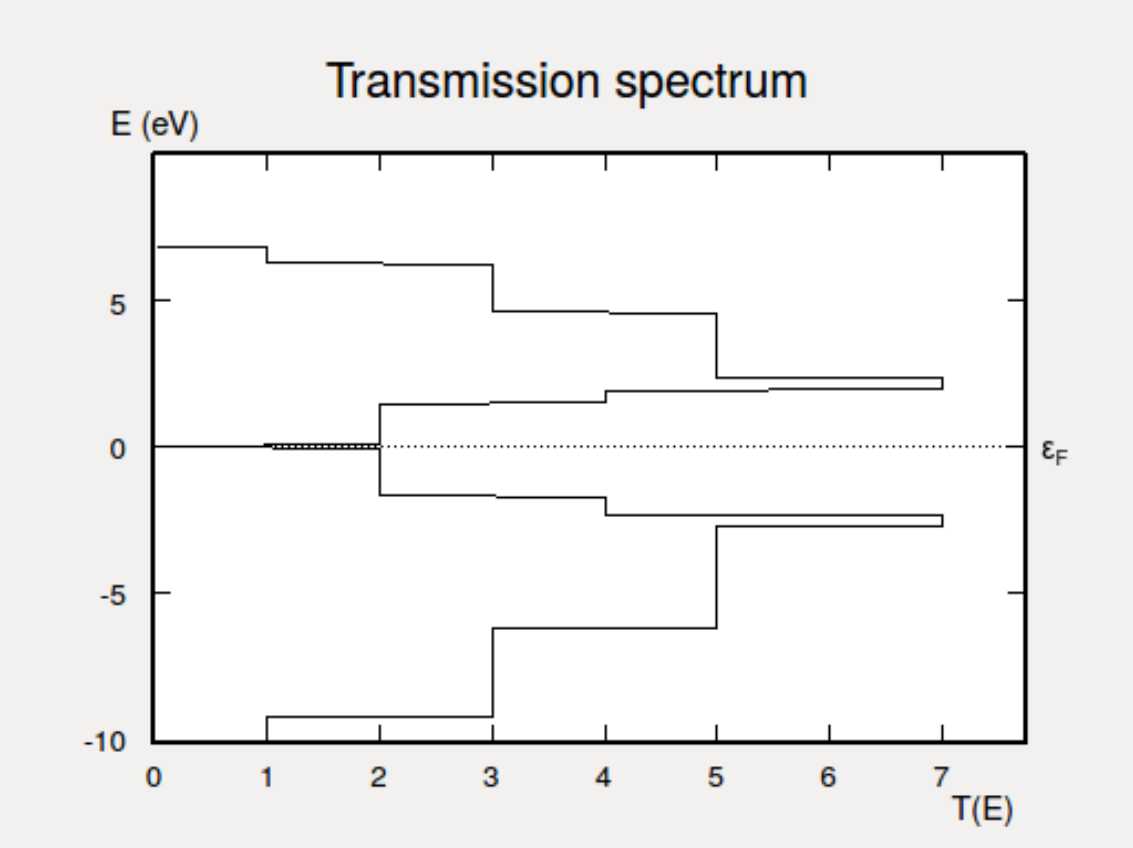
•TransmissionSpectrum: Set the energy range to -10 to 10 eV and the number of points to 501.

Run the script using the **Job Manager** .

In the LabFloor you will now find the **TransmissionSpectrum** object, which you can plot using the **2D Plot** plugin.

- TransmissionSpectrum

$T(\mathbf{k}, \epsilon)$

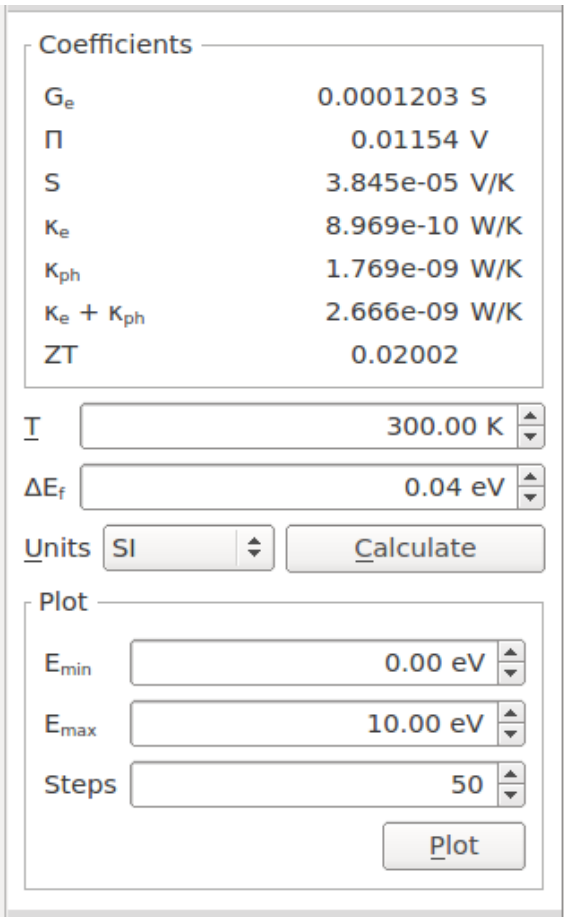


The electronic transmission is clearly non-zero in the vicinity of the Fermi level. However, if you zoom in, you will see that the transmission is zero exactly at the Fermi level. This is also clear if you study the transmission data using the **Text Representation** plugin.

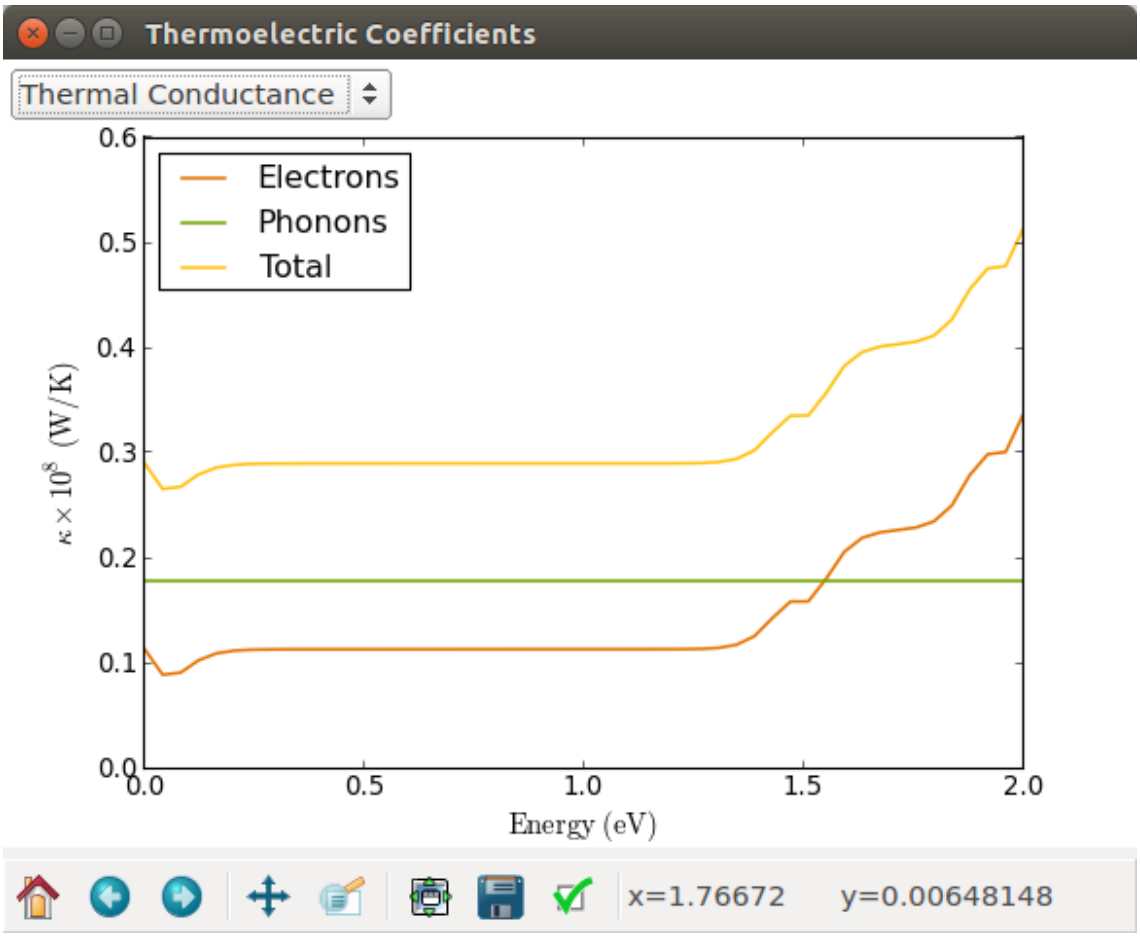
Thermoelectric transport properties

Now you have all the ingredients for calculating thermoelectric transport properties such as conductance, Peltier coefficient, Seebeck coefficient, and heat transport coefficients of electrons and phonons, all of which are ingredients in the thermoelectric figure of merit, *ZT*.

Select from the LabFloor both the **PhononTransmissionSpectrum** AND the **TransmissionSpectrum** objects (by holding down Ctrl while clicking them). Open the **Transmission Coefficient** plugin and set the Fermi level shift to 0.04 eV, corresponding to the band edge. Finally, click **Calculate** to compute all the transport coefficients, and in particular the value of *ZT* at room temperature.



By clicking **Plot**, you can also get an interactive window with plots of the different transport quantities as a function og energy.



Compare *ZT* for doped and undoped CNTs

If you compare the two values of *ZT* for the two devices with and without ^{14}C doping, you will find that the effect of isotopic doping is to slightly increase *ZT*. The effect is indeed very small.

One way to illustrate this result is by plotting *ZT* as a function of the energy for both CNTs in one figure. Once again, a bit of Python scripting will do the job. Use the following piece of Python code, or simply download the script: `plot_zt.py`.

```
from AddOns.TransportCoefficients.Utilities import evaluateMoments
from NL.CommonConcepts import PhysicalQuantity as Units

# Set input parameters
e_min = 0.0
e_max = 2.0
n_steps = 400
temperature = 300.00*Kelvin
energies = numpy.linspace(e_min, e_max, n_steps) * eV

# Read phonon transmission spectrum from file cnt_phonons.nc
# and get the phonon conductance at the specified temperature
phonon_transmission_spectrum = nlread('cnt_phonons.nc', PhononTransmissionSpectrum)[0]
conductance_phonons = phonon_transmission_spectrum.thermalConductance(phonon_
```

(continues on next page)

(continued from previous page)

```
↪temperature=temperature)

# Read phonon transmission spectrum from file cntc14_phonons.nc
# and get the phonon conductance at the specified temperature
phonon_transmission_spectrum_c14 = nlread('cntc14_phonons.nc',␣
↪PhononTransmissionSpectrum)[0]
conductance_phonons_c14 = phonon_transmission_spectrum_c14.thermalConductance(phonon_
↪temperature=temperature)

# Read electron transmission spectrum from file cnt_electrons.nc
electron_transmission_spectrum = nlread("cnt_electrons.nc",TransmissionSpectrum)[0]

# define lists for ZT data
zzt = []
zzt_c14 = []

# Calculate the ZTs at each energy
for energy in energies:
    # Calculate moments of the electron transmission spectrum
    k0, k1, k2 = evaluateMoments(electron_transmission_spectrum, temperature, energy)
    # Calculate transport properties.
    conductance       = k0*Units.e**2
    peltier           = k1/(k0*Units.e)
    seebeck           = k1/(k0*Units.e*temperature)
    thermal_electrons = (k2*k0-k1*k1)/(temperature*k0)
    total_thermal     = thermal_electrons + conductance_phonons
    total_thermal_c14 = thermal_electrons + conductance_phonons_c14
    zt                = conductance*seebeck**2*temperature/total_thermal
    zt_c14            = conductance*seebeck**2*temperature/total_thermal_c14
    zzt.append(zt)
    zzt_c14.append(zt_c14)

# Plot the ZT as function of energy
zzt = numpy.asarray(zzt)*1e3
zzt_c14 = numpy.asarray(zzt_c14)*1e3
import pylab
pylab.plot(energies,zzt,'-',label='CNT',linewidth=1.0,color='b')
pylab.plot(energies,zzt_c14,'--',label='14C-CNT',linewidth=1.0,color='r')
pylab.xlabel('Energy (eV)')
pylab.ylabel('ZT x 1000')
pylab.legend(loc='upper center')
pylab.show()
```

The script produces the figure shown above. It is clear that isotopic doping increases the value of *ZT*, though only marginably.

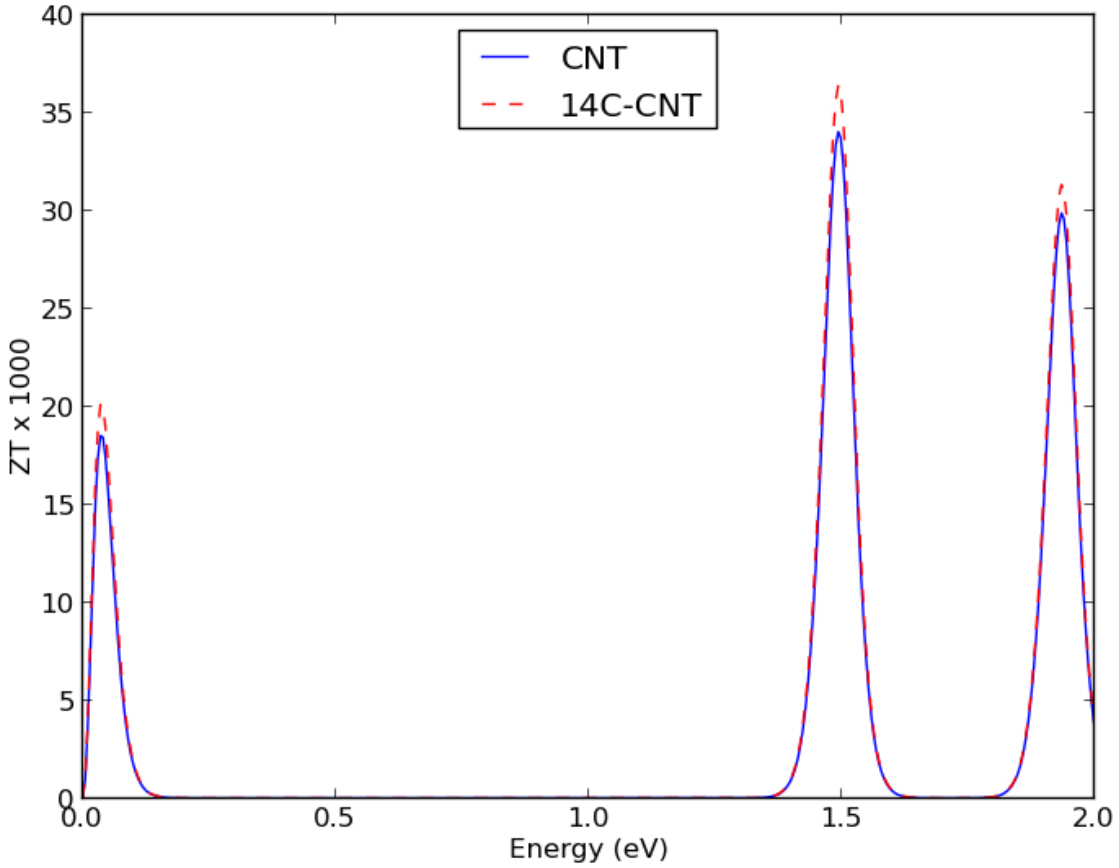


Fig. 3.76: *ZT* at 300 K for a CNT with a natural isotopic distribution (blue line) and for a CNT with a full ring of ^{14}C impurities (red broken line).

References

3.5.8 Graphene nanoribbon device: Electric properties

The purpose of this tutorial is to show you how the **ATK-SE** package can be used to calculate the electric properties of a **nano-scale transistor device**. You will compute quantities like the electron transmission spectrum, conductance, I–V curve, and electron thermal transport.

Attention: Graphene nanoribbon device with a gate and a dielectric spacer

The device system considered in this tutorial consists of a z-shaped graphene nanoribbon on top of a dielectric and controlled by a metallic gate, as illustrated below. It forms a metal–semiconductor–metal junction. By applying a gate potential across the central region, the system can function as a field effect transistor (FET).

Attention: This tutorial assumes that you have already built the small nanoribbon transistor as described here: *Small nanoribbon transistor*. If you have not, please do so before continuing with this tutorial.

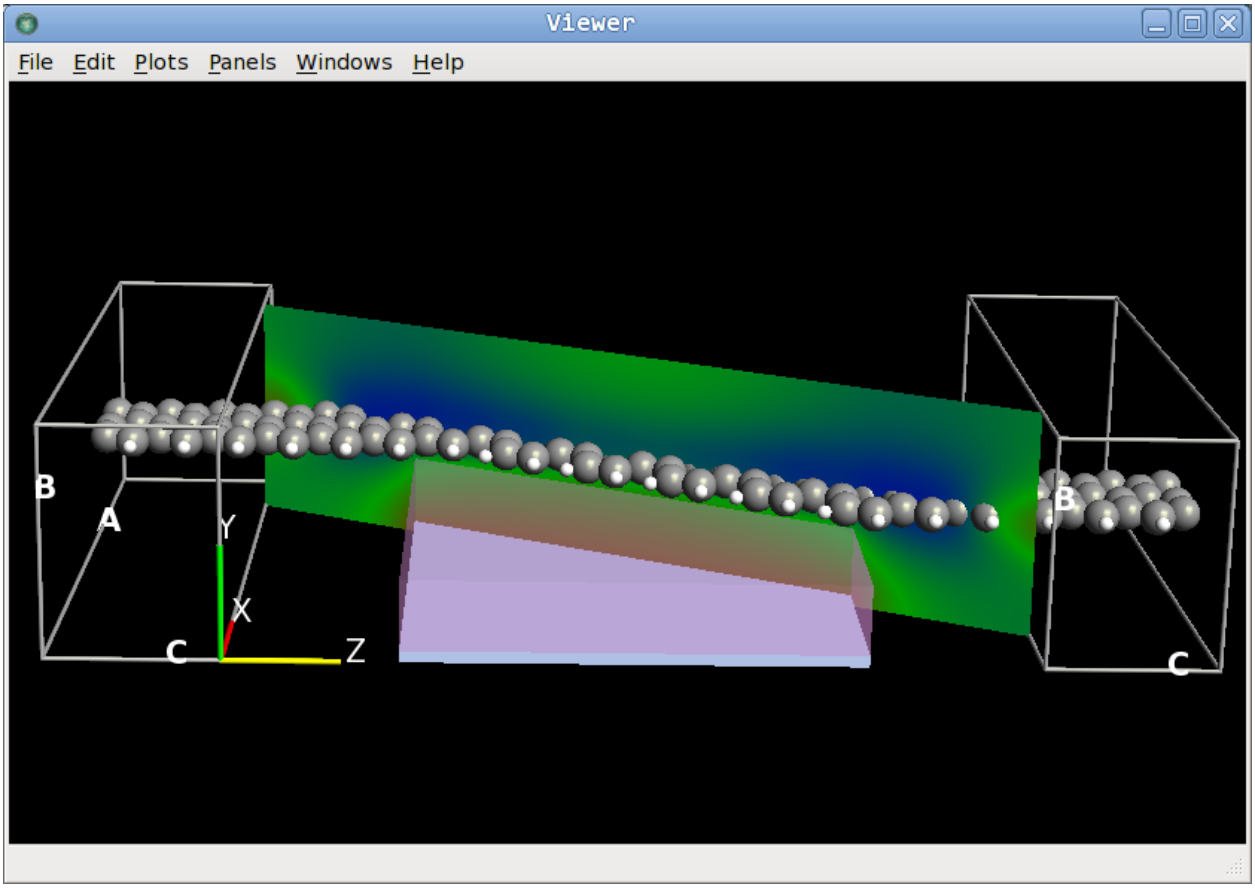


Fig. 3.77: The device system considered in this tutorial consists of a z-shaped graphene nano-ribbon, on top of a dielectric and controlled by a metallic gate. The contour plot illustrates the electrostatic potential through the system.

The device is illustrated in the figure above. It consists of three regions; a central region and two electrodes. You will calculate the following properties of the system.

•Transmission spectrum

•Temperature dependent conductance

•Temperature induced current

•Conductance as function of gate potential and temperature

•Current as function of bias, gate potential and temperature

Get started

Create a new QuantumATK project and add the already built nanoribbon device to the project:

•Open the Builder and use *Add* → *From Files* to find the saved `device_configuration` (in QuantumATK Python or HDF5 format) and add it to the Stash.

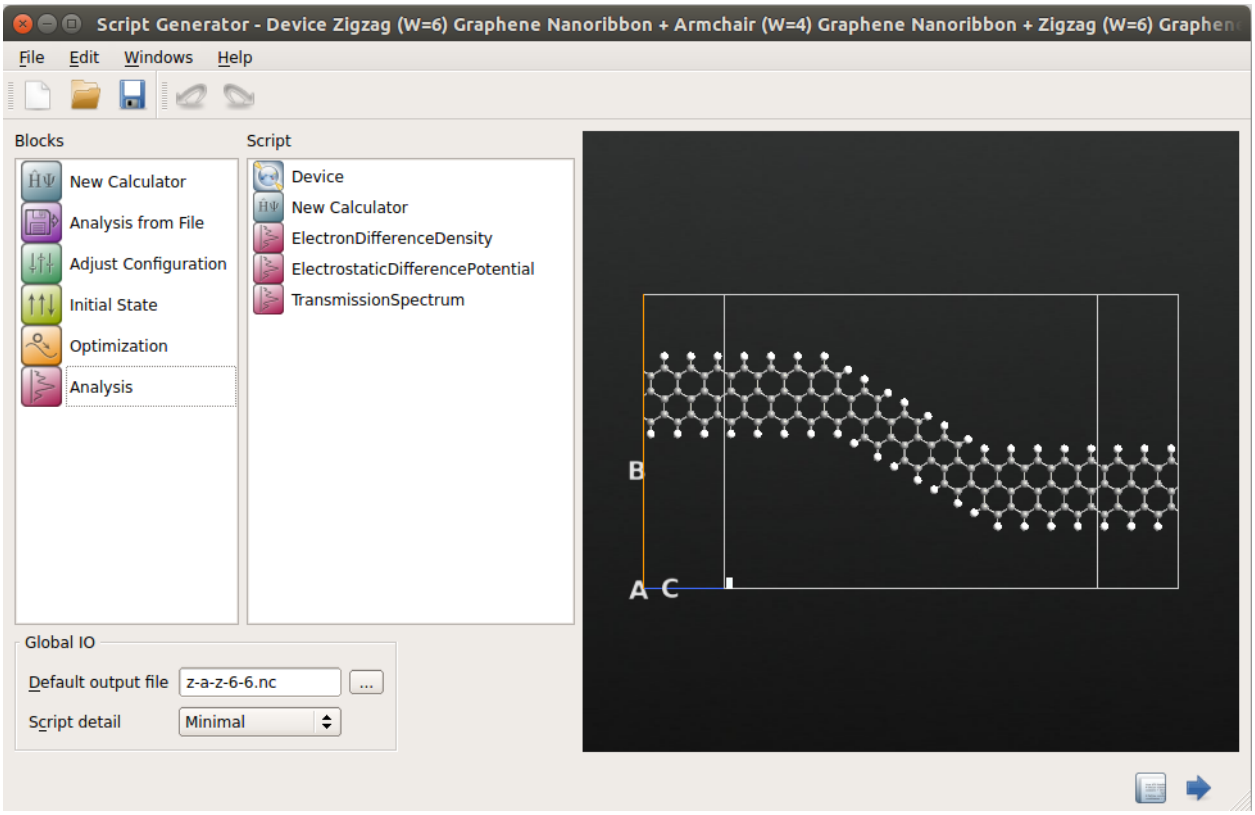
Electron transmission spectrum

Send the configuration from the Builder to the **Scripter** by using the icon in the lower right-hand corner of the Builder window.

Then set up the semi-empirical electron transmission calculation. Add the following blocks to the script:

- **New Calculator**
- *Analysis* → **ElectronDifferenceDensity**
- *Analysis* → **ElectrostaticDifferencePotential**
- *Analysis* → **TransmissionSpectrum**

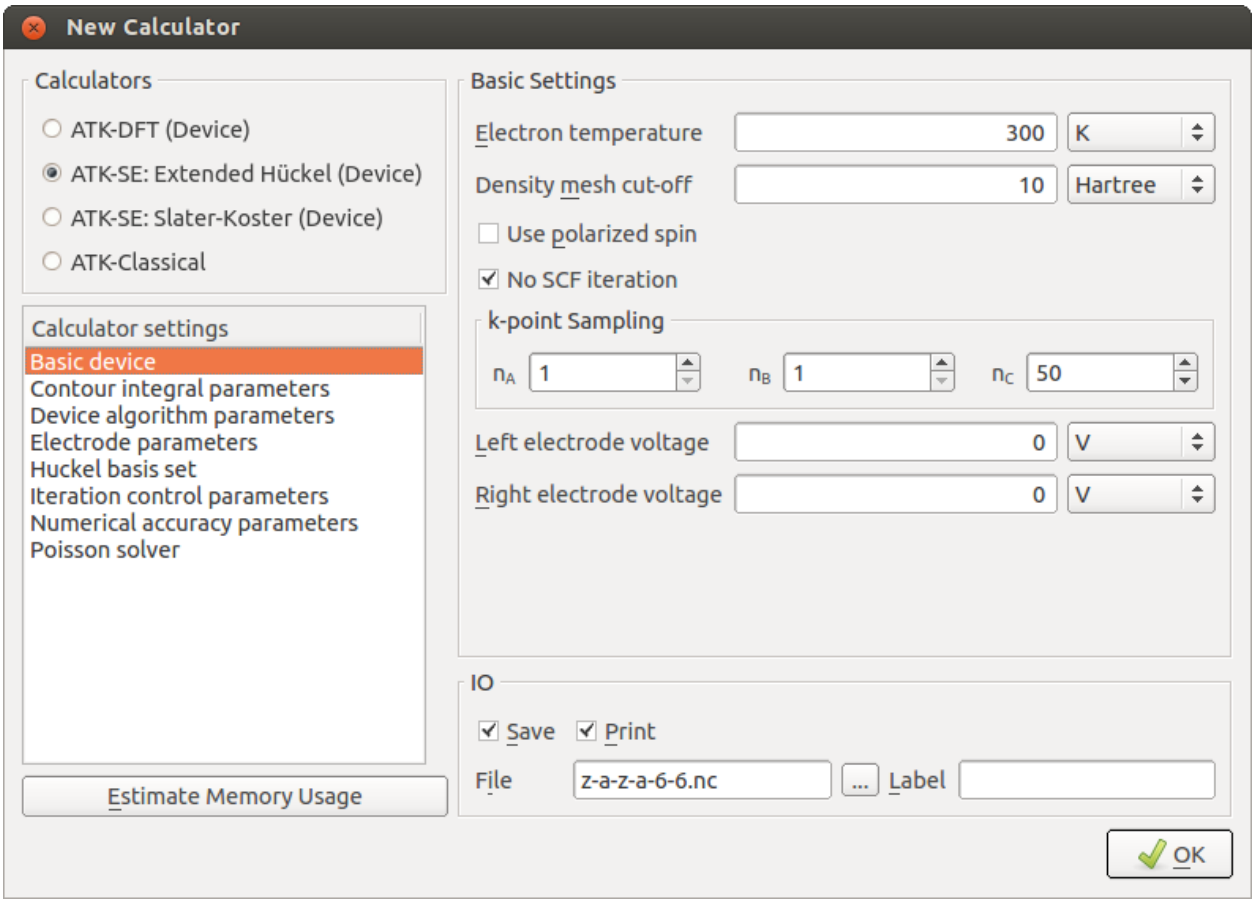
Then change the output filename to `z-a-z-6-6.hdf5`.



Calculator settings

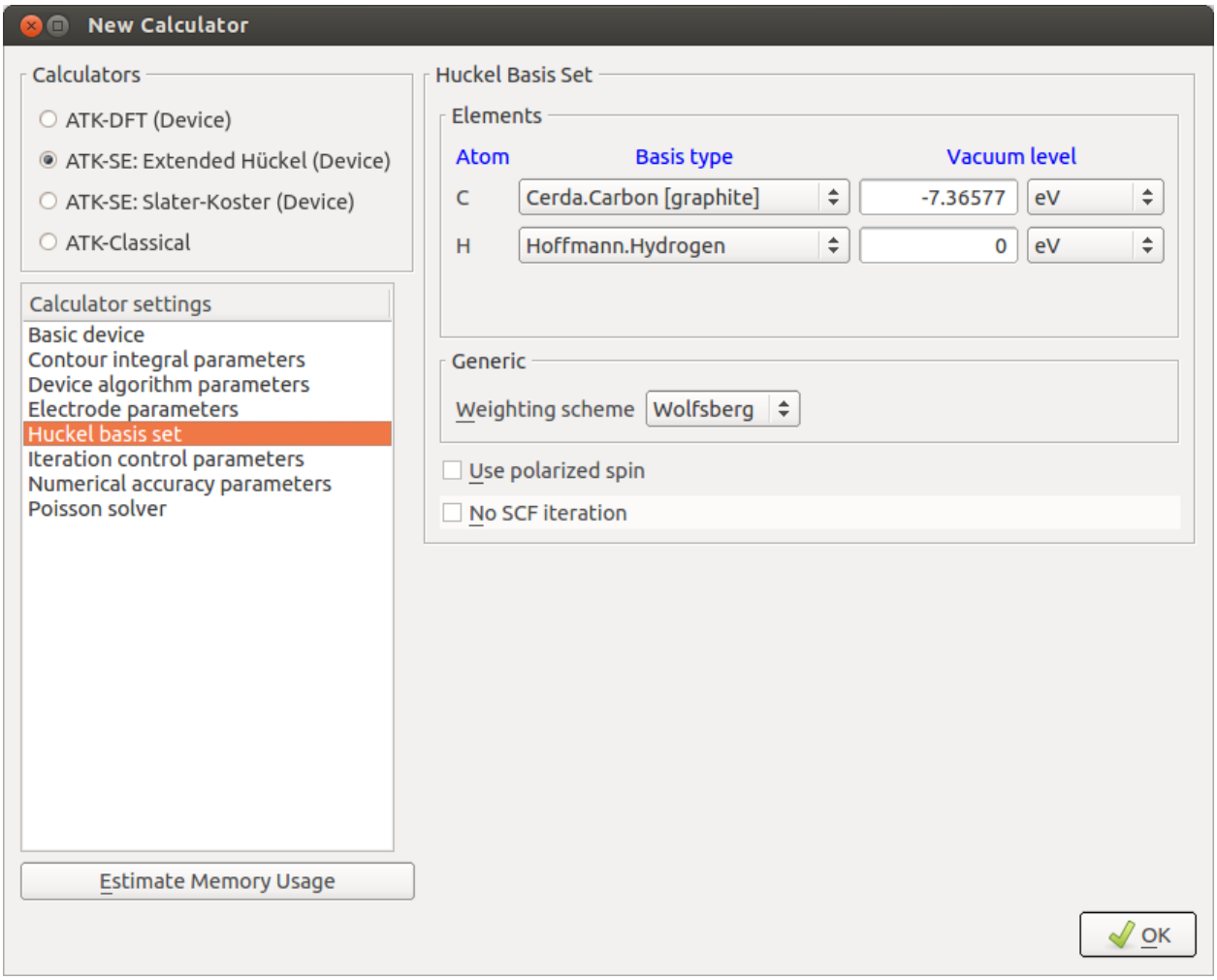
Open the calculator block and edit the settings:

- Select the *ATK-SE: Extended Hückel (Device) calculator*, and lower the number of k-points in the C direction to 50, which should be sufficient for this calculation.
- Uncheck the No SCF iteration box, to perform a self consistent calculation.



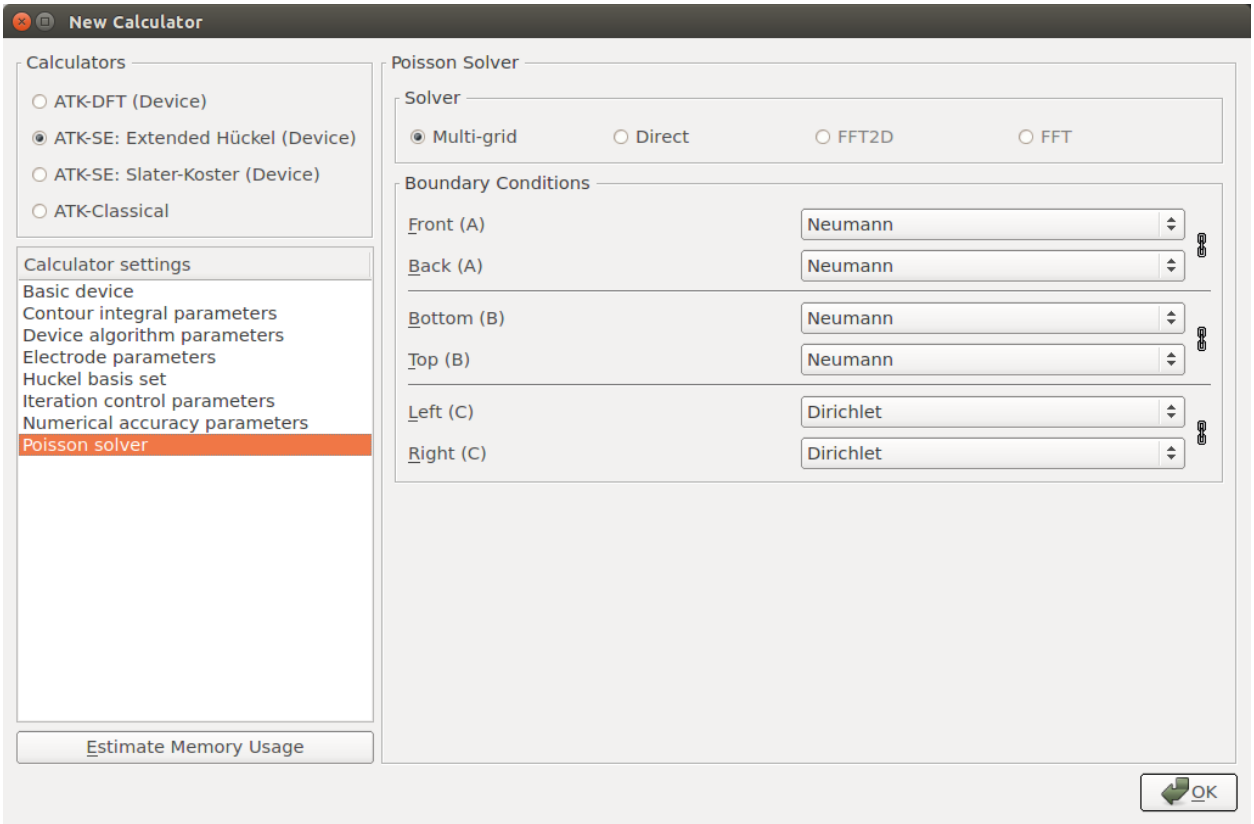
•The next step is to change the basis set for the calculation to use parameters specifically optimized for describing carbon structures[1]. To this end, enter the Huckel basis set menu and select the *Cerda.Carbon [Graphite]* and *Hoffmann.Hydrogen* basis sets.

[1] J.ăCerda and F.ăSoria. Accurate and transferable extended Hückel-type tight-binding parameters. *Phys. Rev. B*, 61:7965–7971, Mar 2000. doi:10.1103/PhysRevB.61.7965.



Note: The default vacuum level for each basis set is chosen such that the carbon and hydrogen parameters they can be used together, even though they are fitted to different reference systems.

•Make sure to use the Multigrid Poisson solver and Neumann boundary conditions along the A and B directions, which are perpendicular to the transport direction. See the image below.



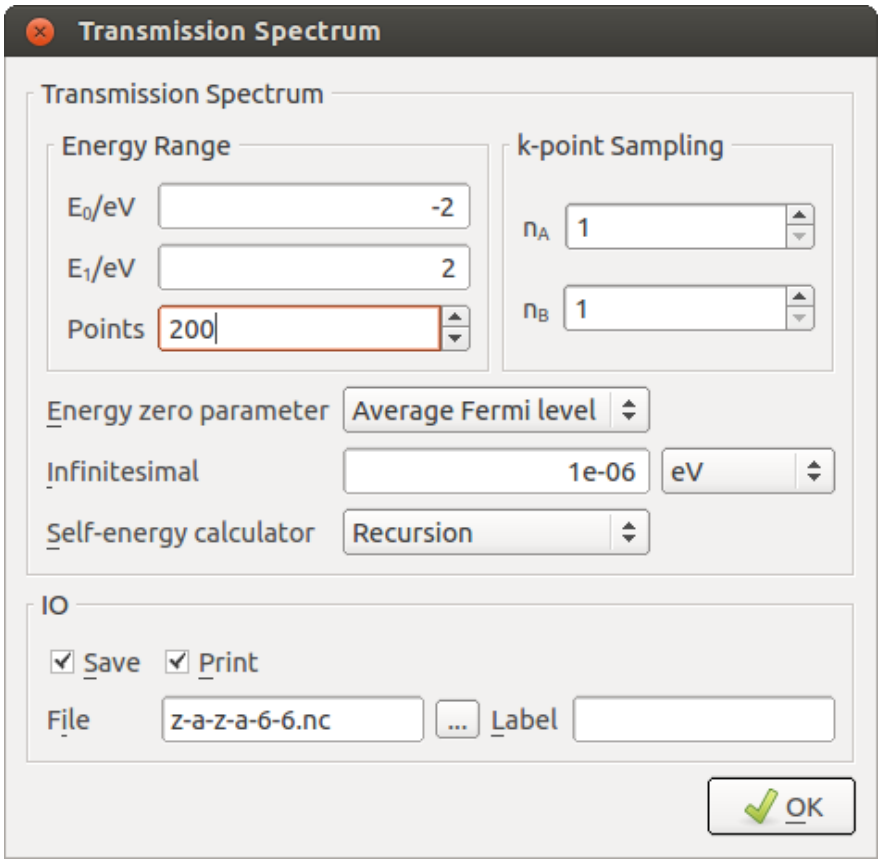
Attention: It is import to use **Neumann boundary conditions** along **non-periodic** directions perpendicular to C, when your device has a metallic gate. Why is this?

The Neumann boundary condition imposes the constraint that the derivative of the electrostatic potential should be zero on that boundary. Put diffferently; the electrostatic potential should be constant at the boundary (but make take on any value). This is the appropriate boundary condition when the device has a metallic gate!

Setup the analysis blocks

•Keep the default settings in the **ElectronDifferenceDensity** and **ElectroStaticDifferencePotential** blocks.

•In **TransmissionSpectrum**, increase the number of sampling points to 200.



You have now completed setting up the calculation. Save the script in the project directory as `z-a-z-6-6.py`.

Run the calculation

Execute the calculation by sending the script to the **Job Manager**, again using the botton. You may be asked to save the script again. Click the botton to start the job. It will take 5 to 15 minutes, depending on your computer.

Note: What is actually happening during the calculation?

The job will first perform a self-consistent calculation for each of the electrodes. The self-consistent electrode Hartree potential is then used as a boundary condition in a self-consistent calculation for the central region of the device. Finally, the transmission spectrum and other analysis objects of the structure are calculated.

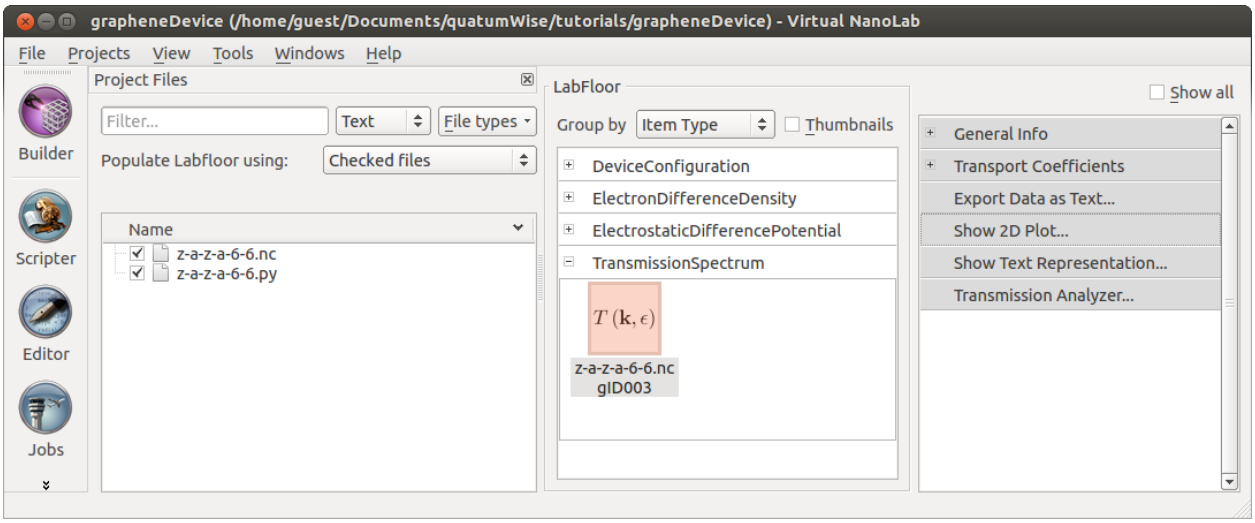
Tip: If you accidentally closed down the Scripter window, or even the QuantumATK main window, your work is not lost! You can simply drag-and-drop the saved Python scipt onto the Scripter to open it for editing. Similarly, you can directly send any saved script to the Job Manager.

Examine the results

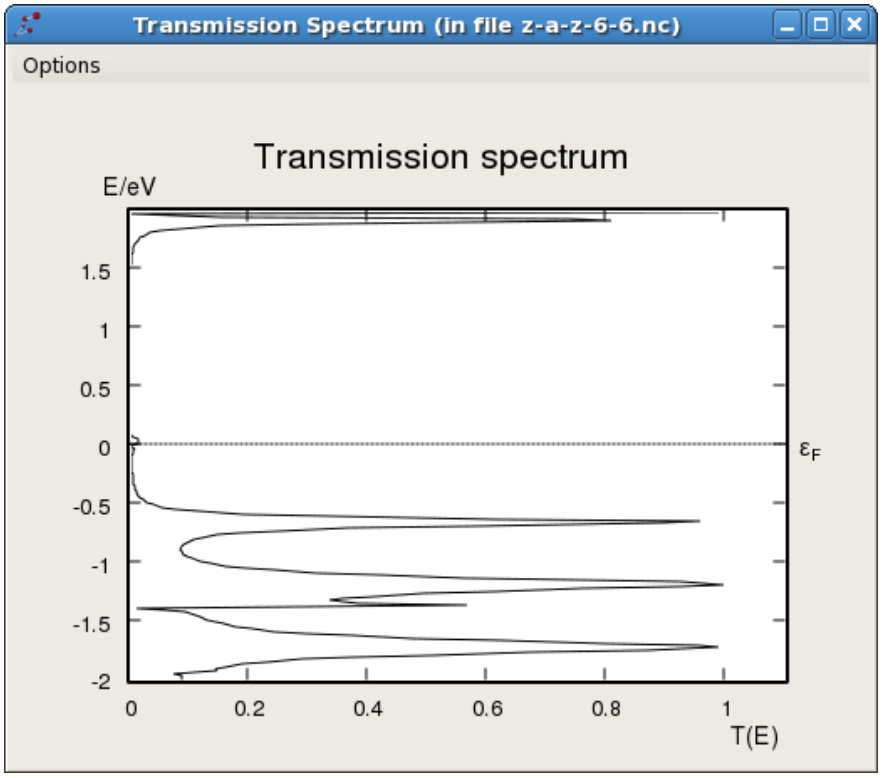
Transmission Spectrum

Switch to the QuantumATK main window, where you will note that the HDF5 file `z-a-z-6-6.hdf5` has been generated. The contents of the file should already be displayed on the **LabFloor**. It contains the **DeviceConfiguration** and the requested analysis objects:

- **ElectronDifferenceDensity** $\delta n(\mathbf{r})$
- **ElectrostaticDifferencePotential** $\delta V_{\mathbf{E}}(\mathbf{r})$
- **TransmissionSpectrum** $T(\mathbf{k}, \epsilon)$



Select the TransmissionSpectrum object and use the **2D Plot** plugin to visualize the calculated transmission spectrum.



Note: The transmission spectrum has a low value in the energy range [-0.5, 1.5] eV, corresponding to the energy window within the band gap of the central semi-conducting armchair-edge ribbon. The asymmetric position of the electrode Fermi levels relative to the band edges, i.e. the shift of the valence band edge of the central region towards the electrode Fermi levels, is due to the applied gate potential of -1V.

Electrostatic potential

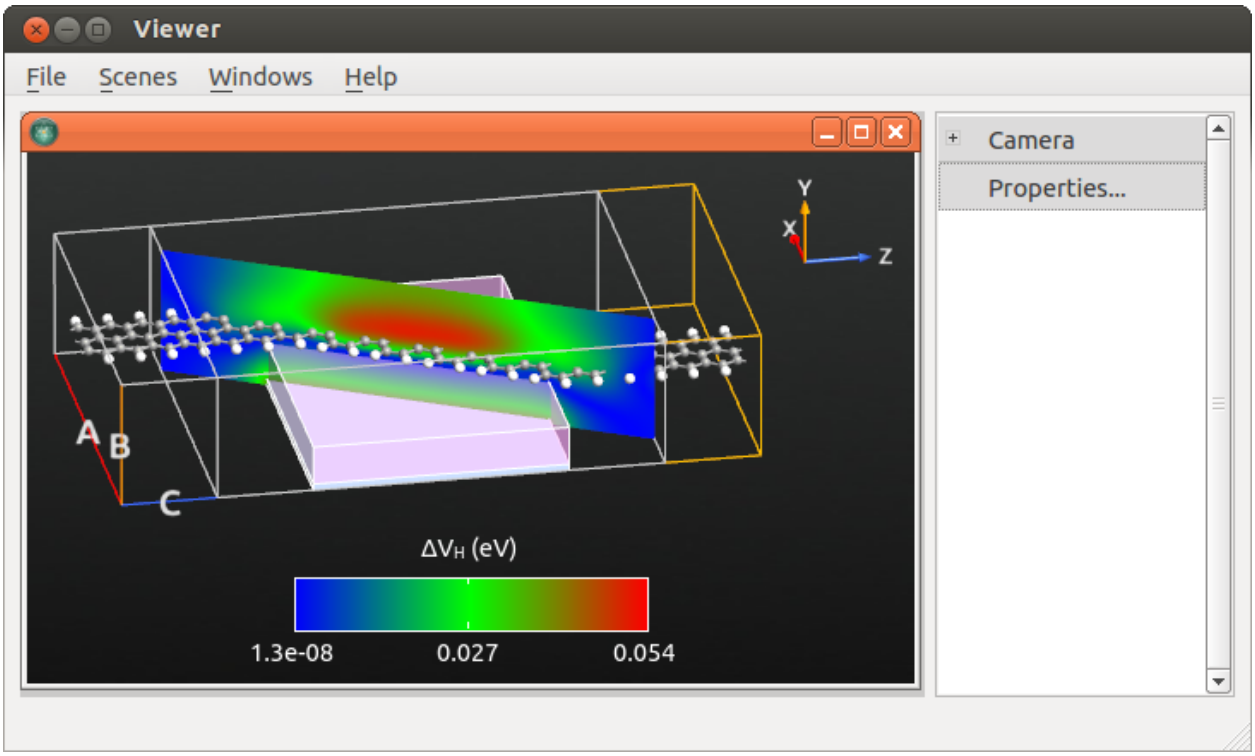
To visualize the effect of the gate potential, select the **ElectrostaticDifferencePotential** object and click the **Viewer** plugin. Select Contour Plot from the dialogue that pops up.

To make the plot more interesting, add a 3D rendering of the graphene junction by dropping the **DeviceConfiguration** from the LabFloor onto the open scene in the Viewer. Then rotate the contour plot:

•Click **Properties** and select CutPlanes from the top.

•Change the rotation angle, θ_y, to 151ř and set the color map to RGB.

•Finally, rotate the camera of the 3D view with the mouse to obtain a plot similar to the image shown below.



If you wish to **export** a copy of the contour plot, right-click in the Viewer window, and choose Export (or use Ctrl+E). You can choose to export as PNG og JPG graphics. The image is now stored on disk and can be used for further processing in any other third party application of your choice.

Effect of the Gate Potential

You will now investigate the effect of changing the gate potential. In the previous section, the gate potential was fixed at -1 V. You will now vary the potential in the range of [-2,2] V and calculate the transmission spectrum at each gate potential.

Gate potential loop

For each value of the gate potential you should perform a new self-consistent calculation to obtain the corresponding transmission spectrum. Such a loop is conveniently done by the following QuantumATK script. You can download it here: `gate_loop.py`

```
#read in the old configuration
device_configuration = nlread("z-a-z-6-6.nc",DeviceConfiguration)[0]
calculator = device_configuration.calculator()
metallic_region0 = device_configuration.metallicRegions()[0]
# Define gate_voltages
gate_voltage_list=numpy.linspace(-2.0,2.0,17)*Volt
for gate_voltage in gate_voltage_list:
    # set the gate potential
    device_configuration.setMetallicRegions(
        [metallic_region0(value = gate_voltage)] )
    # make a copy of the calculator and attach it to the configuration
    # restart from the previous scf state
```

(continues on next page)

(continued from previous page)

```
    device_configuration.setCalculator(calculator(),
        initial_state=device_configuration)
    #Analysis
    filename= 'gatescan-6-6.nc'
    electron_density = ElectronDifferenceDensity(device_configuration)
    nlsave(filename, electron_density,object_id='dens'+str(gate_voltage))
    electrostatic_potential = ElectrostaticDifferencePotential(device_configuration)
    nlsave(filename, electrostatic_potential, object_id='pot'+str(gate_voltage))
    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,200)*eV,
        )
    nlsave(filename, transmission_spectrum,object_id='trans'+str(gate_voltage))
    nlprint(transmission_spectrum)
```

The script implements the following workflow:

1. Read the self-consistent calculation obtained in the previous chapter and use this as the starting point for the calculations:

```
#read in the old configuration
device_configuration = nlread("z-a-z-6-6.nc",DeviceConfiguration)[0]
calculator = device_configuration.calculator()
metallic_region0 = device_configuration.metallicRegions()[0]
```

2.Set up a list of gate voltage values [-2.0, -1.75, , 2.0]*Volt:

```
# Define gate_voltages
gate_voltage_list=numpy.linspace(-2.0,2.0,17)*Volt
```

3. Loop through the gate voltage list, and for each value modify the value of the gate voltage in the metallic region:

```
for gate_voltage in gate_voltage_list:
    # set the gate potential
    device_configuration.setMetallicRegions(
        [metallic_region0(value = gate_voltage)] )
```

4.Initialize the self-consistent state of the device configuration by calling the `calculator()` method.

```
    # make a copy of the calculator and attach it to the configuration
    # restart from the previous scf state
    device_configuration.setCalculator(calculator(),
        initial_state=device_configuration)
```

This method will make a copy of the previous Huckel calculator using the same settings. Since a new calculator is attached, a self-consistent calculation will automatically be performed the next time a property of the system is required:

5.Finally, the same analysis objects as in the previous chapter are calculated and save them into a HDF5 file. Note that for each analysis object a specific **object_id** is specified. This will make it easier to retrieve the correct entries from the HDF5 file:

```
    #Analysis
    filename= 'gatescan-6-6.nc'
```

(continues on next page)

(continued from previous page)

```
    electron_density = ElectronDifferenceDensity(device_configuration)
    nlsave(filename, electron_density,object_id='dens'+str(gate_voltage))
    electrostatic_potential = ElectrostaticDifferencePotential(device_configuration)
    nlsave(filename, electrostatic_potential, object_id='pot'+str(gate_voltage))
    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,200)*eV,
        )
    nlsave(filename, transmission_spectrum,object_id='trans'+str(gate_voltage))
    nlprint(transmission_spectrum)
```

Run the script using the **Job Manager** or execute it from a terminal:

```
$ atkpython gatescan-6-6.py > gatescan-6-6.out
```

The script will take a while to complete, since 17 self-consistent calculations are needed. The total execution time may be between 2 and 8 hours, depending on your hardware.

Conductance as a function of gate potential

The script in the previous section produced the data file `gatescan-6-6.hdf5`. It should contain all the calculated transmission spectra. Try to inspect them using the **Transmission Analyzer** to see how the transmission spectrum changes with the gate potential.

Note: You will find that all the spectra look rather similar and that the main difference between the different spectra is a shift of the valence and conduction band edges relative to the Fermi level.

This is the main function of the metallic gate: It can be used to tune the band edge positions relative to the Fermi level, and thereby control the electric properties of the device!

You will now use the data to calculate the conductance of the junction as a function of the gate potential for different electrode temperatures. Use the following script, which is a slight modification of the script used in the previous chapter. You can also download it here: `conductance_diff_gate.py`.

```
#make list of relevant temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list of relevant gate voltages
gate_voltage_list=numpy.linspace(-2.0,2.0,17)*Volt
#make list to hold the conductance calculations
conductance_list=numpy.zeros(len(gate_voltage_list)*len(temperature_list))
conductance_list=conductance_list.reshape(len(gate_voltage_list),
                                          len(temperature_list))
#specify the filename for the netcdf data file
filename="gatescan-6-6.nc"
#loop through the gate voltages
for n in range(len(gate_voltage_list)):
    transmission_spectrum=nlread(filename,
        object_id="trans"+str(gate_voltage_list[n]))[0]
    #loop through the temperature list
    for i in range(len(temperature_list)):
```

(continues on next page)

(continued from previous page)

```
        conductance_list[n,i]=transmission_spectrum.conductance(
            electrode_temperatures=(temperature_list[i],temperature_list[i]))
#plot the conductance as function of gatevoltage
import pylab
pylab.figure()
# make curve for each temperature
for i in range(len(temperature_list)):
    pylab.semilogy(gate_voltage_list,conductance_list[:,i])
pylab.xlabel("Gate Voltage (V)")
pylab.ylabel("Conductance (S)")
pylab.show()
```

The script produces the following figure:

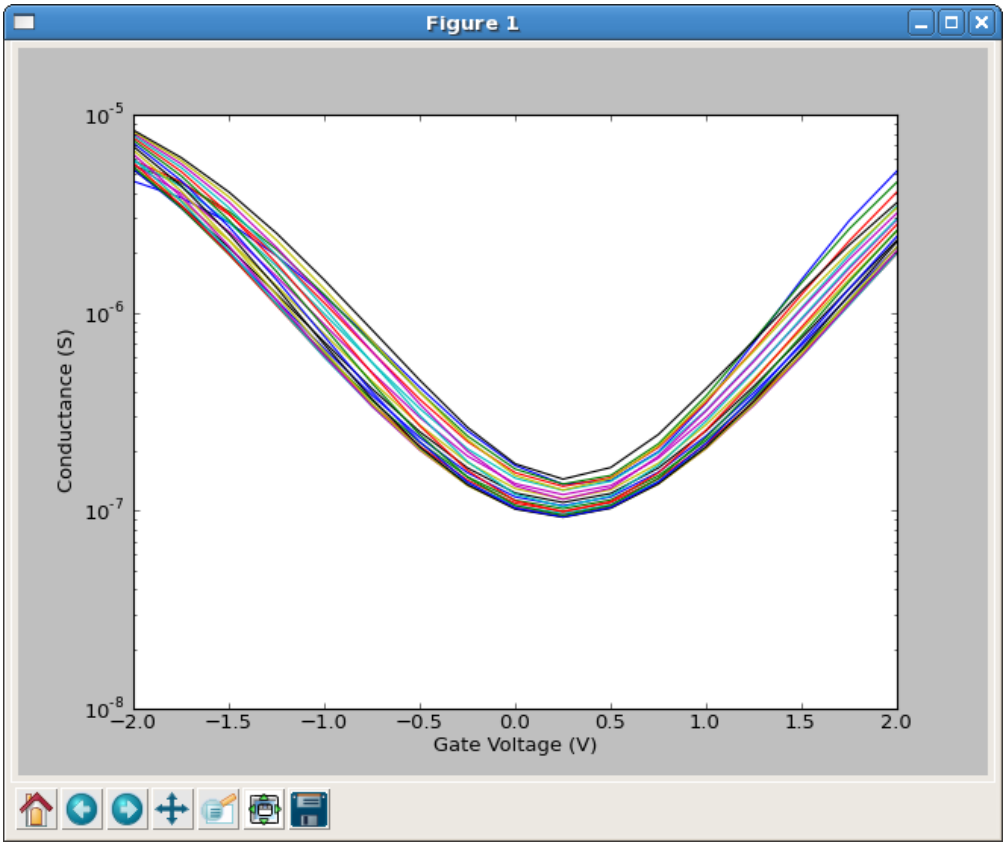


Fig. 3.78: Conductance as a function of the gate potential. Different curves are for different electron temperatures, namely 0, 100, 200, , 2000 Kelvin.

The conductance has a minimum for a gate potential of about 0.25 Volt. At this point, the Fermi level is positioned in the mid gap between the valence and conductance band of the central ribbon. Note that there is only a weak dependence of the conductance on the electron temperature, which illustrates that the transmission is dominated by electron tunnelling.

Tip: This tutorial relies heavily on scripting access to inputs and outputs of the **TransmissionSpectrum** object. For further information on all the options and methods available, see the ATK Reference Manual.

Also note that we use the pylab package to generate plots. This is readily available with `atkpython` and is very handy

for making plots with Python scripts.

Linear response calculations

You just saw that the transmission spectra for different gate potentials are rather similar and that the main difference is a shift of the band edges relative to the Fermi level. This suggests that you may avoid the self-consistent loop and simply shift the Fermi level rigidly and calculate all the transmission spectra using linear response theory.

The following script implements a linear response calculation of the conductance as function of the gate potential, assuming that the effect of the gate potential is a simple shift of the relative positions of the electrode Fermi levels within the transmission spectra. You can download it here: `linear_response.py`.

```
#make list of relevant temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list of relevant gate voltages
gate_voltage_list=numpy.linspace(-2.0,2.0,17)*Volt
#make list to hold the conductance calculations
conductance_list=numpy.zeros(len(gate_voltage_list)*len(temperature_list))
conductance_list=conductance_list.reshape(len(gate_voltage_list),
                                          len(temperature_list))
#read the reference transmission spectrum from the netcdf data file
gate_potential_ref = 0.*Volt
transmission_spectrum = nlread("gatescan-6-6.nc",
    object_id="trans"+str(gate_potential_ref))[0]
#loop through the gate voltages
for n in range(len(gate_voltage_list)):
    #loop through the temperature list
    for i in range(len(temperature_list)):
        conductance_list[n,i]=transmission_spectrum.conductance(
            electrode_temperatures=(temperature_list[i],temperature_list[i]),
            electrode_voltages=(gate_voltage_list[n]-gate_potential_ref,
                                gate_voltage_list[n]-gate_potential_ref))
#plot the conductance as function of gatevoltage
import pylab
pylab.figure()
# make curve for each temperature
for i in range(len(temperature_list)):
    pylab.semilogy(gate_voltage_list,conductance_list[:,i])
pylab.xlabel("Gate Voltage (V)")
pylab.ylabel("Conductance (S)")
pylab.show()

```

The script implements a quite simple workflow:

1. It reads in the transmission spectrum from the calculation with 0 V gate potential:

```
#read the reference transmission spectrum from the netcdf data file
gate_potential_ref = 0.*Volt
transmission_spectrum = nlread("gatescan-6-6.nc",
    object_id="trans"+str(gate_potential_ref))[0]
```

2. Instead of setting the gate potential, the electrode voltages are varied such that the electrode Fermi levels are shifted

relative to each other:

```
            electrode_voltages=(gate_voltage_list[n]-gate_potential_ref,
                                gate_voltage_list[n]-gate_potential_ref))
```

The script produces the figure below. We see that the main main trends of the fully self-consistent calculations are reproduced by the linear response procedure.

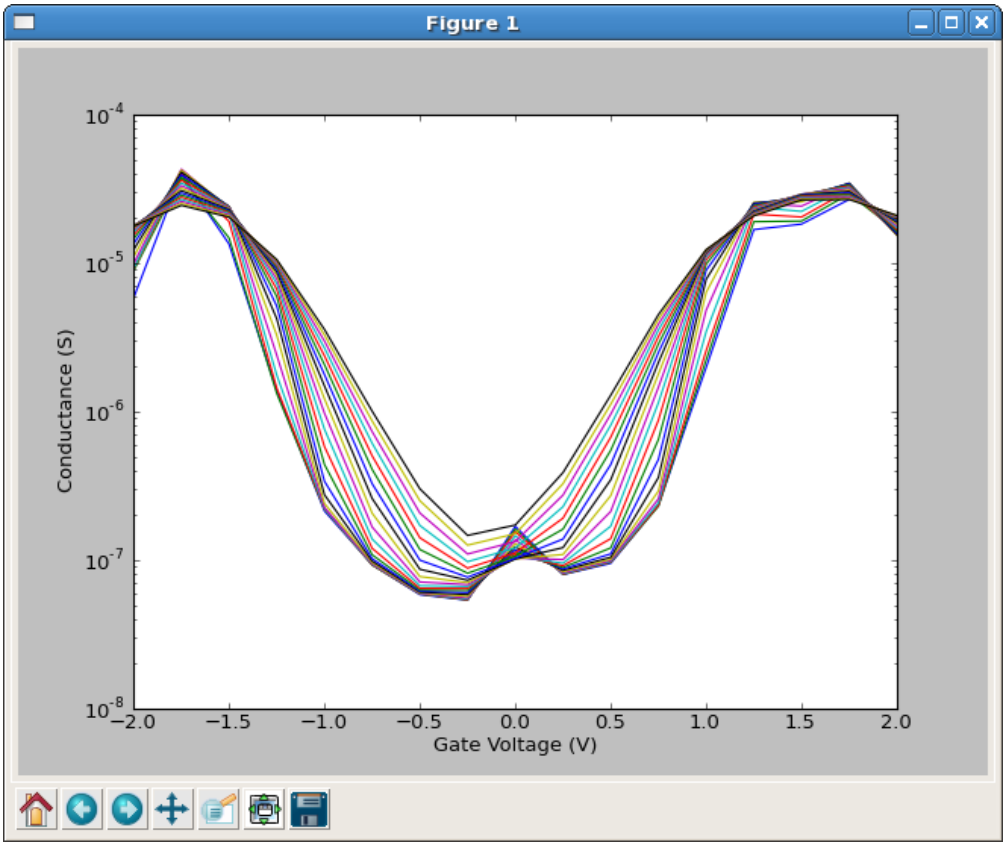


Fig. 3.79: Conductance as function of the gate potential calculated using linear response. Different curves are for electron temperatures 0, 100, 200, , 2000 Kelvin.

I–V characteristics

You will now investigate the effect of changing the external bias across the electrodes in the range 0-2 Volt. The calculations follow a similar procedure as the gate potential scan in the previous section.

Electrode bias loop

Use the script displayed below. You can download it here: `electrode_loop.py`.

```
# Read in the old configuration
device_configuration = nlread("z-a-z-6-6.nc",DeviceConfiguration)[0]
calculator = device_configuration.calculator()
# Output filename
filename = 'ivscan-6-6.nc'
# Define bias voltages
voltage_list=numpy.linspace(0.0,2.0,9)*Volt
for voltage in voltage_list:
    # Set new calculator with modified electrode voltages on the configuration
    # use the self consistent state of the old calculation as starting input.
    device_configuration.setCalculator(
    calculator(electrode_voltages=(-0.5*voltage,0.5*voltage)),
               initial_state=device_configuration)
    #Analysis
    electron_density = ElectronDifferenceDensity(device_configuration)
    nlsave(filename, electron_density, object_id='dens'+str(voltage))
    electrostatic_potential = ElectrostaticDifferencePotential(device_configuration)
    nlsave(filename, electrostatic_potential, object_id='pot'+str(voltage))
    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,200)*eV)
    nlsave(filename, transmission_spectrum, object_id='trans'+str(voltage))
    nlprint(transmission_spectrum)
```

The calculation uses the calculation for the junction with an applied gate potential of -1 V as a starting point. This value for the gate potential is fixed through the entire calculation. The electrode voltages are modified through the `electrode_voltages` variable of the `DeviceHuckelCalculator`.

The entire calculation will take more than 10 times longer than the single zero bias calculation. During the loop, the results will be saved in the file `ivscan-6-6.hdf5`. The file may be inspected during the calculation to follow how the calculation is progressing.

The plot below shows the electrostatic potential for a bias of 1 Volt, i.e. -0.5 V on the left electrode and +0.5 V on the right electrode. Notice how the electrostatic potential in the left electrode is closer to the electro-static potential of the gate.

Self-consistent I–V characteristics

You can now use all the **TransmissionSpectrum** objects in `ivscan-6-6.hdf5` to calculate the I–V curve for a gate potential of -1 V. Use the script shown below, which combines the methods presented in the previous sections. You can also download it here: `IV_curve.py`.

```
#make list of relevant temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list of relevant gate voltages
voltage_list=numpy.linspace(0.0,2.0,9)*Volt
#make list to hold the current calculations
current_list=numpy.zeros(len(voltage_list)*len(temperature_list))
current_list=current_list.reshape(len(voltage_list),len(temperature_list))
```

(continues on next page)

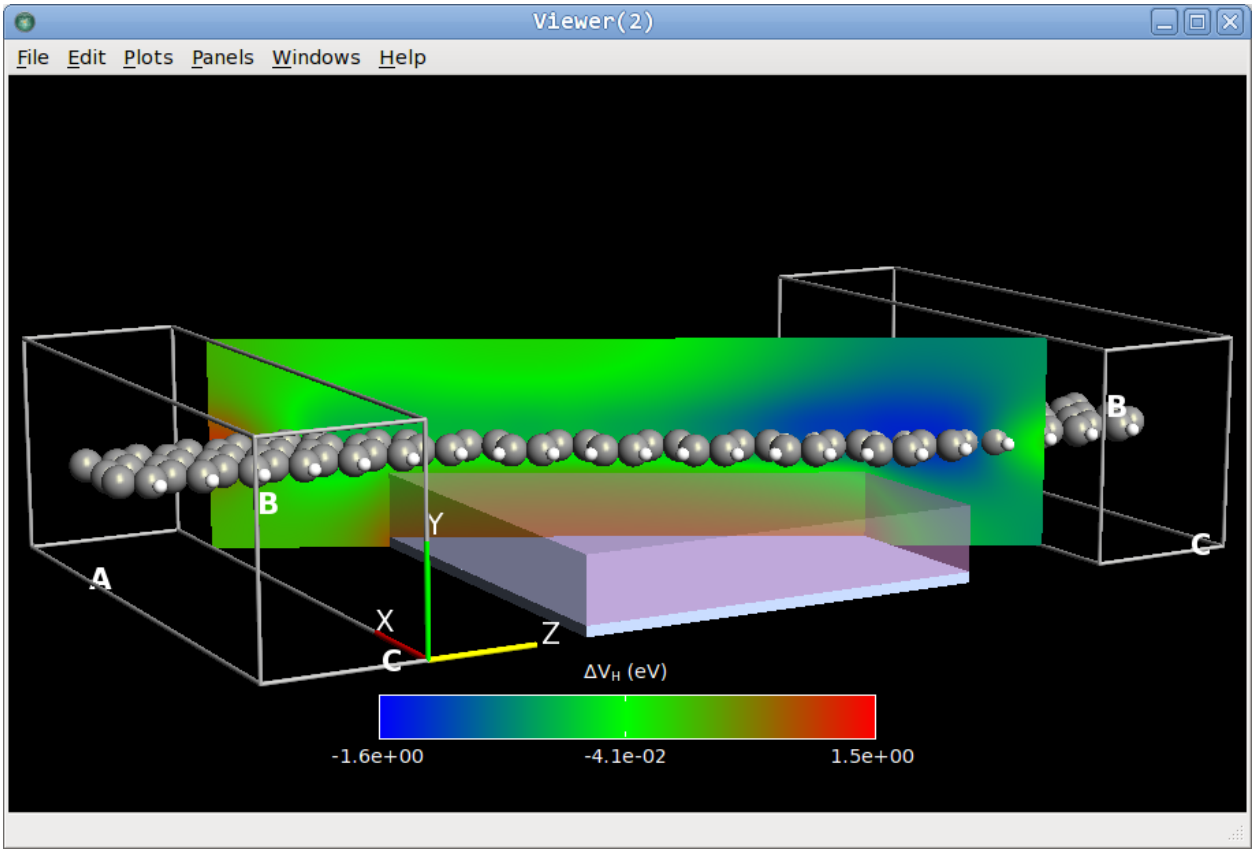


Fig. 3.80: Electrostatic potential of the junction with a symmetrically applied bias of 1 Volt and a gate potential of -1 Volt.

(continued from previous page)

```
#specify the filename for the netcdf data file
filename="ivscan-6-6.nc"
#loop through the gate voltages
for n in range(len(voltage_list)):
    transmission_spectrum=nlread(filename, object_id="trans"+str(voltage_list[n]))[0]
    #loop through the temperature list
    for i in range(len(temperature_list)):
        current_list[n,i]=transmission_spectrum.current(
            electrode_temperatures=(temperature_list[i],temperature_list[i]))
#plot the current as function of voltage
import pylab
pylab.figure()
# make curve for each temperature
for i in range(len(temperature_list)):
    pylab.plot(voltage_list,current_list[:,i])
pylab.xlabel("Bias (V)")
pylab.ylabel("Current (A)")
pylab.show()
```

The result of the calculation is illustrated below.

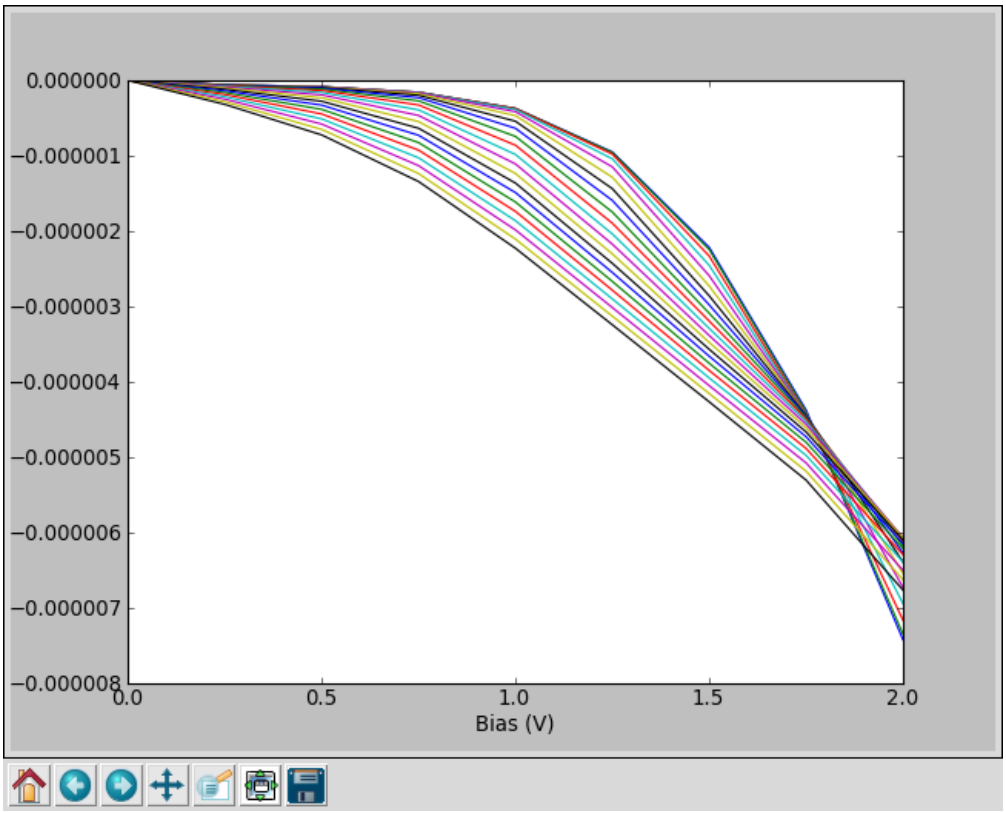


Fig. 3.81: Self-consistent current-voltage characteristics of the nanoribbon junction with -1 V gate potential. Curves are for electrode temperatures 0, 100, 200, .. 2000 Kelvin. The electric current increases with temperature.

Linear response I–V characteristics

Similar to the linear response approximation for the gate potential presented in the previous section, you may use a linear response approximation to calculate the finite-bias transmission spectra from the zero-bias spectrum. The following script performs a calculation of the linear response current for an electrode temperature of 300 Kelvin:

```
#make list of relevant temperatures
temperature=300*Kelvin
#make list of relevant gate voltages
voltage_list=numpy.linspace(0.0,2.0,9)*Volt
#make list to hold the current calculations
current_list=numpy.zeros(len(voltage_list))
current_list_lin=numpy.zeros(len(voltage_list))
#read the reference transmission spectrum from the netcdf data file
potential_ref = 0.*Volt
filename="ivscan-6-6.nc"
transmission_spectrum0 = nlread(filename, object_id="trans"+str(potential_ref))[0]
for n in range(len(voltage_list)):
    transmission_spectrum=nlread(filename, object_id="trans"+str(voltage_list[n]))[0]
    current_list[n]=transmission_spectrum.current(
        electrode_temperatures=(temperature,temperature))
    current_list_lin[n]=transmission_spectrum0.current(
        electrode_temperatures=(temperature,temperature),
        electrode_voltages=(-0.5*(voltage_list[n]-potential_ref),
        0.5*(voltage_list[n]-potential_ref) ) )
#plot the current as function of bias
import pylab
pylab.figure()
pylab.plot(voltage_list,current_list,label="self consistent")
pylab.plot(voltage_list,current_list_lin,label="linear response")
pylab.xlabel("Bias (V)")
pylab.ylabel("Current (A)")
pylab.legend(loc="lower left")
pylab.grid(True)
pylab.show()
```

The figure below compres the linear response results to the self-consistent ones. The agreement is quite good for low bias, but at large bias the linear response current does start to deviate from the self-consistently calculated current.

When is the linear response approximation valid?

The previous sections illustrate how to perform calculations of the current as function of the electrode bias or the gate potential. Even though the methodology is based on the extended Huckel formalism, such calculations can be quite time consuming, and an alternative is to invoke the linear response approximation in and use non-self-consistent calculations, which are much faster than self-consistent ones.

However, an important question emerges: **When is it OK to use linear response instead of fully self-consistent calculations?** You have already seen that linear response works well for a relatively small electrode bias, but how about the influence of the gate potential?

The scripts presented in the previous sections can be easily modified to vary the gate potential and the electrode bias simultaneously. Below is a comparison between the self-consistent current and the linear response current. For a small bias and gate potential the two are in good agreement, illustrating that the linear response current is a good first approximation.

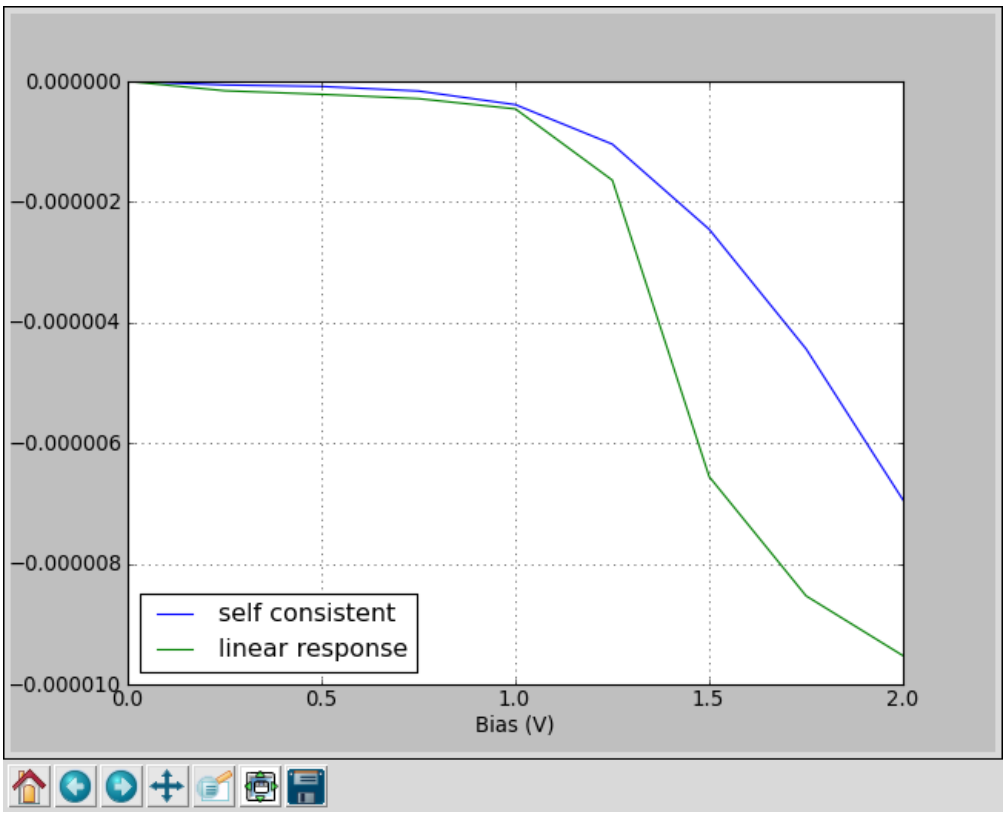


Fig. 3.82: Comparison of the self-consistent (blue) and linear response (green) current-voltage characteristics of the junction with a -1 V gate potential and an electrode temperature of 300 Kelvin.

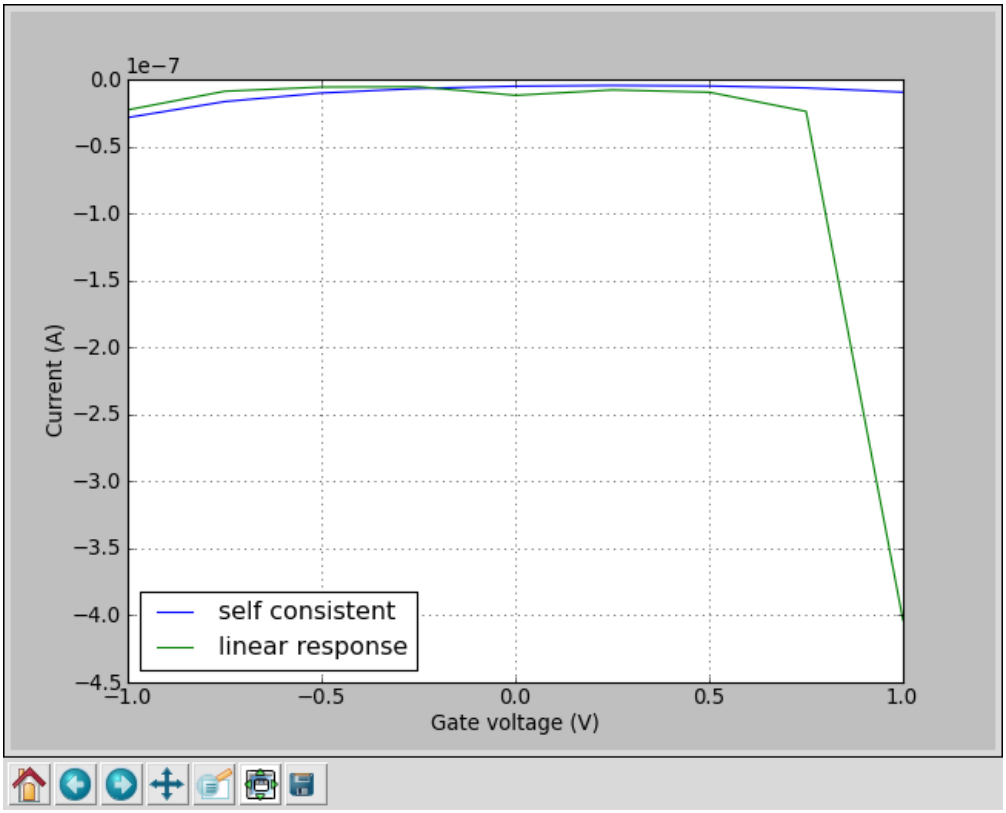


Fig. 3.83: Current for an applied bias of 0.2 Volt as a function of the gate potential. The blue curve was obtained from a fully self-consistent calculation, while the green curve was obtained from a linear response calculation using the transmission spectrum of a system with zero gate potential and bias.

The scripts that generated the self-consistent data at the plot can be downloaded here: `potentials_vary.py`, `electrode_0.2V_gate_varies.py`.

Further analysis with ATK-SE

Some of the other analyses that can be performed with **QuantumATK** are

- 3D visualization of scattering states.
- 3D visualization of transmission pathways.
- Projected density of states.

The main part of the analysis in this tutorial are performed using QuantumATK scripts. However, analysis can in most cases also be performed directly from the QuantumATK GUI. For further information, see the *Transport calculations with QuantumATK*.

Temperature dependent conductance

In this section you will learn how the transmission spectrum can be used to calculate the conductance of the device as a function of the left and right electrode temperatures, T_L and T_R. At $T_L = T_R = 0$, the conductance is determined by the transmission coefficient at the Fermi Level, while for finite electrode temperatures, the conductance depends on the value of the transmission coefficient in an energy window around the Fermi level. The zero-bias conductance is given by

$$\sigma(T_L) = \frac{2e^2}{h} \int T(E)\ f'(\eta(E)) \frac{dE}{k_b T_R},$$

where $f'(\eta(E)) = f'\left(\frac{E-E_F^L}{k_b T_R}\right)$ is the derivative of the Fermi function[2].

In semiconductors the conductance is often determined by hot electrons or holes propagating within the conduction or valence band, so-called thermionic emission. The hot electrons are located in the energy range of the tails of the Fermi function. Thus, for an accurate calculation of the conductance from thermionic emission, it is important that the energy range of the transmission spectrum is such that the tails of the Fermi functions are properly sampled. The transmission spectrum figure in the previous chapter shows that both the valence and conduction band edges of the central region are within the energy range of the transmission spectrum. From that data you will therefore be able to accurately determine the temperature dependent conductance.

Use the following QuantumATK script to calculate and plot the temperature dependent conductance. It can be downloaded here: `temperature_dependent.py`.

```
# Read first transmission spectrum from file z-a-z-6-6.hdf5
transmission_spectrum = nlread("z-a-z-6-6.hdf5",TransmissionSpectrum)[0]
#make list of temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list for holding conductance
conductance_list=numpy.zeros(len(temperature_list))
#calculate the conductance for each temperature in the temperature list
for i in range(len(temperature_list)):
    conductance_list[i]=transmission_spectrum.conductance(
        electrode_temperatures=(temperature_list[i], temperature_list[i]))
#plot the conductance as function of temperature
import pylab
```

(continues on next page)

[2] H.ăH.ăB. Sorensen, P.ăC. Hansen, D.ăE. Petersen, S.ăSkelboe, and K.ăStokbro. Efficient wave-function matching approach for quantum transport calculations. *Phys. Rev. B*, 79:205322, May 2009. doi:10.1103/PhysRevB.79.205322.

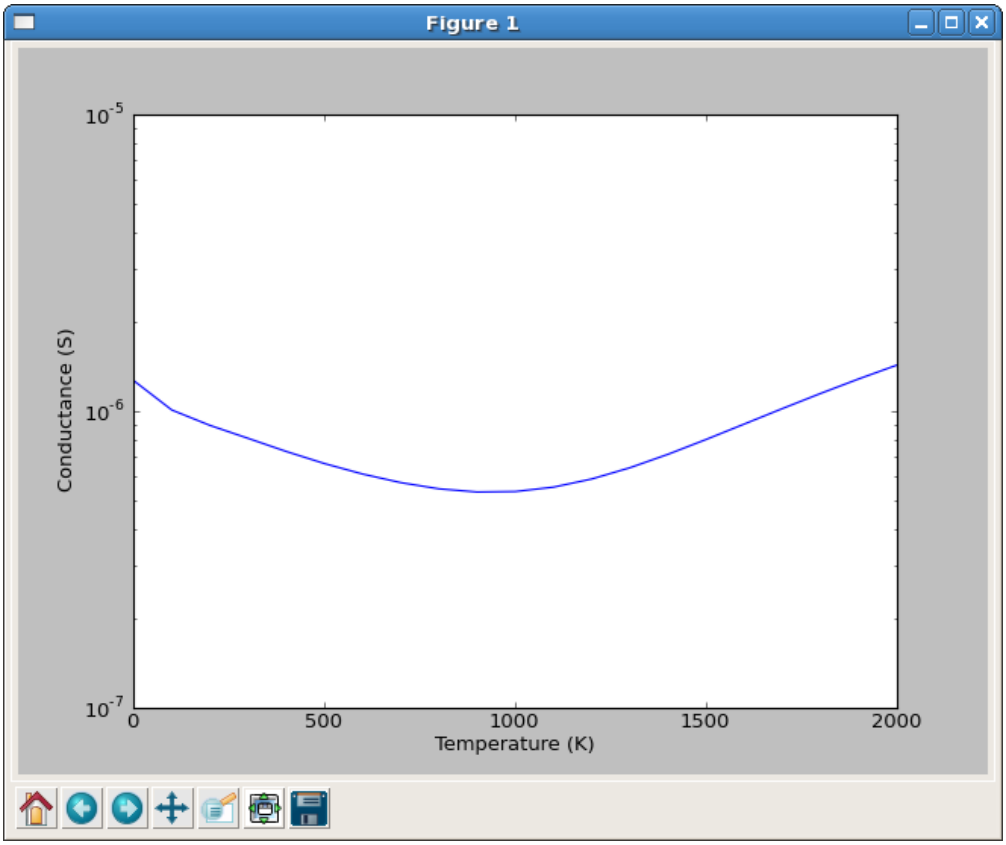
(continued from previous page)

```
pylab.figure()
pylab.semilogy(temperature_list,conductance_list)
pylab.xlabel("Temperature (K)")
pylab.ylabel("Conductance (S)")
pylab.show()
```

The script reads the previously calculated transmission spectrum and then uses the `conductance()` method for calculating the conductance for a number of electrode temperatures. Save the script and run it using the Job Manager or run it from a terminal:

```
$ atkpython temperature_dependent.py
```

You will obtain the figure below.



The temperature dependent conductance for this device is typical for a field effect transistor: The initial decrease in conductance as a function of temperature is related to a large conductance at the Fermi level caused by electron tunneling. At larger temperatures, the usual increase of conductance with temperature starts dominating. The tunneling effect will be suppressed in a longer device, as you will see in the next section.

Comparison to results for a longer device

Attention: This section employs a device with a longer nanoribbon. Build the device as described here: *A longer nanoribbon transistor*.

You will now calculate the transmission spectrum of the nanoribbon transistor with a longer channel, and compare its temperature dependent conductance with the shorter system investigated earlier.

Electron conductance

Open the large transistor configuration in the Builder, and confirm that it looks as expected. Next step is to set up the calculations needed for analyis of the electronic conductance in exactly the same way as outlined in the previous section for the shorter device. There are two ways to do this:

1. Send the device to the **Scripter** and repeat all the steps that were needed to prepare the script `z-a-z-6-6.py`, and save the new script as `z-a-z-20-6.py`.
2. Alternatively, you may use the **Editor** to generate the new script:

- Save the new geometry into the file `z-a-z-20-6.py`.
- Copy the lines specifying the calculation from `z-a-z-6-6.py`, and insert them into `z-a-z-20-6.py`.

Note: Remember to change the HDF5 output filename to `z-a-z-20-6.hdf5`.

- The part of the script specifying the calculation is found below the lines indicating Calculator:

```
#########################################################
# Calculator
#########################################################
```

- Run the script using the Job Manager or from a terminal. The calculation will take approximately twice as much time as the previous calculation for the shorter junction.
- When the calculation is done, use the script below to plot and compare the temperature dependent conductance of the two junctions. You can download it here: `temperature_dependent_compare.py`.

```
# Read first transmission spectra from fil z-a-z-6-6.hdf5 and z-a-z-20-6.hdf5
transmission_spectrum6 = nlread("z-a-z-6-6.hdf5",TransmissionSpectrum)[0]
transmission_spectrum20 = nlread("z-a-z-20-6.hdf5",TransmissionSpectrum)[0]
#make list of temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list for holding conductance
conductance_list6=numpy.zeros(len(temperature_list))
conductance_list20=numpy.zeros(len(temperature_list))
for i in range(len(temperature_list)):
    conductance_list6[i]=transmission_spectrum6.conductance(
        electrode_temperatures=(temperature_list[i],temperature_list[i]) )
    conductance_list20[i]=transmission_spectrum20.conductance(
        electrode_temperatures=(temperature_list[i],temperature_list[i]) )
#plot the conductance as function of temperature
```

(continues on next page)

(continued from previous page)

```
import pylab
pylab.figure()
pylab.semilogy(temperature_list,conductance_list6,label="short junction")
pylab.semilogy(temperature_list,conductance_list20,label="long junction")
pylab.xlabel("Temperature (K)")
pylab.ylabel("Conductance (S)")
pylab.legend(loc="lower left")
pylab.grid(True)
pylab.show()
```

Below is the result of the calculation. The green curve shows the result for the longer junction. Note that in this case the conductance increases strongly with temperature, indicating that the dominating transport mechanism is thermionic emission rather than electron tunneling.

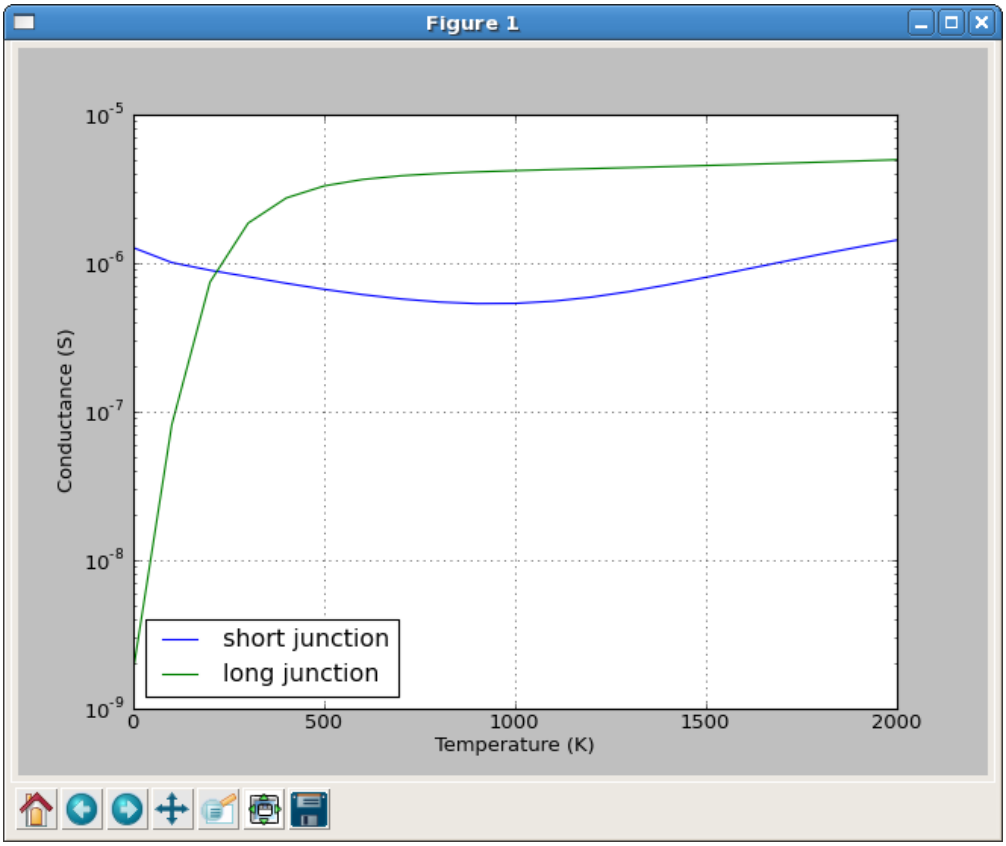


Fig. 3.84: Conductance as a function of the electron temperature for the short (blue) and 20 unit long (green) graphene nanoribbon transistor.

Electron thermal transport

In the previous section you calculated the conductance of the junction. To obtain an electrical current it is usually necessary to apply an external bias. This section shows how it is also possible to drive a current through the junction by having the two electrodes at different temperatures. The QuantumATK script for the calculation is displayed below. You can download the script here: `electrodes_diff_temperature.py`.

```
# Read first transmission spectra from file z-a-z-6-6.hdf5 and z-a-z-20-6.hdf5
transmission_spectrum6 = nlread("z-a-z-6-6.hdf5",TransmissionSpectrum)[0]
transmission_spectrum20 = nlread("z-a-z-20-6.hdf5",TransmissionSpectrum)[0]
#make list of temperatures
temperature_list=numpy.linspace(0,2000,21)*Kelvin
#make list for holding current
current_list6=numpy.zeros(len(temperature_list))
current_list20=numpy.zeros(len(temperature_list))
#fix the temperature in the left electrode
temperature_left=300*Kelvin
for i in range(len(temperature_list)):
    current_list6[i]=transmission_spectrum6.current(
        electrode_temperatures=(temperature_left,temperature_list[i]) )
    current_list20[i]=transmission_spectrum20.current(
        electrode_temperatures=(temperature_left,temperature_list[i]) )
#plot the current as function of temperature in right electrode
import pylab
pylab.figure()
pylab.plot(temperature_list,current_list6,label="short junction")
pylab.plot(temperature_list,current_list20,label="long junction")
pylab.xlabel("Right Electrode Temperature (K)")
pylab.ylabel("Current (A)")
pylab.legend(loc="lower left")
pylab.grid(True)
pylab.show()
```

The electron temperature of the left electrode is now fixed at 300 Kelvin, while the temperature of the right electrode is varied from 0 to 2000 Kelvin. The figure below shows the output of the calculation. The change in temperature has the strongest effect on the longer junction (the green curve). The positive direction of the current is from left to right, thus, the electrical current goes from the high temperature to the low temperature.

If you use the **Transmission Analyzer** to plot the transmission spectrum for the long junction, you will see that the valence band of the central region is closest to the Fermi Level. The charge transport is therefore dominated by hole transport. The electrode with the highest temperature will have most holes, which will propagate to the other electrode, giving rise to a current in this direction.

References

3.5.9 Silicon nanowire field-effect transistor

Version: 2016.0

Downloads & Links

`bandstructure_analyzer.py` `nanodevice_gatescan.py` `conductance_plot.py`

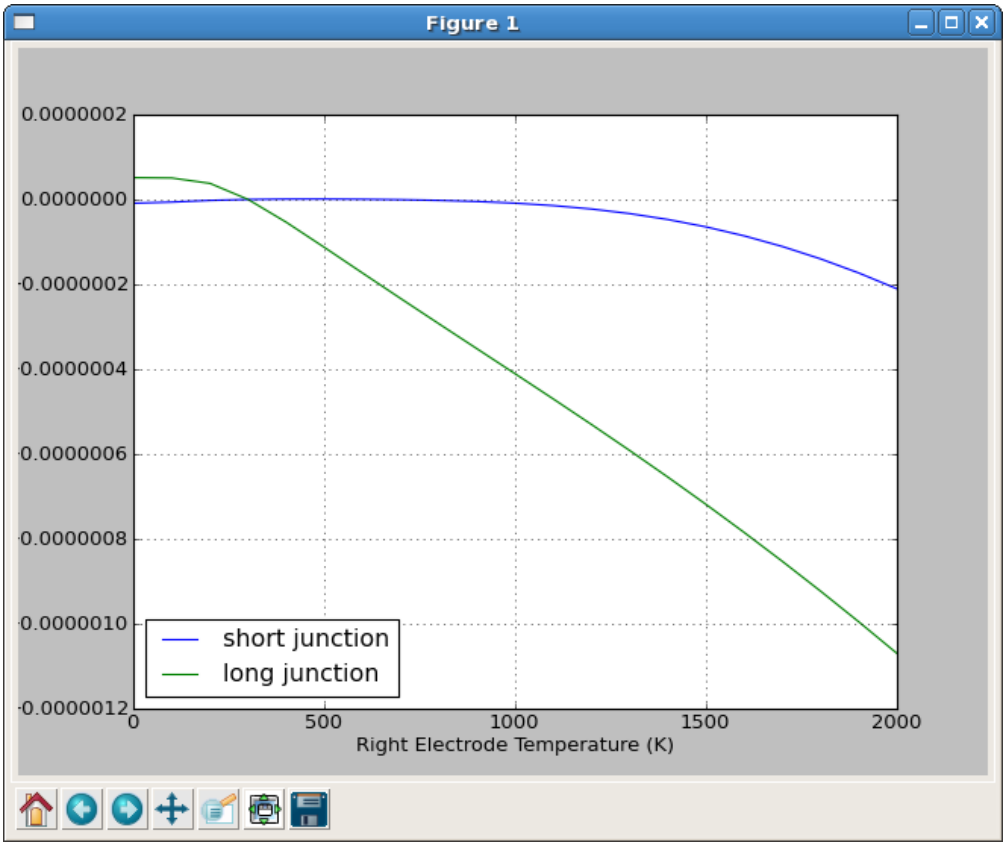


Fig. 3.85: Current as a function of the electron temperature of the right electrode for the short (blue) and 20 unit long (green) graphene junction. The electron temperature of the left electrode is fixed at 300 Kelvin.

Introduction

This tutorial shows you how to set up and perform calculations for a device based on a silicon nanowire. You will define the structure of a hydrogen passivated Si(100) nanowire, and set up a field-effect transistor (FET) structure with a cylindrical wrap-around gate. The tutorial ends with calculations of the device conductance as a function of gate bias.

Note: You will primarily use the graphical user interface QuantumATK for setting up and analyzing the results. The underlying calculation engines for this tutorial are ATK-DFT and ATK-SE. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK Reference Manual.

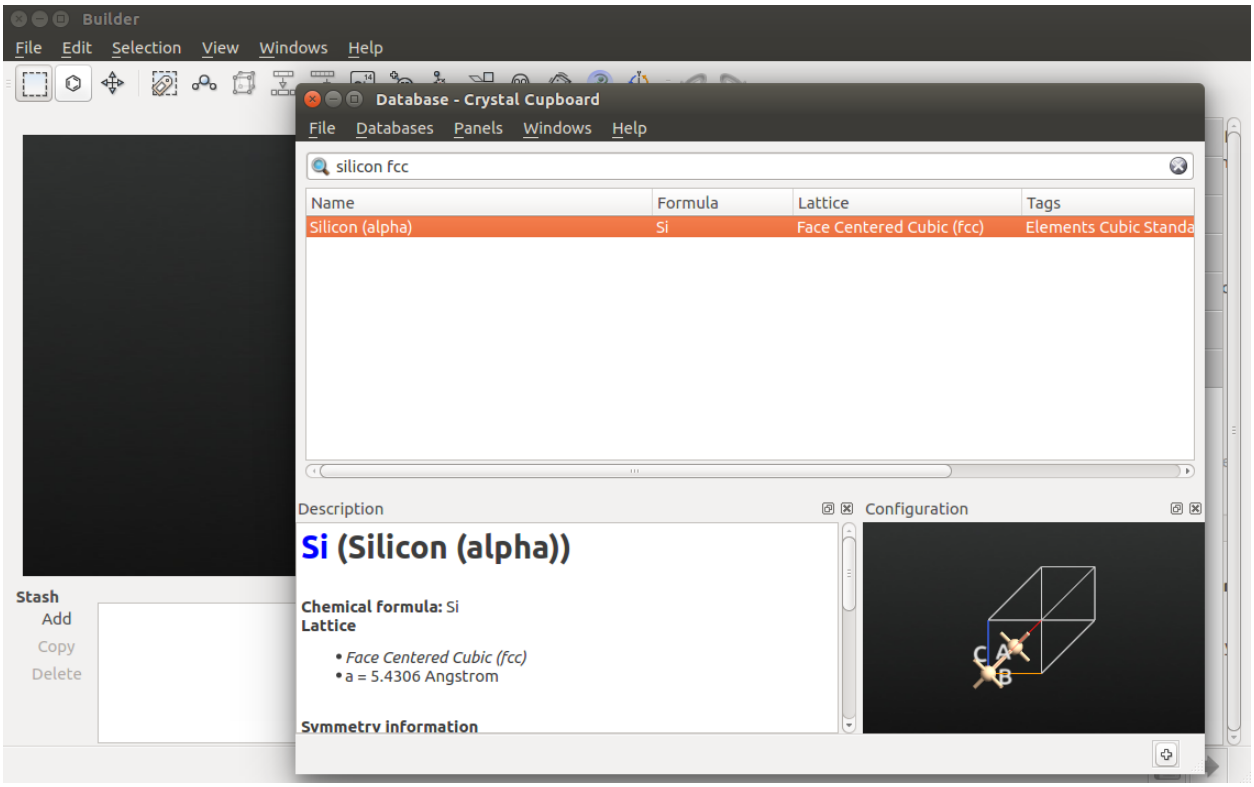
Band structure of a Si(100) nanowire

First step is to set up the Si(100) nanowire and optimize the geometry. You should use the ATK-DFT calculator for this. We then compute the band structure of the nanowire using 3 different computational models; DFT-GGA, DFT-MGGA, and the Extended Hückel method.

Building the nanowire

Open QuantumATK and create a new empty project called silicon_nanowire. Then launch the **Builder**.

- Go to *Add* → *From Database* and type silicon fcc to locate the diamond phase of silicon.
- Add the *Silicon (alpha)* bulk configuration to the **Stash** (double-click or use the icon).



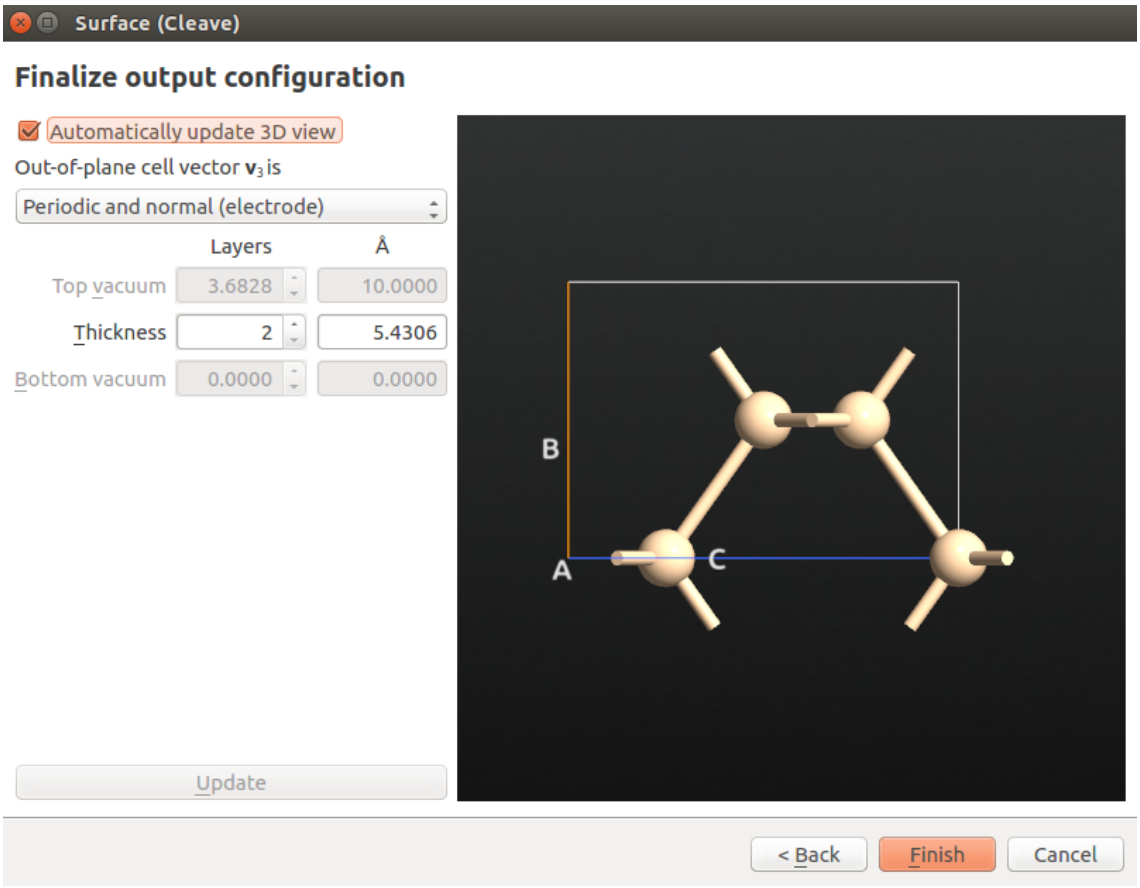
In the Builder, use the *Builders* → *Surface (Cleave)* tool to create the Si(100) facet:

•keep the default (100) cleave direction, and click *Next*;

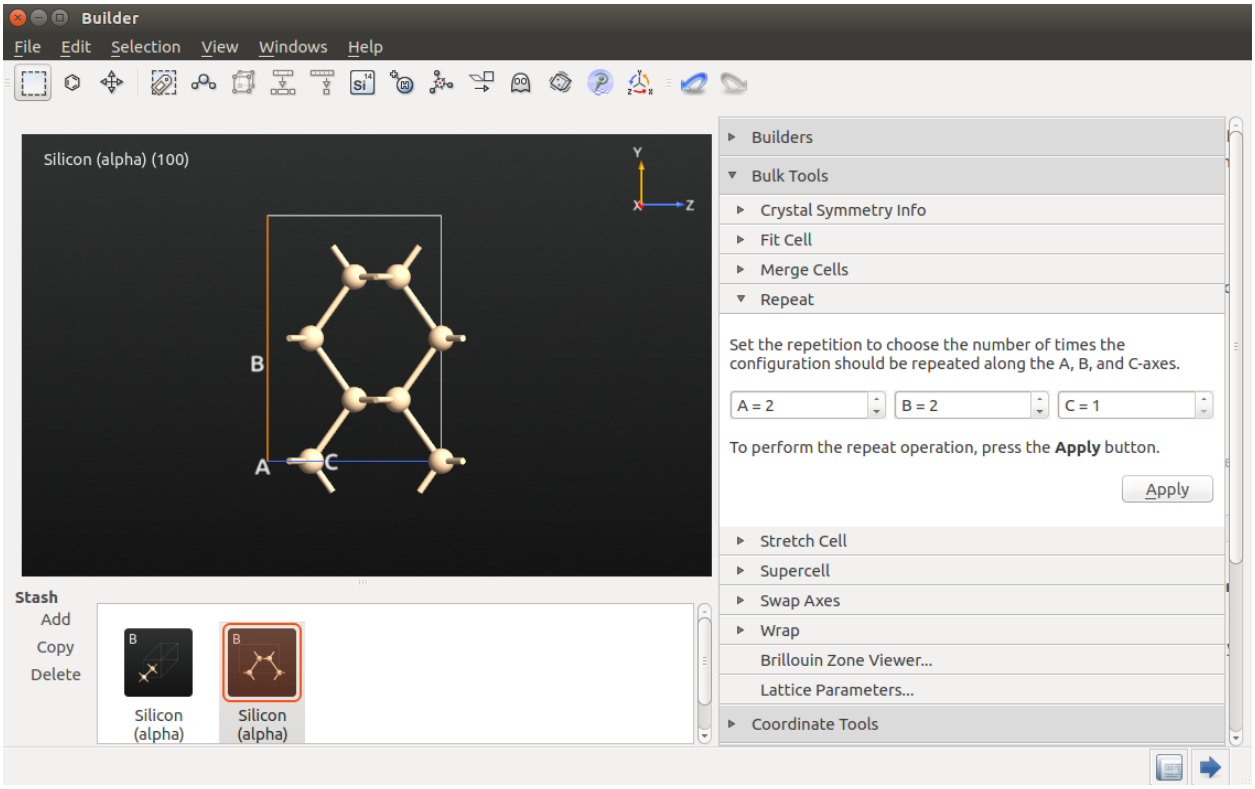
•keep the default surface lattice, and click *Next*;

•keep the default supercell, which will ensure that the wire direction is perpendicular to the surface, and click *Next*.

•Click *Fiish* to add the cleaved structure to the Stash.



Next, use the *Bulk Tools* → *Repeat* tool to repeat the structure twice along the A and B directions:

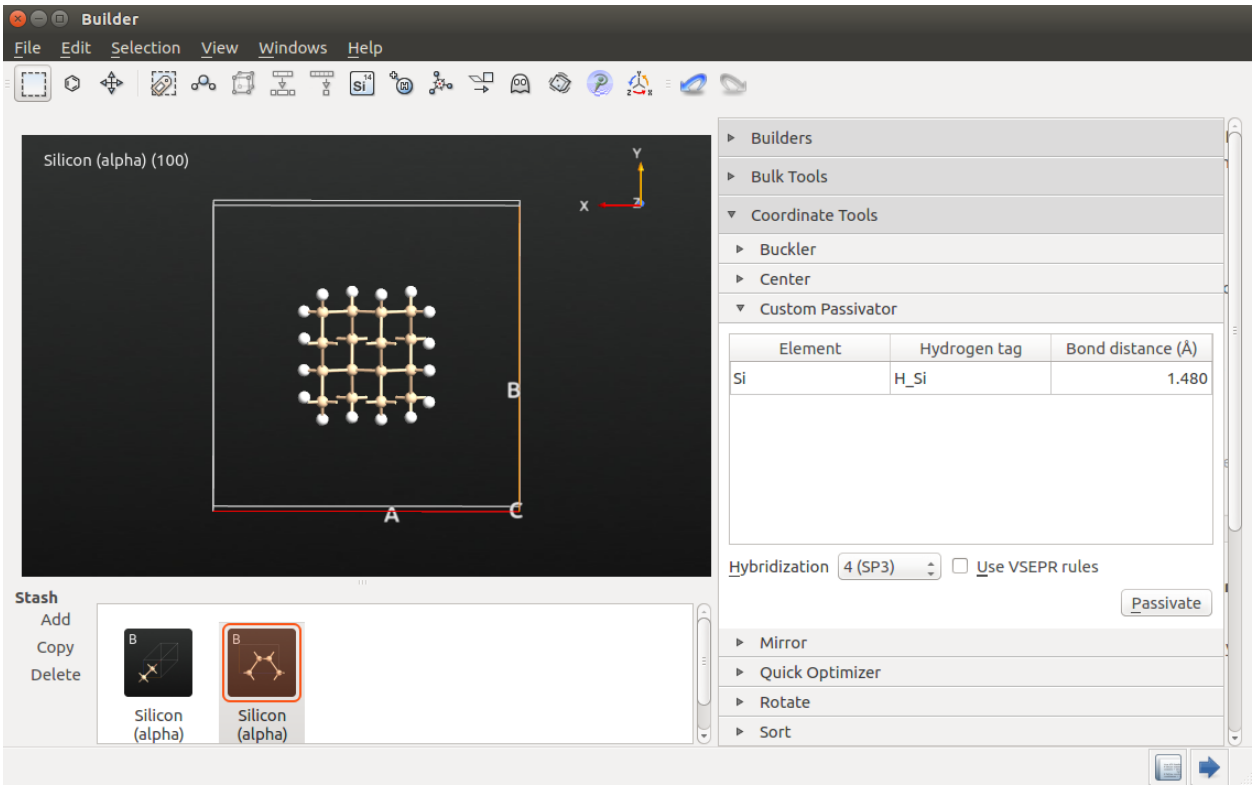


Finalize the nanowire by following these steps:

•use the *Bulk Tools* → *Lattice Parameters* tool to set the A and B lattice vector lengths to 20 Å;

•use the *Coordinate Tools* → *Center* tool to center the structure along all directions;

•go to *Coordinate Tools* → *Custom Passivator* and apply the 4(SP3) type hydrogen passivation to the nanowire.



Send the H-passivated Si(100) nanowire to the **Script Generator** by using the in the lower right-hand corner of the Builder window.

Setting up and running the calculations

You should now relax the nanowire and compute the band structure. The DFT-GGA method is used for geometry optimization, while the band structure is computed using DFT-GGA, DFT-MGGA, and the Extended Hückel model.

Important: The TB09 meta-GGA and Extended Hückel models can not be used for relaxation.

In the Script Generator, change the default output file name to `si_100_nanowire.hdf5`, and add the following blocks to the script:

- New Calculator
- OptimizeGeometry
- Bandstructure
- New Calculator
- Bandstructure

- New Calculator
- Bandstructure

Open the first **New Calculator** block, and make the following changes to the calculator settings:

•set the k-point sampling to 1x1x11;

•change the exchange-correlation potential to GGA.

Open the second **New Calculator** block, and make the following changes:

•set the k-point sampling to 1x1x11;

•change the exchange-correlation potential to MGGA.

Open the third **New Calculator** block, and make the following changes:

•select the **ATK-SE: Extended Hückel** calculator;

•uncheck No SCF iteration to make the calculation selfconsistent;

•set the k-point sampling to 1x1x11;

•increase the density mesh cut-off to 20 Hartree;

•go to the **Hückel basis set** tab, and select the Hoffmann.Hydrogen and Cerda.Silicon [GW diamond] basis sets. The latter has been fitted to GW calculations, and gives an excellent description of the silicon bandstructure, including the value of the band gap.

Finally, open each **Bandstructure** script block and set the number of points per segment to 100.

Tip: The default value for the density mesh cut-off is often sufficient, but by increasing it you may sometimes obtain a slightly higher accuracy at a small additional computational cost.

TB09 meta-GGA c-parameter

You should also adjust the value of the c-parameter used in the TB09 meta-GGA method. This is essentially a fitting parameter that can be used to adjust the calculated band gap of the material. If the c-parameter is not defined by the user, a value is automatically calculated from the electronic structure of the material. However, this will not work for a system with vacuum regions, like a nanowire.

You will therefore use a fixed value, c=1.0364, which has been fitted to yield an accurate prediction of the fundamental band gap in silicon (1.13 eV).

Transfer the QuantumATK Python script to the **Editor** using the icon. Then locate the script line defining the variable `exchange_correlation` for the MGGA exchange-correlation, and set the fixed value of c to 1.0364, as illustrated below.

```
#-----------------------------------------
# Exchange-Correlation
#-----------------------------------------
exchange_correlation = MGGA.TB09LDA(c=1.0364)
```

Running the calculations

Save the QuantumATK Python script as `si_100_nanowire.py` and execute it using the **Job Manager**. It will take just a few minutes to complete the calculations.

Analyzing the results

Once the calculations have finished, the HDF5 data file `si_100_nanowire.hdf5` should appear on the QuantumATK **LabFloor**. Locate the Bandstructure items inside this file. The order of the item IDs corresponds to the order in which the band structures were computed and saved, i.e. GGA, MGGA, and Hückel.

Select one of the band structures and open the **Bandstructure Analyzer** in the right-hand Panel Bar to inspect the computed band structure. You can do this for all three band structures, or use the **Compare Data** plugin to plot a direct comparison of the three band structures.

Detailed band structure analysis

In order to compare the band gaps obtained with the three computational models, it is convenient to use `bandstructure_analyzer.py`, which uses QuantumATK Python for detailed analysis of the gaps in a band structure. The script is shown below:

```
from QuantumATK import *
import pylab
# Custom analyzer for calculating the band gap of a bandstructure

#helper function to find minima
def fitEnergyMinimum (i_min, energies, k_points, nfit=3):
    """
    Function for fitting the energy minimum located around i_min.

    @param i_min   : approximate position of the energy minimum.
    @param energies: list of energies.
    @param k_points: list of k_points which correspond to the energies.
    @param nfit    : order of the polynomium.
    @return d2_e, e_min, k_min : second derivative of energy,
                                 minimum energy, minimum k_point.
    """
    #list of energies
    n = len(energies)
    efit = numpy.array([energies[(n+i_min-nfit/2+i)%n] for i in range(nfit)])
    kfit = numpy.array([i-nfit/2 for i in range(nfit)])
    #special cases
    if i_min == 0: #assume bandstructure symmetric around zero
        for i in range(nfit/2):
            efit[i] = energies[nfit/2-i]

    if i_min == n-1: #assume bandstructure symmetric around end point
        for i in range(nfit/2+1, nfit):
            efit[i] = energies[n-1+nfit/2-i]

    #make fit
```

(continues on next page)

(continued from previous page)

```
    p = numpy.polyfit(kfit,efit,2)
    i_fit_min = -p[1]/2./p[0]
    pf = numpy.poly1d(p)
    e_min = pf(i_fit_min)
    i0 = int(i_fit_min+i_min+n)
    w = i_fit_min+i_min+n-i0
    k_min = (1-w)*k_points[i0%n]+w*k_points[(i0+1)%n]

    return p[0], e_min, k_min

def analyseBandstructure(bandstructure, spin):
    """
    Function for analysing a band structure and calculating bandgaps.

    @param bandstructure :  The bandstructure to analyze
    @param spin          :  Which spin to select from the bandstructure.
    @return  e_val, e_con, e_gap :  maximum valence band energy,
                                    minimum conduction band energy,
                                    and direct band gap.
    """
    energies = bandstructure.evaluate(spin=spin).inUnitsOf(eV)

    #some placeholder variable to help finding the extrema
    e_valence_max = -1.e10
    e_conduction_min = 1.e10
    e_gap_min = 1.e10
    i_valence_max = 0
    i_conduction_min = 0
    i_gap_min = 0
    n_valence_max = 0
    n_conduction_min = 0
    n_gap_min = 0

    # Locate extrema
    for i in range(energies.shape[0]):
        # find first state below Fermi level
        n = 0
        while n < energies.shape[1] and energies[i][n] < 0.0:
            n += 1

        # find maximum of valence band
        if (energies[i][n-1] > e_valence_max):
            e_valence_max = energies[i][n-1]
            i_valence_max = i
            n_valence_max = n-1
        # find minimum of conduction band
        if (energies[i][n] < e_conduction_min):
            e_conduction_min=energies[i][n]
            i_conduction_min=i
            n_conduction_min=n
        # find minimum band gap
        if (energies[i][n]-energies[i][n-1] < e_gap_min):
```

(continues on next page)

(continued from previous page)

```
            e_gap_min = energies[i][n]-energies[i][n-1]
            i_gap_min = i
            n_gap_min = n-1

    # Print out results
    a_val, e_val, k_val = fitEnergyMinimum(i_valence_max,
                                           energies[:,n_valence_max],
                                           bandstructure.kpoints())
    print('Valence band maximum    %7.4f eV at [%6.4f, %6.4f,%6.4f]   ' \
                         %(e_val, k_val[0], k_val[1], k_val[2]))

    a_con, e_con, k_con = fitEnergyMinimum(i_conduction_min,
                                           energies[:,n_conduction_min],
                                           bandstructure.kpoints())
    print('Conduction band minimum %7.4f eV at [%6.4f, %6.4f,%6.4f]   ' \
                         %(e_con, k_con[0], k_con[1], k_con[2]))

    print('Fundamental band gap    %7.4f eV ' % (e_con-e_val))

    a_gap, e_gap, k_gap = fitEnergyMinimum(i_gap_min,
                            energies[:,n_gap_min+1]- energies[:,n_gap_min],
                            bandstructure.kpoints())

    print('Direct band gap         %7.4f eV at [%6.4f, %6.4f,%6.4f]   ' \
                         %(e_gap, k_gap[0], k_gap[1], k_gap[2]))
    return e_val, e_con, e_gap

def analyzer(filename, **args):
    """
    Find band gaps of band structures in netcdf file.
    """

    if filename == None:
        return

    #read in the bandstructure you would like to analyze
    bandstructure_list = nlread(filename, Bandstructure)
    if len(bandstructure_list) == 0 :
        print('No Bandstructures in file ', filename)
        return

    for s in [Spin.All]:
        b_list = []
        n = 0
        for b in bandstructure_list:
            print('Analyzing bandstructure number ', n)
            b_list = b_list + [analyseBandstructure(b,s)]
            print()
            print()
            n += 1

        x = numpy.arange(len(b_list))
```

(continues on next page)

(continued from previous page)

```
        e_val =  numpy.array([b[0] for b in b_list])
        e_con =  numpy.array([b[1] for b in b_list])
        e_indirect = e_con-e_val
        e_direct = numpy.array([b[2] for b in b_list])


analyzer("si_100_nanowire.hdf5")
```

Download the script (link: `bandstructure_analyzer.py`), and execute it using the **Job Manager**. You should get the following output:

```
Analyzing bandstructure number  0
Valence band maximum    -1.6395 eV at [0.0000, 0.0000,0.0121]
Conduction band minimum  1.6400 eV at [0.0000, 0.0000,0.0000]
Fundamental band gap     3.2795 eV
Direct band gap          3.2799 eV at [0.0000, 0.0000,0.0000]


Analyzing bandstructure number  1
Valence band maximum    -1.8434 eV at [0.0000, 0.0000,0.1501]
Conduction band minimum  1.8450 eV at [0.0000, 0.0000,0.0000]
Fundamental band gap     3.6884 eV
Direct band gap          3.6914 eV at [0.0000, 0.0000,0.0026]


Analyzing bandstructure number  2
Valence band maximum    -1.7737 eV at [0.0000, 0.0000,0.0000]
Conduction band minimum  1.7735 eV at [0.0000, 0.0000,0.0000]
Fundamental band gap     3.5472 eV
Direct band gap          3.5472 eV at [0.0000, 0.0000,0.0000]
```

The output shown above gives the band structure analysis for the DFT-GGA, DFT-MGGA, and Extended Hückel models. The script gives the valence band maximum, conduction band minimum, fundamental band gap, and direct band gap from the valence band maximum. The fundamental band gap for the Si(100) nanowire is significantly larger than that of bulk silicon, and all three computational models place it at the Gamma point or close to it.

Furthermore, the TB09-MGGA model gives a gap which is about 0.5 eV larger than the GGA model, which is in accordance with the difference in the band gaps of bulk Si calculated with the GGA and TB09-MGGA methods. The Extended Hückel model is in agreement with the TB09-MGGA method.

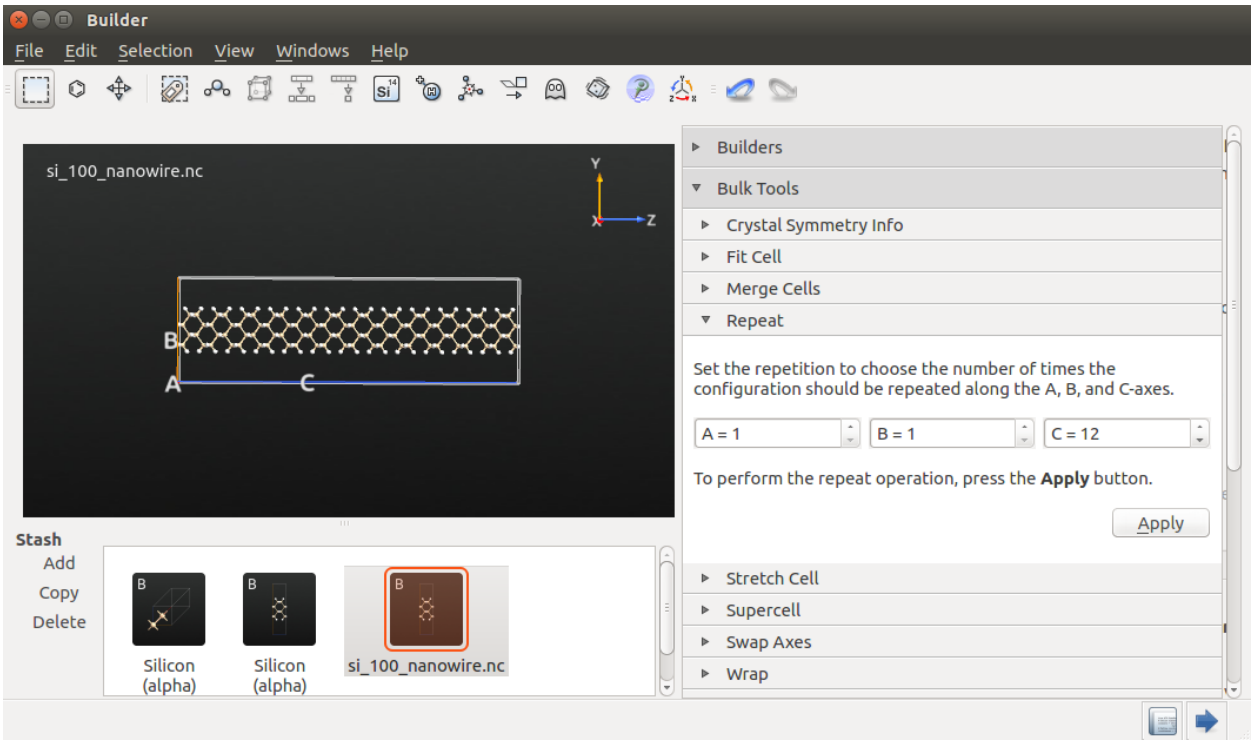
Si(100) nanowire FET device

You will now consider a Si(100) nanowire field-effect transistor with a cylindrical wrap-around gate. You should first build the device configuration, including the gate and doping of the silicon nanowire, and then use the Extended Hückel model to compute device properties such as zero-bias transmission and conductance vs. gate voltage.

FET device configuration

You should use the relaxed nanowire configuration as a starting point for building the device. The contents of the file `si_100_nanowire.hdf5` should be available on the QuantumATK **LabFloor**. It contains four bulk configurations: The first one *(gID000)* is the unrelaxed structure, while the remaining ones correspond to the relaxed structure, calculated with different methods (DFT-GGA, DFT-MGGA and Extended Hückel).

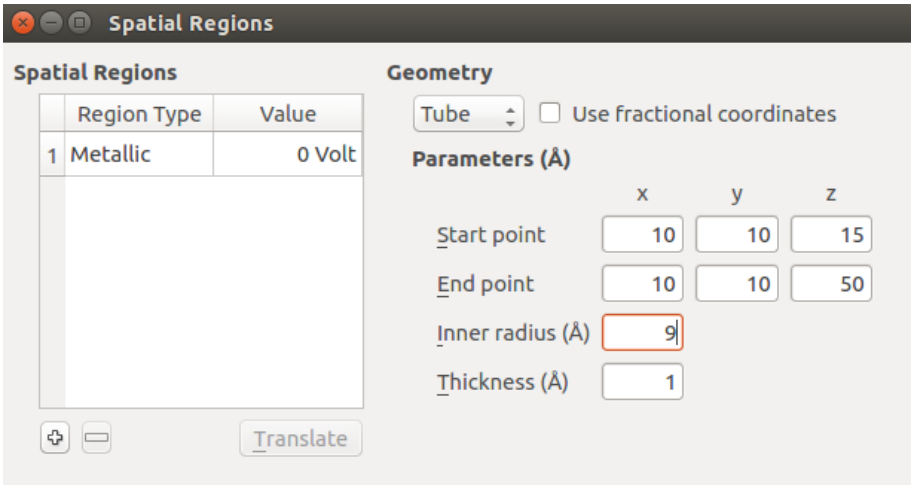
Send the configuration with ID *gID001* to the **Builder**. It will appear in the Builder Stash. Then use the *Bulk Tools* → *Repeat* tool to repeat the configuration 12 times along C.



Setting up the gate

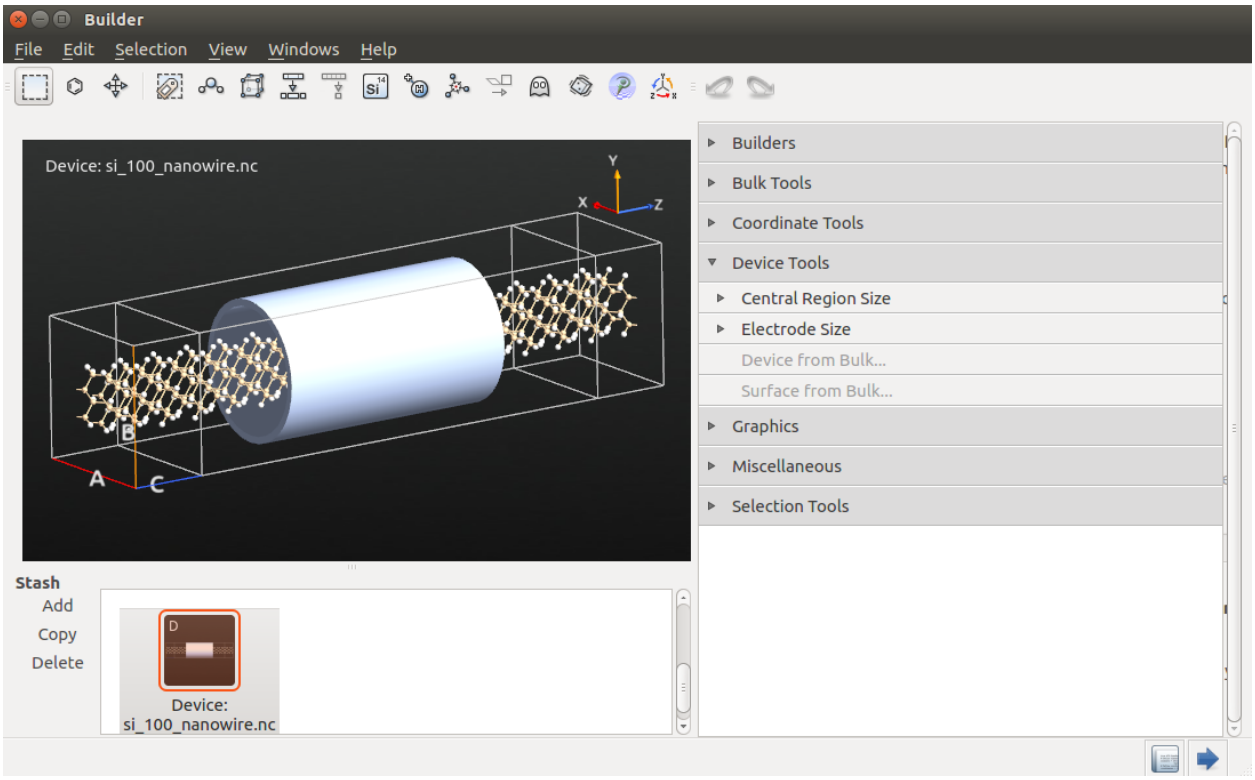
Next, use the *Miscellaneus* → *Spatial regions* tool to define the metallic wrap-around gate:

- First, add a new metallic region with a value of 0 Volt.
- Under *Geometry*, select **Tube** to create a cylindrical region.
- Define the geometry of the tube by entering the parameters shown in the image below. The cylinder will extend to the edge of the simulation cell along A and B, and will cover most of the central part of the nanowire along C.



Build the device

Use the *Device Tools* → *Device From Bulk* tool to build the Si(100) nanowire FET device from the bulk nanowire configuration. Use the default suggestion for the electrode lengths.



Then send the device configuration to the **Script Generator**.

Doping the Si(100) wire

You will now introduce doping in the nanowire. Instead of explicitly adding dopant atoms, which would result in a very high doping concentration for this relatively small device, you will set up the system as a **p-i-n junction**, by adding certain amounts of charge to the electrodes. This will lead to shifts in the Fermi levels of the two electrodes, resulting in a built-in potential in the device.

Use the mouse to select all atoms in the left electrode and open the *Miscellaneus* → *Doping* tool. In the window that shows up, choose the following settings:

•**Doping Type**: p-type

•**Value**: $4e^{+19}$

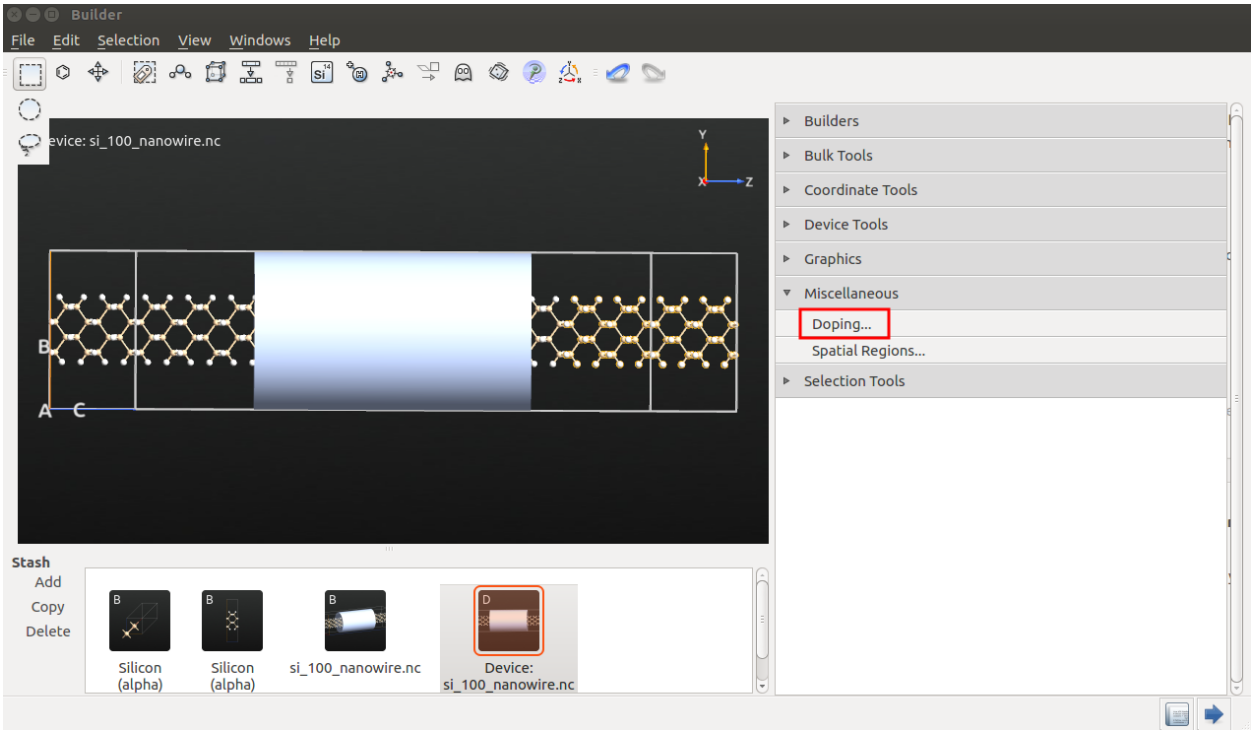
•**Unit**: e/cm^3 (left electrode)

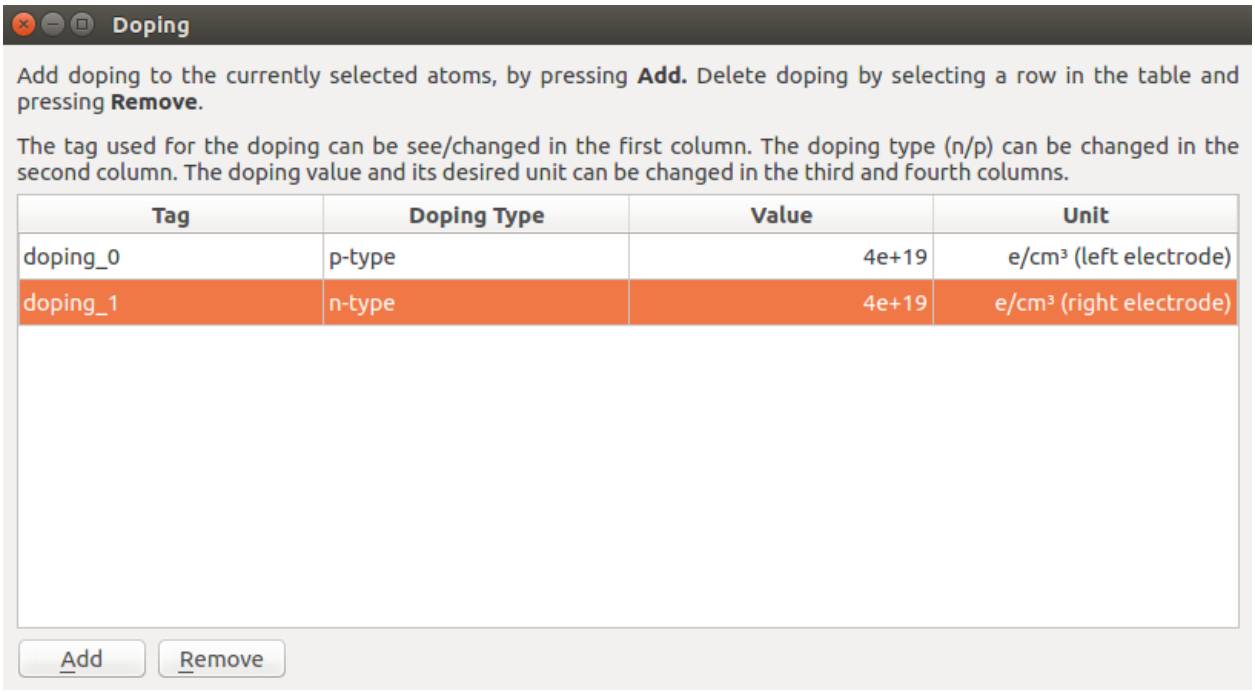
Next, close the Doping window and select all atoms in the right electrode. Open again the Doping tool, and choose the following settings:

•**Doping Type**: n-type

•**Value**: $4e^{+19}$

•**Unit**: e/cm^3 (right electrode)





Zero gate voltage calculation

You will now calculate the transmission spectrum of the p-i-n doped Si(100) nanowire FET device at zero gate potential using the Extended Hückel model. In the **Script Generator**, change the default output file name to `si_100_nanowire_fet_pin.hdf5` and add the following blocks to the script:

- New Calculator
- TransmissionSpectrum
- ElectronDifferenceDensity
- ElectrostaticDifferencePotential

Open the **New Calculator** block, select the Extended Hückel device calculator, and set up the calculator parameters similarly to what you did above for the bulk nanowire calculations:

- Uncheck No SCF iteration to make the calculation selfconsistent.
- Increase the density mesh cut-off to 20 Hartree.
- Go to the *Hückel basis set* tab and select the Hoffmann.Hydrogen and Cerda.Silicon [GW diamond] basis sets.
- Under *Poisson solver*, use Neumann boundary conditions in the A and B directions.

In the **TransmissionSpectrum** block, set the energy range to -4 to +4 eV with 301 points.

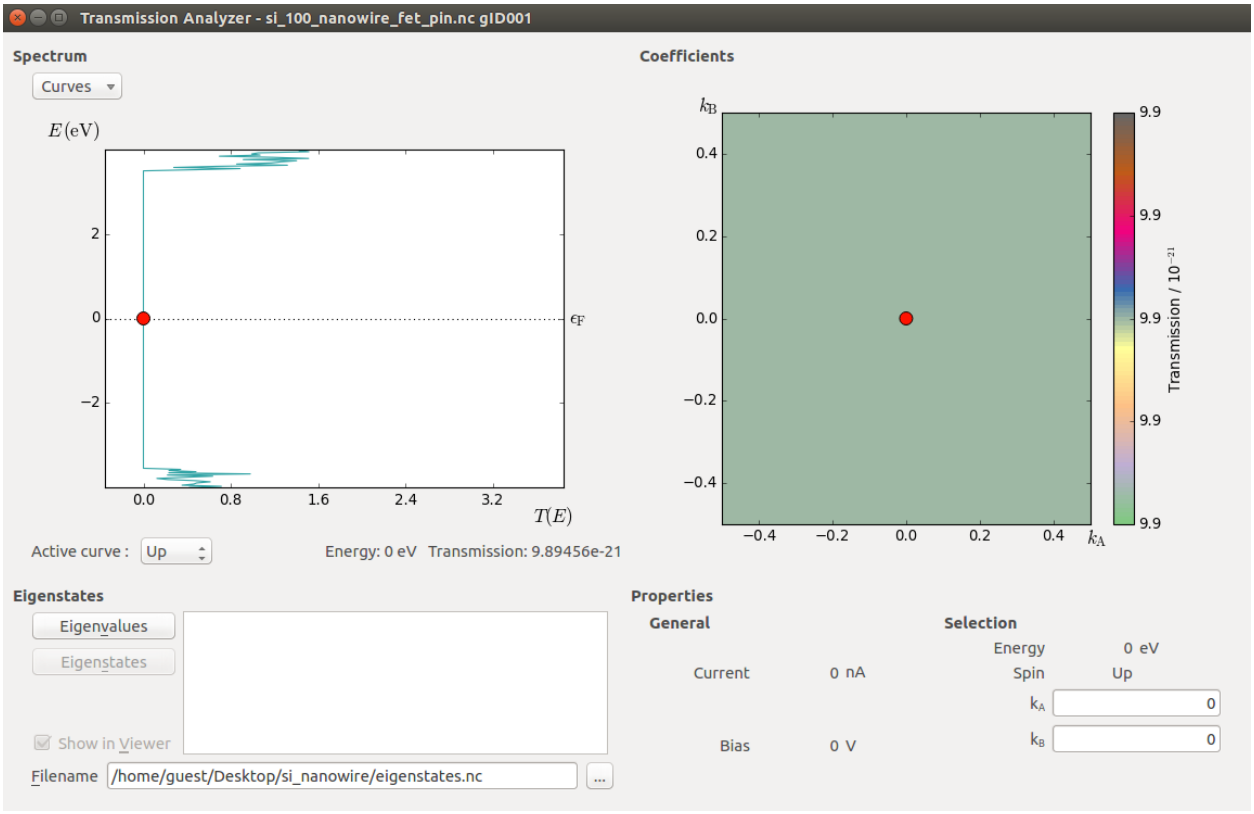
Save the QuantumATK Python script as `si_100_nanowire_fet_pin.py` and execute it using the **Job Manager**.

The calculation will take about 30 minutes if executed in parallel with 4 MPI processes.

Note: It would be physically incorrect to use periodic boundary conditions along the A and B directions when there are gates present in the system. We therefore choose Neumann boundary conditions along A and B instead.

Analyzing the results

Once the calculation is completed, locate the `file si_100_nanowire_fet_pin.hdf5` HDF5 file on the **Labfloor** and unfold it. Select the TransmissionSpectrum item and use the **Transmission Analyzer** to plot the transmission spectrum.



The transmission spectrum is shown above. The transmission is zero inside the nanowire band gap, which appears to be roughly 7 eV, significantly larger than computed earlier. This is due to the p- and n-doping, which has moved the valence and conduction band edges with respect to the Fermi level.

Performing a gate scan

Finally, you will calculate the transmission spectrum for different values of the gate bias. You can then plot the device conductance as function of gate bias.

The calculations are most conveniently done using QuantumATK Python scripting. Use the script `nanodevice_gatescan.py` for this. Download it to the active QuantumATK project folder, which also contains the file `si_100_nanowire_fet_pin.hdf5`. The script is reproduced below.

```
# Read in the old configuration
device_configuration = nlread("si_100_nanowire_fet_pin.nc",DeviceConfiguration)[0]
```

(continues on next page)

(continued from previous page)

```
calculator = device_configuration.calculator()
metallic_regions = device_configuration.metallicRegions()

# Define gate_voltages
gate_voltage_list=[-1, 0, 1.0, 2.0, 3.0, 4.0, 5.0, 6, 7, 8,]*Volt
# Define output file name
filename= "nanodevice_huckel.nc"

# Perform loop over gate voltages
for gate_voltage in gate_voltage_list:
    # Set the gate voltages to the new values
    new_regions = [m(value = gate_voltage) for m in metallic_regions]
    device_configuration.setMetallicRegions(new_regions)

    # Make a copy of the calculator and attach it to the configuration
    # Restart from the previous scf state
    device_configuration.setCalculator(calculator(),
         initial_state=device_configuration)
    device_configuration.update()
    nlsave(filename, device_configuration)

    # Calculate analysis objects
    electron_density = ElectronDifferenceDensity(device_configuration)
    nlsave(filename, electron_density)

    electrostatic_potential = ElectrostaticDifferencePotential(device_configuration)
    nlsave(filename, electrostatic_potential)

    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-4,4,301)*eV,
        kpoints=MonkhorstPackGrid(1,1),
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        )
    nlsave(filename, transmission_spectrum)
    nlprint(transmission_spectrum)
```

Execute the script using the **Job Manager** or from a command line. The script consists of 8 different calculations and it will take about 5 hours running in parallel with 4 MPI processes. You can, however, decide to reduce this time by reducing the number of gate voltages that the transmission spectrum is computed for – simply edit the variable `gate_voltage_list`.

Analyzing the gate scan

Download and execute the script `conductance_plot.py`, which reads the computed transmission spectra, calculates the conductance for each, and plots the conductance against gate bias.

```
# Read the data
transmission_spectrum_list = nlread("nanodevice_huckel.hdf5", TransmissionSpectrum)
configuration_list = nlread("nanodevice_huckel.hdf5", DeviceConfiguration)

conductance = numpy.zeros(len(configuration_list))
gate_bias = numpy.zeros(len(configuration_list))
for i, configuration in enumerate(configuration_list):
    transmission_spectrum = transmission_spectrum_list[i]
    energies = transmission_spectrum.energies().inUnitsOf(eV)
    spectrum = transmission_spectrum.evaluate()
    gate_bias[i] = configuration.metallicRegions()[0].value().inUnitsOf(Volt)
    conductance[i] = transmission_spectrum.conductance()

# Sort the data according to the gate bias
index_list = numpy.argsort(gate_bias)

# Plot the spectra
import pylab
pylab.figure()
ax = pylab.subplot(111)
ax.semilogy(gate_bias[index_list],  conductance[index_list])
ax.set_ylabel("Conductance (S)", size=16)
ax.set_xlabel("Gate Bias (Volt)", size=16)
ax.set_ybound(lower=1e-26)

for g,c in zip(gate_bias[index_list],  conductance[index_list]):
    print(g,c)
pylab.show()
```

3.5.10 Exploring Graphene

In this tutorial you will build a range of graphene structures and study some of their properties. With QuantumATK you can easily twist and stretch the structures using plug-in modules. You can then use QuantumATK to calculate their electronic and vibrational properties. You will also learn to automate calculations on large structures using Python scripting.

Hint: In many QuantumATK projects (depending on the study of course) you will need the appropriate plugins/AddOns, which can be installed via the **AddOn Manager**.

In this particular tutorial, you will need the `TubeWrapper.zip` and **CNTBuilder** plugins. Please check that these are available, and install them if they are not. You can install addons following the instructions in the *AddOns* page.

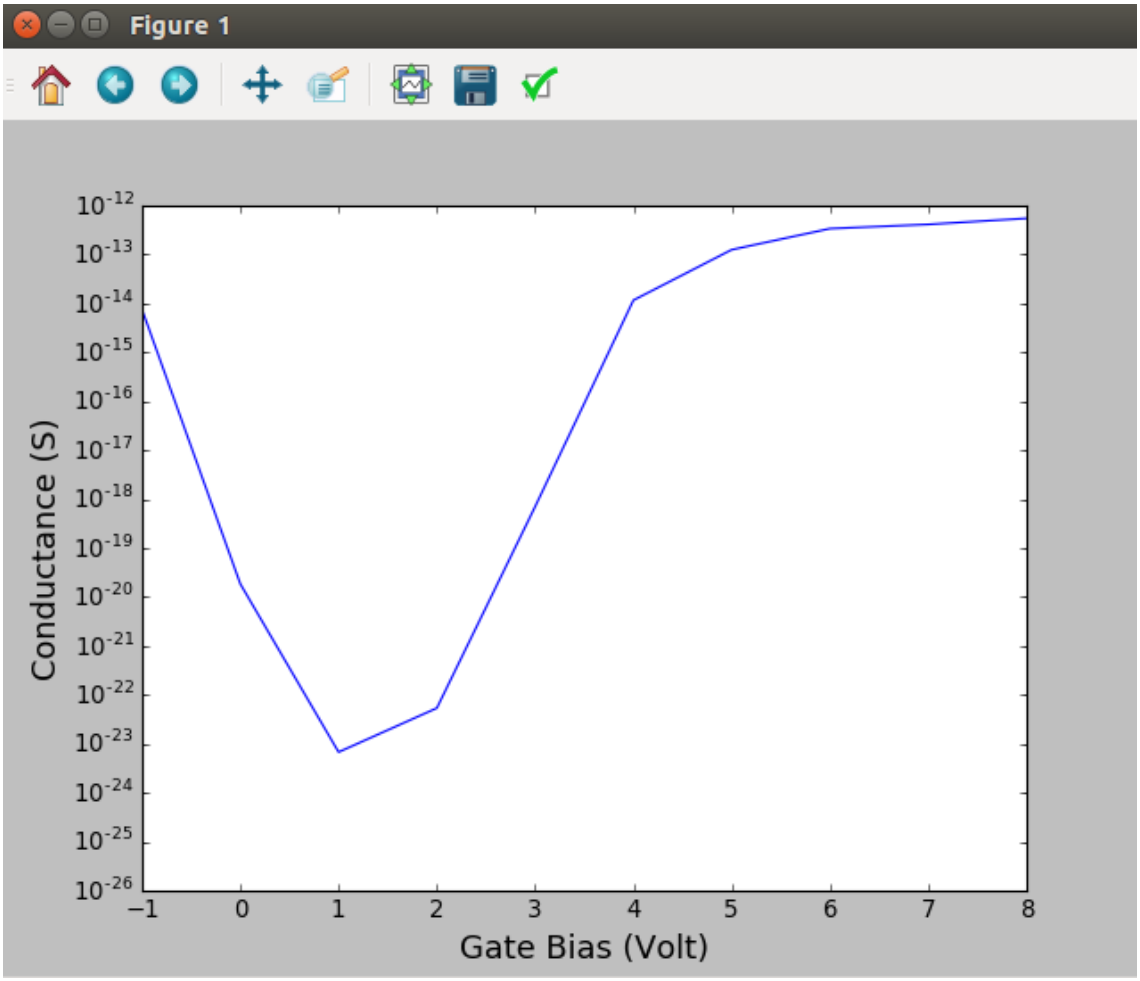
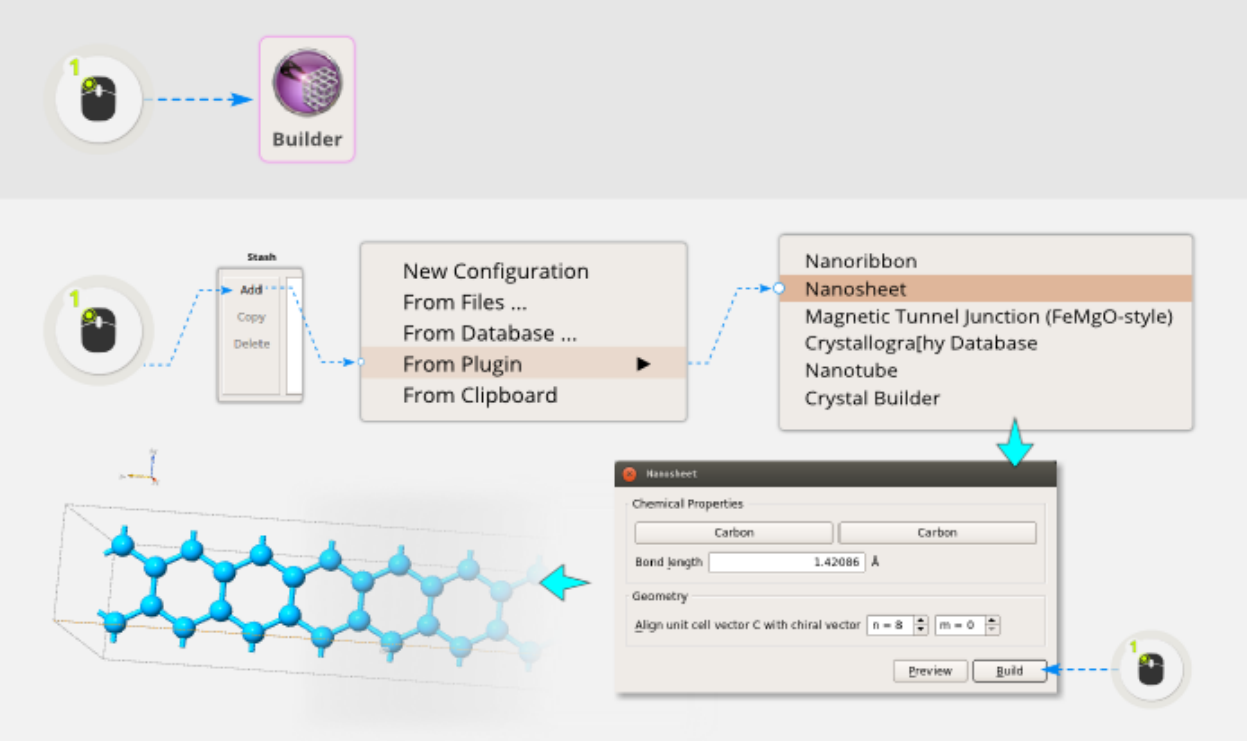


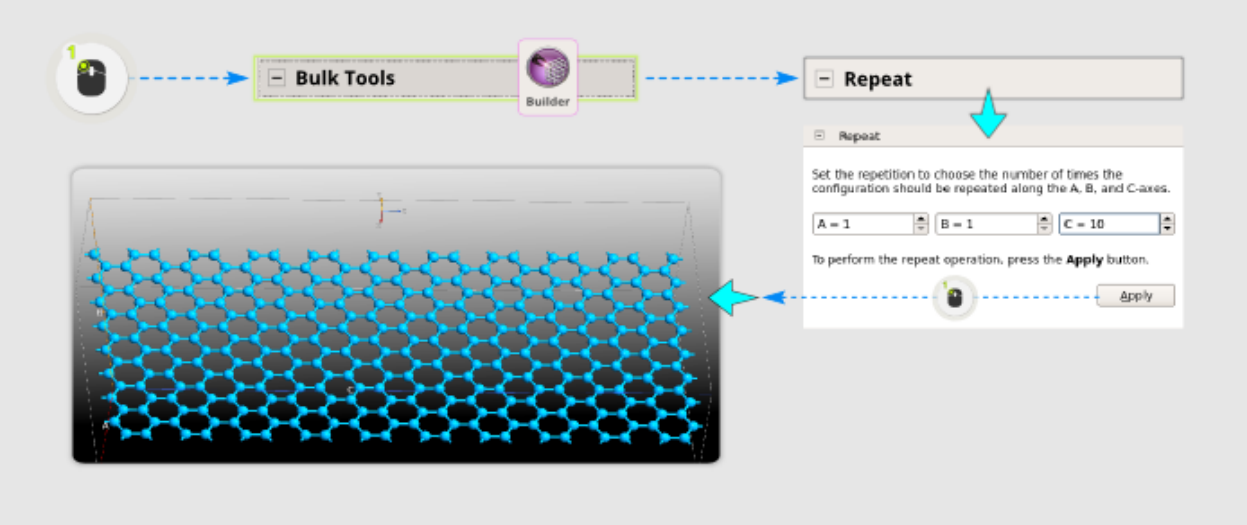
Figure 1
Conductance (S)
Gate Bias (Volt)
10-12
10-13
10-14
10-15
10-16
10-17
10-18
10-19
10-20
10-21
10-22
10-23
10-24
10-25
10-26
−1
0
1
2
3
4
5
6
7
8

Build a graphene sheet

Open the QuantumATK Builder . It is then easy to create a graphene sheet using a plugin; simply click *Add* → *From Plugin* → *Nanosheet*. Leave the option for Chemical Properties at defaults. In the Geometry options, choose a chiral vector of (n,m)=(8,0), and click *Build*.

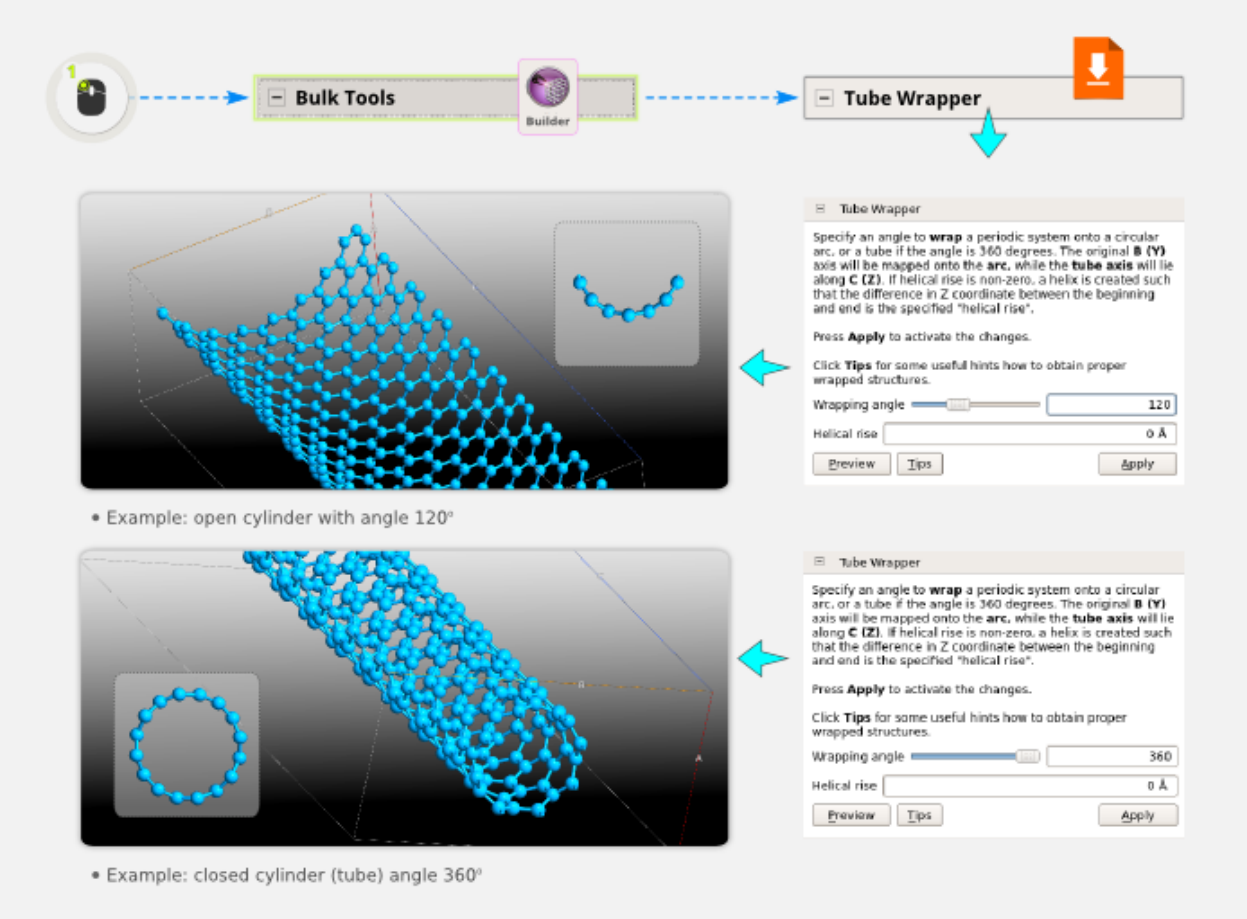


You now have a basic building block for the structure. Use the *Bulk Tools* → *Repeat* plugin to repeat it 10 times along the C-direction:



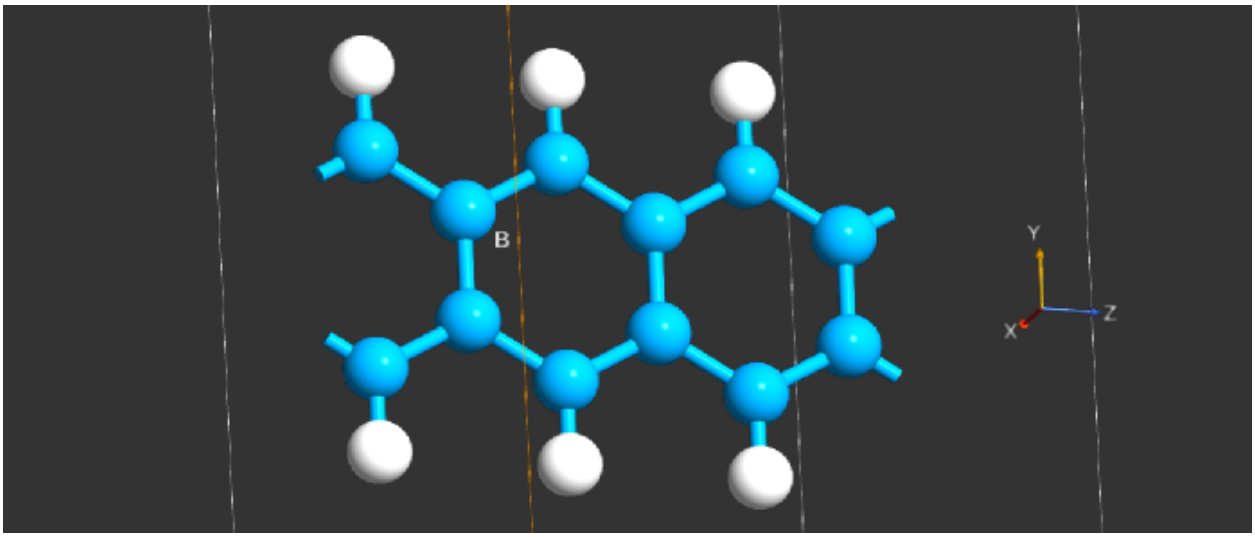
Build a CNT

You can now use the TubeWrapper plugin to wrap the nanosheet into a carbon nanotube (CNT). It is possible to turn the nanosheet into an open or a closed cylinder:



Transmission spectrum of a GNR

Let us calculate the electronic transmission spectrun of a graphene nanoribbon (GNR). First, build the ribbon using the *Add* → *From Plugin* → *Nanoribbon* plugin. Choose chiral indices (n,m)=(1,1) and repeat the structure 3 times the C-direction:



Note: In order to correctly calculate the transmission spectrum for a bulk system, the bulk must qualify as a valid device electrode. This requires that the C-axis is perpendicular to the A,B plane, and that it is sufficiently long that the atoms in the unit cell only have Hamiltonian matrix elements with atoms in the nearest neighbour cells along C. This condition is usually fullfilled if the C-vector is longer than 7 Å.

Next, transfer the structure to the Script Generator using the botton, and double-click the **New Calculator** icon to add a calculator block to the script.

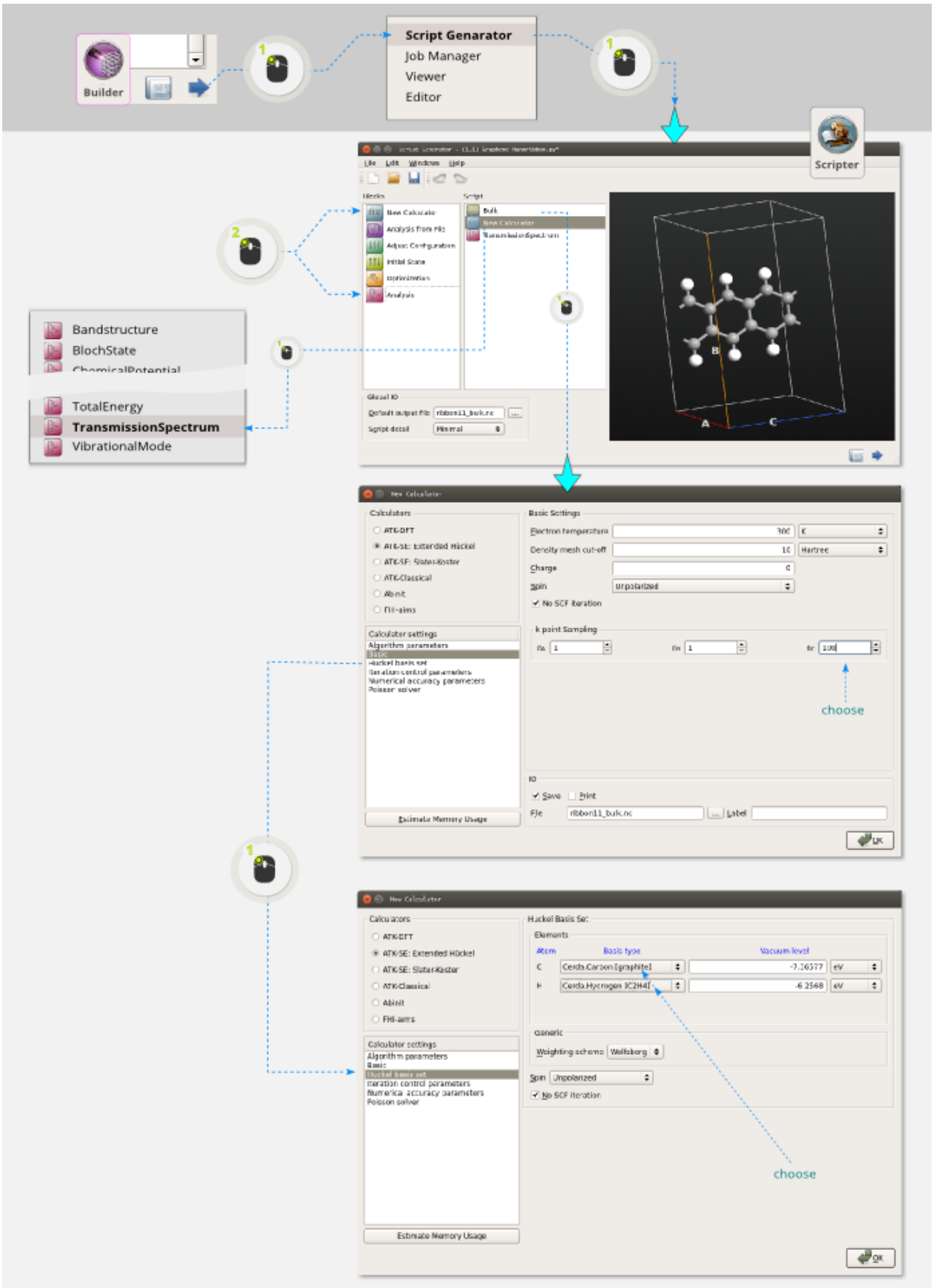
•First, change the default output file to `ribbon11_bulk.hdf5`.

•Then open the added calculator block to start editing the settings.

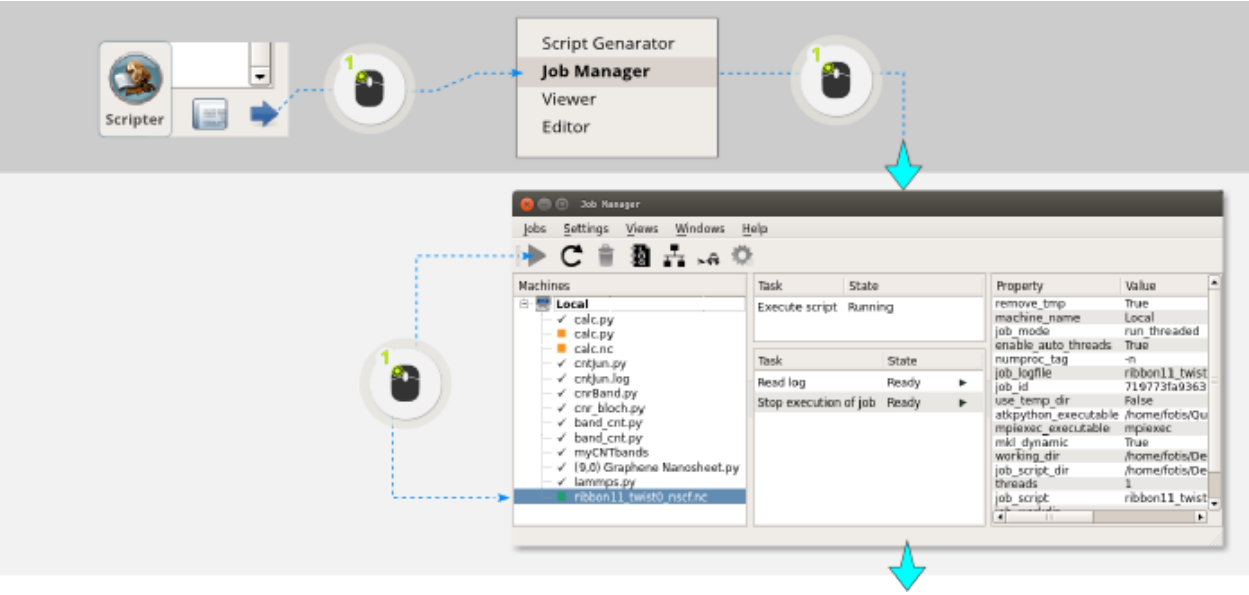
•Select the **Extended Huckel** calculator, and make sure the number of k-points is 1 for the A and B directions, and high for the C-direction, e.g. 100 k-points along C.

•In the **Huckel basis set** options, select *Cerda.Carbon [graphite]* for carbon and *Cerda.Hydrogen [C2H4]* for hydrogen.

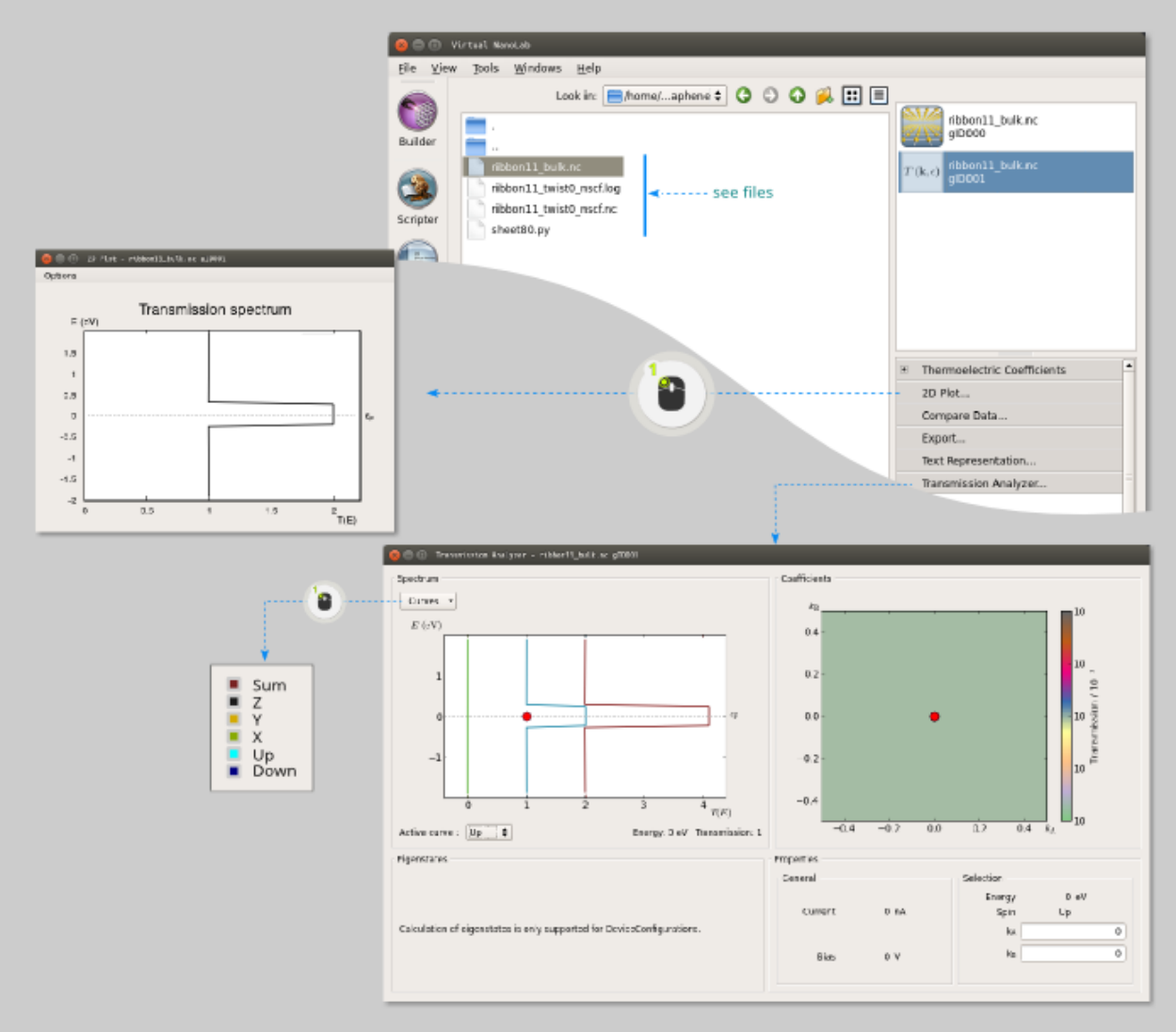
•Finally, add a **TransmissionSpectrum** block to the script, and save the Python script as `ribbon11_bulk.py`.


Builder
Script Genarator
Job Manager
Viewer
Editor
Scripter
Bandstructure
BlochState
TotalEnergy
TransmissionSpectrum
VibrationalMode
choose
choose

Send the script to the **Job Manager** in order to execute it. The job starts when you click the icon.



When the calculation is finished (it will run extremely fast), locate the output file **ribbon11_twist0_nscf.hdf5** in the QuantumATK file browser window. Select the file and notice that the contents of the file are displayed in the QuantumATK panel:

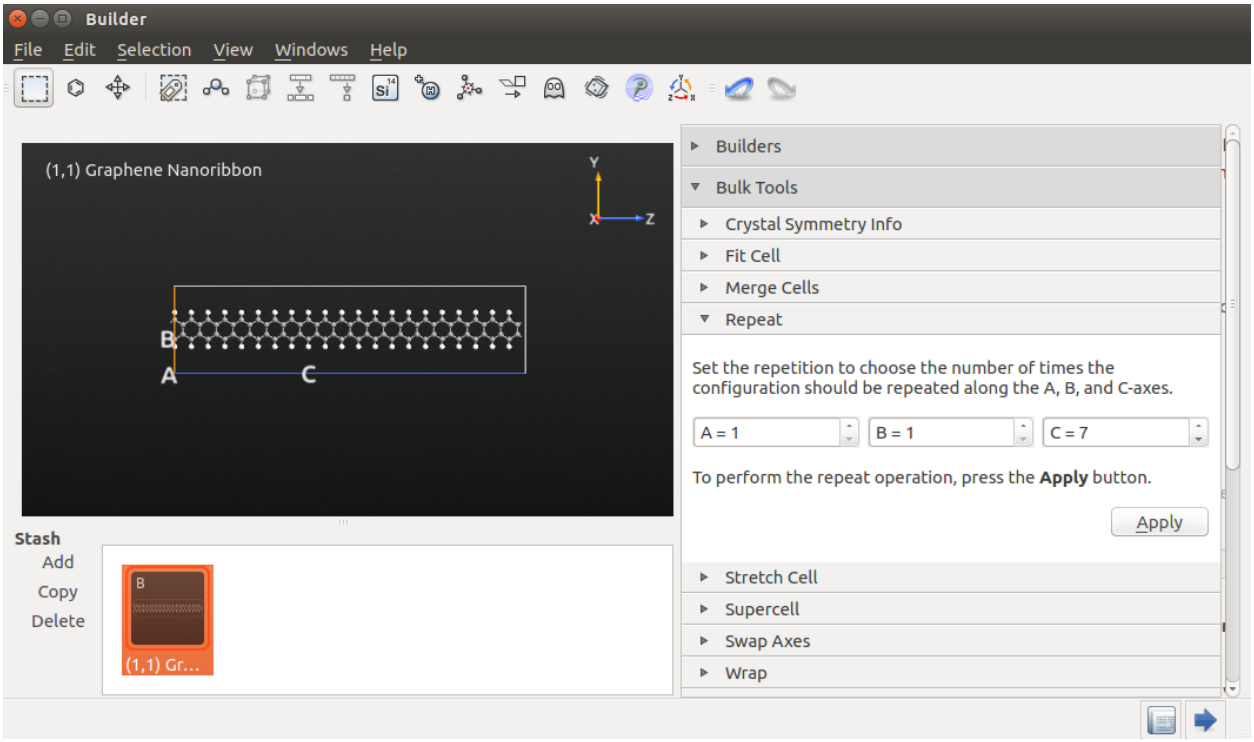


You can now select the TransmissionSpectrum object, which has this icon: $T(\mathbf{k}, \epsilon)$ and use the **2D Plot** or **Transmission Analyzer** plugins to plot the electronic transmission spectrum.

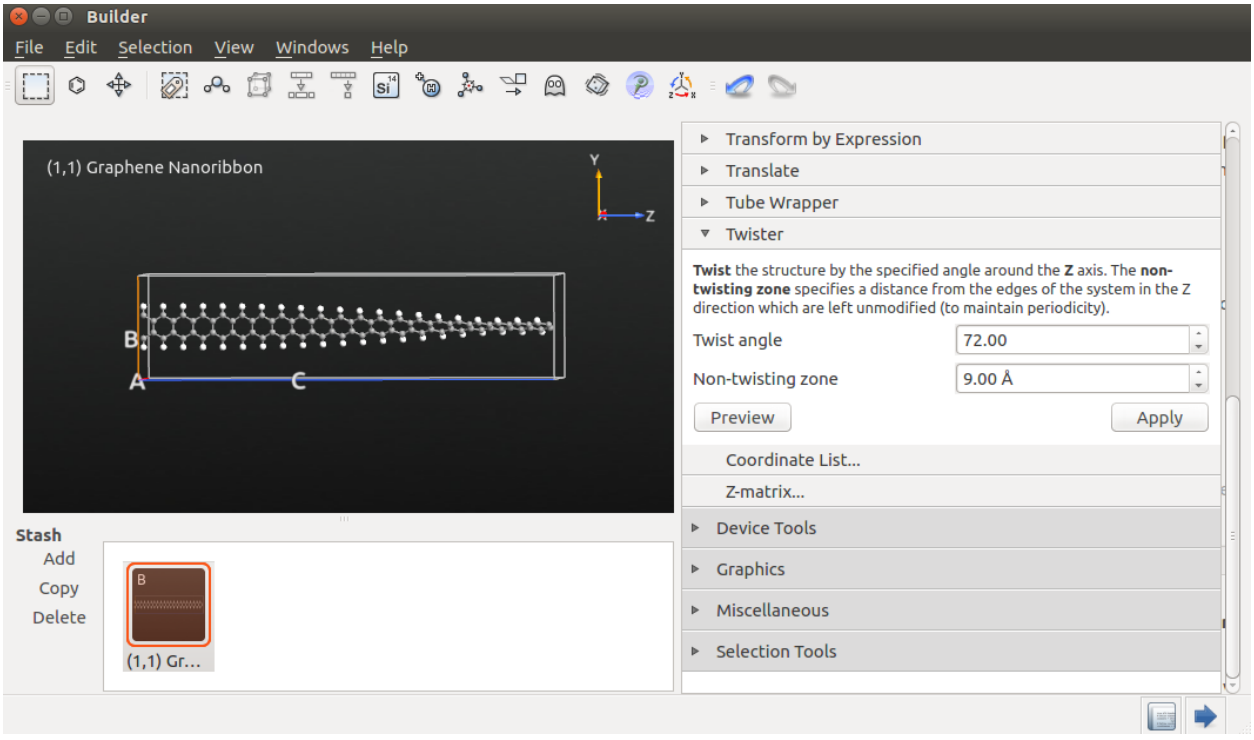
Twisted nanoribbon

In this section you will learn how to twist a graphene nanoribbon using the QuantumATK **Twister** plugin, and then compute the transmission spectrum.

Open the Builder, and select the nanoribbon created in the previous section. Then use the *Bulk Tools* → *Repeat* tool to repeat the configuration 7 times in the C direction:



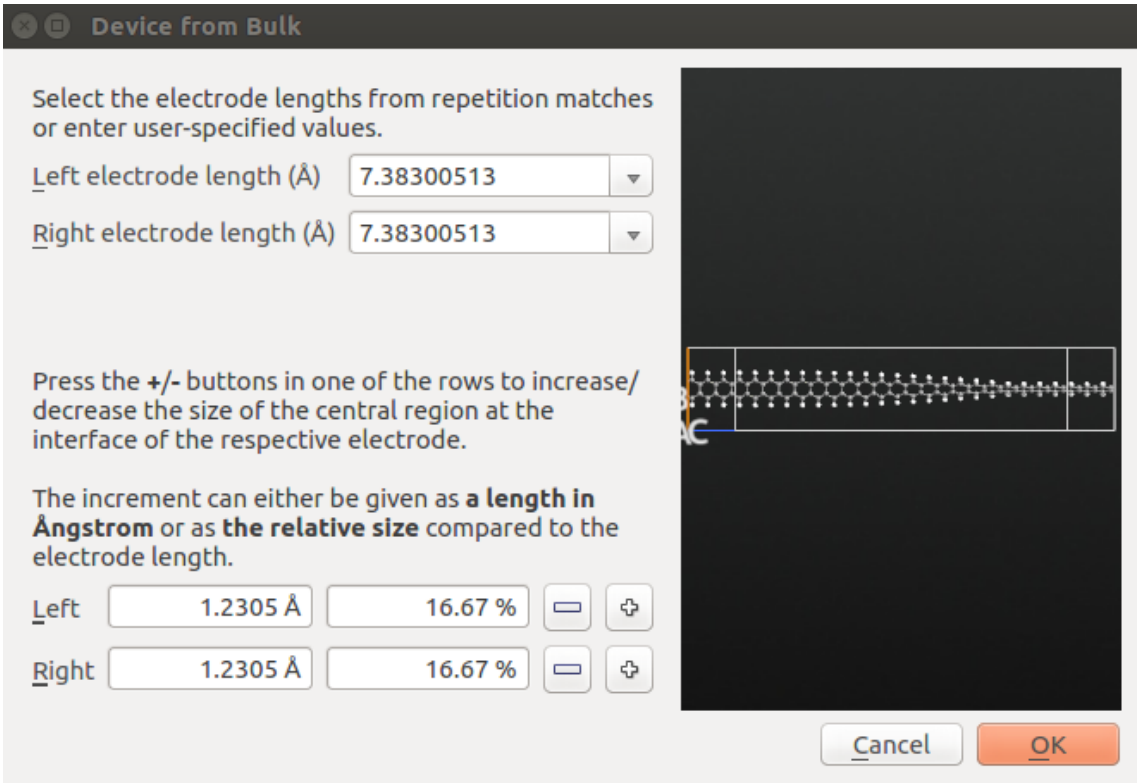
Then open the *Coordinate Tools* → *Twister* plugin, and set a Twist angle of 72 degrees and a Non-twisting zone of 9 Å. Click **Apply** to perform the twisting operation.



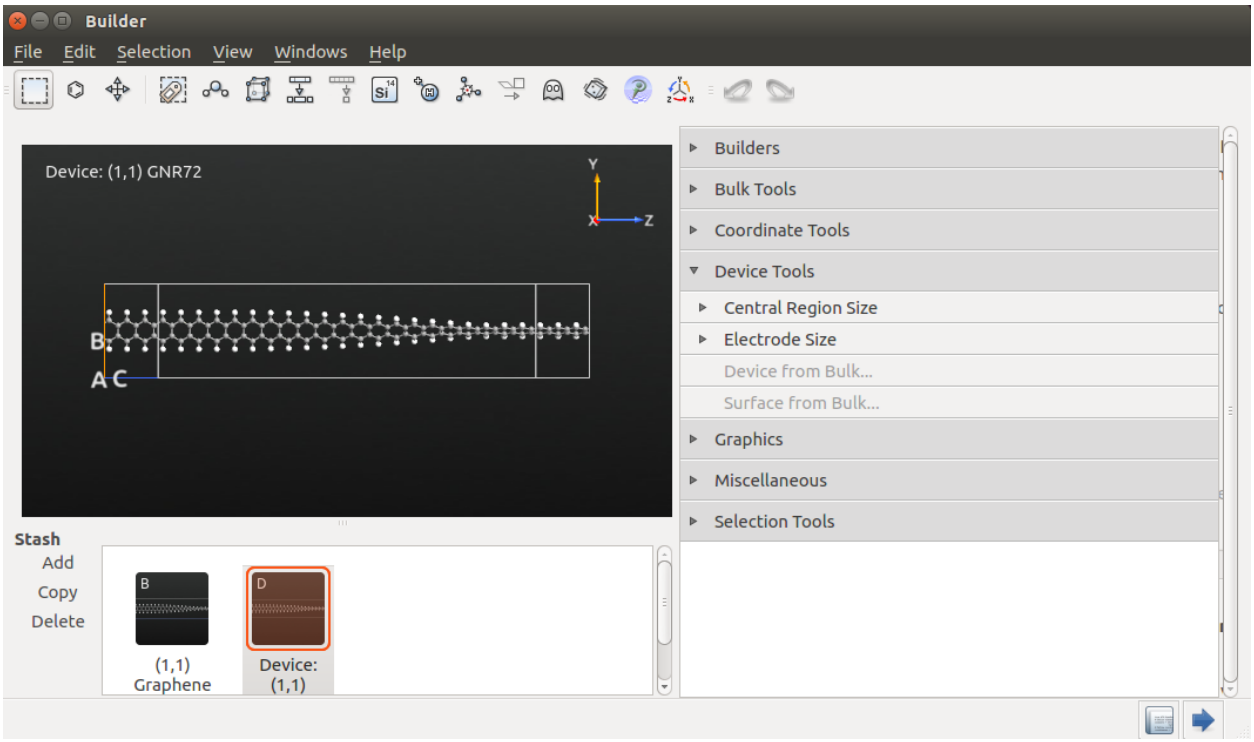
Certain parts of the system to the left and right of the structure are not twisted; the length of the non-twisted part is specified by the value of Non-twisting zone. The rest of the structure is twisted by the sepcified angle.

Transmission spectrum

You will now calculate the transmission spectrum of the twisted graphene nanoribbon. First, you need to convert the twisteds ribbon into a device – use the *Device Tools* → *Device from Bulk* tool for this.



Then rename the device configuration as `GNR_twist` and send it to the **Script Generator** to create the required ATK Python script.



In the Scripter, add the following script blocks:

- New Calculator
- Transmission Spectrum

Open the **New Calculator** block and use the following settings:

•ATK-SE: Extended Hückel (Device) calculator;

•1x1x100 k-point grid;

•Cerda.Carbon [graphite] basis set for C and Cerda.Hydrogen[C2H4] basis set for H.

Select the following settings in **TransmissionSpectrum** block:

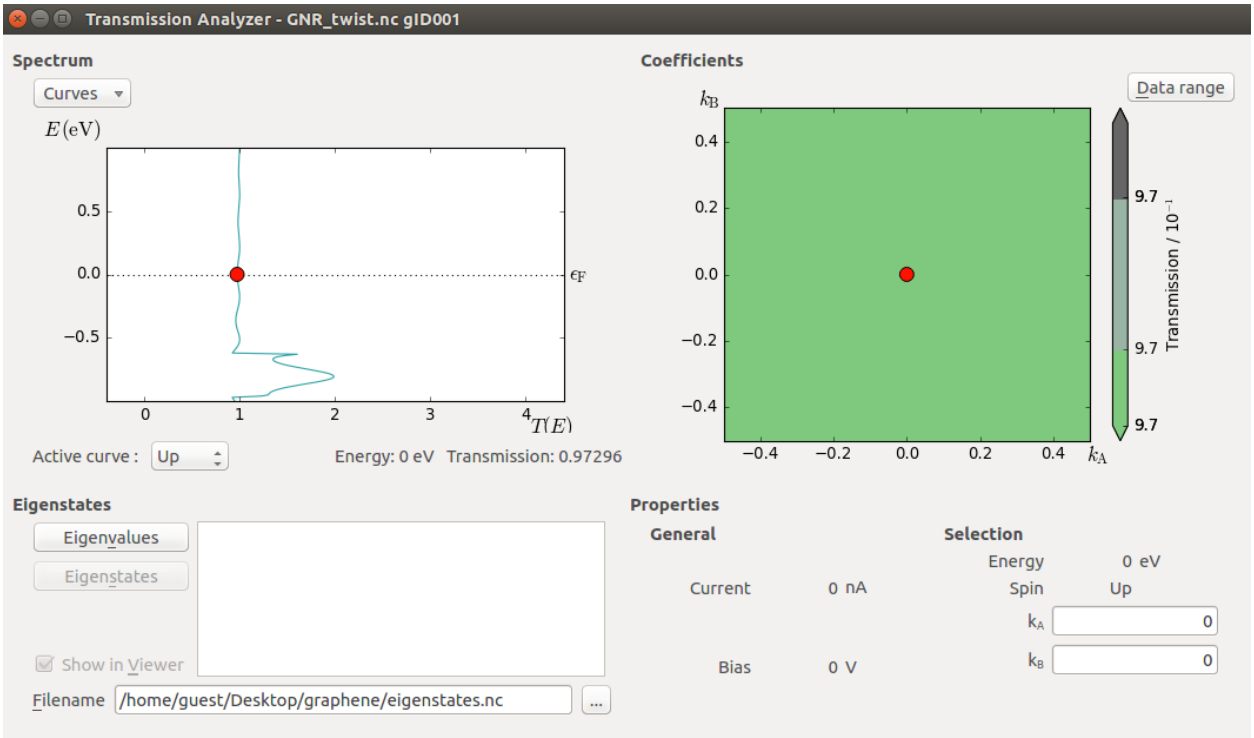
•energy range of [-1,1] eV and 201 energy points;

•Krylov self-energy calculator;

•1x1 k-point grid.

Then save the script as `GNR_twist.py` and execute it using the **Job Manager**. This calculation will take just a few minutes.

The HDF5 data file `GNR_twist.hdf5` should now have appeared on the QuantumATK **LabFloor**. Select the *TransmissionSpectrum* item and use the **Transmission Analyzer** plugin to visualize the computed transmission spectrum.



Note that the transmission peak has shifted down in energy as compared to the non-twisted ribbon. The twist represents a source of electron scattering, since it breaks the translational symmetry of the non-twisted ribbon electrodes along the transport direction.

Möbius nanoribbon

Next, we are going to create a Möbius nanoribbon. Download the script `Moebius.py`, which is also reproduced below. The script loads a basic nanoribbon from the file `nanoribbon.hdf5`, then performs the required operations of repeating, twisting, and wrapping the ribbon, and finally saves the Möbius ribbon in `moebius.hdf5`.

```
from NL.Math.Utilities import rotationMatrix
import math
from QuantumATK import *

def twister_displacement(x, rotation_angle_per_z, rotation_axis,
                         rotation_axis_center, z_start, z_end):
    """
       Function for twisting a 1-d structure
       @param x : Coordinates of 1-d structure
       @param rotation_angle_per_z : size of twist in angle/length
       @param rotation_axis : axis to apply twist along
       @param rotation_axis_center : center of the rotation axis
       @param z_start : z value for starting the twist
       @param z_end : z value for ending the twist
    """

    # do not twist for z > z_end
    z = x[2]
    z = min(z,z_end)
```

(continues on next page)

(continued from previous page)

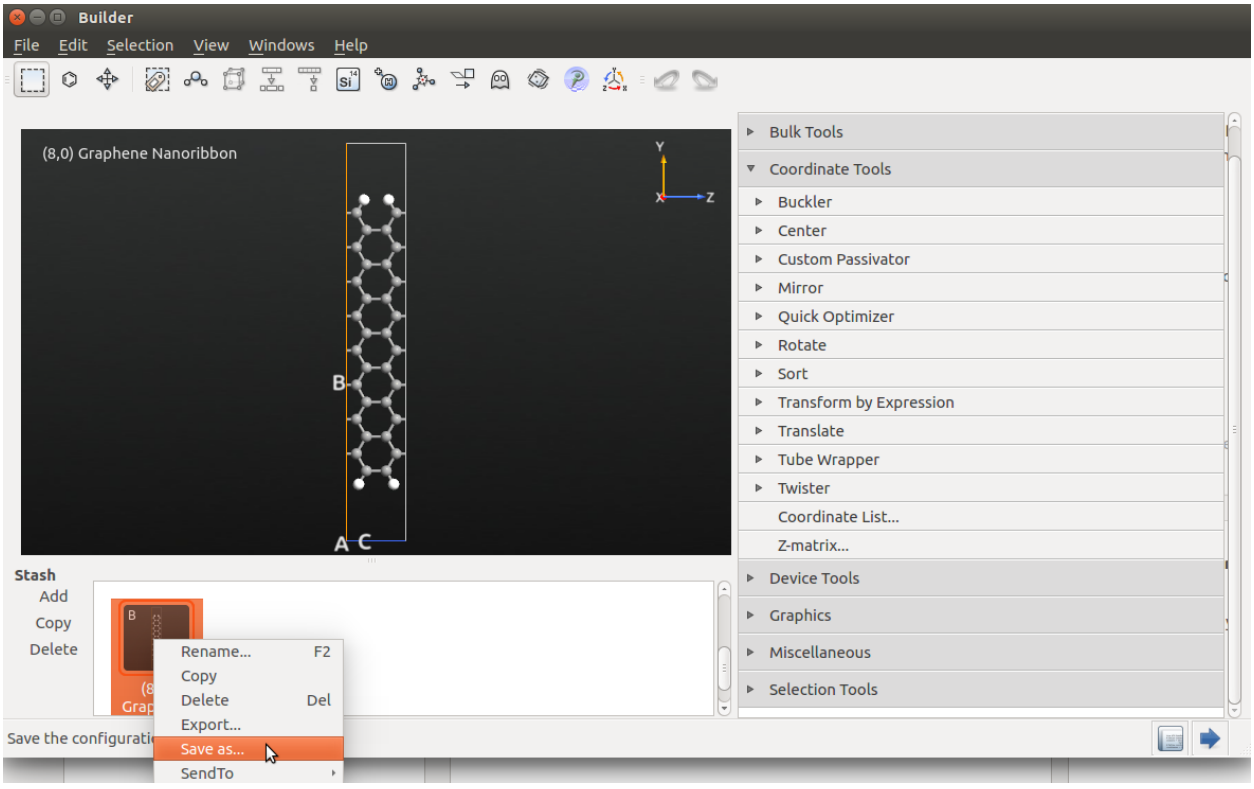
```
    # do not twist for z < z_start
    z = z - z_start
    z = max (z,0.0)
    # find twist angle
    theta = z*rotation_angle_per_z
    # calculate the rotation matrix
    rotation_matrix = rotationMatrix(theta, *rotation_axis)
    # apply rotation
    return rotation_axis_center+numpy.dot(rotation_matrix, x-rotation_axis_center)

def wrapping_displacement(x, width, wrapping_angle):
    """
       Function for converting a nanosheet coordinate into a partly wrapped nanotube
       @param x : Coordinates of nanosheet atom
       @param width : Width of the nano-sheet
       @param wrapping_angle : maximum wrapping angle of the nanotube in radians
    """
    # calculate the average radius of the incomplete wrapped tube
    radius = width/wrapping_angle
    # find the angle of the current atom
    angle = (x[2]-width/2.)/radius
    # calculate the radius of the current atom
    atom_radius = radius+x[1]

    # return atom position of the wrapped atom
    return numpy.array([x[0], atom_radius*math.cos(angle),atom_radius*math.sin(angle)])

def Moebius(ribbon, n, m, repetition):
    """
       Function for generating a moebius molecule
       @param n : Chiral vector index
       @param m : Chiral vector index
       @param repetition : Repetition along z
    """

    # build n,m ribbon
    #ribbon = NanoRibbon(n,m)
    ribbon = ribbon.repeat(1,1,repetition)

    # get properties of the ribbon
    lattice = ribbon.bravaisLattice()
    elements = ribbon.elements()
    cartesian_coordinates=ribbon.cartesianCoordinates().inUnitsOf(Angstrom)

    # calculate the length of the 1-d structure
    z_length = numpy.linalg.norm(lattice.primitiveVectors()[2].inUnitsOf(Angstrom))

    # calculate twist parameters
    rotation_angle_per_z =  math.pi /z_length
    rotation_axis = numpy.array([0,0,1])
    rotation_axis_center = numpy.sum(cartesian_coordinates,axis=0)/len(cartesian_
↪coordinates)
```

(continues on next page)

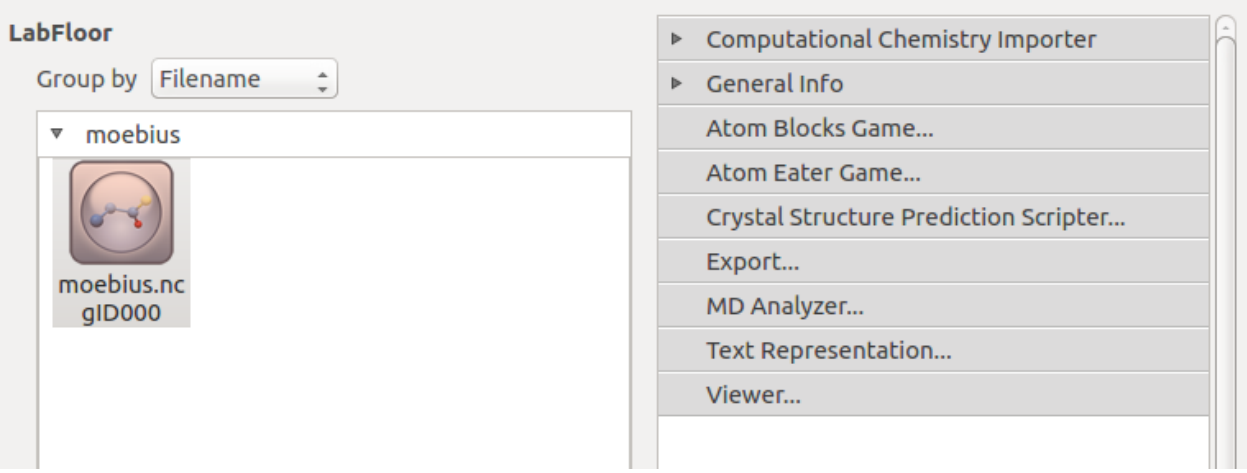
(continued from previous page)

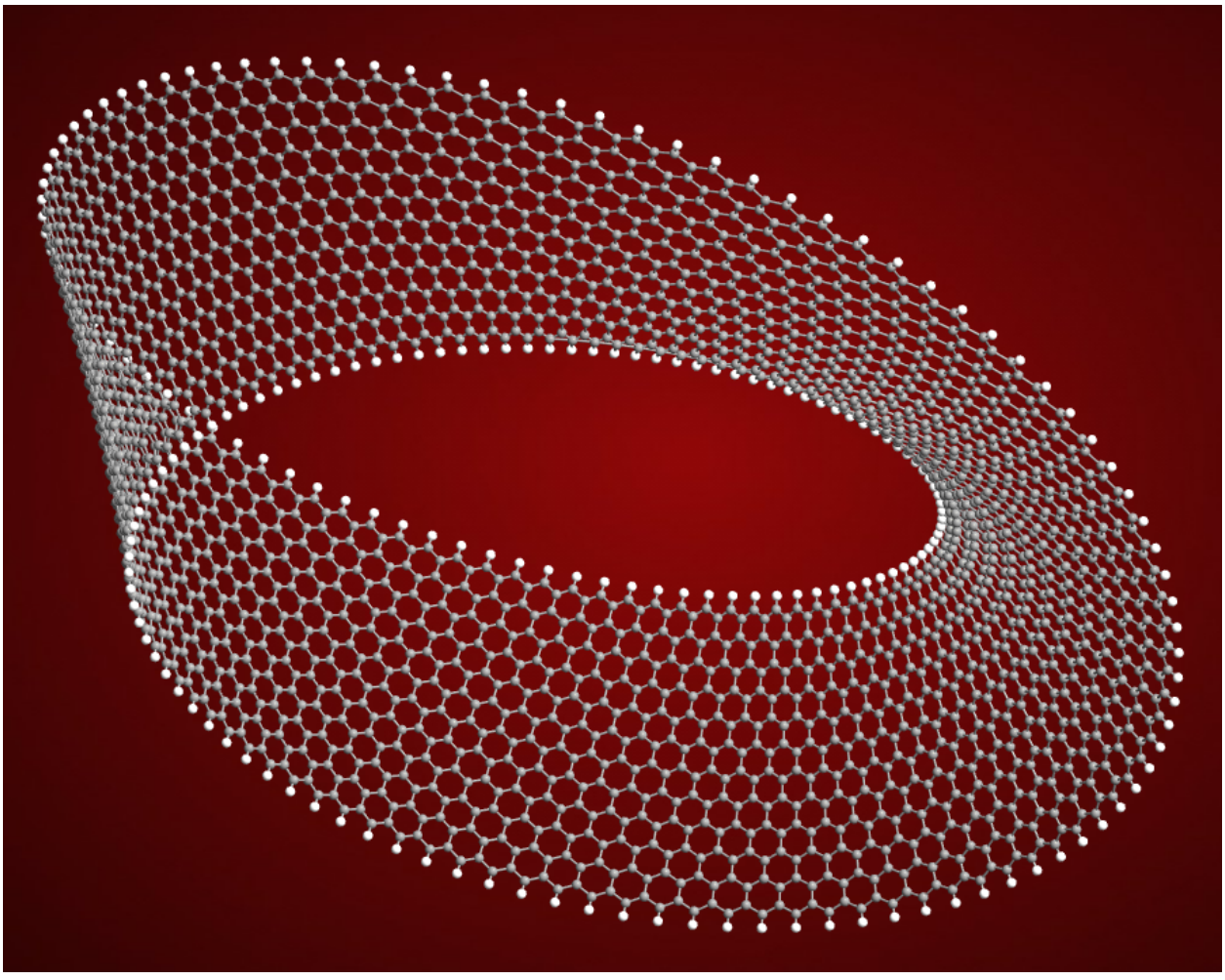
```
    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : twister_displacement(c, rotation_angle_per_z, rotation_axis,
                                rotation_axis_center, 0.,z_length)
    # apply the function to find new displaced coordinates
    cartesian_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)
    cartesian_center = numpy.sum(cartesian_coordinates,axis=0)/len(cartesian_coordinates)
    cartesian_coordinates = cartesian_coordinates - cartesian_center


    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : wrapping_displacement(c, z_length,2.0*math.pi)
    # apply the function to find new displaced coordinates
    cartesian_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)

    return MoleculeConfiguration(
            elements=elements,
            cartesian_coordinates=cartesian_coordinates * Angstrom
            )

ribbon = nlread('ribbon.nc', BulkConfiguration)[-1]
moebius = Moebius(ribbon,1,1,100)
nlsave('moebius.nc', moebius)
```

You first need to build a basic nanoribbon. Use the *Add* → *From Plugin* → *Nanoribbon* plugin and select, for example, the chiral vector (n,m)=(6,6). Save the created ribbon in the Project Folder as `ribbon.hdf5`.



Then make sure `Moebius.py` is also located in the Prpject Folder, and execute it using the Job Manager or from the command line. The output file `moebious.hdf5` should appear in the Project Files list and the Möbius configuration should appear on the LabFloor. Use the **Viewer** to visualize the structure.





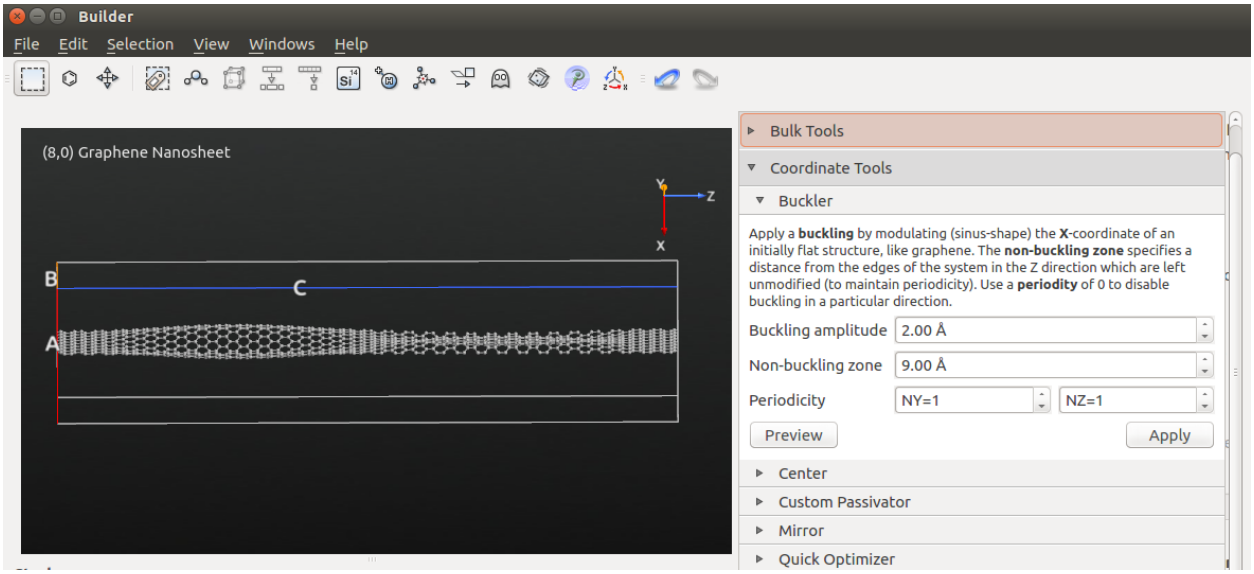
Buckling a graphene sheet

In this final section you will learn how to buckle a graphene sheet using the QuantumATK **Buckler** plugin.

First, create a nanosheet as shown above in section *Build a graphene sheet*. This time, repeat the nanosheet 30 times in the C direction.

Then open the *Coordinate Tools* → *Buckler* plugin, and set the Buckling amplitude to 2 Å and the Non-buckling zone to 9 Å. Select NY = 1 and NZ = 1 to enable buckling along the Y and Z axes.

Click **Apply** to perform the buckling operation.



3.6 2D Materials

3.6.1 Meta-GGA and 2D confined InAs

Version: 2017

In this tutorial you will learn about two topics: (i) bandstructure calculations by density functional theory (DFT) with the meta-generalized- gradient approximation (meta-GGA) exchange-correlation (XC) functional of Tran & Blaha (TB09) and (ii) calculations of confined structures with focusing on how to passivate dangling bonds on the surface. After completing the tutorial, you will be able to:

- Perform calculations with the TB09 meta-GGA XC functional to obtain accurate band gaps of semiconductors;
- Set up and passivate confined III-V systems (InAs, GaAs, InP,);
- Use special pseudopotentials with non-integer charges for the passivating hydrogens, that is, artificial hydrogen atoms or pseudo-hydrogens, which are needed in order to passivate III-V and II-VI surfaces since normal hydrogen passivation leads to metallic surface states;
- Use compensation charges on the passivating (normal) hydrogen atoms as an alternative to using pseudo-hydrogens.

Note: If you are not yet familiar with QuantumATK, go through the basic tutorials first. The underlying calculation engines for this tutorial is ATK-DFT, about which one can find the details in QuantumATK Reference Manual.

TB09 meta-GGA

Short introduction to TB09 meta-GGA

DFT is infamous for predicting too small band gaps, when calculated from the Kohn-Sham energy eigenvalues. However, recent advances of meta-GGA functionals have enabled us to compute accurate band gaps.

Meta-GGA functionals belong to the third rung of the so-called Jacobs ladder of XC functionals, as they not only include the local density $\rho(\mathbf{r})$ (as in LDA; the first rung) and the gradient of the density $\nabla\rho(\mathbf{r})$ (as in GGA; the second rung), but also the kinetic-energy density $\tau(\mathbf{r})$. In 2009, Tran and Blaha[1] showed that accurate band gaps could be obtained for a broad range of materials. In their formulation, the exchange potential is given by

$$v_x^{TB}(\mathbf{r}) = cv_x^{BR}(\mathbf{r}) + \frac{3c-2}{\pi}\sqrt{\frac{4\tau(\mathbf{r})}{6\rho(\mathbf{r})}},$$

where $\tau(\mathbf{r}) = 1/2\sum_{i=1}^{N}|\nabla\psi_i(\mathbf{r})|^2$ is the kinetic-energy density, $\psi_i(\mathbf{r})$ is the ith Kohn-Sham orbital, and $v_x^{BR}(\mathbf{r})$ is the Becke-Roussel exchange potential[2]. The parameter c is calculated as

$$c = \alpha + \beta\left[\frac{1}{\Omega}\int_\Omega \frac{|\nabla\rho(\mathbf{r})|}{\rho(\mathbf{r})}d\mathbf{r}\right]^{1/2},$$

where $\alpha = -0.012$ and $\beta = 1.023\ \mathrm{Bohr}^{1/2}$ were determined by fitting to reproduce the experimental band gaps of a large number of semiconductors and insulators and Ω is the volume of the cell. The correlation potential used by Tran and Blaha is an ordinary LDA correlation.

TB09 meta-GGA in QuantumATK

In **ATK** you can use the XC functional by Tran and Blaha in two ways. You can specify the XC functional as

```
exchange_correlation = MGGA.TB09LDA
```

in which case the c-parameter is determined self-consistently based on the expression above. This is the default, and the way the script is set up by **Script Generator** in **QuantumATK**.

You can also set the value of the c-parameter manually:

```
exchange_correlation = MGGA.TB09LDA(c=1.0)
```

The electron and kinetic-energy densities are still calculated fully self-consistently, but using the fixed value of c.

If the c-parameter is calculated self-consistently, the value will be printed in the log file. It can also be extracted in a script as

[1] F.Tran, and P. Blaha, Phys. Rev. Lett., **102**, 226401, (2009)

[2] A.D. Becke, and M.R. Roussel, Phys. Rev. A., **39**, 3761, 1989

```
c = calculateTB09C(bulk_configuration)
```

after the self-consistent calculation.

Note: Using a self-consistent determination of the c-parameter should only be done for a **bulk configuration** that is periodic in all directions. A confined system such as a slab or a nanowire will include large regions of vacuum. Since the c-parameter is calculated as an integral over the whole unit cell volume, the contributions from the vacuum region will lead to incorrect, even diverging results. You will most likely see warning messages in the log file if you attempt this .

For confined systems one must first determine an appropriate value for c for the corresponding bulk system. This can be done either self-consistently or by fitting the c-parameter to, e.g., the experimental band gap (see *Fitting the TB09 meta-GGA c-parameter*). The so-determined c-parameter can then be used for the confined structure.

Bulk InAs band structure with TB09 meta-GGA

Setting up bulk InAs

You should now setup an InAs bulk crystal. In **Builder**, click **Add** → **Database**, type *InAs* in the search field, and add the structure to the Stash.

The next step is to change the lattice constant to the experimental room temperature lattice constant, $a = 6.0583$[3]

- In the panel to the right, open **Bulk tools** → **Lattice Parameters**
- Importantly, you must select to keep **fractional** coordinates constant (see the figure below), else the symmetry of the crystal will be broken when the lattice constant is changed.
- Set $a = 6.0583$ Å.

Next, send the structure to **Script Generator**.

Setting up the script

In the following you will calculate the band structure of bulk InAs using the TB09 meta-GGA functional with the c-parameter calculated self-consistently and non-self-consistently.

For this purpose:

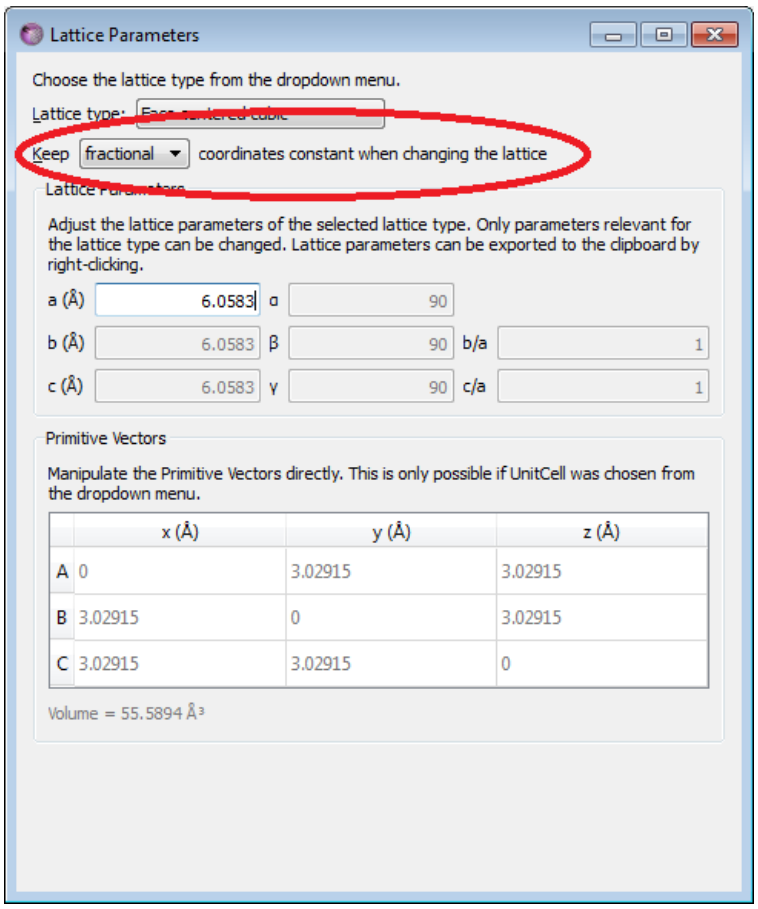
- Add **New Calculator** and **Bandstructure** blocks.

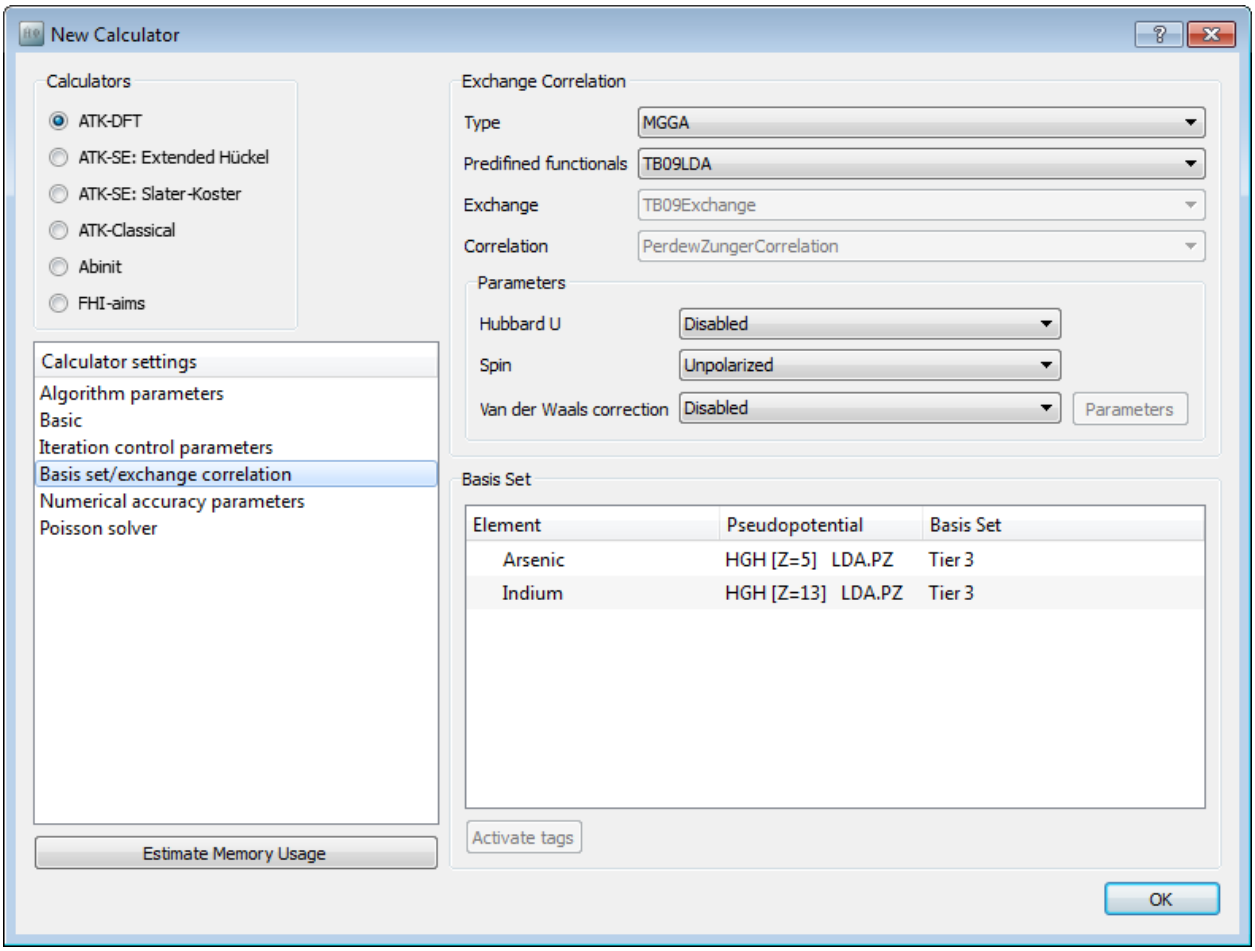
Open the **New Calculator** block, and make the following changes:

- Set Exchange correlation to meta-GGA
- Set the k-point sampling to 7, 7, 7.
- Under **Basis set/exchange correlation**, change the pseudopotentials to HGH[Z=5], LDA.PZ for Arsenic and HGH[Z=13], LDA.PZ for Indium. Choose the Tier 3 basis set for both elements.
- Finally set the output filename to *InAs_hgh.nc*.

Save the script, then send it to **Job Manager**, by using the Send To icon, and run the job. The calculation will take just a minute or two.

[3] http://www.ioffe.rssi.ru/SVA/NSM/Semicond/InAs/index.html

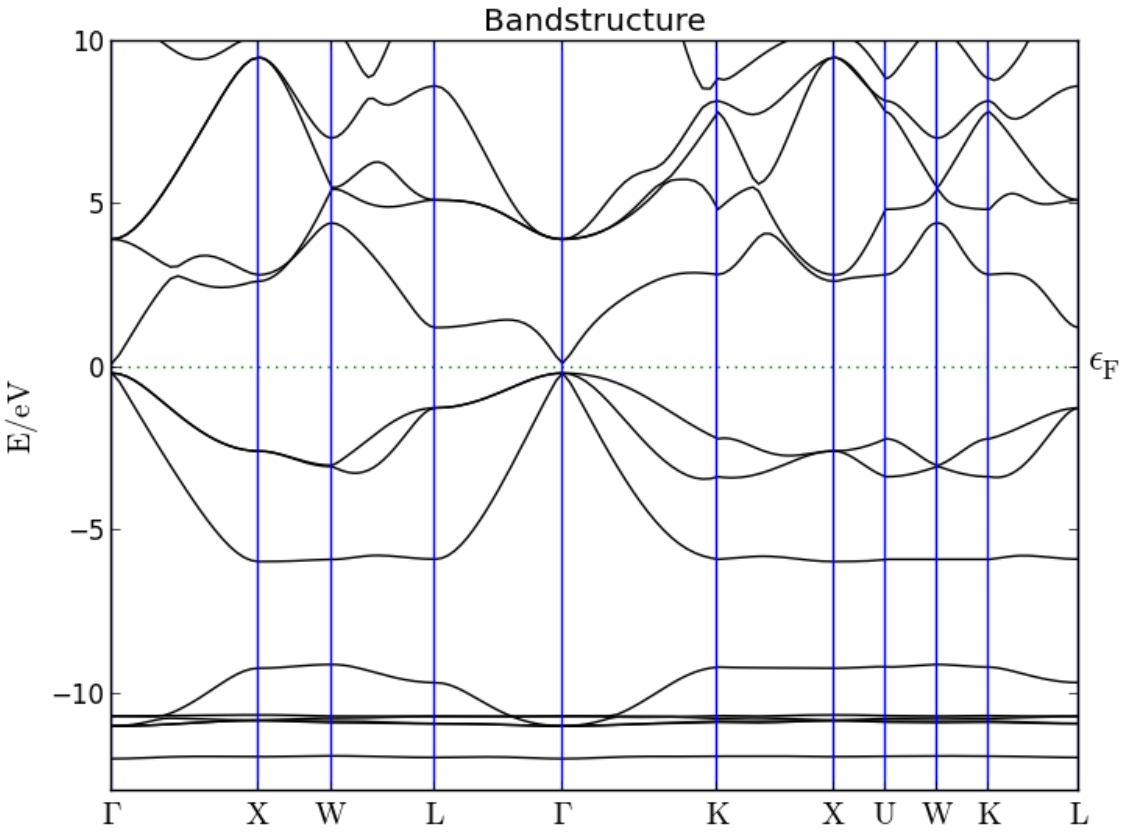

Lattice Parameters
Choose the lattice type from the dropdown menu.
Lattice type:
Keep
fractional
coordinates constant when changing the lattice
Adjust the lattice parameters of the selected lattice type. Only parameters relevant for the lattice type can be changed. Lattice parameters can be exported to the clipboard by right-clicking.
a (Å)
6.0583
α
90
b (Å)
6.0583
β
90
b/a
1
c (Å)
6.0583
γ
90
c/a
1
Primitive Vectors
Manipulate the Primitive Vectors directly. This is only possible if UnitCell was chosen from the dropdown menu.
x (Å)
y (Å)
z (Å)
A 0 3.02915 3.02915
B 3.02915 0 3.02915
C 3.02915 3.02915 0
Volume = 55.5894 Å³


New Calculator
Calculators
ATK-DFT
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-Classical
Abinit
FHI-aims
Calculator settings
Algorithm parameters
Basic
Iteration control parameters
Basis set/exchange correlation
Numerical accuracy parameters
Poisson solver
Estimate Memory Usage
Exchange Correlation
Type
MGGA
Predifined functionals
TB09LDA
Exchange
TB09Exchange
Correlation
PerdewZungerCorrelation
Parameters
Hubbard U
Disabled
Spin
Unpolarized
Van der Waals correction
Disabled
Parameters
Basis Set
Element
Pseudopotential
Basis Set
Arsenic
HGH [Z=5] LDA.PZ
Tier 3
Indium
HGH [Z=13] LDA.PZ
Tier 3
Activate tags
OK

Note: For self-consistent calculations of the c-parameter, more accurate band structures are usually obtained using the HGH pseudopotentials which include semi-core states. This is the case for HGH[Z=13] above for Indium, which include the ten 4d electrons, in contrast to HGH[Z=3] which only includes the 3 valence electrons (5s and 5p states). Semi-core pseudopotentials are however only provided in **ATK** for some elements; for Arsenic there is only a HGH[Z=5] pseudopotential, without pseudocore states.

Band structure for self-consistently determined c

When the calculation finishes, locate the band structure object on the LabFloor and click **Bandstructure Analyzer** to plot the band structure like below.



You will find that the calculated band gap is very close to the experimental value 0.354 eV. This is actually a quite general trend, when using the HGH pseudopotentials. The figure below shows the calculated vs. experimental band gaps for a range of semiconductors and insulators. Results are shown for both LDA and TB09 meta-GGA calculations (additional examples are provided in Ref.[Page 1020, 1]) and show that while LDA always underestimates the band gaps (the famous band gap problem in DFT), the TB09 meta-GGA results are generally quite close to the experimental values, and typically as accurate as the results of much more expensive hybrid functional or even GW calculations.

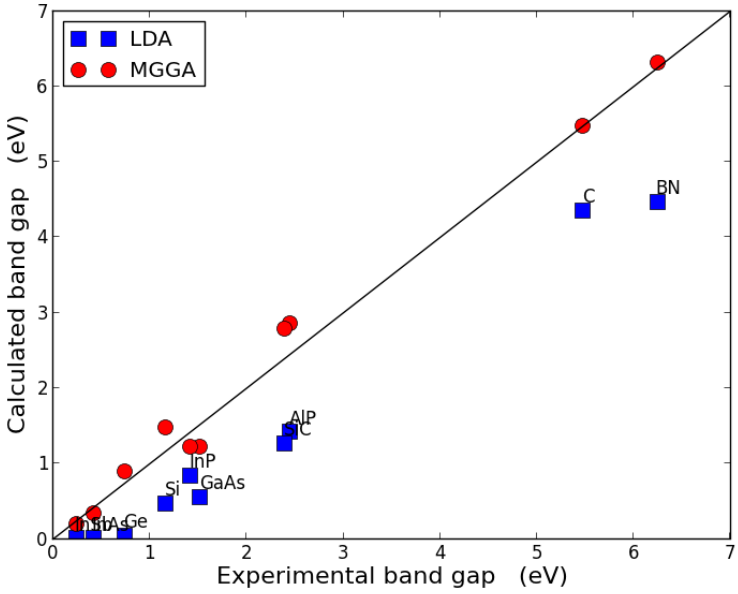


Fig. 3.86: Calculated vs. experimental band gaps. The calculations are performed with LDA (blue squares) and TB09 meta-GGA (red circles) XC functionals. For the TB09 meta-GGA calculations, the HGH pseudopotential have been used with a Tier 4 basis set, and the c-parameter is calculated self-consistently.

Fitting the TB09 meta-GGA c-parameter

While HGH pseudopotentials in general gives rather accurate band gaps by computing the c-parameter self-consistently, it might be desirable to get an even closer agreement with experimental values. In order to achieve that, it is possible to manually set the TB09 meta-GGA c-parameter such that the calculated band gap coincides with the experimental one. This can also be used as a way to compensate for a slightly less accurate pseudopotential or basis set, as will be illustrated in this section.

Setting up the script

Go back to **Script Generator** and open the **New Calculator** block, and make the following settings:

•Under :menuselection:**Basis set** → **exchange correlation**, change the pseudopotentials to FHI[Z=5], LDA.PZ for Arsenic and FHI[Z=3], LDA.PZ for Indium. That is, choose pseudopotentials without semicore states.

•Set the basis set to DoubleZetaPolarized for both elements.

•Change the output filename to InAs_c_fit.nc

Send the script to **Editor**, by using the Send To icon.

In **Editor**, go to the line where the XC function is specified and add the c-parameter argument to the MGGA.TB09LDA functional.

```
#-----------------------------------------
# Exchange-Correlation
#-----------------------------------------
exchange_correlation = MGGA.TB09LDA(c=1.0)
```

Then send the script to **Job Manager** (again use the Send To icon) and run the calculation. This time it will only take a few seconds.

When the calculation is done, plot the band structure and measure the band gap. If the the gap is larger than the experimental value (0.354 eV), then go back to **Editor** and decrease the c-value a bit and run again. If the band gap is smaller, you must increase the c-value. Try a few iterations to see if you can get a close agreement with the experimental value.

The figures below show the calculated band gap (left panel) for four different c-values (0.9, 0.95, 1.0, 1.05). Due to the very linear behavior, we can safely find the optimal c-parameter at the intersect between the interpolated line of the calculated band gaps and the experimental gap. This yields an optimal c-value of 0.936, which we will use in the following.

The right panel shows the calculated conduction band effective mass for the same four c-values. Also indicated is the experimental conduction band mass (0.023 m_e). The lines cross at c=0.935, so - very importantly - approximately the same c-value optimizes both the band gap and the effective mass, thus giving a very good description at least of the lowest Γ valley of InAs.

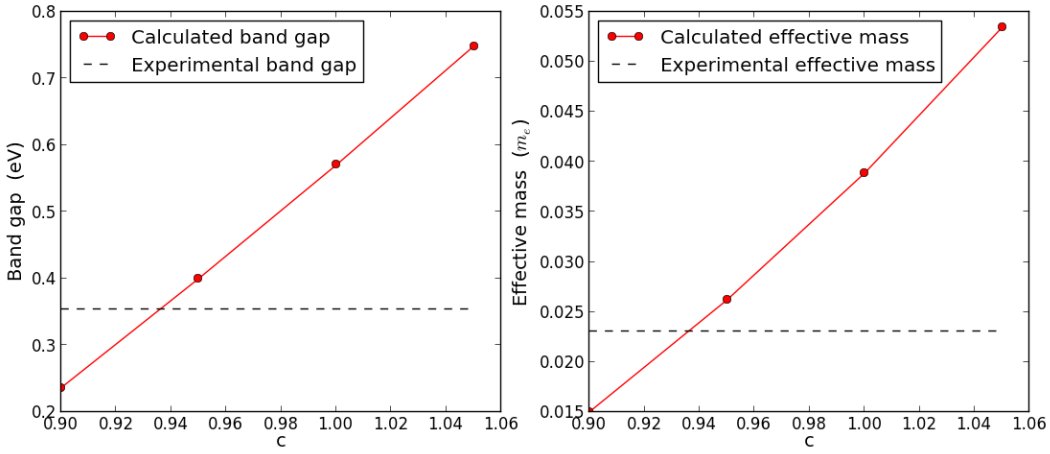


Fig. 3.87: Calculated band gap (left) and conduction band effective mass (right) vs. the TB09 meta-GGA c-parameter. Increasing c will always lead to a larger band gap. Both the band gap and the effective mass are optimized by approximately the same c-parameter (0.936).

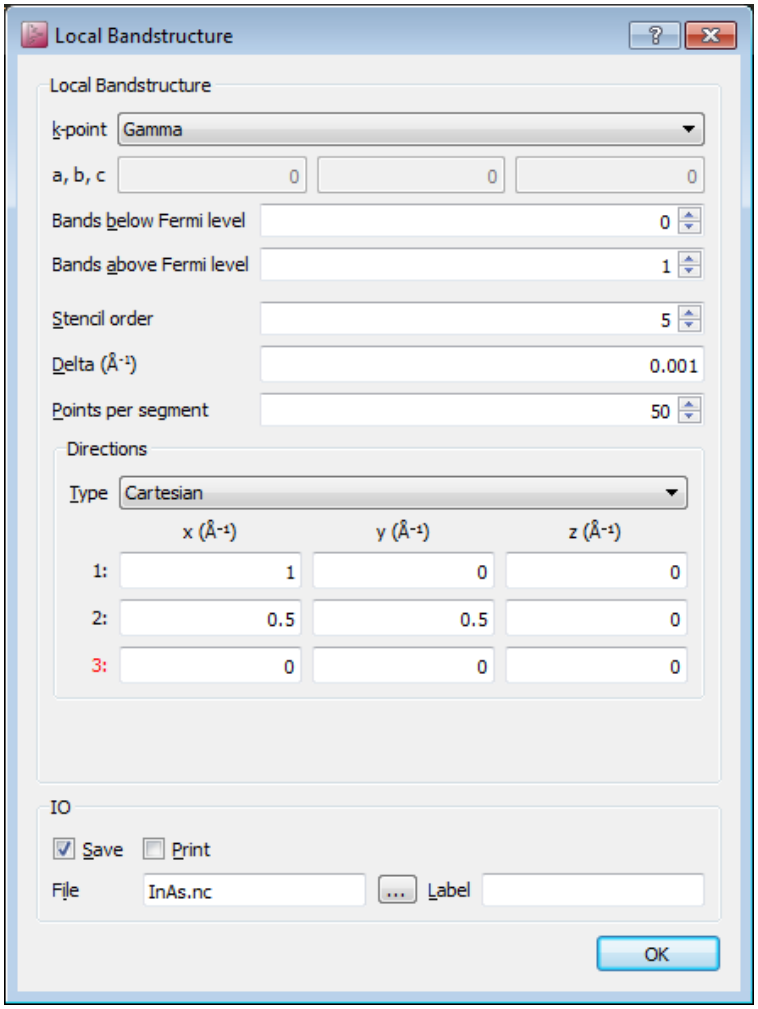
Band structure with optimized c-parameter

Before continuing to study confined InAs structures, you will first calculate and analyze the bulk band structure with the optimized c-parameter in some more detail.

Go back to **Script Generator** and make the following adjustments:

- Delete **Bandstructure** object.
- Insert an **LocalBandstructure** block and open it (see figure below).
  - Set Bands below Fermi level to 0.
  - Set Points per segment to 50.
  - Add a second direction: 0.5, 0.5, 0
  - Otherwise keep the default.

•Change the output file name to InAs_bulk_c_optimized.nc.



Transfer the script to **Editor** and insert the optimized c-parameter value:

```
#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = MGGA.TB09LDA(c=0.936)
```

Finally, send the script to **Job Manager** and run the calculation.

To analyze the results, use **Local Bandstructure Analyzer**.

LocalBandstructure object calculates the effective mass m^* and further fits the band structure to the function $E(1+\alpha E) = \frac{\hbar^2 k^2}{2m^*}$, where α is the parameter to fit. You can control the range of the fit with the number of **Fit points**. This non-parabolic dispersion relation provides a much better fit to the TB09 meta-GGA band structure than the usual parabolic model $E = \frac{\hbar^2 k^2}{2m^*}$, as shown in the figure below. The red dots indicate the range used for fitting α.

From the LocalBandstructure Analyzer, we see that the InAs conduction band in the [1,0,0] direction is very accurately described by $m^* = 0.028\, m_e$ and $\alpha = 2.869\, eV^{-1}$, while the parabolic model only describes the band well in the vicinity of the conduction band minimum.

Try to change the **Direction** to $(0.5, 0.5, 0)^{-1}$ to calculate the local band structure for this direction. Note that the effective mass is the same as before, i.e. it is isotropic, while the non-parabolicity parameter α is not isotropic.

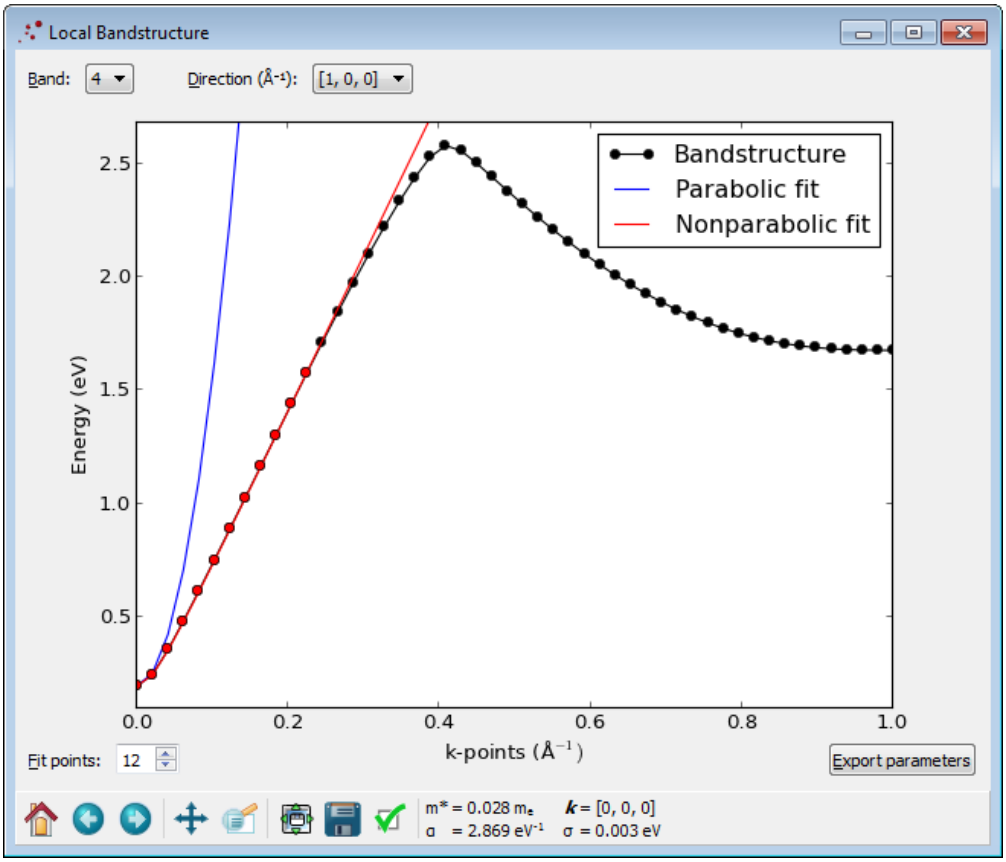


Fig. 3.88: Local bandstructure analyzer. The black dots show the calculated band structure data, while the blue and red curves show the results of a parabolic and non-parabolic model, respectively. The fitted α value is shown below the plot together with the effective mass.

Setting up and passivating an InAs slab

Slab structure

You will now setup an InAs slab which is confined in the [100] direction.

Go back to the **Builder** window, where it will be assumed that you still have the bulk InAs crystal from previous.

Next, **Builders** → **Surface (Cleave)** to cleave the crystal along the (100) direction.

•Keep the default (100) Miller indices, and press Next.

•Keep the default surface lattice vectors, and press Next.

•Keep the default supercell - this will ensure that the wire direction is perpendicular to the surface.

•Finalize your configuration by choosing Non-periodic and normal (Slab) configuration with a thickness of 8 layers and a top and bottom vacuum of 5 Å.

•Press Finish to add the cleaved structure to the Stash.

Adding hydrogens

The surfaces of the so-created bare InAs slab have dangling bonds due to unpaired In and As atoms. This will lead to metallic surface states with energies in the band gap. In order to passivate the surfaces you will now add hydrogen atoms.

•Open **Coordinate Tools** → **Custom Passivator**.

•Set Hybridization to 4 (SP3).

•Change the bond distances to 1.3 Å (otherwise neighboring hydrogen atoms will be too close).

•Press the Passivate button.

Note: The **Custom Passivator** adds tags to the added hydrogen atoms - this is essential for this tutorial, so dont change or remove them. You can inspect the tags by opening **Select** → **Tags**. You will later use these tags to identify H atoms bonded to In and As, respectively, when specifying compensation charges and using specialized pseudopotentials.

The structure should now resemble the figure below.

Send the structure to **Script Generator**.

Band structure with default hydrogen atoms

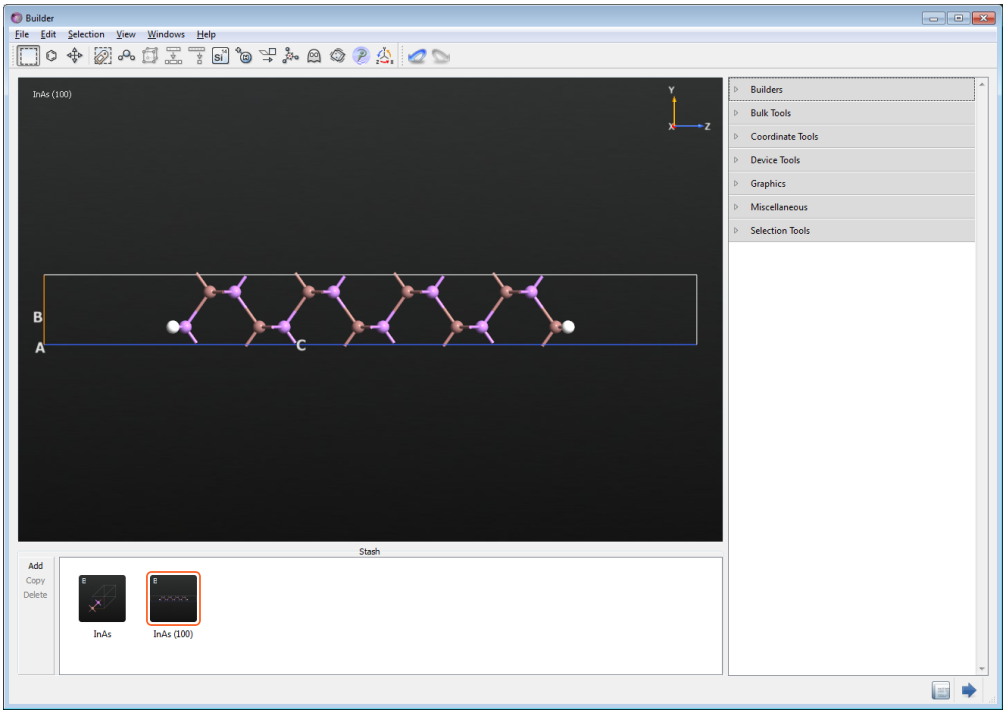
In the following you will calculate the band structure using default settings for the passivating hydrogen atoms. For this purpose:

•Add a New Calculator.

•Add Analysis/Bandstructure.

Open the **New Calculator** block, and set exchange correlation to meta-GGA and the k-point sampling to 7, 7, 1. Note that the FHI pseudopotentials are selected by default.

Finally, open the **Bandstructure** block and set the Brillouin zone route to Y,G,X, and increase the points per segment to 100.



As discussed in *Short introduction to TB09 meta-GGA*, the self-consistent calculation of the c-parameter only works for bulk systems without vacuum regions, whereas for a slab calculation one should use a fixed value. Therefore, send the script to **Editor** and specify the c-parameter you found in *Band structure with optimized c-parameter* to reproduce the experimental band gap for the FHI pseudopotentials:

```
#-----------------------------------------
# Exchange-Correlation
#-----------------------------------------
exchange_correlation = MGGA.TB09LDA(c=0.936)
```

Save the script, then send it to **Job Manager** (again use the Send To icon) and run the calculation. This time it will take a bit longer, around 5 minutes.

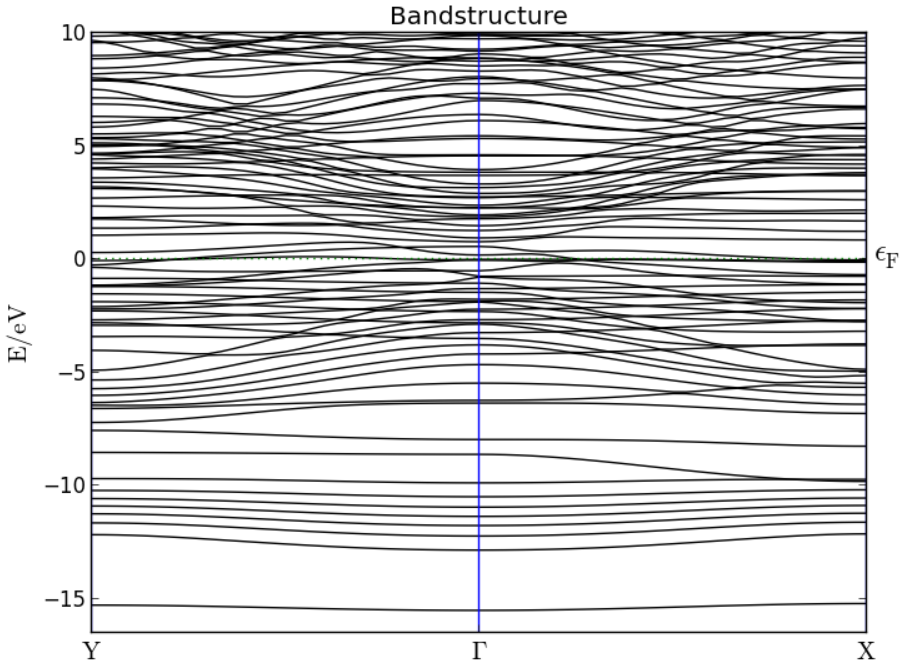
Note: When running the script, you may observe a warning in the log file:

```
################################################################################
# WARNING                                                                      #
#                                                                              #
# The computed TB09 meta-GGA XC potential                        #
# diverged in   0.01 % of the simulation volume, and was                       #
# truncated to be in the range [-10.0000,  10.0000] Hartree                    #
#                                                                              #
################################################################################
```

You do not need to worry about this because you have set the c-parameter to a fixed value. If, on the other hand, you were trying to calculate it self-consistently, the warning would be relevant. However, as explained in *Short introduction to TB09 meta-GGA*, the self-consistent evaluation of the c-parameter involves an integral over the entire calculation unit cell, including the vacuum regions, where the TB09 meta-GGA XC potential can diverge. For confined systems like slabs and nanowires, you therefore always need to specify the c-parameter manually.

Analyzing the results

If you now plot the result of this calculation with the Bandstructure Analyzer you will find that there is no band gap, but instead a lot of more or less flat bands in the region around the Fermi energy. These are due to surface states, which arise due to inadequate passivation of the surface.



In order to get rid of the surface states (or rather to push them away from the Fermi level) the hydrogen atoms need to be treated differently. The following two sections illustrate two possible ways to eliminate the surface states, to produce a proper band structure for your InAs slab.

Passivation using pseudo-hydrogen

In the following you will calculate the band structure and effective mass using so-called pseudo-hydrogen atoms containing fractional charges. The use of pseudo-hydrogen is a well-known procedure in the literature to passivate III-V and II-VI surfaces[4].

Return to the **Script Generator** window and open the **New Calculator** block. Make the following settings to set different basis sets for the hydrogen atoms bonded to Indium and Arsenic, respectively:

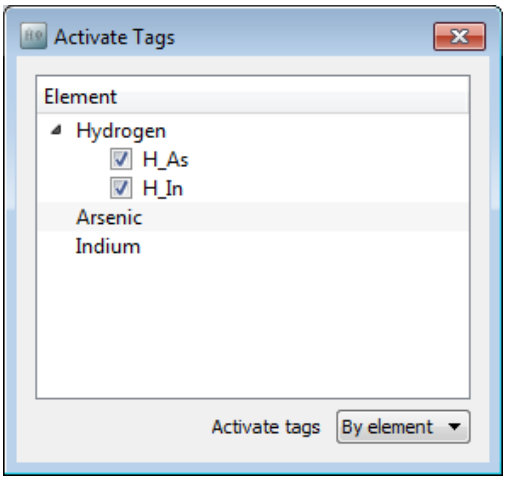
•Under **Basis set/exchange correlation** click the **Activate tags** button and check both H_As and H_In for hydrogen.

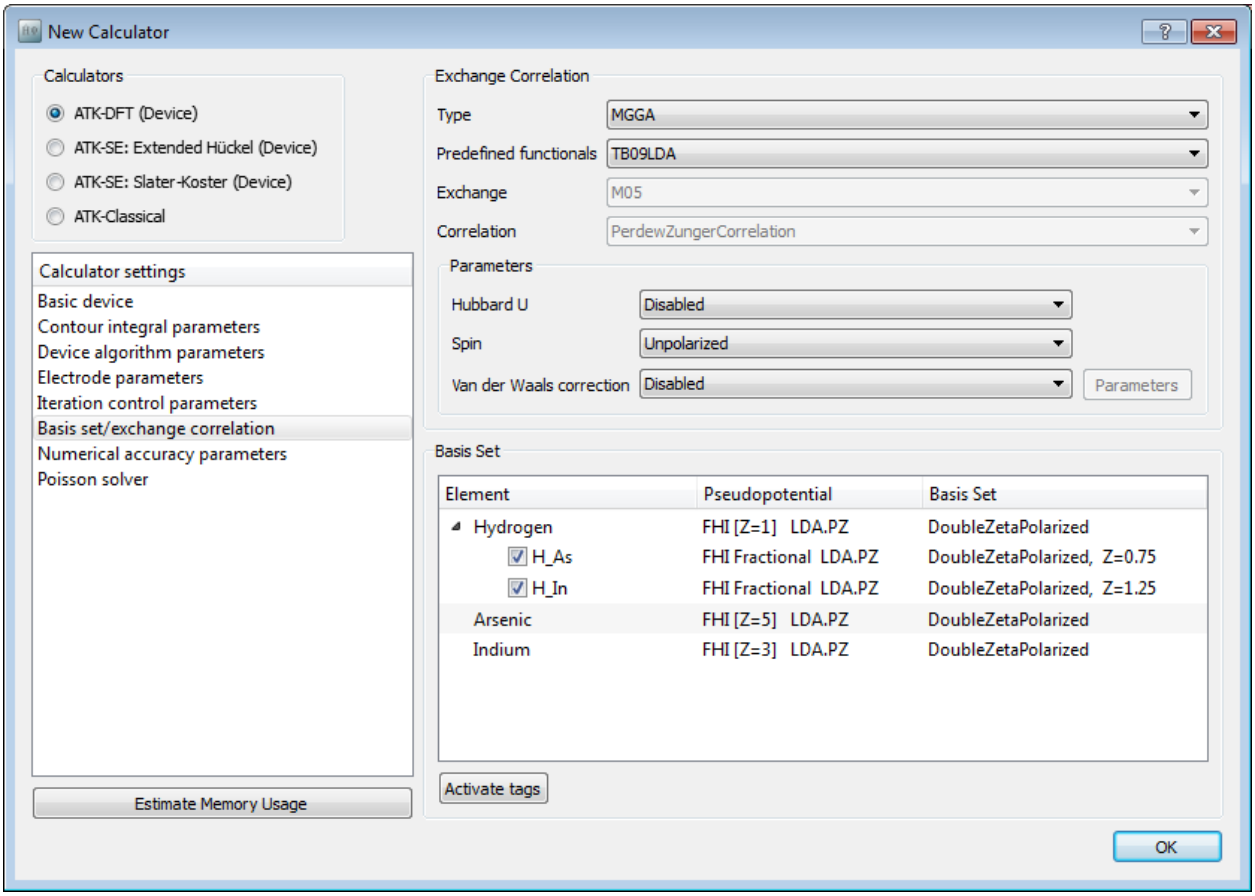
This technique, i.e. using tags, allows you to assign a different basis sets to atoms of the same element, which can be a useful trick in other situations too.

•Change the pseudopotential for the H_As atoms to FHI Fractional LDA.PZ, and set the basis set to DoubleZetaPolarized, Z=0.75

•Change the H_In pseudopotential to FHI Fractional LDA.PZ, and the basis set to DoubleZetaPolarized, Z=1.25

[4] J.Li, and L.-W. Wang, Phys. Rev. B, **72**, 125325, (2005)


Activate Tags
Element
Hydrogen
H_As
H_In
Arsenic
Indium
Activate tags
By element


New Calculator
Calculators
ATK-DFT (Device)
ATK-SE: Extended Hückel (Device)
ATK-SE: Slater-Koster (Device)
ATK-Classical
Calculator settings
Basic device
Contour integral parameters
Device algorithm parameters
Electrode parameters
Iteration control parameters
Basis set/exchange correlation
Numerical accuracy parameters
Poisson solver
Estimate Memory Usage
Exchange Correlation
Type
MGGA
Predefined functionals
TB09LDA
Exchange
M05
Correlation
PerdewZungerCorrelation
Parameters
Hubbard U
Disabled
Spin
Unpolarized
Van der Waals correction
Disabled
Parameters
Basis Set
Element
Pseudopotential
Basis Set
Hydrogen
FHI [Z=1] LDA.PZ
DoubleZetaPolarized
H_As
FHI Fractional LDA.PZ
DoubleZetaPolarized, Z=0.75
H_In
FHI Fractional LDA.PZ
DoubleZetaPolarized, Z=1.25
Arsenic
FHI [Z=5] LDA.PZ
DoubleZetaPolarized
Indium
FHI [Z=3] LDA.PZ
DoubleZetaPolarized
Activate tags
OK

Note the following important points.

•Two separate hydrogen basis sets are defined: one for hydrogens bonded to Indium atoms, and one for those bonded to Arsenic atoms. The basis sets use different pseudopotentials with charge 1.25 and 0.75 electrons, respectively. (From the widget it may appear that they use the same, but the fractional pseudopotentials are automatically selected to match the basis set charge.)

•The use of these pseudopotential charges can be understood as follows[Page 1031, 4]. The surface of an unpassivated nanocrystal consists of dangling bonds which will introduce band gap states. The purpose of a good passivation is to remove these band gap states. One way to do so is to pair the unbonded dangling bond electron with other electrons. If a surface atom has m valence electrons, this atom will provide m/4 electrons to each of its four bonds in a tetrahedral (sp3) crystal. To pair these m/4 electrons in each dangling bond, a passivating agent should provide (8 m) / 4 additional electrons. Since As has m=5, it should be passivated with pseudo-hydrogen containing (8-5)/4=0.75 electrons. Likewise, In has m=3 valence electrons, and should be passivated with pseudo-hydrogen with (8-3)/4=1.25 electrons.

•Both types of hydrogen atoms are still charge neutral, i.e. there are 1.25 electrons and a core with charge 1.25, and similarly for the 0.75 case.

Add a LocalBandstructure object **Analysis** → **LocalBandstructure** and adjust the following parameters:

•Set bands below Fermi level to 0.

•Increase Points per segment to 50.

Set the output file name to InAs_slab_pseudohydrogen.nc, and send the script to **Editor** and make the usual adjustment of the c-parameter.

```
#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = MGGA.TB09LDA(c=0.936)
```

Finally, save the script and send it to **Job Manager** to run the calculation.

Results

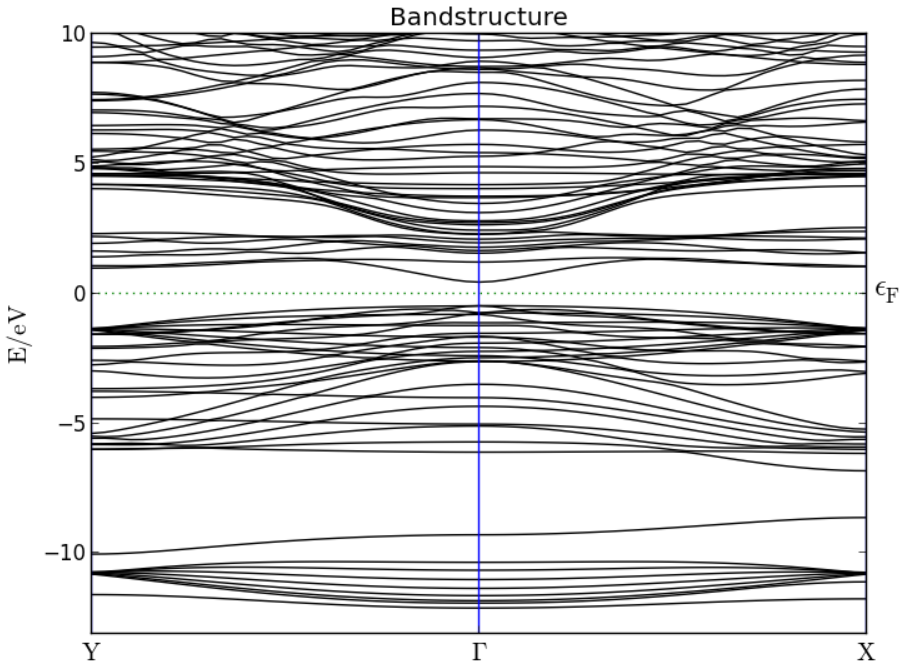
Plotting the calculated band structure with the Bandstructure Analyzer you can observe that a band gap has opened up. The pseudo-hydrogen atoms have thus indeed moved the surface states away from the Fermi energy, as intended.

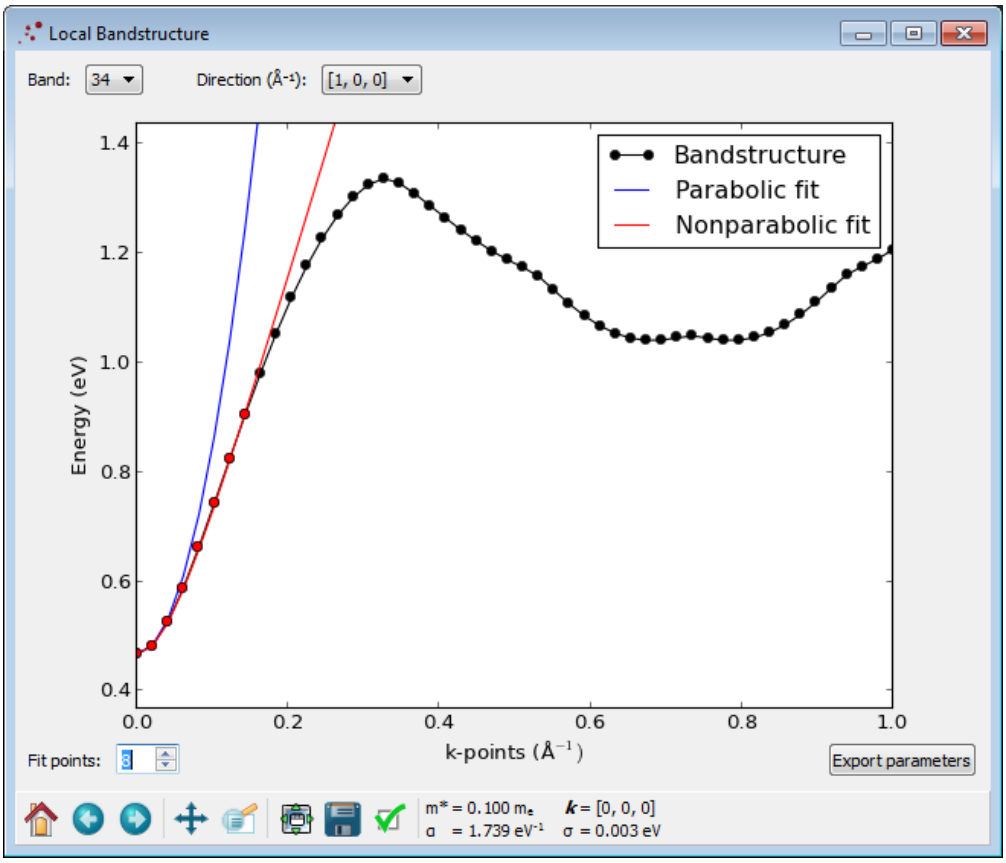
By inspecting the band structure, you can see that the band gap has increased to a value of 0.1 eV, compared to the bulk value 0.354 eV, as a result of quantum confinement.

Next you may inspect the calculated effective mass and non-parabolicity parameter using the **Local Bandstructure Analyzer**.

Notice that the effective mass is increased to $0.1\,m_e$ compared to the bulk value (0.028), while the non-parabolicity parameter α has decreased to 1.739 (bulk value was 2.869). In other words, an effective mass description is better for the slab than for the bulk InAs.

In the section *Analytical band structure of a 2D slab* you will analyze the effect of quantum confinement on both the band gap and the effective mass in more detail.


Bandstructure
10
5
0
−5
−10
E/eV
ϵ_F
Y
Γ
X


Local Bandstructure
Band: 34
Direction (Å⁻¹): [1, 0, 0]
Bandstructure
Parabolic fit
Nonparabolic fit
Energy (eV)
1.4
1.2
1.0
0.8
0.6
0.4
0.0
0.2
0.4
0.6
0.8
1.0
k-points (Å⁻¹)
Fit points: 3
Export parameters
m* = 0.100 mₑ
α = 1.739 eV⁻¹
k = [0, 0, 0]
σ = 0.003 eV

Passivation using compensation charges

An alternative to using pseudo-hydrogen atoms is to use compensation charges on the hydrogen atoms. This adds external potentials corresponding to +0.25 electron charge for hydrogens binding to Indium and -0.25 electron charge for hydrogens binding to Arsenic, resembling the extra +/-0.25 electrons in the pseudo-hydrogen pseudopotentials used in *Passivation using pseudo-hydrogen* above.

•Go to **Script Generator** and open **New Calculator** block.

•Go to **Basis set/exchange correlation** and remove the check marks for H_As and H_In. This sets the basis set to the usual (uncharged) DoubleZetaPolarized for all hydrogen atoms.

•Set the output file name to InAs_slab_compensation_charge.nc.

The compensation charges will be defined in a script. Therefore, transfer the script to **Editor** (again use the Send To icon) and insert the Python code listed below before the specification of the calculator.

```
#------------------------------------------------------------
# Add compensation charges to the hydrogen atoms
#------------------------------------------------------------

# Compensation charge for H_As and H_In
charge_H_As = -0.25
charge_H_In =  0.25

# Set compensation charge
compensation_As = [('H_As',charge_H_As)]
compensation_In = [('H_In',charge_H_In)]

compensation = compensation_As + compensation_In
bulk_configuration.setExternalPotential(AtomicCompensationCharge(compensation))

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
```

Note: The tags on the hydrogen atoms are used to find their indices. Tags can be used for many purposes!

ATK will automatically use a free charge corresponding to the total sum of all compensation charges, in order to keep the entire system neutral still.

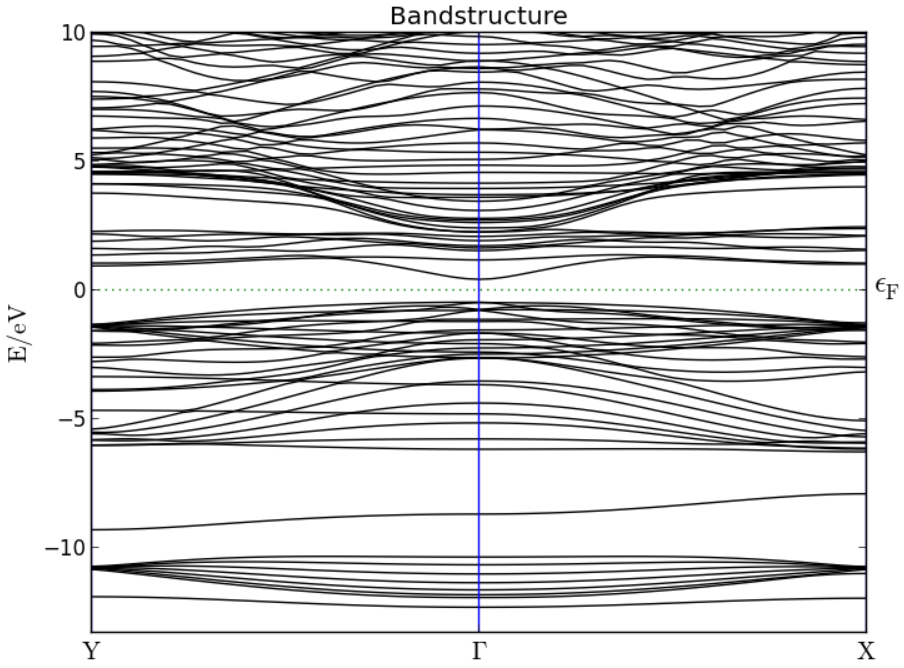
Again, add the c-parameter argument to the MGGA.TB09LDA functional:

```
#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = MGGA.TB09LDA(c=0.936)
```

Save the modified script and transfer it to **Job Manager** and run the calculation.

Results

Plotting the results, you should obtain an image like below.



Again there is a clear band gap around the Fermi energy, and the band structure is generally very similar to the one obtained with the pseudo-hydrogens.

Non-parabolicity in confined structures

In the section *Band structure with optimized c-parameter* you learned that the bulk conduction band structure in InAs is very well described with a non-parabolic model:

$$E(1+\alpha E) = \frac{\hbar^2 k^2}{2m^*}$$

where $m^* = 0.028\, m_e$ and $\alpha = 2.87\, eV^{-1}$. In this section, you will use this expression to develop an analytical expression for the conduction band dispersion of both the 2D slab and an InAs nanowire. In particular, you will find that the values of m^* and α determined for bulk InAs give an accurate description also of the confined non-parabolic band structures, when treated the correct way.

Analytical band structure of a 2D slab

The equation above only gives the energy as an implicit function of k. A solution to the quadratic equation lead to the explicit dispersion relations

$$E(k_x, k_y, k_z) = \frac{-1 + \sqrt{1 + \frac{2\alpha\hbar^2}{m^*}(k_x^2 + k_y^2 + k_z^2)}}{2\alpha},$$

where it has been assumed that m^* and α are independent of the crystal orientation. This is an excellent approximation for the electron mass m^*, whereas α does vary, as seen above.

In a confined system, i.e. a 2D slab or a 1D nanowire, the allowed wavelengths become quantized in the confined directions. In the case of a slab confined in the z-direction, one may assume that the effective potential can be approximated by a one-dimensional infinite potential well of width W. The allowed wavelengths in this directions are then $\lambda_n = \frac{2W}{n}$, $n = 1, 2, 3, ...$ Only the lowest band with $n = 1$ is relevant here, and the corresponding wavenumber is $k_z = \frac{2\pi}{\lambda_1} = \frac{\pi}{W}$. Inserting this in the equation above yields:

$$E_{2\mathrm{D}}(k_x, k_y) = \frac{-1 + \sqrt{1 + \frac{2\alpha\hbar^2}{m^*}(k_x^2 + k_y^2 + \frac{\pi^2}{W^2})}}{2\alpha},$$

From this expression you can readily calculate the shift in band gap due to quantization:

$$\Delta E_{\mathrm{gap}} = E^{2D}(0, 0) = \frac{-1 + \sqrt{1 + \frac{2\alpha\hbar^2\pi^2}{m^* W^2}}}{2\alpha}.$$

It is also possible to obtain an expression for the Γ-point effective mass $m^* = \hbar^2(\partial^2 E(k)/\partial k^2|_{k=0})^{-1}$. Try yourself to derive the expression

$$m^*_{2\mathrm{D}} = m^* \sqrt{1 + \frac{2\alpha\hbar^2\pi^2}{m^* W^2}}.$$

In order to evaluate the above expressions you only need to know the slab width W. The width might be defined in various ways, but if you consider the confinement of the conduction band Bloch state calculated at the Γ-point, you will find that it extends over a width of approximately $W = 30$ Å in the slab you have defined. Using this number, you get the following values for the band gap increase and effective mass:

$$\Delta E_{\mathrm{gap}} = 0.62eV$$

$$m^*_{2\mathrm{D}} = 0.11 m_e,$$

which compare quite well with the values you found in *Results*, namely $\Delta E_{\mathrm{gap}} = (0.96 - 0.354)$ eV $= 0.61$ eV, and $m^*_{2\mathrm{D}} = 0.1\, m_e$.

Thus, knowing the effective mass and α for bulk InAs is enough to get a very reasonable description of the band structure in a 2D confined InAs slab. A similar approach can be applied to a nanowire, as will be shown in *Nanowire band structure* below.

Note that the very significant increase in effective mass by more than a factor of three cannot be accounted for by a parabolic model $E(k) = \frac{\hbar^2 k^2}{2m^*}$ in which the effective mass remains constant, and only the band gap is affected by quantum confinement.

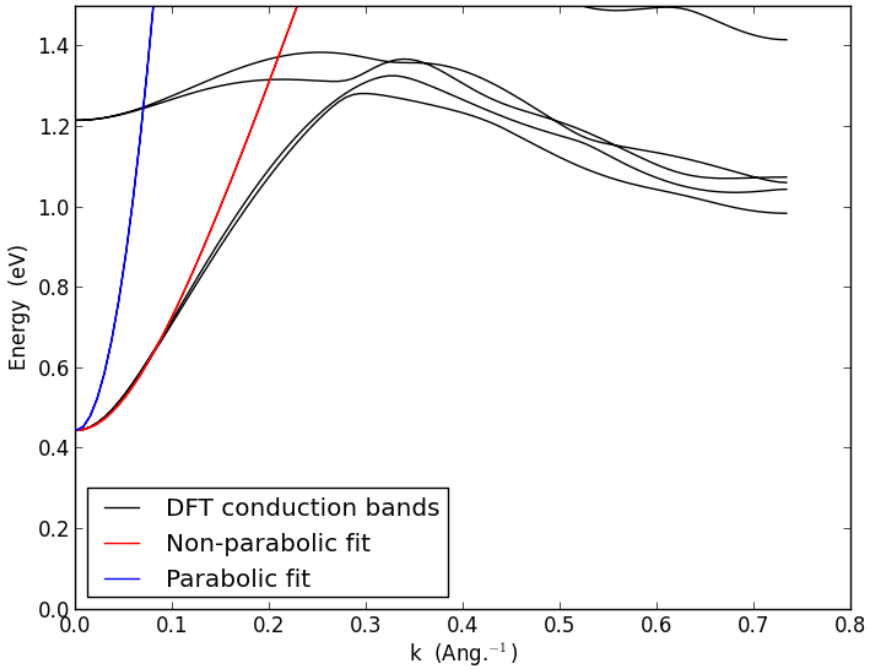
In order to visually check the quality of the non-parabolic model you can run the script `analyze_nanowire.py` shown below.

```
import pylab as pl
from NL.CommonConcepts.Configurations.Utilities import fractional2cartesian


#-------------------------------------------------------------------------
# Model data
#-------------------------------------------------------------------------

# alpha parameter fitted to bulk InAs
alpha_bulk = 2.869

# InAs bulk conduction band effective mass
meff_bulk = 0.028*electron_mass

# Effective width of slab
slab_width = 30*Angstrom


#-------------------------------------------------------------------------
# Load DFT calcularted bandstructure and effective mass
#-------------------------------------------------------------------------

# Read bulk configuration
configuration = nlread('InAs_slab_pseudohydrogen.nc',BulkConfiguration)[0]

# Read bandstructure
bandstructure = nlread('InAs_slab_pseudohydrogen.nc',Bandstructure)[-1]

# Read Effective mass
effective_mass = nlread('InAs_slab_pseudohydrogen.nc',EffectiveMass)[0]
meff = effective_mass.evaluate(band=1)[0][0][0]

# Get the fractional kpoints
kpoints = bandstructure.kpoints()


# Get reciprocal lattice vectors (used to convert frational k to cartesian k)
G = configuration.bravaisLattice().reciprocalVectors()

# K-points in cartesian coordinates, Ang^-1
k_cart = fractional2cartesian(kpoints, G)

# Get |k| as 1D array
k = numpy.zeros(len(kpoints))
for ii,ki in enumerate(k_cart.inUnitsOf(Angstrom**-1)):
    k[ii] = (ki[0]**2 + ki[1]**2 + ki[2]**2)**0.5

# Add unit
k = k*Angstrom**-1

# Get all the bands
bands = bandstructure.evaluate().inUnitsOf(eV)
```

(continues on next page)

(continued from previous page)

```
# Energies at the Gamma-point
E0 = bands[0,:]

# Index of conduction band
conduction_band_index = numpy.where(E0 > 0.0)[0][0]

# Get the energies of the conduction band
conduction_band = bands[:,conduction_band_index]

# conduction band minimum
cbm = min(conduction_band)

# Evaluate non-parabolic model bandstructure
E_non_parabolic = (-1 + (1 + 2*alpha_bulk*(hbar**2/meff_bulk*(k**2 + numpy.pi**2/(slab_
↪width)**2 ) ).inUnitsOf(eV) )**0.5 )/(2*alpha_bulk)

# Align conduction band minimum to DFT
E_non_parabolic = E_non_parabolic - min(E_non_parabolic) + cbm

# Evaluate parabolic model bandstructure and align CBM
E_parabolic = (0.5*hbar**2*k**2/meff_bulk).inUnitsOf(eV) + cbm


# Ananlytical effective mass of slab
m_slab = meff_bulk * (1 + 2*alpha_bulk*(hbar**2*numpy.pi**2/(meff_bulk*slab_width**2)).
↪inUnitsOf(eV) )**0.5
print ''
print 'Analytical slab mass = %.4f m_e' %m_slab.inUnitsOf(electron_mass)
print 'DFT slab mass = %.4f m_e\n' %0.095

# Analytical band gap increase
delta_gap = (-1 +  (1 + 2*alpha_bulk*(hbar**2*numpy.pi**2/(meff_bulk*slab_width**2)).
↪inUnitsOf(eV) )**0.5)/(2*alpha_bulk)
print 'Analytical band gap increase = %.4f eV' %delta_gap
print 'DFT gap change = %.4f eV' %(0.91-0.354)
print ''


pl.figure()
pl.plot(k,conduction_band,'k',label='DFT conduction bands')
pl.plot(k,bands[:,conduction_band_index+1:conduction_band_index+5],'k')
pl.plot(k,E_non_parabolic,'r',label='Non-parabolic fit ')
pl.plot(k,E_parabolic,'b',label='Parabolic fit')
pl.xlabel('k  (Ang.$^{-1}$)')
pl.ylabel('Energy  (eV)')
pl.ylim([0,1.5])
pl.legend(loc=0)

pl.show()
```



Nanowire band structure

As a final task, you will investigate if the analytical model can also be used to estimate the band structure of a nanowire. Since the nanowire calculations are rather time-consuming we will just show the results, and leave the actual calculations as an exercise.

You can create an InAs nanowire by going back to the **Builder** window and follow these steps:

•Repeat the InAs slab 6 times along the A-axis.

•Change the lattice parameters so that the A-vector is 40 Å in the x-direction.

•Passivate the nanowire in the same way as the slab, using the **Custom Passivator**.

•Center the coordinates.

Send the structure to **Script Generator** and use exactly the same procedure as for the slab calculations with pseudo-hydrogen atoms. Additionally,

•Set the k-point sampling to (1,9,1)

•Add **Bandstructure** and change the Brillouin zone route to G, Y.

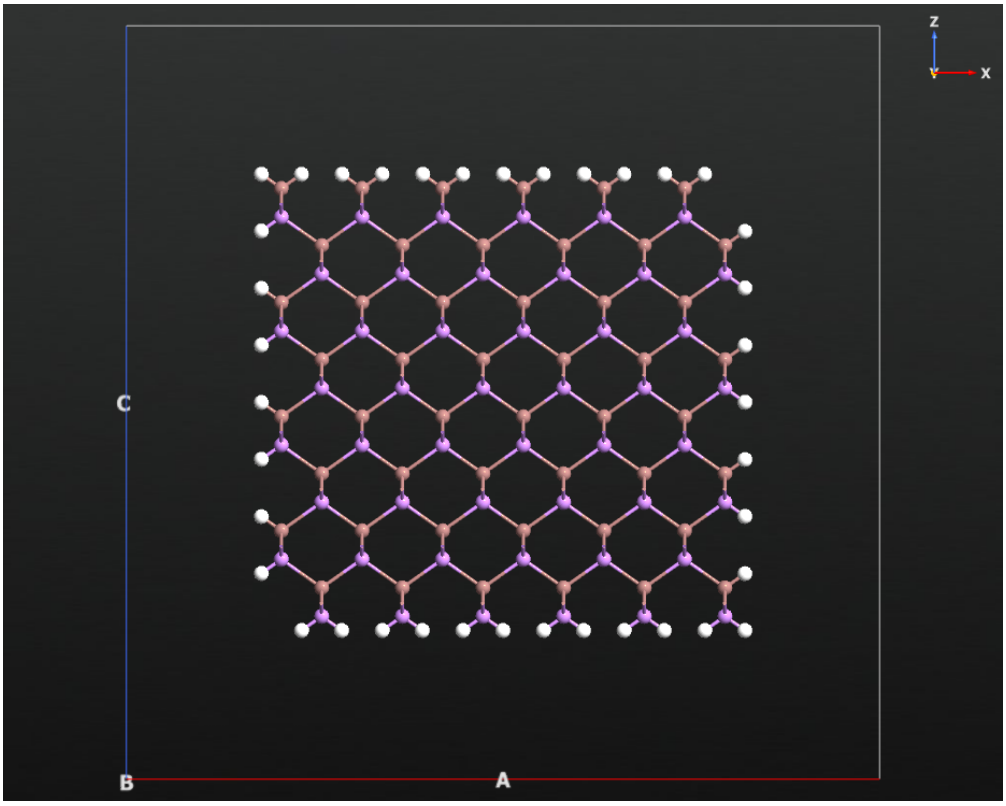
•Add Analysis/EffectiveMass and set Direction to Cartesian, and set (x,y,z)=(0,1,0).

•Set the output file name to InAs_nanowire.nc.

•Send the script to **Editor** and set the TB09 meta-GGA c-parameter to 0.936.

•Finally send the script to **Job Manager** and run it. Note that this calculation might take about an hour to finish.

When the calculation is finished you can run the Python script `analyze_nanowire.py` shown below in order to analyze the quality of the non-parabolic model .




```
import pylab as pl
from NL.CommonConcepts.Configurations.Utilities import fractional2cartesian


#-------------------------------------------------------------------------
# Model data
#-------------------------------------------------------------------------

# alpha parameter fitted to bulk InAs
alpha_bulk = 2.869

# InAs bulk conduction band effective mass
meff_bulk = 0.028*electron_mass

# Effective widths of nanowire
Dx = 30*Angstrom
Dz = 30*Angstrom


#-------------------------------------------------------------------------
# Load DFT calcularted bandstructure and effective mass
#-------------------------------------------------------------------------

filename = 'InAs_nanowire.nc'

# Read bulk configuration
```

(continues on next page)

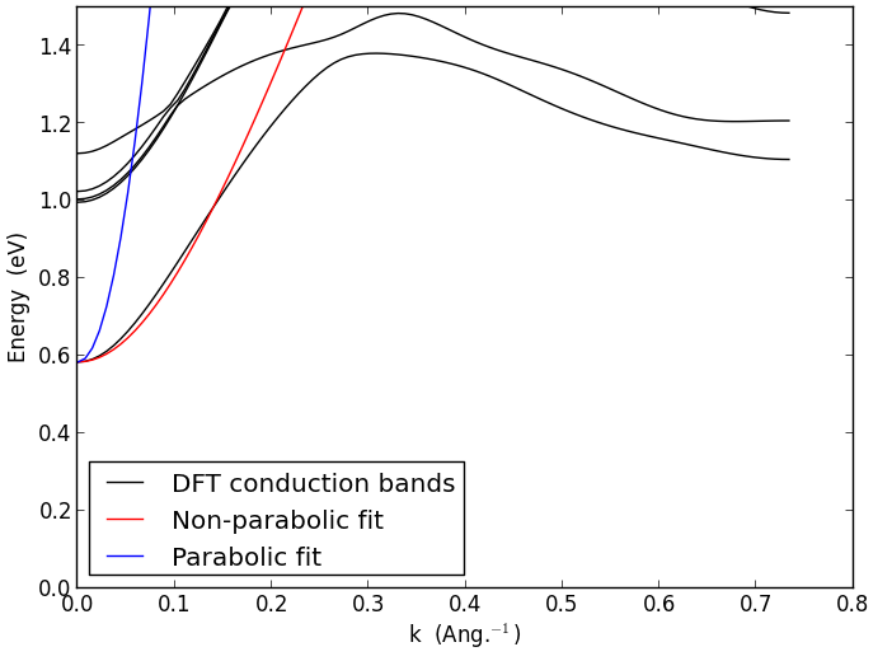
(continued from previous page)

```
configuration = nlread(filename,BulkConfiguration)[0]

# Read bandstructure
bandstructure = nlread(filename,Bandstructure)[-1]

# Read Effective mass
effective_mass = nlread(filename,EffectiveMass)[0]
meff = effective_mass.evaluate(band=1)[0][0][0]

# Get the fractional kpoints
kpoints = bandstructure.kpoints()


# Get reciprocal lattice vectors (used to convert frational k to cartesian k)
G = configuration.bravaisLattice().reciprocalVectors()

# K-points in cartesian coordinates, Ang^-1
k_cart = fractional2cartesian(kpoints, G)

# Get |k| as 1D array
k = numpy.zeros(len(kpoints))
for ii,ki in enumerate(k_cart.inUnitsOf(Angstrom**-1)):
    k[ii] = (ki[0]**2 + ki[1]**2 + ki[2]**2)**0.5

# Add unit
k = k*Angstrom**-1

# Get all the bands
bands = bandstructure.evaluate().inUnitsOf(eV)

# Energies at the Gamma-point
E0 = bands[0,:]

# Index of conduction band
conduction_band_index = numpy.where(E0 > 0.0)[0][0]

# Get the energies of the conduction band
conduction_band = bands[:,conduction_band_index]

# conduction band minimum
cbm = min(conduction_band)

# Evaluate non-parabolic model bandstructure
E_non_parabolic = (-1 + (1 + 2*alpha_bulk*(hbar**2/meff_bulk*(k**2 + numpy.pi**2/(Dx)**2␣
↪+ numpy.pi**2/(Dz)**2 ) ).inUnitsOf(eV) )**0.5 )/(2*alpha_bulk)

# Align conduction band minimum to DFT
E_non_parabolic = E_non_parabolic - min(E_non_parabolic) + cbm

# Evaluate parabolic model bandstructure and align CBM
E_parabolic = (0.5*hbar**2*k**2/meff_bulk).inUnitsOf(eV) + cbm
```

(continues on next page)

(continued from previous page)

```
# Ananlytical effective mass of slab
m_slab = meff_bulk * (1 + 2*alpha_bulk*(hbar**2*numpy.pi**2/(meff_bulk)*(1/Dx**2 + 1/
↪Dz**2 ) ).inUnitsOf(eV) )**0.5
print ''
print 'Analytical nanowire mass = %.4f m_e' %m_slab.inUnitsOf(electron_mass)
print 'DFT nanowire mass = %.4f m_e\n' %meff

# Analytical band gap increase
delta_gap = (-1 +  (1 + 2*alpha_bulk*(hbar**2*numpy.pi**2/(meff_bulk)*(1/Dx**2 + 1/Dz**2␣
↪) ).inUnitsOf(eV) )**0.5)/(2*alpha_bulk)
print 'Analytical band gap increase = %.4f eV' %delta_gap
print 'DFT gap change = %.4f eV' %(1.548-0.354)
print ''


pl.figure()
pl.plot(k,conduction_band,'k',label='DFT conduction bands')
pl.plot(k,bands[:,conduction_band_index+1:conduction_band_index+5],'k')
pl.plot(k,E_non_parabolic,'r',label='Non-parabolic fit ')
pl.plot(k,E_parabolic,'b',label='Parabolic fit')
pl.xlabel('k  (Ang.$^{-1}$)')
pl.ylabel('Energy  (eV)')
pl.ylim([0,1.5])
pl.legend(loc=0)

pl.show()
```



As for the InAs slab, the lowest conduction band in the nanowire is much better described by the the non-parabolic

model than by the parabolic/effective mass model. Although the agreement between the non-parabolic model and the DFT band structure is not quite as good as for the slab, it still provides a very reasonable estimate.

References

3.6.2 Opening a band gap in silicene and bilayer graphene with an electric field

Version: 2017

In this tutorial you will learn how to insert metallic gate electrodes in an QuantumATK bulk calculation and use them to apply electric fields. You will use these features to study the opening of a band gap in bilayer graphene and in silicene under the presence of an electric field. It is assumed that you are familiar with the basic functionalities of **QuantumATK**.

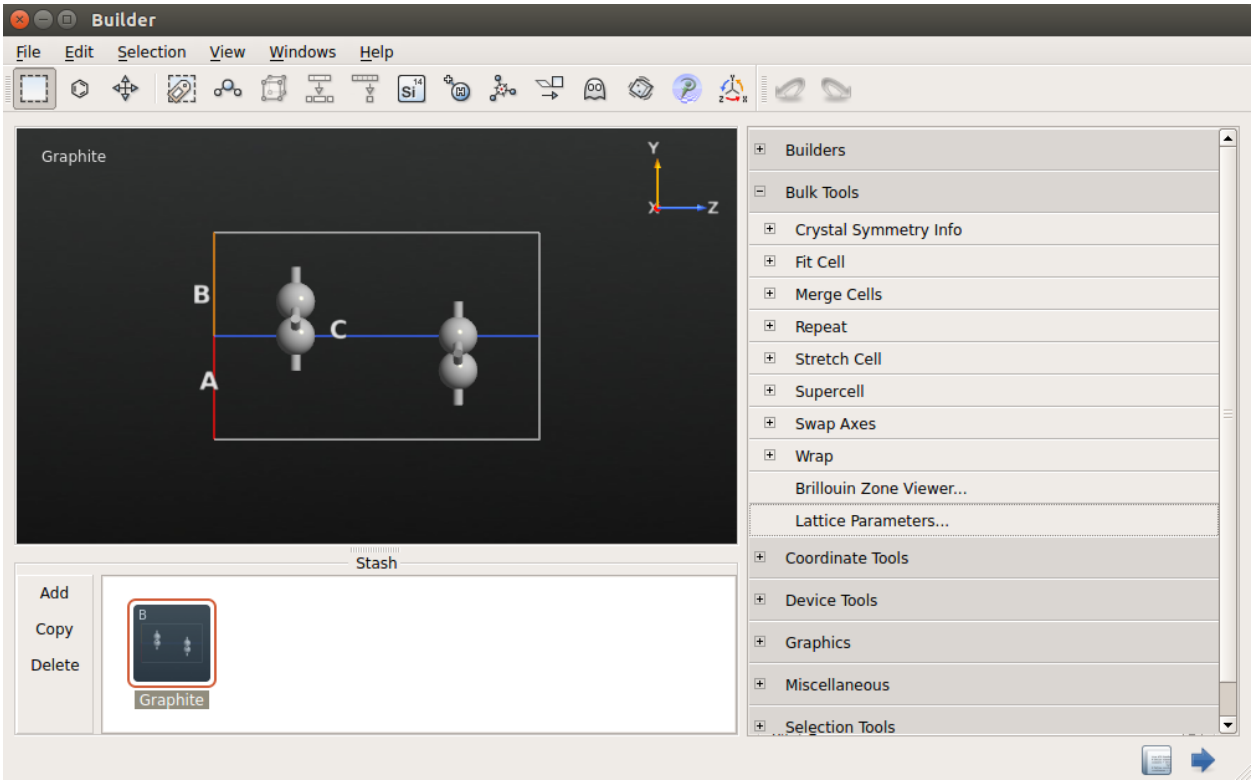
Bilayer graphene

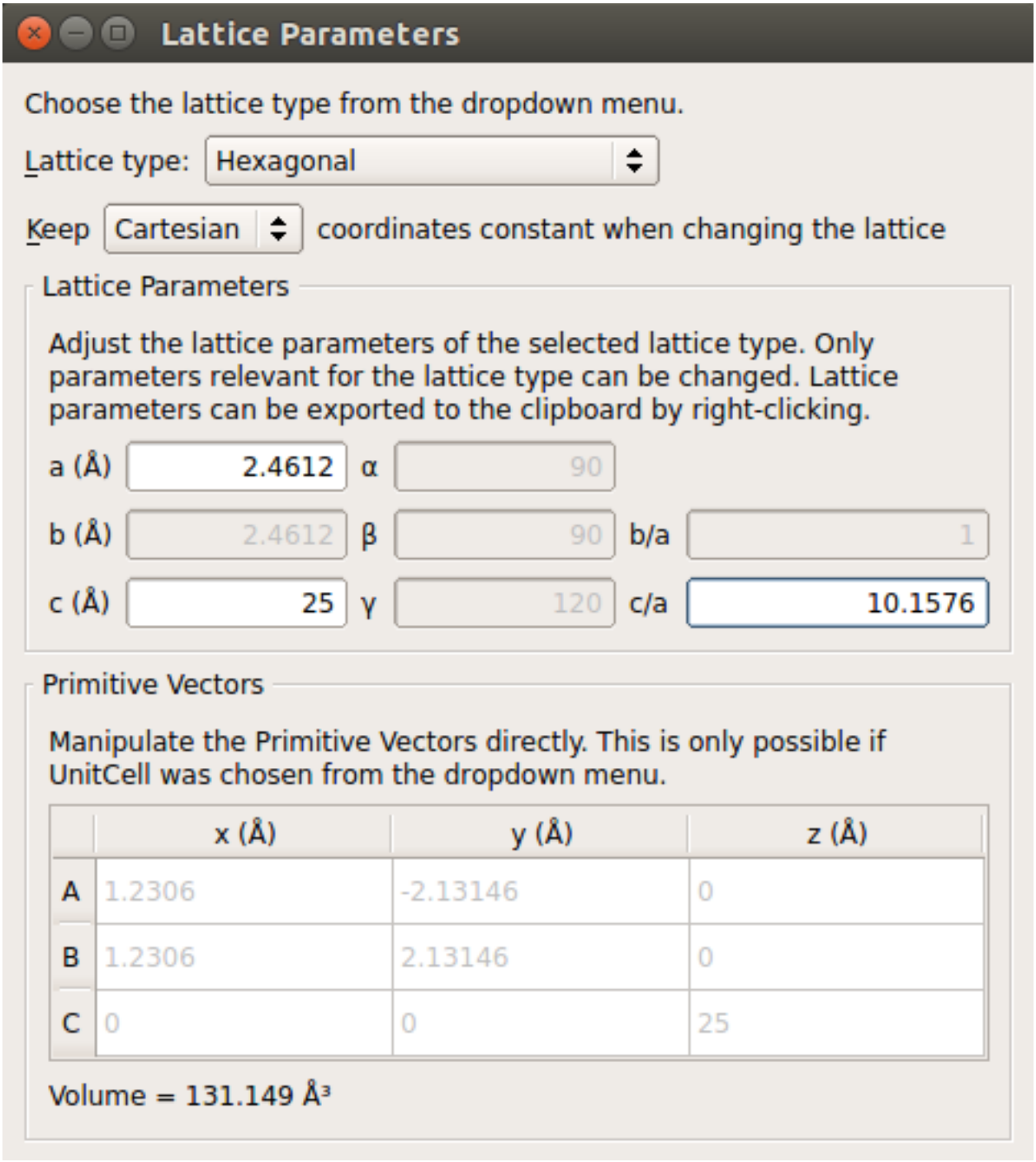
Building a bilayer graphene structure

1. Create a new empty project and open **Builder** (icon).

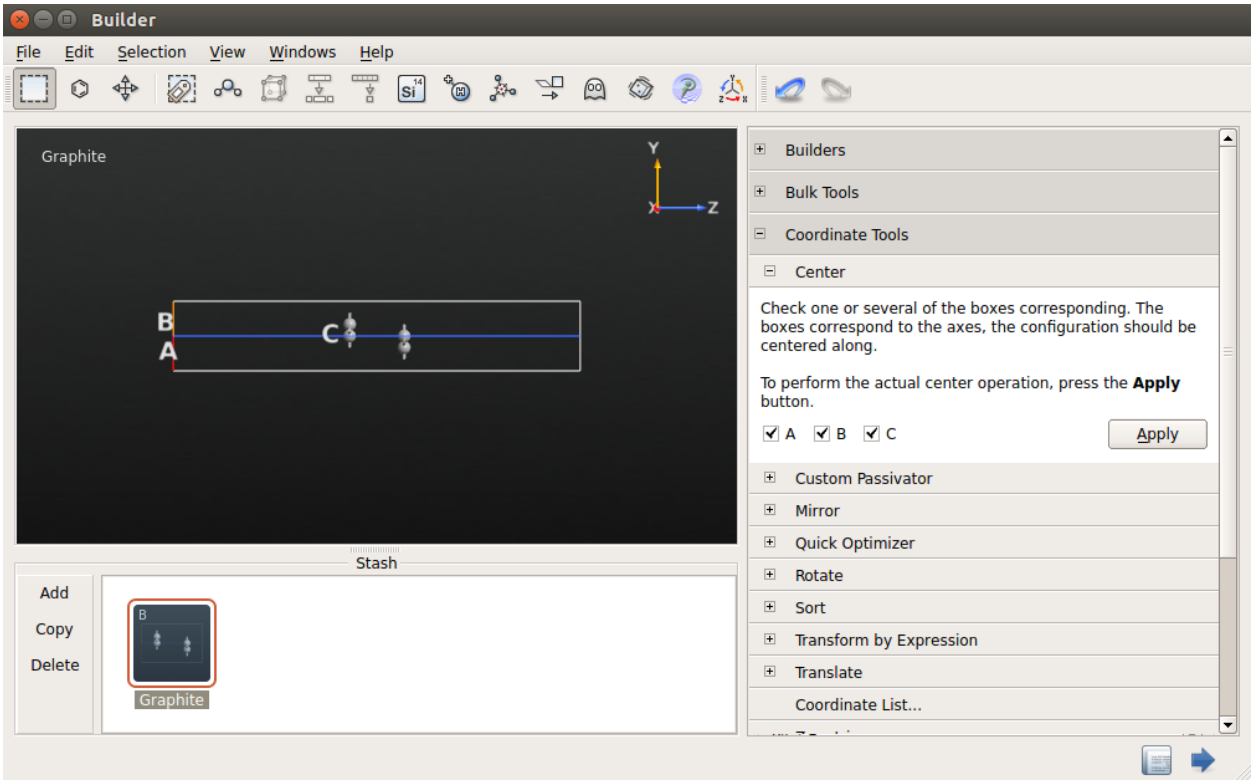
2. Insert graphite (not graphene!) from **Database**.

3. Open *Bulk Tools* → *Lattice Parameters*. Select Cartesian coordinates and set the value of *c* lattice constant to be 25 Å.



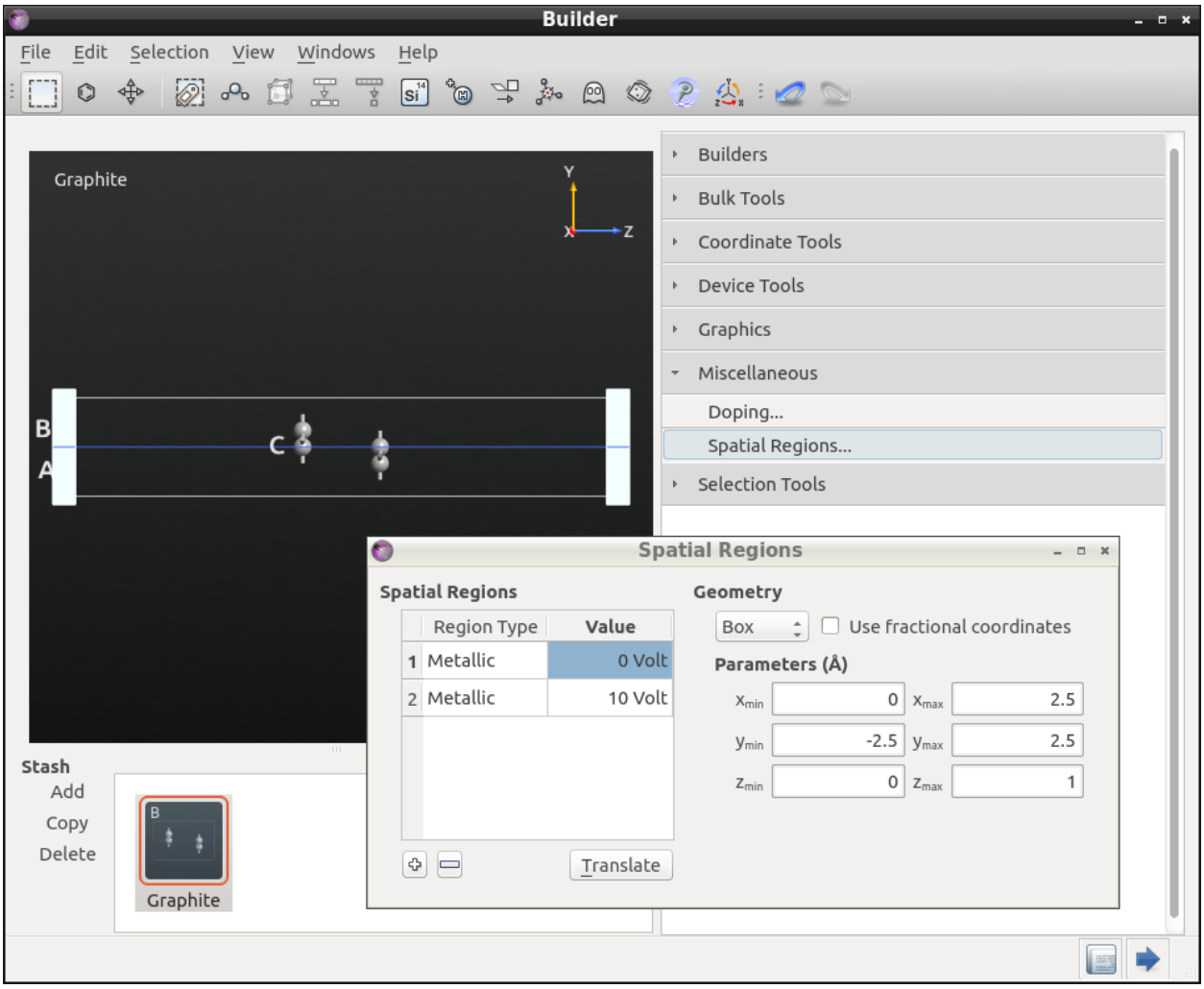


4.Go to *Coordinate Tools* → *Center* and click *Appy* to center the system.



5. To insert the metal electrodes, open *Miscellaneous* → *Spatial Regions*, right-click the table (the area under the headers Region Type/Value) and click *Insert region* twice.

6. Set the voltage to 0 V for the first metallic region, and 10 V for the second. Change their size so that they cover the entire hexagonal unit cell (see the figure below). It does not matter if the regions stick outside the cell, those parts will be ignored in the calculation anyway.

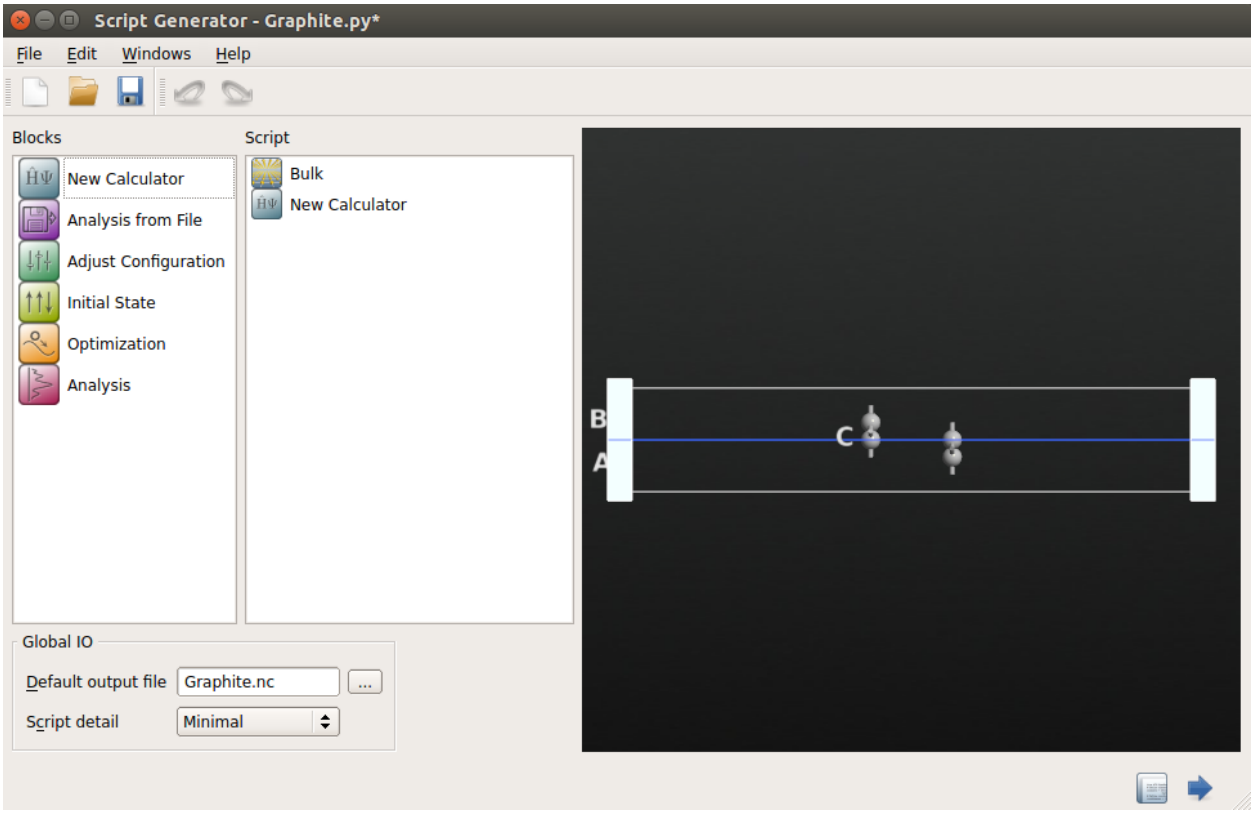


The applied voltage results in 4 V/nm in the z direction, just above the field that has been found in experiments to open a band gap.

Calculation and analysis

We will then analyze the band structure of the graphene bilayer under the static electric field.

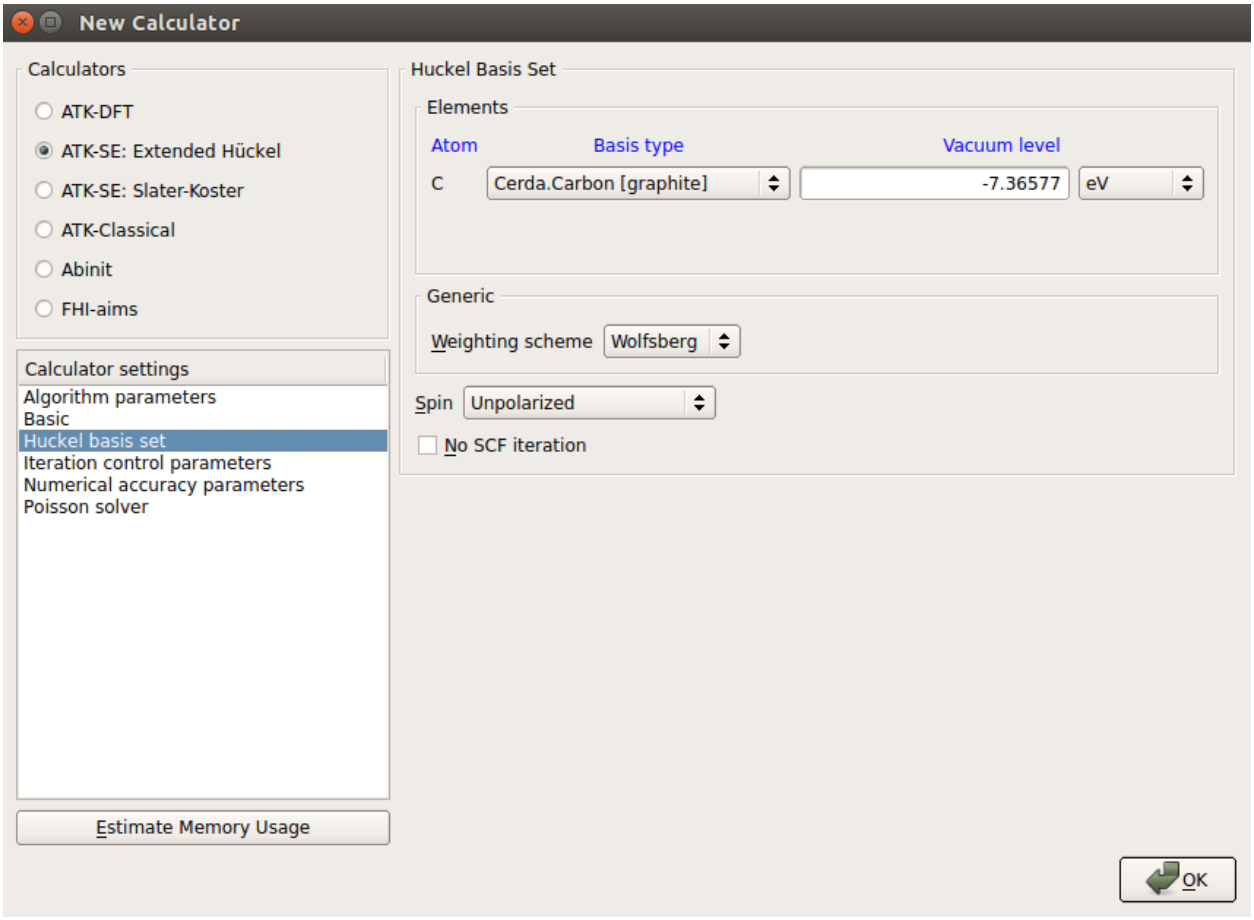
1. Send the graphene bilayer structure to **Script Generator**.
2. Insert a **New Calculator** block



3. Open the **New Calculator** block, and choose the *ATK-SE: Extended Hückel* method. For good accuracy, set the k-points to be 9x9x1.

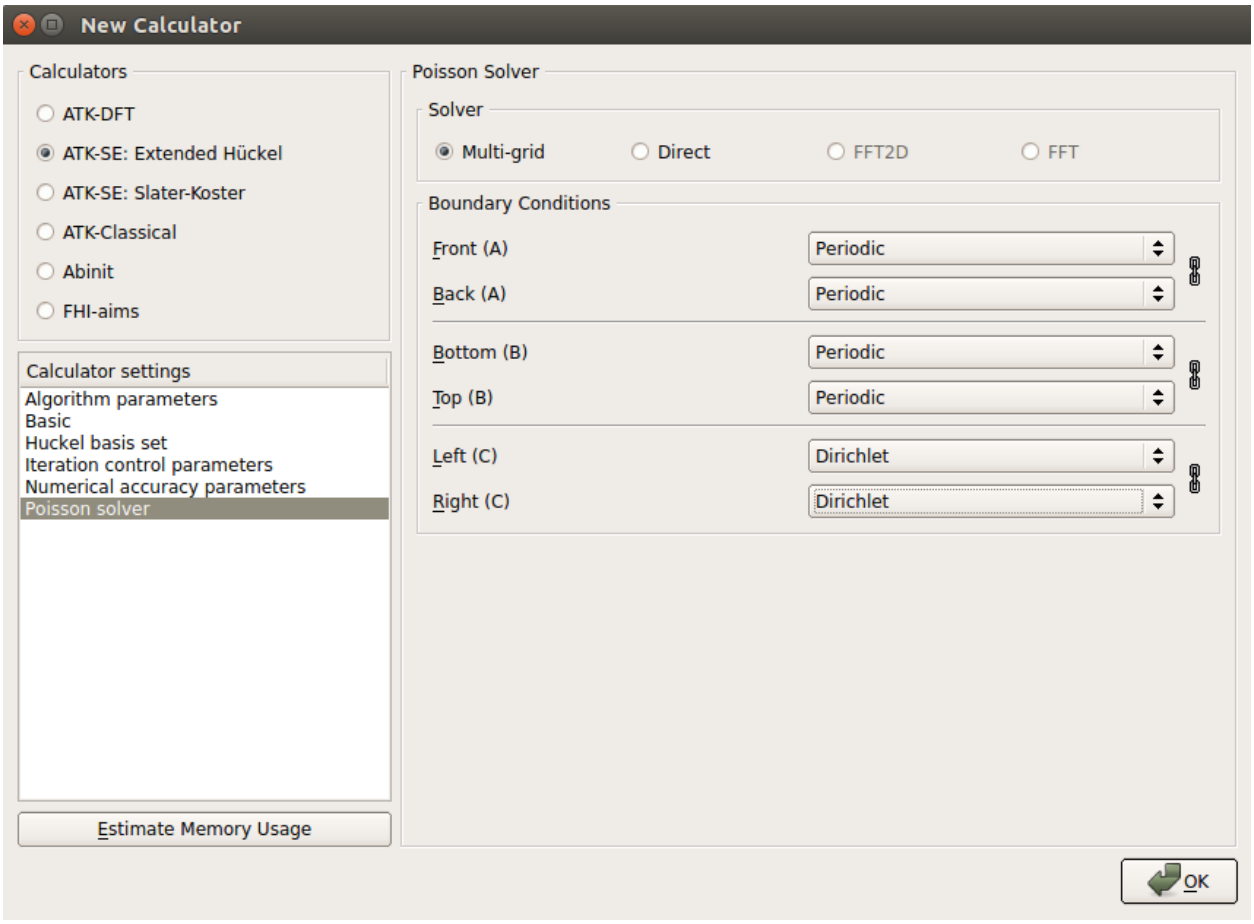
4. Under *Huckel basis set* select the *Cerda. Carbon [graphite]* basis set, which describes the sp2 bonding of graphene very well.

5. Untick the *No SCF iteration* option to carry out the computation self-consistently.



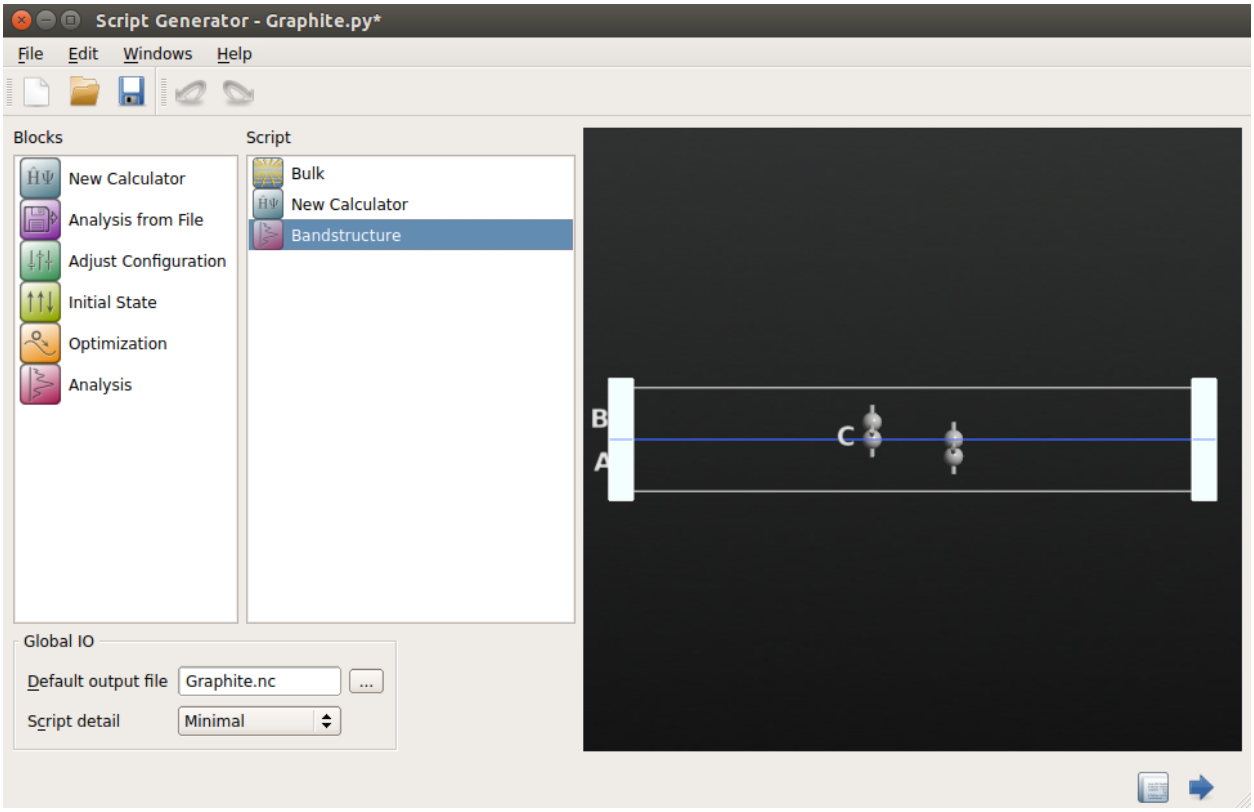
Warning: It is extremely important to include self-selconsistency; otherwise the field will not have any effect at all!

6. Go to *Poisson solver*, set *Multi-grid* in **Solver**, and choose *Dirichlet* in **Boundary Conditioins** on both sides in the C direction.

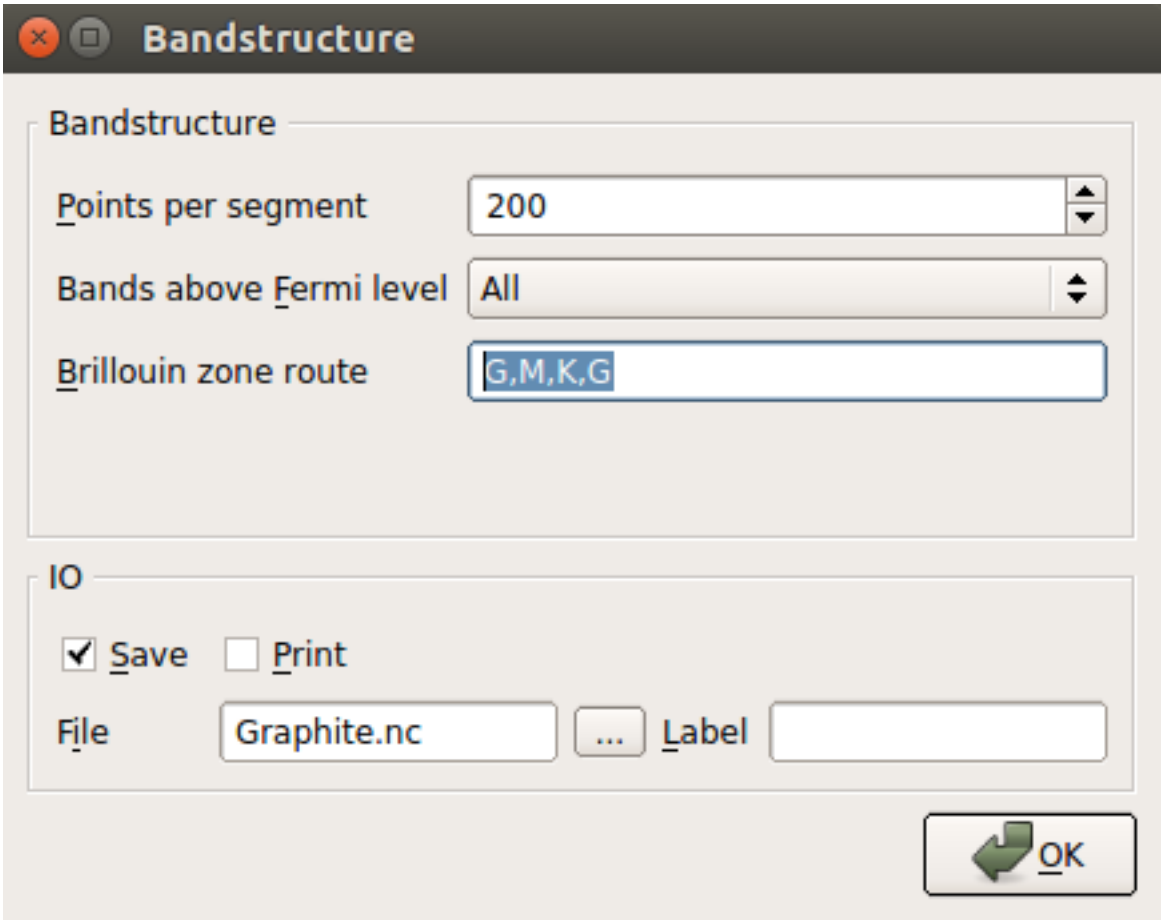


Note: For this particular calculation, you can choose any boundary condition in the C direction since the metal gates will fix the potential at the boundaries anyway, but *Dirichlet* is the physically correct condition when a constant value of the potential is desired at the boundary.

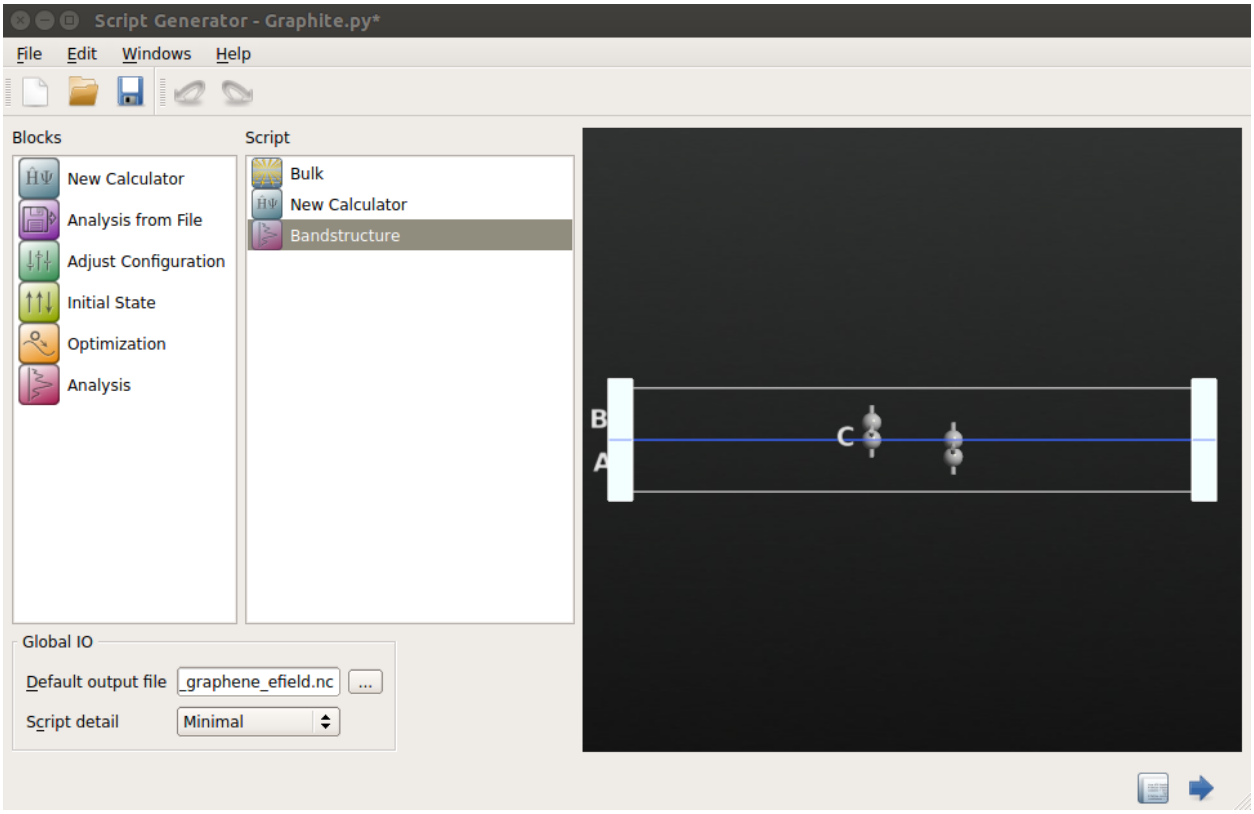
7. Add a **Bandstructure** block by clicking *Analysis* → *Bandstructure*.



8.Double-click the **Bandstructure** block and change the route to G, M, K, G. Also increase the number of points per segment to 200.

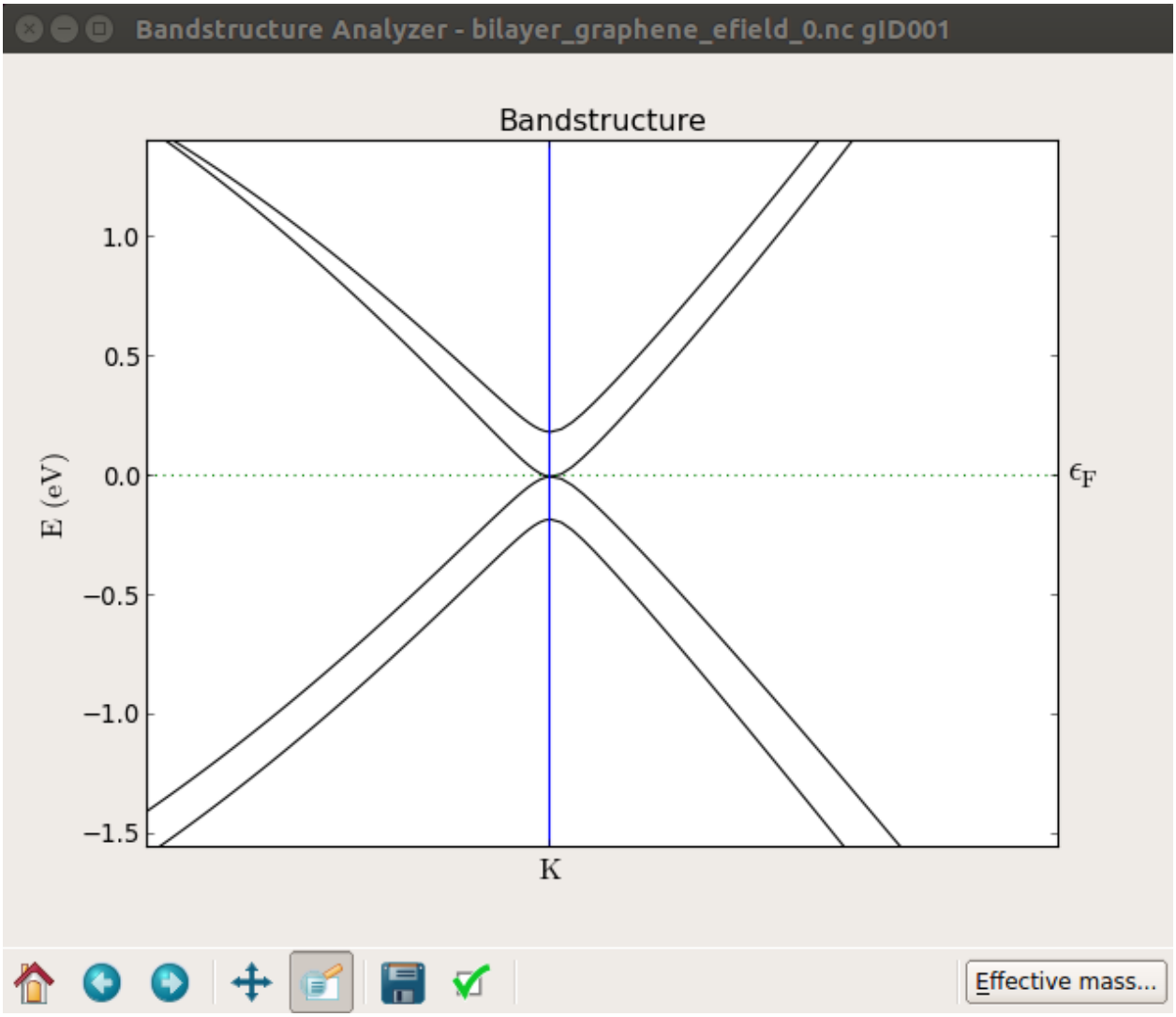


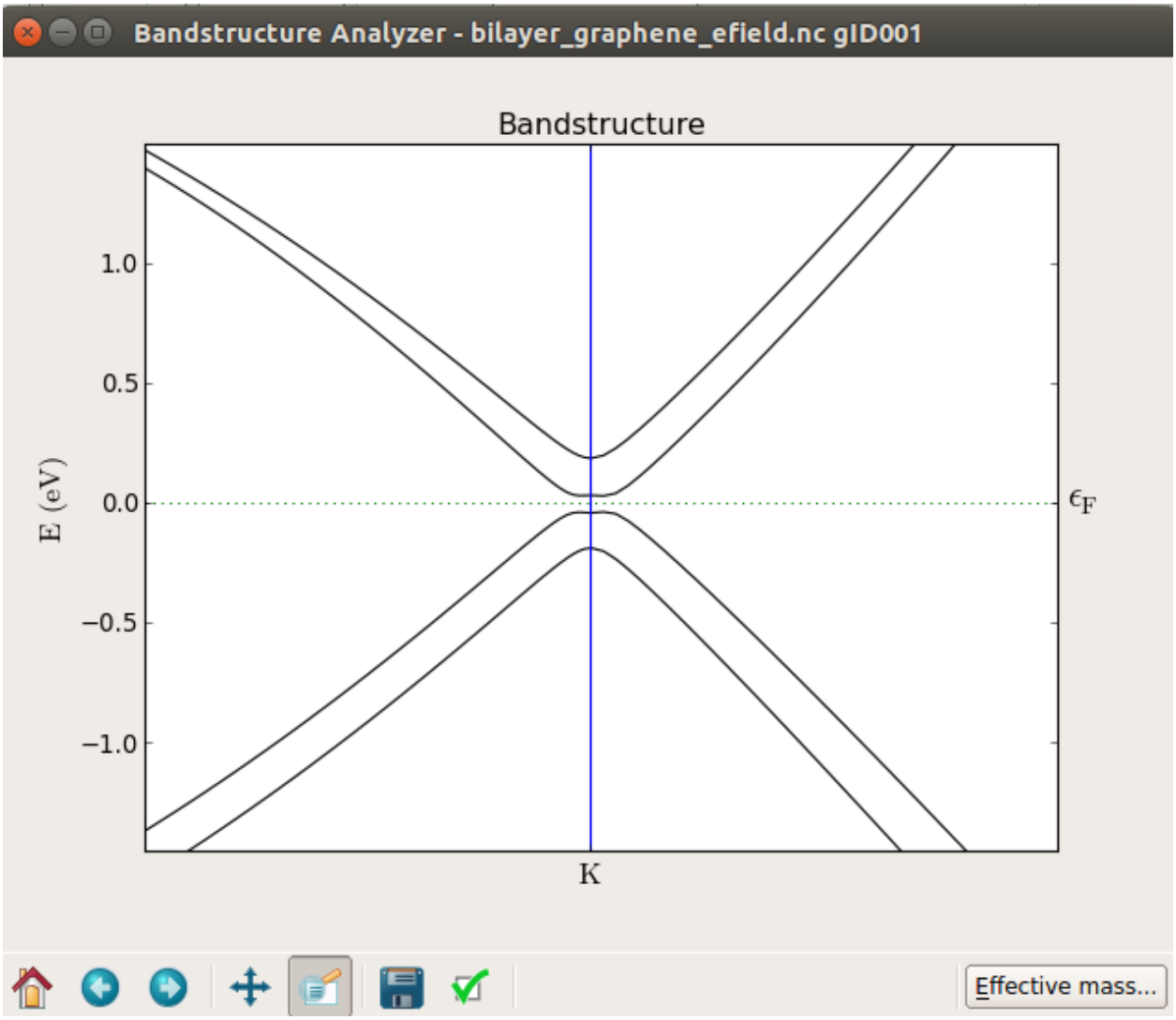
9.Name the output file *bilayer_graphene_efield.nc*.

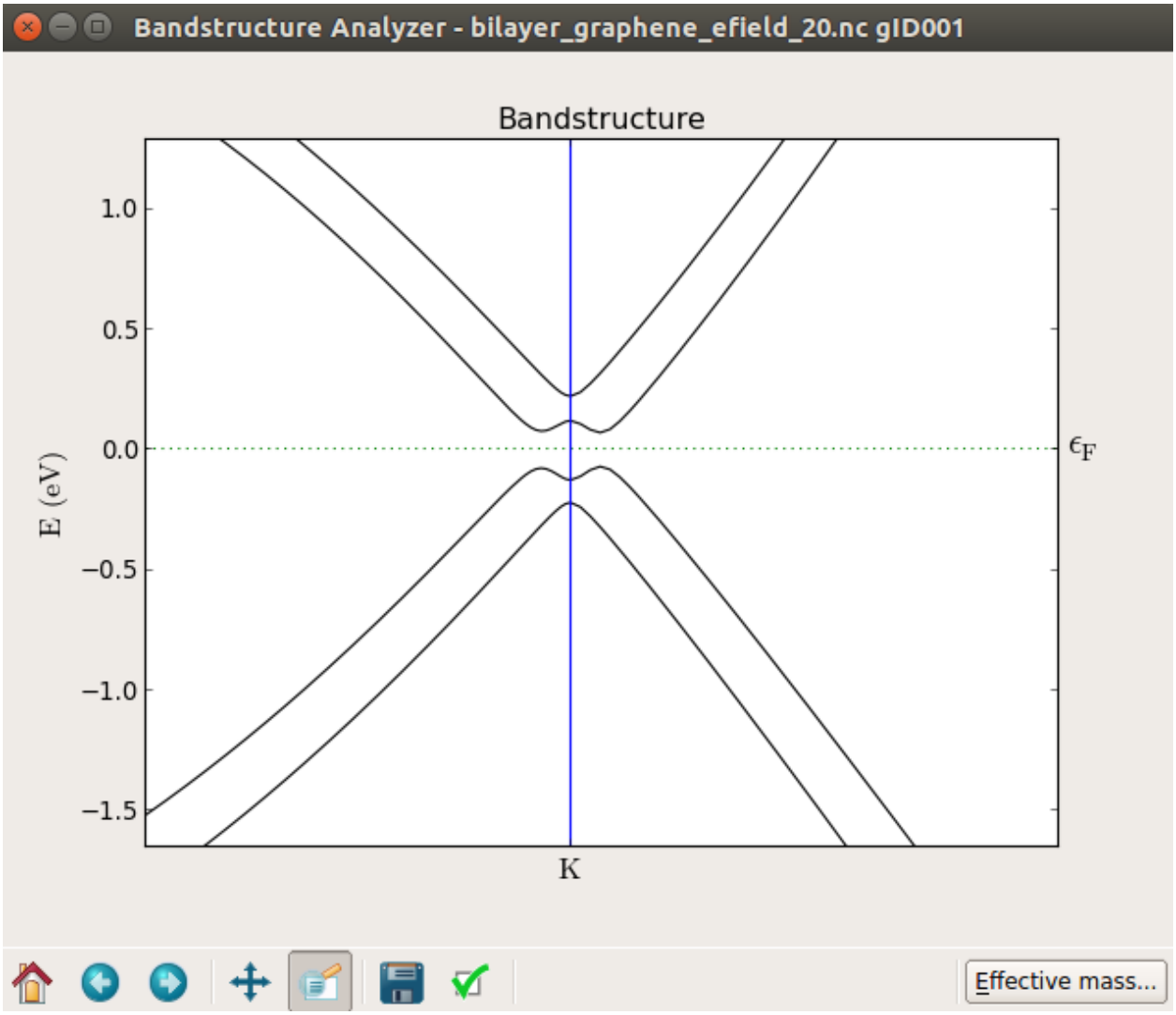


10. Send the calculation to **Job Manager**, save the Python script in the window that appears and run the calculation. It should take no more than 20-30 seconds to run.

    After the calculation, visualize the band structure and zoom in around the K point, where a band gap is opening with increasing the strength of the electric field. The following figures, from top to bottom, show the band structure of the bilayer graphene with an applied electric field of 0, 10, and 20 Volts, respectively.


Bandstructure Analyzer - bilayer_graphene_efield_0.nc gID001
Bandstructure
1.0
0.5
0.0
−0.5
−1.0
−1.5
E (eV)
ϵ_F
K
Effective mass...


Bandstructure Analyzer - bilayer_graphene_efield.nc gID001
Bandstructure
1.0
0.5
0.0
−0.5
−1.0
E (eV)
ϵ_F
K
Effective mass...

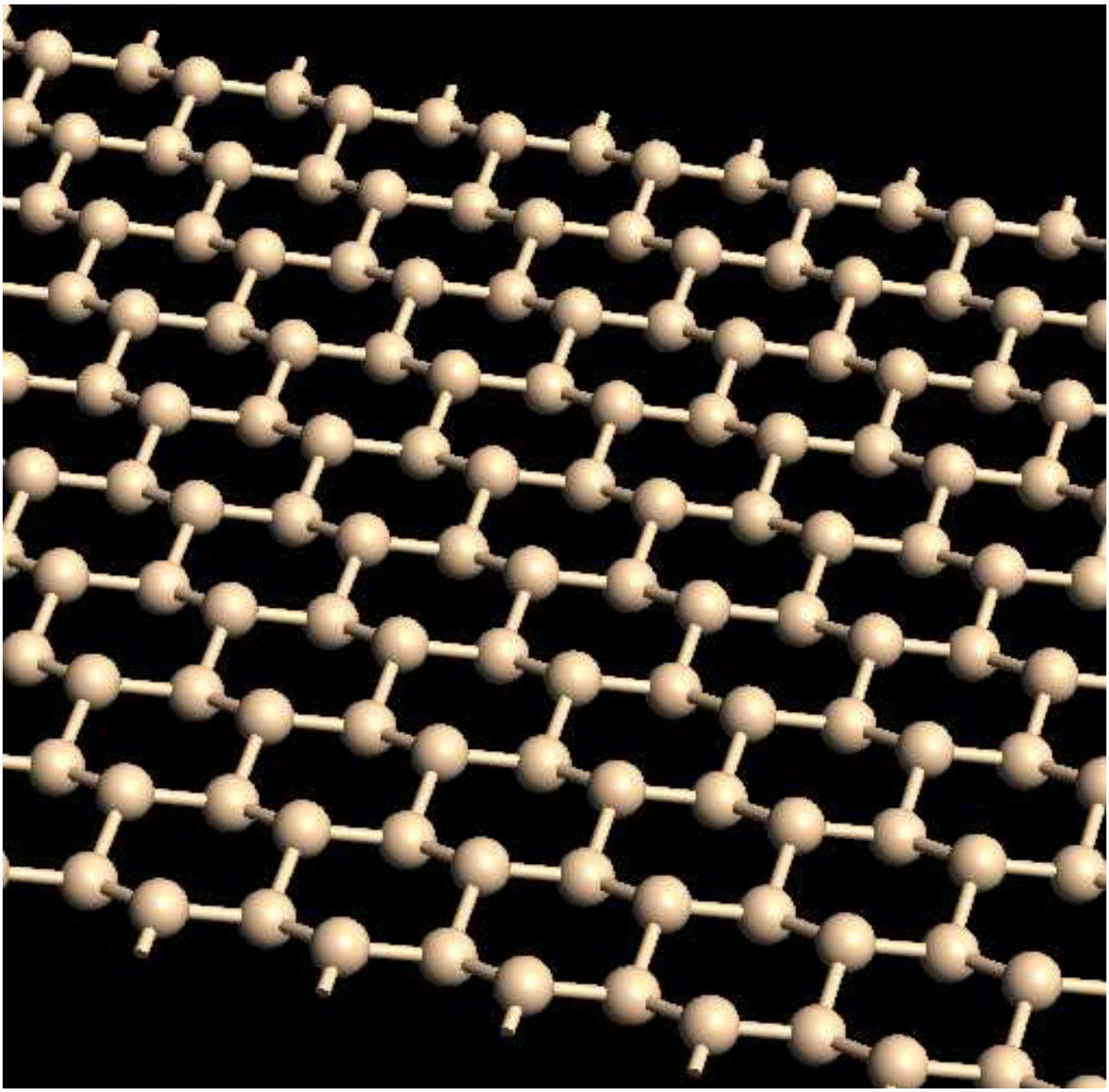


Note:

- You can use DFT as well, the results will be similar.
- No attention was paid to the interlayer distance. You can change it a bit to see how the band structure changes. Note, however, that you cannot use DFT/GGA to optimize this distance because no energy minimum will be found, due to the absence of dispersion corrections (the layers are bound by Van der Waals forces, not captured in standard GGA). LDA works, fortuitously, because of a cancellation of errors, but the results cannot really be trusted.

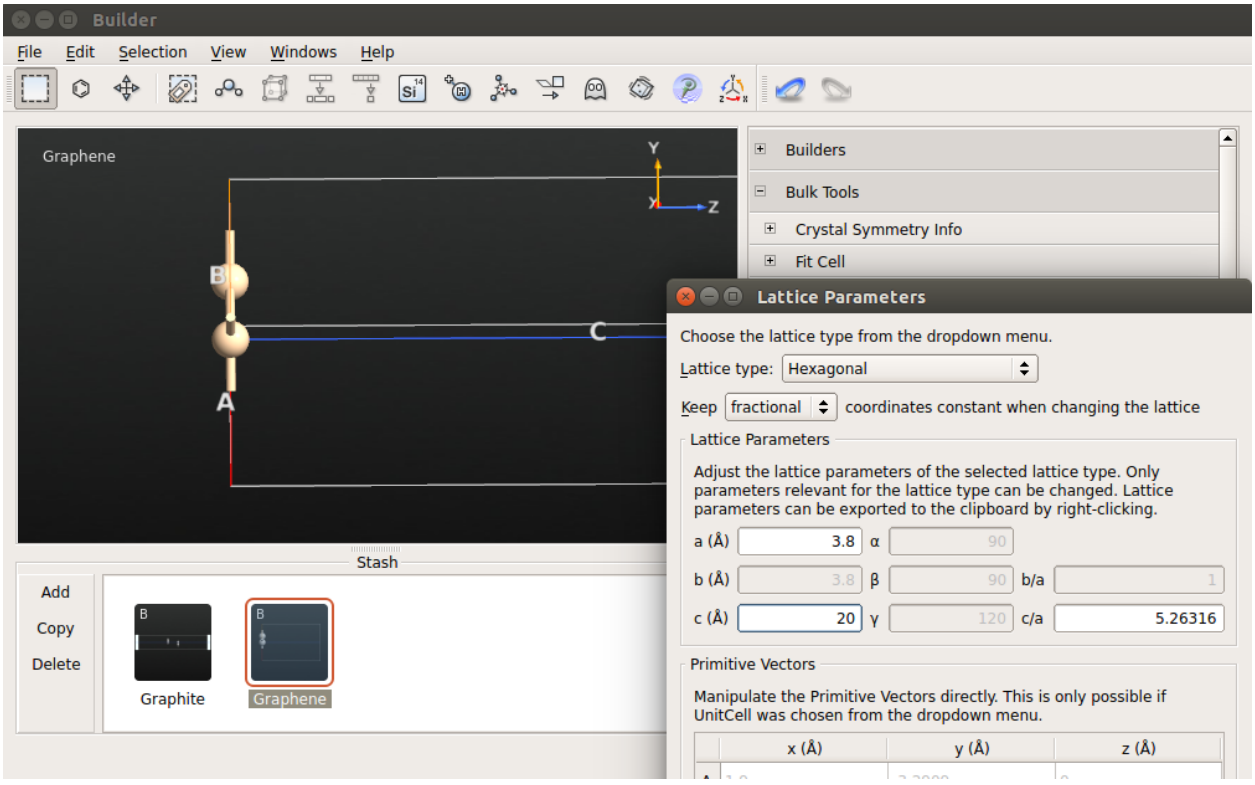
Silicene

To study the effect of electric fields on silicene you first need a good model of this material. The approach will be to start with graphene and turn it into silicene, optimize the geometry, and then repeat the same steps followed to calculate the bilayer graphene under the presence of an electric field.

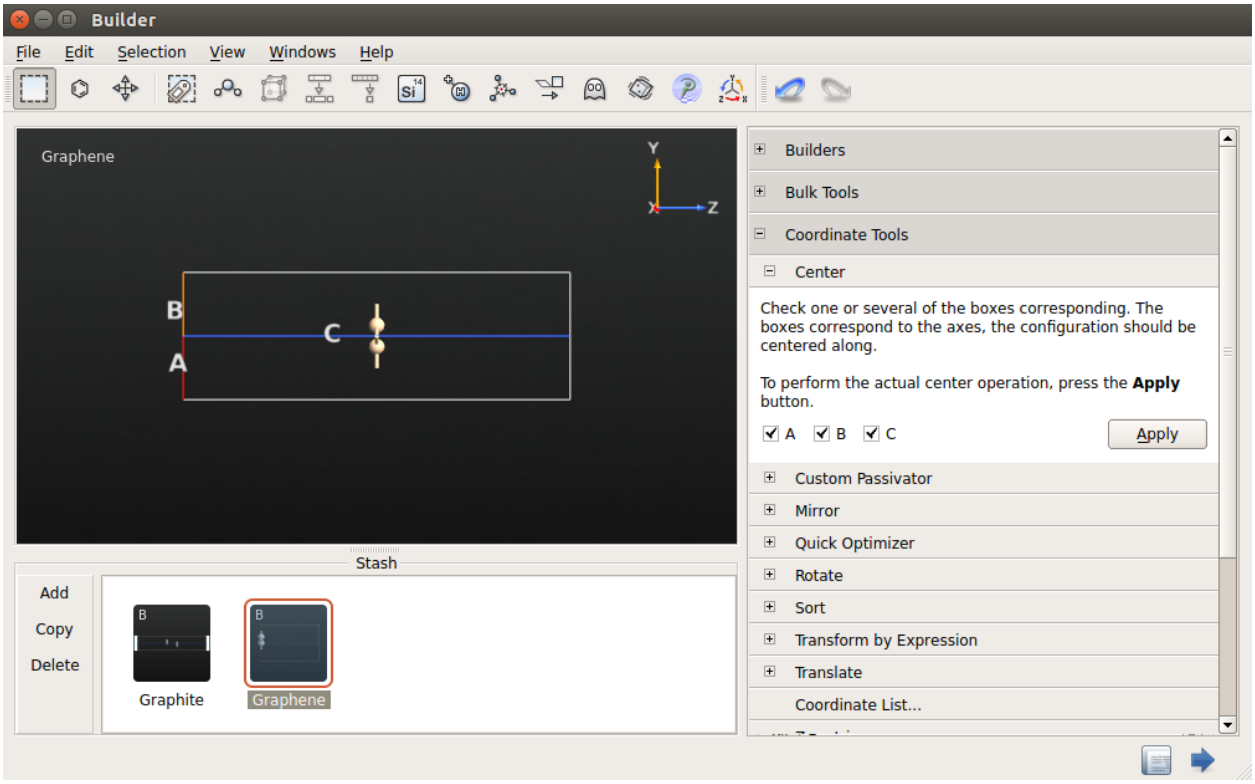


Optimizing the geometry

1. In **Builder**, add graphene (not graphite this time) from **Database** to **Stash** following the same steps explained above for the bilayer graphene.
2. Change the two C atoms to Si by selecting them (using the left mouse button while holding down the `Ctrl` button of your keyboard) and using the *Periodic Table* tool .
3. The lattice constant of silicene is not the same as for graphene, but rather than guess it, it will simply be part of the geometry optimization to determine it. To start from a reasonable guess, click *Bulk Tools* → *Lattice Parameters*. Make sure to keep the fractional coordinates constant, set the lattice parameter a to be 3.8 Å and increase the c lattice parameter to 20 Å.



4. In **Builder**, go to *Coordinate Tools* → *Center*, in order to center the configuration.



5. The flat geometry where both Si atoms are in the same plane is a local minimum in the potential energy surface. Therefore, click the *Rattle* tool a few times to add a small perturbation to the coordinates. This ensures that the geometry optimization will converge to the global minimum.

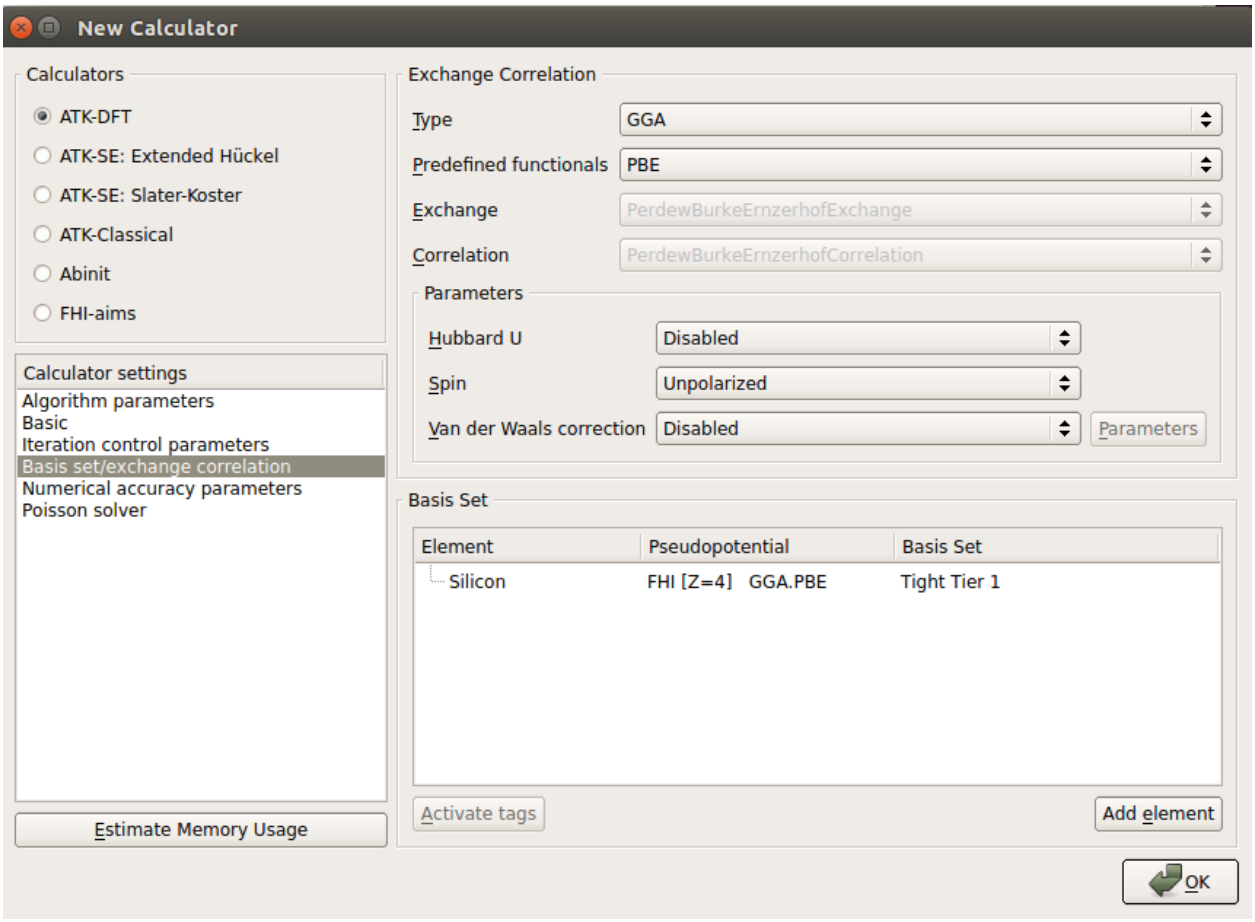
6. Send the geometry to **Script Generator** by using the *Send To* button .

7. In **Script Generator**, insert **New Calculator** and **OptimizeGeometry** blocks by double-clicking the icons in the *Blocks* panel:

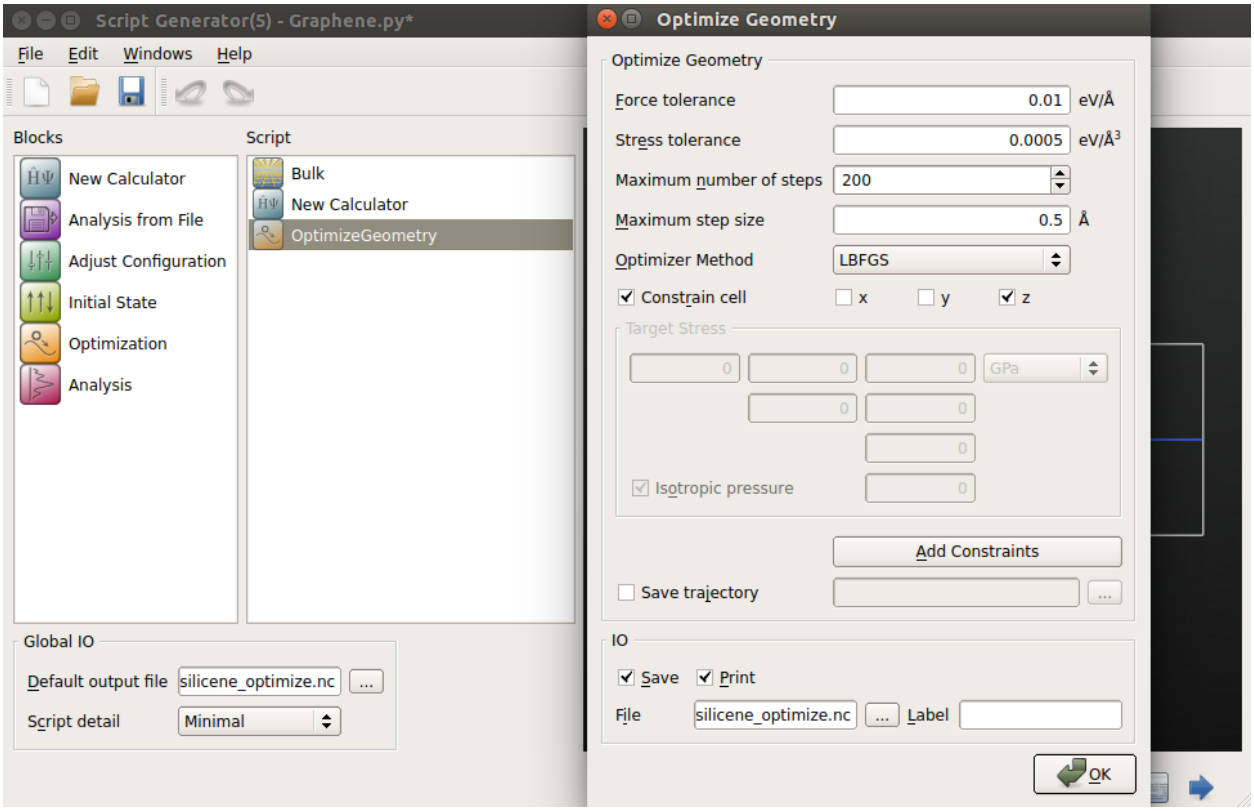
- New Calculator.
- *Optimization* → *OptimizeGeometry*.

8. Use 21x21x1 k-points for good accuracy, and choose GGA-PBE for exchange-correlation. You can in principle use the default DoubleZetaPolarized basis set, but here you will use the Tight Tier 1 basis set, which provides slightly better

results.



9. In the **OptimizeGeometry** block, set the force tolerance to be small (0.01 eV/Å) and the stress tolerance to be even smaller (0.0005 eV/Å^3). Unconstrain the cell in the x and y directions.



10. Name the output file *silicene_optimize.nc* and save the Python script.

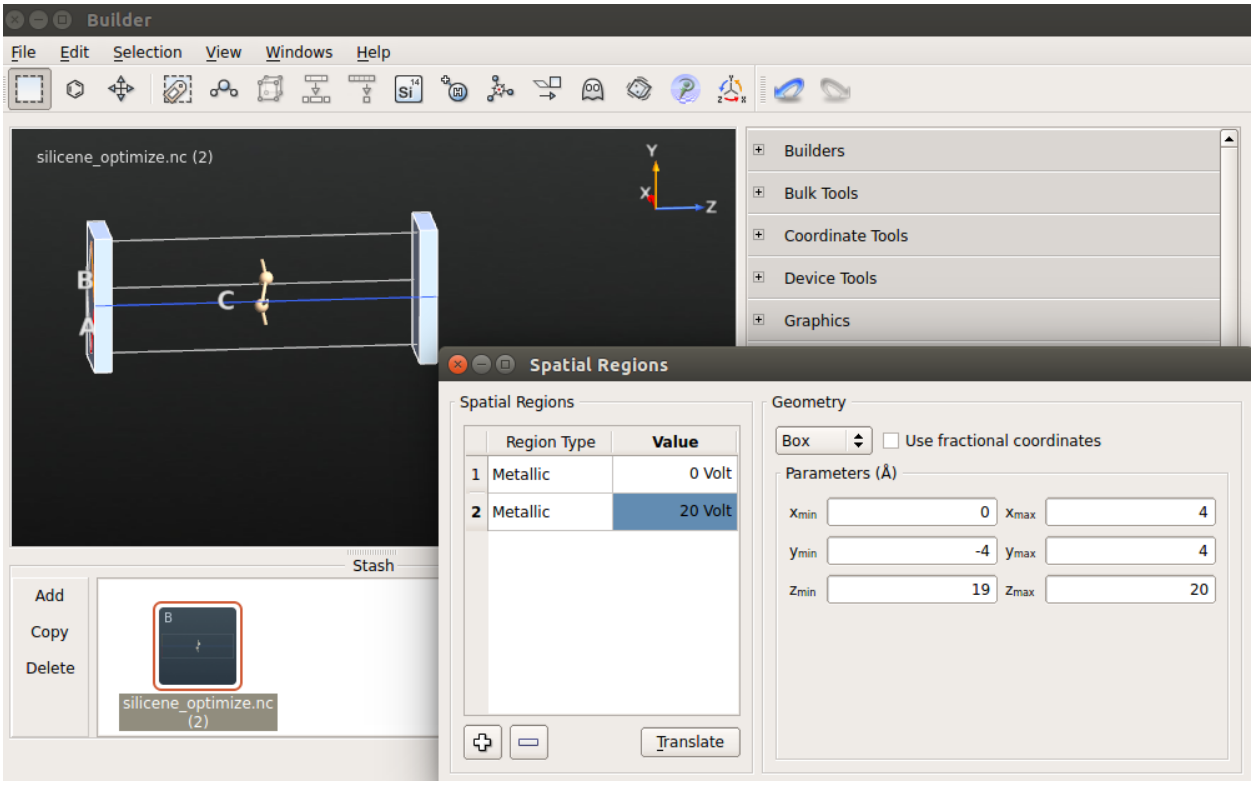
11. Send the script to **Job Manager**, save the script and run the job. The calculation will take about 10 minutes. When it finishes, drop the Bulk Configuration with ID *gID001* on **Builder**, and inspect the geometry. You will find a nicely optimized silicene sheet with a buckling of about 0.5 Å and an optimized lattice constant of 3.86 Å. Both values are in good agreement with those reported in the literature in[1] and[2].

[1] S. Cahangirov, M. Topsakal, E. Akturk, H. Sahin, and S. Ciraci, Two- and One-Dimensional Honeycomb Structures of Silicon and Germanium, Physical Review Letters 102, 236804 (2009)

[2] N. D. Drummond, V. Zolyomi, and V. I. Falko, Electrically tunable band gap in silicene, Physical Review B 85, 075423 (2012)

Band structure

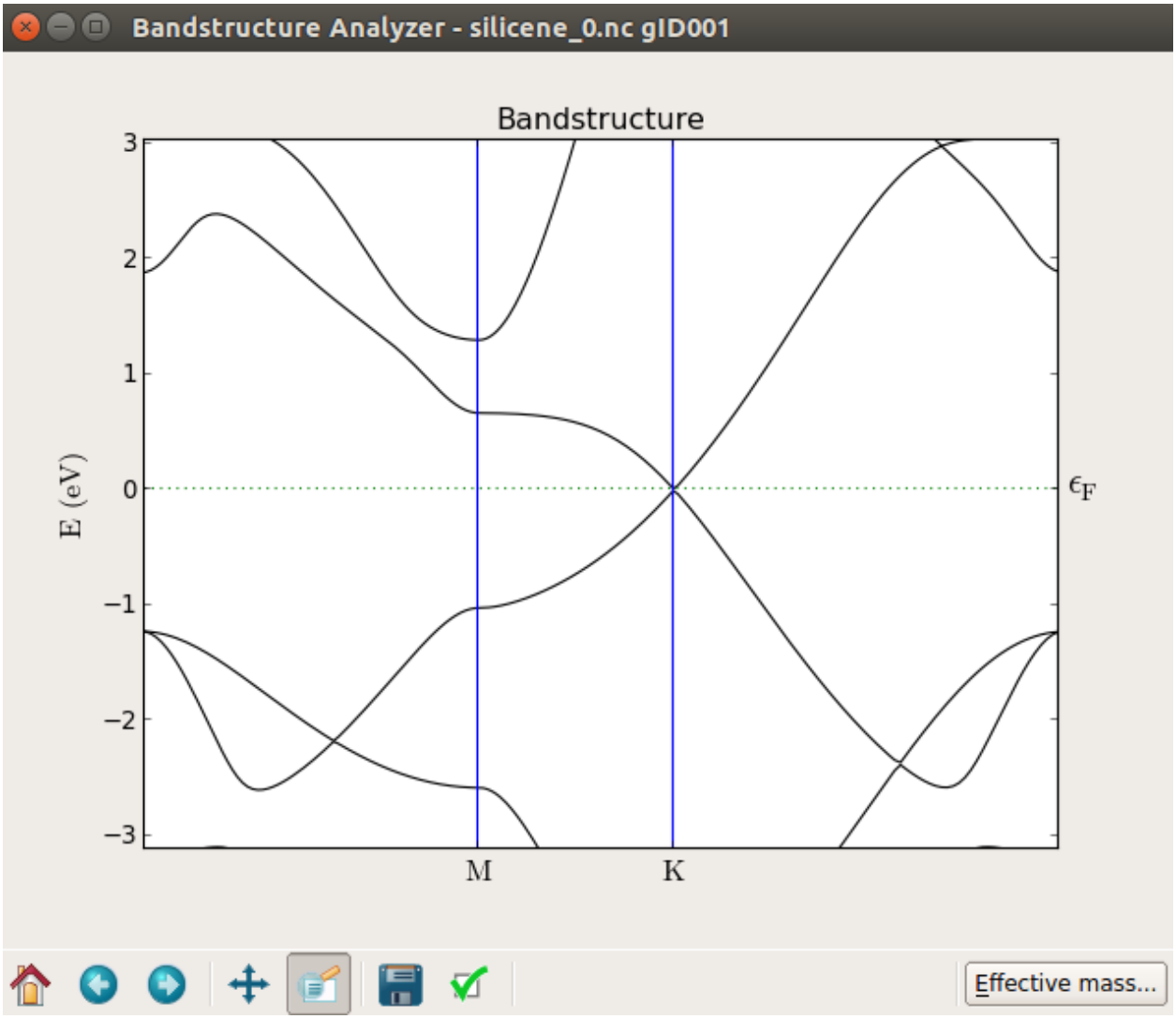
Using the optimized structure, repeat the steps above for the bilayer graphene to insert metallic gates and compute the band structure with electric fields. Notice that, in the case of silicene, you need to make the spatial region larger in the XY plane, since the cell is also larger. Use 20 V as the voltage on the second electrode.

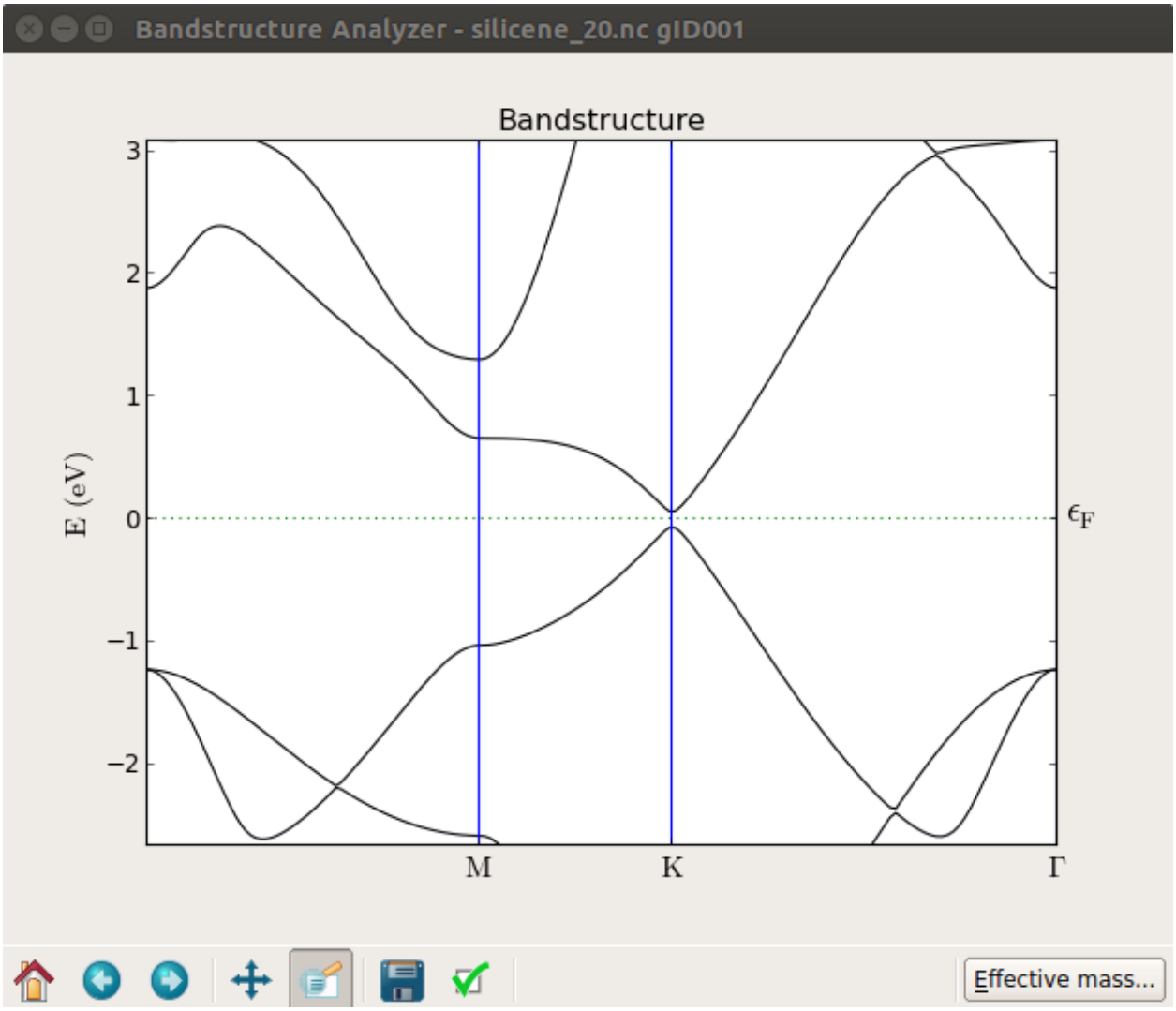


Note: You should use the same calculator settings as for the optimization. The calculation will take less than a minute.

In the plot below you can see the band structure at zero field (top) and with 20 V applied on the gate (bottom), which produces a band gap of about 0.1 eV. By trying different values of the voltage you will find that the band gap increases essentially linearly with the field as shown in[3].

[3] R. Zhou, Structural And Electronic Properties of Two-Dimensional Silicene, Graphene, and Related Structures, Master Thesis, Wright State University, Electrical Engineering, Dayton OH, USA (2012)


Bandstructure Analyzer - silicene_0.nc gID001
Bandstructure
E (eV)
3
2
1
0
−1
−2
−3
M
K
ϵ_F
Effective mass...



Notice that in this case, if you look closely, the band gap is not exactly at the K point.

Note: If you apply a transverse field to a single graphene layer, no gap opens up. The reason is that the graphene is completely flat, and so the field just gives a constant shift of the potential. In the case of silicene the gap appears because the sheet is naturally buckled.

References

More reading

- Z. Ni et al., Tunable Bandgap in Silicene and Germanene, Nano Letters 12, 113-118 (2011)
- C. Xu et al., Giant magnetoresistance in silicene nanoribbons, Nanoscale 4, 3111-3117 (2012)
- Y. Wang et al., Half-metallic Silicene And Germanene Nanoribbons. Towards High-performance Spintronics Device, Nano 7, 1250037 (2012)
- L. Pan et al., Thermoelectric properties of armchair and zigzag silicene nanoribbons, Phys. Chem. Chem. Phys. 14, 13588 (2012)

•H. Li et al., High performance silicene nanoribbon field effect transistors with current saturation, European Physical Journal B 85, 1 (2012)

•R. Quhe et al., Tunable and sizable band gap in silicene by surface adsorption, Sci. Rep. 2, 853 (2012)

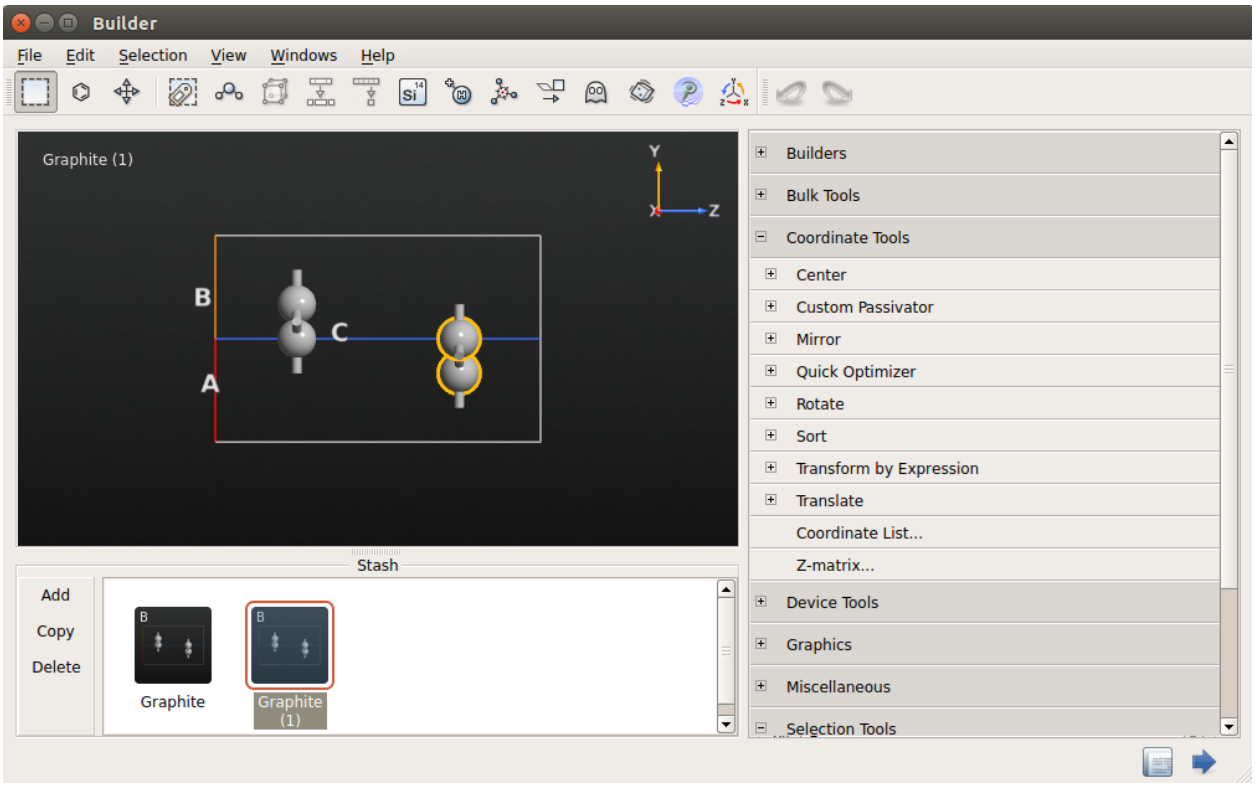
For a similar study on hexagonal boron-nitride (hBN), see

•K. Tang et al., Electronic and transport properties of a biased multilayer hexagonal boron nitride, European Physical Journal B 85, 301 (2012)

3.6.3 Commensurate supercells for rotated graphene layers

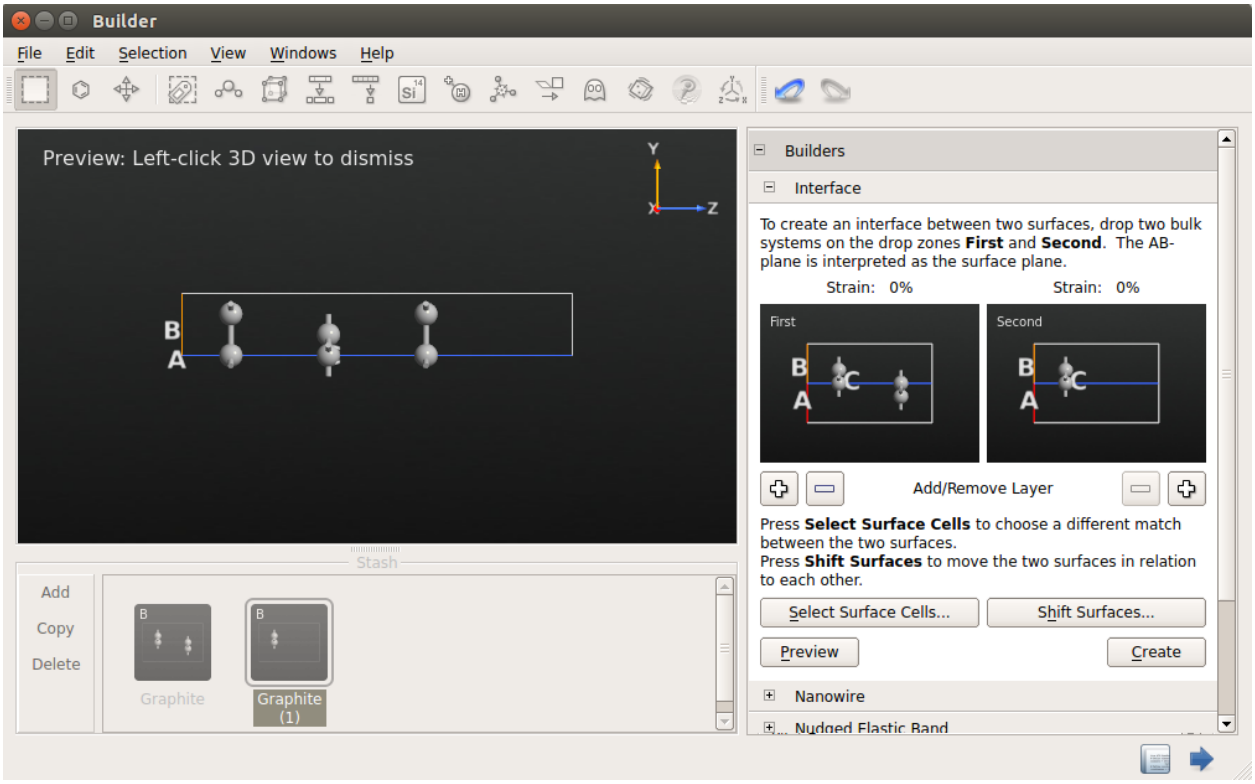
QuantumATK: Interface builder Platforms: In this tutorial you will learn how to set up a commensurate supercell for rotated graphene layers. You will consider a case from Ref.[1] as an example in which the topmost layer of a three layer graphene stack is rotated by 21.8ř with respect to the other two layers.

1. In the **Builder**, Click *Add* → *From Database*. Locate *graphite* and add it to the **Stash** by clicking the icon in the lower right-hand corner.
2. In the **Stash**, select the newly added item and click *Copy* to make an identical copy of the graphite structure.
3. Select the copied structure and delete the uppermost layer of carbon atoms highlighted in the picture below.

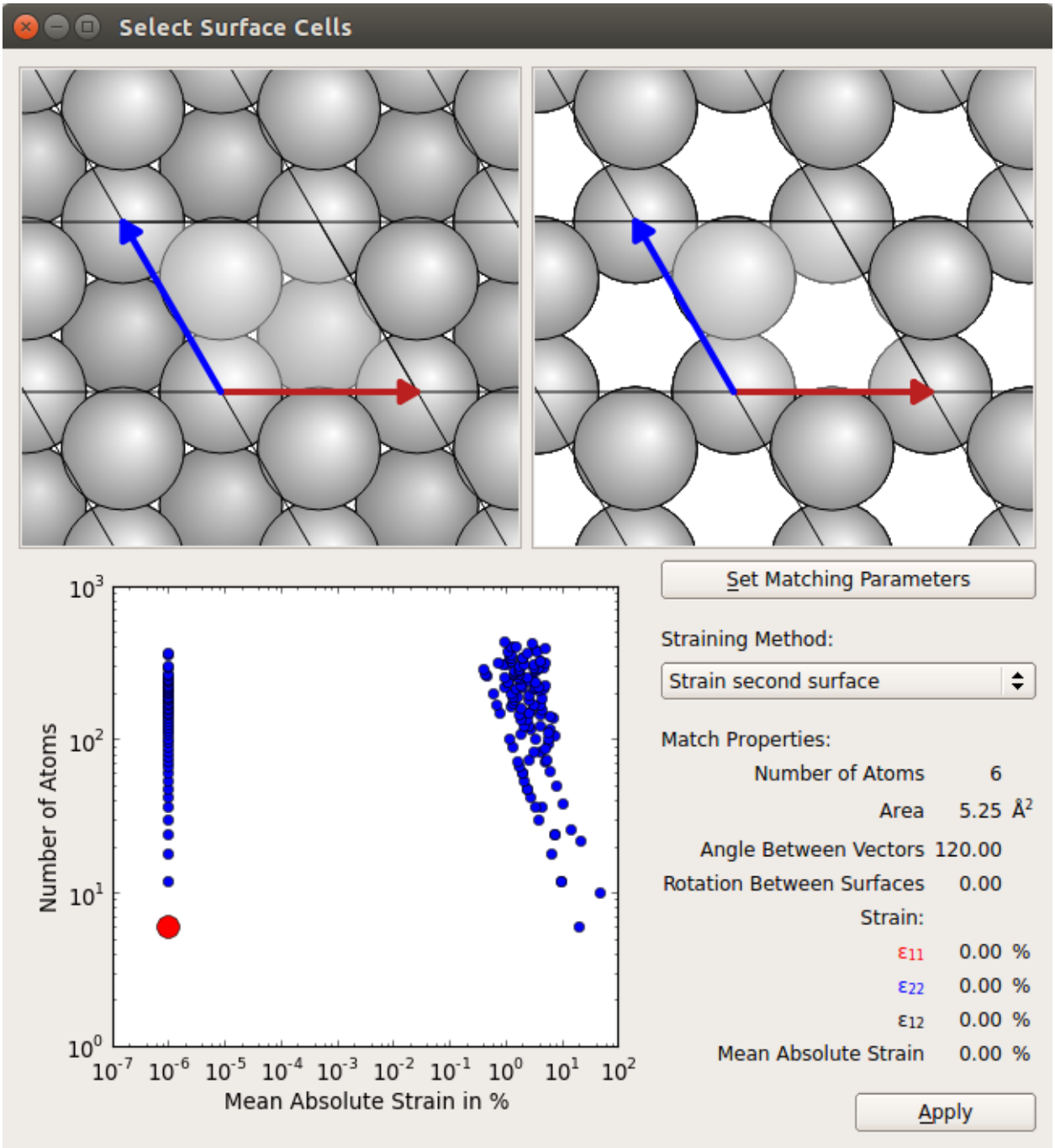


[1] E. Cisternas, M. Flores and P. Vargas, Phys. Rev. B 78, 125406 (2007)

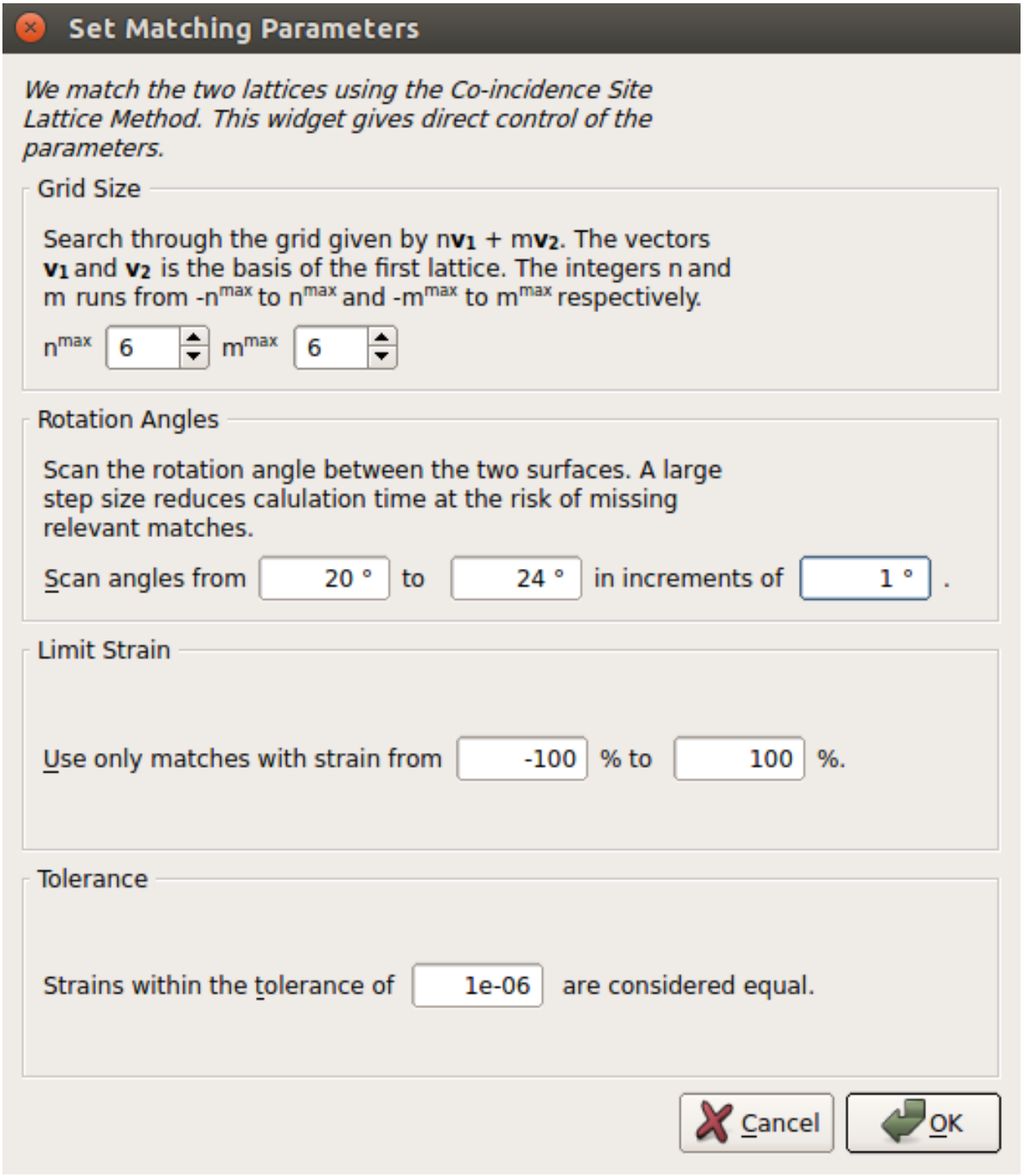
4.Click on *Builders* → *Interface tool*. Drag the 2-layer structure and drop it in the first slot, and the 1-layer structure in the second, as shown in the figure below.



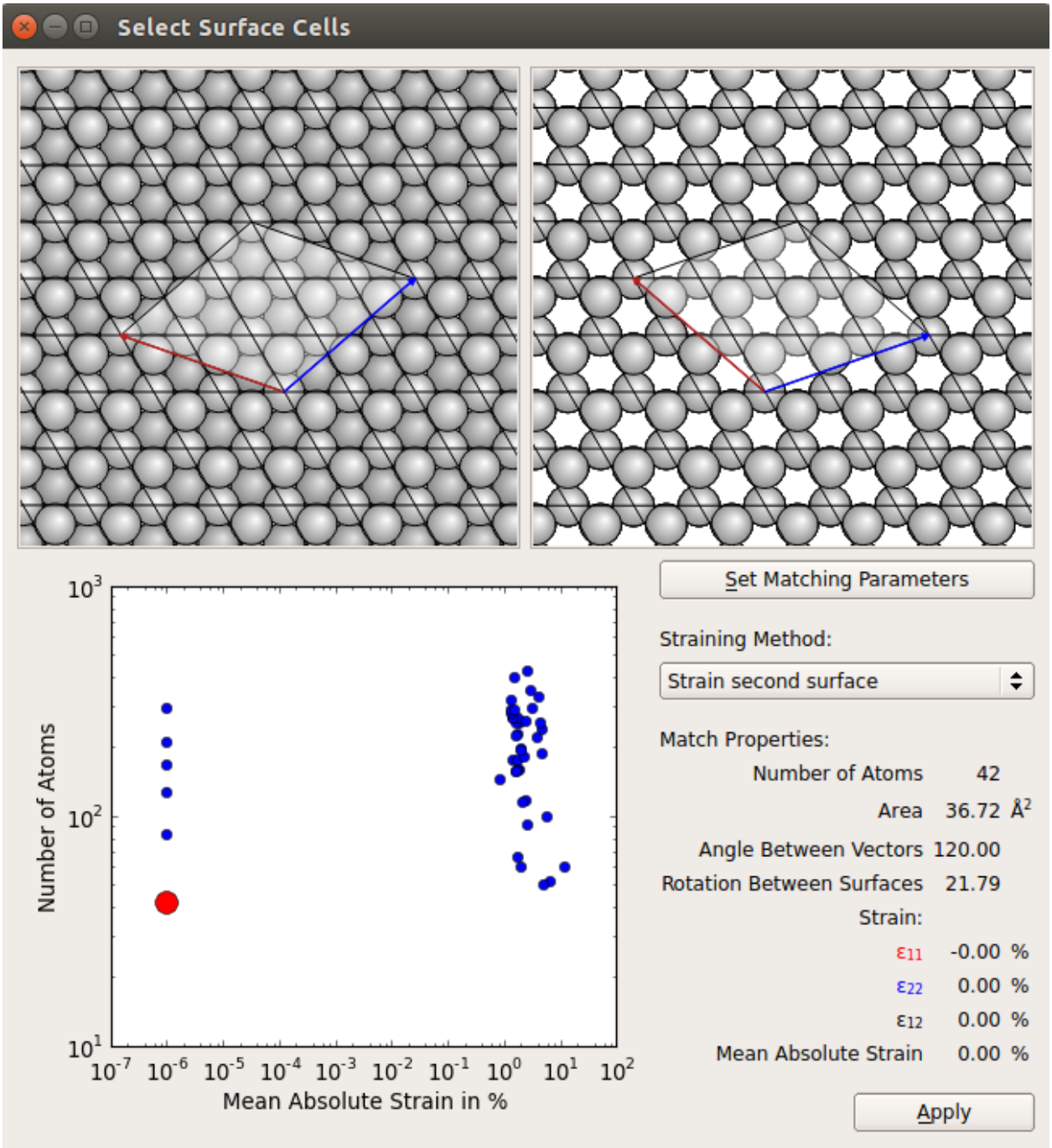
5.Click the button *Select Surface Cells*, and in the window that opens up click the button *Set Matching Parameters*.



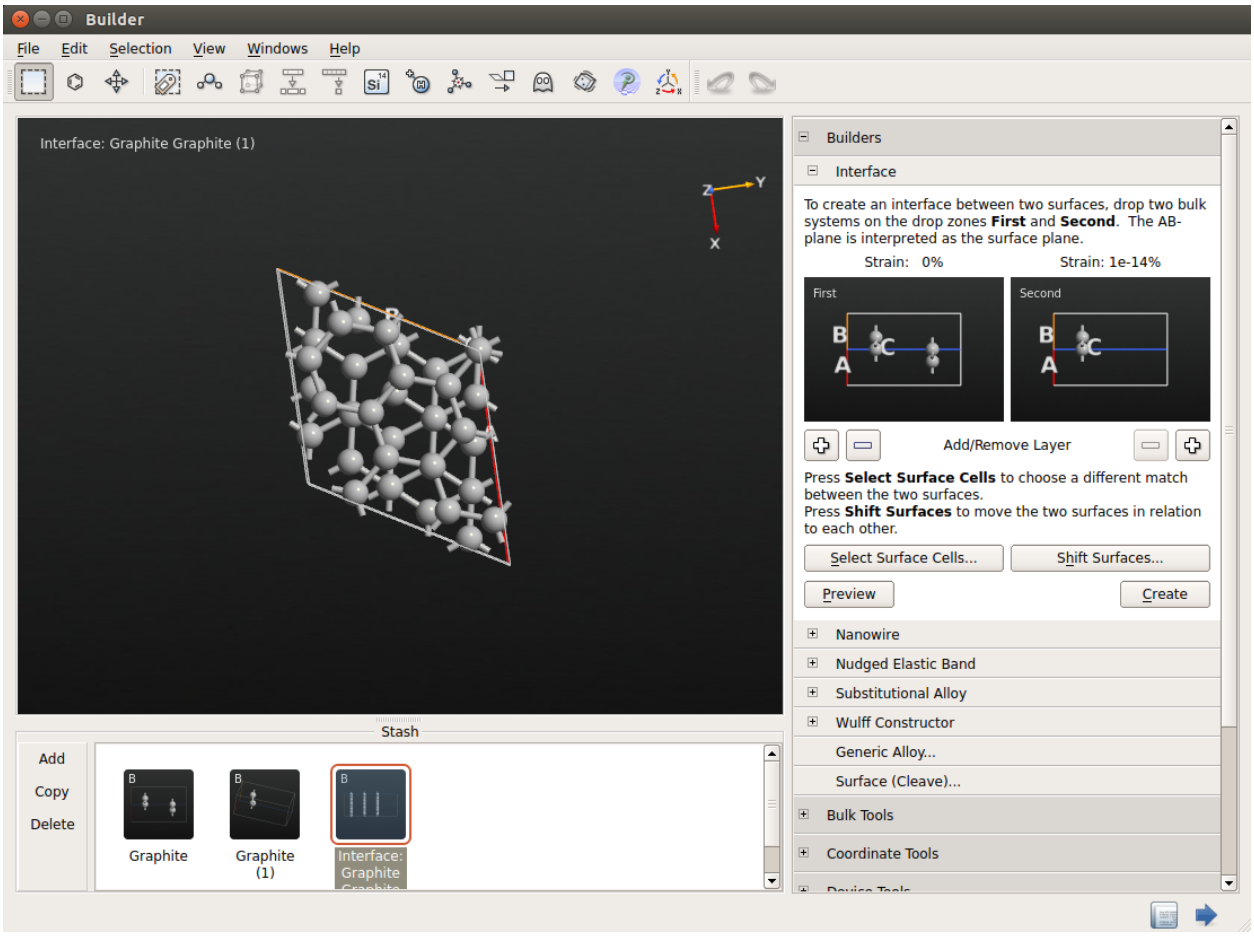
6. In the **Set Matching Parameters** widget that opens up, set the range of rotation angles to be searched from 20ř to 24ř, in increment of 1ř, and click *OK*.



7. The structure with the lowest number of atoms is normally selected by default in the 2D plot on the left-hand bottom side of the **Select Surface Cells** widget (the red spot in the figure below). In this case this is also the structure with the lowest strain (in fact, no strain at all, i.e. its a perfectly commensurate match). This is indeed the geometry you want, with the two lattices rotated by 21.79ř one with respect to each other. Click *Apply* to select this cell.



8. In the 3D window in the **Builder** you will now see a preview of the structure. You can rotate it, and if you would prefer to change some parameters you can still do it (for instance, you can add more layers of graphite to the substrate by clicking the plus button on the left under the first slot). Otherwise press *Create* to finalize the geometry.



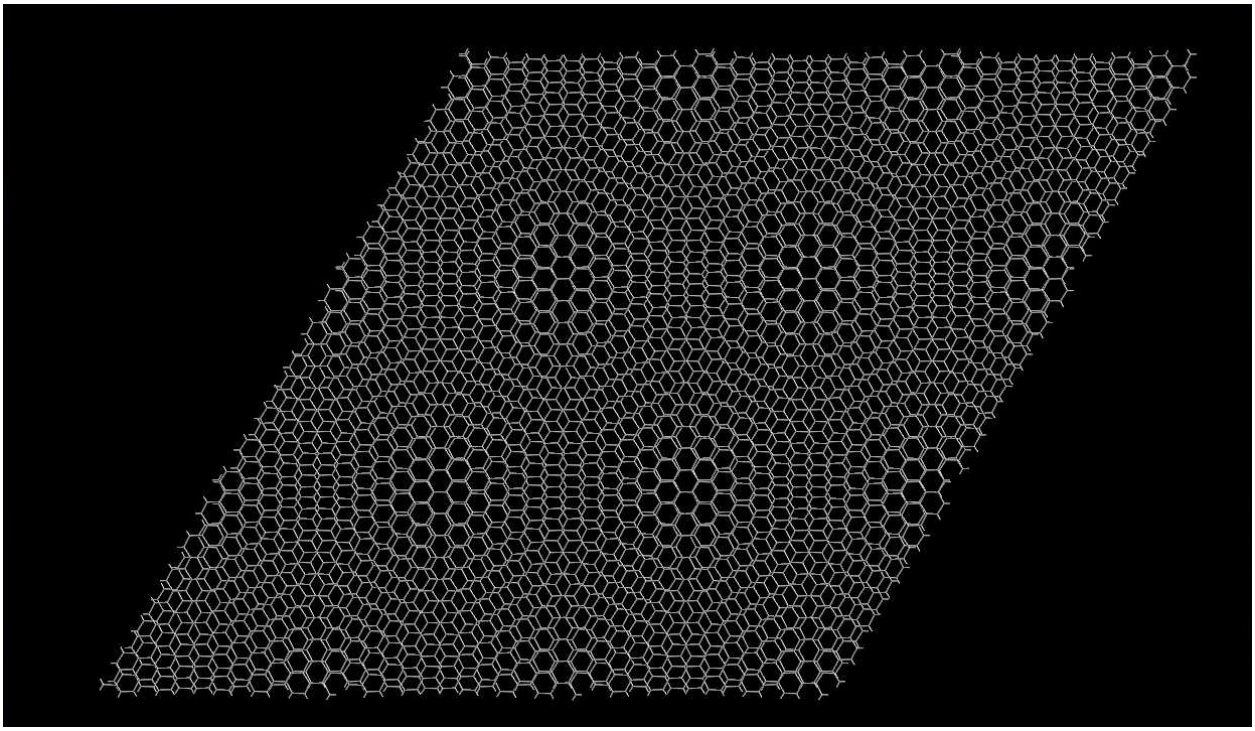
Additional rotated structures

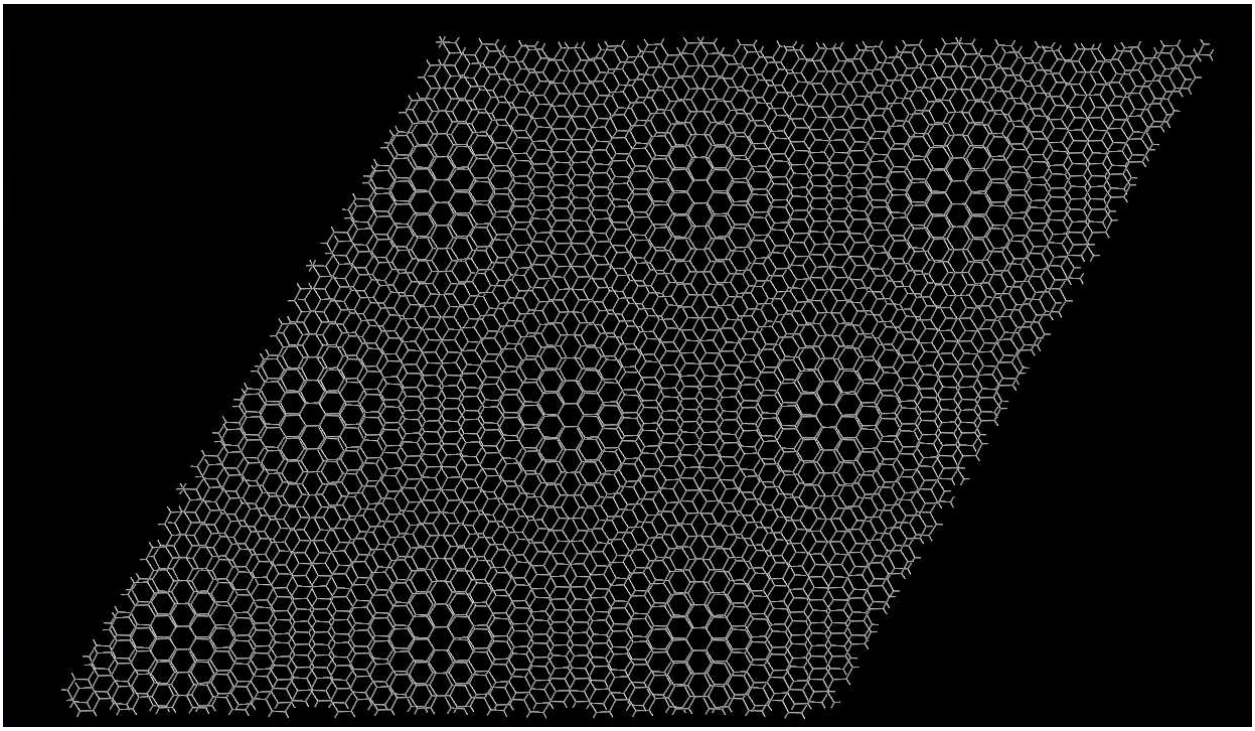
For more ideas on commensurate structures, see e.g. Ref.[2]. All the structures reported in the paper can easily be built using the procedure shown above. However, as noted in the article, the number of atoms becomes very large in many cases. The simple cases shown in Figure 7 in Ref.[Page 1070, 2] are quite manageable - note that in these structures there are only 2 layers rotated with respect to each other, not 3 as above. Both A-A and A-B stacking sequences can be used, so the easiest way is actually still to build the structure as above, and then remove either the middle or bottom layers, and adjust the layer separation.

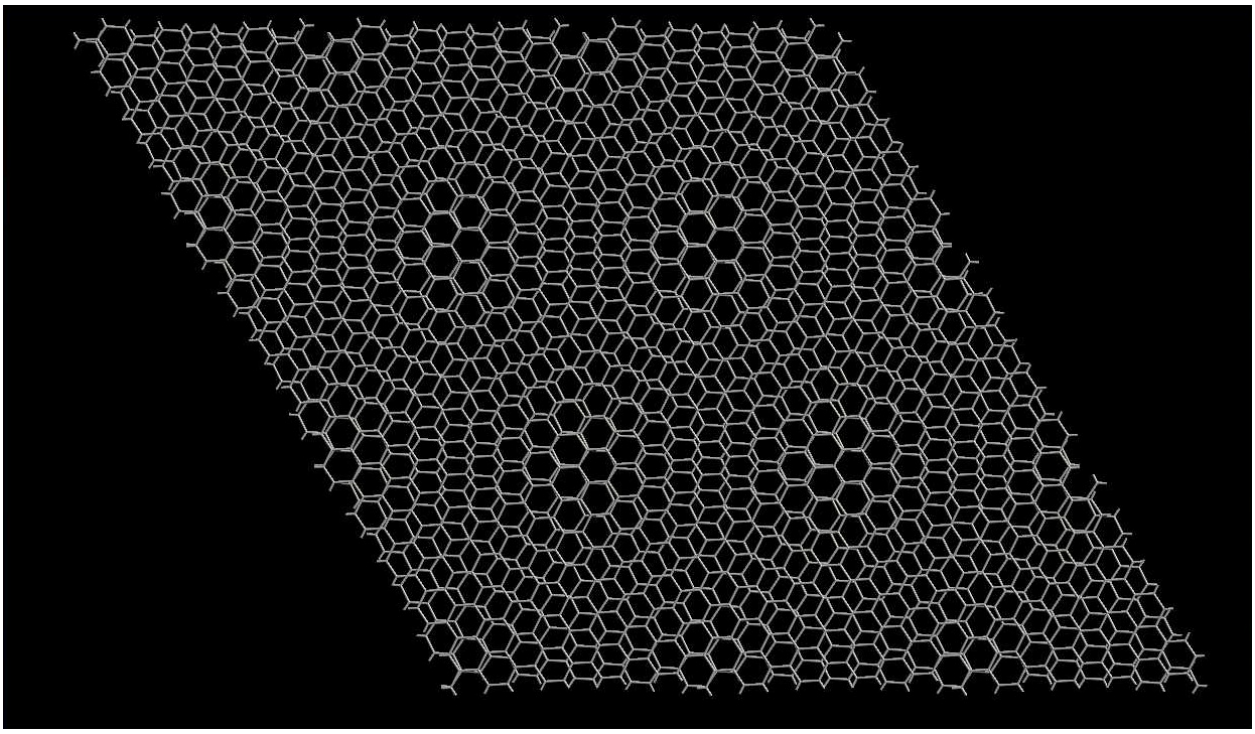
Note that in these systems it will be necessary to increase the parameters nmax and mmax too in the **Set Matching Parameters** widget. For instance, to find the 5.1ř supercell shown in Figure 7a in Ref.[2], you need to set them to 14. Notice that in this case the default suggestion in the **Select Surface Cells** widget may not be the structure with zero strain, if there are strained structures with a smaller number of atoms. So make sure to always select a structure with no strain (there may be a few, naturally take the one with the smallest number of atoms), by clicking the blue dots in the plot (the active choice is indicated by the red dot as shown above).

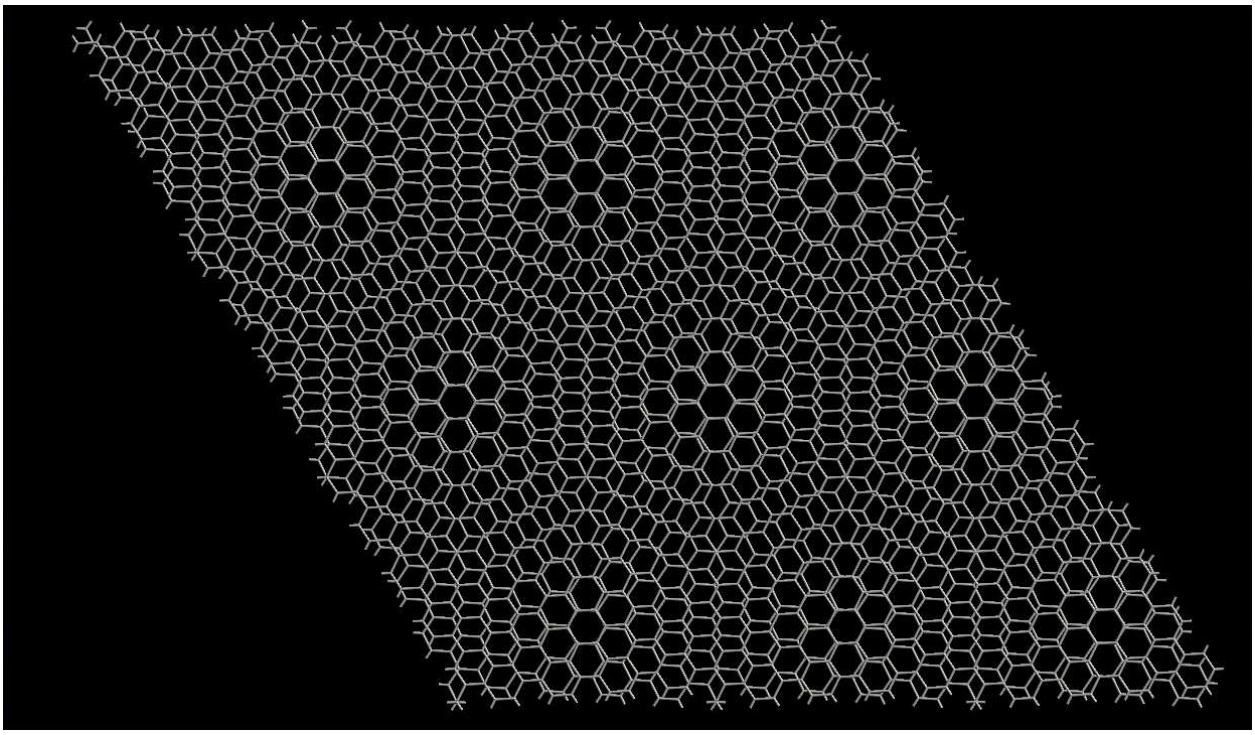
[2] J. M. Campanera, G. Savini, I. Suarez-Martinez, and M. I. Heggie, Phys. Rev. B 75, 235449 (2007)

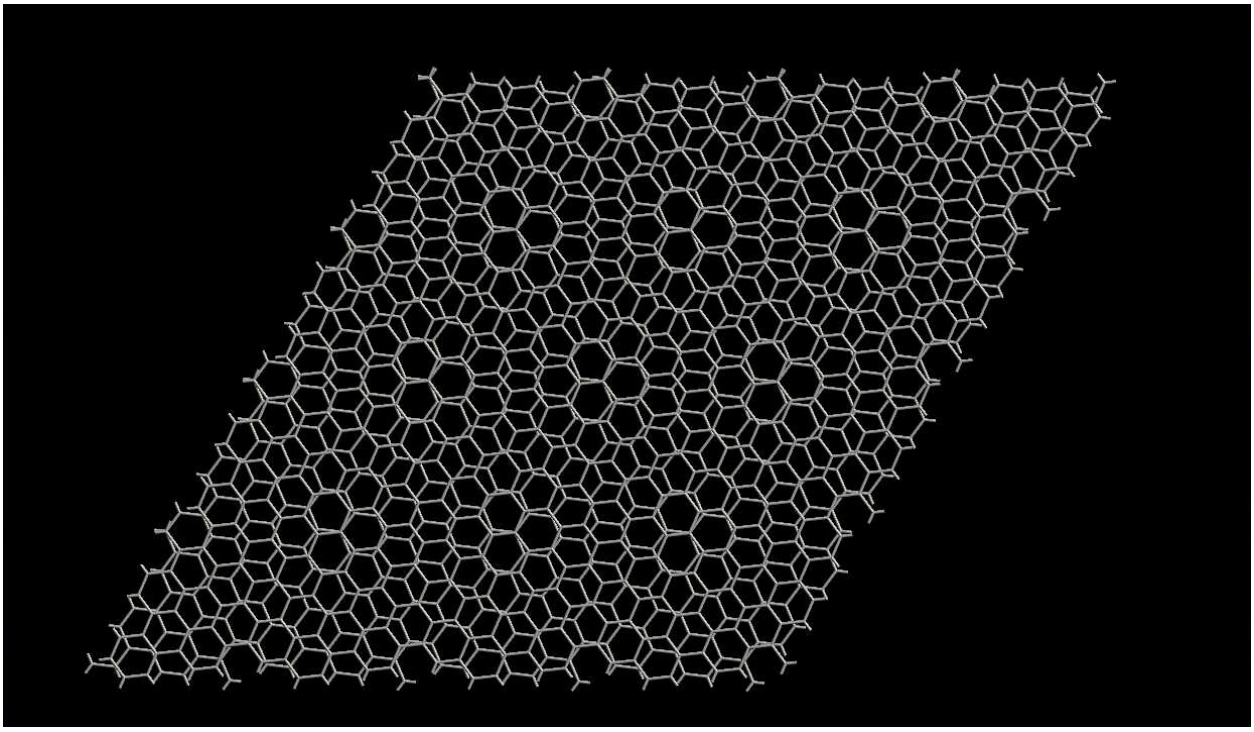
Here below you can find a few examples of the rotated bilayer graphene structures that can be created using QuantumATK and the procedure explained above.

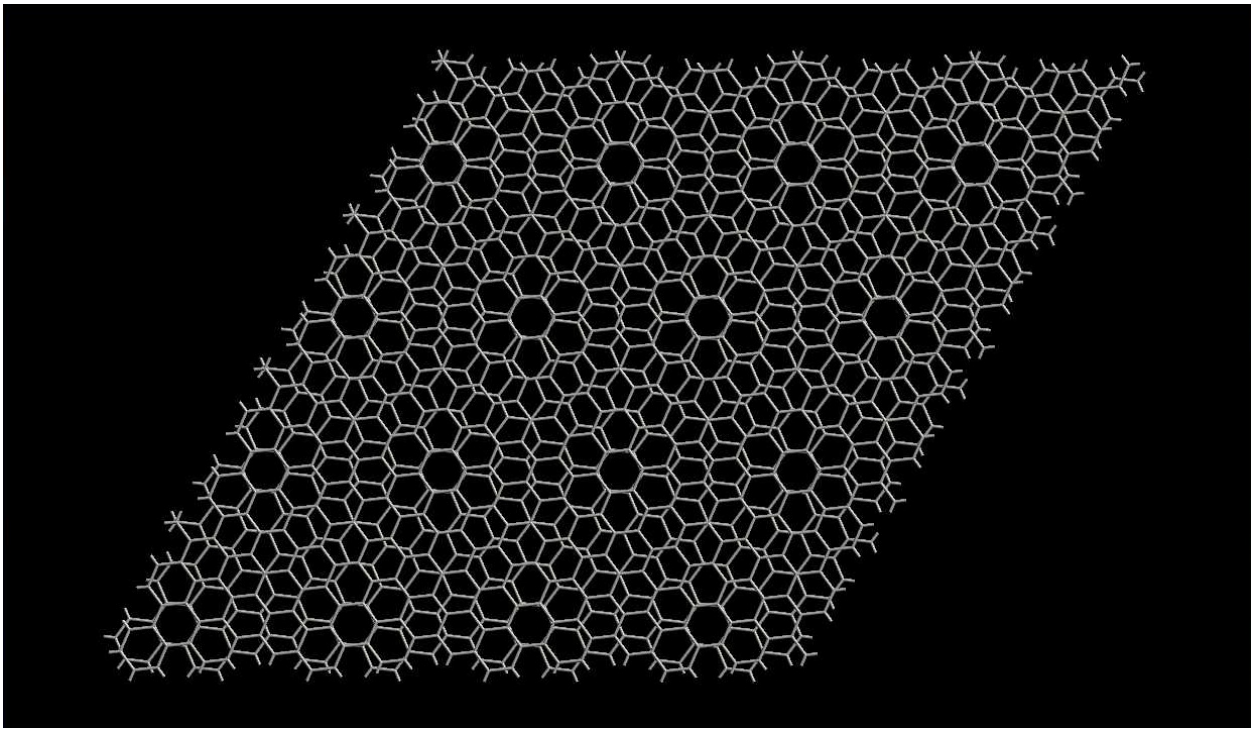


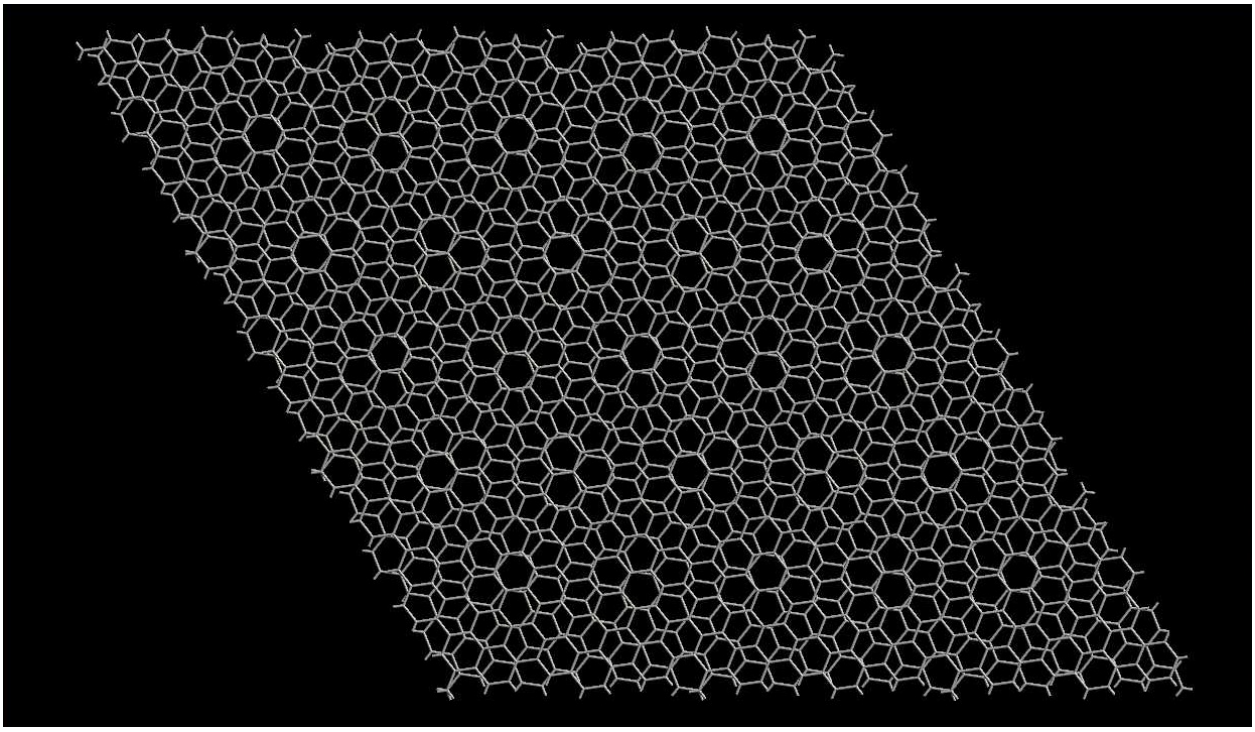


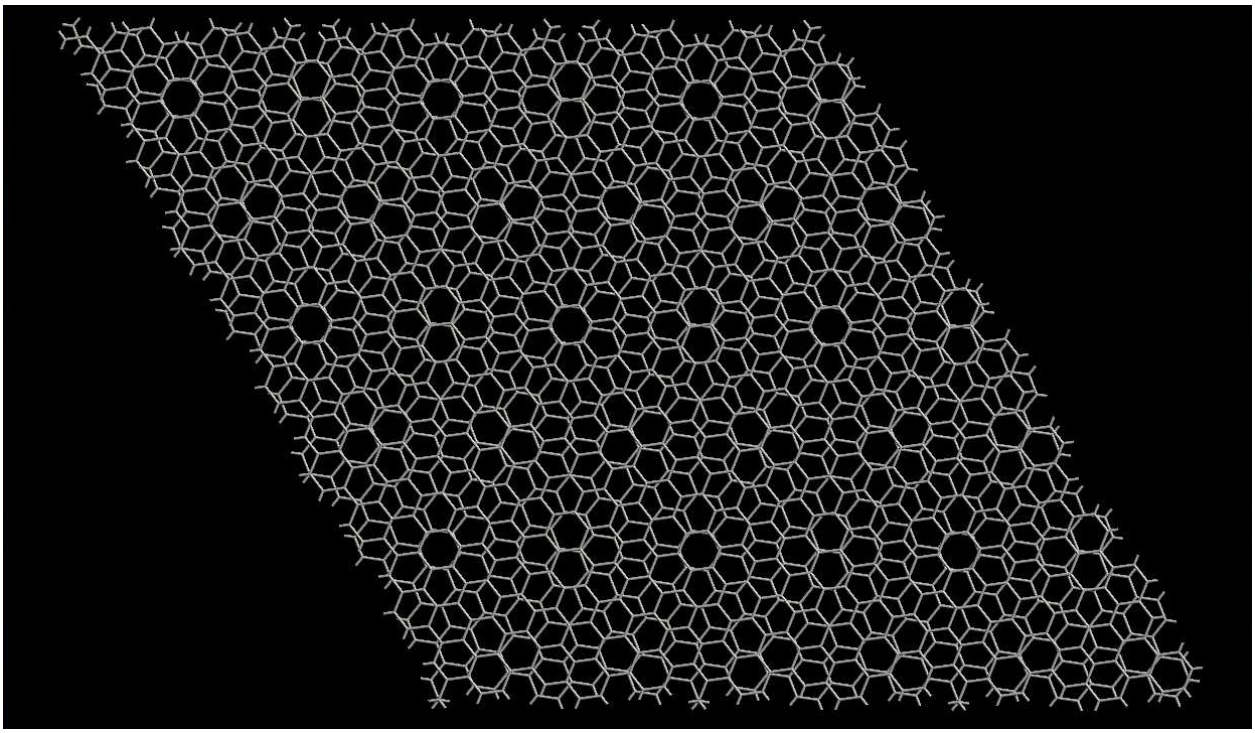


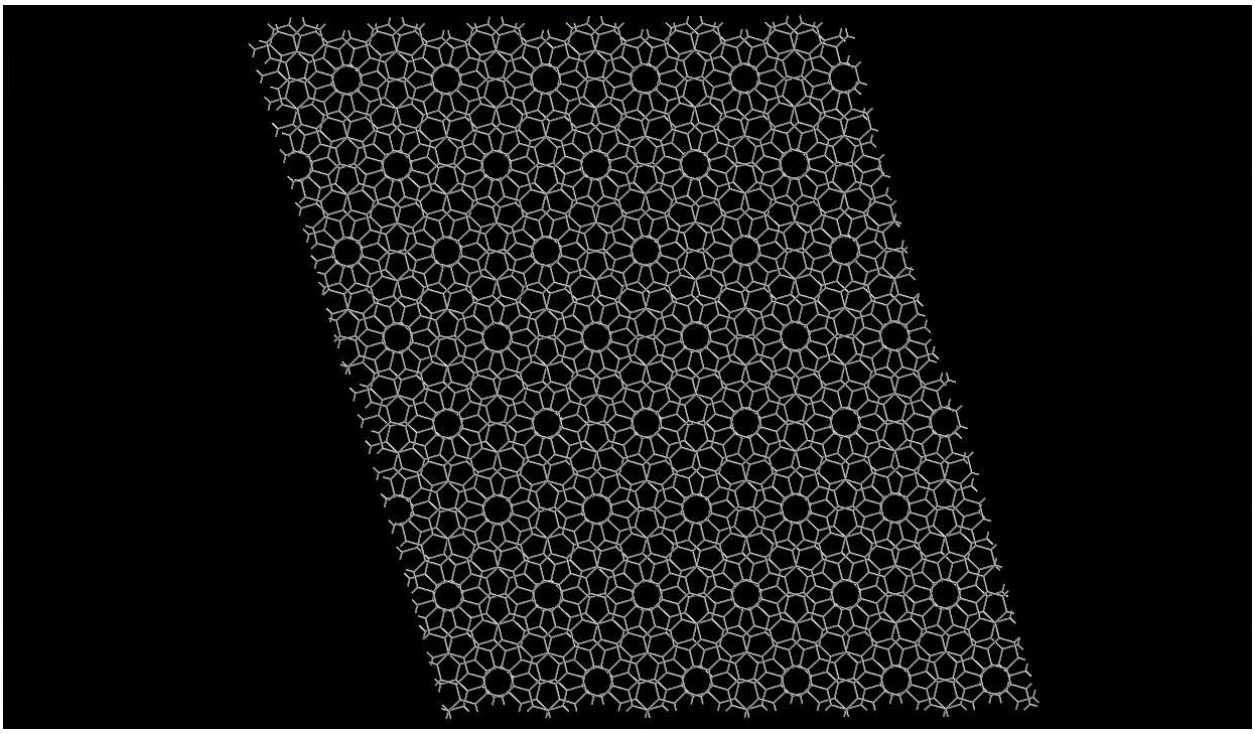


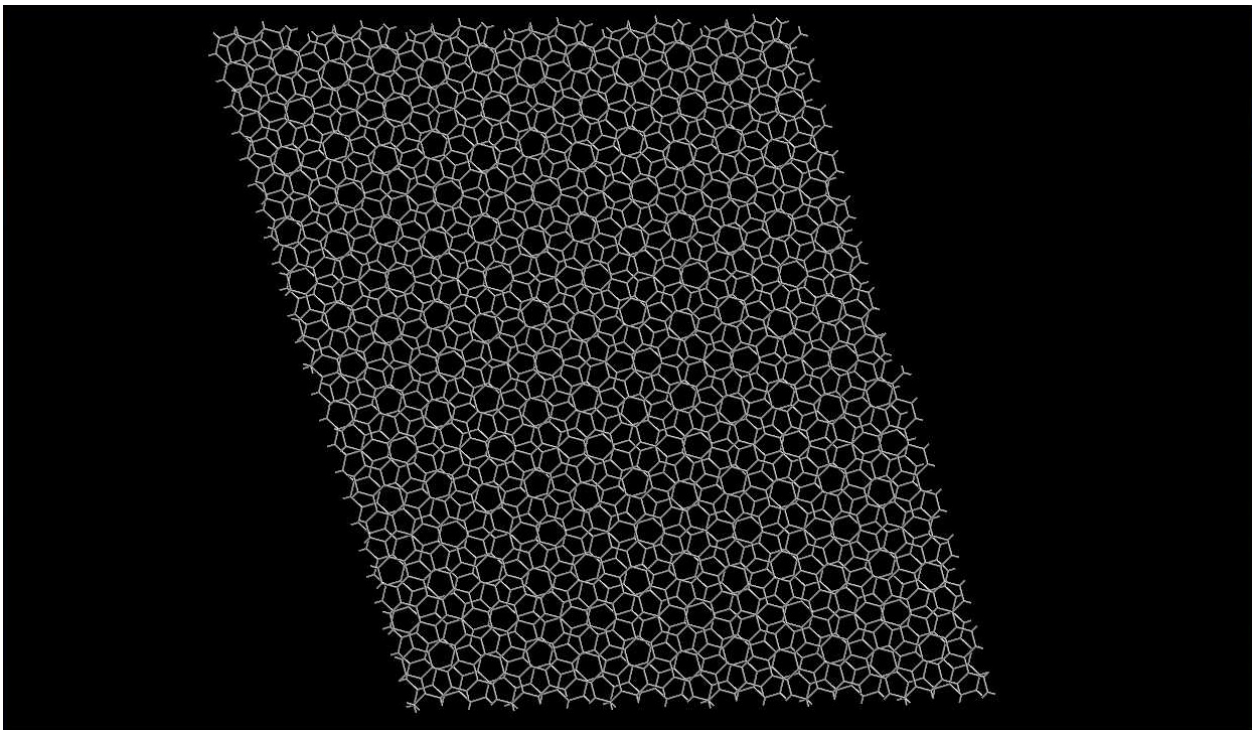












References

3.6.4 Spin-dependent Bloch states in graphene nanoribbons

Version: 2017

Graphene nanoribbons (GNRs) are very interesting systems for novel electronics applications. Depending on the shape of the ribbon edge, the ribbons can have metallic or semiconducting characteristics. As we will see in this tutorial, spin also plays an important role for the transport properties of GNRs.

You will use the capabilities of **QuantumATK** and **ATK** to study the spin-dependent bandstructure of a zigzag GNR. These ribbons are metallic if the calculation is performed without accounting for spin, but when spin is included, a band gap opens up. By plotting conduction and valence band Bloch states for various k-points, You will see how the two spin-components are localized on opposite sides of the ribbon.

Note: This tutorial will involve calculations that take at most 5-10 minutes on a usual laptop, so there is no real need to use a separate, more powerful computer or to run the jobs in parallel.

It will be assumed that you are familiar with the basic operations in the software corresponding to the level of experience gained from completing the basic QuantumATK Tutorials. No proficiency in programming will be necessary.

Band structure of a zigzag nanoribbon

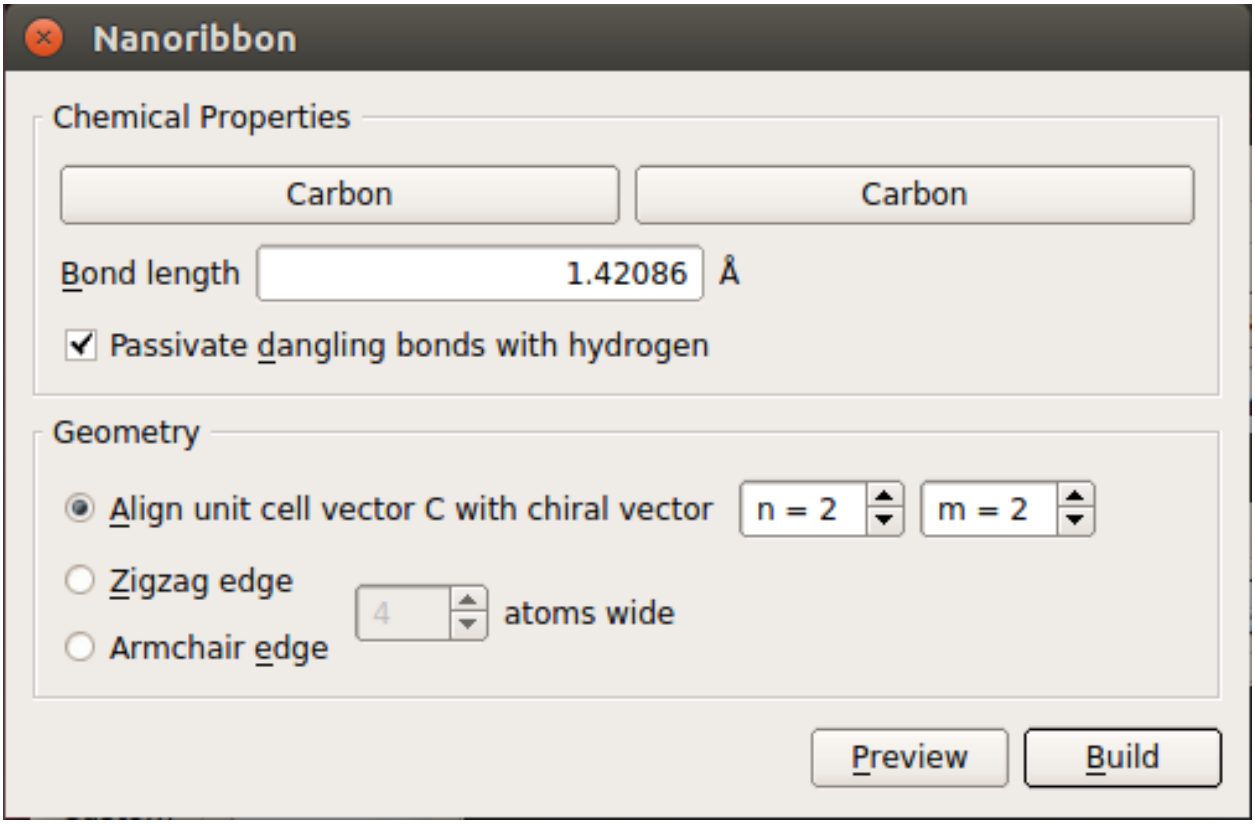
We will begin by constucting a graphene zigzag nanoribbon and calculate the bandstructure. Spin will be added in the next section.

Constructing the Graphene Nanoribbon

Start **QuantumATK**, create a new project and give it a name, then select it and click *Open*. Launch **Builder** by clicking the Builder icon on the tool bar.

In the Builder, click *Add* → *From Plugin* → *Nanoribbon*. This launches the Graphene Ribbon builder tool.

Leave all parameters at their default values and click *Build*.



The repetition is automatically set to the minimal value. This reduces calculation time and makes it easier to analyze the band structure.

There are periodic boundary conditions perpendicular to the ribbon, and the unit cell padding distance is automatically set to 10 Å, to reduce any residual electrostatic interactions with the periodic images of the ribbon.

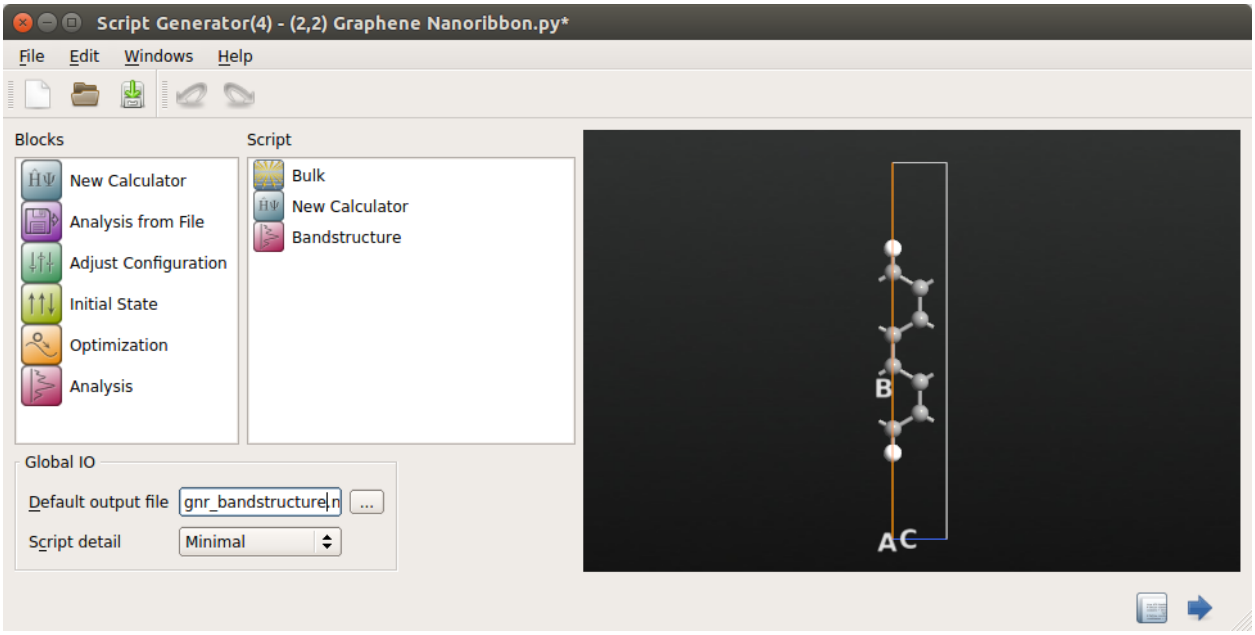
Calculating the graphene band structure

Now send the structure to the **Script Generator** by clicking the Send To icon in the lower right-hand corner of the window, and select **Script Generator** (the default choice, highlighted in bold) from the pop-up menu.

In the Script Generator,

- Add a New Calculator block .
- Add an *Analysis* → *Bandstructure* block .
- Change the default output filename to `gnr_bandstructure.hdf5`.

The Script Generator should now look like this:



Open the **New Calculator** by double-clicking it.

Most of the default parameters are good enough, but a few of them need adjustment.

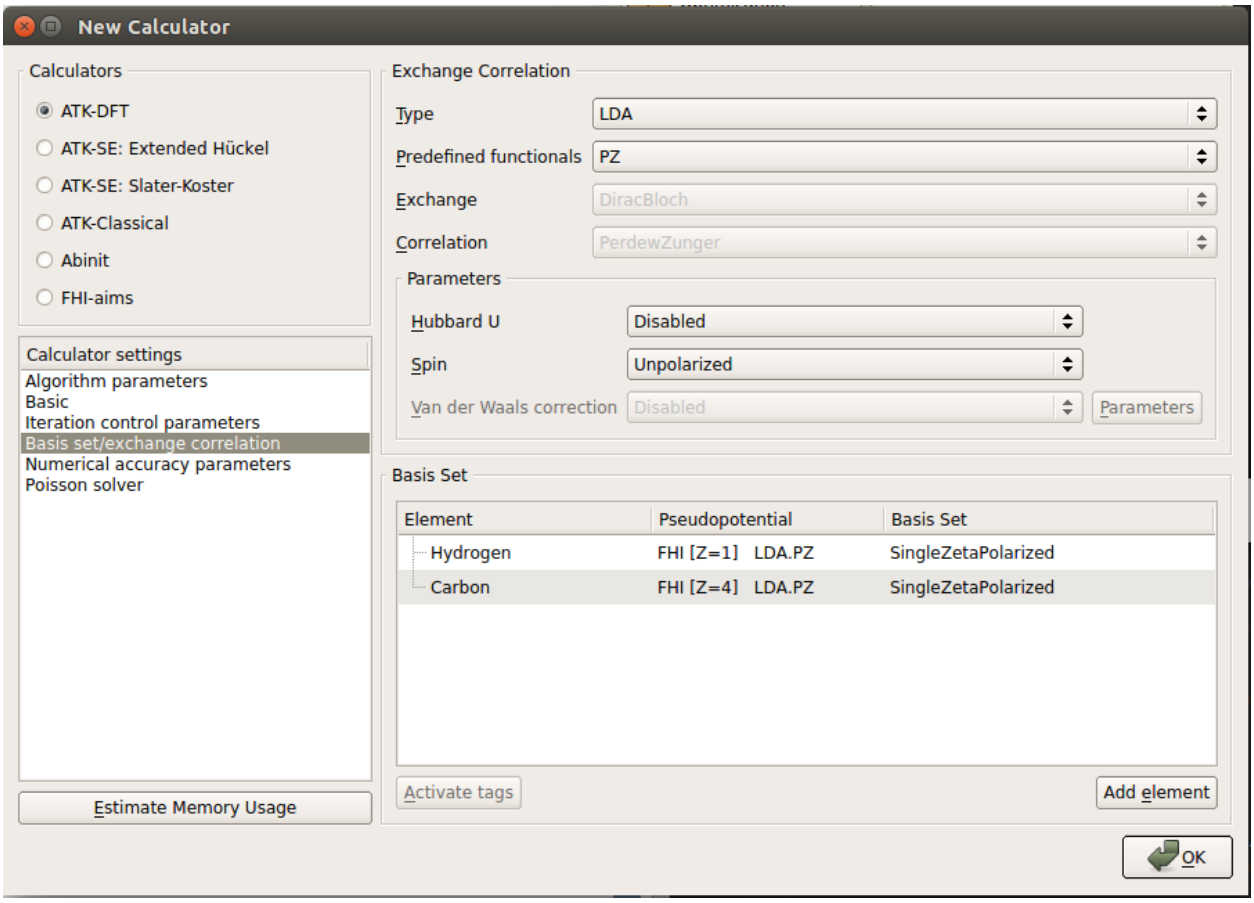
- You must carefully consider the **k-point sampling**. Since the unit cell along the ribbon is quite short, there must be a large number of k-points along the C-direction (along the ribbon).

Select 50 k-points in the C-direction.

This is a good trade-off between calculation time and accuracy. You only need a single point along the transverse A- and B-directions, since the system is not periodic in these directions.

- Next, under **Calculator settings**, choose **Basis set/exchange correlation** and reduce the basis set to SingleZetaPolarized for each element in the box on the lower right. This is to gain some speed; for this particular system, that level of accuracy is sufficient.

The calculator dialog should now have the following settings

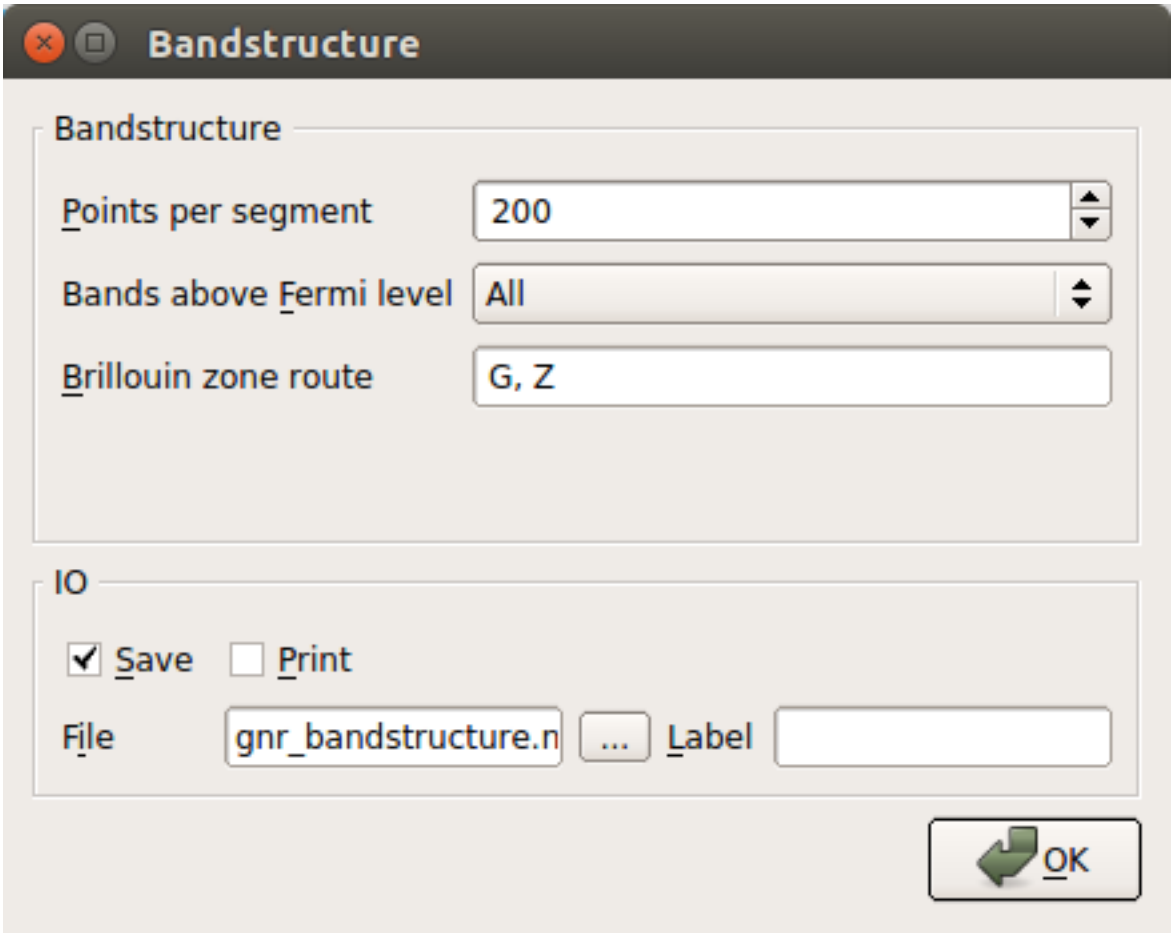


•Finally, click the **OK** button.

To specify the band structure calculation, double-click the **Bandstructure** block.

•In the appearing dialog, change **Points pr. segment** to 200. This implies that each band will be calculated with a resolution of 200 points. Also notice that the default suggested route in the Brillouin zone (from G (0,0,0) to Z (0,0,1/2)) is the appropriate one.

Note: The label G is used in **ATK** (and **QuantumATK**) to designate the Γ point, when setting up the script. In the band structure plot, the proper Greek label will be shown (see below).



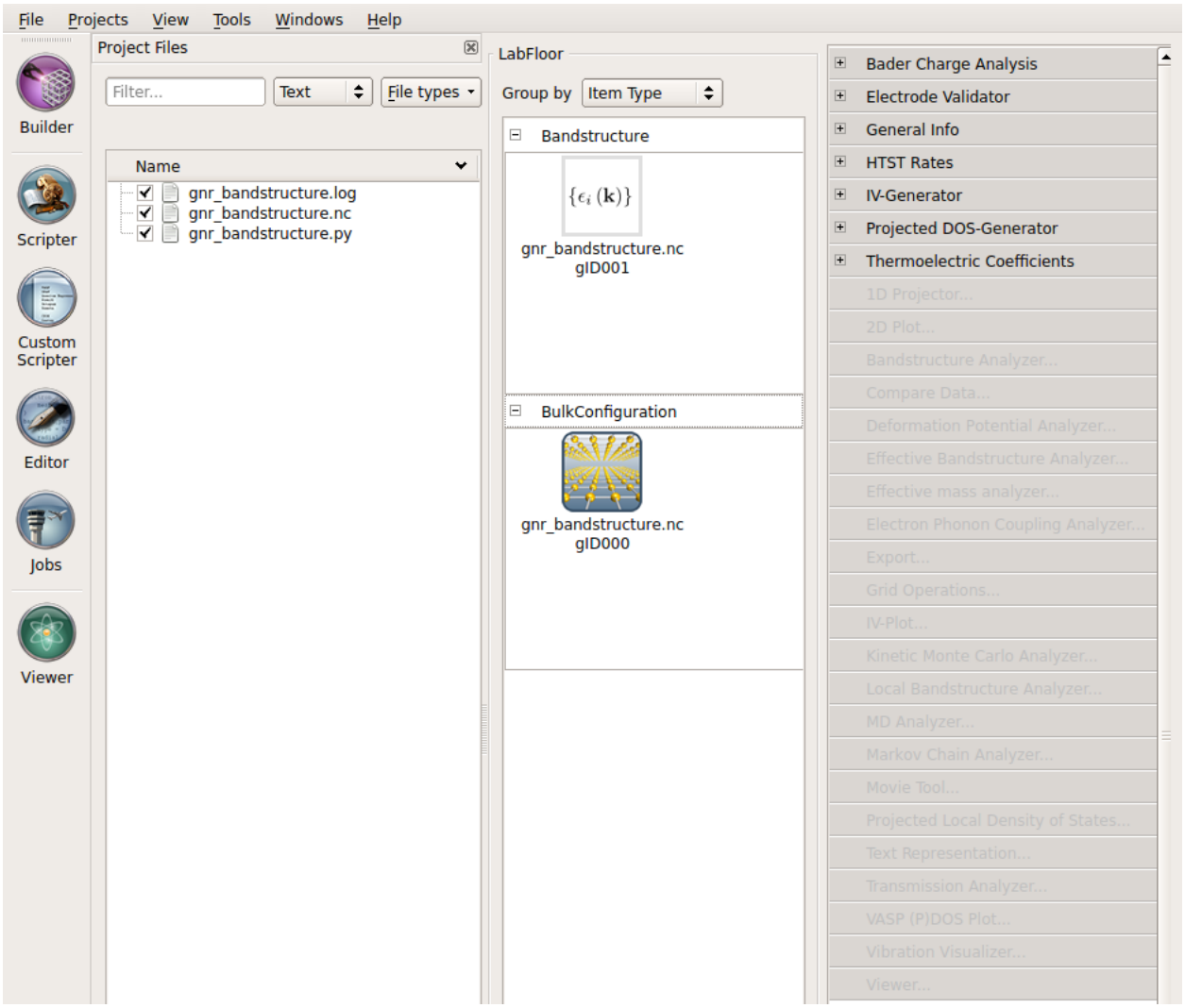
•Finally, click the **OK** button.

Now run the calculation by clicking the *Send To* icon and select **Job Manager** from the pop-up menu. Save the Python script under an appropriate name, such as `gnr_bandstructure.py` and click **OK**.

In the Job Manager, click the Start icon to run the job; the calculation takes only seconds to complete.

Analyzing the results

Return to the main QuantumATK window. The file `gnr_bandstructure.hdf5` should now be visible under Project Files in the left column. In the middle column, called the LabFloor, select *Group by Item Type*.



On the LabFloor, you can see that the file contains two objects: the calculated Bandstructure and the BulkConfiguration itself. The latter object contains not only the geometry, but the full state of the calculation. We will use this in the next chapter to calculate the Bloch functions without having to rerun the self-consistent calculation. For this, you will need to know the Id of the BulkConfiguration, which is shown after the file name; it is gID000 in this case.

Now, to plot the band structure, select the **Bandstructure** object (gID001), and click the **Bandstructure Analyzer** button in the plugin panel on the right.

The resulting plot is shown in the figure below. Note how the conduction and valence bands stick together for high values of k_z. This is a characteristic feature of zigzag ribbons (in our model) and clearly makes our ribbon metallic.

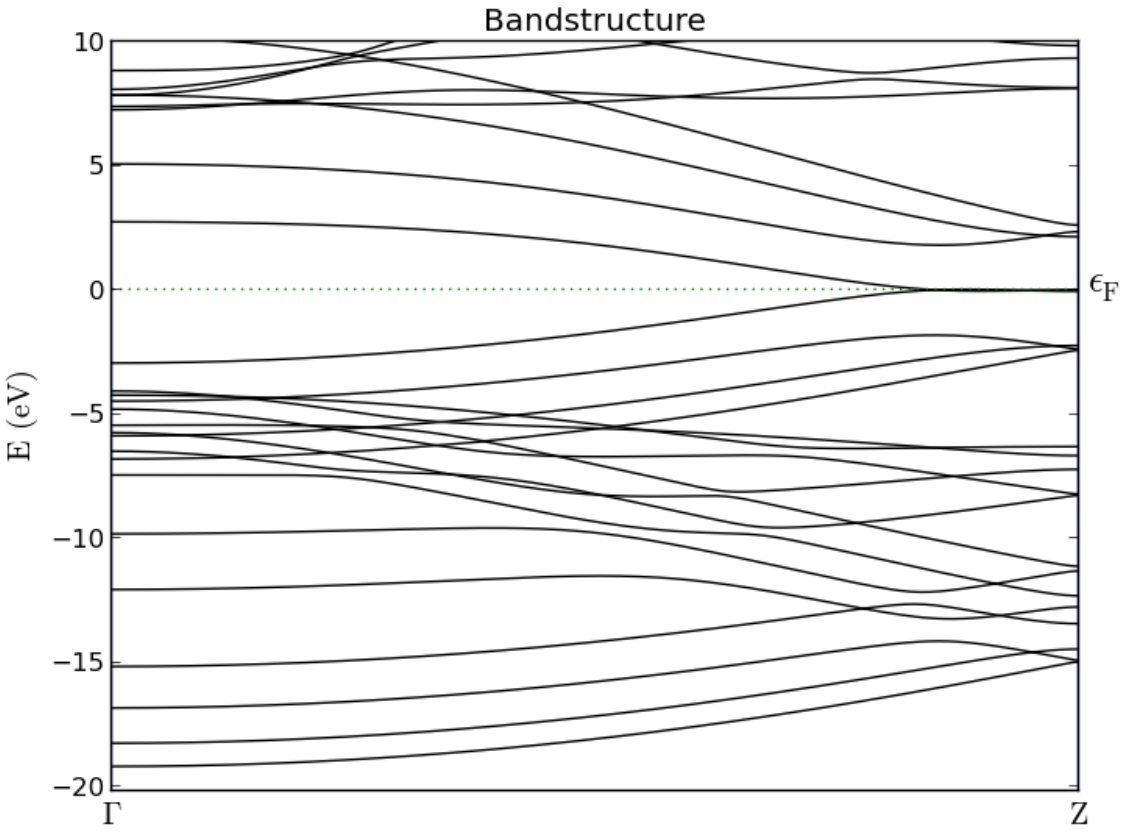


Fig. 3.89: Band structure of a zigzag graphene nanoribbon. The band gap at the Γ- point is roughly 5 eV, but closes entirely at the Z-point.

Bloch states

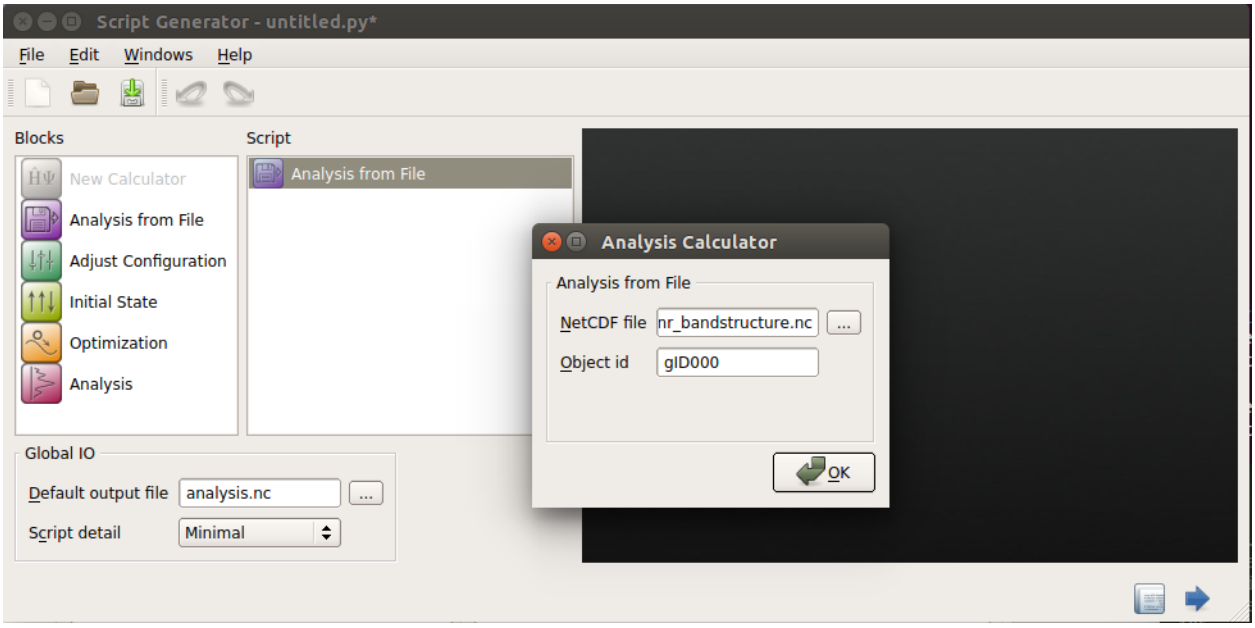
A very useful feature in **ATK** is its ability to calculate and plot Bloch states, which can be used to investigate the symmetry of certain bands and how this may relate to the transport properties. You will now use this capability to see how Bloch states close to the Γ point are delocalized across the ribbon, while states with higher k_z become increasingly localized towards the edges.

The ribbon consists of 8 carbon and 2 hydrogen atoms in each unit cell. A carbon atom contributes four valence electrons ($2s^2 2p^2$) and hydrogen only one (1s). Thus there are 34 electrons in the system. Each band is doubly degenerate (due to spin) and hence there will be 17 valence bands. Counting from 0, the band index of the highest valence band is 16 and the index of the lowest conduction band will be 17.

Setting up the calculation

You will now set up a script that computes the Bloch functions of these two bands, in three different k-points. The self-consistent state of the ribbon was saved in the file `gnr_bandstructure.hdf5`, and you can just restore it for this analysis, instead of rerunning the self-consistent loop. This is a very convenient way to separate the actual self-consistent calculation, that you perhaps prefer to run on a cluster (at least for a bigger system), and the subsequent analysis that often requires less resources and can be run directly via the Job Manager in **QuantumATK**.

- From the main QuantumATK window, launch **Script Generator** by clicking the Script Generator icon .
- The double-click the **Analysis from File** icon (it is the only available choice when we start the Script Generator in this way and without dropping a configuration on it).
- Double-click the icon and specify the file name `gnr_bandstructure.hdf5` in the appearing dialog.
- You should also specify the object ID of the configuration in this file you want to use for the analysis. As we found out in the previous chapter, this is gID000 for our case (it usually is, if the HDF5 file only contains one configuration).



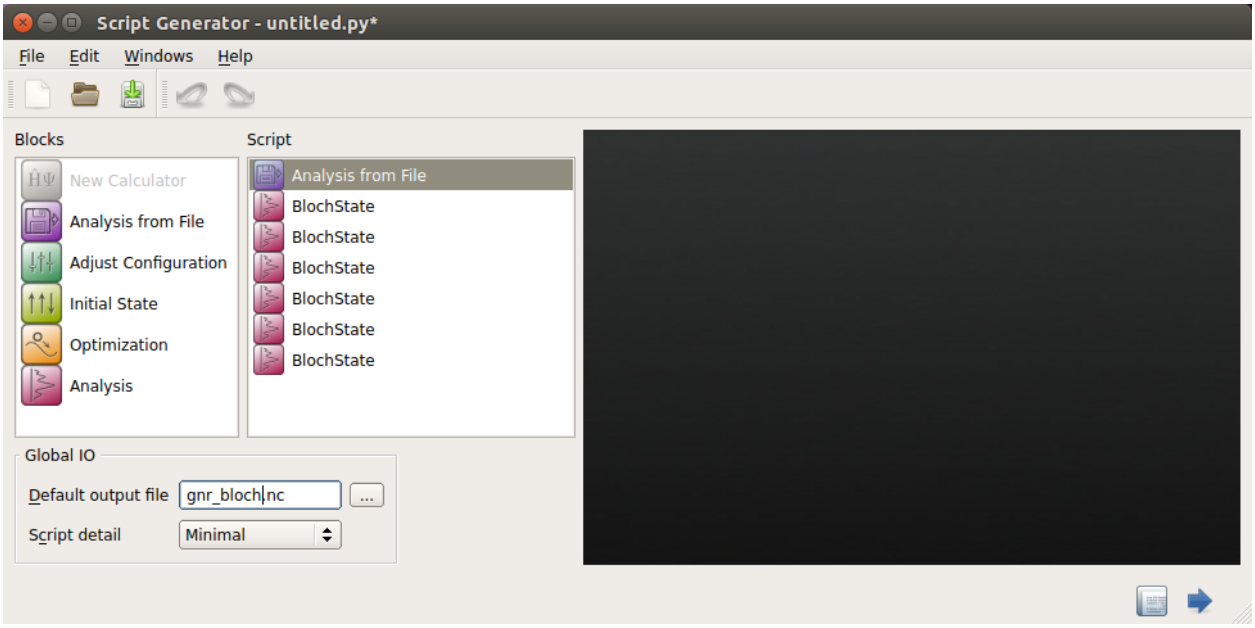
- Then click the **OK** button.

Now, we need to set up the calculation of the Bloch states.

•Add *Analysis* → *BlochState* 6 times. This should give 6 **BlochState** icons in the middle area of the Script Generator.

•Change the output file to `gnr_bloch.hdf5`.

The Script Generator window now loooks as follows:



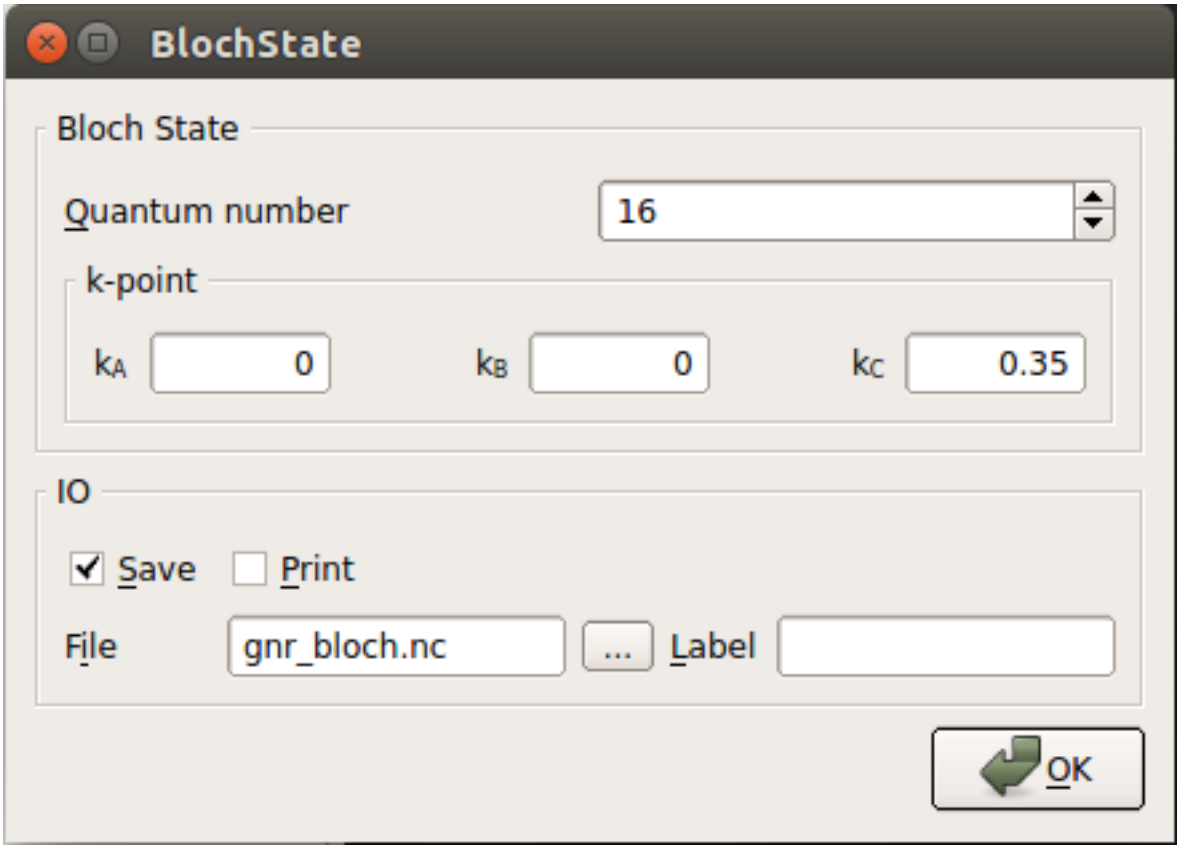
Double-click each of the first three Bloch state icons, enter 16 as the Quantum number (i.e. the band index), but give each one different k-points:

•(0, 0, 0)

•(0, 0, 0.35)

•(0, 0, 0.5)

In the second case, the BlochState window should look like the one shown below:



Repeat this procedure for the three remaining Bloch state icons, but this time enter 17 as the value for the Quantum number.

Finally, like before, send the job to the Job Manager and start the execution of the job.

Note: The calculation is very fast, but the generated HDF5 file will be quite large (about 120 Mb), which is why the data is saved in a separate file. This way, we avoid bloating the original file containing the self-consistent calculation (which would make it slower to read, if you desire to do some other analysis).

Analyzing the results

Once the job finishes, return to the main QuantumATK window. The results from `gnr_bloch.hdf5` should automatically be included in the LabFloor (see that the `gnr_bloch.hdf5` is checked under the under Project Files). Select Group by Item Type and expand the BlochState block.

Click on the last (`gnr_bloch.hdf5` gID005) of the six BlochState objects and select **Viewer** and then make sure the Plot type is set to Isosurface before pressing **OK**. Do this for each of the others, continuing with gID004, until all 6 are shown in the Viewer window. Now, select *Scenes* → *Tile* from the menu at the top of the Viewer window, in order to arrange the plots neatly. Note that the plots are now arranged with the newest window in the upper left corner, meaning that we have the valence band in the top row and the conduction band in the bottom row, with the gamma point to the left and the Z point on the right.

Tip: You may also add the subsequent Bloch state objects to the Viewer window by drag-and-drop. However, in this case you need to make sure that you do not drop the objects on an existing plot, as it will then be added to this instead

of being created as a separate plot.

To adjust the properties of each plot, click **Properties** from the menu on the right. This must be done separately for each window.

- Under **Isosurface** (the first tab), set the **Isovalue** to 0.05 and select the HSV color map.
- You can also go to the **Scene** tab and hide the axes by removing the tick from **Compass enabled**.

Below the visualization of the 6 different Bloch states are shown in the viewer. The quantum number is 16 for the top three and 17 for the bottom three, and the k-point coordinates are: (0,0,0), (0,0,0.35) and (0,0,0.5), starting from the left.

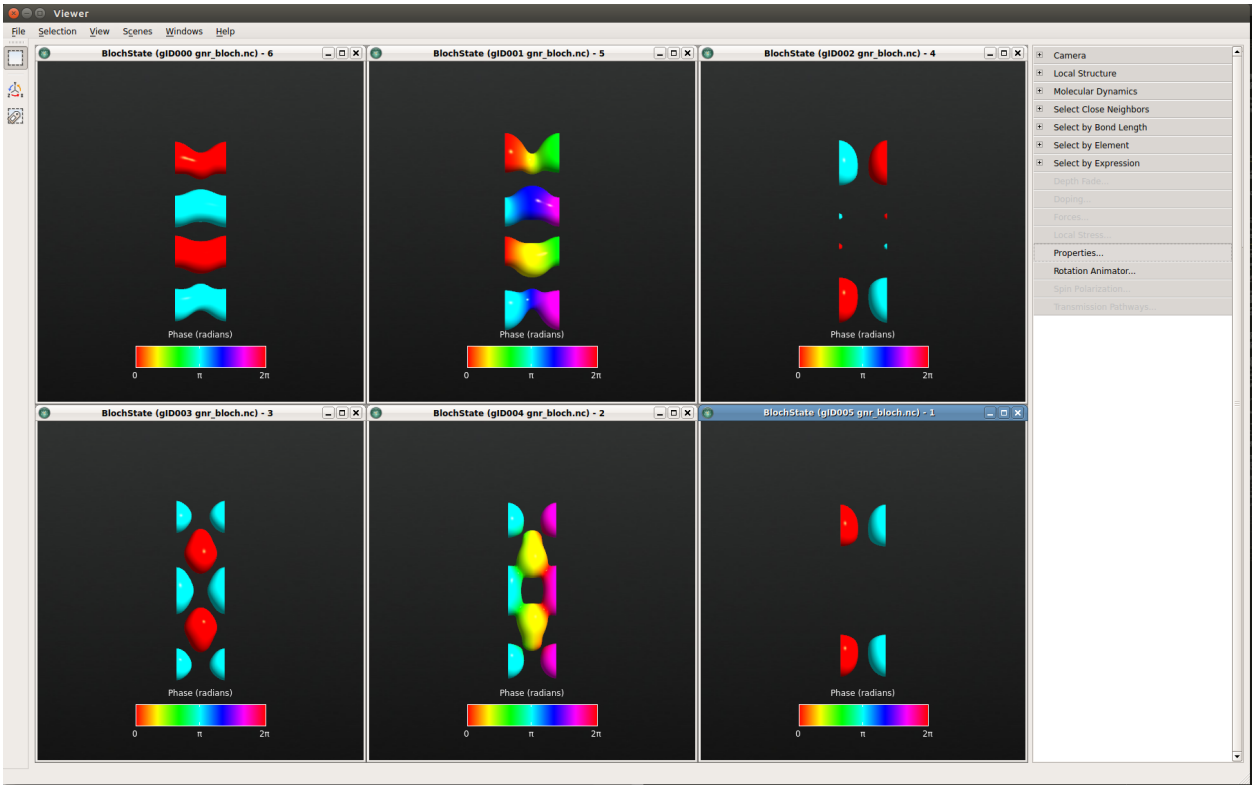


Fig. 3.90: From left to right: The Γ point, k=0.35, and the Z point. The top and bottom rows display the valence and conduction bands, respectively. (You may obtain a phase shift of π in your figures compared to this image; this has no physical relevance, of course.)

Looking at the respective Bloch functions, first note that the wave functions at G and Z are real (as expected), and second that there is a distinct difference between valence and conduction band Bloch functions. The occupied valence band states appear to be connected in the ribbon direction, while the (unoccupied) conduction band states are more oriented across the ribbon. At Z, the states become localized towards the edges of the ribbon.

In the next section, you will see how spin splits the degeneracy around Z, and how the two spin states become localized on opposite edges.

Tip: To superimpose the geometry of the ribbon on the plot, drag the BulkConfiguration object from the file `gnr_bandstructure.hdf5` onto a BlochState window in the open Viewer window.

Introducing spin

Setting up the calculation

You will now redo the entire calculation with spin-polarization included. The work-flow is very similar, and each step is therefore only outlined this time.

1. Set up the calculation like you did for the bandstructure calculation: Go to the builder and send the graphene nanoribbon to **Script Generator** (the builder stash should still contain the graphene nanoribbon from last time).

2. Add a **New Calculator** block and set the same parameters as before (SingleZetaPolarized basis set and (1,1,50) k-points). In addition, in the **New Calculator** dialog window, set Spin to Polarized and check that the exchange-correlation functional has automatically changed to LSDA.

3. The default in **ATK** is to start a spin-polarized calculation with a symmetric initial spin density. However, in this case, this will result in a local minimum corresponding to a symmetric (ferromagnetic) state. To get the real anti-ferromagnetic ground state of the graphene nanoribbon we have to set up the initial spin configuration with opposite polarizations on the two nanoribbon edges (the middle carbon atoms and the hydrogen can be left unpolarized).

   To set up the initial spin state of the system:

   - Double-click the **Initial State** icon to add it and then open it.
   - Set **Initial state type** to User spin, and then under **Spin**, set the default spin of both carbon and hydrogen to 0.
   - Then in the 3D view, select (by using Ctrl+left-mouse click) the two upper carbon atoms and verify that their numbers are 0 and 7 in the list. Set their spin values to 1. Repeat this procedure for the two lower carbon atoms (they should be number 1 and 2 in the list) setting their initial values to -1. The hydrogen atoms and the four middle carbon atoms can be left with spin values equal to 0. The Initial State dialog now look as shown below: (dont be confused by the fact that the carbon atoms are not ordered by Y coordinate).

Tip: If you have no idea about the spin state of the system, you may select Random spin. It sets a random spin on each atom, and dependent on the random values, the system may converge to different spin states of the system. The spin state with lowest total energy is the ground state.

---

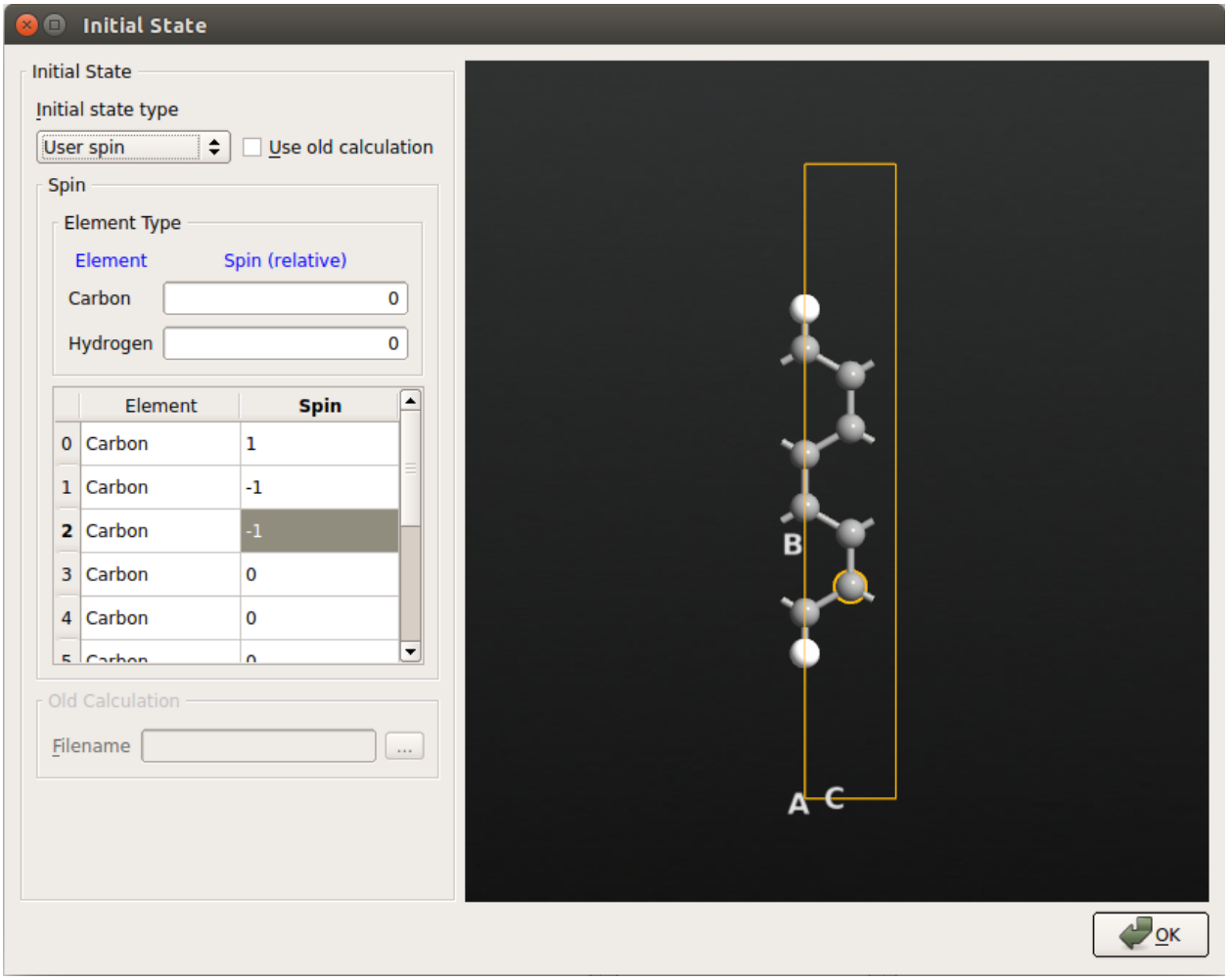
4. Add a **Bandstructure** calculation for the spin polarized calculation and again set **Points pr. segment** to 200.

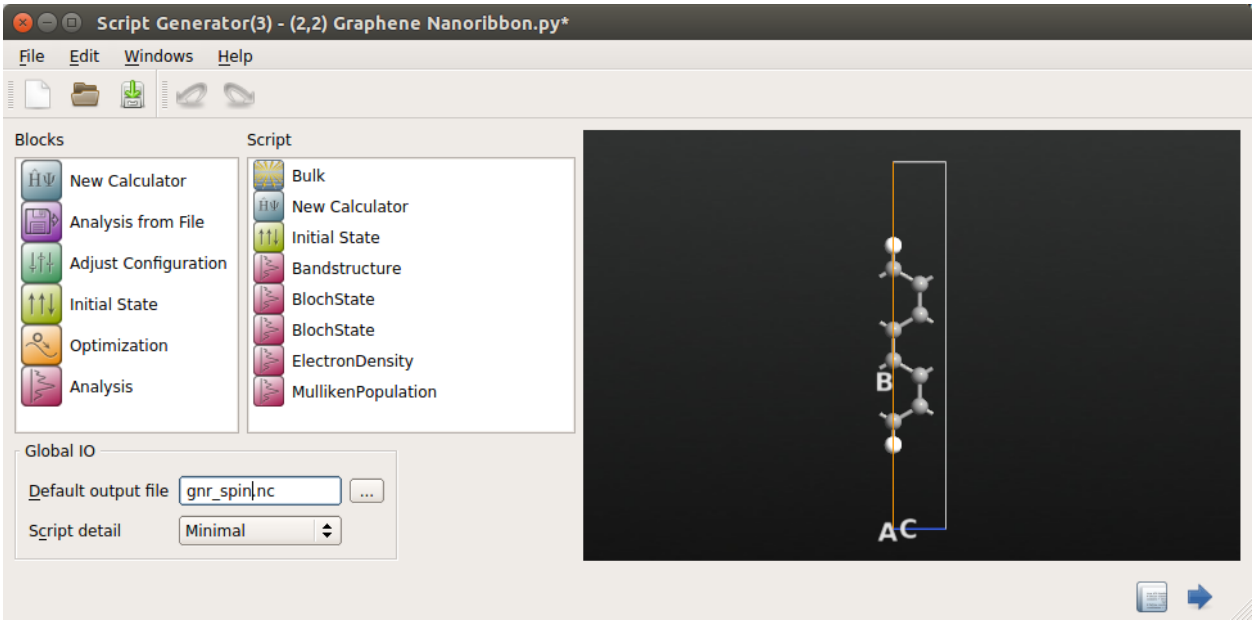
5. This time, you may just as well include the Bloch states from the beginning to get everything done in one script. For the following analysis, it is sufficient to only calculate the Bloch functions in the Z point. Add two Bloch state icons all defined at the Z point (0,0,1/2). Let one of these have Quantum number 16 and the other have the Quantum number 17.

6. We will also need the electron density later, so add also an *Analysis→ElectronDensity* object.

7. Another way to look at the spin density is through the Mulliken populations, so add also an *Analysis→MullikenPopulation* object.

8. Then set the name of the output file for the calculation in the Script Generator window to `gnr_spin.hdf5`. The Script Generator window should now look like the one shown below.


Initial State
Initial State
Initial state type
User spin
Use old calculation
Spin
Element Type
Element
Spin (relative)
Carbon
0
Hydrogen
0
Element
Spin
0 Carbon 1
1 Carbon -1
2 Carbon -1
3 Carbon 0
4 Carbon 0
Old Calculation
Filename
...
B
A
C
OK



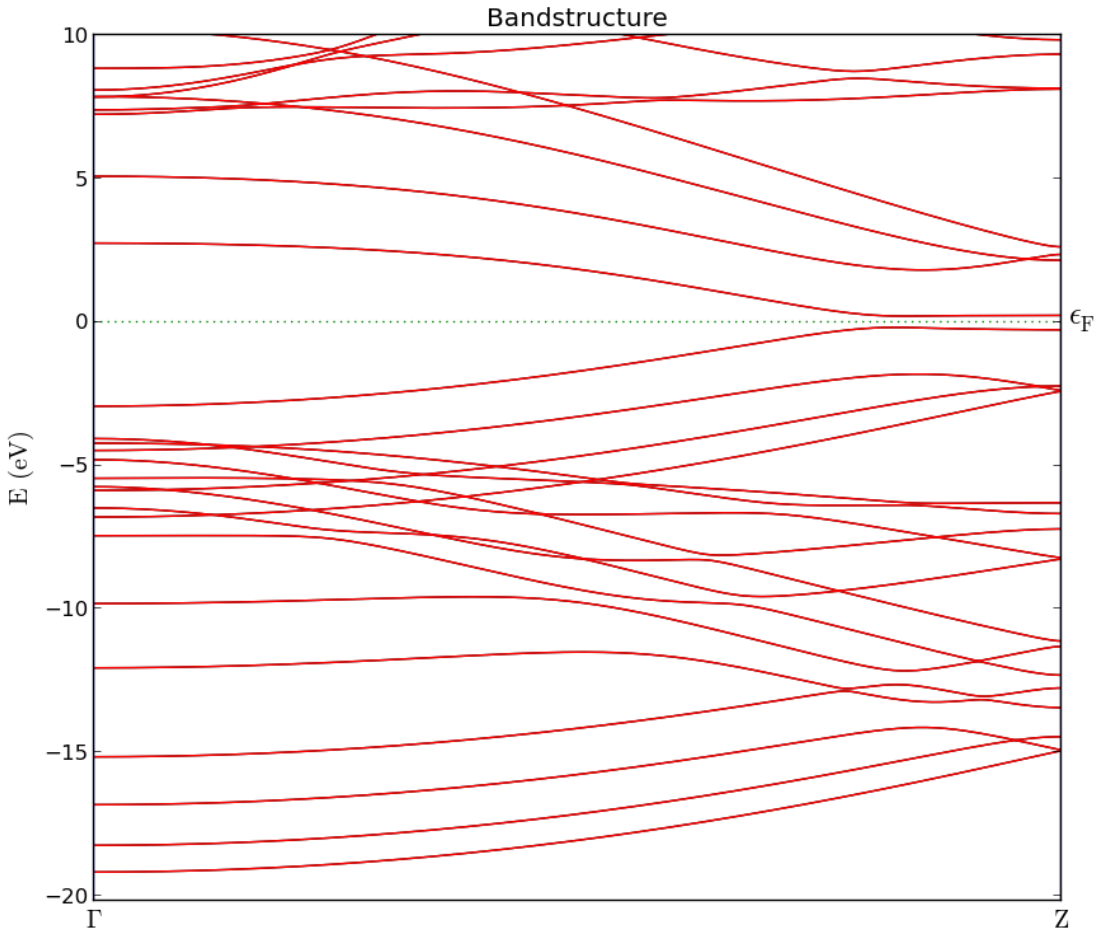
9.Send the script to the Job Manager and run the calculation.

Analyzing the results

By inspecting the column DD in the calculation log you will see, by summing the entries, that there is no total spin-polarization in this system (the total spin up and down will both be 17), which is very reasonable; carbon is not magnetic.

By plotting, you can also observe that there is no spin-dependence in the band structure; the up and down states are completely degenerate (otherwise you would see both blue and red curves in the band structure plot).

However, the inclusion of spin does have a profound effect on the bandstructure. The earlier degeneracy of the conduction and valence bands close to k_z =0.5 is broken, and a band gap has opened up.



Plot the corresponding Bloch states at the Z point and select to plot spin up and down for each of the two BlochState objects. If you adjust the view properties as in the previous chapter, you clearly see how the two spin valence states are localized on opposite edges of the ribbon. The conduction band states behave completely analogously, but are localized on opposite edges compared to the valence band states.

Electron density and Mulliken populations

Since the up and down Bloch states are localized on opposite edges, there must be a difference in the total spin up and down electron density.

The Mulliken populations is the easiest way to get a first view of this, and it is most conveniently viewed by selecting the MullikenPopulation object on the LabFloor and selecting **Text Representation** on the right. The resulting window is shown below. Under the column Total, the two spin-channels are shown on seprate lines, and we find that the two edge carbon atoms (numbers 0 and 1) have a surplus population of 0.25 of either spin-up or spin-down.

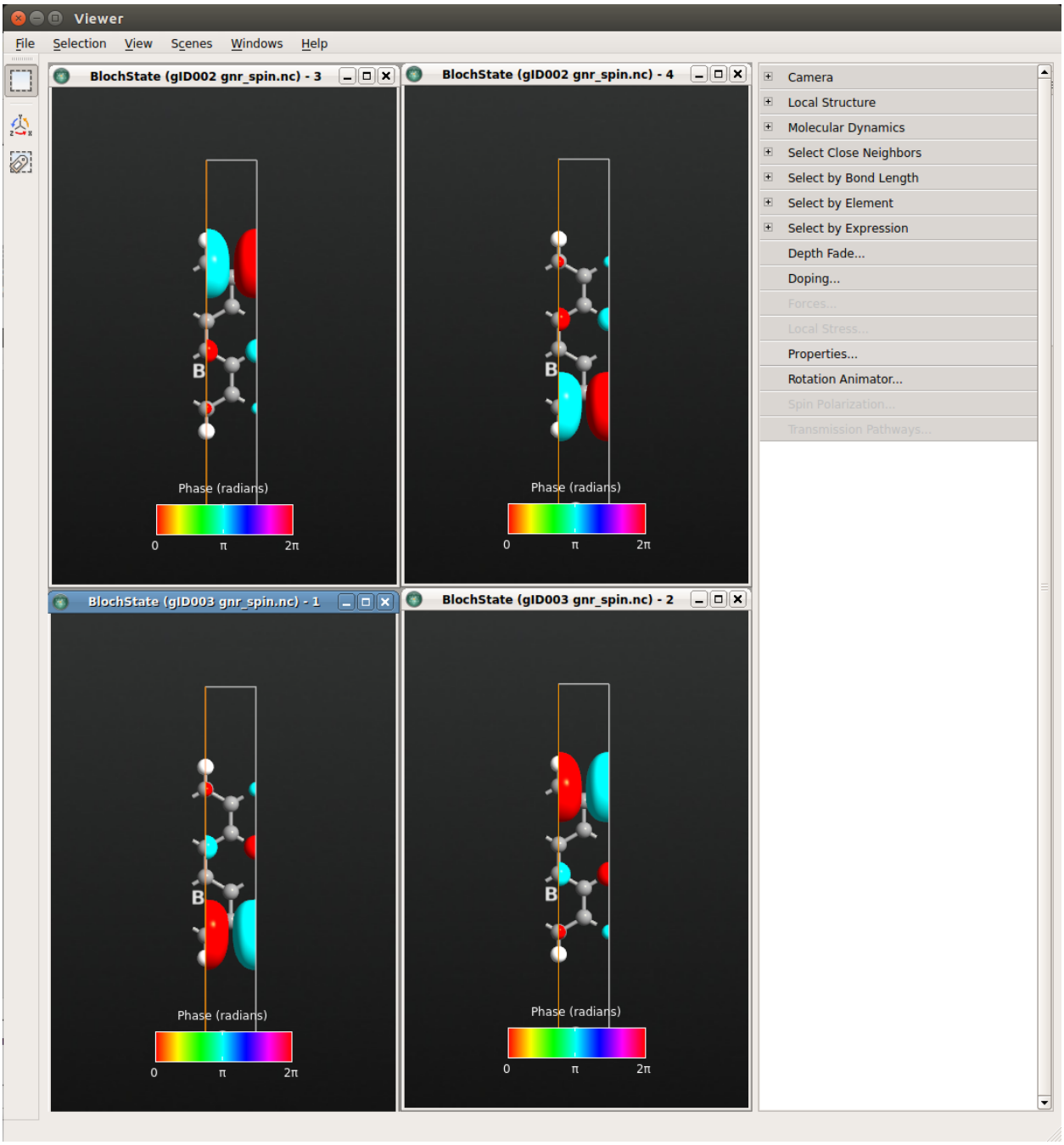


Fig. 3.91: Spin up (right) and down (left) states at the Z point for the highest valence band (bottom) and lowest conduction band (top).

Text Representation - gnr_spin.nc gID014

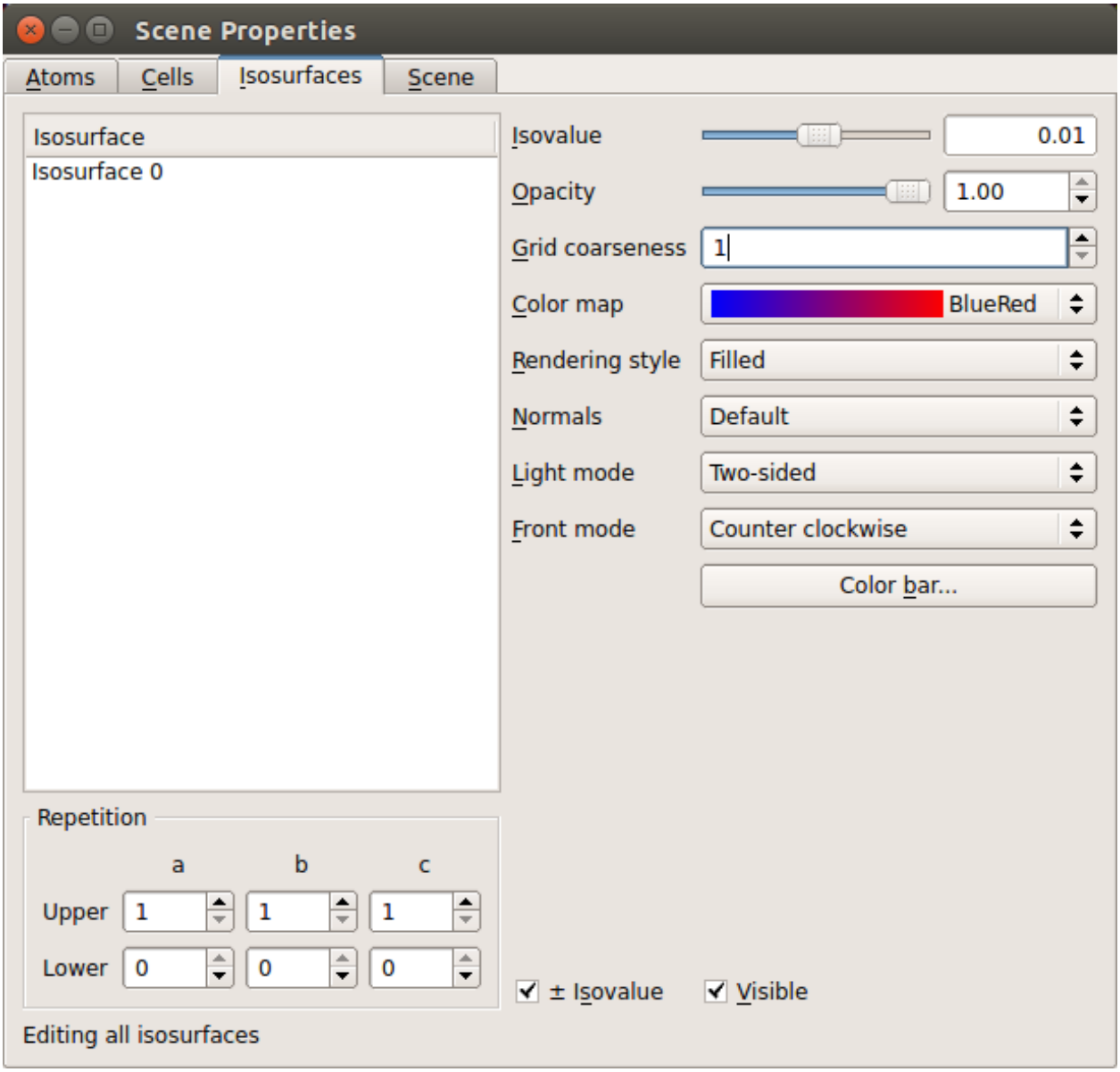
```
# Item:  0
# File:  /home/ulrik/tutorial-projects/12975-spin-dependent-bloch-states-in-
graphene-nanoribbons/gnr_spin.nc
# Title: gnr_spin.nc - gID014
# Type:  MullikenPopulation
+-------------------------------------------------------------------------------+
|                                                                               |
| Mulliken Population Report                                                    |
|                                                                               |
| ----------------------------------------------------------------------------- |
|                          |                                                    |
| Element   Total  Shell   | Orbitals                                           |
|                          |                                                    |
|                          |      s                                             |
|   0   C   2.170  0.498   |  0.498                                             |
|           1.926  0.486   |  0.486                                             |
|                          |      y       z       x                             |
|                  1.642   |  0.543  0.507  0.592                               |
|                  1.410   |  0.534  0.499  0.376                               |
|                          |     xy      zy   zz-rr      zx  xx-yy              |
|                  0.030   |  0.002  0.010  0.007  0.007  0.004                 |
|                  0.031   |  0.002  0.010  0.007  0.007  0.005                 |
|                          | ---------------------------------------------- |
|                          |      s                                             |
|   1   C   1.926  0.486   |  0.486                                             |
|           2.170  0.498   |  0.498                                             |
|                          |      y       z       x                             |
|                  1.410   |  0.534  0.499  0.376                               |
```

You can also choose to plot the spin difference density, $\rho_\uparrow - \rho_\downarrow$, or the normalized difference between the spin up and down density, $(\rho_\uparrow - \rho_\downarrow)/(\rho_\uparrow + \rho_\downarrow)$.

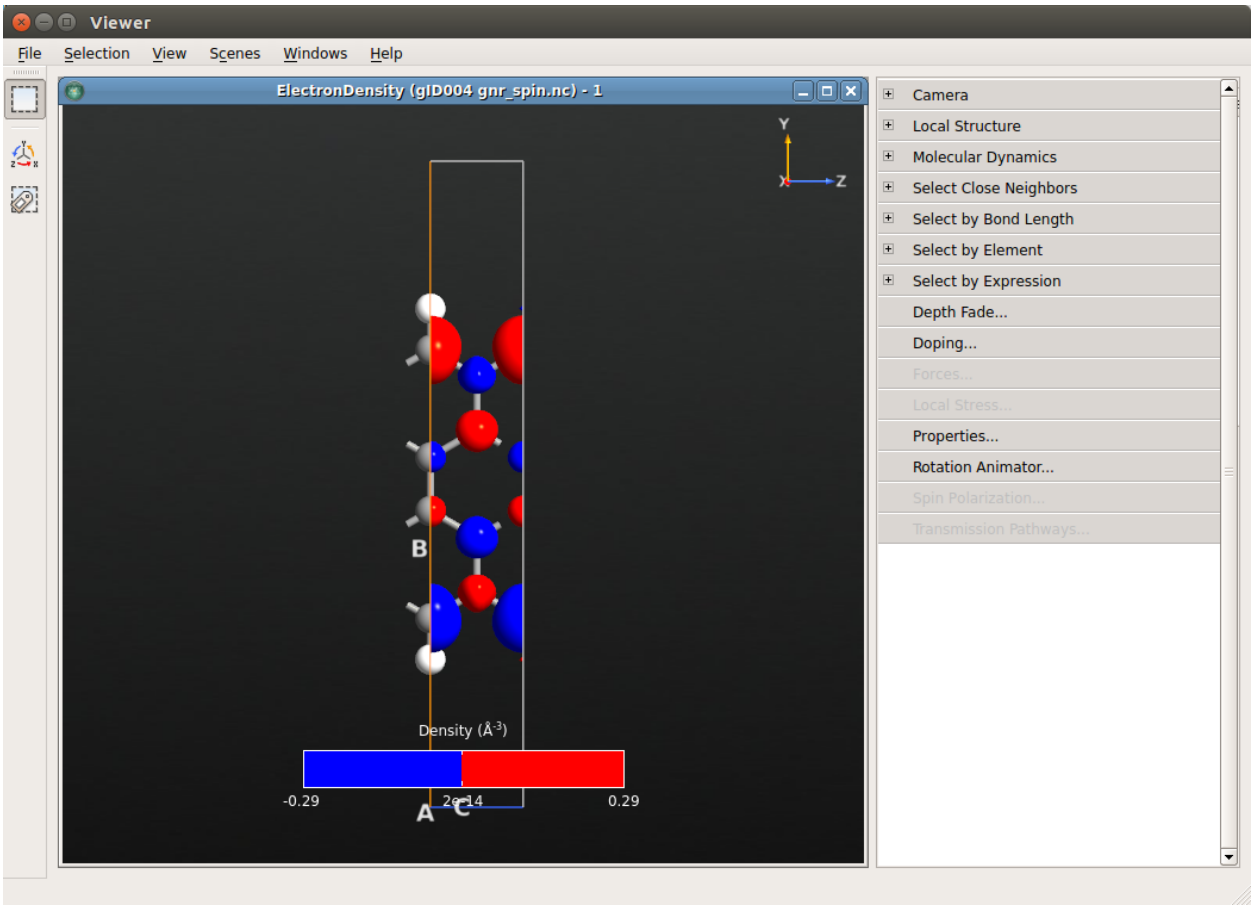
All the information you need is contained in the ElectronDensity object saved in the `gnr_spin.hdf5` file.

Plotting the spin density

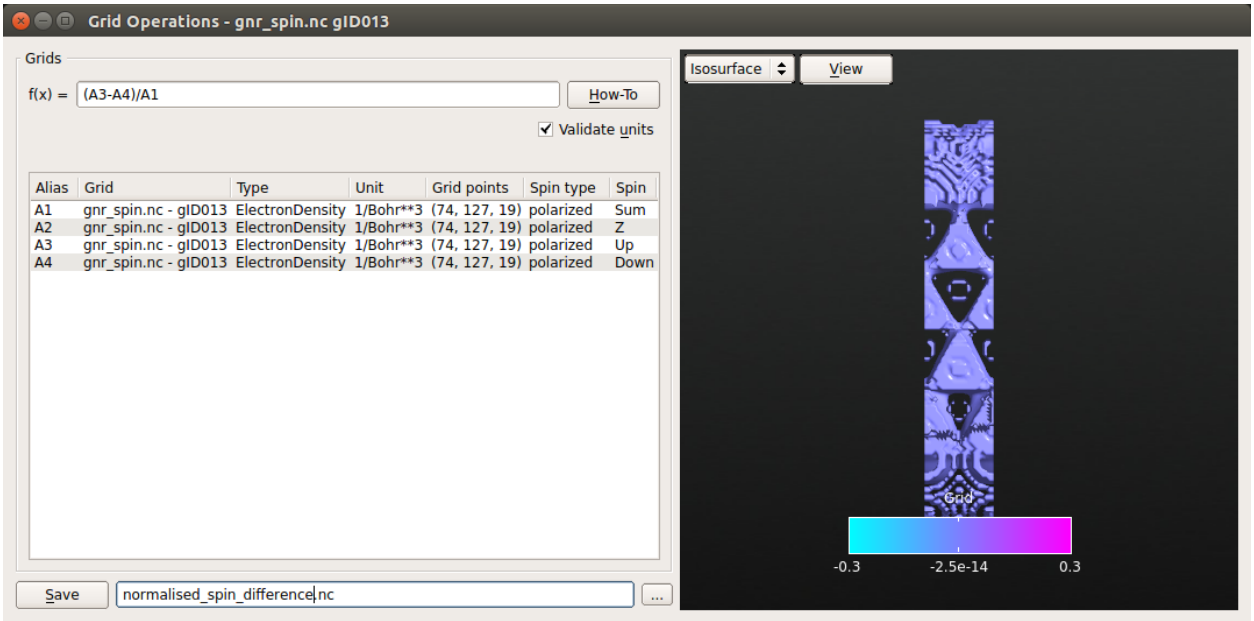
First we plot the direct spin difference density. Select the ElectronDensity object on the LabFloor and click **Viewer**. Go to properties and set the *Isovalue* to 0.01, the *Color map* to BlueRed and check the box for $\pm$ *Isovalue* at the bottom. The window should now look as the one shown below.



If you also overlay the bulk configuration as explained previously, you should get a plot like the one shown below.

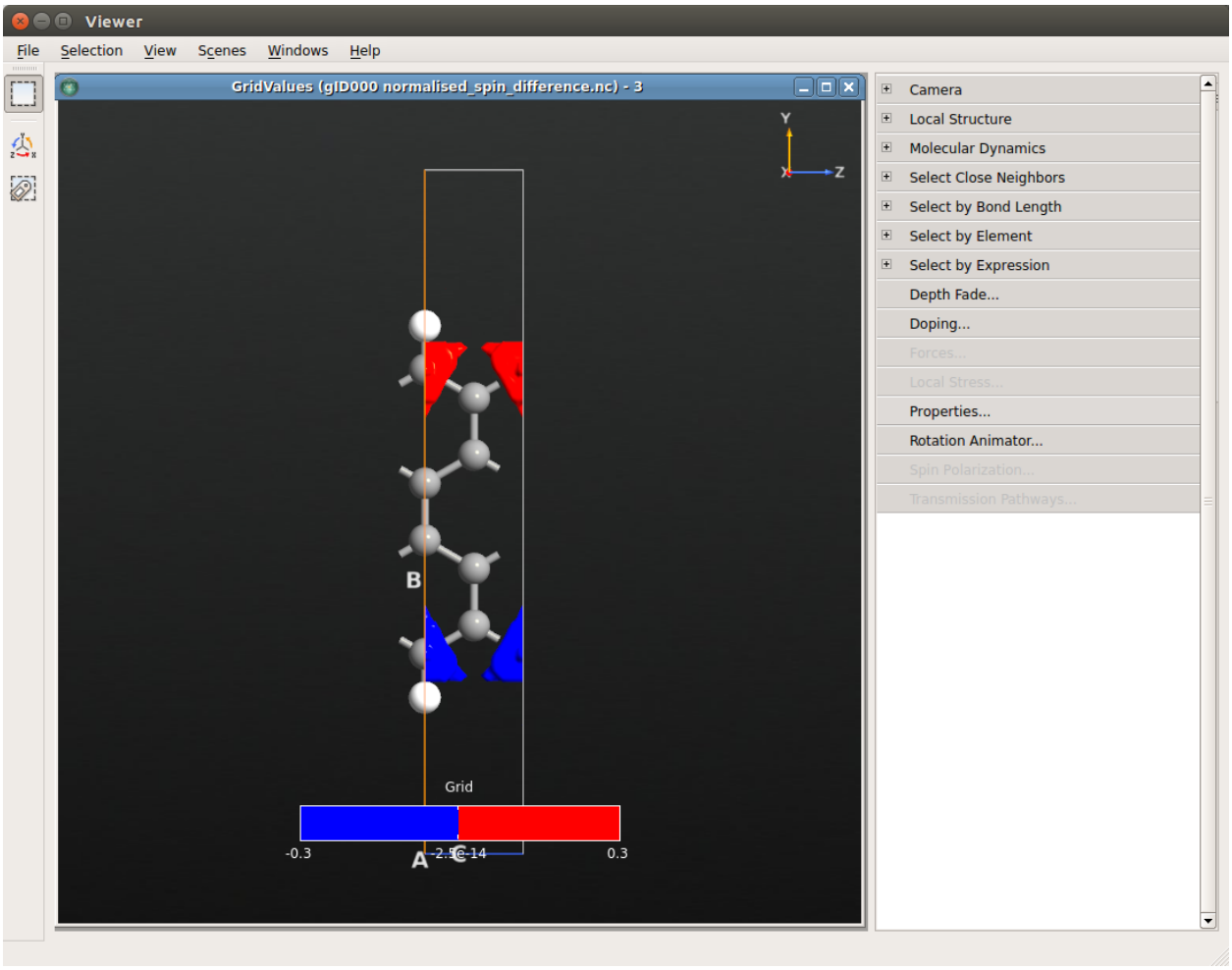


In order to plot the normalised spin difference density, go to LabFloor and select ElectronDensity in the `gnr_spin.hdf5` file and click on **Grid Operations**. Here, you can see the elements that are contained in the ElectronDensity object which are marked by an alias. In the *f(x)* field you can now write any mathematical expression. In this case write (A3-A4) / A1. Finally, save the result in a new file called `normalized_spin_difference.hdf5`.



Note: You can select more then one object present in the LabFloor by holding Ctrl and click on **Grid Operations**. In this way you will be able to operate on different data set from different calculations. However, note that the dimensions of the grids must be the same and the units of the grids in the expression must be consistent.

The new file will appear in the LabFloor. Select it and plot the content with the Viewer. In order to get a better view, open **Properties** on the menu in the right-hand side. Again, check ± *Isovalue* at the bottom, change Isovalue to 0.02 in the upper right and change the color map to BlueRed.



You can also average the spin polarization along the x-axis (perpendicular to the graphene sheet). The following script will generate the plot below, using the matplotlib module which is integrated in **ATK**.

`density_average.py`

Drop the script on the Job Manager, and it will generate the plot. The plot will be saved in the file `av.png`.

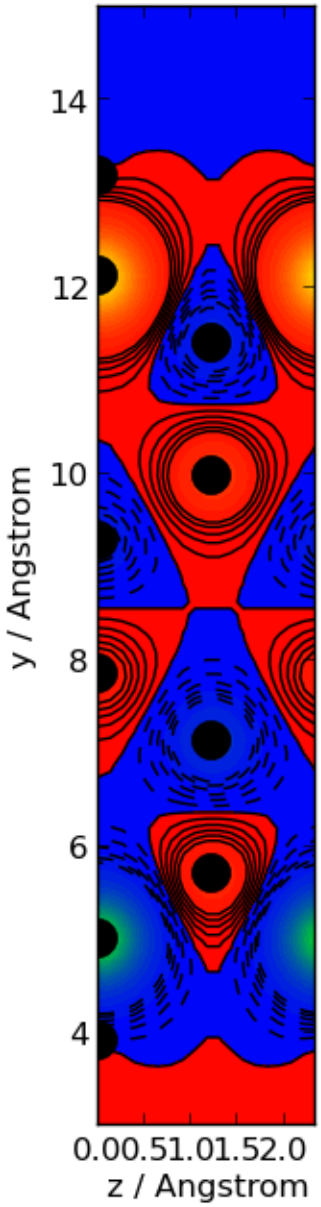


Fig. 3.92: Averaged spin polarization density. The filled circles indicate the atom positions.

References

3.7 Phonons & Thermal Transport

3.7.1 Calculating and using Dynamical Matrix

Version: U-2022.12

Downloads & Links

DynamicalMatrix

The *DynamicalMatrix* is an object that is used to calculate many properties related to dynamics of the crystal lattice e.g *Mobility*, *RamanSpectrum*, *DeformationPotential*, *DielectricTensor* and more. In this tutorial you will learn how to calculate the dynamical matrix and use it to perform a *PhononBandstructure* calculation. In particular, you will learn how to set up the calculation and which parameters are critical. You will also learn how to optimize the performance of this type of calculations by an optimal parallelization strategy.

- You will first perform a **DFT-LCAO** calculation with the LDA exchange-correlation functional. You will go through all the relevant parameters used in a phonon calculation.
- Towards the end, you will use the classical potentials available in the **ForceFieldCalculator** and compare the results.

Prerequisites

This tutorial is an introduction to the Dynamical Matrix functionality in QuantumATK and only requires basic familiarity with the QuantumATK interface. This can be achieved via the following basic tutorials and guides:

- *Introduction to the Workflow Builder*
- *Introduction*
- *Organize your data in the Nanolab data view*
- *Geometry optimization: CO/Pd(100)*
- *Job Manager for local execution of QuantumATK scripts*

Create the Workflow

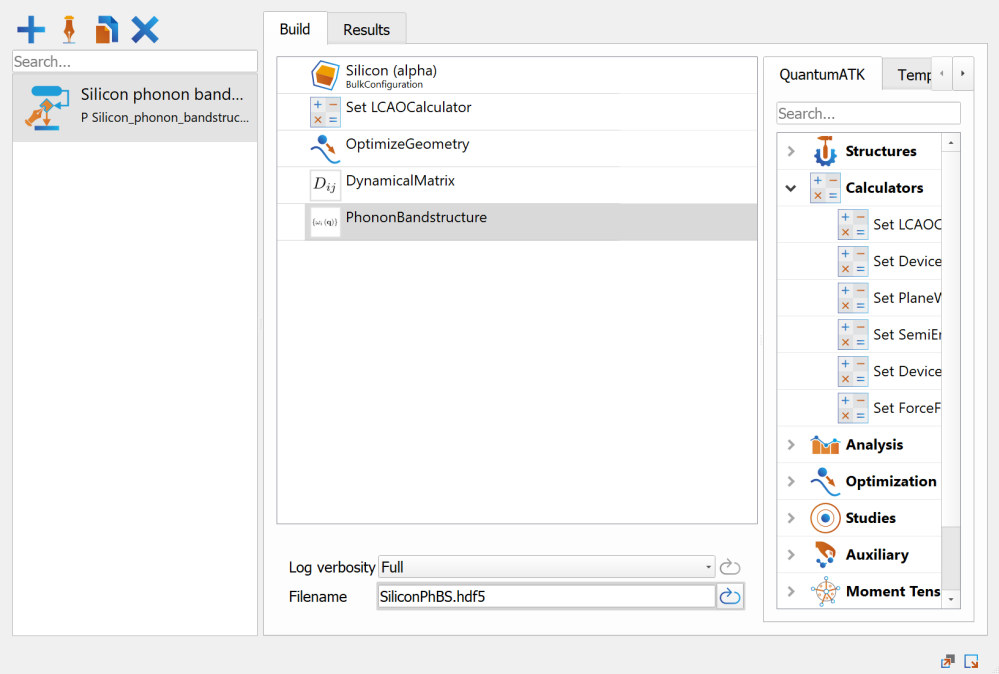
Go to the **Builder** and import the *Silicon (alpha)* structure from the built-in database. Then send the structure to the *Workflow Builder* and add the following script blocks:

1. **LCAOCalculator**
2. **OptimizeGeometry**

3. **DynamicalMatrix**

4. **PhononBandstructure**

The resulting workflow should look like this:



Now we will go through the settings of each block and edit the important parameters.

LCAOCalculator Settings

To get good results for the dynamical matrix one has to converge the computational settings with respect to a few key parameters:

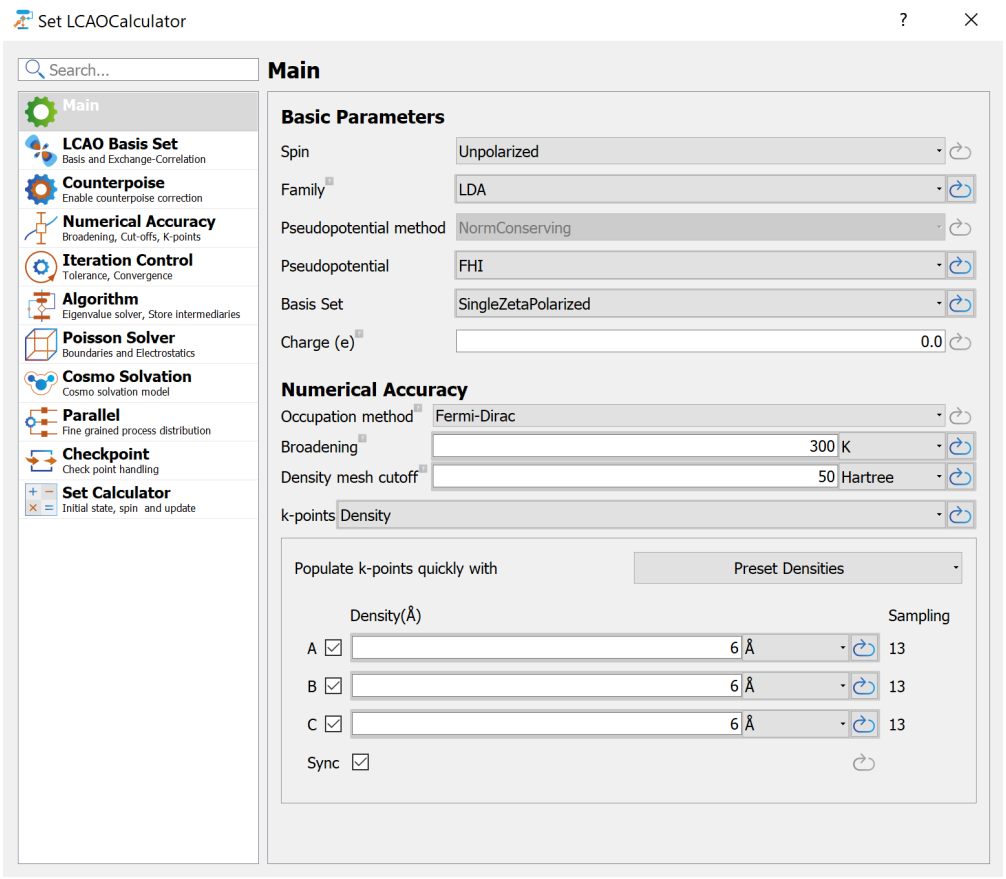
Note:

- In this example we use the light weight Single Zeta Polarized basis set. This basis set will result in a decent phonon-bandstructure at low computational cost. For production simulations we would recommend using e.g. the default PseudoDojo-Medium basis set which is generally more accurate.
- Please also note that the **Density mesh cutoff** needed is material and pseudo potential specific, so while 50 Hartree is enough for SIlicon, it is too low for other materials. Generally one should start from the default value and increase until convergence is reached.

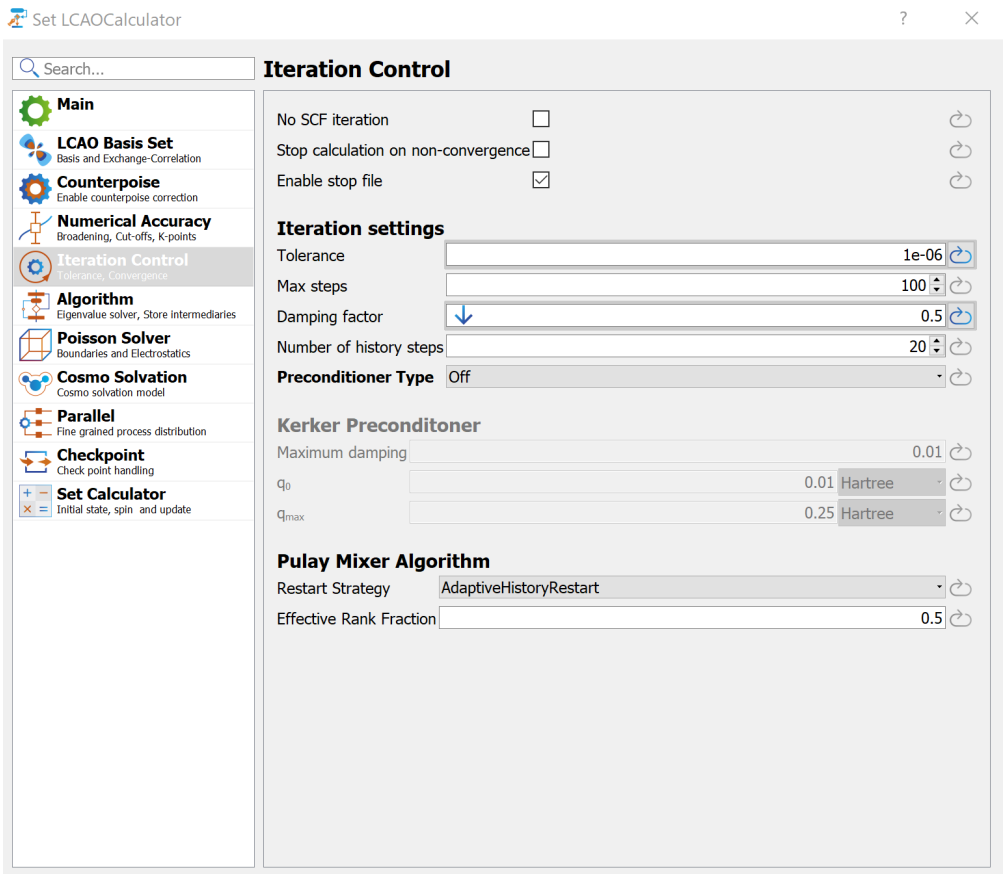
In the *Workflow Builder* do the following:

- Open the **LCAOCalculator** block.
- In the **Main** section, in **Basic Parameters** select *LDA* under **Family**, *FHI* under **Pseudopotential** and *SingleZetaPolarized* under **Basis Set**.

•In the **Main** section, in **Numerical Accuracy** set **Broadening** to *300 Kelvin*, Increase **Density mesh cutoff** to *50 Hartree* and k-point density to *6 Angstrom*.



•In the **Iteration Control** section, set a **Damping factor** of *0.5*. Also reduce the **Tolerance** to *0.000001*.

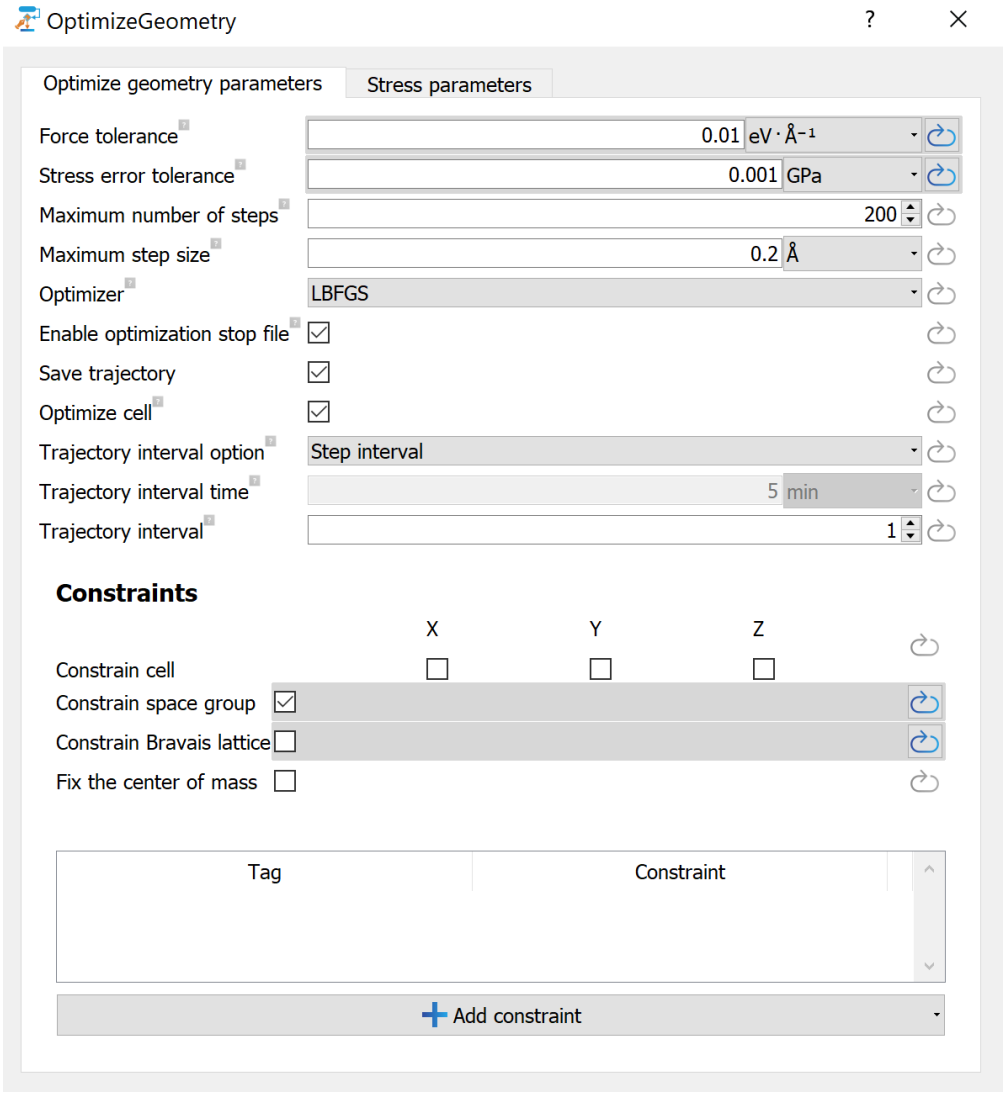


Tip:

- It can often be necessary to increase mesh cutoff and k-point density from the default values for DynamicalMatrix calculations to ensure convergence of the phonon bandstructure.
- For non-metals in general we recommend using *300 Kelvin* broadening.
- For DynamicalMatrix calculations it is generally recommended to use a tighter tolerance than the default one.
- It is possible to reduce the number of iteration steps by tuning the damping factor and number of history steps used.

Lattice optimization Settings

Open the **OptimizeGeometry** block, and modify the parameters as in the figure below to perform a force and stress minimization with tighter-than-default convergence parameters.



Tip:

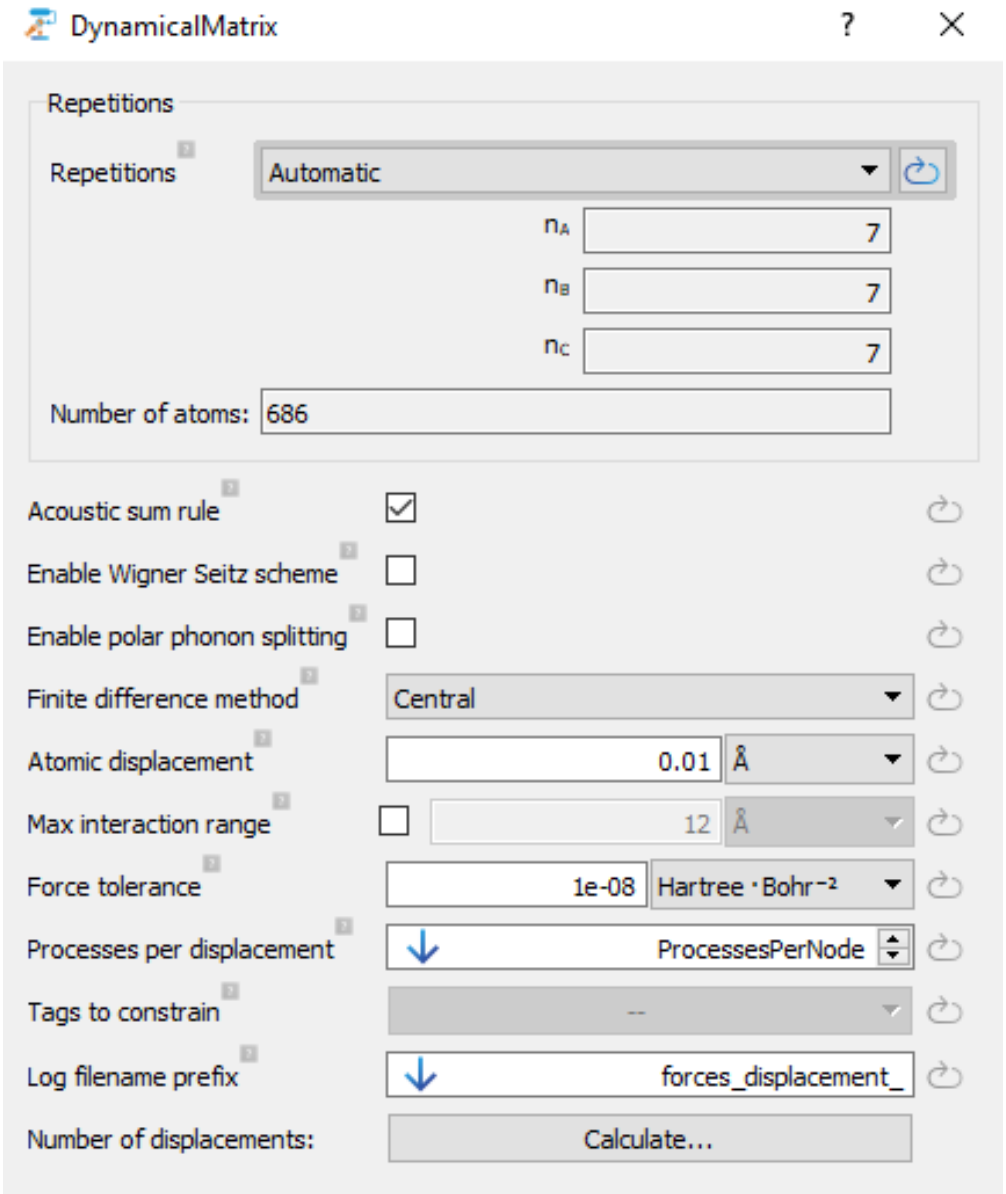
- We generally recommend reducing the target stress and forces when calculating the dynamical matrix.
- To get good structures for calculations of electronic properties the default settings are usually sufficient.

Dynamical matrix Settings

The dynamical matrix is the second derivative of the energy, corresponding to the first derivative of the forces. The first derivative of the forces are calculated using a finite difference scheme, where each symmetrically unique atom in the central cell is displaced along all cartesian directions, also called frozen phonon calculations.

For crystals with small unit cells, a periodically repeated super-cell of the unit cell is often needed to accurately account for long-range interactions in the dynamical matrix. **QuantumATK** can handle this automatically. By default cell is repeated such that all atoms within a pre-defined element-dependent distance from the atoms in the unit cell are included in the repeated cell. This is enough for most systems. It is also possible to directly set the number of repetitions used in each direction. You can see more details in the *Notes* section of the manual entry for `DynamicalMatrix`.

•In the D_{ij} **Dynamical Matrix** object we will use the default settings.



•Open the $\{\omega_\nu(\mathbf{q})\}$ **PhononBandstructure** block and choose 400 points per segment and the G, X, K, G, L Brillouin zone

route.

Note: The TAT of the calculation is highly sensitive on the number of repetitions for the simple fact that the repeated system can become very large. For highly complex or large systems where the automatically detected repetitions result in a very high TAT one can sometimes reduce the number of repetitions. It is however important for production calculations to always check convergence with respect to e.g. phononbandstructure.

To test the effect of using different numbers of repetitions, we repeat the calculation setting **Repetitions** to 3x3x3 and 5x5x5. The full workflow can be downloaded here `Silicon_phonon_bandstructure.hdf5`. To change the repetitions set **Repetitions** to *Custom* and choose the desired numbers.

Running the calculation

You are now ready to run your DFT phonon bandstructure calculation. This can be done by sending the script to the **Job Manager** using the **Send To** icon .

However, there are a couple of details to consider before running the job:

Parallelization

In **QuantumATK**, the dynamical matrix can be calculated by parallelizing over displacements. The number of displacements needed depends on the system and can be calculated by opening the **Dynamical Matrix** object and pressing the *Calculate* button under **Number of Displacements**.

The amount of processes reserved for each displacement can be controlled using the **Processes per displacement** parameter. In general we recommend one of two parallelization strategies:

1. The default setting *ProcessesPerNode* divides the the displacements across nodes. When running on a single node this will result in all displacements being calculated in serial.
2. Depending on the system considered and the hardware setup it is sometimes advantageous to run all displacements in parallel on as many processes as possible. For that strategy one should take the total number of processes subtract one which will be reserved as a delegator and then evenly distribute the remaining processes on the number of displacements.

   For this tutorial, we will use the default *ProcessesPerNode* setting on a single 20 core node.

System size

The three systems run in this tutorial, 3x3x3, 5x5x5 and 7x7x7 repetitions, consists of 54, 250 and 686 Silicon atoms, respectively. Besides the running time you should also consider the amount of memory required.

Timing

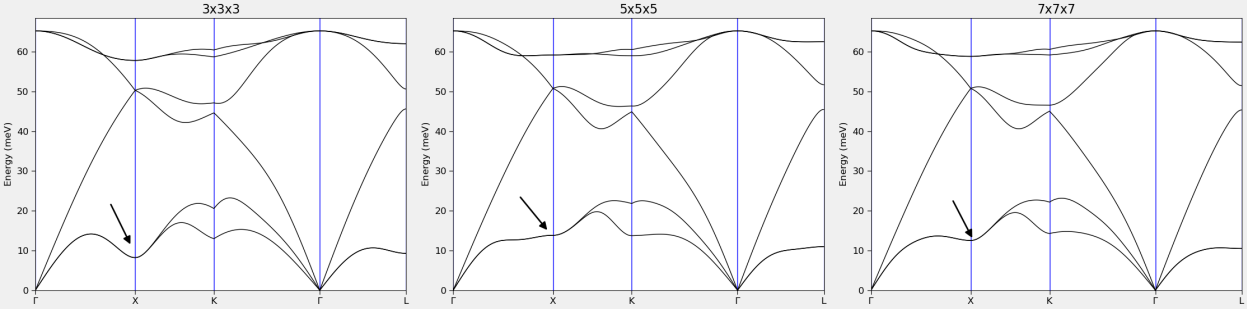
The table below reports the total time required to run the entire phonon bandstructure workflow with the parameters set above. The calculations are run in parallel with 20 MPI processes:

Repetition	3x3x3	5x5x5	7x7x7
Total time	1m29s	22m26s	1h39m44s

Analyzing the results

When the job is done you will find in the **Data** view the **PhononBandStructure** object. Highlight it and press *Open* to display it.

The figure below reports results for the 3x3x3, 5x5x5 and 7x7x7 systems defined above.



You immediately notice that the 3x3x3 repetition is too small to give even qualitatively good results, while the intermediate size system is already a pretty decent approximation. However, you can also notice that a small difference exists between the acoustic modes near the X-point for the 5x5x5 and the 7x7x7 systems.

The dynamical matrix is now stored together with the **BulkConfiguration**. This means you can read the dynamical matrix and perform another **PhononBandstructure** calculation (on a different Brillouin path for example) or a **PhononDensityOfStates** calculation in just a few seconds.

Speeding up the calculation with ForceFields

QuantumATK offers a large set of classical potentials in the **ForceFieldCalculator**. These potentials are well suited to study vibrational properties of materials, such as the phonon bandstructure of silicon calculated in this tutorial.

Note: Classical potentials are available only for a limited amount of systems. As your system becomes more complex including e.g. interfaces, defects or many elements you will often find that no relevant classical potentials exist. In this case **QuantumATK** offers the ability to train your own machine learned potential using Moment Tensor Potentials. Please get in touch if you would like to try out this functionality and do not have access already.

Send the silicon bulk configuration to the **Workflow Builder**. Add a **ForceFieldCalculator**. Now a stress optimization and phonon bandstructure calculation can be set up as explained above.

In **ForceFieldCalculator** select the most appropriate potential available for silicon, such as the Tersoff or the Stillinger and Weber potentials.

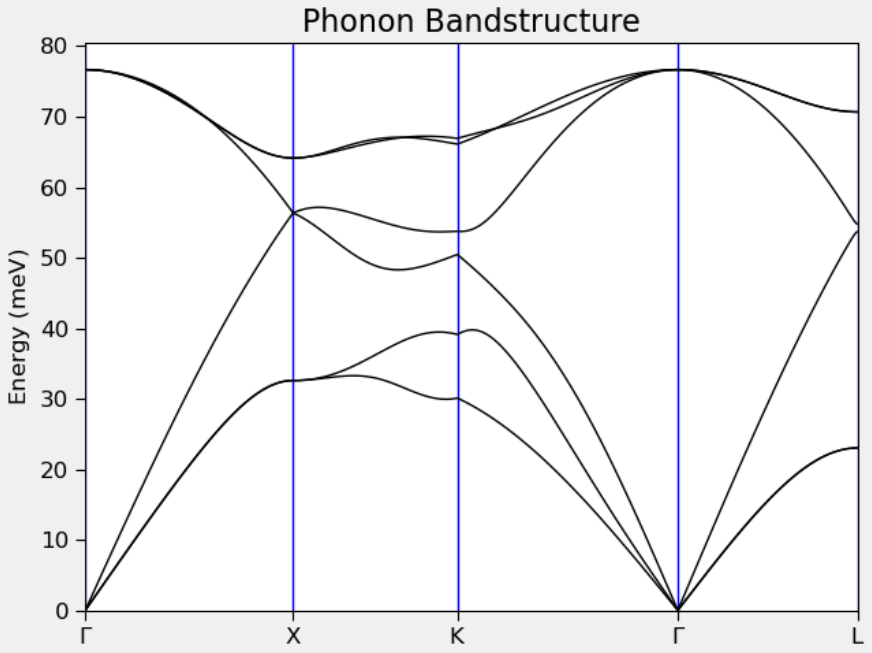
Each potential is provided with a literature reference, which you should check to determine if the potential is well suited for your system and for the property you are interested in.

Set up the **OptimizeGeometry** block and the **PhononBandstructure** analysis object exactly as for the **LCAOCalculator** case and run the calculation. The calculation will take just a few seconds on a normal desktop machine.

The phonon bandstructure of silicon calculated with the *Tersoff_si_2005 potential*[1] is reported in the figure below.

1

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Phys. Rev. B, **71**, p. 035211, (2005).



References

3.7.2 Vibrational modes and Vibration Visualizer

In this tutorial you will study the vibrational properties of nanoscale systems using the **VibrationalMode** analysis object and the **Vibration Visualizer** plugin. Two systems are investigated:

1. Monolayer MoS_2.
2. A nanophononic metamaterial

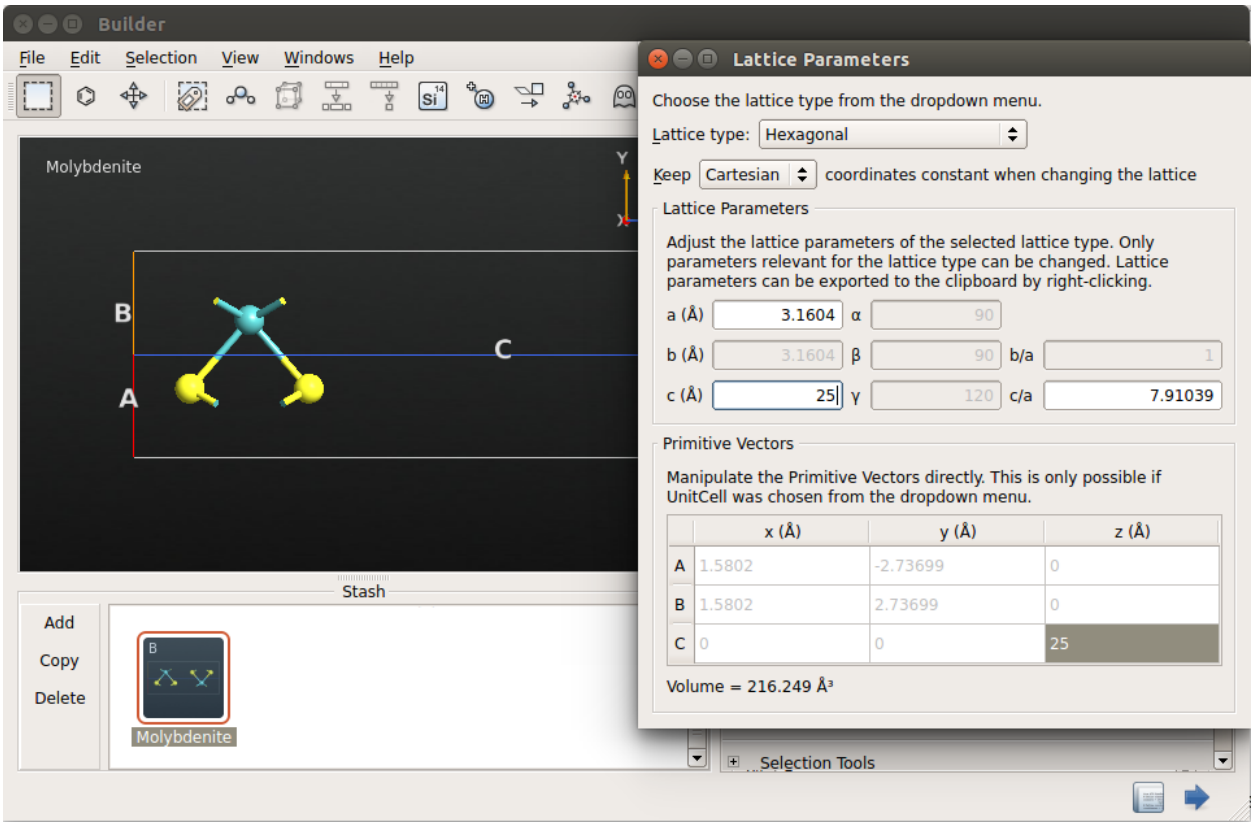
Note: Calculation of vibrational properties relies on the **dynamical matrix** of the configuration, which describes the force response on all atoms due to out-of-equilibrium displacements of all other atoms.

MoS_2 monolayer

Generate the MoS_2 structure

Open the **Builder**, and do the following:

- Use the *Database* plugin via *Add* → *From Database* and locate molybdenite. Add it to the Stash.
- Use the mouse to select and delete the 3 atoms to the right in the cell. This leaves a single layer of MoS_2.
- Open *Bulk Tools* → *Lattice Parameters* and change the z direction to 25 Å.



•Center the structure using *Coordinate Tools* → *Center*.

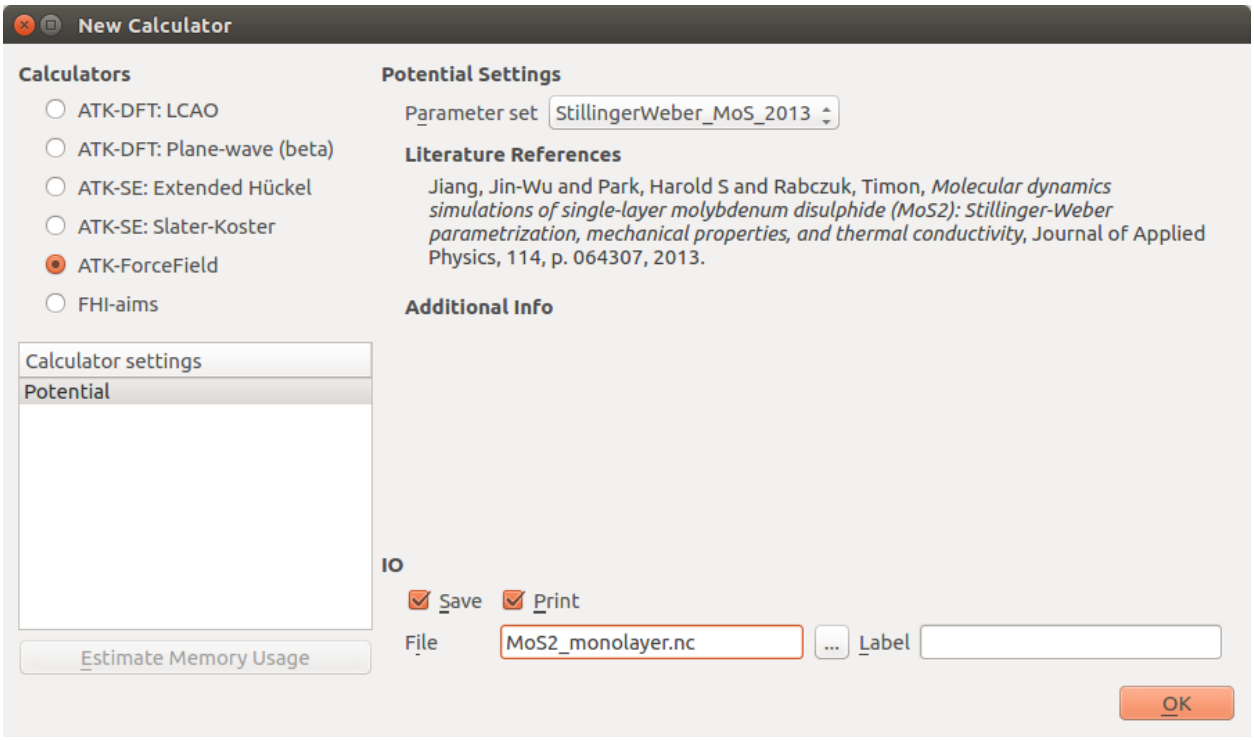
•Send the structure to the **Script Generator** by using the icon. The next step is to create the script to optimize the structure.

Optimize the MoS_2 structure

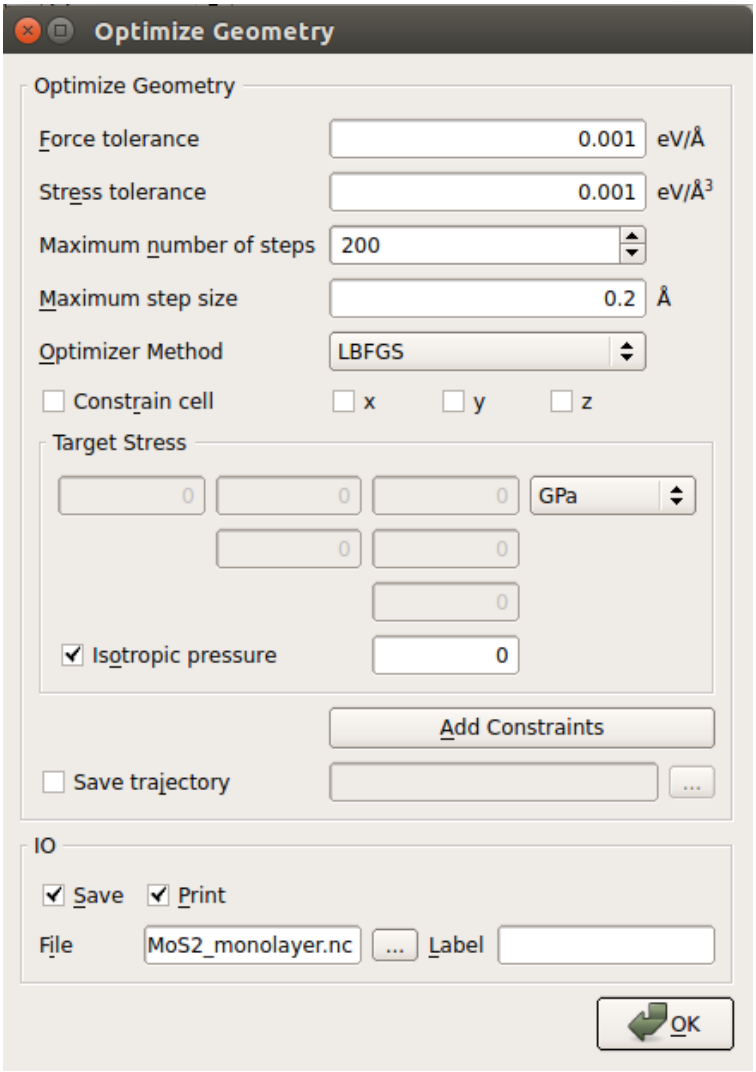
•Add **New Calculator** and **OptimizeGeometry** blocks.

•In New Calculator, use the **ATK-ForceField** engine and in particular the StillingerWebber_MoS_2013 potential`.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.



•In **OptimizeGeometry**, reduce the force and stress tolerance to 0.001 eV/Å and 0.001 eV/Å^3, click OK and save the script before running the optimization.



•Save the script as `mos2_relax.py`. Run the script by sending it to the Job Manager, or from a terminal with the command

```
$ atkpython mos2_relax.py > mos2_relax.out
```

An example script `mos2_relax.py` is available to download.

Note: In the case of monolayer, the potential agrees exactly with Ref.[1]. However, the Lennard-Jones potential reported in the reference for the inter-layer coupling has not yet been implemented.

---

[1] Jin-Wu Jiang, HaroldăS. Park, and Timon Rabczuk. Molecular dynamics simulations of single-layer molybdenum disulphide (MoS2): Stillinger-Weber parametrization, mechanical properties, and thermal conductivity. *Journal of Applied Physics*, 114(6):, 2013. doi:10.1063/1.4818414.

Phonon band structure and density of states

The phonon band structure and the phonon density of states are now easily obtained:

•Drop the optimized MoS_2 `BulkConfiguration` onto the Scripter and add a New Calculator by double clicking it.

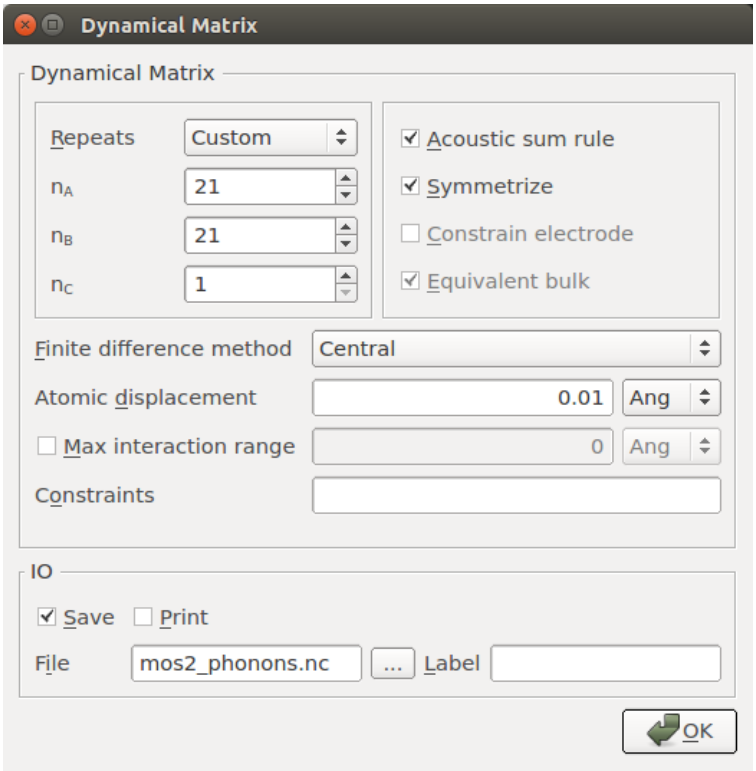
•Change the default output file to `mos2_phonons.nc`.

•Check if the calculator is set to **ATK-ForceField**.

•Add the *Analysis → PhononDensityOfStates* and *Analysis → PhononBandstructure* analysis objects.

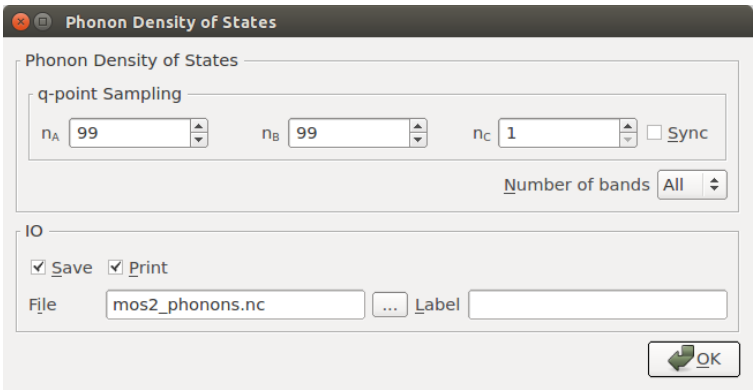
•Note that the *DynamicalMatrix* was automatically added also.

•Open the DynamicalMatrix block and select *Repeats → Custom* and set the repeats to (21,21,1).



•Open **PhononBandstructure** and set Points pr. segment to 400 and set the Brillouin zone route to the sequence [G, M, K, G].

•In *Analysis→PhononDensityOfStates*, untick Sync and set the q-point sampling to (99, 99, 1).



Run calculation and analyze results

Run the calculation. Once finished, the **LabFloor** should contain the **PhononBandstructure** and **PhononDensityOfStates** objects:

- PhononBandstructure

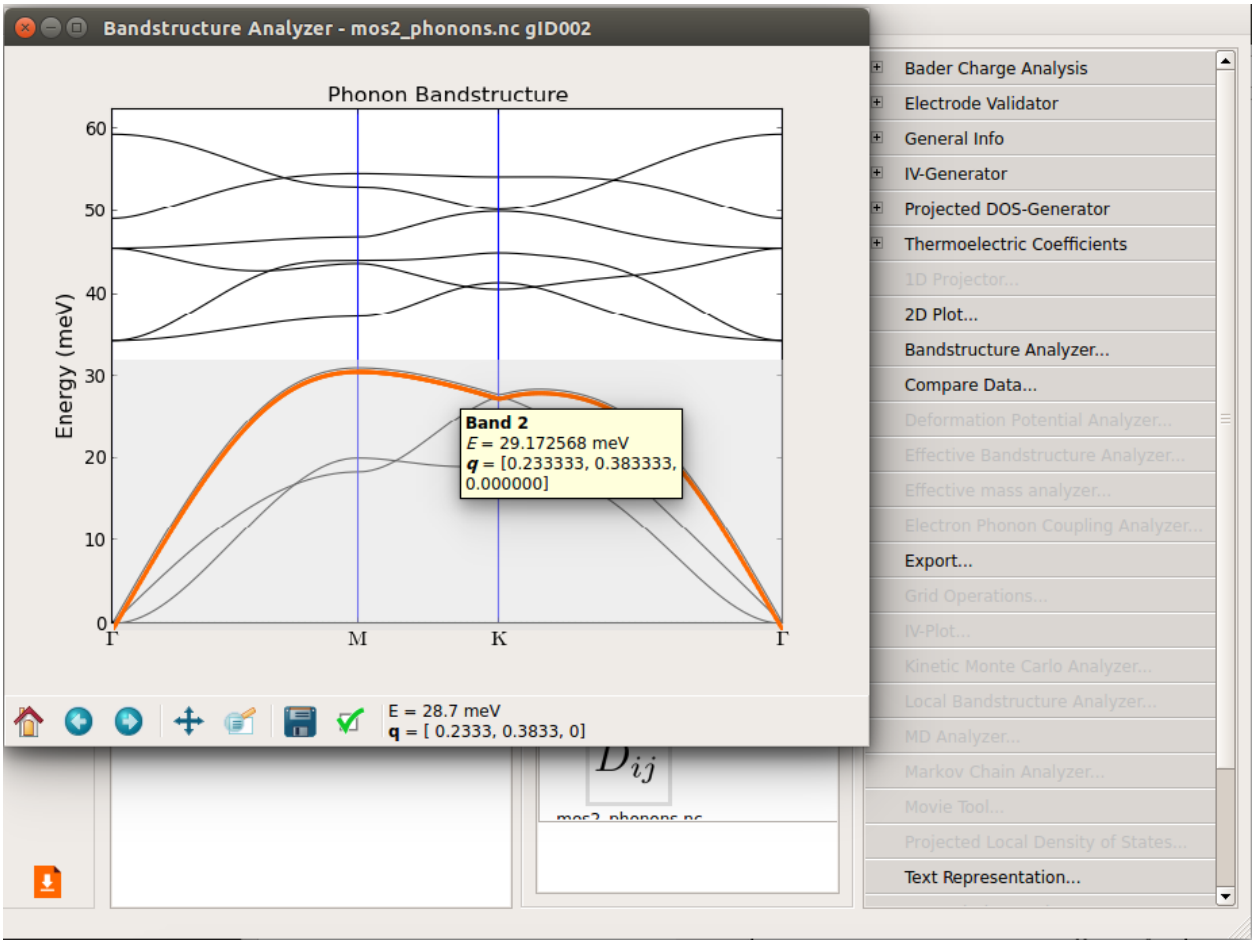
$$\{\omega_i(\mathbf{q})\}$$

- PhononDensityOfStates

$$D(\omega)$$

Inspect the phonon band structure and density of states using the **Bandstructure Analyzer** and **2D Plot** tools. The results are in complete agreement with literature data[2].

[2] Jin-Wu Jiang, HaroldăS. Park, and Timon Rabczuk. MoS2 nanoresonators: Intrinsically better than graphene? *Nanoscale*, 6:3618–3625, 2014. doi:10.1039/C3NR05991J.



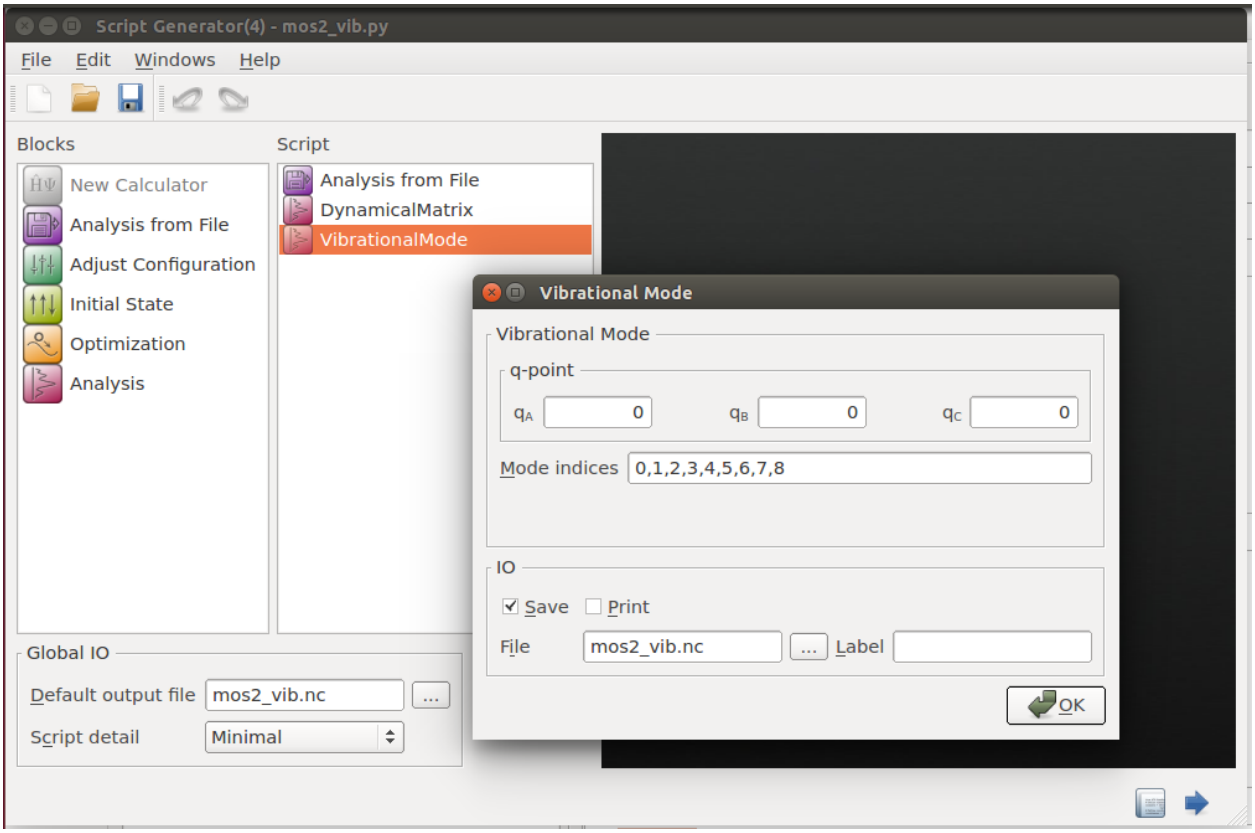
Vibrational Modes

To visualize the vibrational modes, a simple post-processing is sufficient:

•Open the Script Generator and do the following:

•Add an **Analysis from File** block, double click to open it, select the output file `mos2_phonons.nc`, and choose the subject id as gID000.

•Add a *Analysis→VibrationalMode* analysis object and select the q-point coordinates and mode indices of interest.



The script will recalculate the dynamical matrix, which is inconvenient. To reuse the already calculated dynamical matrix, send the script to the editor and change the line defining the dynamical matrix such that it reads object gID001 from the file `mos2_phonons.nc`.

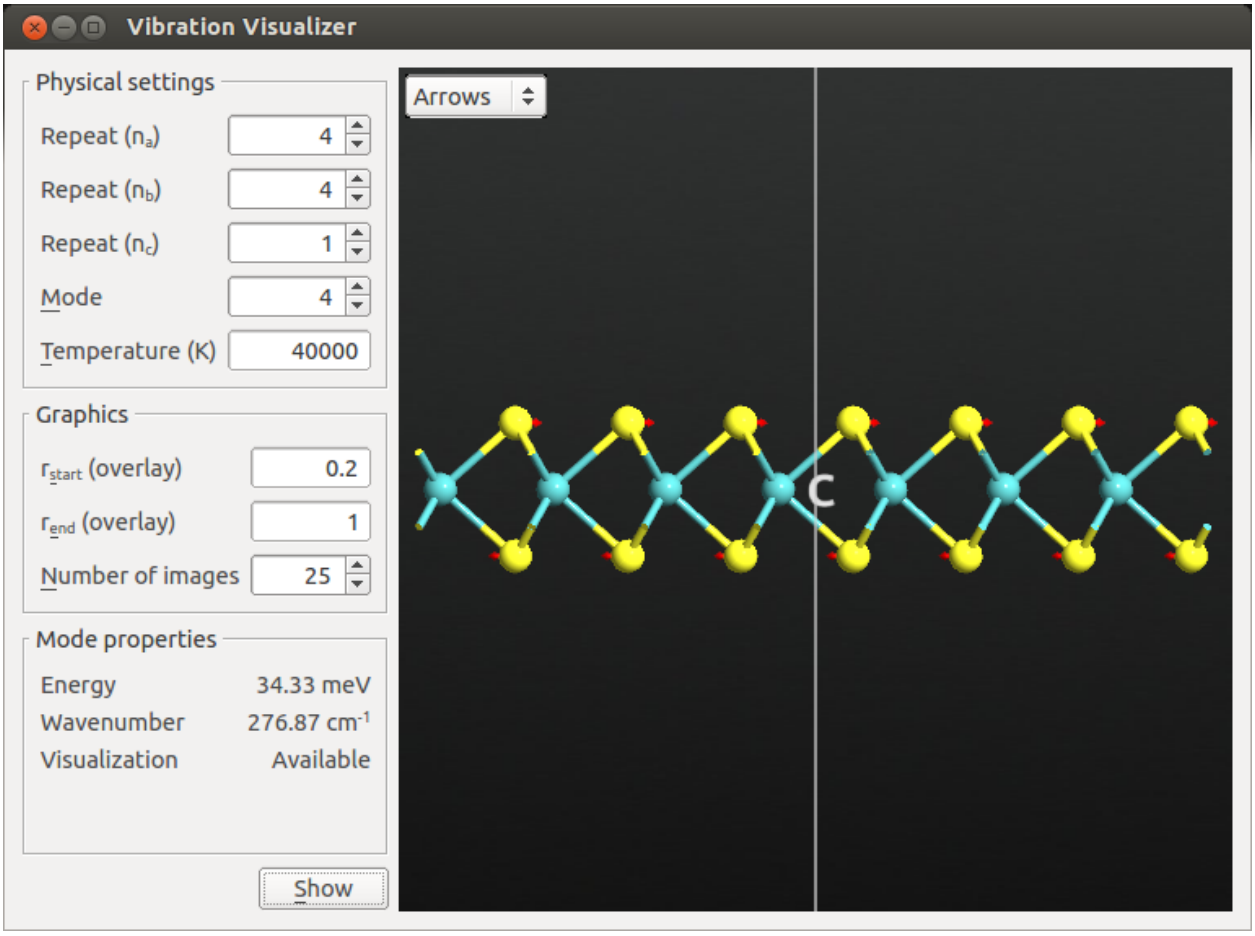
```
# ------------------------------------------------------------
# Analysis from File
# ------------------------------------------------------------
configuration = nlread('mos2_phonons.nc', object_id='gID000')[0]

# ------------------------------------------------------------
# Dynamical Matrix
# ------------------------------------------------------------
dynamical_matrix = nlread('mos2_phonons.nc', object_id='gID001')[0]

# ------------------------------------------------------------
# Vibrational Mode
# ------------------------------------------------------------
vibrational_mode = VibrationalMode(
    configuration=configuration,
    dynamical_matrix=dynamical_matrix,
    kpoint_fractional=[0, 0, 0],
    mode_indices=[0,1,2,3,4,5,6,7,8]
    )
nlsave('mos2_vib.nc', vibrational_mode)
```

Execute the script, and when the calculation is complete you will find the **VibrationalMode** object on the **LabFloor**:

- VibrationalMode

$\mathbf{u}_{i,\mathbf{q}}(\mathbf{r})$

Select it and open it by clicking the **Vibration Visualizer** plugin. You can then visualize the vibrational mode as an animation.



Once you are done with the settings you can right click on the plot to export as a picture or as an animated gif. The two animated gifs below illustrate mode 4 at (276.87 cm^{-1}) and at M (252.32 cm^{-1}), respectively.

You can play around with the Vibration Visualizer and compare the results with the literatures values in[Page 1110, 234].

[3] Mahdi Ghorbani-Asl, Nourdine Zibouche, Mohammad Wahiduzzaman, AugustoăF. Oliveira, Agnieszka Kuc, and Thomas Heine. Electromechanics in MoS2 and WS2: nanotubes vs. monolayers. *Scientific Reports*, 2013. doi:10.1038/srep02961.

[4] Rusen Yan, JeffreyăR. Simpson, Simone Bertolazzi, Jacopo Brivio, Michael Watson, Xufei Wu, Andras Kis, Tengfei Luo, Angela R.ăHight Walker, and HuiliăGrace Xing. Thermal Conductivity of Monolayer Molybdenum Disulfide Obtained from Temperature-Dependent Raman Spectroscopy. *ACS Nano*, 8(1):986–993, 2014. doi:10.1021/nn405826k.

Nanophononic metamaterials

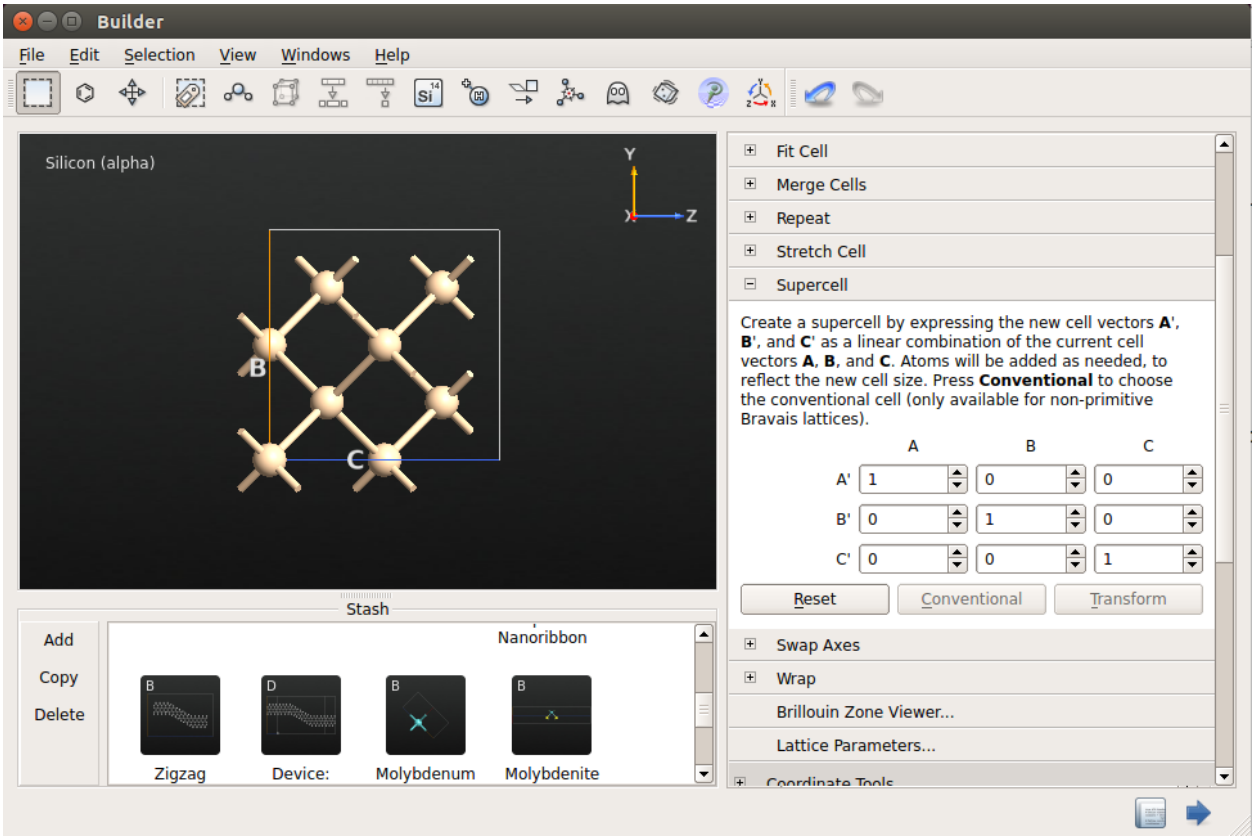
Another great example of the vibrational mode analysis is simulation of a nanophononic metamaterial. Bruce L. Davis and Mahmoud I. Hussein recently reported that specific vibrational modes of pillared silicon thin films will strongly suppress the thermal conductivity of the films due to the local resonance[5]. Lets study the vibrational feature of the structure using **ATK**:

Calculation Setup

Open the **Builder** and do the following:

•Use *Add* → *From Database* to add Silicon (alpha) to the Stash.

•Transform the primitive cell into the conventional cell using *Bulk Tools* → *Supercell*.



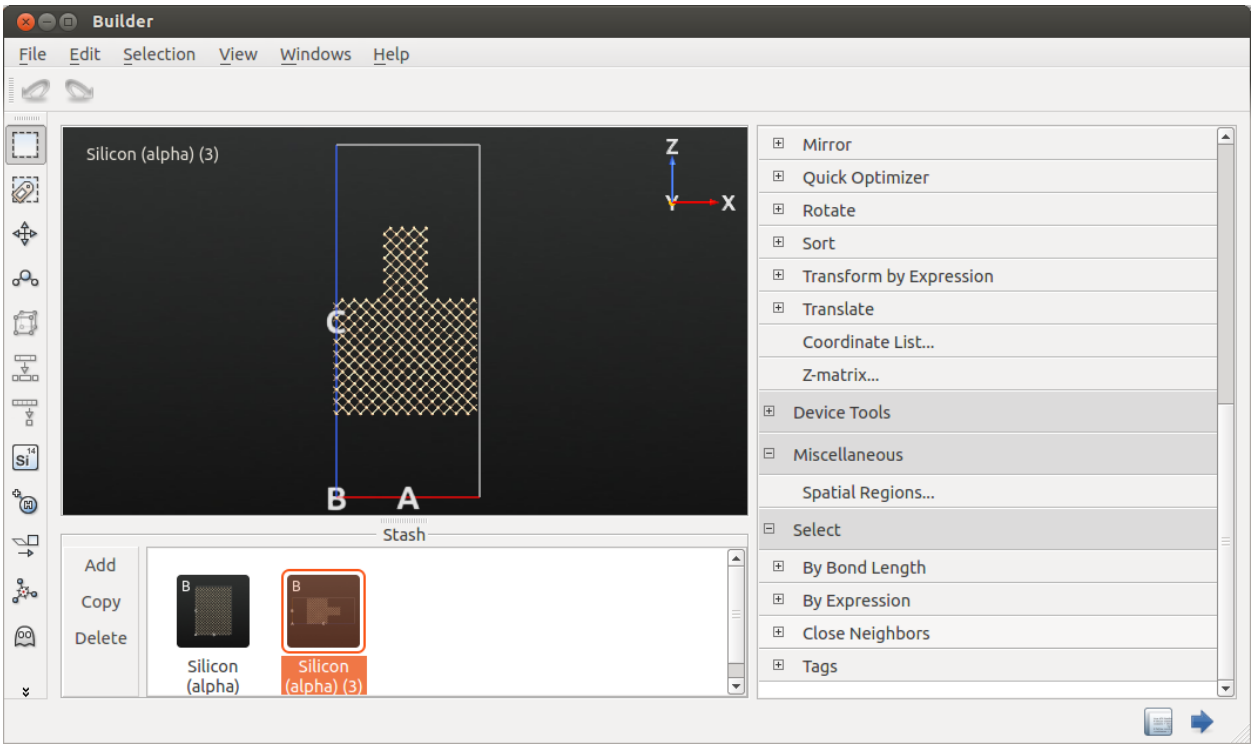
•Use the *Bulk Tools* → *Repeat* to repeat the structure 6x6x8 times to create a configuration with 2304 silicon atoms.

•Use the **Selection Tools**, in particular *Selection by Expression*, to cut out a pillared structure.

Tip: The expression z > 20 and x < 10 will select one particular part of the configuration that could then be removed by *Edit* → *Delete Atoms*.

•Adjust the structure by increasing the lattice vector and pillar pointing direction to create a vacuum region above and below the pillared film.

[5] BruceăL. Davis and MahmoudăI. Hussein. Nanophononic Metamaterial: Thermal Conductivity Reduction by Local Resonance. *Phys. Rev. Lett.*, 112:055505, Feb 2014. doi:10.1103/PhysRevLett.112.055505.



•Send the configuration to the **Scripter** and add a **Calculator**, a **PhononBandstructure** analysis object and two **VibrationalMode** objects.

•Setup the calculator to use ATK-ForceField.

Tip: You can leave the settings in **DynamicalMatrix** by defaults when calculating a 3D structure.

•Open **PhononBandstructure** and

–set Points pr. segment to 100,;

–set Number of bands to 50;

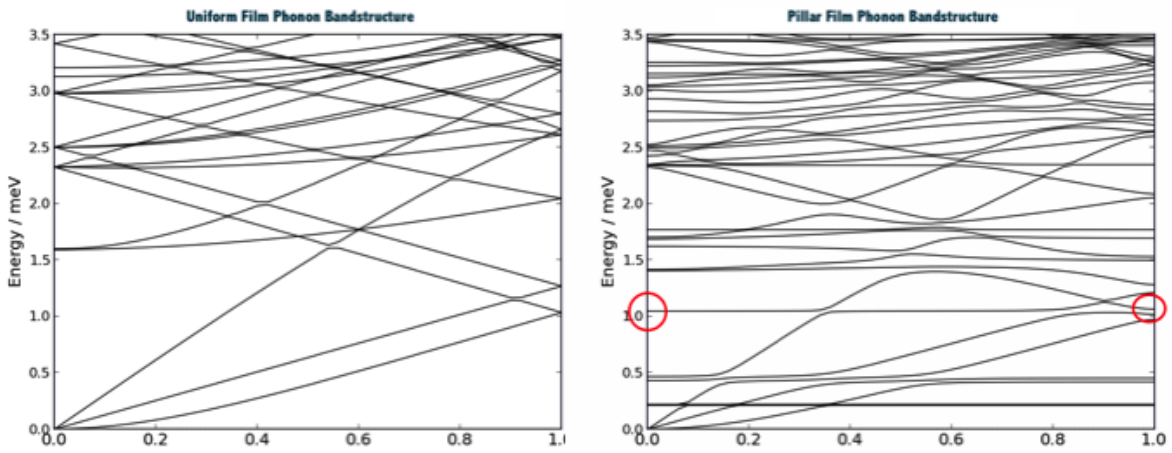
–set the Brillouin zone route to [G, X].

•Edit both **VibrationalMode** objects such as to calculate the vibrational modes in the point (0, 0, 0) and in the X point (0.5, 0, 0) and select mode indices [0,1,2,3,4,5,6,7,8] in both objects.

•Run the calculation. It will take a while for a single CPU core, but will be very efficient if the job is parallelized.

Results

Plot the calculated phonon bandstructure, and compare it with the phonon bandstructure of a uniform film. You can see that the array of pillars introduces a series of localized states (without dispersion), which are the states that lower the thermal conductivity of the silicon film, as explained in Ref.[Page 1114, 5].



The vibrational modes illustrated below are calculated for the localized state marked in the bandstructure plot above at and X points, respectively. The displacements are enhanced for a clearer view. This is controlled by the Temperature setting in the Vibration Visualizer plugin.

References

3.7.3 Phonons, Bandstructure and Thermoelectrics

Downloads & Links

ATK Reference Manual Basic QuantumATK Tutorial

Introduction

The purpose of this tutorial is to illustrate the use of the **ATK** phonon module. The tutorial is applied to small toy problems such that the calculations are very fast. In the first chapter you will calculate the phonon bandstructure and density of states, and learn about how these modules work. In the second chapter you will study a device system and calculate the electronic and phonon thermal transport coefficients, as well as the electrical transport coefficients. From those coefficients you obtain the thermoelectric figure of merit ZT.

Note: You will primarily use the graphical user interface **QuantumATK** for setting up and analyzing the results. To familiarize yourself with **QuantumATK**, it is recommended to go through the Basic QuantumATK Tutorial.

The underlying calculation engines for this tutorial is **ATK-ForceField** and **ATK-SE**. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK Reference Manual.

In order to run this tutorial, you must have a license for ATK-SE. If you do not have one, you may obtain a time-limited demo license by contacting us via our website: Contact QuantumATK.

Phonon Bandstructure of a Graphene Nanoribbon

Setting up the geometry

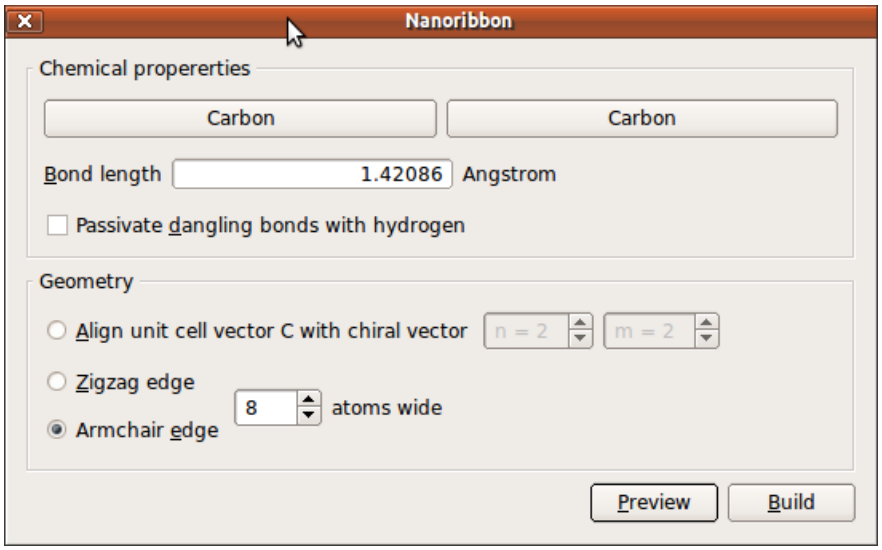
Start **QuantumATK** and create a new project. Launch the **Builder** by clicking the icon on the toolbar.

In the builder, click *Add* → *From Plugin* → *Nanoribbon*.

•Uncheck the `Passivate dangling bonds with hydrogen` checkbox.

•Select `Armchair` edge.

•Select the structure 8 atoms wide.



Press the `Build` button to build the structure.

In order to make sure that the phonon module can detect that the structure is one dimensional, you need to have ~ 7 Å vacuum on both sides of the structure. To this end open *Bulk Tools* → *Lattice Parameters*.

•Increase A-x to 20 Å.

•Increase B-y to 30 Å.

and close the widget.

Next, open *Coordinate Tools* → *Center*, and press `Apply` to center the nanoribbon in the unit cell. Send the structure to the **Script Generator**, by using the **Send To** icon in the lower right-hand corner of the **Builder** window.

Defining the phonon calculation

In the following you will optimize the geometry and calculate the Phonon bandstructure and DOS of the nanoribbon using a Tersoff potential.

In the **Script Generator**

•Add a **ForceFieldCalculator**

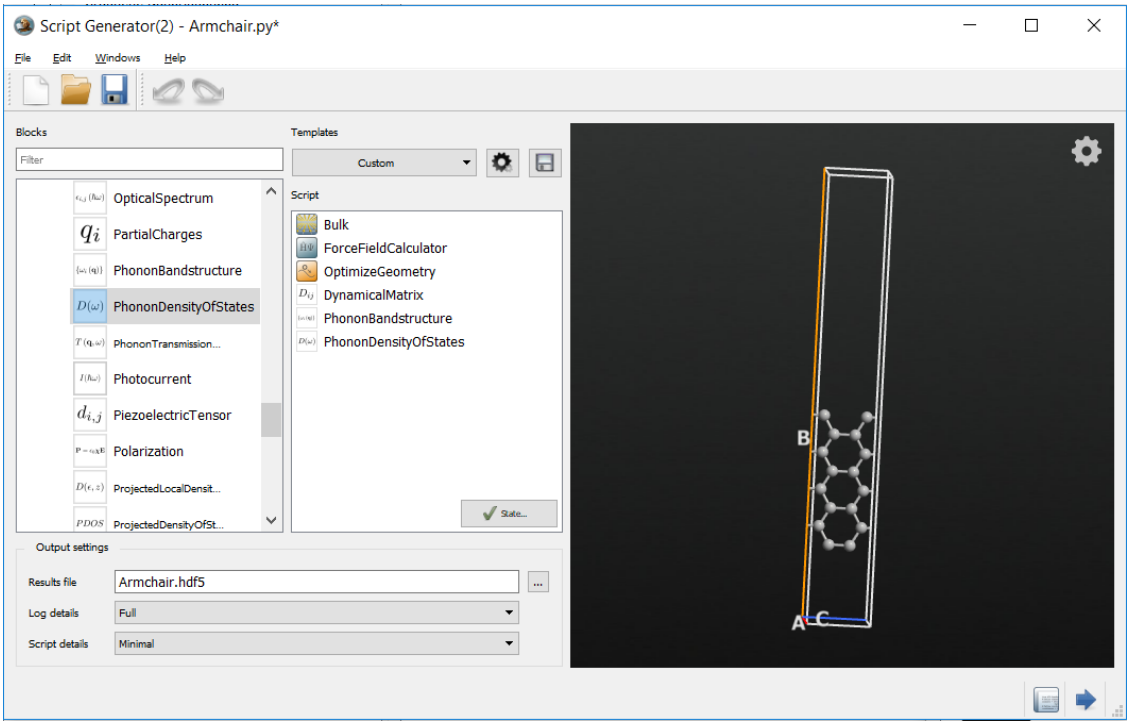
•Add an **Optimization > OptimizeGeometry**

•Add an **Analysis > PhononBandstructure**

•Add an **Analysis > PhononDensityOfStates**

•Change the output file to `armchair.hdf5`.

If you take a look at the blocks in the **Script Generator**, you will notice that a **DyamicalMatrix** block has automatically been added as the selected phonon analysis functions require the calculation of the dynamical matrix.

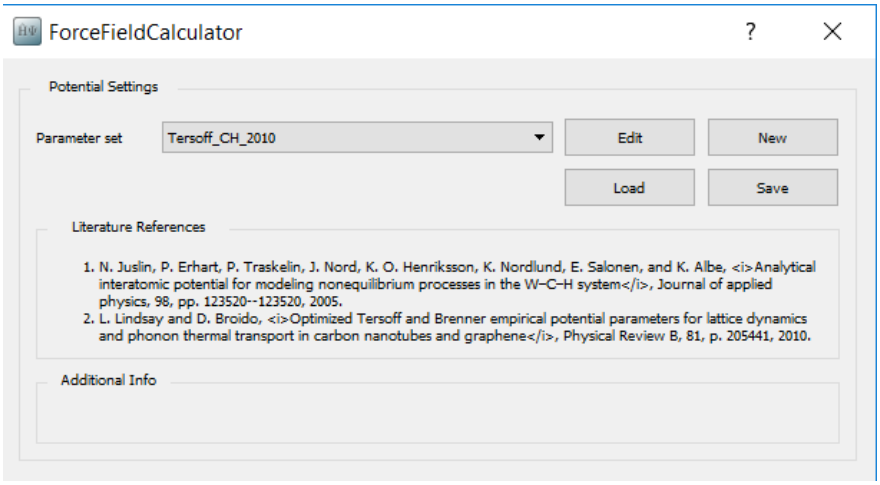


Adjusting script components

To set up the calculation parameters, make the following changes:

•Double-click the **ForceFieldCalculator** block and select the *Tersoff_C_2010* potential.

Note that depending on the elements present in your configuration you will be able to choose among several potentials. A literature reference is also shown, giving you the possibility to check exactly how the potentials have been generated. It is very important that the potential you will use has been generated for a similar system or for similar conditions as your system.



In this case you can immediately see that the *Tersoff_C_2010 potential* is potentially well suited for your phonon analysis: Optimized Tersoff and Brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene.

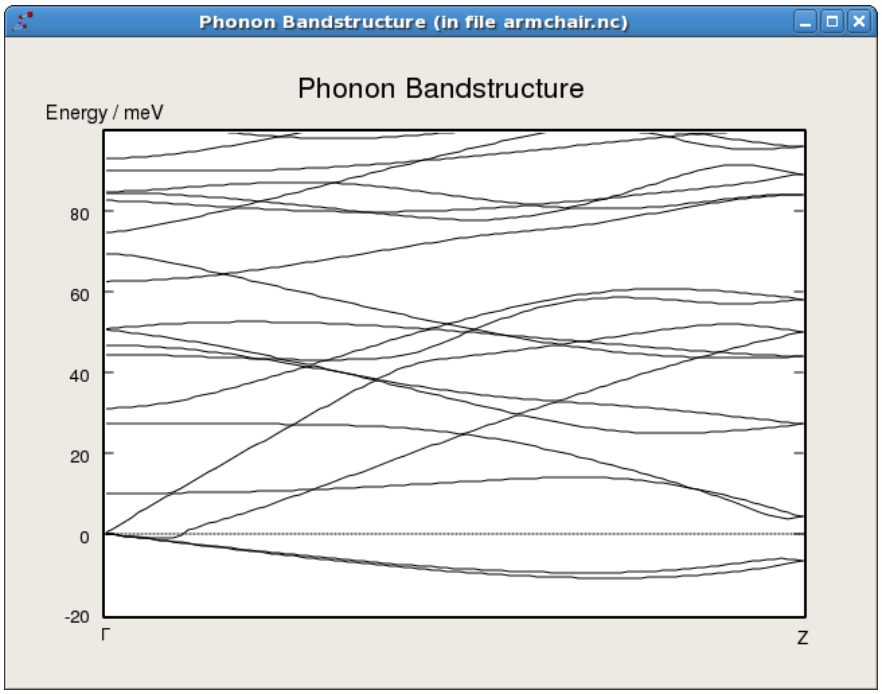
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

- Double-click the **OptimizeGeometry** and set maximum force to 0.01 eV/Å.
- Double-click the **PhononBandstructure** and set `Points per segment` to 100.
- Double-click the **PhononDensityOfStates**. In the *q-point Sampling* box select `Sampling`, uncheck the `Sync` box, and set `qc` to 51.

Transfer the script to the **Job Manager** (again using the Send To icon) and run the calculation.

Analyzing the Results

The generated output file `armchair.hdf5`, visible on the **LabFloor**, now contains a *PhononBandstructure* object. Select this PhononBandstructure object on the **LabFloor** and press **2D Plot** in the plugin panel. Zoom into the low frequency area to get the figure below.



Note the negative bands arising from stress in the nanoribbon which makes it unstable perpendicular to the plane.

Origin of negative phonon bands

The negative bands seen in the phonon spectrum indicate that the structure is at a saddle point and can gain energy by further relaxing. The problem is that you only performed a force optimization and not a stress optimization, thus there is a large stress in the structure and it would like to release the stress by relaxing perpendicular to the graphene plane.

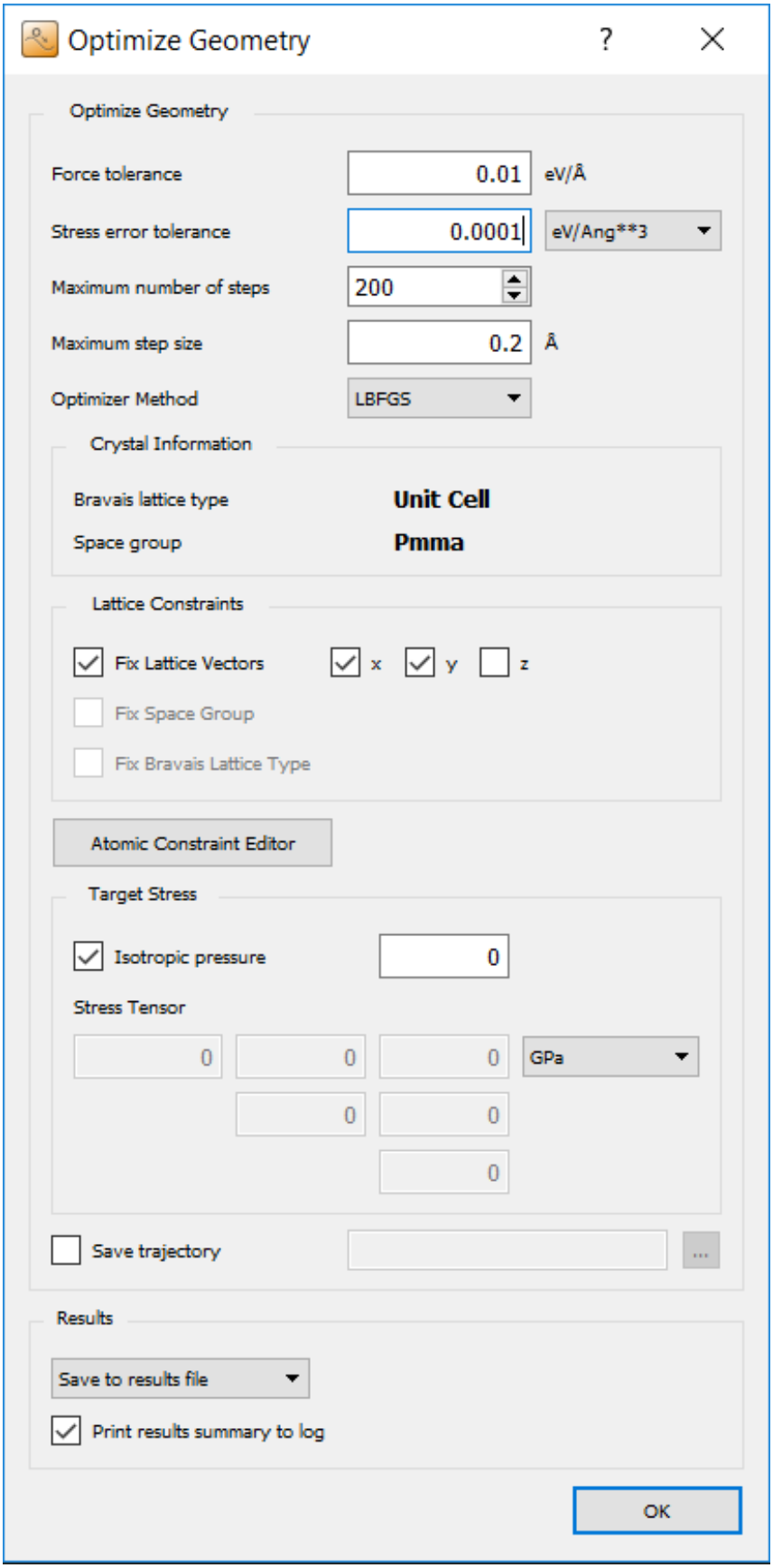
Relaxing the stress in the configuration

In order to relax the stress in the configuration, open the **Script Generator** again.

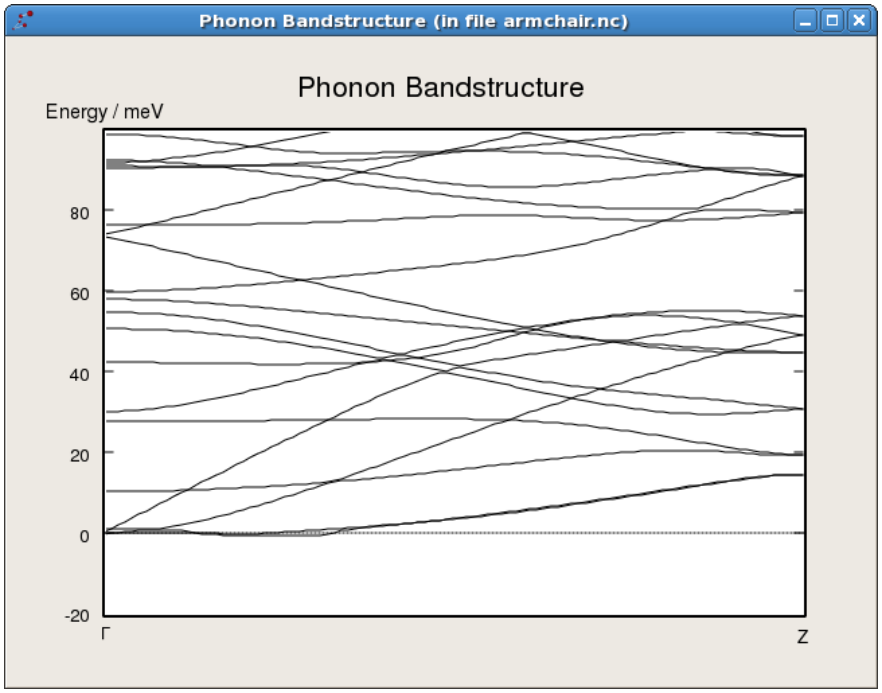
Tip: All open tools are available from the *Windows* menu at the top of each **QuantumATK** tool.

Next open the **OptimizeGeometry** block.

- Set `Maximum stress` to 0.0001 eV/Å ^{3}.
- Remove check from `z` in `Constrain cell` to allow optimization of the z-component of the cell.

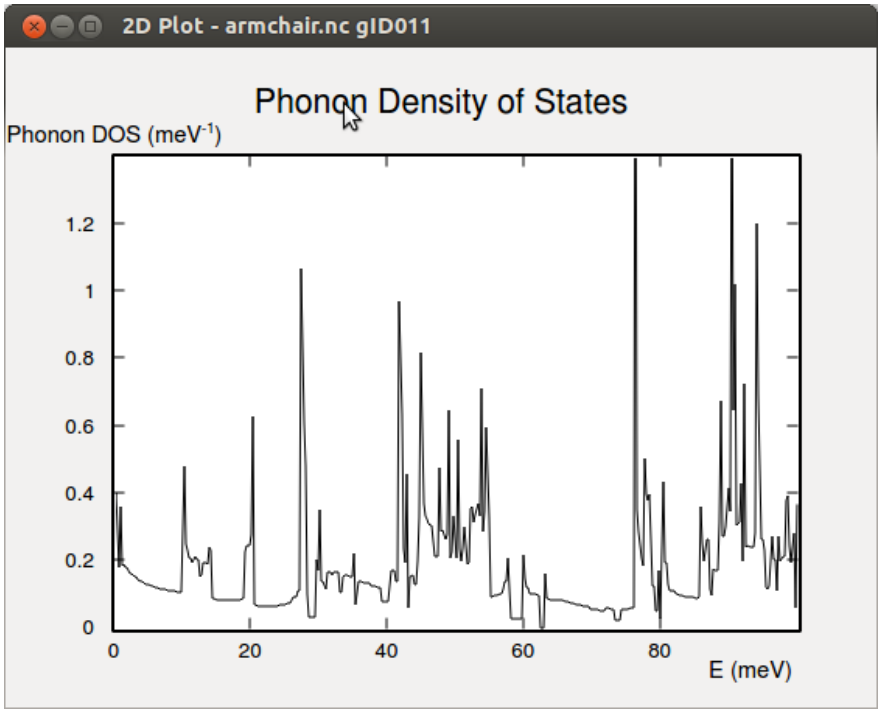

Optimize Geometry
Optimize Geometry
Force tolerance
0.01
eV/Å
Stress error tolerance
0.0001
eV/Ang**3
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer Method
LBFGS
Crystal Information
Bravais lattice type
Unit Cell
Space group
Pmma
Lattice Constraints
Fix Lattice Vectors
x
y
z
Fix Space Group
Fix Bravais Lattice Type
Atomic Constraint Editor
Target Stress
Isotropic pressure
0
Stress Tensor
0
0
0
GPa
0
0
0
Save trajectory
Results
Save to results file
Print results summary to log
OK

Press OK to close the widget, transfer the script to the **Job Manager** and run the calculation. When you open the phonon bandstructure you will see that the the negative bands are now gone.



The density of states

The `armchair.hdf5` file also contains the phonon density of state (DOS) of the armchair nanoribbon with and without stress relaxation. Below the phonon DOS of the fully (i.e. forces- and stress-) optimized system is shown.



From the phonon DOS it is possible to calculate the free energy, entropy, and zero point energy of the system. In QuantumATK this is implemented in the methods `freeEnergy()`, `entropy()`, and `zeroPointEnergy()` of the PhononDensityOfStates object. The script below shows how to evaluate the entropy:

```
# Read the last configuration in the output file
configuration = nlread('armchair.hdf5', BulkConfiguration)[-1]
# Read the last phonon DOS in the output file
phonon_dos = nlread('armchair.hdf5', PhononDensityOfStates)[-1]

# Define the temperature
t = 300*Kelvin

# Calculate the entropy
s = phonon_dos.entropy(t)

# Print results
print 'Entropy = ', s
```

Execute the script by dropping it on the Job Manager, and in the **Log** you will get the output:

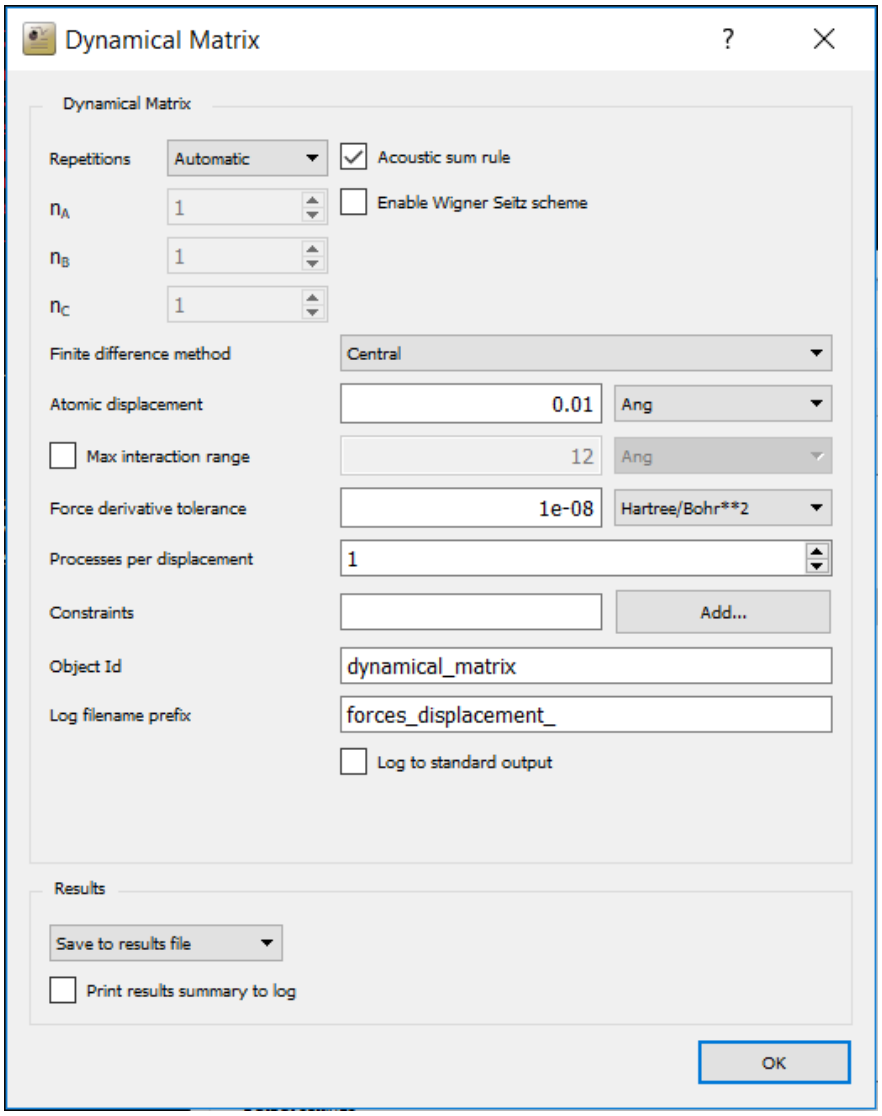
```
Entropy =  4.36888578559 meV/K
```

Algorithmic Details of the Phonon Calculator

The phonons are calculated from the dynamical matrix of the system. The dynamical matrix is the second derivative of the system, corresponding to the first derivative of the forces. The first derivative of the forces are calculated using a finite difference scheme, where the system is displaced along each degree of freedom in the system, also called *frozen phonon* calculations.

When you add a new phonon analysis in the **Script Generator**, it checks if a dynamical matrix is already present. If this is not the case, a **DynamicalMatrix** study object will automatically be added to calculate the dynamical matrix. The **DynamicalMatrix** object of a configuration can be stored in an `.hdf5` file and read again to be used in another calculation.

The details of the calculation of the dynamical matrix is controlled by the parameters in the **DynamicalMatrix** object.



The `Repetitions` parameter allows you to specify how many repetitions of the central cell are employed to calculate forces when the atoms in the central region are displaced. Setting it to `Automatic` will make sure that enough repetitions are used, to fit around each atom in the central region a buffer zone of four times its covalent radius, in which the force derivatives are taken into account. For the current system the **Log** file reports:

```
| Phonon: Automatically detected repetitions = [1 1 5]                    |
```

The size of this buffer region can be changed either by using `Custom` repetitions or by enabling and setting the `Max interaction range` parameter, which replaces the default buffer radius of four times the covalent radius. Furthermore, you can change some other parameters, that determine e.g. whether the acoustic sum rule or symmetries are enforced, as well as numerical parameters, e.g. the finitie difference method or the displacement magnitude of the atoms in the central cell.

Note: In most cases the default values should work well. However, if you are using classical potentials with a long interaction range (e.g. potentials which include coulomb interactions), then it might be necessary to increase the `Max interaction range` to a value somewhat larger than the largest cutoff-radius of the potential. You can normally tell if this is necessary by unchecking `Acoustic sum rule` and inspecting the acoustic bands at the gamma point. If these

bands are not zero, in most cases the maximum interaction range is too low.

K-points used for ATK-DFT and ATK-SE

From QuantumATK-2019.03 onwards, the k-point sampling and density-mesh-cutoff will be **automatically** adapted to the given number of repetitions when setting up the super cell inside *DynamicalMatrix*. That means you can specify the calculator settings for the unit cell and use it with any desired number of repetitions in dynamical matrix calculations.

In previous versions of the code the number of k-points had to be adapted for the dynamical matrix calculation. This is now handled automatically, so the k-points used for phonon calculations do not need to be modified.

Calculating Electrical and Heat Transport for a Graphene Nanoribbon

In this section we will calculate the phonon transmission coefficient for the armchair nanoribbon from which the phonon thermal conductance can be obtained. This will be combined with a calculation of the electron transmission coefficient from which the conductance, Peltier coefficient and electron thermal conductance can be obtained. The different parameters are combined for calculating the thermoelectric figure of merit, ZT. The methodology for the calculations are described in Ref.[1].

Setting up the graphene nanoribbon device

The first step is to build a device geometry from the relaxed armchair nanoribbon. In the **QuantumATK** main window select the last configuration in the `armchair.hdf5` file and drag and drop the configuration onto the **Builder**.

In the builder perform the following operations:

•*Bulk Tools* → *Repeat*: Repeat 5 times in **C**-direction.

•*Device Tools* → *Device From Bulk*: Default values are fine, press `OK` to create the device.

Send the resulting device configuration to the **Script Generator**.

Defining the transmission calculations

In the following you will setup the phonon and electron transmission calculations. For the phonon calculation you will use the Tersoff potential while using a carbon pi tight-binding model for the electron transmission calculation.

•Add a **ForceFieldCalculator**.

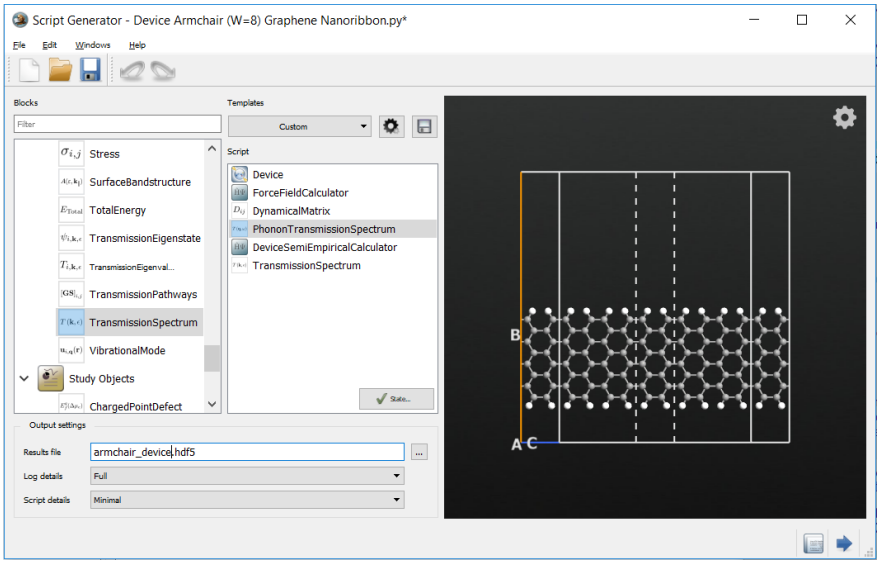
•Add an **Analysis > PhononTransmissionSpectrum**.

•Add a **SemiEmpiricalCalculator**.

•Add an **Analysis > TransmissionSpectrum**.

•Change the output file to `armchair_device.hdf5`.

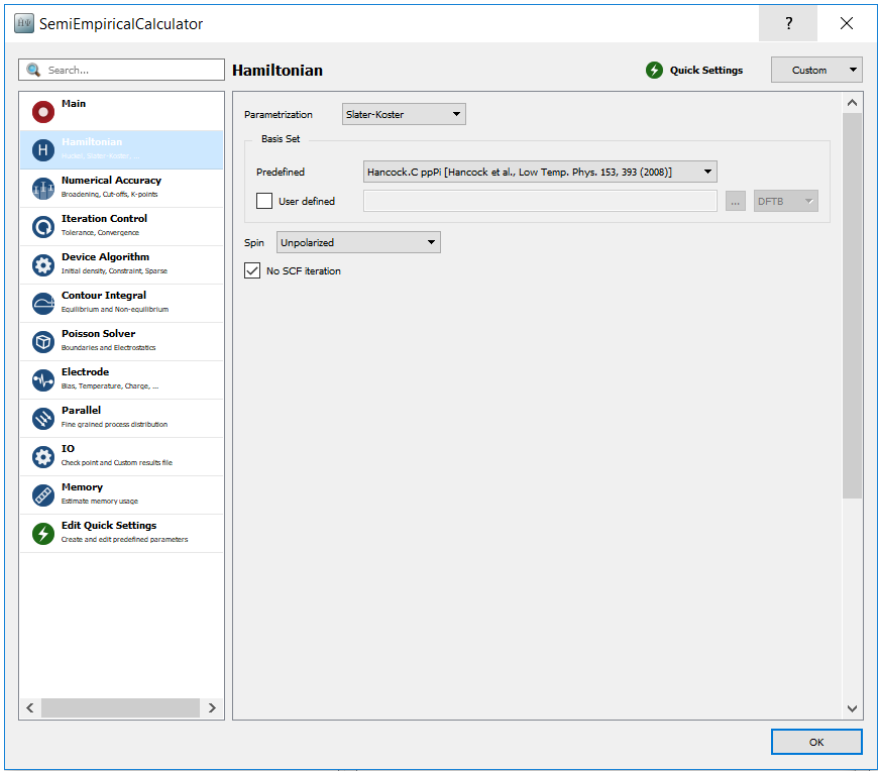
[1] T. Markussen, A.-P. Jauho , M. Brandbyge: Surface-Decorated Silicon Nanowires: A Route to High-ZT Thermoelectrics Phys. Rev. Lett 103, 055502 (2009)



Adjusting script components

To set up calculation parameters, make the following changes:

- Double-click the **ForceFieldCalculator** and select the *Tersoff_C_2010* potential. In the results box select `Do not save` and uncheck the `Print results summary to log` box.
- No changes need to be made to the **PhononTransmissionSpectrum**.
- Double-click the second **SemiEmpiricalCalculator** and select the `Hamiltonian` settings. In the *Basis Set* box select the *Hancock.C ppPi* basis set.

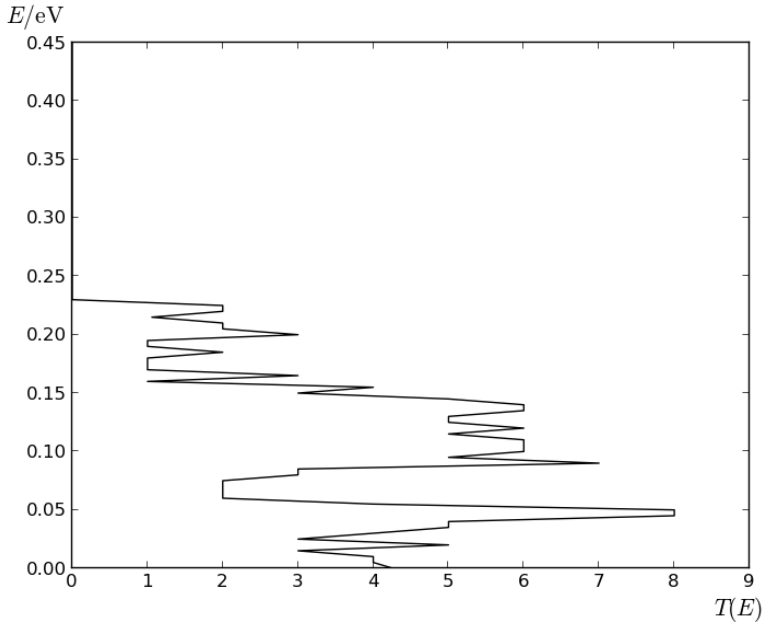


•No changes need to be made to the **TransmissionSpectrum**.

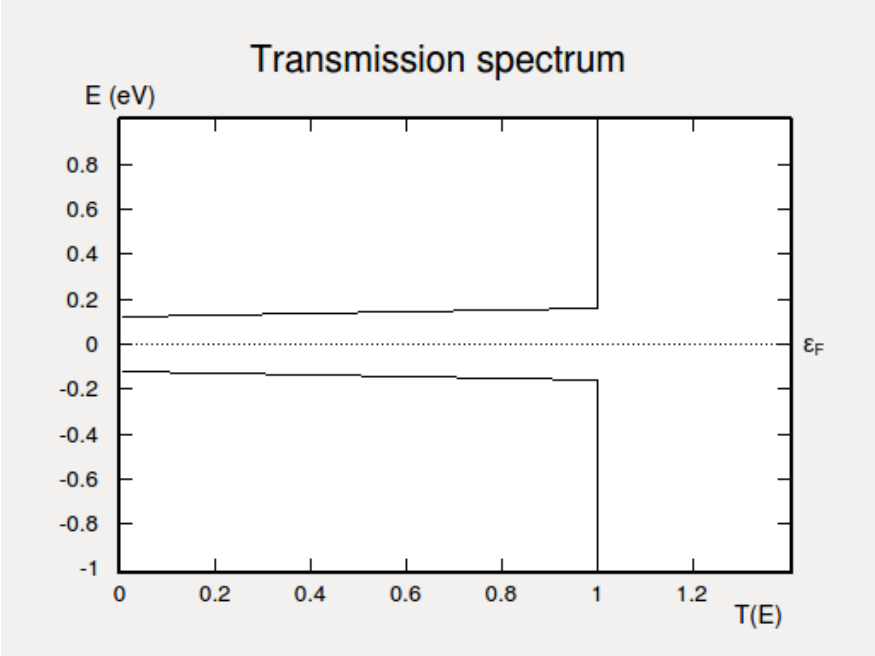
Transfer the script to the **Job Manager** and run the calculation.

Analyzing the results

Now inspect the result file `armchair_device.hdf5` in the **QuantumATK** main window. You may select the *Phonon-TransmissionSpectrum* and the electron *TransmissionSpectrum* and plot them using the **Transmission Analyzer** or the **2D Plot** analysis plugin.



The phonon transmission spectrum of the armchair nanoribbon.



The electron transmission spectrum of the armchair nanoribbon, zoomed into the energy range from -1 to 1 eV.

The thermo-electric figure of merit

In this section you will calculate the linear response transport coefficients of an applied voltage difference (dU) or temperature difference (dT) between the two electrodes. The coefficients are:

The conductance,

$$G_e = \left.\frac{dI}{dU}\right|_{dT=0} .$$

The Peltier coefficient,

$$\Pi = \left.\frac{I_Q}{I}\right|_{dT=0} .$$

The Seebeck coefficient,

$$S = -\left.\frac{dU}{dT}\right|_{I=0} = \frac{\Pi}{U} .$$

The heat transport coeficient of electrons, κ_e, and phonons κ_{ph},

$$\kappa = \left.\frac{dI_Q}{dT}\right|_{I=0} ,$$

where $I_Q = dQ/dT$ is the heat current.

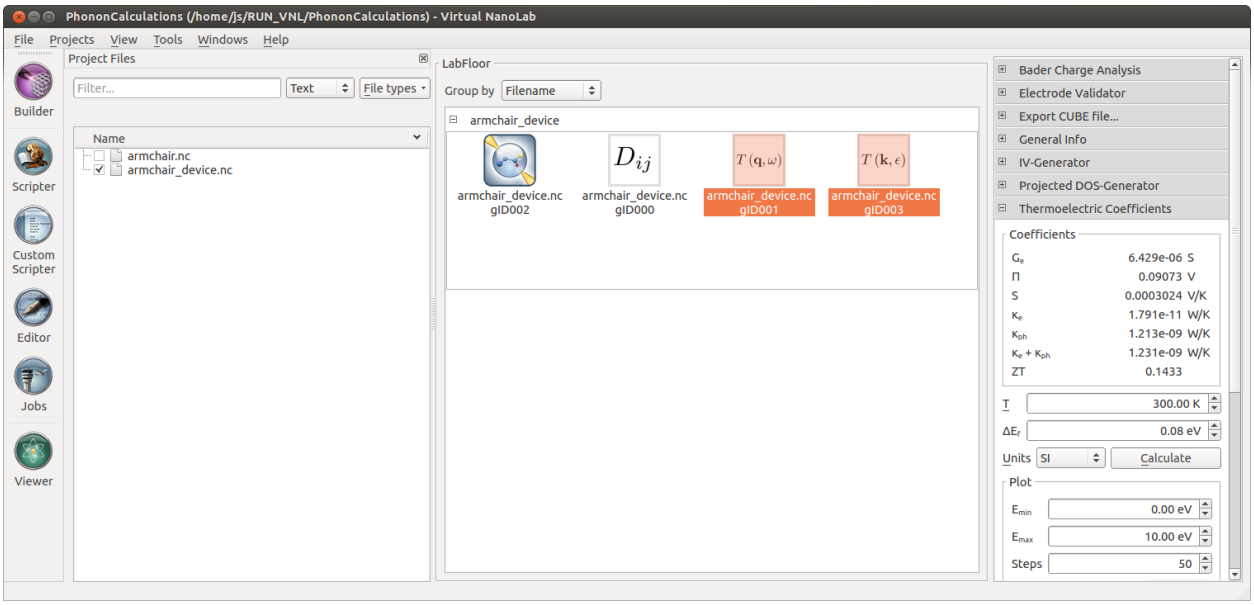
From these coefficients ypu can obtain the thermoelectric figure of merit

$$ZT = \frac{G_e S^2 T}{\kappa} ,$$

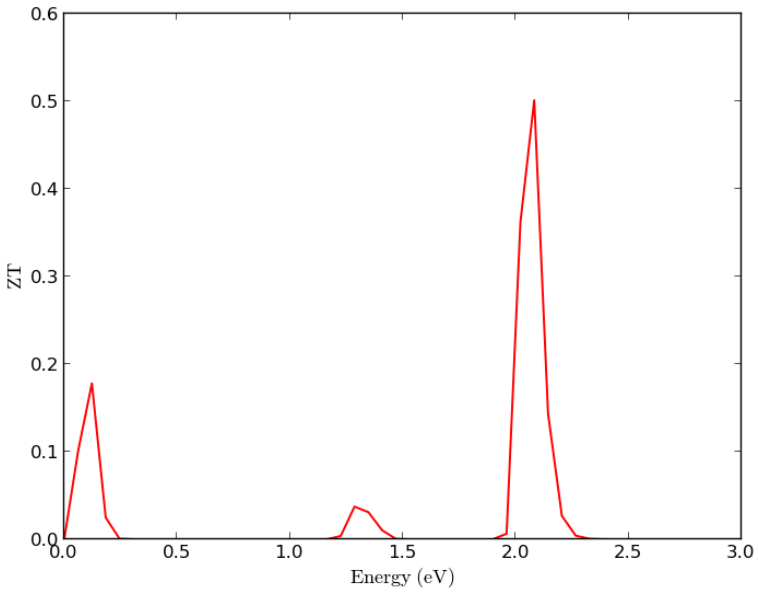
which quantifies how efficient a temperature difference can be converted into a voltage difference in a thermoelectric material.

These linear response coefficients are calculated by the **Thermoelectric Coefficients** plugin. To perform the calculation, select the *PhononTransmissionSpectrum* and the electron *TransmissionSpectrum* objects simultanously and press the `Calculate` button of the plugin.

For an undoped nanoribbon, the Fermi level is in the middle of the band gap. To get a significant Peltier coefficient the Fermi level must be at a band edge, corresponding to a doped nanoribbon. The plugin allows for shifting the Fermi level, and the following shows the result for $\Delta E_F = 0.08eV$, corresponding to an n-doped nanoribbon.



The **Thermoelectric Coefficients** plugin allows you to plot the different thermoelectric coefficients as a function of the energy. For example, the figure below reports the figure of merit ZT at 300K as a function of the energy. The peaks corresponding to the band edges are clearly visible.



References

3.7.4 Inelastic Electron Spectroscopy of an H_2 molecule placed between 1D Au chains

Version: 2017

This tutorial is focused on simulating inelastic electron tunneling spectroscopy (IETS)[1] for a device consisting of two one-dimensional (1D) gold electrodes and an H_2 molecule placed in between. After DFT calculations of the dynamical matrix and the Hamiltonian derivatives with respect to vibrations of the H_2 molecule, IETS is computed and analyzed based on the lowest-order expansion (LOE)[2] and extended LOE (XLOE)[3] approximations.

Note: It is assumed that you are familiar with QuantumATK. If not, go through the basic QuantumATK Tutorials first.

Introduction

We consider electron transport through a device consisting of an H_2 molecule clamped between 1D gold electrodes. In a spectroscopy experiment, if the applied bias is comparable to the energy of one of the vibrational modes of the H_2 molecule, a new inelastic transmission channel opens in which the transmission of electrons is mediated by the emission of a phonon. The coupling process is due to the coupling between electronic and vibrational degrees of freedom, that is, to the electron-phonon coupling. At a voltage where an inelastic channel opens, the derivative of the conductance, i.e., the second-derivative of the current has a peak. In experiments, IETS is defined by

$$\mathrm{IETS} \equiv \frac{dI^2/dV^2}{dI/dV},$$

and is measured to investigate, e.g., the electron-phonon coupling strength, heating, configuration, and local doping.

In the tutorial, you will first compute the dynamical matrix and the Hamiltonian derivatives with respect to vibrations of the H_2 molecule by DFT calculations, and then calculate and analyze the IETS signal based on two different approximations: LOE and XLOE. The rest of this tutorial is hence organized as follows:

1. Device setup,
2. Calculation of the dynamical matrix and the Hamiltonian derivatives,
3. Calculation of IETS based on LOE and XLOE,
4. Analysis of IETS based on the vibrational modes of the H_2 molecule,
5. Comparison between the LOE and XLOE results.

[1] M.ăA. Reed. Inelastic Electron Tunnelling Spectroscopy. *Materials Today*, 2008. doi:10.1016/S1369-7021(08)70238-4.

[2] T.ăFrederiksen, M.ăPaulsson, M.ăBrandbyge, and A.-P. Jauho. Inelastic transport theory from first principles: Methodology and application to nanoscale devices. *Phys. Rev. B*, 75:205413, May 2007. doi:10.1103/PhysRevB.75.205413.

[3] J.-T. Lu, R.ăB. Christensen, G.ăFoti, T.ăFrederiksen, T.ăGunst, and M.ăBrandbyge. Efficient calculation of inelastic vibration signals in electron transport: Beyond the wide-band approximation. *Phys. Rev. B*, 89:081405, Feb 2014. doi:10.1103/PhysRevB.89.081405.

Device setup

Start up **QuantumATK** and create a new project. Instead of using **Builder** to create the device configuration, we use the QuantumATK Python script `au-h2-au_0.py`, which provides the Au|H_2|Au device. Download and save the script in the project folder, and open it from **Script Generator**.

•Add a **New Calculator** block and select GGA as the exchange correlation functional in the setting.

Note: The k-point sampling is by default 1x1x93, which should, however, be modified properly if the device is periodic in the A and/or B directions.

•Set the output-file name *au-h2-au.nc*, and save the python script as *au-h2-au.py*.

Note: The python script must now be the same as `au-h2-au.py`. To save time, one may just download the script and use it.

•Send the script to **Job Manager** and run it.

Calculation of IETS

The IETS calculation requires information of the dynamical matrix, which is defined in terms of the force for displacing the hydrogen atoms back and forth in each spatial direction. The computation involves six DFT calculations, which are all independent with each other, and is hence performed efficiently by using six CPU cores.

•In the **Script Generator**, click on *Analysis* from File and select *au-h2-au.nc*.

•Add a *Analysis* → *DynamicalMatrix* object and set it as follows:

•Add also a *Analysis* → *VibrationalModes* object and keep the default settings.

The IETS calculation requires information of the Hamiltonian derivatives as well. This calculation also consists of six independent parts, and is hence carried out efficiently using six CPU cores.

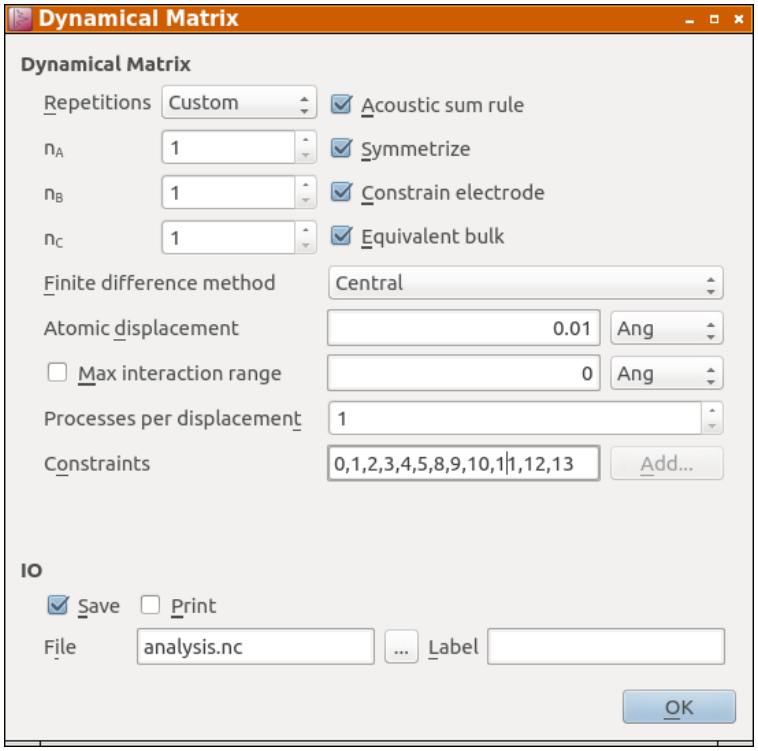
•Add a *Analysis* → *HamiltonianDerivatives* block and set it as follows:

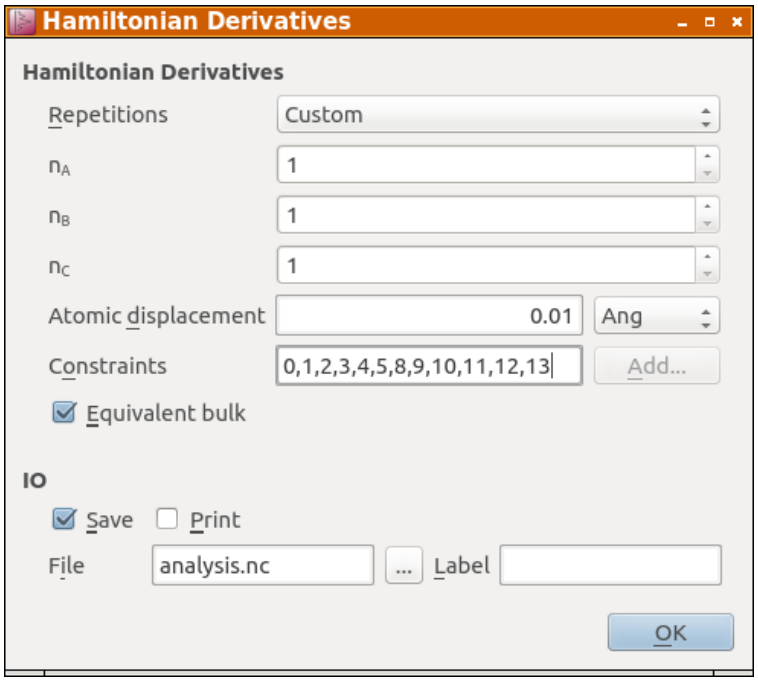
•Add an *Analysis* → *InelasticTransmissionSpectrum*, choose LOE as the method, and set the rest as follows:

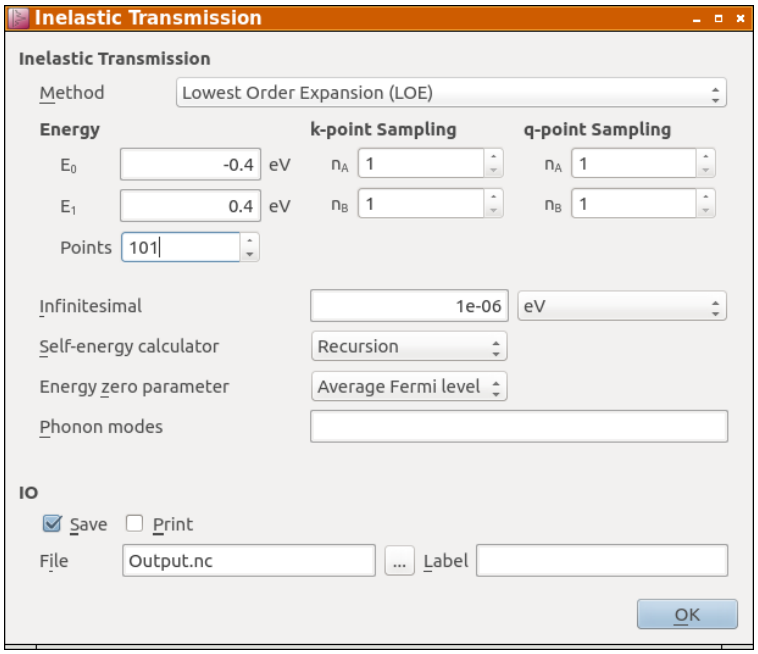
•Add another *Analysis* → *TransmissionEigenstates* block, and set it in the same way except choosing XLOE as the method.

•Set the output-file name *analysis.nc*, and save the python script as *analysis.py*.

Note: The python script must be the same as `analysis.py`. To save time, one may just download the script and use it.


Dynamical Matrix
Dynamical Matrix
Repetitions
Custom
Acoustic sum rule
n_A
1
Symmetrize
n_B
1
Constrain electrode
n_C
1
Equivalent bulk
Finite difference method
Central
Atomic displacement
0.01
Ang
Max interaction range
0
Ang
Processes per displacement
1
Constraints
0,1,2,3,4,5,8,9,10,11,12,13
Add...
IO
Save
Print
File
analysis.nc
...
Label
OK


Hamiltonian Derivatives
Hamiltonian Derivatives
Repetitions
Custom
n_A
1
n_B
1
n_C
1
Atomic displacement
0.01
Ang
Constraints
0,1,2,3,4,5,8,9,10,11,12,13
Add...
Equivalent bulk
IO
Save
Print
File
analysis.nc
...
Label
OK



•Send the script to **Job Manager** and run it.

The calculation will take less than one hour if six CPU cores are used, and save every result in *analysis.nc*.

Note: See *InelasticTransmissionSpectrum* in QuantumATK Manual, which provides a brief description about IETS and how it is computed from the dynamical matrix and the Hamiltonian derivatives. See also, e.g., Ref.[Page 1130, 2] for more details.

Analysis

Vibrational modes of the H_2 molecule

In the Labfloor, look at the vibrational modes of the H_2 molecule by selecting the $\mathbf{u}_{i,\mathbf{q}}(\mathbf{r})$ Vibrational Mode object in the *analysis.nc* file with the help of **Vibration Visualizer** plugin.

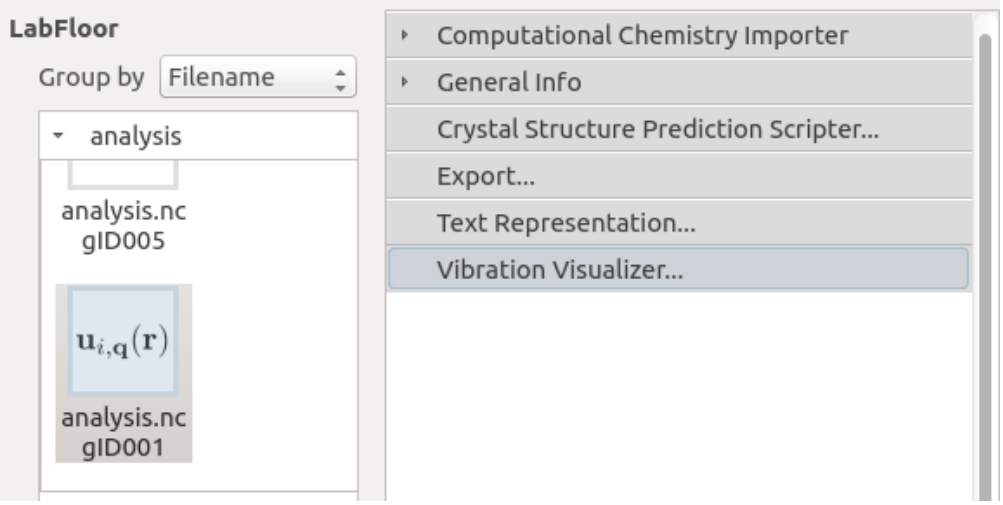
There are, in total, six modes: Four transverse modes 0, 1, 2, 3; two longitudinal modes 4 and 5. The two pairs of modes (0 and 1) and (2 and 3) are, respectively, doubly degenerated. Note that the temperature is set $T = 1000\ K$ in the movies below to see the vibrations clearly.

Mode 0: 25.44 meV; Mode 1: 25.63 meV (the small energy difference is due to numerical error)

Mode 2: 60.54 meV; Mode 3: 60.56 meV (the small energy difference is due to numerical error)

Mode 4: 127.46 meV

Mode 5: 264.06 meV



IETS

Then look at IETS. Select one of the $T(\mathbf{k},\mathbf{q})$ InelasticTransmissionSpectrum objects $T(\mathbf{k},\mathbf{q})$ in LabFloor and click on the **Inelastic Transmission Spectrum Analyzer** plugin, a new plugin available from **ATK2017**.

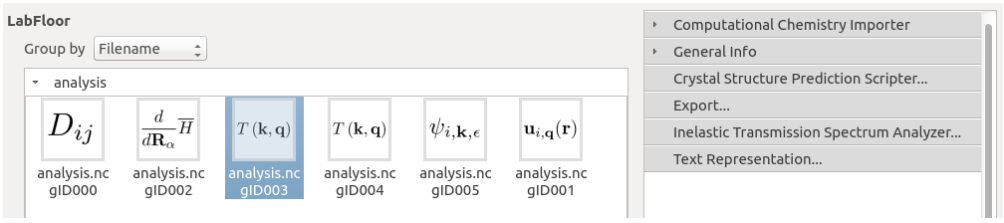
Set the panel as follows and click **Show**.

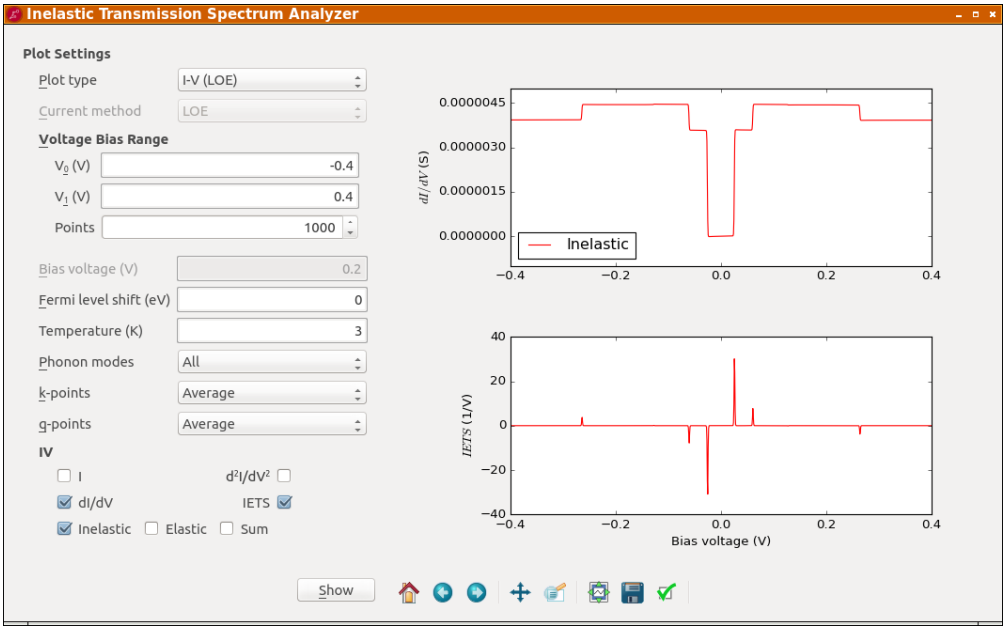
> **Attention:** Because the density of states (DOS) of the gold electrodes is almost flat within this bias window, there only exists a minor difference between the LOE and XLOE results. See the discussion below for more details.

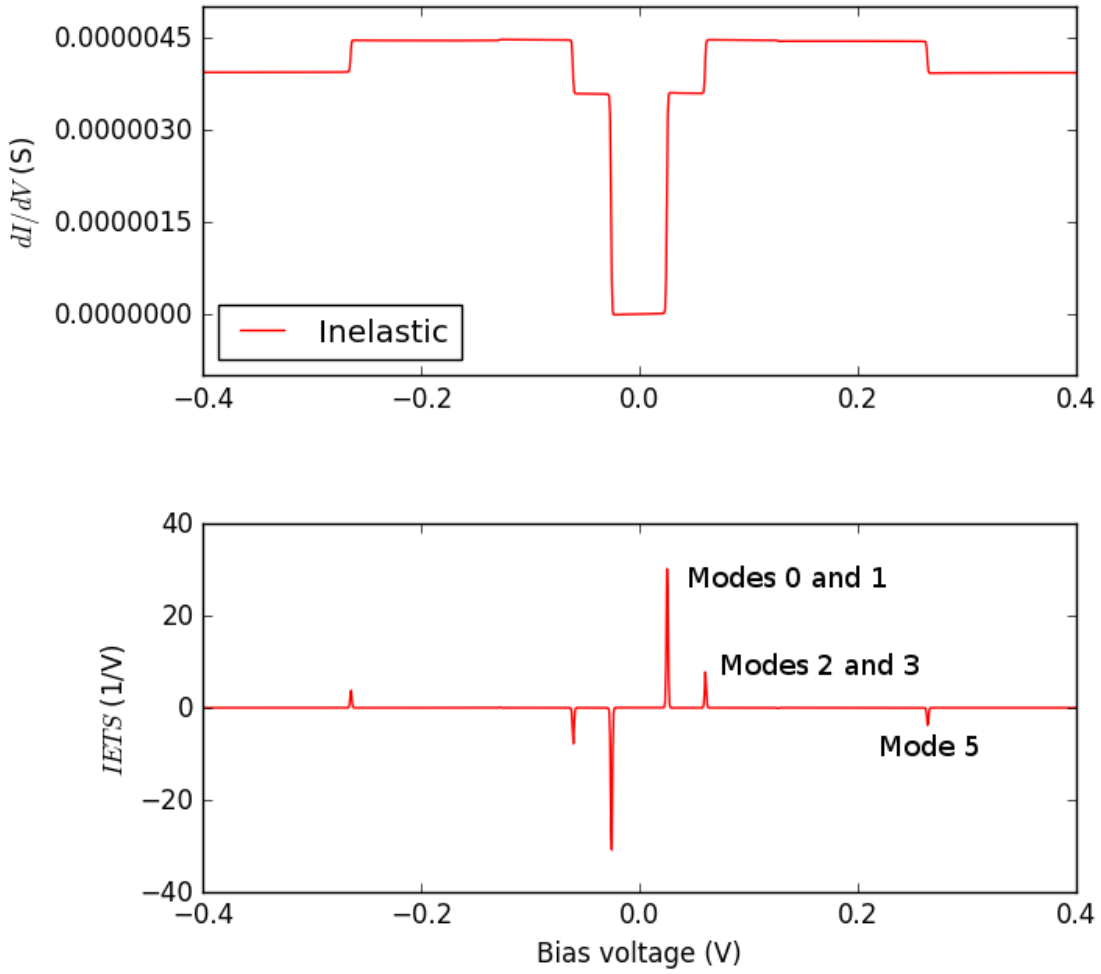
There appear three significant inelastic features (steps in conductance and peaks in IETS). Note that the inelastic conductance is increased by the coupling to the low-frequency transverse modes 0-3 and decreased by the coupling to the high-frequency longitudinal mode 5. Analyzing the scattering states clarifies the reason of the increases.

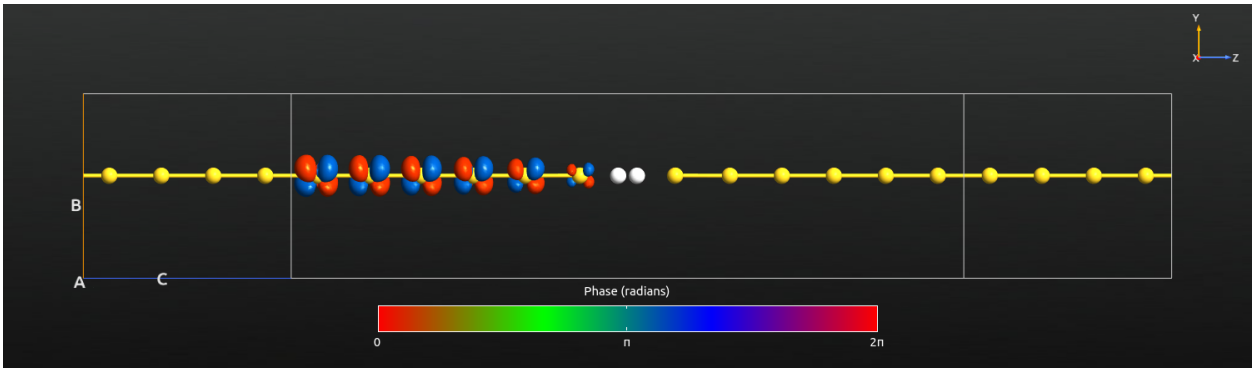
- Drag the $\psi_{i,\mathbf{k},\epsilon}$ TransmissionEigenstates object $\psi_{i,\mathbf{k},\epsilon}$ in LabFloor into **Viewer**. Then drag the DeviceConfiguration into **Viewer**.
- Go to properties and, under the Isosurface flag, set the Isovalue to 0.5.

The scattering eigenstate has a clear *d*-symmetry on the gold chain. Hence, it couples only very weakly to the *s*-like states of the hydrogen atoms. The elastic transmission is thus small. The coupling to the transverse modes 0-3 can, however, open new inelastic channels and leads to positive steps in conductance and steps in IETS.


LabFloor
Group by
Filename
analysis
analysis.nc gID000
analysis.nc gID002
analysis.nc gID003
analysis.nc gID004
analysis.nc gID005
analysis.nc gID001
Computational Chemistry Importer
General Info
Crystal Structure Prediction Scripter...
Export...
Inelastic Transmission Spectrum Analyzer...
Text Representation...

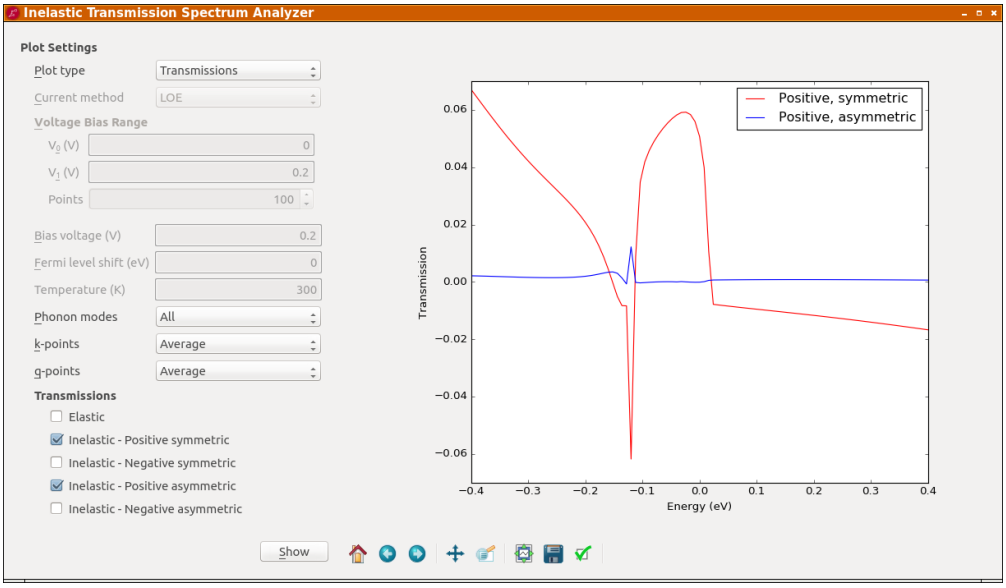

Inelastic Transmission Spectrum Analyzer
Plot Settings
Plot type
I-V (LOE)
Current method
LOE
Voltage Bias Range
Points
1000
Bias voltage (V)
0.2
Fermi level shift (eV)
Temperature (K)
Phonon modes
All
k-points
Average
q-points
Average
IV
dI/dV
Inelastic
Elastic
Sum
IETS
Show
Inelastic
Bias voltage (V)


0.0000045
0.0000030
0.0000015
0.0000000
dI/dV (S)
Inelastic
−0.4
−0.2
0.0
0.2
0.4
40
20
0
−20
−40
$IETS$ (1/V)
Modes 0 and 1
Modes 2 and 3
Mode 5
Bias voltage (V)


B
A
C
Phase (radians)
0
π
2π

Detailed analysis of transmission: Comparison between LOE and XLOE

In the same window of the **Inelastic Transmission Spectrum Analyzer** plugin, now choose **Transmissions** in Plot type, set the options as shown below, and click **Show**. In this case, one will look at the positive symmetric and asymmetric parts of the inelastic transmission:



To understand some details of the transmission, note that, in either LOE or XLOE, the current as a function of the bias voltage V and the temperature T reads

$$I(V,T) = \sum_{\lambda} \left[\mathcal{I}_{\lambda}^{sym}(V,T)\mathcal{T}_{\lambda}^{sym}(\epsilon_F) + \mathcal{I}_{\lambda}^{asym}(V,T)\mathcal{T}_{\lambda}^{asym}(\epsilon_F) \right],$$

where ϵ_F is the Fermi energy, and the rest of the new symbols are defined as follows: In either approximation, the universal functions appear in the same form,

$$\mathcal{I}_{\lambda}^{sym}(V,T) = \frac{G_0}{2e} \sum_{\sigma=\pm} \sigma(\hbar\omega_\lambda + \sigma eV) \left(\coth \frac{\hbar\omega_\lambda}{2k_BT} - \coth \frac{\hbar\omega_\lambda + \sigma eV}{2k_BT} \right),$$

$$\mathcal{I}_{\lambda}^{asym}(V,T) = \frac{G_0}{2e} \int_{-\infty}^{\infty} d\epsilon \left[n_F(\epsilon - eV) - n_F(\epsilon) \right] \cdot \mathcal{H}_{\epsilon'} \left[n_F(\epsilon' - \omega_\lambda) - n_F(\epsilon' + \omega_\lambda) \right](\epsilon),$$

where $G_0 = 2e^2/h$ is the conductance quantum, $n_F(\cdot)$ is the Fermi-Dirac distribution function, and $\mathcal{H}_{\epsilon'}[\cdot](\epsilon)$ means the Hilbert transform. In XLOE, the transmission functions are given by

$$\mathcal{T}_{\lambda}^{sym}(\epsilon) = \mathrm{Tr}\left[M_\lambda \tilde{A}_L(\mu_L) M_\lambda A_R(\mu_R) \right] + \mathrm{Im}\, B_\lambda(\epsilon),$$

$$\mathcal{T}_{\lambda}^{asym}(\epsilon) = 2\mathrm{Re}\, B_\lambda(\epsilon),$$

with

$$\mu_L = \epsilon,$$

$$\mu_R = \epsilon \pm \hbar\omega_\lambda,$$

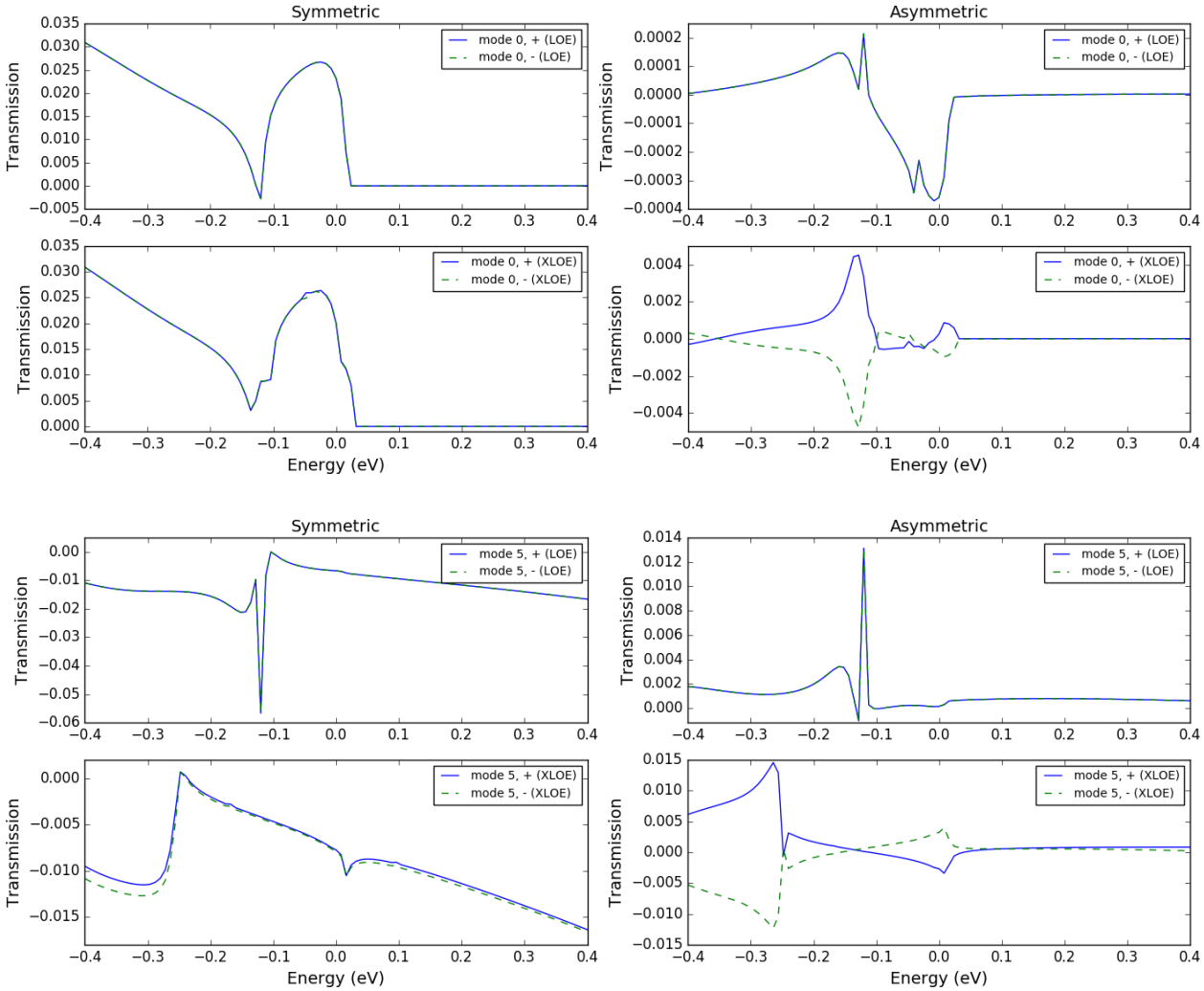
$$\tilde{A}_L(\epsilon) = G^a(\epsilon)\Gamma_L(\epsilon)G^r(\epsilon),$$

and

$$B_\lambda(\epsilon) = \mathrm{Tr}\big[M_\lambda A_R(\mu_L)\Gamma_L(\mu_L)G^r(\mu_L)M_\lambda A_R(\mu_R) - M_\lambda G^a(\mu_R)\Gamma_L(\mu_R)A_R(\mu_R)M_\lambda A_L(\mu_L)\big].$$

This XLOE expression of the transmission functions reduces to the LOE expression by setting $\mu_L = \mu_R = \epsilon$. Importantly, in the expression of the current, note that the Greens function, the self-energies, and the spectral functions in the transmission functions are evaluated at two different energies $\epsilon_F \pm \hbar\omega_\lambda$ in XLOE, whereas they are all evaluated at the same energy ϵ_F in LOE. See Ref.[Page 1130, 3] for more details.

For direct comparison between the LOE and XLOE results, see here below the inelastic transmissions for each phonon mode. The lines of positive/negative (a)symmetric transmissions indicate the plots of $\mathcal{T}_\lambda^{(a)sym}(\epsilon)$ as functions ϵ with setting $\mu_R = \epsilon \pm \hbar\omega_\lambda$. The positive and negative (a)symmetric parts are exactly the same in LOE by definition, but different in XLOE.



The LOE and XLOE results are in good agreement in areas where the transmission varies slowly as a function of ϵ. Around -0.11 eV, however, the wide-band approximation in LOE, i.e., the approximation $\mu_L = \mu_R = \epsilon$, breaks down due to the presence of a resonance in the DOS of the 1D gold chain. As a result, there appears a difference between the LOE and XLOE results. The symmetric and asymmetric inelastic transmissions for mode 5 show the difference very clearly. The single peak at -0.11 eV in LOE splits into two peaks in XLOE; the separation is about 264 meV, which corresponds to the phonon energy of mode 5.

References

3.7.5 Interfacial thermal conductance

Version: 2016.1

The continuous downsizing of modern electronic devices implies the increasing complexity of a nano-device. The thermal conductivity is hard to predict when the device is formed by more than one material. More often, device engineers want to maximize or minimize the thermal conductivity by constructing interfaces between different materials.

In this tutorial, you will learn how to use molecular dynamics with a classical potential to simulate the heat flow through an interface and calculate the interfacial thermal resistance.

Introduction

You will simulate the heat current through a model grain boundary in a silicon crystal by imposing a temperature gradient between a heat source and a heat sink.

Theory

With a given temperature gradient, the bulk heat conductivity κ_{bulk} of a material is related to the heat flux via the Fourier equation,

$$\frac{\langle dQ/dt \rangle}{A} = -\kappa_{bulk} \langle dT/dz \rangle,$$

where $\langle dQ/dt \rangle$ represents the average heat flux (i.e. transferred kinetic energy per time) along the gradient, and A is the area perpendicular to the direction of the heat flux.

The presence of an interface between the temperature source and sink often causes a discontinuity (a temperature jump) ΔT in the temperature profile, since the interface provides additional thermal resistance R to the heat current. In this case, the interfacial thermal conductance G, often referred to as Kapitza conductance, can be defined as

$$G = \frac{1}{R} = \frac{\langle dQ/dt \rangle}{\Delta T}$$

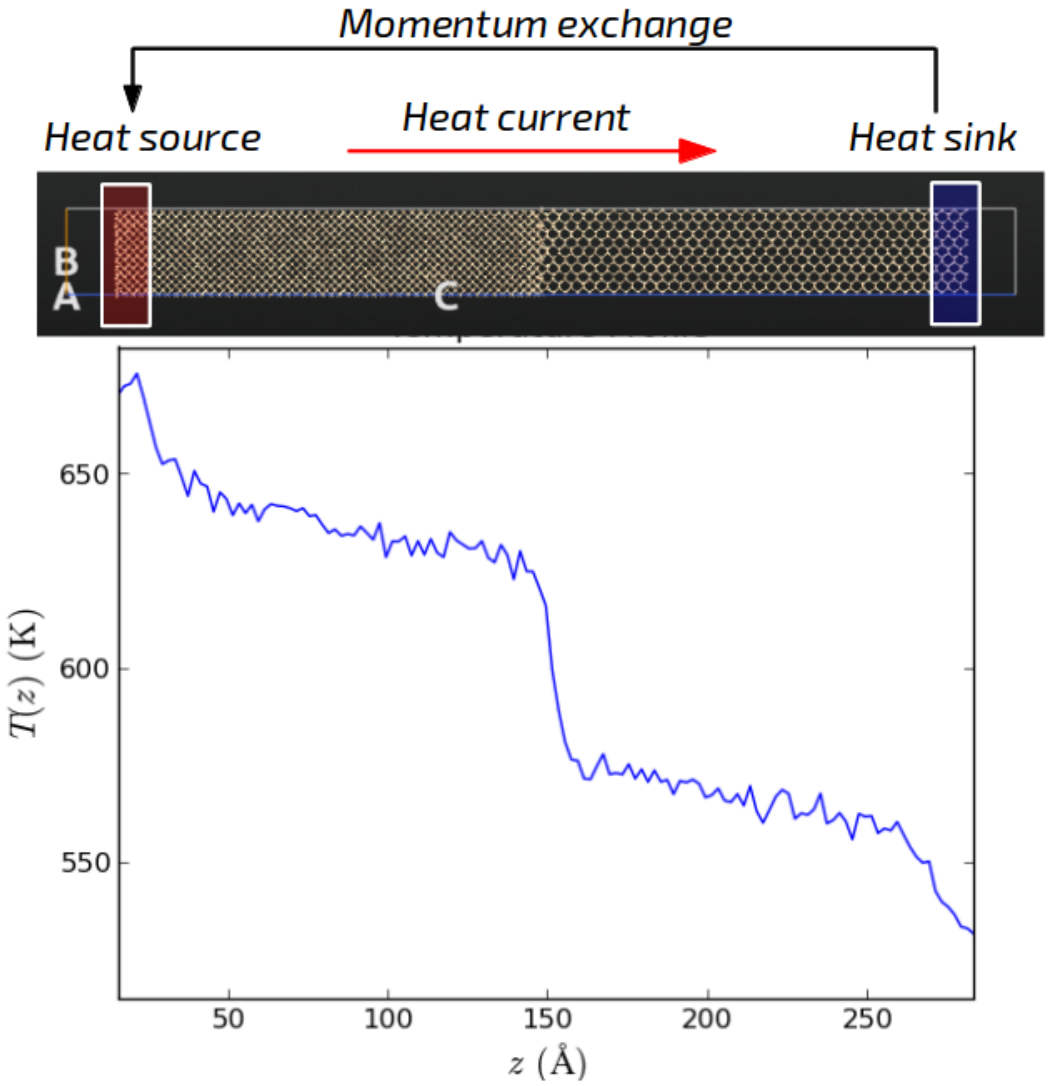
Note: Sometimes it makes sense to normalize G to the cross-sectional area, e.g. if you want to compare similar interfaces with different cross sectional areas.

Method

In this tutorial, we will use two different methods to calculate the heat conductivity. In *the first section*, we will use molecular dynamics simulations, and in *the second section* we will use the non-equilibrium Greens function method.

Reverse non-equilibrium molecular dynamics (RNEMD)

When using molecular dynamics (MD) simulations, the most common technique to impose a temperature gradient in MD simulations is the so-called non-equilibrium momentum exchange or reverse non-equilibrium molecular dynamics (RNEMD) method[1][2]. In this method two groups of atoms are selected as the heat source and the sink, respectively. During the simulation the momenta of the hottest atom in the sink and the coldest atom in the source are exchanged according to special criteria to conserve the total energy and momentum[Page 1140, 2]. This external heat flux is balanced by a heat current from source to sink within the system, causing a measurable temperature gradient (and a temperature jump if an interface is present).



[1] F.ăMueller-Plathe. A simple nonequilibrium molecular dynamics method for calculating the thermal conductivity. *J. Chem. Phys.*, 106(14):6082–6085, 1997. doi:10.1063/1.473271.

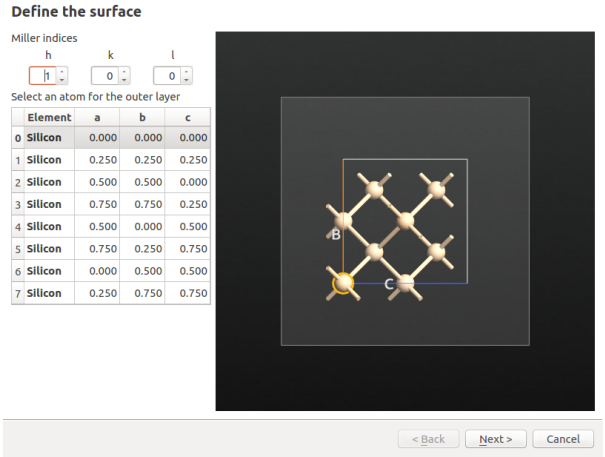
[2] C.ăNieto-Draghi and J.ăB. Avalos. Non-equilibrium momentum exchange algorithm for molecular dynamics simulation of heat flow in multi-component systems. *Mol. Phys.*, 101(14):2303–2307, 2003. doi:10.1080/0026897031000154338.

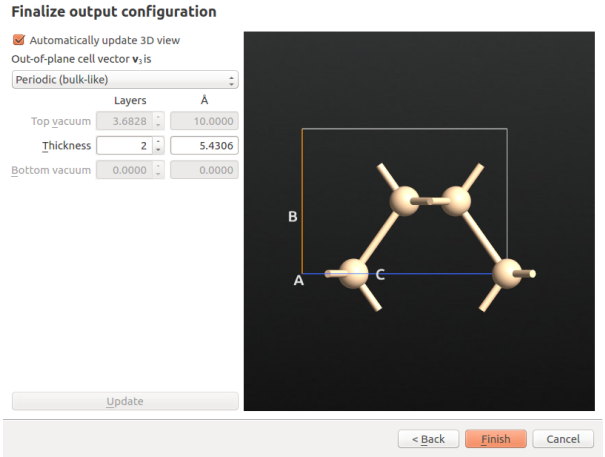
Preparing the grain boundary

You will consider a model grain boundary between two silicon crystal grains with different orientations. Although you can direct the thermal current in any arbitrary direction during the simulation, we will preserve the convention that current runs along the z-direction. Therefore, while building the system you should always make sure that the interface configuration is oriented correctly. If necessary, you should rotate the system to make sure the thermal current is oriented along the z-axis.

You will create an interface between a (100) and a (110) silicon surface. Open the **Builder** , and add Silicon (alpha) via *Add* → *From Database*, and

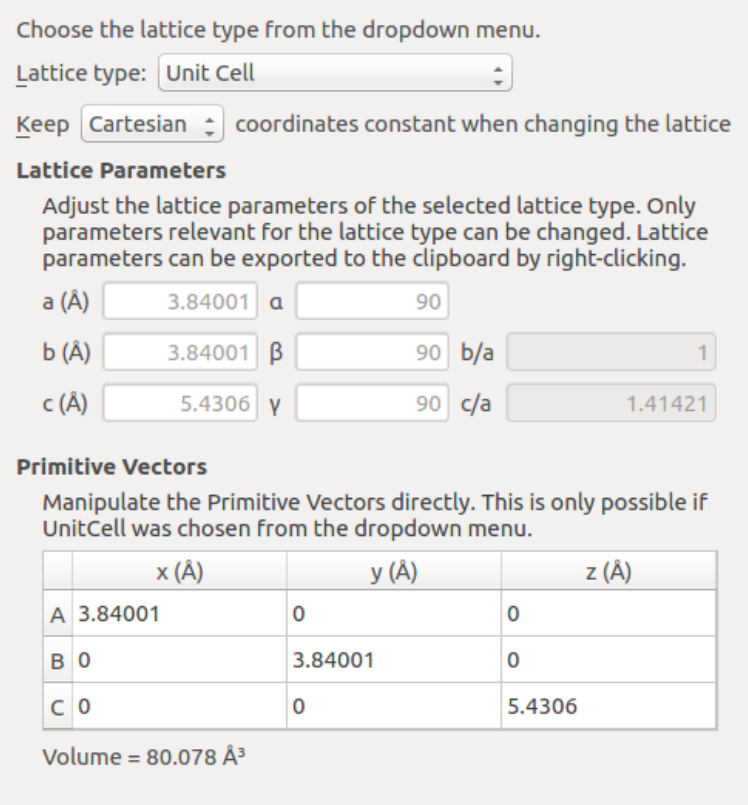
- To generate a (100) surface base cell make sure the newly added Silicon alpha configuration is selected and,
  - Open the *Surface (Cleave) tool* in the right-hand-panel under **Builders**.
  - Select the (100) Miller indices, click Next, leave the 1x1 surface lattice, click Next, and select Periodic (bulk-like).
  - Click Finish to complete the surface creation.





- Repeat the above steps for the (110) structure.

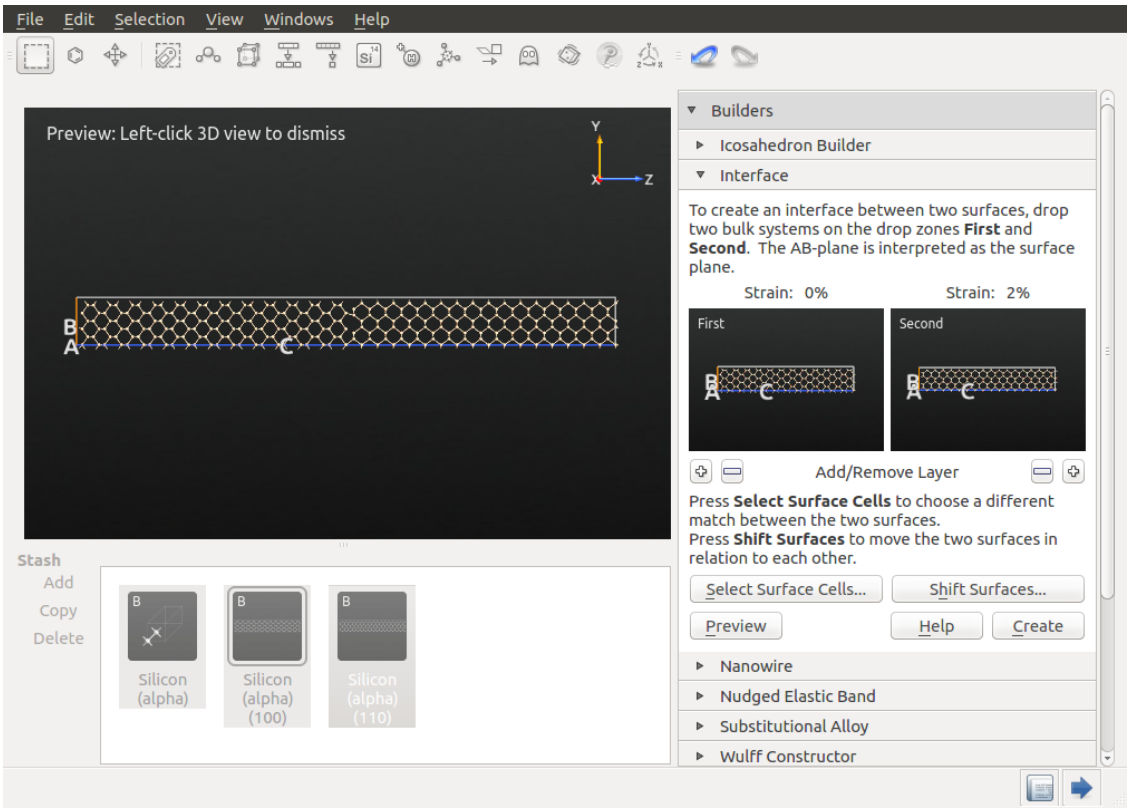
To avoid a large strain when joining the two cells with different lattice constants, you should choose suitable repetitions to obtain commensurate cell sizes. The cell sizes can be looked up and modified under *Bulk tools* → *Lattice parameters*.



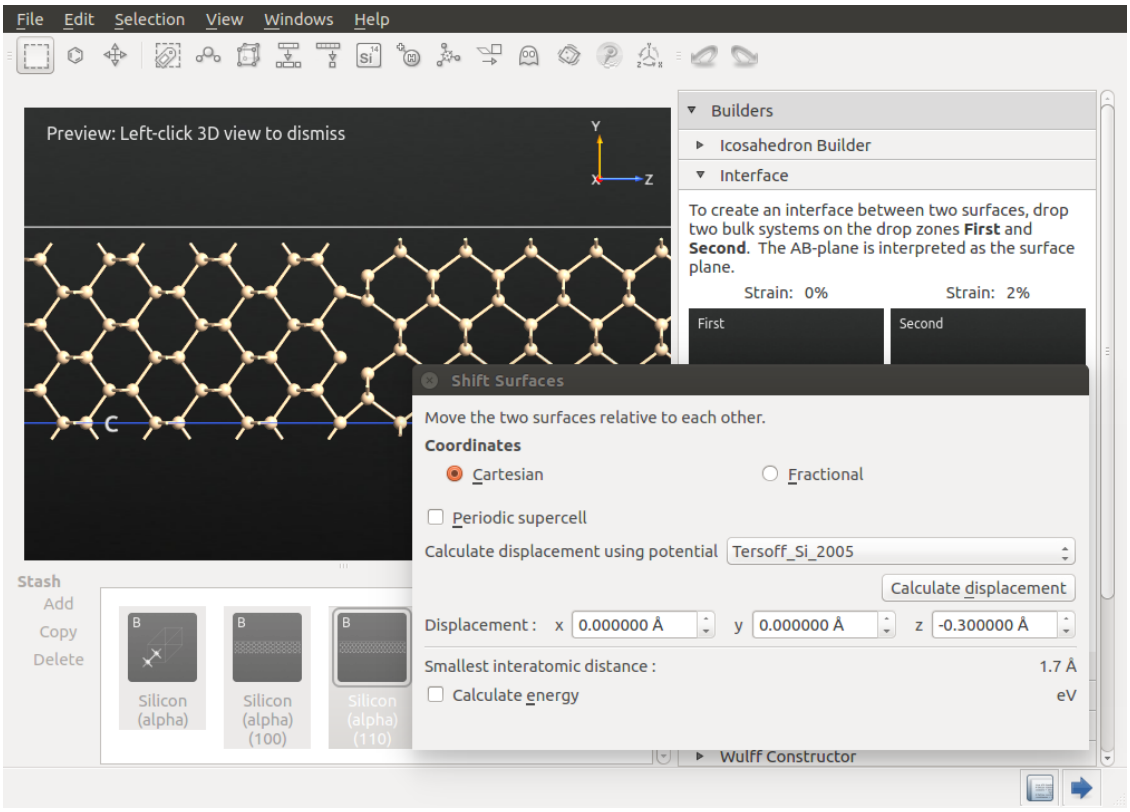
The cell of the (100) surface has dimensions 3.84 x 3.84 x 5.43 $Å^3$, while the (110) cell has a size of 3.84 x 5.43 x 3.84 $Å^3$. Therefore, a 3 x 3 x 12 repetition for the (100) cell, and a 3 x 2 x 17 repetition for the (110) cell, provides supercells of almost equivalent sizes. You can apply such repetitions using the *Bulk tools* → *Repeat plugin*. Set the desired number of repeats and click *Apply*.

Note: Generally, for such non-equilibrium thermal current simulations it is important to choose a sufficient cell length in the direction of the heat flow (in this case the C-vector), whereas the results are less sensitive to the geometry of the cross-sectional area (i.e. A x B). Therefore, the simulation cells employed typically have a stick-like shape.

Creating an interface with two structures could be a tedious task, but the **Interface Builder** helps you to complete it in a simple and flexible manner. Open *Builders* → *Interface* in the right-hand-panel and drag and drop the (100) supercell onto the left field and the (110) supercell onto the right field.



The preview of the interface should appear automatically (if not click Preview). If you zoom into the interface, you will notice a gap between the two surfaces. You can adjust this gap by clicking *Shift Surfaces* and adding a negative offset in the z-direction until bonds between the two surfaces are displayed. Moreover, you can align the two surfaces in the x and y-directions, as well. The proposed values should provide a good initial structure. Close the *Shift Surfaces* tool and click *Create* to create the interface structure.

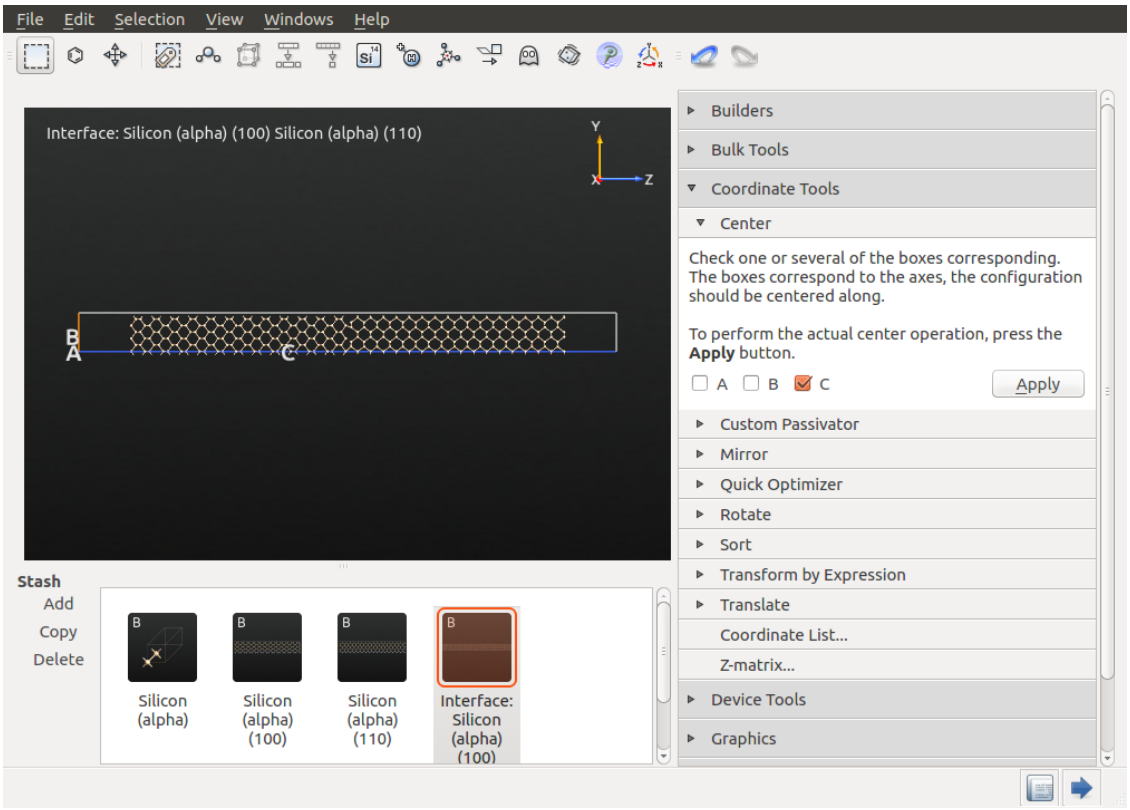


Note: The displayed bonds are visual guides only and do not have a strict physical meaning. For the interface creation the appearance of bonds simply indicates if the initial guess is reasonable. The interface will be optimized, annealed, and equilibrated using molecular dynamics to reduce spurious features to the largest extent.

The supercell in this calculation should be non-periodic in the C-direction, meaning that the system is terminated by vacuum at either end. The lateral dimensions (A and B) will remain periodic.

To obtain such a setup, simply increase the length of the C-vector. This can be done by opening *Bulk Tools* → *Lattice Parameters* and increasing the C_z entry from about 130 Å to 160 Å. Make sure that you keep the Cartesian coordinates.

Finally, the slab should be centered in the middle of the supercell. Therefore, open *Coordinate Tools* → *Center*, uncheck A and B and click *Apply*. The result should look like below.



Note: An alternative to this surface-terminated setup is to simulate a fully periodic system, where the heat flow is directed into both the positive and negative z-axes away from the heat source[3][4]. This setup requires two identical interfaces and a simulation cell which is twice as long.

Setting up the equilibration simulation

Prior to simulating the actual heat flow, the system has to be equilibrated properly in order to reduce artifical defects due to the interface generation, remove the lateral stress, and prepare a system which is equilibrated at the desired target temperature. To do this, send the final interface to the **Script Generator** by clicking the Send To icon, and add 3 blocks:

- New Calculator
- *Optimize* → *OptimizeGeometry*
- *Optimize* → *MolecularDynamics*

Change the Default output file name to `Silicon_grain_boundary.nc`, and open the **New Calculator** block.

[3] P.ăC. Howell. Comparison of molecular dynamics methods and interatomic potentials for calculating the thermal conductivity of silicon. *The Journal of Chemical Physics*, 137(22):, 2012. doi:10.1063/1.4767516.

[4] Akbar Bagri, Sang-Pil Kim, RodneyăS. Ruoff, and VivekăB. Shenoy. Thermal transport across Twin Grain Boundaries in Polycrystalline Graphene from Nonequilibrium Molecular Dynamics Simulations. *Nano Letters*, 11(9):3917–3921, 2011. doi:10.1021/nl202118d.

•Select *ATK-ForceField* and the *StillingerWeber_Si_1985 potential*.

•Uncheck *Save and Print* and close the widget by clicking OK.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

Open the **OptimizeGeometry** block and set the Force tolerance to 0.1 eV / Å. The remaining parameters can be left at their default values. Uncheck *Save and Print* and close the widget.

Note: The optimization is necessary to remove initial large destabilizing forces that might occur after the interface generation. As an interfaced is introduced, a small force is most likely unavoidable.

The next step is to allow the lateral cell vectors to relax and equilibrate to the target temperature using the Molecular Dynamics method. The entire thermal transport simulation will be performed at an average temperature of 600 K. To do this, open the **MolecularDynamics** block to set up the parameter for the NPT simulation, i.e. a simulation at constant temperature and pressure.

•Select the Type as NPT Martyna-Tobias-Klein.

•Set the number of steps to 50 000 and the log interval to 1000.

•Choose a suitable trajectory name, e.g. `traj_grain_boundary_equilibrate_NPT.nc`.

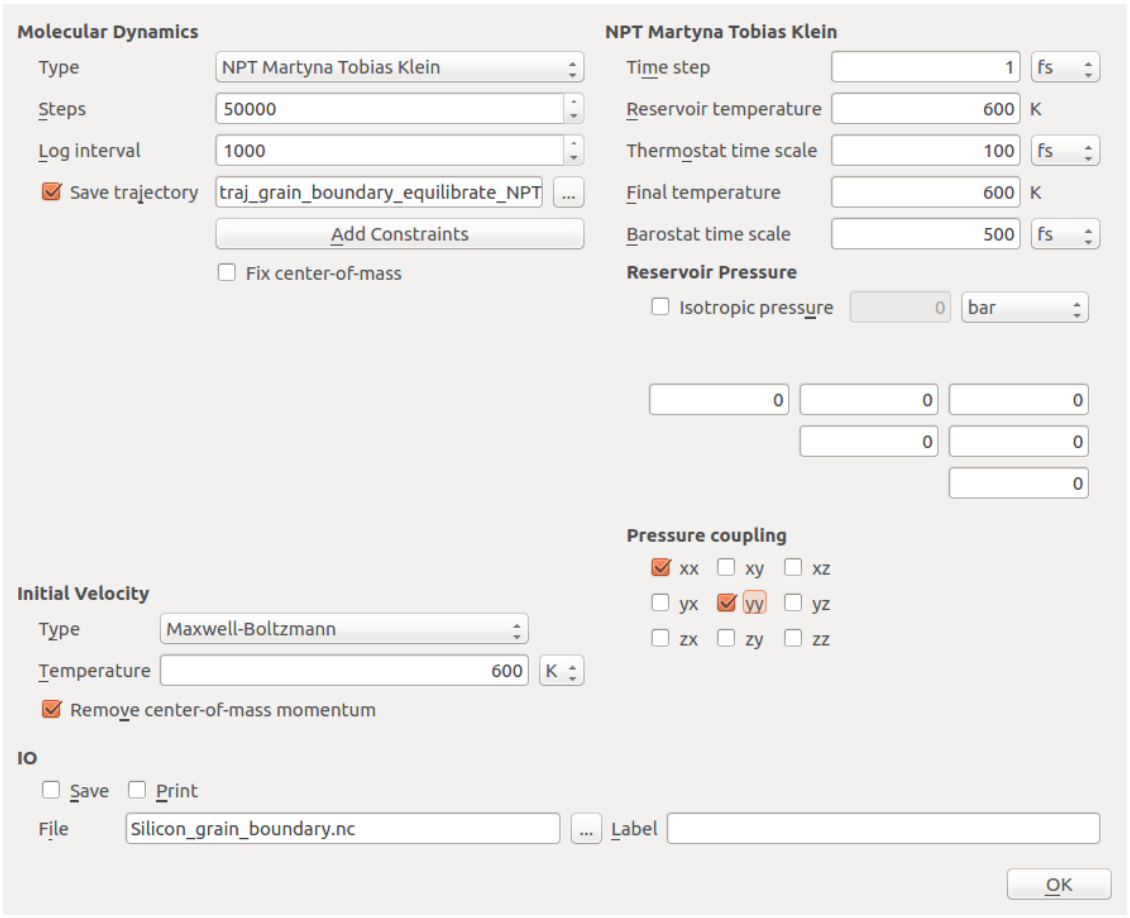
•Set the Reservoir temperature to 600 K, and the External stress to 0 bar.

•Uncheck Isotropic pressure and disable the Stress coupling in the zz-direction, as the free surface in this direction will automatically adjust.

•Select the Maxwell-Boltzmann-Distribution under *Initial Velocity*, set the temperature to 600 K, and make sure the box is checked to remove the center-of-mass momentum.

•Uncheck *Save and Print*, as the trajectory is saved anyway.

The final settings should look like the following.

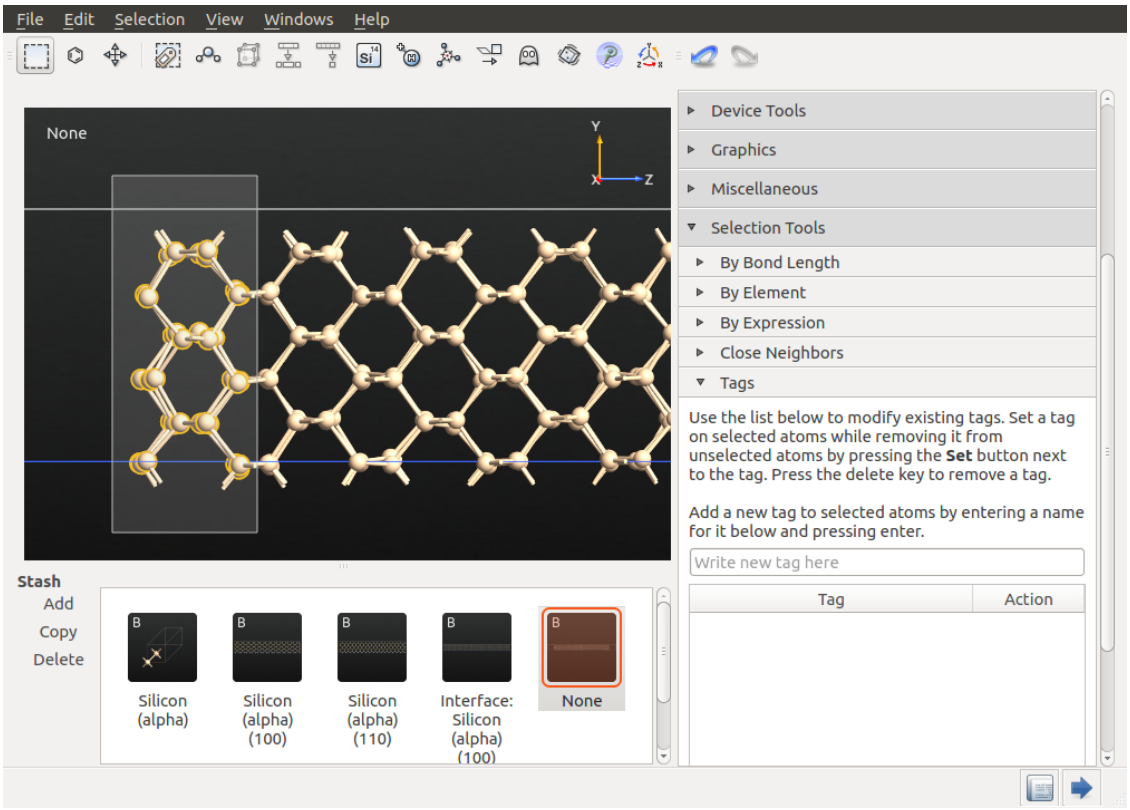


Send the script to the **JobManager** , save as `grain_boundary_equilibrate_NPT.py` when asked, and run it.

We will use the resulting structure from the NPT simulation for the following simulations. To this end, once the simulation has finished, select the trajectory on the Lab floor in the main QuantumATK window and open it with the Movie Tool. Play the trajectory and send the last snapshot to the **Builder** using the icon.

Setting up the non-equilibrium momentum exchange simulation

You now have to define the regions for the heat source and sink in the heat current simulations. Zoom into the left end of the configuration and select the last 4 atom layers (i.e. the first unit cell), as shown in the picture below.



•Open the *Select* → *Tags* plugin, enter a suitable tag name, e.g. heat_source, and press Return. The selected atoms will now be marked by this tag name and can be referred to during the simulation.

•Repeat the same procedure at the other end of the configuration. Use a different tag name this time, e.g heat_sink.

Send the configuration to the **Script Generator**, add a **New Calculator** and two **MolecularDynamics** blocks.

•Edit the calculator block in the same way as described above, i.e. selecting *ATK-ForceField* with the *StillingerWeber_Si_1985 potential*.

First, you will perform another equilibration simulation at 600 K, this time at constant volume, to prepare a well equilibrated system. Open the first **MolecularDynamics** block, and,

•Select the Type as NVT Nose Hoover.

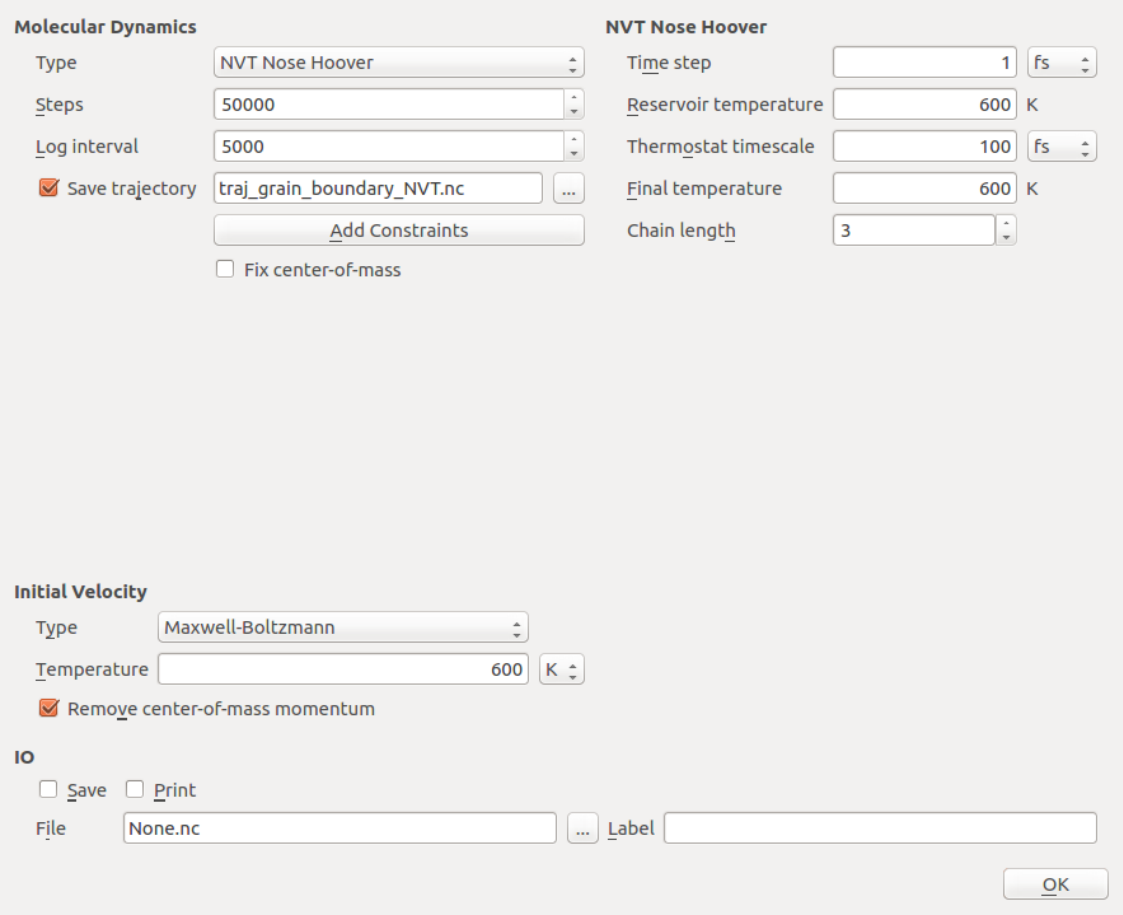
•Set the number of steps to 50 000, and the log interval to 5000.

•Choose a suitable trajectory filename, e.g. `traj_grain_boundary_equilibrate_NVT.nc`.

•Select Maxwell-Boltzmann initial velocities at 600 K, and make sure that the center-of-mass removal box is checked.

•Set the Reservoir temperature to 600 K, the final temperature will automatically change to 600K too. Leave the rest of the settings at default values.

•Uncheck Save and Print and close the widget by clicking OK.



In the second **MolecularDynamics** block, the actual non-equilibrium simulation will be set up. To do this:

•Select the type as Non-Equilibrium Momentum Exchange.

•Set the number of steps to 200 000, and the log interval to 1000.

•Select a suitable trajectory filename, e.g. `traj_grain_boundary_nonEq.nc`.

•Set the initial velocities to ConfigurationVelocities, which means that the velocities stored in the configuration will be used.

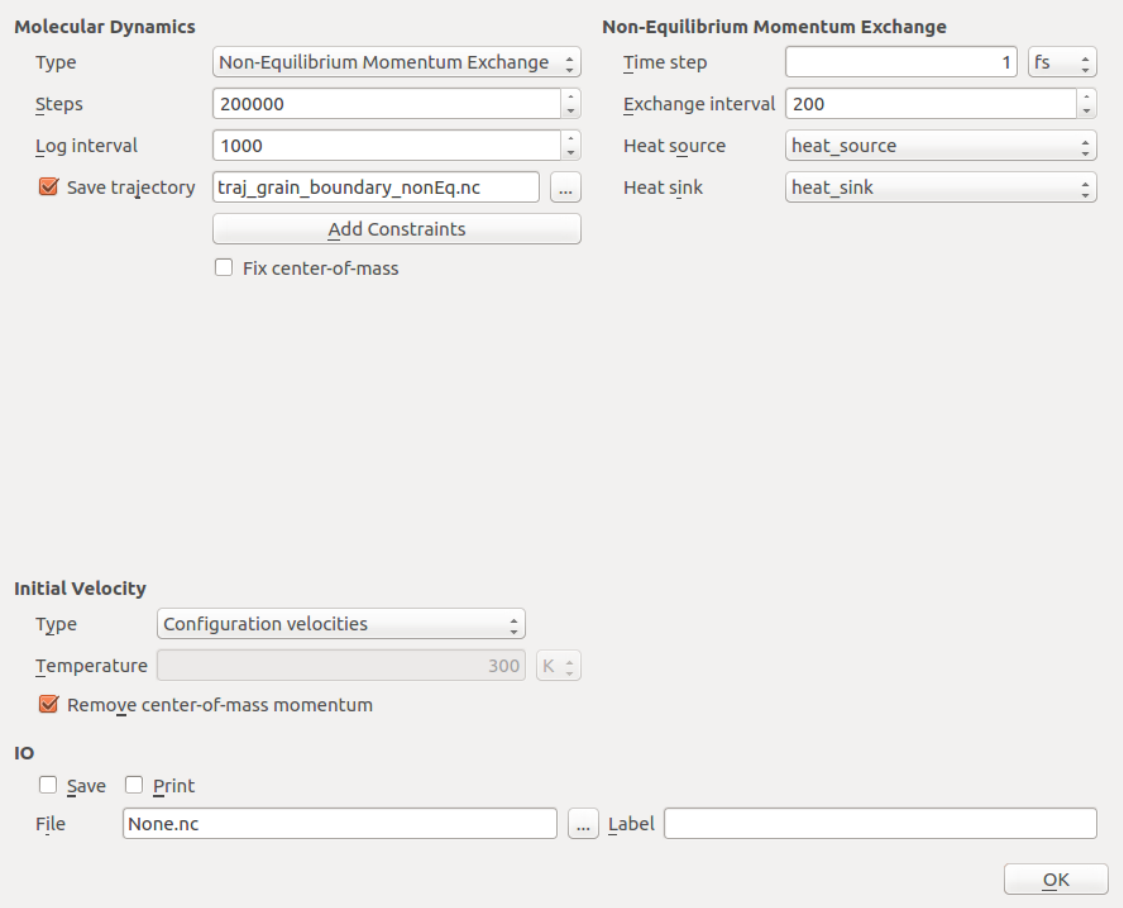
•Leave the Time step at the default value of 1 fs.

•Set Exchange interval entry to 200, meaning that the exchange is triggered every 200 steps.

•Uncheck Save and Print and close the widget by clicking OK.

Tip: Lowering the value of the *Exchange interval* entry will increase the transferred kinetic energy per simulation time, which causes a larger temperature gradient and a more distinct temperature profile. This will make the evaluation easier, but you generally should be careful with large temperature gradients, as their magnitude often by far exceeds the typical experimental temperature gradients.

Via the Heat source and Heat sink fields you can associate the tags, you have defined in the Builder, with heat source and sink, respectively. The final settings should look like the following.



Note: This simulation is set up for demonstration purposes to run reasonably fast - it should finish in about 10-15 minutes. For actual production simulations, you should use a larger (i.e. wider and much longer) system as well as a longer simulation time (e.g. 100 000 steps in the NVT equilibration and 800 000 steps or even more in the non-equilibrium simulation) to obtain reliable results and good statistics. As such NEMD simulations can be quite sensitive to finite size effects, you should always check your results for convergence with respect to system size and simulation time, if possible.

•Run the entire simulation via the **JobManager** or save the script and run it at a terminal using **atkpython**.

Analyzing the results

After the simulations have completed, the trajectory files will appear on the LabFloor. Open the non-equilibrium trajectory in the Movie Tool and you will see that the average temperature remains constants during the entire simulation and the slab does not drift, as this would blur the temperature profile.

Investigating the interfacial thermal conductance

Open the same trajectory with the **MD Analyzer** plugin to investigate the thermal conductivity,

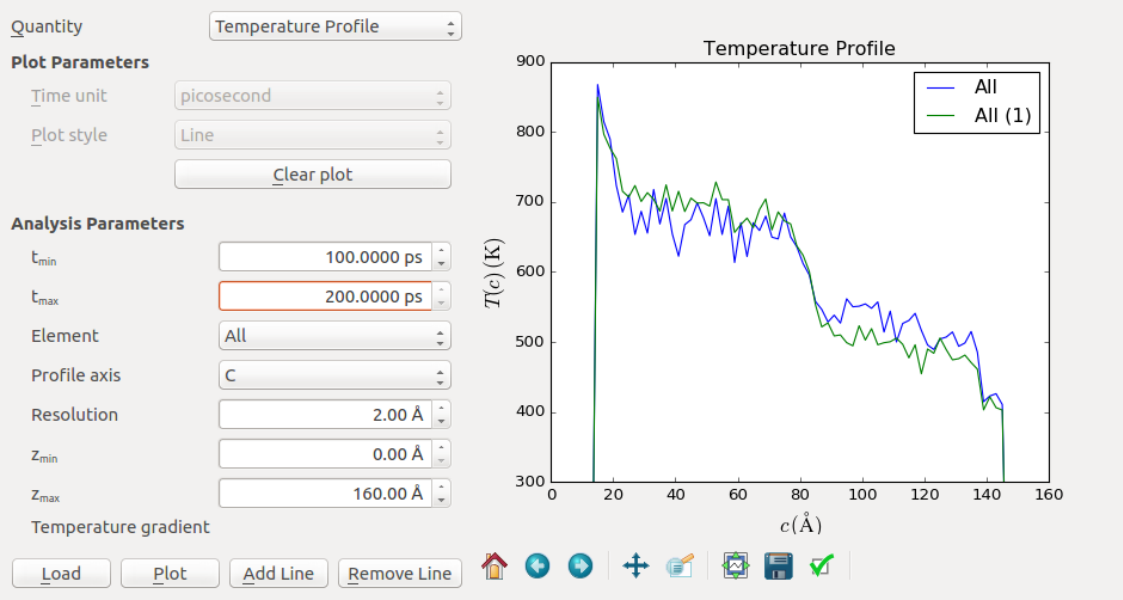
- Select the Temperature profile button and click *Load*. It may take some time.

> **Attention:** Here, you should carefully check the convergence of the temperature profile with respect to the simulation time. If you do not achieve a steady-state, you should run another non-equilibrium simulation with the same settings starting from the final snapshot of the previous one (in this case without the NVT equilibration).

To check whether the temperature profile has converged, you can compare the temperature profile between 1 and 50 ps, with that between 100 and 200 ps.

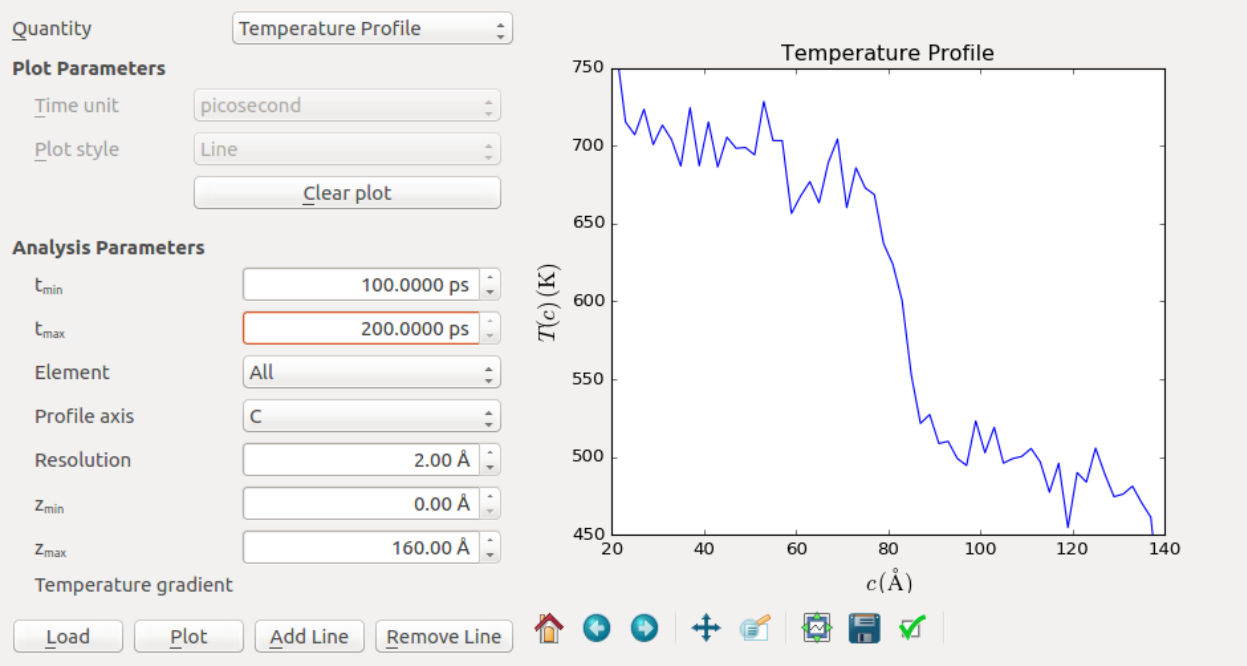
Tip: You can use the *Add Line* button to plot several curves on the same plot to make the comparison easier.

Use the zoom tool in the toolbar or the *Edit plot* button (on the bottom right) to zoom into the important features of the profile. Except for statistical fluctuations, your temperature profiles should be very similar to the plots shown here. In the plot below, the blue and green curve correspond to the intervals 1-50 ps and 100-200 ps, respectively.



The temperature profile shows several typical features. First, a positive and a negative peak are visible at the beginning and end of the profile. These regions designate the heat source and sink, respectively. This non-linear part arises from the artificial transfer of kinetic energy due to the momentum exchange. You should always discard these regions in your analysis.

The next prominent feature is the temperature jump around z=80 Å. This is the interface region which provides additional thermal resistance. Let us take a closer look by zooming into this region (shown below for the 100-200 ps curve). Here it can be convenient to use the *Edit plot* button to set the minimum and maximum and both the x and y-axis.



The temperature jump connects around $T_1 = 700$ K and $T_2 = 500$ K, so the jump is about $\Delta T = 200$ K. To calculate the Kapitza conductance you also need to know the average heat flux. This value can be read from the output at the bottom of the log file. In this example case, a value of $\langle dQ/dt \rangle = 0.0010$ eV/fs is obtained. These values can then combined to calculate the Kapitza conductance by using the formula above.

As the units have to be converted correctly, the most convenient way to calculate this value is to use the built-in unit engine of QuantumATK.

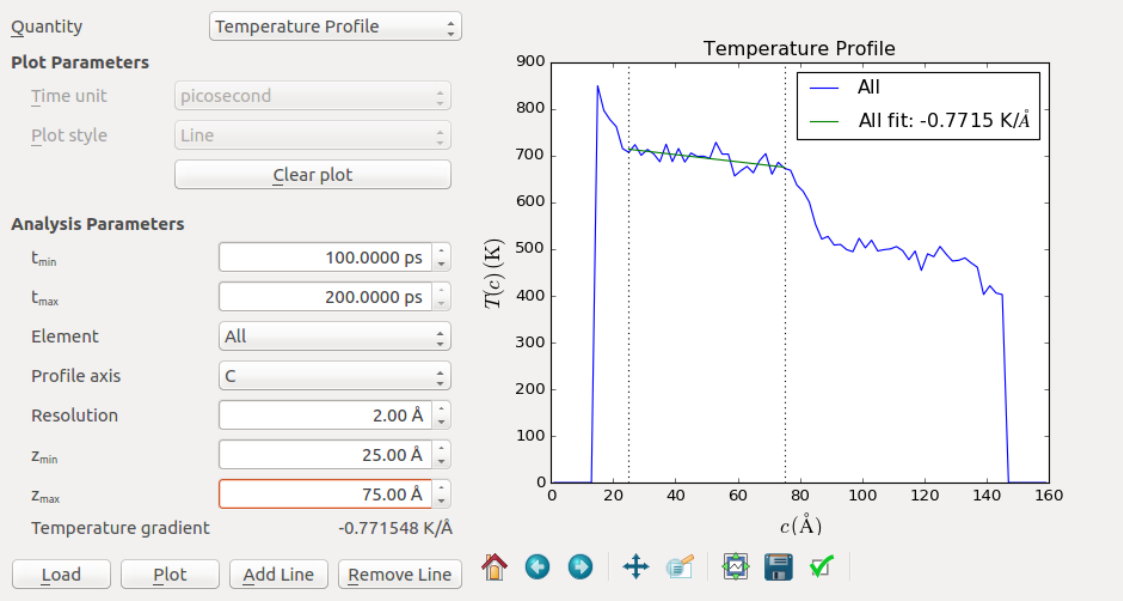
•Open the Editor and enter the following equation

```
dq_dt = 0.0010*eV/fs
delta_T = 200.0*Kelvin
G = dq_dt / delta_T
print("Kapitza conductance in Watt/Kelvin:  ",G.inUnitsOf(Joule/Second/Kelvin))
```

•Run this script via the **JobManager** (or atkpython) and save it as `kapitza.py`. The log file will display the calculated Kapitza conductance in W/K. In this example, the result is 8 x 10^{-10} W/K.

Investigating the grain size dependence of the thermal conductivity

You can also investigate the effect of the grain size, in this case modelled by the distance between the heat source or sink to the boundary, which is approximately 50 Å. Here, you will need to consider *thermal conductivity* instead of *interfacial conductance*. Use the MD Analyzer to extract the temperature gradient. Input reasonable numbers in Z_{min} and Z_{max} to select the linear region of the profile.



The program will automatically fit the slope and display the temperature gradient in the Parameters widget. A value of -0.77 K/Å is obtained for the (100) oriented grain in this example. You can repeat the procedure for the right grain and obtain a similar slope (around -0.90 K/Å). The slopes of the left and the right are unlikely to be identical, due to statistical fluctuations and different orientations.

We can now insert the calculated slope into the Fourier equation and, together with the average transferred energy per time and the cross-sectional area, calculate the thermal conductivity within the grain in a script. The cross-sectional area *A* can be obtained from the A and B cell vectors. You can find these by opening the *Bulk tools* → *Lattice parameters* for the interface configuration in the Builder. Open once again the `kapitza.py` and add the following lines in the bottom of the script. You can repeat the lines to make a calculation for both the (100) and (110) regions. When you have added the lines, send it to the Job Manager.

```
dq_dt = 0.0010*eV/fs
A = 11.46*11.24*Angstrom**2
dT_dz = 0.77*Kelvin/Angstrom
G = dq_dt / dT_dz / A
print("Grain conductivity in Watt/Kelvin/Meter:  ",G.inUnitsOf(Joule/Second/Kelvin/
↪Meter))
```

The result for this example is 16.1 W/K/m for the (100) region. For the (110) oriented grain the conductivity amounts to 13.8 W/K/m. These values are considerably smaller than the simulated bulk thermal conductivity for silicon at 600 K, κ_{bulk} = 150 W/K/m, reported in Ref[Page 1145, 3] using the same Stillinger-Weber potential. This behaviour is expected, since the thermal conductivity of a confined grain generally is lower than the corresponding bulk value[Page 1145, 4]. This is due to the fact that the mean free path of the phonons is often much larger than 1 μm, which means that the simulated grain size suppresses the thermal conductance[Page 1145, 3][Page 1145, 4].

To study the grain size dependence, you can repeat the simulation, preparing configurations with different lengths of the two crystal cells, as shown in Ref.[Page 1145, 4].

The bulk conductivity can be obtained by performing several simulations at increasing grain sizes l, plotting the inverse conductivity $1/\kappa(l)$ over the inverse grain size $1/l$ and extrapolating to infinite grain sizes[Page 1145, 3Page 1145, 4].

Non-equilibrium Greens function method

In this section you will apply another method, the non-equilibrium Greens function (NEGF) technique, to calculate the interface thermal conductance. You will use NEGF to calculate the phonon transmission spectrum and from that quantity obtain the thermal conductance. For more details about phonon transmission using the NEGF technique see e.g. Ref.[5] or the tutorial *Thermoelectric effects in a CNT with isotope doping*.

Preparing the configuration

For this method, we will set up a different calculation but the same interface configuration. Therefore, return to the **Builder** and select the interface configuration you created earlier.

This time, before setting up the script, convert this configuration into a device configuration by clicking *Device Tools* → *Device from Bulk*. Here you can leave all settings at default values and press OK.

Setting up the NEGF simulation

Send the device configuration to the **Script Generator** by clicking the Send To icon, and add 3 blocks:

- New Calculator
- *Optimize* → *OptimizeGeometry*
- *Analysis* → *PhononTransmissionSpectrum*

This will automatically add a **DynamicalMatrix** block as well.

Change the Default output file name to `Silicon_grain_boundary_NEGF.nc`, and open the **New Calculator** block.

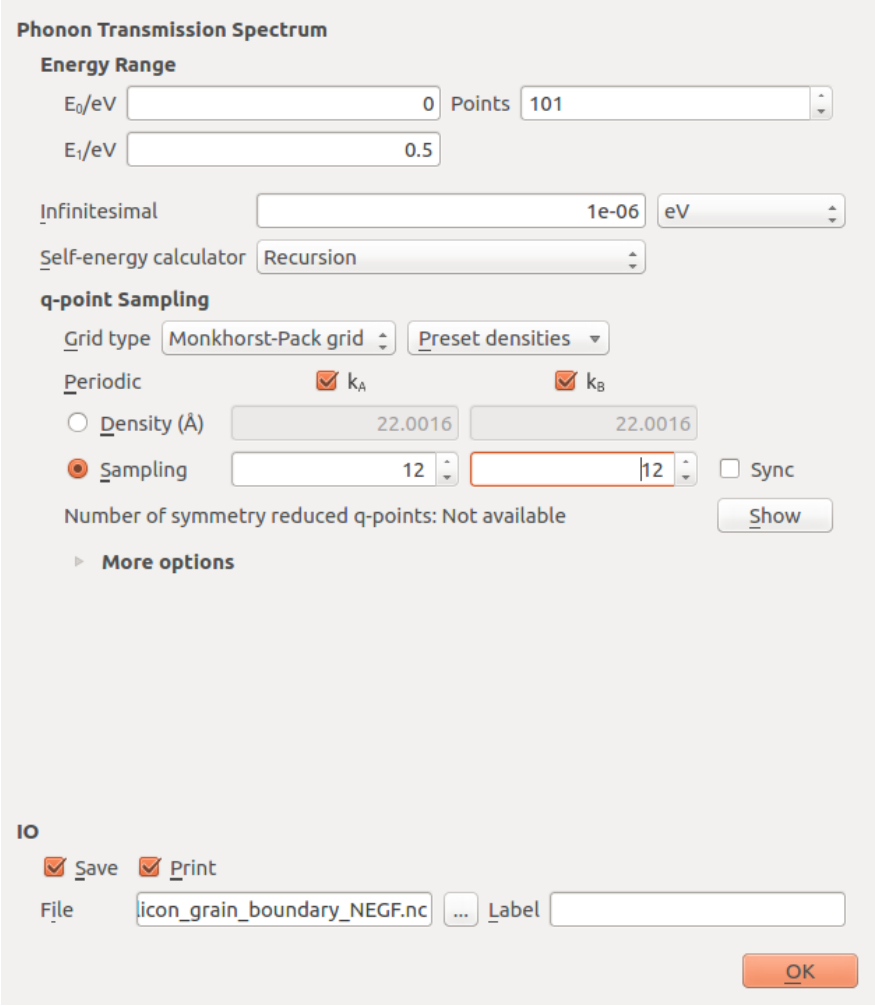
•Select *ATK-ForceField* and the *StillingerWeber_Si_1985 potential*.

•Uncheck *Save and Print* and close the widget by clicking OK.

Open the **OptimizeGeometry** block and set the Force tolerance to 0.001 eV / Å. The remaining parameters can be left at their default values. Uncheck *Save and Print* and close the widget.

Next, open the **PhononTransmissionSpectrum** block and set the **q-point Sampling** to *Sampling* and enter 12 for the sampling in both directions. The settings should look like below

[5] T.ăMarkussen, A.-P. Jauho, and M.ăBrandbyge. Electron and phonon transport in silicon nanowires: Atomistic approach to thermoelectric properties. *Phys. Rev. B*, 79:035415, Jan 2009. doi:10.1103/PhysRevB.79.035415.

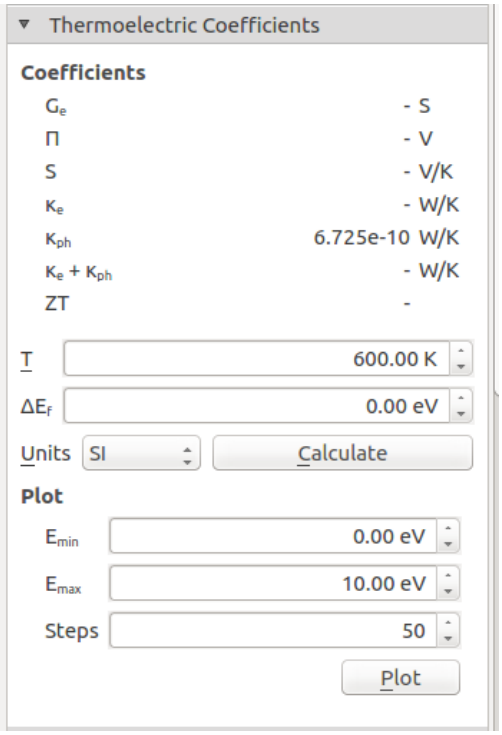


Click OK to close it. If you are running QuantumATK 2016.3 or earlier versions, then before running the script, you should make sure that *Auto scan project files* is unchecked under *File → Preferences* in the main QuantumATK window - this will prevent a slight bug form occuring during the simulation.

Then send the script to the **JobManager** . Save as `grain_boundary_NEGF.py` when asked, and run it. The simulation should finish in about 5-10 minutes.

Analyzing the results

After the simulations have completed, the phonon transmission spectrum object will appear on the LabFloor. Select it and expand the Thermoelectric Coefficients plugin on the right-hand panel. To compare with the MD calculation of the thermal properties, set the temperature to 600 K and click Calculate. The thermal conductance should be printed in the window, and should be similar to the one shown below



Comparing results between RNEMD and NEGF

From the NEGF calculations we find a conductance of 6.7 x 10 $^{-10}$ W/K, which is already quite close to the result calculated with RNEMD (which was 8 x 10 $^{-10}$ W/K). The slight deviation can be attributed, among others, to the absence of inelastic phonon scattering in NEGF, and to finite size effects in our small model system, which affects in particular the particular the NEMD simulations.

References

3.8 Molecular Dynamics

3.8.1 How to Setup Basic Molecular Dynamics Simulations

Version: U-2022.12

Downloads & Links

Molecular Dynamics `Silicon_bulk.py` `md_tutorial_nve_1fs.hdf5` `md_tutorial_nvt_300k.hdf5` `md_tutorial_nvt_1000k.hdf5` `md_tutorial_npt_300k_1bar.hdf5`

In this tutorial, you will learn how to setup basic molecular dynamics (MD) simulations and analyze the resulting trajectories using the NanoLab GUI in **QuantumATK** . At the end of this tutorial, you will be familiar with simulating systems in a microcanonical ensemble (NVE), canonical ensemble (NVT) and isothermal-isobaric ensemble (NPT), using **QuantumATK**.

Pre-requisites

•Basic familiarity with **NanoLab** and **atkpython**

•Basic familiarity with Molecular Dynamics simulations. If needed, you can read our *technical note on Molecular Dynamics* for a basic introduction to MD along with best practices, tips and tricks when setting up and analyzing MD simulations.

NVE Simulations

Let us setup a microcanonical ensemble simulation in which the number of particles (N), volume of the simulation cell (V) and energy of the system (E) remains constant. Import the geometry we will need, `Silicon_bulk.py`, to your Builder stash.

Workflow

1.Select the silicon supercell in **Builder** stash.

2.Click on the icon and send the bulk configuration to the **Workflow Builder**

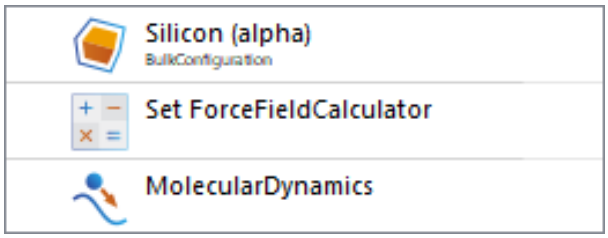
3.In the **Workflow Builder**, change the output **Filename** at the bottom of the middle panel to `md-nve-1fs.hdf5`.

4.Add the following blocks to the **Build** panel by double clicking on the corresponding icons in the **QuantumATK** tab in the right-hand panel:

•Under **Calculators**: *Set ForceFieldCalculator*

•Under **Optimization and Dynamics**: *MolecularDynamics*

The resulting workflow will look like the below picture

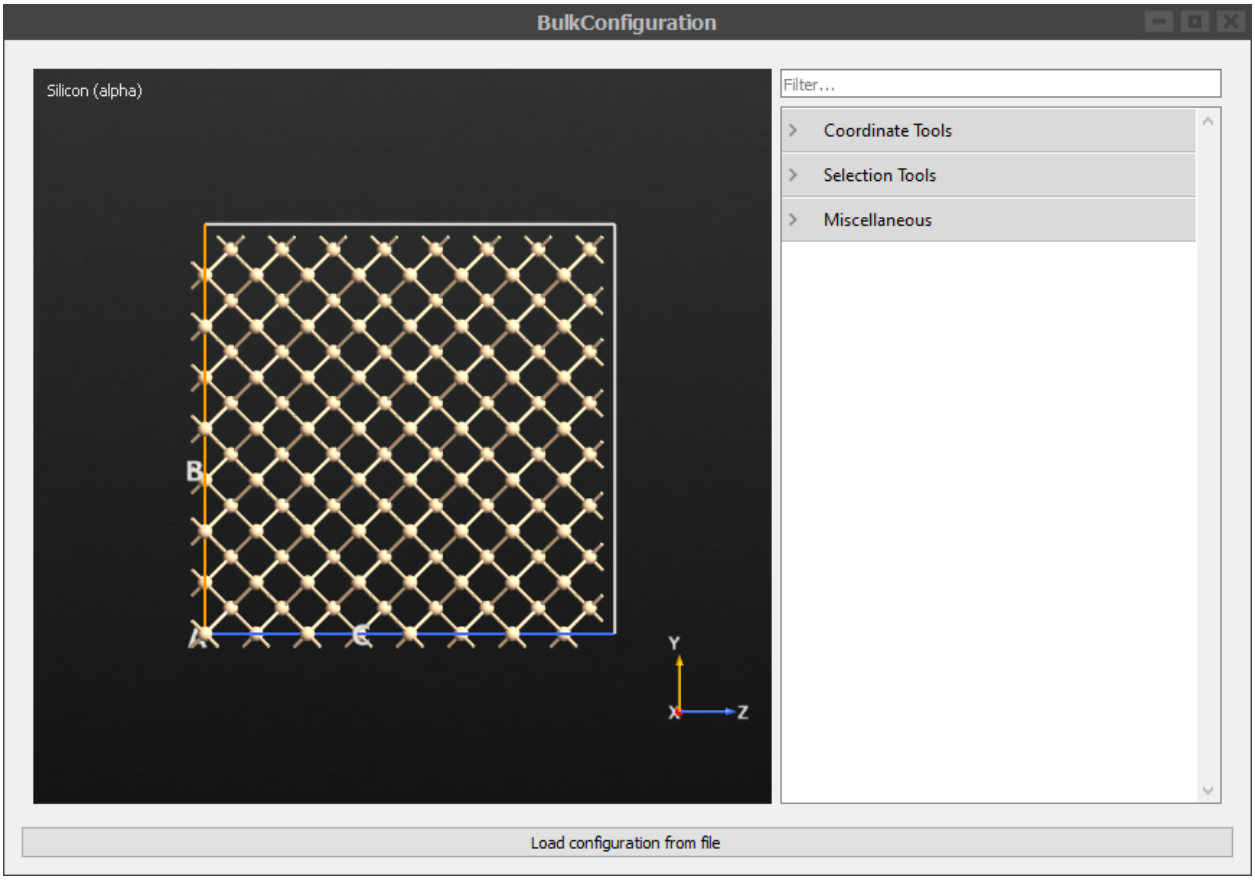


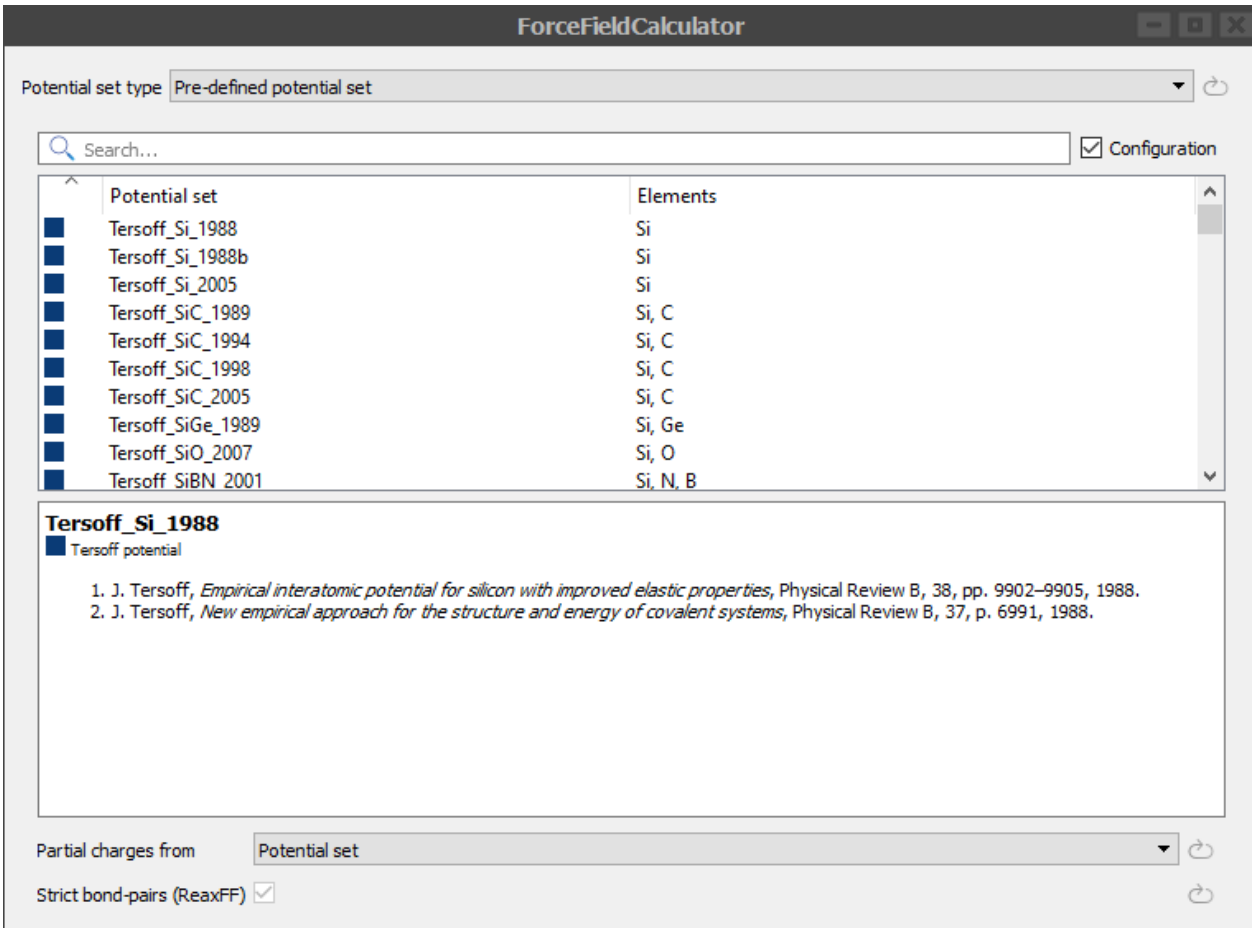
5.Double click the *Silicon (alpha)* block to visualize the bulk supercell. This is the system we will run MD for.

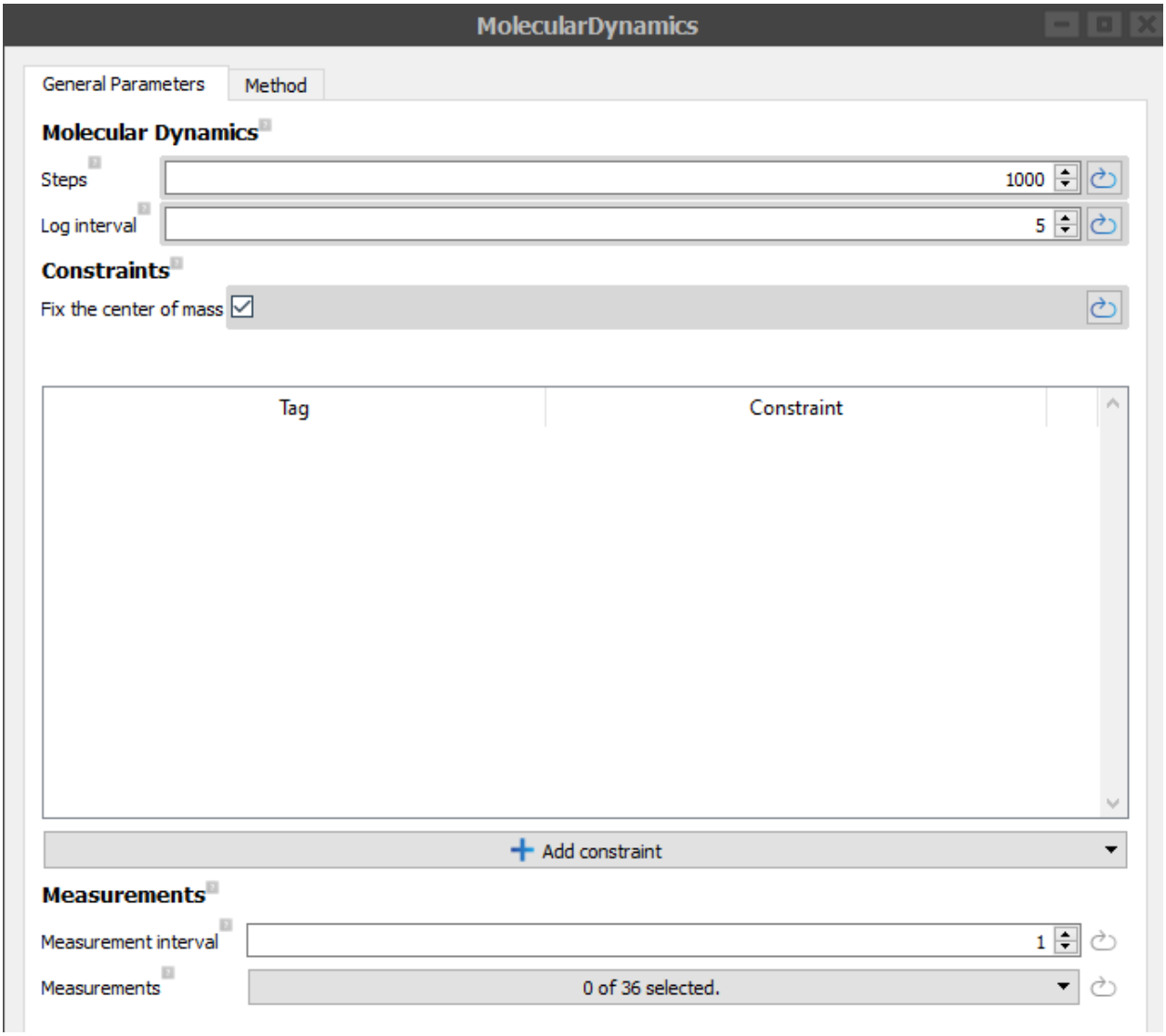
6.Double-click on the *Set ForceFieldCalculator* block and select the *Tersoff_Si_1988* parameter set as shown below

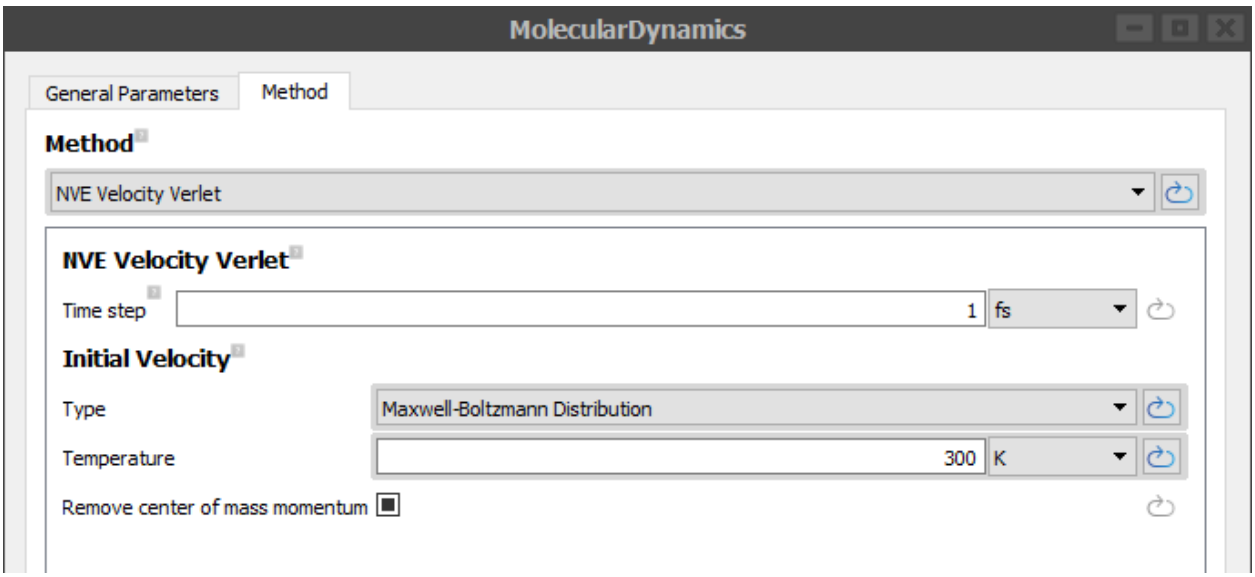
7.Double-click on the *MolecularDynamics* block, set the **Steps** to *1000*, **Log interval** to *5* and set *Fix the center of mass* constraint

8.Go to **Methods** tab and choose **Method** as *NVE Velocity Verlet* from the pull down menu, set **Initial velocity Type** to *Maxwell-Boltzmann Distribution* and **Temperature** to *300 K*.


BulkConfiguration
Silicon (alpha)
B
A
C
Y
X
Z
Filter...
Coordinate Tools
Selection Tools
Miscellaneous
Load configuration from file


ForceFieldCalculator
Potential set type
Pre-defined potential set
Search...
Configuration
Potential set
Elements
Tersoff_Si_1988
Si
Tersoff_Si_1988b
Si
Tersoff_Si_2005
Si
Tersoff_SiC_1989
Si, C
Tersoff_SiC_1994
Si, C
Tersoff_SiC_1998
Si, C
Tersoff_SiC_2005
Si, C
Tersoff_SiGe_1989
Si, Ge
Tersoff_SiO_2007
Si, O
Tersoff_SiBN_2001
Si, N, B
Tersoff_Si_1988
Tersoff potential
1. J. Tersoff, Empirical interatomic potential for silicon with improved elastic properties, Physical Review B, 38, pp. 9902–9905, 1988.
2. J. Tersoff, New empirical approach for the structure and energy of covalent systems, Physical Review B, 37, p. 6991, 1988.
Partial charges from
Potential set
Strict bond-pairs (ReaxFF)


MolecularDynamics
General Parameters
Method
Molecular Dynamics
Steps
1000
Log interval
5
Constraints
Fix the center of mass
Tag
Constraint
Add constraint
Measurements
Measurement interval
1
Measurements
0 of 36 selected.

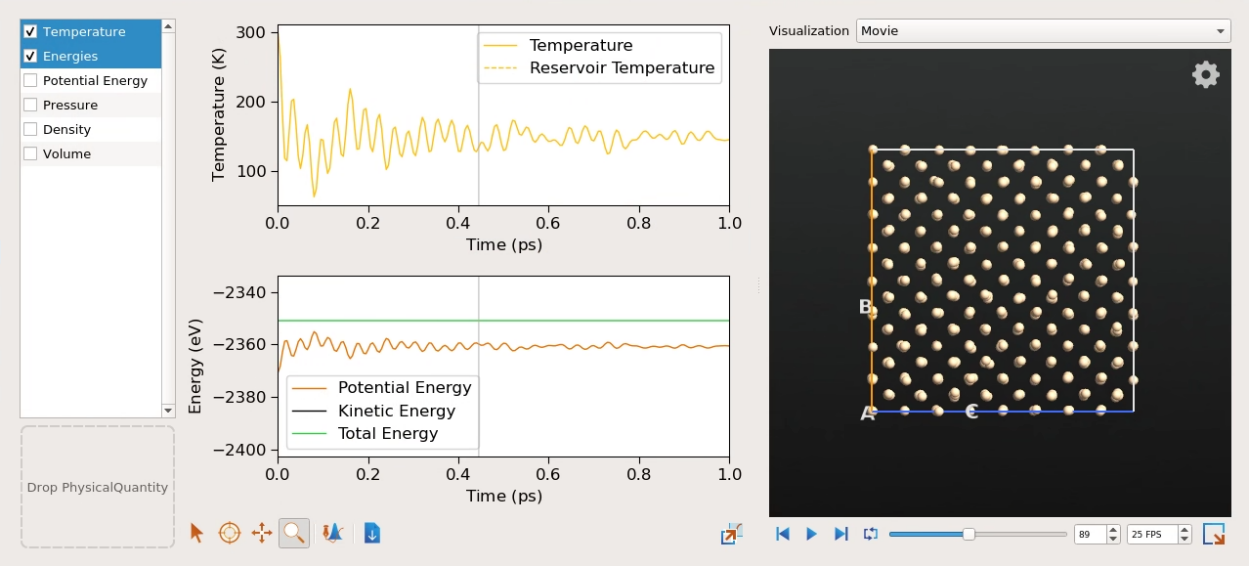

MolecularDynamics
General Parameters
Method
Method
NVE Velocity Verlet
NVE Velocity Verlet
Time step
1
fs
Initial Velocity
Type
Maxwell-Boltzmann Distribution
Temperature
300
K
Remove center of mass momentum

9. Send the workflow to the **Job Manager** as a script using and save the script as `md-nve-1fs.py`. This script runs in a few seconds, it can run locally.

   Alternatively, download the workflow `md_tutorial_nve_1fs.hdf5` to your project folder, open it in the Workflow Builder Tool in the Nanolab, export to script and submit the job.

Analyzing the Results

Once the simulation is finished, select the `md-nve-1fs.hdf5` file from the project folder and double-click the *MDTrajectory_0* object from the Data View. The trajectory will open in the **Movie Tool** plugin. Using the **Movie Tool** plugin, you can investigate the temperature as well as the kinetic, potential, and total energies of the system as a function of simulation time. Simultaneously, an animation of the motion of the atoms is displayed.



If you take a look at the temperature, you will observe the temperature dropping immediately and then oscillate around the set temperature. The reason is that the displayed temperature is the instantaneous temperature of the system, calculated from the kinetic energy of the atoms, via $E_{kin}(t) = 3/2Nk_BT(t)$. Since the atoms are not free but interact with the neighboring atoms in the lattice, their initial kinetic energy will be partially converted into potential energy. For an *NVE* ensemble, a crucial parameter to monitor is the conservation of the total energy and find that the total energy curve in the **Movie Tool** remains constant with only small oscillations which are acceptable.

Tip: Try changing some MD parameters and see how they affect dynamics. For more tips, refer to *Molecular Dynamics*.

NVT Simulations

In NVT, we keep the temperature constant instead of the energy as in the NVE case. Download the workflow `md_tutorial_nvt_300k.hdf5` to your project folder and open it in the Workflow Builder Tool in the Nanolab.

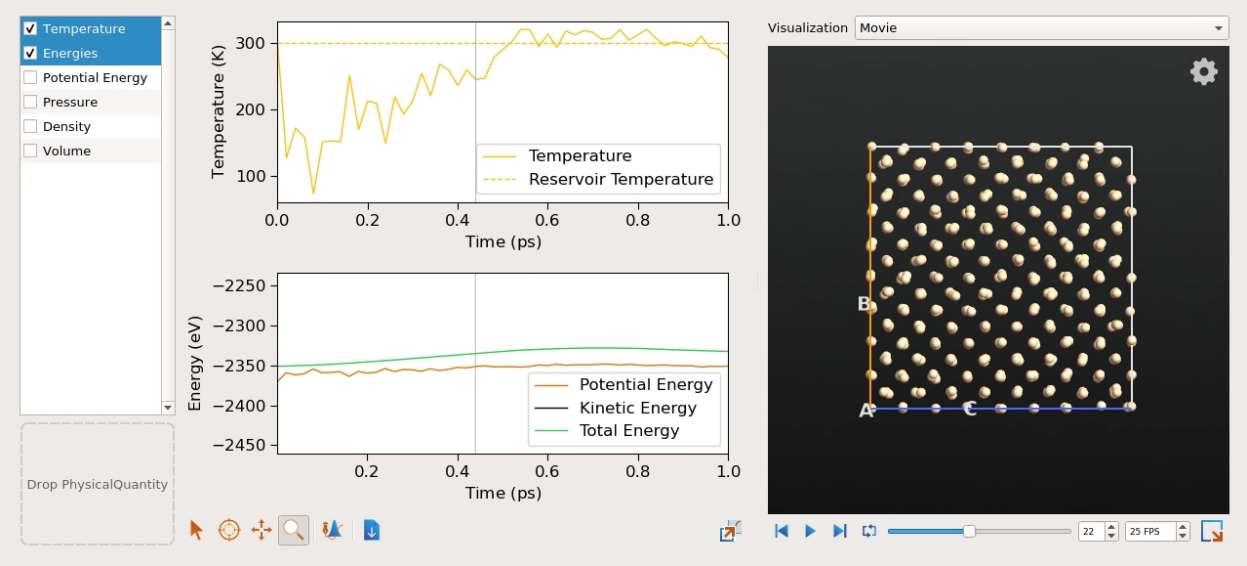
Workflow

The workflow should look similar to the NVE case, but the MolecularDynamics block should list *NVT* as the **MD method** and a **reservoir temperature** of *300 K* will be listed. In this example, we use the *Nose-Hoover* type thermostat coupling to the reservoir. For more details on this method and other available thermostats, refer to the technical note on *Molecular Dynamics*.

After loading the workflow, export it as a script and submit the job.

Analyzing the Results

After the simulation is completed, open the NVT MD trajectory object with the **Movie Tool** and take a look at the temperature curve.



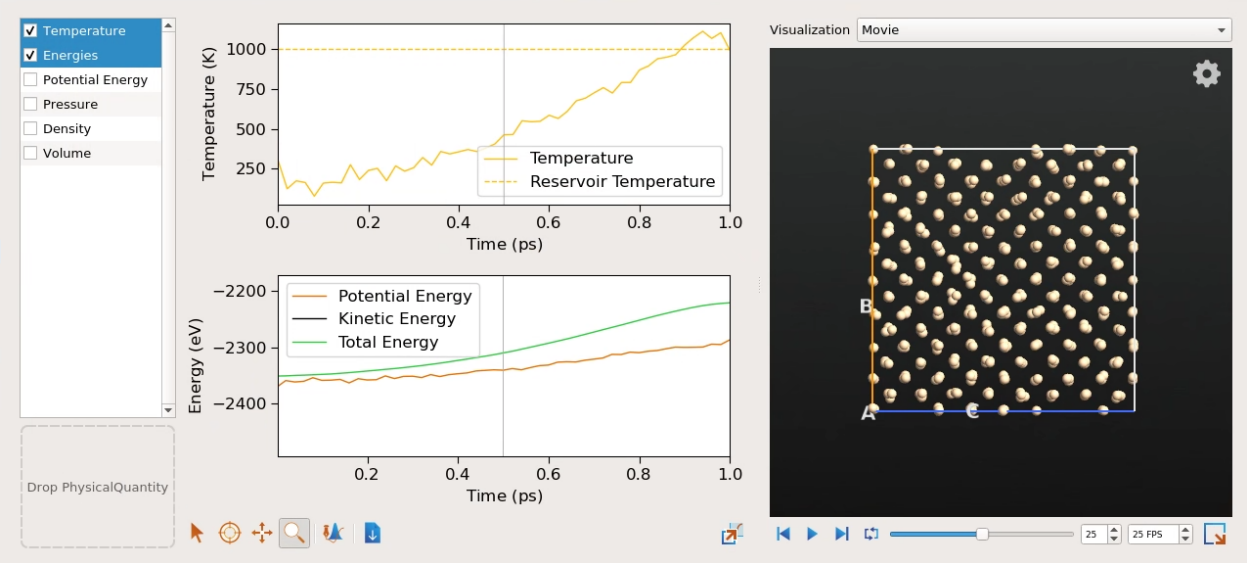
You will find that after a short initial equilibration period the curve fluctuates slightly around the selected reservoir temperature of 300 K. In fact, it does not look very different from the results of the NVE simulation in the previous section. In contrast to the NVE simulation, however, the final equilibrium temperature of the system is independent of the initial value, set via the Maxwell-Boltzmann distribution, and is instead coupled to the reservoir temperature. Let us take a closer look at this now.

Download the workflow `md_tutorial_nvt_1000k.hdf5` in which the **reservoir temperature** is set to *1000 K*. Leave the **initial velocities** as a *Maxwell-Boltzmann* distribution at 300 K and run the script.

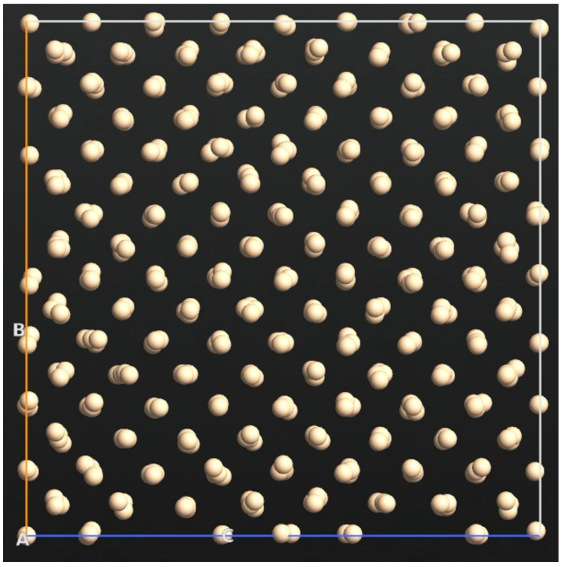
On opening the resulting trajectory in the movie tool, we see the temperature curve starting at an initial value of 300 K (as selected via the *Maxwell-Boltzmann distribution*).

Subsequently, as the system is in contact with the heat bath, the temperature quickly increases towards the reservoir temperature of 1000 K. You may also notice an increase in the total energy. In contrast to *NVE* simulations, this is not a sign of poorly chosen parameters, but reflects the fact that energy is exchanged with the virtual heat bath.

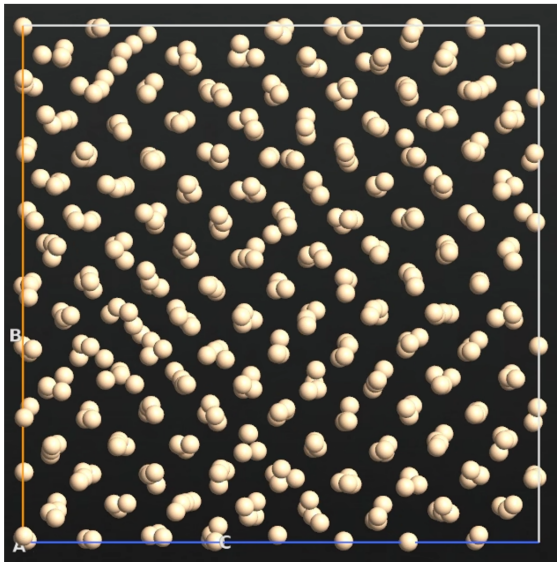
During the simulation at 1000 K, the motion of the atoms increases compared to the previous simulation at 300 K.


Temperature
Energies
Potential Energy
Pressure
Density
Volume
Drop PhysicalQuantity
Temperature (K)
1000
750
500
250
0.0
0.2
0.4
0.6
0.8
1.0
Time (ps)
Temperature
Reservoir Temperature
Energy (eV)
−2200
−2300
−2400
Potential Energy
Kinetic Energy
Total Energy
Time (ps)
Visualization
Movie
B
A
C
25
25 FPS

300 K


B
A
C

1000 K


B
A
C

If you increase the reservoir temperature to even higher values (e.g 3000 K), you will find that the crystal will begin to melt and become partially amorphous. Annealing crystals at high temperatures is in fact a common way to generate amorphous structures of a material. You can learn more about this method in the tutorial *Generating Amorphous Structures*.

Warning: Setting too high a temperature on a minimum energy configuration might result in the generation of very large forces on the atoms which could destabilize the system. Instead, slowly heat up the system from room temperature using a suitable *heating rate*.

NPT Simulations

In this ensemble, we keep the number of particles, pressure and temperature constant. Download the workflow `md_tutorial_npt_300k_1bar.hdf5` to your project folder and open it in the **Workflow Builder** tool in the NanoLab.

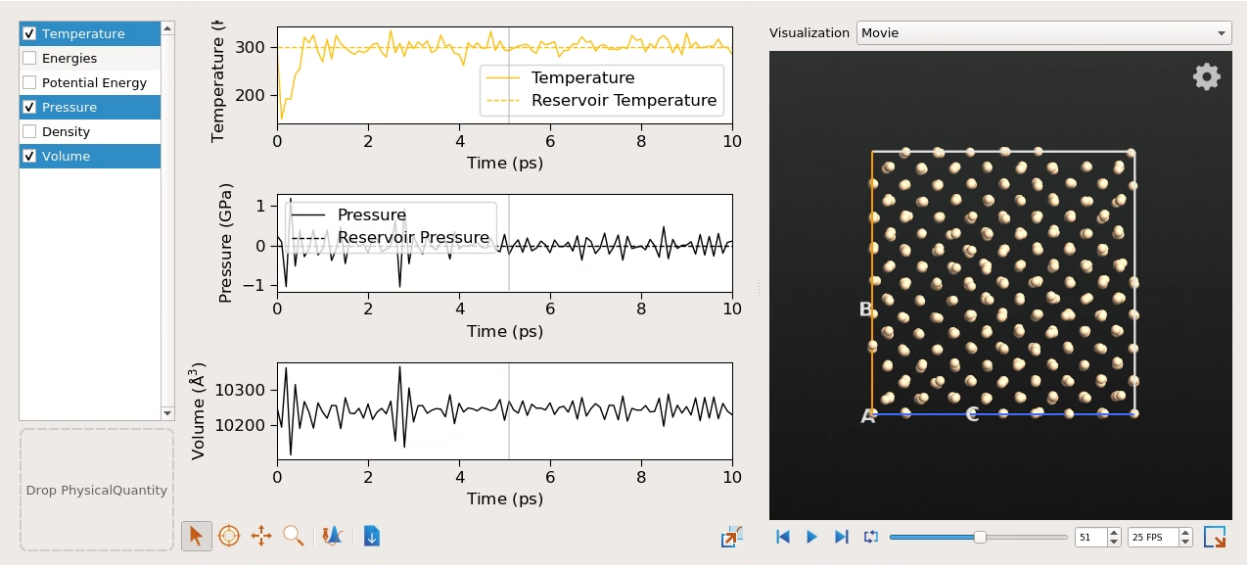
Workflow

The main difference you will observe in the MD block is the choice of NPT method. We use **NPT Martyna Tobias Klein** barostat in this example. For more details about this method and other NPT methods, refer to *Molecular Dynamics*. As you will notice the **reservoir temperature** and **pressure** are set to *300 K* and *1 bar*, respectively and an *isotropic coupling* is selected. This is because are simulating the system at standard temperature and pressure conditions and also allow the system to relax equally on all three lattice directions. Anisotropic coupling can be enabled to choose to fix some simulation cell boundaries.

Export the workflow as script and submit the job.

Analyzing the Results

Opening the trajectory with the **Movie Tool**, you can monitor the changes in the size and shape of the cell during the simulation. The temperature fluctuates around the selected reservoir temperature of 300 K.



To inspect the volume fluctuations you can select the *volume* measurement in the **Movie Tool**. The cell volume will show pronounced oscillations. These oscillations are a typical outcome of the *Nose-Hoover-like* temperature and pressure

control. As long as their magnitude stays within reasonable boundaries, and does not increase dramatically throughout the simulation, there is no need to worry. The period of these oscillations can be changed by the *Barostat timescale* setting. The amplitude and frequency of the oscillations increases with decreasing barostat timescale.

The pressure during the simulation, seen by selecting the *Pressure* measurement in the **Movie Tool**, shows a similar behavior. It oscillates around the target pressure of 1 bar, but the amplitude is large. Such large fluctuations of the instantaneous pressure values are in fact not unusual in MD simulations, as the pressure is calculated based on the stress, which varies considerably with small changes of the volume. Moreover, pressure is a macroscopic property, and for such a small system it is more reasonable to consider the corresponding average rather than the instantaneous values.

Conclusion

You have learnt how to use the Workflow Builder tool to set up basic molecular dynamics simulations of NVE, NVT and NPT ensembles. You also know how to view the resulting trajectories using the Movie tool and interpret the results.

Feel free to change the MD settings and run the simulations to see how the dynamics changes and to familiarize yourself with the method.

You can try adding multiple MD blocks within a workflow to simulate the system in different ensembles in a sequential manner. For more inspiration, checkout predefined MD templates in the Templates section of the Workflow Builder.

3.8.2 Simulating Thin Film Growth via Vapor Deposition

Introduction

In this tutorial you will learn how to use QuantumATK to run molecular dynamics (MD) simulations to simulate thin film growth onto a substrate via vapor deposition processes.

Vapor deposition is a widely used technique for the production of thin films composed of crystalline or amorphous solids on a substrate material. Simulations of such processes can help understand the underlying microscopic mechanisms, and the effect of the process parameters such as pressure, temperature, etc. on the resulting atomistic structure and kinetics.

In this tutorial you will simulate the growth of silicon carbide (SiC) onto a crystalline SiC substrate, similar to the study in Ref.[1]. You will use empirical potentials which facilitate the simulation of larger systems and time scales than accessible in ab-initio methods, such as DFT. The simulations shown in this tutorial primarily fall under the category physical vapor deposition (PVD). However, chemical vapor deposition (CVD) processes can, in principle, be simulated as well, given that either suitable force fields to describe the chemical reactions are involved, or sufficient computational resources for the use of ab-initio methods are available.

For this tutorial you should be familiar with the basic functionalities of molecular dynamics as described in the *How to Setup Basic Molecular Dynamics Simulations*. You will learn how to use Python scripting together with the hook functionality available in QuantumATKs molecular dynamics routines to run advanced simulations.

Note: The exact modified embedded atom model (MEAM) potential used in Ref.[Page 1165, 1] is currently not available in QuantumATK. Instead, you will use a Tersoff potential, which, unfortunately, will not produce the crystalline layers shown in the reference paper. The simulation technique, though, is of course independent of the exact potential, and can be used for all kinds of materials.

---

[1] Kang, K.-H; Eun, T.; Jun, M.-C.; Lee, B.-J.: Governing factors for the formation of 4H or 6H-SiC polytype during SiC crystal growth: An atomistic computational approach. J. Cryst. Growth, 389, 120 (2014)

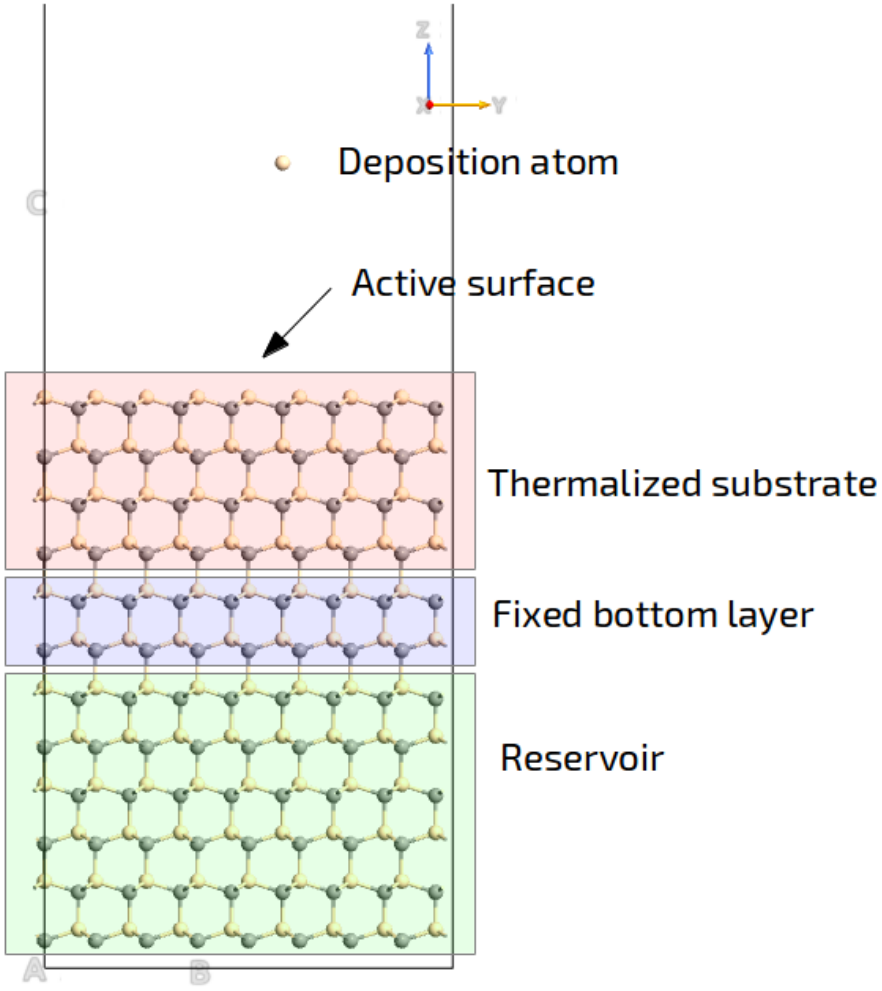
Note: The limited accessible simulation time requires to operate at vapor pressure values, which are considerably higher than in experiments. You should always keep this in mind when you interpret the simulation results. A more detailed discussion can be found in the general *remarks* section.

Simulation Strategies

There are different techniques to run a deposition simulation. The main difference to conventional equilibrium MD simulations is that the number of particles in the system increases due to the introduction of vapor atoms or molecules. In QuantumATK two strategies are possible in order to achieve this:

1. Run a new simulation for each newly introduced atom or molecule. The entire deposition simulation can then be performed in an outer loop over the deposited atoms/molecules.
2. Keep all atoms or molecules that should be deposited in a reservoir in the simulation cell. For each new deposition event, take one of these atoms out of the reservoir and place it above the substrate.

The first approach has the advantage that only the atoms that are really needed are present in the simulation cell, which improves the efficiency of the simulation. However, in QuantumATK it is not possible to save the entire simulation into a single *MDTrajectory* object for later visualization and analysis, due to the varying number of atoms. Therefore, the second approach is chosen in this tutorial. When preparing the simulation one has to pay attention that the atoms in the reservoir do not interact significantly with the active part of the system, in particular with the surface at which the adsorption takes place. In the present case, the reservoir will be represented by a crystal that is attached at the bottom of the substrate.



Preparing the System

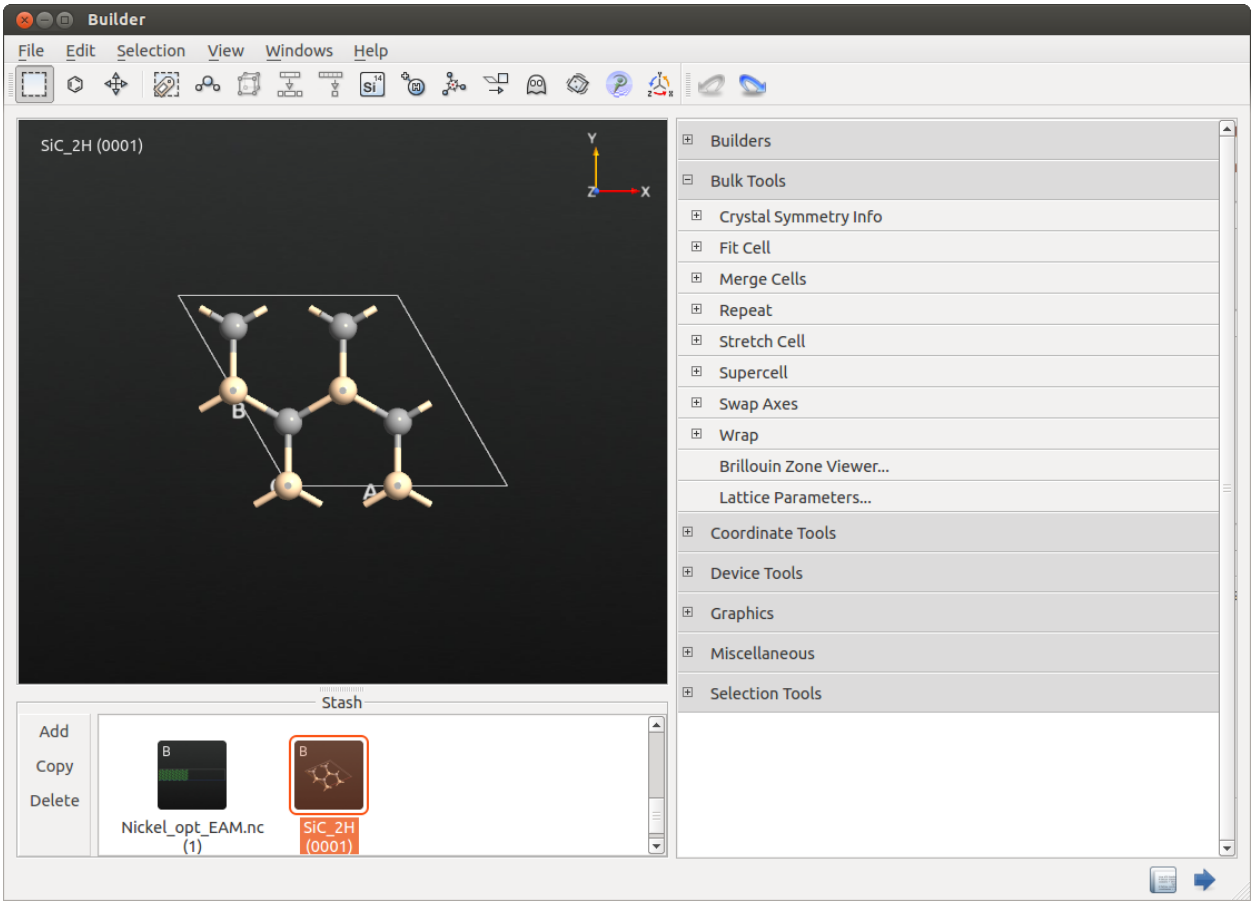
Silicon carbide crystals can occur in different polymorphs. In this tutorial, the substrate will be built from the basic *2H-SiC* crystal structure, which is in the configuration database in **QuantumATK**. You can of course also use a different SiC polymorph to prepare your substrate crystal.

Open the Builder, click *Add* → *From Database*, search for *SiC*, and add the *Wurtzite* structure.

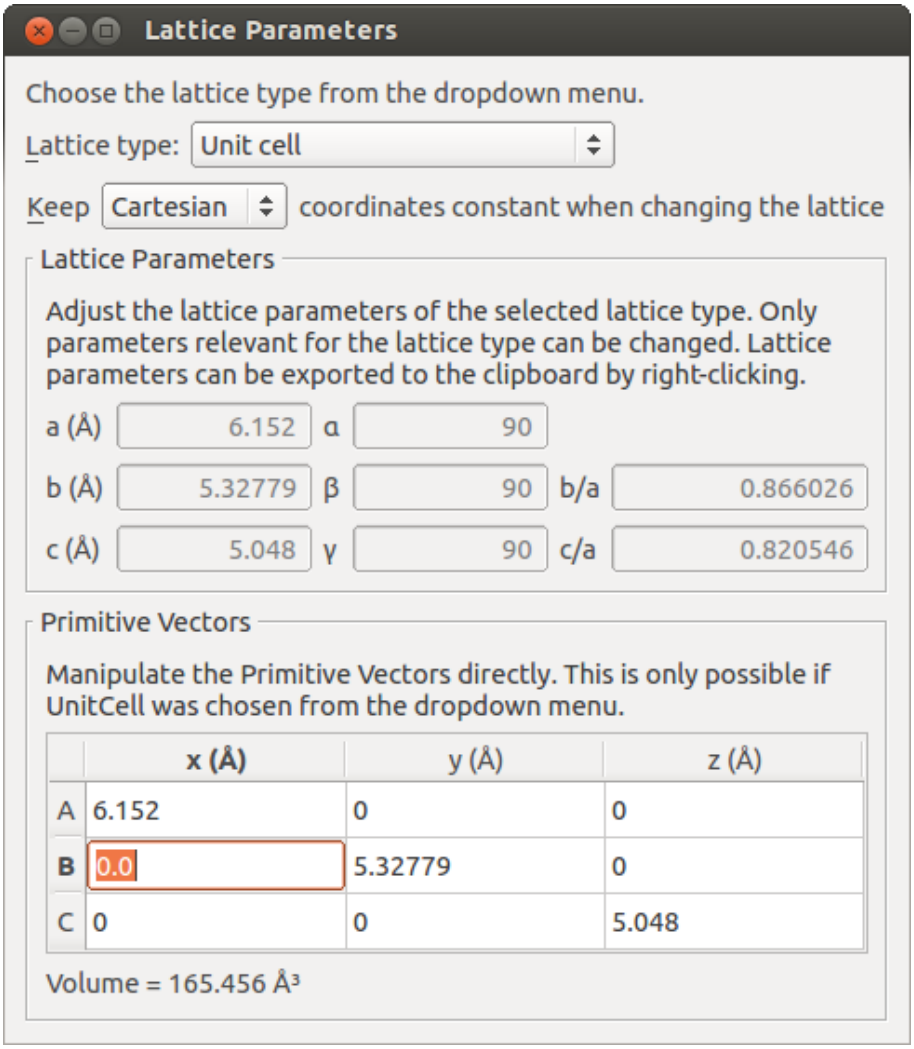
Preparing the Surface and Reservoir

For this simulation we will consider the (0001) surface of the SiC crystal. To build this surface use the *Builders* → *Surface (Cleave)* plugin in the panel bar on the right-hand-side. Select the (0001) Miller indices and a 2x2 surface lattice. Finally, select a *Periodic (bulk-like)* out-of-plane cell vector and a thickness comprising one layer. Click `Finish` to finalize the cleaving process.

The resulting cell still has a hexagonal shape in the x-y-plane, which can be inconvenient in MD simulations.

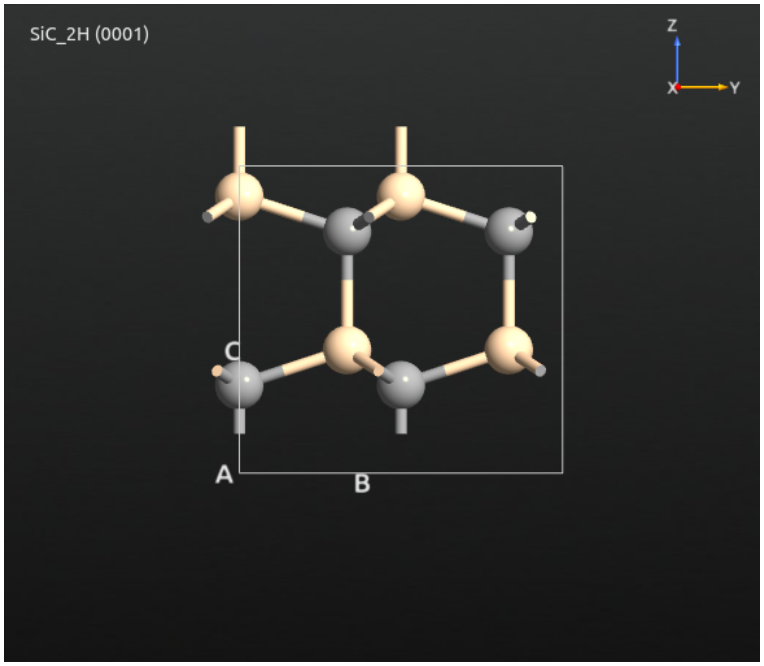


Due to the hexagonal symmetry, the surface lattice can easily be converted into an orthogonal lattice. To this aim open the *Bulk Tools* → *Lattice Parameters* widget, select to *Keep* → *Cartesian coordinates*, and set the x-component of the **B**-vector to zero.



Use the *Bulk Tools* → *Wrap* plugin to fold the atoms into the new simulation cell, and you will see that the crystal structures remains the same.

Due to the cleaving and wrapping, the termination of the top layer might not always be ideal. In this case, a termination with dangling, i.e. one-fold, coordinated carbon atoms is obtained, which most likely wont be stable in simulations. To obtain a better termination, shift the entire configuration by invoking *Coordinate Tools* → *Translate*, using a z-translation of -0.2 Å, and finally wrapping the configuration into the box via *Bulk Tools* → *Wrap*. The final termination should look like the following, featuring a more stable termination at bottom and top.



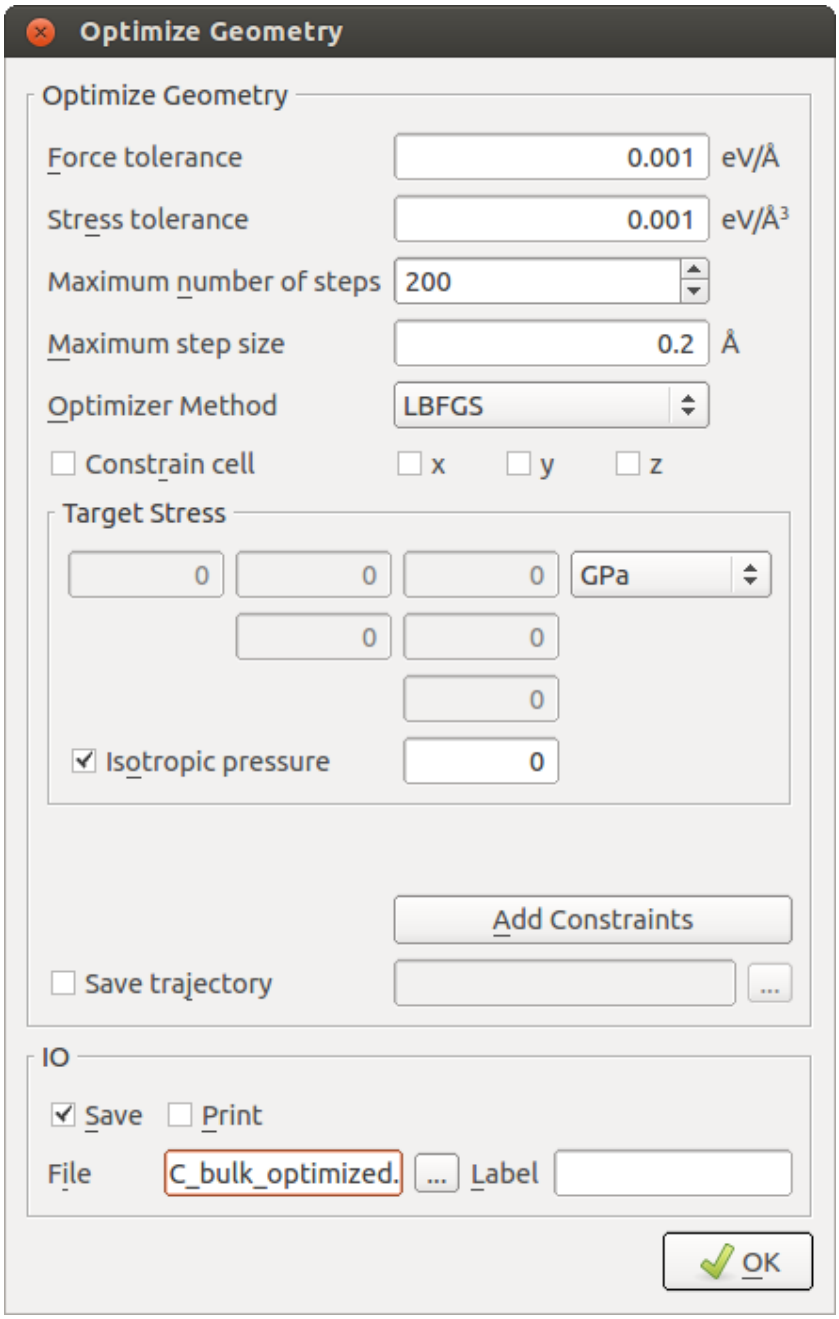
The next step is to create a supercell of suitable size via *Bulk Tools* → *Repeat*. Use a 3x3 repetition in the **A**-**B**-plane. The **C**-vector must be repeated in such a way that a substrate with sufficient crystal layers is created, as well as a reservoir containing enough of atoms to form the desired thickness of the deposited film. For the present example, use 3 layers for the substrate. To keep the simulation time reasonably short you will use only one layer for the reservoir, which gives a total number of 4 repetitions along he **C**-axis in total. If you later want to perform real production simulations, you may of course increase the number of reservoir layers (and the number of steps in the deposition simulation).

Now you need to mark the different parts of the system via tags. First select the upper two layers of the system to be the thermalized substrate, as shown in the schematic picture of the *reservoir* approach, and tag them with the name substrate. Then select the next lower crystal layer as the fixed bottom layer of the substrate, and assign the tag bottom. Finally, select the lowest crystal layer as the reservoir with the tag name reservoir.

Optimizing the Lattice Constants

Before building the slab configuration, the bulk system should be optimized under the potential used for the actual deposition simulation to obtain the optimal lateral lattice constants. To this aim send the prepared bulk crystal to the *ScriptGenerator*, and add a *NewCalculator* and an *Optimize* → *OptimizeGeometry* block. In the **New Calculator** settings, select **ATK- Classical** and choose *Parameter set* → *Tersoff_SiC_2005*[2]. Uncheck **Print** and **Save** and close the widget. In the **OptimizeGeometry** settings make the following changes.

[2] P. Erhart and K. Albe: Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide. Physical Review B, 71, 035211, (2005)



Run the simulation and drag the final optimized configuration from the **LabFloor** to the **Builder**.

Setting up the Simulation Cell

Finally, a surface has to be created from the bulk crystal. This can easily be achieved by increasing the height of the cell of the optimized bulk system. To this aim, open the *Bulk Tools* → *Lattice Parameters* widget. After the optimization, some of the off-diagonal cell components might be slightly different from zero. For convenience they should be reset to zero, while keeping the fractional coordinates constant. To adjust the height of the cell, make sure that you select *Keep* → *Cartesian coordinates*. Increase the z-component of the **C**-vector to 100.0 Å to obtain a sufficient vacuum padding between the active surface and the reservoir.

Setting up the Equilibration Simulation

Before you can run a deposition simulation, the surface has to be thermalized at the desired temperature. In the present example, you will operate at a substrate temperature of 2400 K, as suggested in[1]. Send the prepared configuration from the **Builder** to the **Script Generator**. Add a *NewCalculator* and an *Optimize* → *MolecularDynamics* block. In the **New Calculator** settings, make the same changes as in the previous optimization simulation. In the **MolecularDynamics** widget, select *Molecular Dynamics* → *Type* → *NVT Nose Hoover* thermostat, 100 000 steps, a log interval of 5000 steps, and enter a suitable trajectory filename. Select *Initial Velocity* → *Type* → *Maxwell-Boltzmann* initial velocities at 2400 K and set the *reservoir temperature* to 2400 K. (Note that in this case the *reservoir temperature* does *not* refer to the particle reservoir in the simulation cell, but denotes the temperature of the virtual heat reservoir, cf. *How to Setup Basic Molecular Dynamics Simulations*). Uncheck the **Save** and **Print** boxes.

Open the constraints widget by clicking the **Add constraints** button. Both the bottom layer of the substrate and the reservoir atoms should be fixed during the equilibration. This is done by selecting *Constraint* → *Fixed* constraints for these two tagged groups and closing the constraints widget.

Run the simulation via the **Job Manager** or in a terminal using `atkpython` and use the **Movie Tool** to send the final snapshot back to the **Script Generator**.

Setting up the Deposition Simulation

The deposition simulation consists of a normal, long MD simulation, which uses the *hook* functionality to perform a custom operation after each MD step (cf. *MDTrajectory* in the reference manual). In this case, at regular intervals an atom is taken from the reservoir and placed above the surface with a random thermal velocity, which is directed towards the active surface. This customized hook function has to be defined in the script using the Python programming language.

To set up the MD simulation in the **ScriptGenerator**, add a *New Calculator* and an *Optimize* → *Molecular Dynamics* block, and adjust the calculator settings as shown above. In the **Molecular Dynamics** widget, select the *NVT Nose Hoover* type, 800 000 steps, a log interval of 2500 steps, and a suitable trajectory filename. Set a reservoir temperature of 2400 K and select *Configuration Velocities* as initial velocities. Uncheck **Save** and **Print**. Send the simulation script to the **Editor** to make the necessary modifications.

Implementing the Deposition in the Script

At first you have to import the `wrap()` function to wrap coordinates back into the simulation cell.

```
from NL.CommonConcepts.Configurations.Utilities import wrap
```

Then you have to define the class that implements the hook function (the general procedure is similar to the tutorial about *Youngs modulus of a CNT with a defect*). First, define the class name and its constructor method.

```
# ------------------------------------------------------------
# Define a hook class to deposit atoms from the reservoir
# onto the active surface of the substrate.
# ------------------------------------------------------------

class VaporDepositionHook:
    def __init__(self,
                 deposition_interval,
                 configuration,
                 substrate_atoms,
                 vapor_temperature=300.0*Kelvin):
        """
        Constructor
        """
        # Store the parameters.

        # The interval after which a new atoms is deposited.
        self._deposition_interval = deposition_interval

        # Store the original coordinates. They will be used to constrain the reservoir␣
↪atoms.
        self._original_coordinates = configuration.cartesianCoordinates().
↪inUnitsOf(Angstrom)

        # The indices of the atoms that should not be deposited
        self._substrate_atoms = substrate_atoms

        # The vapor temperature.
        self._vapor_temperature = vapor_temperature

        # Deposition count.
        self._deposited_index = 0

        # Get the atoms in the reservoir, i.e. all with z-coordinates lower
        # than the fixed bottom of the substrate.
        self._lowest_substrate = self._original_coordinates[self._substrate_atoms, 2].
↪min()
        self._reservoir_indices = numpy.where(self._original_coordinates[:, 2] < self._
↪lowest_substrate - 0.1)[0].tolist()

        # Get the lattice vectors.
        cell = configuration.bravaisLattice().primitiveVectors().inUnitsOf(Angstrom)
        self._lx = cell[0, 0]
        self._ly = cell[1, 1]
        self._lz = cell[2 ,2]

```

In the constructor, you essentially store the needed parameters for the deposition, such as the interval at which a new atom is taken from the reservoir, and the original coordinates to constrain the atoms in the reservoir.

Finally, you have to define a method that is called whenever the hook function is invoked. Since our hook is implemented as a Python class, the `__call__()` method must be defined to make it behave like a function. All hook functions must take the arguments `step`, `time`, and `configuration`. This method should be implemented as follows.

```
    def __call__(self, step, time, configuration, forces, stress):
        """ Call the hook during MD. """
        # Get the coordinates.
        coordinates = configuration.cartesianCoordinates().inUnitsOf(Angstrom)

        # The reservoir is composed of all atoms that are below the bottom
        # layer of the substrate
        # or above the ceiling of the cell.
        self._reservoir_indices = numpy.where((coordinates[:, 2] < self._lowest_
↪substrate - 0.1) |
                                              (coordinates[:, 2] > self._lz))[0]

        # Freeze the reservoir atoms and reset their positions at every step.
        velocities = configuration.velocities()
        velocities[self._reservoir_indices, :] = 0.0*Angstrom/fs
        configuration._changeAtoms(indices=self._reservoir_indices,
                                   positions=self._original_coordinates[self._reservoir_
↪indices]*Angstrom)

```

At first, the cartesian coordinates are extracted form the current configuration. Then the atoms in the reservoir are determined based on whether their current position is either underneath the substrate or outside the upper cell boundary. For all reservoir atoms the velocities are set to zero, as we do not want them to move, in particular when the reservoir is increasingly eroded by removing atoms from it. Additionally, their positions are reset to the original positions at the beginning of the deposition simulation.

The next part contains the actual deposition of a new atom from the reservoir, and it is therefore only executed once at the beginning of each deposition interval.

```
        # Check if it is time for the deposition of a new atom from the reservoir.
        if (step % self._deposition_interval) == 0:
            # Get elements and velocities.
            elements = numpy.array(configuration.elements())
            velocities = configuration.velocities().inUnitsOf(Angstrom/fs)

            # If the reservoir is empty, do nothing.
            if len(self._reservoir_indices) == 0:
                return

            # Decide which element to deposit next.
            possible_elements = [Silicon, Carbon]
            element_index = self._deposited_index % 2
            next_element = possible_elements[element_index]
            reservoir_elements = elements[self._reservoir_indices]

            # Get the element candidates.
            possible_atoms = numpy.where(reservoir_elements == next_element)[0]

            # If not avaiblable, try the other element.
            if len(possible_atoms) == 0:
                next_element = possible_elements[element_index-1]
                possible_atoms = numpy.where(reservoir_elements == next_element)[0]

```

(continues on next page)

(continued from previous page)

```
            # Return back the global atom indices.
            possible_atoms = self._reservoir_indices[possible_atoms]

            # Pick an atom at the bottom of the reservoir
            lowest_atom = numpy.argmin(coordinates[possible_atoms, 2])
            next_index = possible_atoms[lowest_atom]

            # Place the deposition atom above the surface at a random lateral position.
            new_coords = numpy.array([numpy.random.uniform()*self._lx,
                                      numpy.random.uniform()*self._ly,
                                      self._lz - 15.0])

            configuration._changeAtoms(indices=[next_index], positions=new_
↪coords*Angstrom)
            wrap(configuration)

            # Draw random velocities for the atom according to
            # Maxwell-Boltzmann at the specified temperature.
            m = next_element.atomicMass()
            sig = self._vapor_temperature*boltzmann_constant/m
            sig = sig.inUnitsOf(Ang**2/fs**2)
            sig = numpy.sqrt(sig)

            new_velocity = numpy.random.normal(scale=sig, size=3)
            velocity_norm = numpy.linalg.norm(new_velocity)

            # Set the velocity so that it points towards the active surface.
            new_velocity = numpy.array([0.0, 0.0, -velocity_norm])
            velocities[next_index] = new_velocity

            # Set the new velocities on the configuration.
            configuration.setVelocities(velocities*Angstrom/fs)

            self._deposited_index += 1

# ------------------------------------------------------------
# End of hook class definition
# ------------------------------------------------------------

```

We start by determining which element has to be deposited next. As silicon and carbon are deposited in an alternating manner, this can easily be detected by querying whether the current deposition index is an odd or even number. Then all reservoir atoms of this element are selected and the atom which resides deepest at the bottom of the reservoir is picked. If for some reason no element of the requested type is available in the reservoir, we simply pick the other element instead. New random lateral coordinates, as well as a height at 15.0 Angstrom below the cell ceiling are created and assigned to the deposition atom via the `_changeAtoms()` method of the configuration. For better visibility, the coordinates are folded back into the cell using the `wrap()` function. To model the vapor temperature of the deposition atoms, new velocities are drawn from a Maxwell-Boltzmann distribution and, to ensure that the atom travels towards the surface, the velocity vector is re-aligned. For simplicity, the vector always points vertically to the surface, but in general, different impact angles can easily be implemented, as well. The new velocites are assigned to the deposition atom and the configuration velocities are updated using the `setVelocities()` method. Finally, the running deposition index is incremented, completing the class definition.

After the definition of the VaporDepositionHook class, a new instance is initiated, and the parameters for the current simulation are specified. This should be done immediately before the *Molecular Dynamics* block in the script.

```
# Initialize the hook class.
deposition_hook = VaporDepositionHook(deposition_interval=5000,
                                      configuration=bulk_configuration,
                                      substrate_atoms=substrate+bottom,
                                      vapor_temperature=2400.0*Kelvin)

```

To run the simulation, use the pre-defined MD block from the **Script Generator** with only two slight modifications. In the method definition of the *Nose–Hoover* thermostat, the reservoir_temperature parameter is modified, so that only the group of atoms marked by the substrate tag is coupled to the thermostat.

```
# Define an NVT thermostat where only the substrate atoms are thermalized.
method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=[('substrate', 2400*Kelvin)],
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=initial_velocity
)
```

This is necessary, to ensure that the dynamics of the deposited atoms in the gas phase are not spuriously modified by the thermostat before they have adsorbed to the surface (i.e. the deposited atoms should be in the NVE ensemble). Finally, the deposition_hook object must be passed as an argument to the MolecularDynamics() function.

```
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=bottom,
    trajectory_filename='SiC_deposit_tutorial_md.nc',
    steps=800000,
    post_step_hook=deposition_hook,
    log_interval=2500,
    method=method
)
```

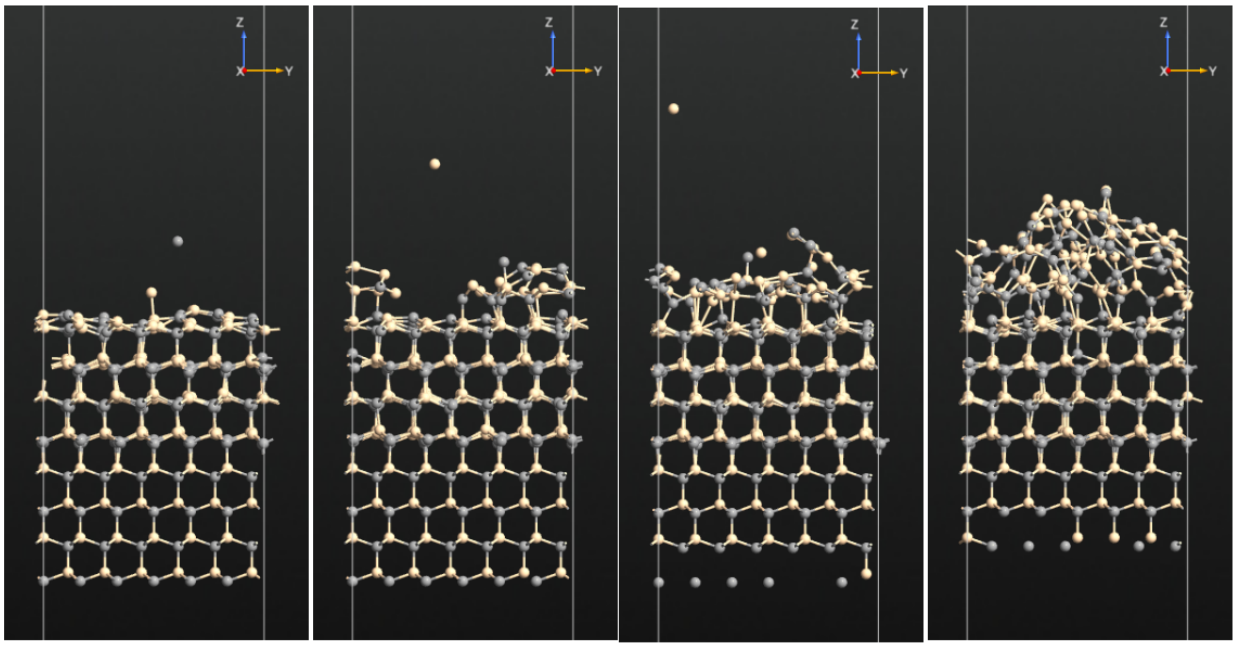
The complete script for the entire simulation can be found in the file Script_deposit_SiC.py.

Note: In QuantumATK version 2016, the NVT Nose–Hoover method is implemented in the NVTNoseHoover class, which is the one used in the script provided above. However, in QuantumATK version 2015 and earlier, that class is named NVTNoseHooverChain. The script provided above will therefore not work with QuantumATK 2015 and earlier. Use instead this script: Script_deposit_SiC_atk2015.py.

Running the Simulation

Run the script via the **Job Manager** or in a terminal. Such a long MD simulation might take a couple of hours to complete. You may notice that some atoms, instead of adsorbing to the surface, are scattered back into the vapor phase. This is not a problem, though, as these atoms, while travelling upwards, will eventually reach the reservoir, where they will automatically be treated as normal reservoir atoms again.

The following figure displays snapshots at different stages of the deposition simulation.



At first, only few atoms have been deposited onto the surface and adsorb forming single adatoms. With more and more atoms being deposited a first layer begins to form above the substrate surface. The first layer still resembles, at least partially, the crystalline structure of the substrate, whereas subsequent layers reveal a more amorphous structure. This must be attributed primarily to the employed potential (see the *note* at the beginning), but to some extent also to the high deposition frequency, as specified by the `deposition_interval` parameter. A larger value for this parameter will decrease the deposition frequency, which may allow for a more controlled integration of the deposited atoms. Remember, if you increase the deposition interval, you have to increase the total number of steps in the deposition simulation by the same factor to make sure all the atoms in the reservoir are being deposited.

Notice how at the same time as the film grows, the reservoir region at the bottom of the slab is depleted, as more and more atoms are taken away. However, the special constraints in the script make sure that the vanishing structure does not de-stabilize the entire system.

General Remarks

This simulation technique is, in principle, applicable to all kinds of materials. Of course, you might need to make several modifications in the script, such as adapting the elements. Instead of single atoms, you can also deposit entire molecules. For production simulations you may want to increase the number of layers in the reservoir to obtain a larger deposited film. Feel free to play around with other parameters, e.g. temperature, or the deposition interval. The deposition frequency, i.e. the inverse of the deposition interval, can be associated with the vapor pressure via the Hertz–Knudsen equation (cf. for instance[3]):

$$\frac{dN_i}{dt} = \frac{p_i A}{\sqrt{2\pi m_i k_B T}},$$

where p_i denotes the partial pressure of element/molecule *i*, *A* the surface area, and m_i the atomic/molecular mass.

Inserting the parameters from the simulation, you will find that the corresponding vapor pressure is around 66 bar, which is far away from the typical experimental near-vacuum conditions. This aspect represents a considerable shortcoming of such simulations, as the accessible simulation time in most cases prevents the use of realistic pressure values, rendering a *time-resolved* interpretation of the growth *kinetics* very difficult. Essentially, it means that the particles impact at a too high rate, which can, for instance, result in an increased local heating, as the reaction heat cannot be transported sufficiently fast from the surface to the thermostatted bulk substrate. However, the obtained *structural model* of deposited films can in most cases be considered a good approximation to the real film structure. You should bear in mind the conditions, though, under which the simulation has been performed and reconsider whether these conditions might have an influence on the deposited structure.

Furthermore, note that the use of a constant deposition interval provides another approximation, as the real process might rather follow a *Poisson distribution* than a uniform distribution. Although a Poisson distribution can easily be implemented into the script (using e.g. pythons `random.poisson` function), this would primarily make sense if the deposition interval can actually be chosen high enough to reproduce the desired real vapor pressure.

References

3.8.3 Simulating Ion Bombardment on Graphene Sheets

The properties of graphene sheets can be tuned by intentionally introducing defects to the material. A promising method to obtain defects in a controlled way is by bombarding the graphene sheet with high-energy ions. Molecular dynamics simulations can be used to elucidate the mechanisms involved in such processes and to improve our understanding of how external parameters, such as the kinetic energy of the incoming ion, affect the formation of defects. In this tutorial, you will simulate ion bombardment of graphene, following the protocol proposed in Ref.[1].

In particular, you will:

- Learn the basic steps required to set up the calculation using **QuantumATK**.
- Learn how to manually introduce modifications to the Python script generated by QuantumATK.
- Run the calculation using **QuantumATK**.

[3] K. Reuter: First-Principles Kinetic Monte Carlo Simulations for Heterogeneous Catalysis: Concepts, Status, and Frontiers. in *Modeling Heterogeneous Catalytic Reactions: From the Molecular Process to the Technical System*, Wiley-VCH (2011)

[1] EdsonăP. Bellido and JorgeăM. Seminario. Molecular Dynamics Simulations of Ion-Bombarded Graphene. *J. Phys. Chem. C*, 116(6):4044–4049, 2012. doi:10.1021/jp208049t.

Setting up the Graphene Sheet:

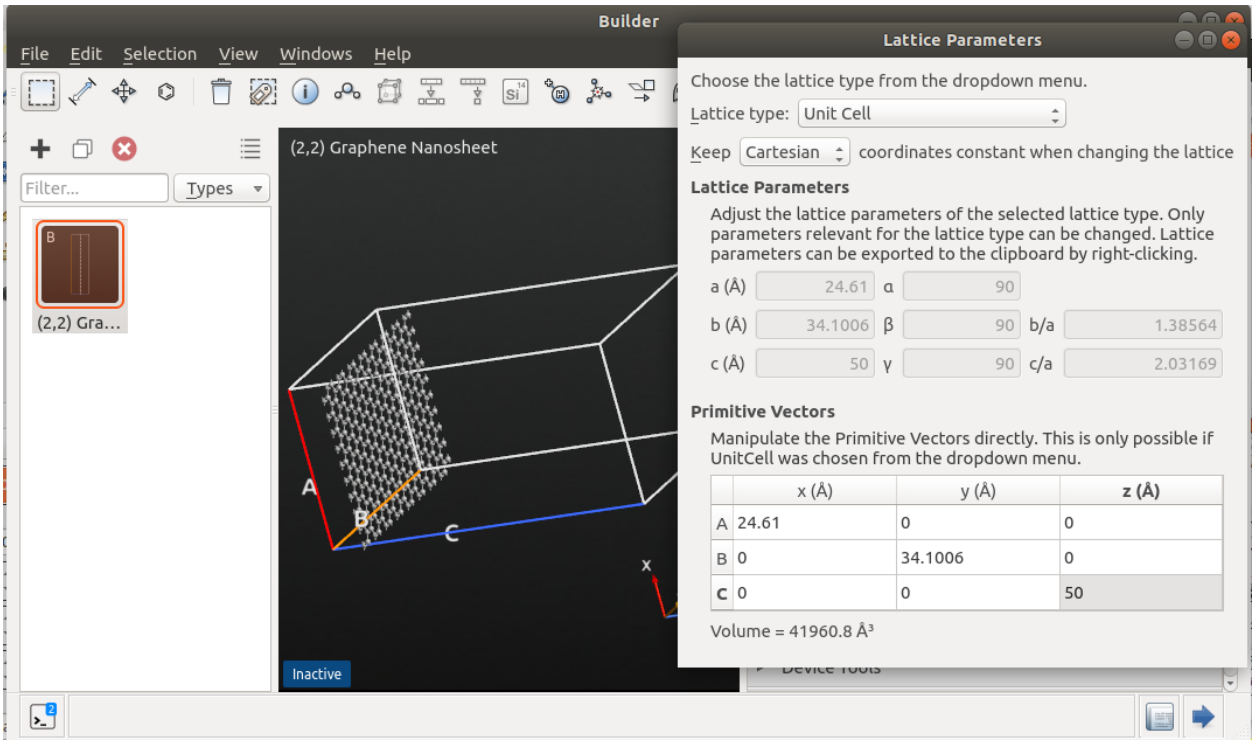
Open the **Builder** and do the following:

•Use the *Nanosheet* plugin via *Add* → *From plugin* → *Nanosheet* to create a graphene unit cell.

•Use the *Bulk Tools* → *Swap Axes* tool to rotate the structure and the cell vectors by clicking **C<->A** and then **Z<->X**. Now the surface is oriented along the Z-axis.

•Use *Bulk Tools* → *Repeat* to create a suitable supercell by giving a desired number of repetitions in the A- and B-directions, e.g. 10x4.

•Use the *Bulk Tools* → *Lattice parameters* widget to increase the z-component of the C-vector to 50.0 Å. Make sure to keep the cartesian coordinates when changing the lattice.



Adding a Bombardment Atom

Bombardment atoms can in principle be added at any location above the surface. In the following, you can choose to add a bombardment atom either above a graphene carbon atom (top site), or above the center of a graphene ring (hollow site).

•To add an atom at the top site:

–Select an atom in the center of the graphene sheet.

–Use *Coordinate Tools* → *Translate* to place the bombardment atom at the desired starting height above the graphene sheet (e.g. 30 Å), by entering a suitable z-component for the **translation vector**.

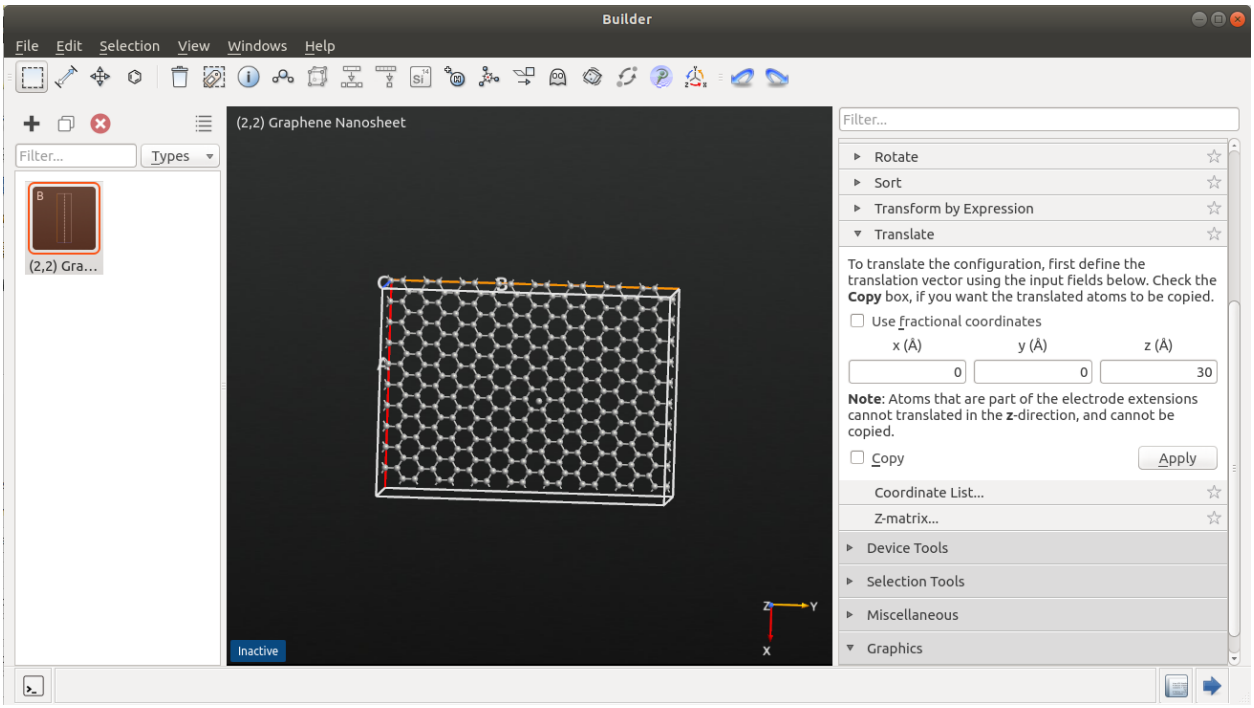
–Check **Copy** and click **Apply**.

•To add an atom in a hollow site:

–Select the atoms belonging to a carbon ring in the center of the sheet.

–Click the icon in the tool bar of the **Builder**.

–Translate the atom to the desired starting position above the surface by selecting the created atom and use the *Coordinate Tools* → *Translate* tool with the following settings.



Note: If you want to modify the exact position of the bombardment atom, you can do this easily, either by using the **Translate** tool, or by editing the coordinates of the last atom (using the *Coordinate Tools* → *Coordinate List* tool).

Setting up the Simulation

You will now set up the calculator for the simulation using the **Script Generator**. It is recommended to optimize the lattice of the graphene sheet with the classical potential before you start the simulations. Details about the calculator and the classical potential are explained below.

•First, send the structure from the **Builder** to the **Script Generator** by clicking the icon .

In the **Script Generator**:

•Add a *Calculators* → *ForceFieldCalculator* block.

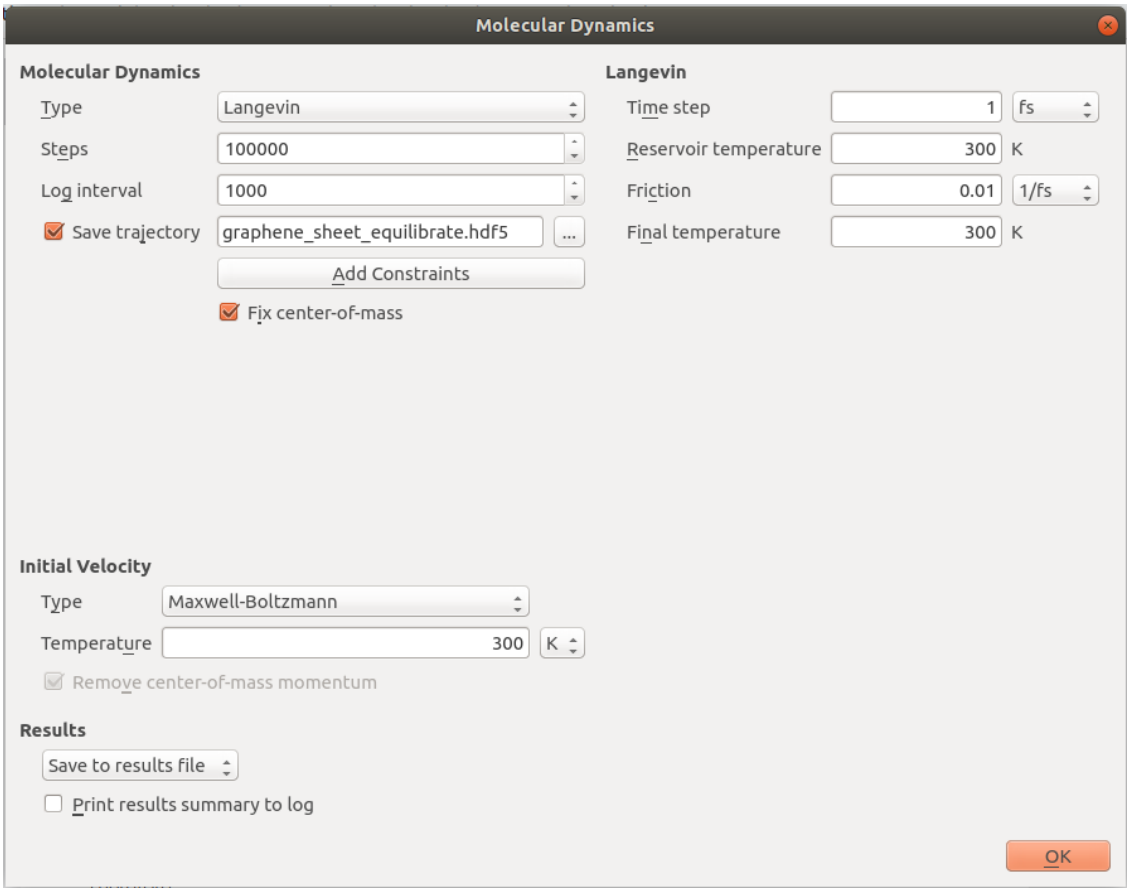
•Add two *Optimization* → *MolecularDynamics* blocks.

•Open the **ForceFieldCalculator** widget. Select the *Tersoff_SiC_1998* parameter set, then click **OK**.

Attention: The *Tersoff_SiC_1998* potential is very similar, but not identical, to the potential used in[Page 1178, 1]. If you feel like using the exact same potential, you can find it in this script: `ion_bombardment.py`.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

•In the first **MolecularDynamics** block, set up the equilibration simulation with the following parameters.

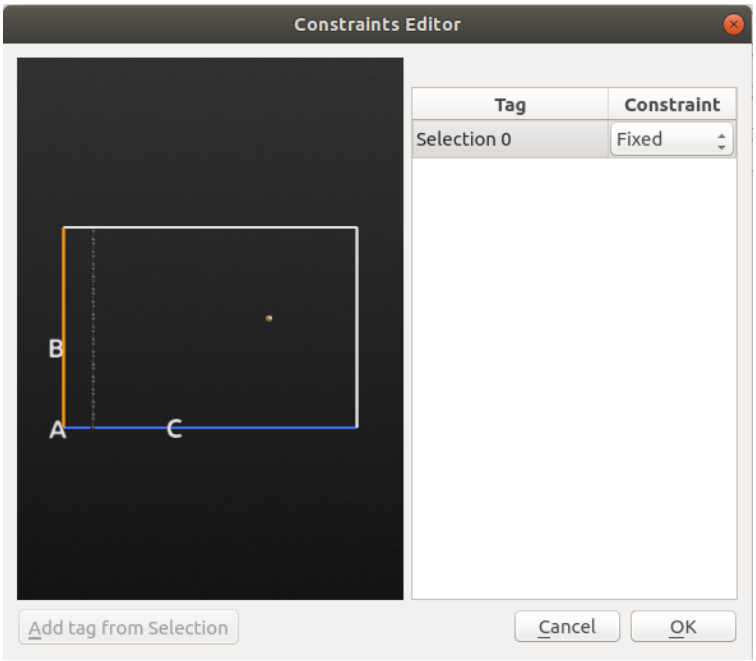


The position of the bombardment atom should be fixed during the equilibration calculation. Therefore:

•In the **MolecularDynamics** block, click **Add Constraints** to open the **Constraints** widget.

•Select the bombardment atom above the surface and click **Add tag from Selection**.

•Set the constraints field to **Fixed** and click **OK**.



In the second **MolecularDynamics** block, you should set the basic parameters of the bombardment simulation:

•Select the **NVE Velocity Verlet** type.

•Set 50000 steps and a log interval of 100.

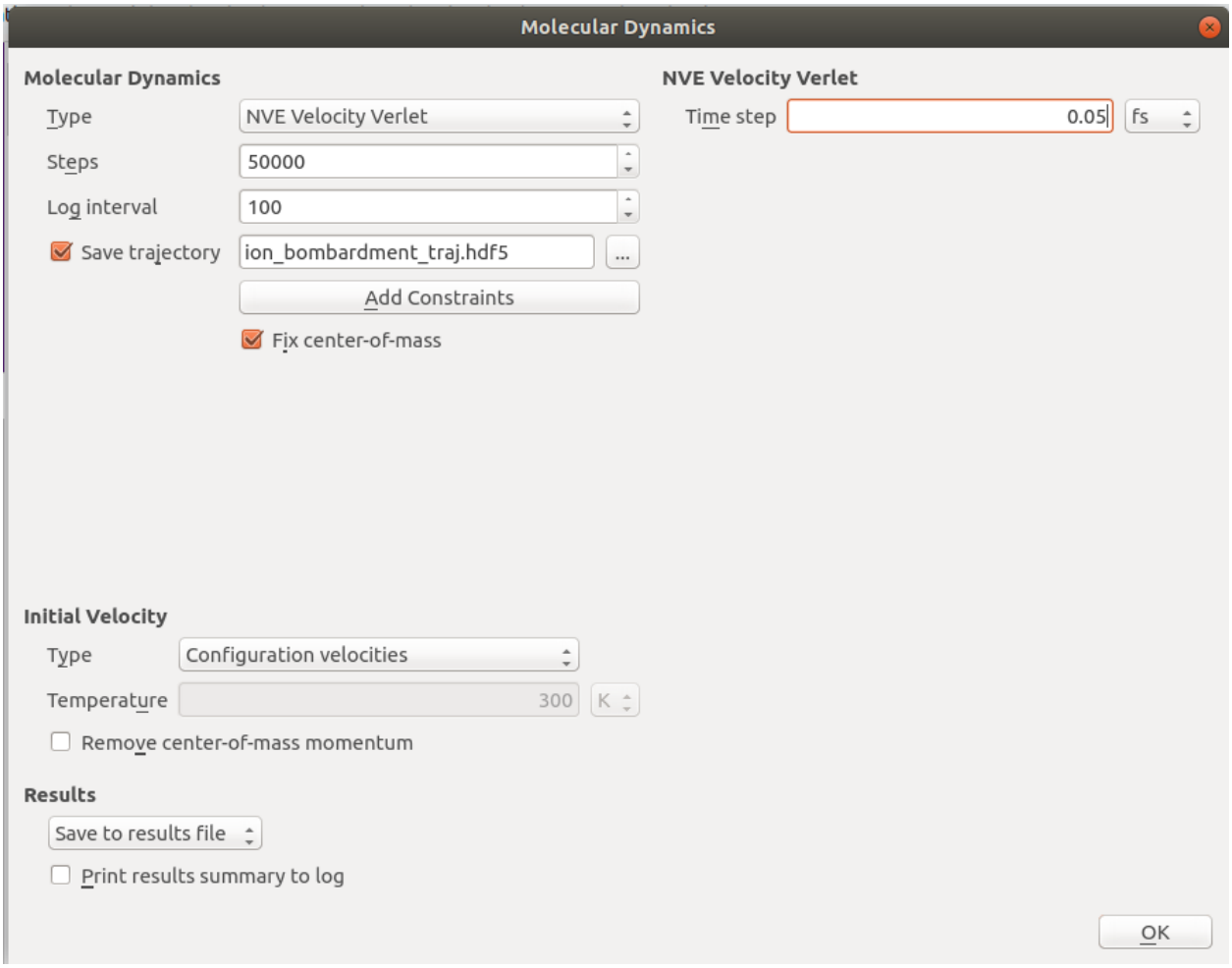
•Enter a suitable filename for the trajectory.

•Set the initial velocities to **Configuration velocities** to use the velocities after equilibration.

•Uncheck the **Remove center-of-mass momentum** box.

•Decrease the **time step** to 0.05 fs.

Note: The kinetic energy of the atom is expected to be very high in this simulation. That is why a relatively small **time step** is necessary to guarantee the convergence. For very high incident energies (>100 eV), the time step may have to be decreased even further to 0.01 fs (cf.Page 1178, 1).



•Send the script to the **Editor**, using the icon.

Modifying the Script

To let the bombardment atom approach the surface at the desired kinetic energy, a few modifications have to be made in the script.

•Right before the second **MolecularDynamics** block, insert the following lines.

```
# Get the velocities from the bulk_configuration
velocities = bulk_configuration.velocities()
# Define the incident energy of the bombardment atom.
incident_energy = 10.0*eV
# Calculate the corresponding velocity and apply it to the last atom.
incident_velocity = ((2.0*incident_energy/Carbon.atomicMass())**0.5).inUnitsOf(Ang/fs)
velocities[-1, :] = [0.0, 0.0, -incident_velocity]*Ang/fs
bulk_configuration.setVelocities(velocities)
```

Note: The velocity vector is set to be perpendicular to the surface in this calculation. Arbitrary incident direction can

be considered by modifying the velocity vectors in the script lines.

- Save the script under a suitable filename, and run the simulation via the **JobManager** or in a terminal.
- After the simulation has completed, you can visualize the trajectory using the **MovieTool** or the **Viewer** to inspect the damage.

References

3.8.4 Uniaxial and Biaxial Stress in Silicon

Introduction

In this tutorial you will learn how to use **QuantumATK** to study the electronic properties of silicon under uniaxial and biaxial stress.

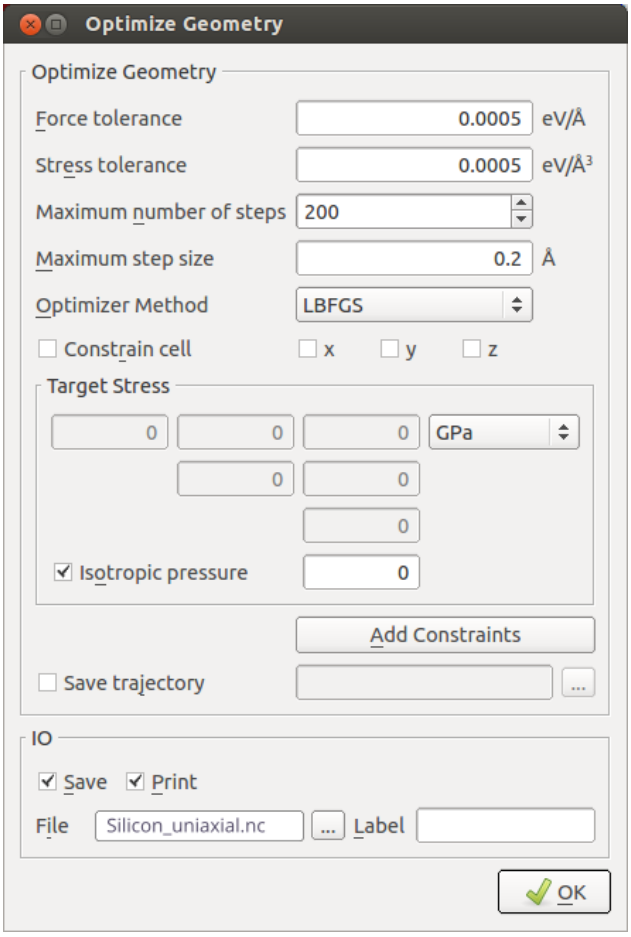
In particular, you will learn how to use the `Target Stress` option for a geometry optimization (**Optimize Geometry**) to apply a specific stress to a crystal. Then you will calculate and analyze the electronic band structure and effective masses in the strained system.

An important aspect of this tutorial is the symmetry of your crystal after the stress is applied. You will have to pay particular attention to this point, and to this end you will find the **Brillouin Zone Viewer** plugin very useful.

Uniaxial Stress

Set up the Calculation

1. In the **Builder**, import a silicon fcc bulk structure from the database and send it to the **Script Generator**.
2. Add a `New Calculator`:
   - use the default LDA exchange-correlation potential and select a 9x9x9 k-point sampling;
   - set the density mesh cut-off to 150 Hartree to have a better description of the silicon electronic structure.
3. Add `OptimizeGeometry` and set these parameters in order to perform a full optimization of the bulk structure:
   - set force and stress tolerances to 0.0005 eV/Å and 0.0005 eV/Å^3;
   - uncheck the `Constrain cell` but keep the target stress at zero.



1.Add `Bandstructure` analysis:

•choose 201 points per segment;

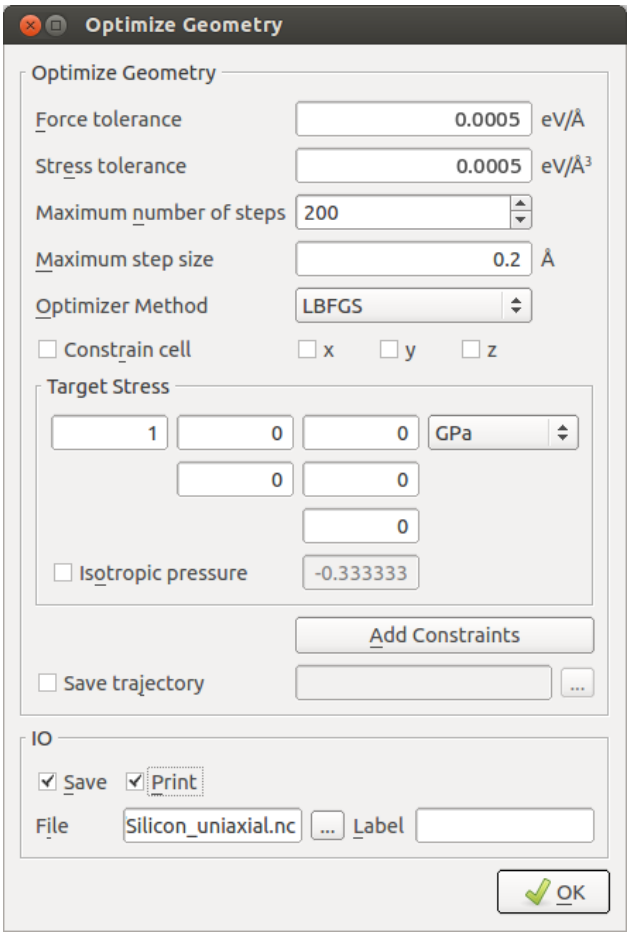
•choose the L, G, X path.

2.Add `OptimizeGeometry` and set these parameters in order to apply a uniaxial stress:

•set force and stress tolerances to 0.0005 eV/Å and 0.0005 eV/Å^3;

•uncheck `Constrain cell`;

•uncheck `Isotropic Pressure` and add a target stress of 1 GPa for the x component of the stress tensor.



Note: The definition of target stress (`target_stress`) is:

- if a single value p is given, it will be interpreted as an external pressure value from which the internal target stress tensor is calculated as $\sigma = \begin{pmatrix} -p & 0 & 0 \\ 0 & -p & 0 \\ 0 & 0 & -p \end{pmatrix}$;
- if a target stress tensor is given, it will be interpreted as the *internal* stress of the system, meaning that a negative entry on the diagonal will result in a compression in the corresponding direction and vice versa. Note that the stress tensor is symmetric, so its only necessary to define the upper triangle.

Pay attention to the fact that the sign convention is different in these two cases!

---

3. Add `Bandstructure` analysis:
   - choose 201 points per segment;
   - choose the L, G, X path.

Note: If a target stress tensor is given, the BravaisLattice of the configuration is automatically converted into a UnitCell, to enable changes in the shape of the cell.

As described below, the high symmetry points of the Brillouin zone will change, since the cell will no longer be fcc after the stress is applied. However, at this point in the setup, the symmetry of the new cell is not known, so you will

have to modify the symmetry points by hand in the Python script.

4. Send the script to the **Editor** and locate the last `Bandstructure` analysis block. Replace X by B in the route; see *the next section* for a detailed explanation on the modified Brillouin zone.

```
# -----------------------------------------------------------
# Bandstructure
# -----------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['L', 'G', 'B'],
    points_per_segment=201,
    bands_above_fermi_level=All
    )
nlsave('Silicon_uniaxial.nc', bandstructure)
```

You can download the full script here: `Silicon_uniaxial.py`.

5. Save and send the script to the **Job Manager** to run the calculations, which should take less than a minute.

Symmetry Considerations

Before analyzing the electronic structure of strained silicon you will need to understand how the Brillouin zone and the notation of high symmetry points are dealt with by **QuantumATK**.

From the **LabFloor** drag and drop the optimized structure (`gID000`) and the strained structure (`gID002`) to the **Builder**.

Plot the Brillouin zone of each structure with the *Bulk Tools->Brillouin Zone Viewer*:

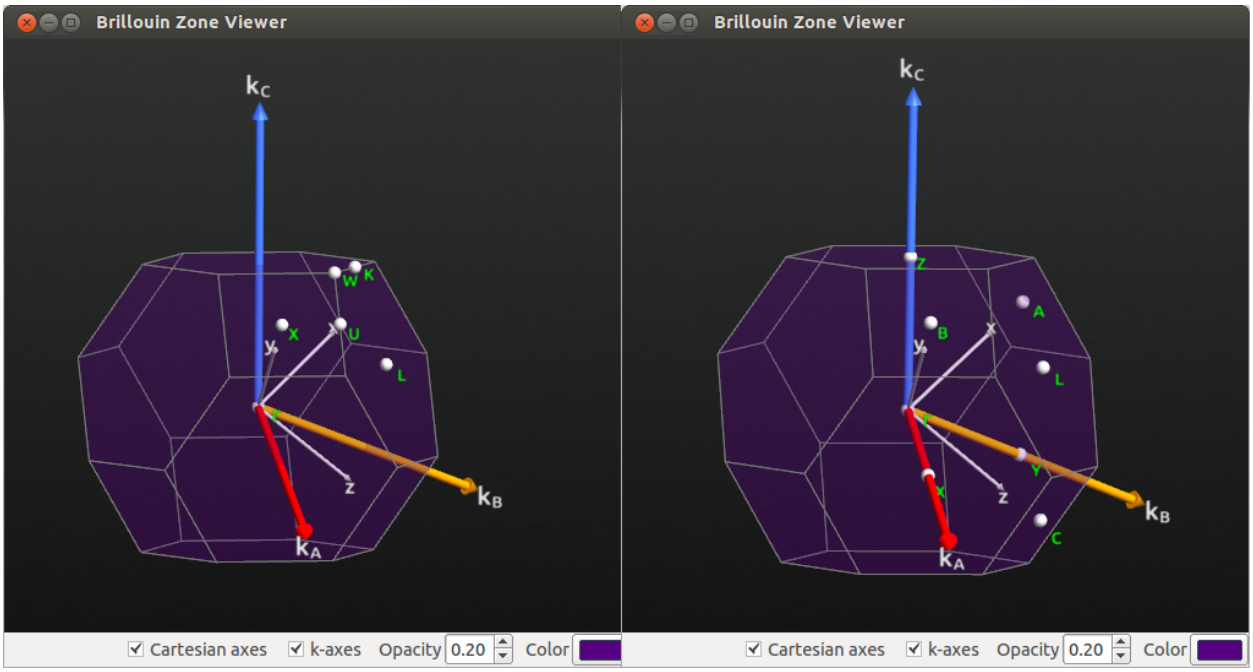
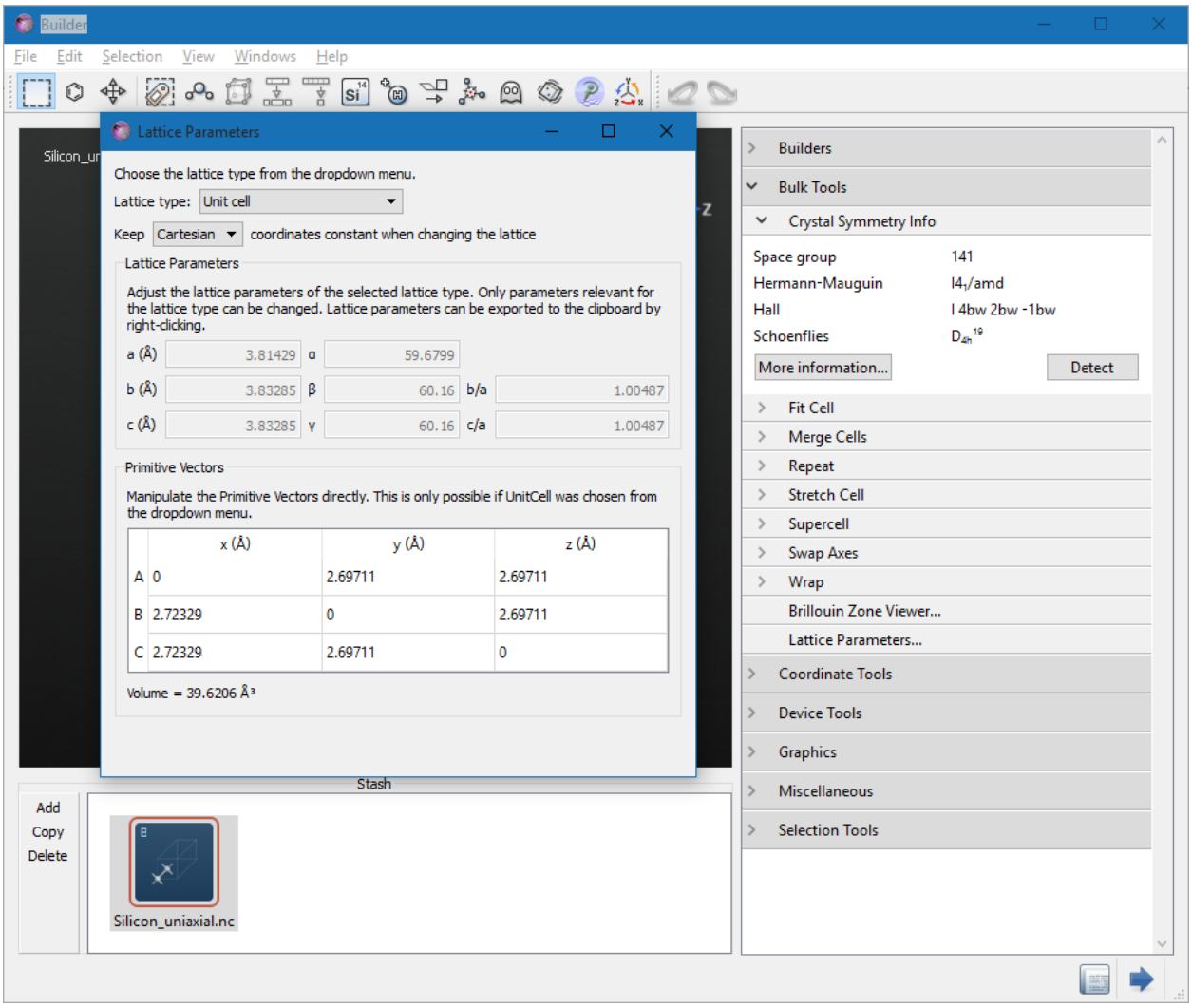


Fig. 3.93: Brillouin zone for fcc bulk silicon when `FaceCenteredCubic` Bravais lattice is used (left). When the Bravais lattice is changed to `UnitCell` type, the notation for high symmetry points is changed according to the figure (right).

The high symmetry points in the UnitCell representation of the strained fcc crystal are related to the fcc symmetry points as follows:

- A, B, C all correspond to the X point in the unstrained fcc Brillouin zone.
- By increasing the stress you can see that B and C are still degenerate, but A is not degenerate with B and C. This is logical in the tetragonal symmetry.
- The L point is still called L, and degenerate with X, Y, Z in the UnitCell notation.

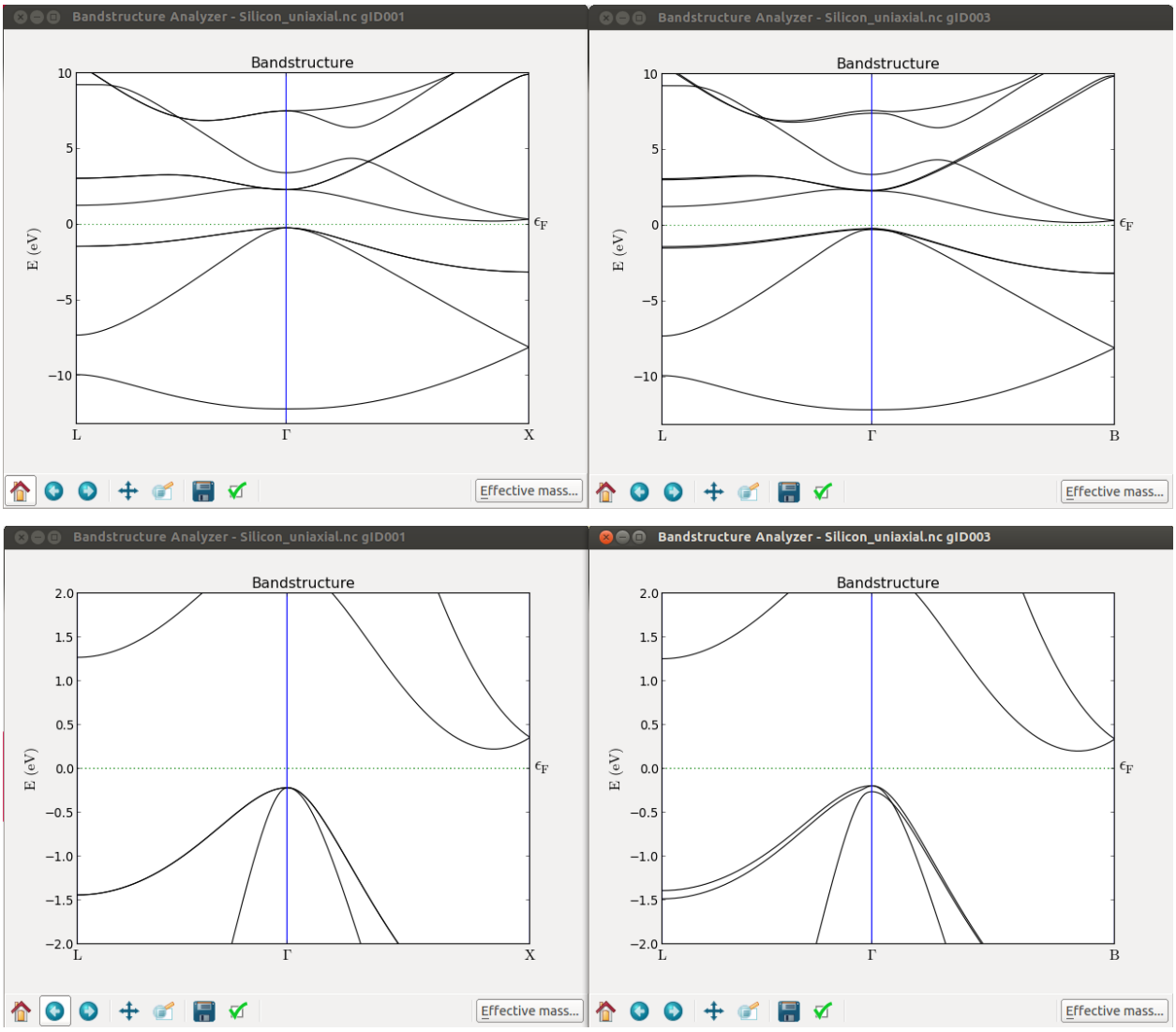
The symmetry of the crystal is actually, as expected, tetragonal (space group 141) after the uniaxial stress is applied, which can be verified by activating the strained crystal on the stash and using *Bulk Tools->Crystal Symmetry Information*, and clicking **Detect**. On inspection of the lattice parameters, you see that, again as expected, all 3 lattice vectors are elongated in the X axis, and contracted in Y and Z (elastic response, Poisson effect), due to the uniaxial deformation.



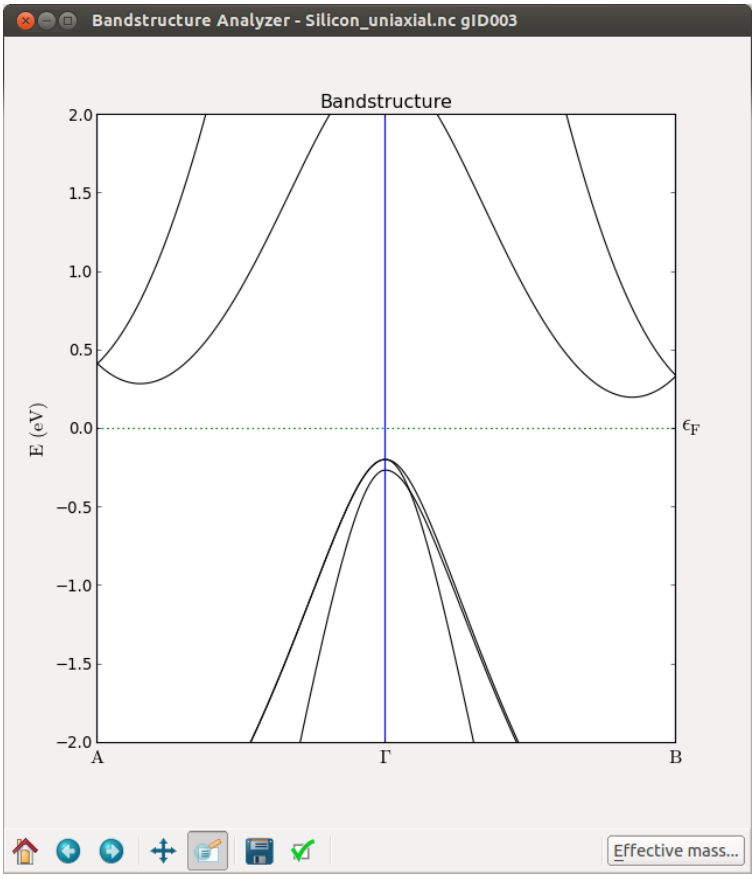
Analyze the Results

Electronic Band Structure

Now plot the band structure of the optimized cell and the uniaxial stressed cell. From the **LabFloor** select the calculated `Bandstructure` object and plot them with the `Bandstructure Analyzer` plugin.



From these plots you can immediately see that, at Gamma, the top of the valence band splits. It is however, more interesting to see that the Δ valley is not degenerate anymore. To see this effect more clearly you can calculate the band structure along the A-G-B path.



By further inspecting the band structure, you can conclude the following about uniaxially strained Si:

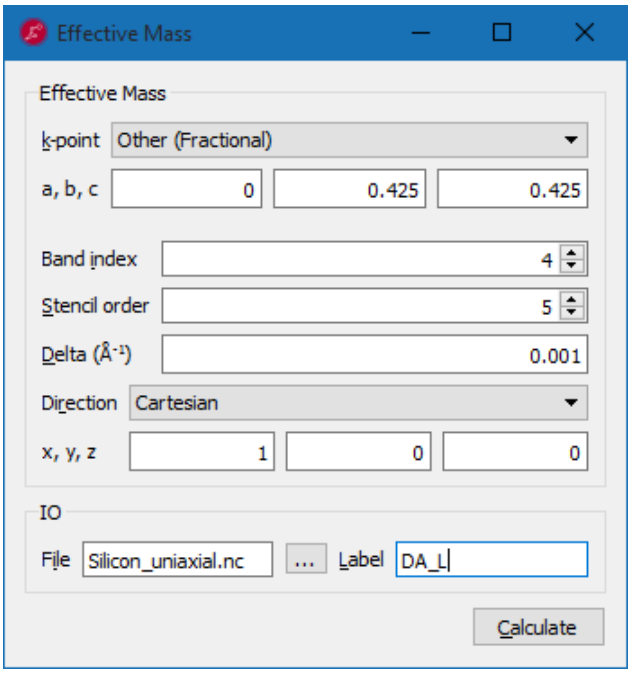
- L remains degenerate
- The 6-fold Δ valleys split into 2+4

The absolute values of the band gaps are however not correct in this simulation, since the LDA exchange-correlation functional was used for simplicity.

Effective Masses

It is also very interesting to consider the effects of strain on the effective mass.

From the band structure plot of the strained structure, click on the `Effective Mass` button.



Following the example in the figure above calculate the effective masses for the Δ valleys, band index 4, along different directions:

•Longitudinal mass at Δ_A: fractional coordinate [0, 0.425, 0.425], along the [1,0,0] cartesian direction;

•Transverse mass (1) at Δ_A: fractional coordinate [0, 0.425, 0.425], along the [0,1,0] cartesian direction;

•Transverse mass (2) at Δ_A: fractional coordinate [0, 0.425, 0.425], along the [0,0,1] cartesian direction;

•Longitudinal mass at Δ_B: fractional coordinate [0.425, 0, 0.425], along the [0,1,0] cartesian direction;

•Transverse mass (1) at Δ_B: fractional coordinate [0.425, 0, 0.425], along the [1,0,0] cartesian direction;

•Transverse mass (2) at Δ_B: fractional coordinate [0.425, 0, 0.425], along the [0,0,1] cartesian direction;

You will find the following results:

•Longitudinal mass at Δ_A (2-fold degenerate): 0.91

•Transverse masses at Δ_A: 0.185 (no split)

•Longitudinal mass at Δ_B (4-fold degenerate): 0.898

•Transverse masses at Δ_B: 0.188 and 0.186

So, while all longitudinal and transverse masses are very close to the original Δ valley masses (0.903 and 0.186, which you can compute from unstrained crystal), you see splittings due to the broken symmetry.

For a more detailed explanation on the calculation of silicon effective masses please refer to the tutorial *Effective mass of electrons in silicon*.

As an exercise, you are encouraged to study the masses at the L point, and investigate possible symmetry breakings.

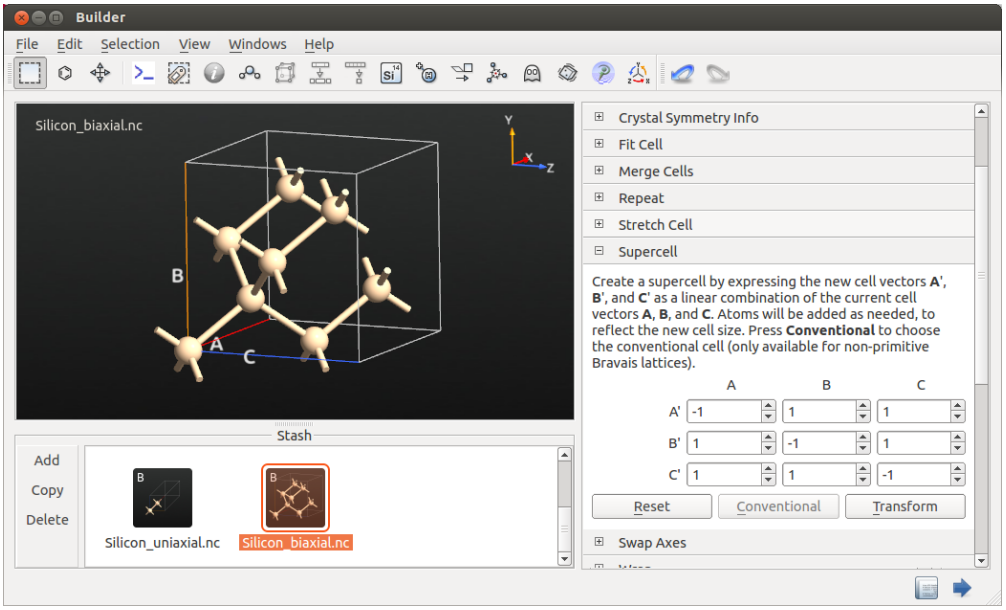
Biaxial Stress

Take the optimized (unstrained) silicon structure, send it to the **Scripter** and set `New Calculator`, `OptimizeGeometry`, and `Bandstructure` as you did in the previous section.

To apply a biaxial stress, set the xx and yy components in the stress tensor to 1 GPa in the `Target Stress` field in the `Optimize Geometry` dialog.

Before running the calculation, send the script to the **Editor** and modify the Brillouin zone path to L, G, B as described above and run the calculation.

Also in this case the cubic lattice distorted is to a tetragonal symmetry. To see that clearly, drag and drop the optimized stressed structure to the **Builder** and apply the supercell transformation as indicated in the figure below.



From *Bulk Tools->Lattice Parameters* you can see the distorted lattice parameters of 5.44 Å and 5.38 Å. Thus, effectively the biaxial stress is equivalent to a uniaxial stress along the [001] direction, with opposite sign.

3.8.5 Adding, Combining, and Modifying Classical Potentials

Introduction

The new **ATK-ForceField** package provides a lot of pre-defined classical potentials covering a broad range of elements and systems. However, if you are interested in a particular potential, which is not yet part of this pre-defined set, you can easily add these parameters and create your own potential, as long as the underlying potential functions are supported. You can also modify single parameters of existing potentials in order to tune the potential to a special reference system. Furthermore, you can combine different potentials to cover a broader range of elements. This tutorial provides a brief overview over this functionality of **ATK-ForceField**.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

For a detailed overview and documentation of the available potential classes and functions in **ATK-ForceField** and the underlying **TremoloX** package, take a look at the TremoloX section of the ATK Reference Manual. There you will find definitions for all functions along with the corresponding parameters.

Note: Be aware that you should always read the underlying publication carefully, if you use a new potential, to ensure that the potential is suited for your desired application.

If possible, you should compare some benchmark values obtained with the classical potential to corresponding experimental values or the results of suitable higher-level calculations, e.g. DFT simulations. This holds in particular for the combination of different potentials, as they might have been parameterized in a certain context and they are not always guaranteed to work in entirely different environments.

Adding a New Classical Potential from Scratch

In this section, you will learn how to add a new parameter set, taken from a potential in the literature, to the **ATK-ForceField** calculator. You will be guided through the complete setup of a potential from scratch to explain the various contents of a **TremoloX** potential set. In practice, it is often more convenient to start from a similar potential, send it to the **Editor** with `Script details` set to Show defaults to report all details of the potential, and then merely edit the parameters in the **Editor**.

As a first example, you will consider a simple titanium oxide crystal. One of the most popular classical potentials for titania is the one proposed by Matsui and Akaogi (MA)[1], as it reproduces the essential properties of both crystal and amorphous titania systems quite well. Later on, you will set up a potential designed for the simulation of amorphous oxide systems[2].

[1] M Matsui and M. Akaogi: Molecular dynamics simulations of the structural and physical properties of the four polymorphs of TiO2. Mol. Sim. 6, 239 (1991)

[2] C.J. Fennell and J.D. Gezelter: Is the Ewald summation still necessary? Pairwise alternatives to the accepted standard for long-range electrostatics. J. Chem. Phys. 124, 234104 (2006)

Setting up the System

To set up a suitable test system, open the **Builder**, click *Add* → *From database*, search for TiO2, and select and add the *Rutile* crystal structure.

You can now send the configuration to the **ScriptGenerator** to set up the desired optimization, molecular dynamics, or analysis script. For now, you can select any arbitrary calculator as a place holder. A typical simulation script could consist of *Calculator*, *Optimization* → *OptimizeGeometry*, and *Analysis* → *ElasticConstants* blocks. For accurate results, set the maximum force and maximum stress values in the **OptimizeGeometry** settings and the maximum force value in the **ElasticConstants** settings to 0.001 eV/Å (respectively eV/Å^3). Furthermore, disable the `Constrain cell` checkbox in the **OptimizeGeometry** widget to optimize the cell parameters as well.

Send the script to the **Editor**.

Setting up the Potential

In the **Editor**, locate the block where the potential is defined (i.e. everything after the **BulkConfiguration** and before the **OptimizeGeometry** block) and delete it.

To define a new TremoloX potential you first need to initialize a new *TremoloXPotentialSet* object. You can do this by adding the following line:

```
potentialSet = TremoloXPotentialSet('MatsuiAkaogi_TiO')
```

The string in parethesis defines the name of the potential, but it does not have any influence on its functionality.

As the potential set is currently empty, you need to add all necessary components. You should start with adding the *ParticleType*. In this case, add one titanium and one oxygen particle type by inserting these two lines into the script:

```
potentialSet.addParticleType(ParticleType(symbol='O', mass=15.9994*atomic_mass_unit,␣
↪charge=-1.098))
potentialSet.addParticleType(ParticleType(symbol='Ti', mass=47.867*atomic_mass_unit,␣
↪charge=2.196))
```

The partial charges are taken from Ref.[Page 1193, 1]. The MA potential consists of two interaction types:

1. A Tosi-Fumi (often also called Buckingham) type potential (i.e. a combination of an exponential, repulsive contribution and an attractive van der Waals part)
2. Electrostatic coulomb interactions, which model the partial ionization of the atoms.

The first part can be added by using the *TosiFumiPotential*, which implements the potential function

$$V(r) = Ae^{B(\sigma - r)} - \frac{C}{r^6} - \frac{D}{r^8}\,.$$

If you compare this form to the equation given in Ref.[Page 1193, 1], you will notice that the last term is not required, implying that D must be set to zero. Take care with the energy unit in Ref.[Page 1193, 1], which is kJ/mol. The parameters must be given in that unit or converted to electron volt. The A, B, C, D, and σ parameters can be set as follows:

```
potentialSet.addPotential(TosiFumiPotential(particleType1='O',
                                            particleType2='Ti',
                                            A=0.811696*kiloJoulePerMol,
                                            B=5.154639*Angstrom**-1,
                                            sigma=2.8171*Angstrom,
                                            C=1215.0*kiloJoulePerMol*Angstrom**6,
                                            D=0.0*eV*Angstrom**8,
```

(continues on next page)

(continued from previous page)

```
                                                r_i=6.0*Angstrom,
                                                r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential(particleType1='O',
                                                particleType2='O',
                                                A=0.979056*kiloJoulePerMol,
                                                B=4.27350*Angstrom**-1,
                                                sigma=3.2678*Angstrom,
                                                C=2916.0*kiloJoulePerMol*Angstrom**6,
                                                D=0.0*eV*Angstrom**8,
                                                r_i=6.0*Angstrom,
                                                r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential(particleType1='Ti',
                                                particleType2='Ti',
                                                A=0.644336*kiloJoulePerMol,
                                                B=6.49351*Angstrom**-1,
                                                sigma=2.3664*Angstrom,
                                                C=506.25*kiloJoulePerMol*Angstrom**6,
                                                D=0.0*eV*Angstrom**8,
                                                r_i=6.0*Angstrom,
                                                r_cut=7.5*Angstrom))
```

The first two arguments in each potential determine which two particle types the potential is supposed to act between. The additional parameters $r_{*\mathrm{i}*}$ and r_{cut} determine the range and cutoff behavior. In this case, the potential follows the original form for distances below 6 Å, whereas for larger distances a smoothing function is invoked for ranges up to 7.5 Å. At larger distances these interactions are assumed to be zero.

The electrostatic interactions are taken into account by adding a suitable Coulomb solver. All Coulomb solvers use the partial charges that you have defined for the particle types. The default algorithm in **ATK-ForceField** is *CoulombDSF* employing a cutoff and a damped shifted force (DSF) to smooth the transition[Page 1193, 2]. To achieve a high precision with this method (which can be important particularly for static calculations, such as elastic constants, phonons, or lattice constants) the cutoff has to be chosen sufficiently high (e.g. 15 Å). Here, we will instead use the smooth-particle-mesh-Ewald method (*CoulombSPME*), which automatically takes care of the long-ranged contributions of the Coulomb interactions. This Coulomb solver is added via the line:

```
potentialSet.setCoulombSolver(CoulombSPME(r_cut=9.0*Angstrom))
```

Finally, you have to initialize the actual *TremoloXCalculator* and pass the created potential set as argument. Following that, you have to attach the calculator to the bulk configuration:

```
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)
```

The remaining part of the script can be left as it is. An example script can be found in the file `TiO2_Matsui.py`.

Now, you can send the script to the **JobManager** and start the calculation. The results should be consistent with Ref.[Page 1193, 1].

A Potential for Amorphous Oxides

In this example, another potential is introduced, which is designed to simulate amorphous materials, such as titanium, silicon, hafnium, or tantalum oxide, as well as their mixed oxides[3].

Note: From **QuantumATK 2015** on, this potential is already available as the pre-defined potential *Trinastic_HfOSiTaTi_2013*.

To test the potential, you can, for instance, use the rutile configuration from the previous example. The potential is similar to the MA potential, shown above, but additionally it includes a *MorsePotential*.

$$V(r) = E_{ij}\left[\left(1 - e^{-k_{ij}(r-r_0)}\right)^2 - 1\right]$$

In fact, the formula presented in Ref.[Page 1196, 3] omits the the last contribution in the squared brackets. However, as this term only represents a constant offset in the energy, it has no influence on the forces in an MD simulation, and we shall therefore neglect this discrepancy here. Although the potential terms could in principle be set up similar to the previous example by using the TosiFumiPotential, we will present an alternative approach by adding all terms individually.

To obtain the potential, you need an exponential repulsive contribution, which will be taken from the *BuckinghamPotential*, a dispersion term, for which you will use a *LennardJonesMNPotential*, and finally a *MorsePotential*.

The potential set up is demonstrated for the example of the Ti-O interactions. The other elements follow the same scheme by inserting the corresponding element-pair-specific potential parameters from Ref.[3].

Start by initializing a new *TremoloXPotentialSet* and adding the particle types with the corresponding partial charges:

```
potential_set = TremoloXPotentialSet('Trinastic_JCP2013')
potential_set.addParticleType(ParticleType(symbol='O', mass=15.9994*atomic_mass_unit,␣
↪charge=-1.2))
potential_set.addParticleType(ParticleType(symbol='Ti', mass=47.867*atomic_mass_unit,␣
↪charge=2.4))
```

Then define the repulsive Buckingham term:

```
potential_set.addPotential(BuckinghamPotential(particleType1='O',
                                               particleType2='Ti',
                                               A=5105.12*eV,
                                               rho=0.2253*Angstrom,
                                               r_i=6.0*Angstrom,
                                               r_cut=7.5*Angstrom))
```

Add the dispersion term. You will only need the attractive part of the Lennard-Jones potential, i.e. the A-parameter has to be set to zero:

```
potential_set.addPotential(LennardJonesMNPotential(particleType1='O',
                                                   particleType2='Ti',
                                                   A=0.0*eV*Angstrom**12,
                                                   B=20.0*eV*Angstrom**6,
                                                   m=12,
                                                   n=6,
                                                   r_cut=10.0*Angstrom))
```

[3] J.P. Trinastic, R. Hamdan, Y. Wu, L. Zhang, H.-P. Cheng: Unified interatomic potential and enery barrier distributions for amorphous oxides. J. Chem. Phys. 139, 154506 (2013)

Add the Morse potential:

```
potential_set.addPotential(MorsePotential(particleType1='O',
                                          particleType2='Ti',
                                          E_0=0.3478*eV,
                                          k=1.9*Angstrom**-1,
                                          r_0=1.96*Angstrom,
                                          r_i=6.5*Angstrom,
                                          r_cut=7.0*Angstrom))
```

Finally, set up a Coulomb solver, e.g. *CoulombSPME* or *CoulombDSF* to calculate the electrostatic interactions, and initialize a new TremoloXCalculator with this potential set:

```
potential_set.setCoulombSolver(CoulombSPME(r_cut=9.0*Angstrom))
calculator = TremoloXCalculator(parameters=potential_set)
```

In Ref.[3], an alternative strong parameterization is proposed, which can be implemented by inserting the corresponding parameters instead of the ones shown above. The implementation of the interactions between the remaining element pairs works similarly. From the parameter table in Ref.[Page 1196, 3], you will find that for some element combinations the Morse energy parameter is zero, which means that no Morse potential is required between these elements.The full potential can be found in the file `script_Trinastic_potential.py.txt`.

Combining a Tersoff and a Lennard-Jones Potential

In this section, you will learn how you can combine Tersoff and Lennard-Jones (LJ) potentials, to simulate a single lithium atom in a graphite cell. This combination of materials is particularly interesting, as it provides a very simple model of a typical anode in lithium ion batteries. The combination of Tersoff and LJ-potentials is typically used for multi-component materials where the individual components interact only weakly with each other, predominantly via repulsive and van der Waals forces. The strong bonded interaction within a component, in this example graphite, can then be described by a Tersoff potential, or another suitable potential for that material.

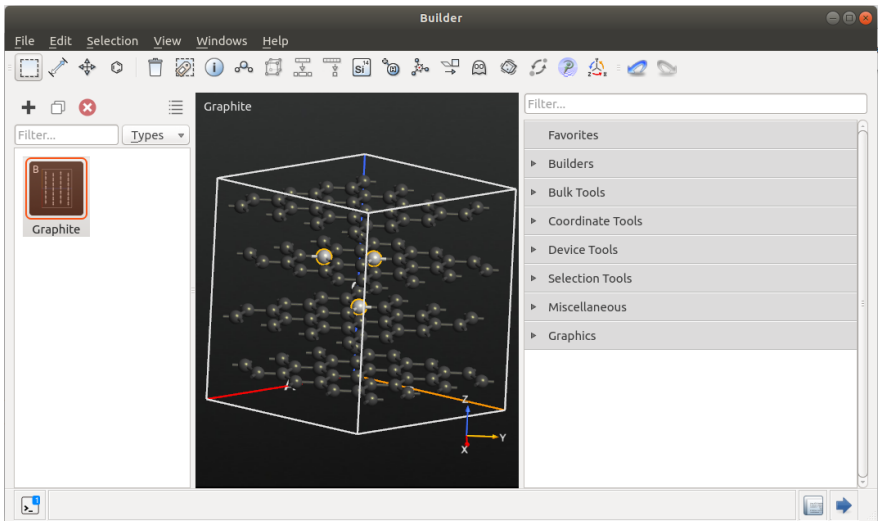
In this example, you will adopt the parameters of Ref.[4]. Be aware that they have not been tested extensively for this system. Therefore, they should only be considered a starting point for your own investigations.

Setting up the System

To set up a test system, open the **Builder** and add a graphite cell via *Add* → *From database*. Increase the size of the cell by clicking *Bulk tools* → *Repeat*, and make a 4x4x2 repetition of the unit cell.

Select a triangle of 3 atoms (one in the second layer, two in the third layer) as shown in the picture.

[4] O. N. Kalugin, V. V. Chaban, O. V. Prezhdo: Microscopic Structure and Dynamics of Molecular Liquids and Electrolyte Solutions Confined by Carbon NanoTubes: Molecular Dynamics Simulations. Carbon Nanotubes - Synthesis, Characterization, Applications (2011), Siva Yellampalli (Ed.).



Place an additional atom in the center of the selected atoms, by clicking on the icon in the toolbar (see picture). This atom will later be converted into lithium, but for now we will retain it as carbon.

Send the configuration to the **ScriptGenerator** using the button. In the **ScriptGenerator**, add a *Calculators* → *ForceFieldCalculator* object, and then the simulation task of your choice, e.g. an *Optimization* → *MolecularDynamics* block. Open the **ForceFieldCalculator** settings, and select the *Tersoff_C_2005* potential (this will be the base potential for the graphite). Under Results, select Do not save and uncheck `Print results to summary log`. In the *Output Settings*, set `Script Detail` to Show defaults, and send the script to the **Editor**.

At first, locate the element list. Change the name of the last element in the list from `Carbon` to `Lithium`. (If you had done this already in the **Builder**, it would not have been possible to select the Tersoff potential in the **ScriptGenerator**, since it doesnt support lithium.)

If you now take a look at the calculator section, you will find the detailed definition of the Tersoff potential. This is due to the *Show defaults* option. Locate the line where the C element definition is added via the `addParticleType()` method. Make a copy (to invoke two `addParticleType()` calls), and modify the relevant lines as shown below:

```
potentialSet.addParticleType(ParticleType(symbol='C',
                                          mass=12.0107*atomic_mass_unit,
                                          charge=None,
                                          # Sigma and epsilon from Ref [3].
                                          sigma=3.3611*Ang,
                                          epsilon=0.004207*eV))
potentialSet.addParticleType(ParticleType(symbol='Li',
                                          mass=6.941*atomic_mass_unit,
                                          charge=None,
                                          # Sigma and epsilon from Ref [3].
                                          sigma=0.826*Ang,
                                          epsilon=0.271115*eV))
```

For carbon, the LJ-parameters epsilon and sigma have been added and set to the values taken from Ref[4]. For lithium, you first have to change the symbol and the atomic mass and then set the corresponding LJ-parameters. Note that in this case, we consider neutral lithium atoms, i.e. without defining a partial charge, as this is the preferred state in the anode (in contrast to lithium ions in the cathode).

The LJ-potential does not become active by specifying the particle parameters alone. You additionally have to add the corresponding potential function. To this aim, add the following two lines:

```
potentialSet.addPotential(LennardJonesPotential(particleType1='C',
                                                particleType2='Li',
                                                r_cut=9.0*Ang))
potentialSet.addPotential(LennardJonesPotential(particleType1='Li',
                                                particleType2='Li',
                                                r_cut=9.0*Ang))
```

This will add LJ-potentials between carbon and lithium, as well as between lithium and lithium. Note that there is no LJ-potential between carbon atoms, as these interactions are sufficiently defined by the Tersoff-potential (the LJ would just be a tiny correction).

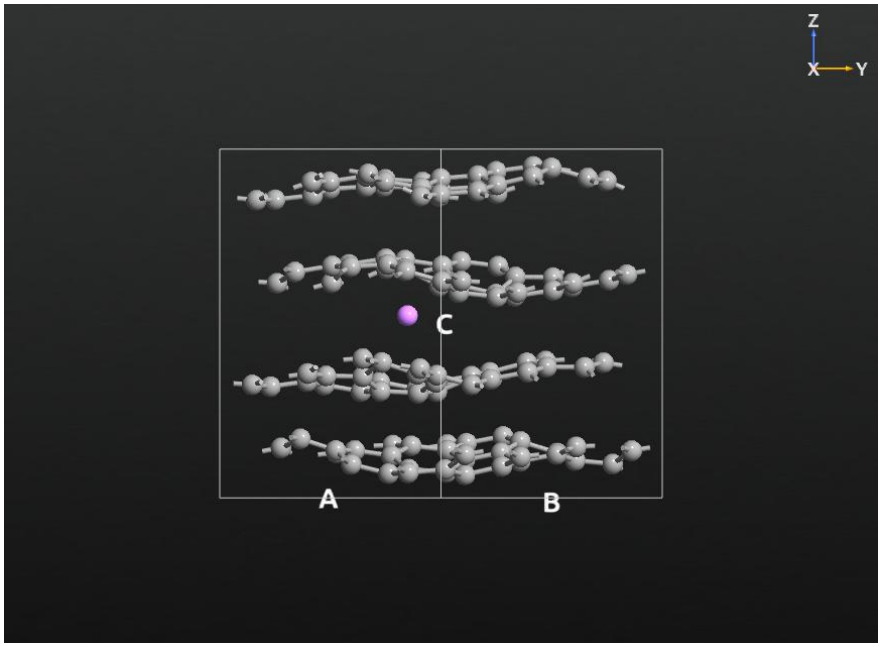
The Lennard-Jones potential in **ATK-ForceField** is defined as

$$V_{ij}(r) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right]$$

and uses the combination rules to obtain σ_{ij} and ϵ_{ij} parameters between different elements that you can find in the *LennardJonesSplinePotential* documentation page.

You should always check whether the combination rules recommended for the chosen LJ-parameters agree with this definition. If not, you should use the more general *LennardJonesMNPotential*.

An example of the final script can be found in `Script_graphite_lithium.py`. You can add any optimization, molecular dynamics, or analysis block, compatible with **ATK-ForceField**. This picture shows a snapshot of an MD simulation of the lithium-graphite system.

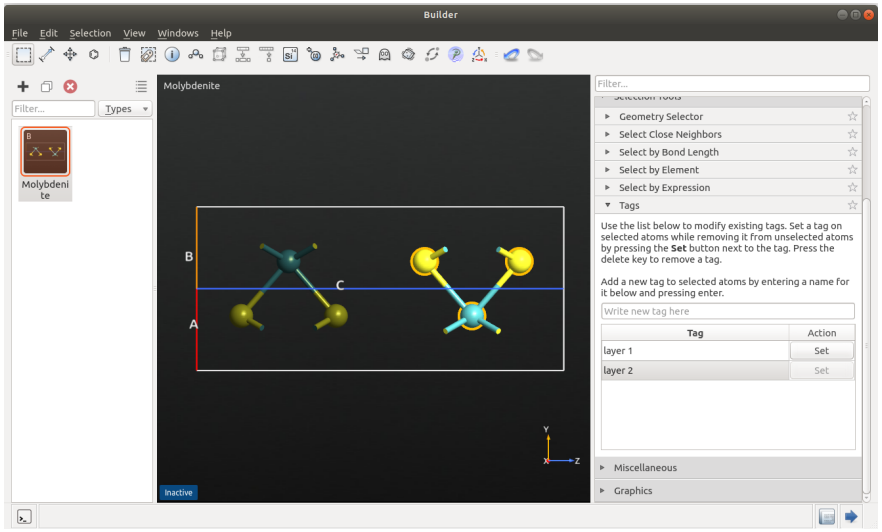


Intra- and Inter-Layer Cohesion in MoS_2

Note: If you are using **QuantumATK 2014**, you should make sure that you have ugraded to **QuantumATK 2014.3** (or later) to run this part of the tutorial, as earlier versions may cause some problems.

In this last example you will learn how to define a potential only on the selected atoms, using tags. We consider a molybdenite (MoS_2) crystal where the intralayer interactions are described by a *Stillinger-Weber* potential, while the comparably weaker inter-layer interactions are again modelled by a *Lennard-Jones* potential, as described in Ref.[Page 1197, 4].

To set up the system, open the *Builder* and search for and add the molybdenite structure from the database. Select one layer by drawing a rectangle around it and add a tag with the name layer1 (via *Selection Tools* → *Tags*). Repeat the procedure for the second layer, using the name layer2.



Send the configuration to the **ScriptGenerator**.

In the **ScriptGenerator** add a *Calculators* → *ForceFieldCalculator* and an *Optimization* → *OptimizeGeometry* block. Open the **ATK-ForceField** calculator and select the *StillingerWeber_MoS_2013* potential. Under Results, select Do not save and close the widget. Open the **OptimizeGeometry** widget and adjust the setting for a geometry and stress optimization, as described in the first example.

Send the final script to the **Editor**.

In the python-script, locate the line where the Stillinger-Weber-potential is defined. Copy the line, and adjust both lines in the following way:

```
sw_layer1 = StillingerWeber_MoS_2013(tags='layer1')
sw_layer2 = StillingerWeber_MoS_2013(tags='layer2')
```

This will define each potential only between the atoms belonging to the corresponding tagged group. Furthermore, you have to define a Lennard-Jones potential, which acts only between the sulfur atoms of different groups. This can be done by adding the following lines of code:

```
# Define a new potential for the interlayer interaction.
lj_interlayer_potential = TremoloXPotentialSet(name="InterLayerPotential")
# Add particle type definitions for both types.
lj_interlayer_potential.addParticleType(ParticleType.fromElement(Molybdenum))
lj_interlayer_potential.addParticleType(ParticleType.fromElement(Sulfur, sigma=3.
```

(continues on next page)

(continued from previous page)

```
↪13*Angstrom, epsilon=0.00693*eV))
# Add Lennard-Jones potentials between the sulfur atoms of different layers.
lj_interlayer_potential.addPotential(LennardJonesPotential('S', 'S', r_cut=10.0 *␣
↪Angstrom))
lj_interlayer_potential.setTags(['layer1', 'layer2'])
```

The Lennard-Jones parameters are taken from Ref.[5]. The last line ensures that the Lennard-Jones-potential only acts between alternating layers.

Finally, all three potential sets have to be combined in the TremoloX-calculator. To accomplish this, add the following lines:

```
# Combine all 3 potential sets in a single calculator.
calculator = TremoloXCalculator(parameters=[sw_layer1, sw_layer2, lj_interlayer_
↪potential])
```

After defining the potential, the script continues as usual, by adding the calculator to the bulk configuration and performing the geometry optimization. The final script can be found in the file `Molybdenite_SW_plus_LJ.py` for comparison.

If you run the script, you will obtain optimized lattice constants, which should be very similar to the ones reported in Ref.[Page 1201, 5].

References

3.8.6 Generating Amorphous Structures

Introduction

Amorphous solid materials become more and more important for a broad spectrum of technologies. In contrast to crystal structures, which are usually well- defined, it is, however, rather difficult to obtain the amorphous structure of a given material in order to perform atomistic simulations. This tutorial gives you some guidelines about generating amorphous structures in **QuantumATK** using molecular dynamics (MD) simulations at different levels of precision.

Amorphous structures, i.e. structures without any periodic order of the building atoms or molecules, can be found in various kinds of materials, e.g. glass or polymers. Their mechanical and electronic properties often differ quite substantially from the respective crystal. For example, the thermal expansion coefficient of amorphous silica is about one order of magnitude lower than the corresponding value for quartz-crystals.

Consequently, there is a growing interest in the employment in amorphous materials, to tailor various material and device properties in a desired way. In this effort, atomistic simulations can help to understand the behaviour of amorphous strucures at the nano- and microscale.

It is, however, not always straightforward to obtain a suitable amorphous structure of a given material for such simulations. In contrast to the corresponding crystal structure there is no unique unit cell or supercell representation, as the structure inherently bears large degree of randomness. Instead, one should ideally consider an ensemble of realizations of such structures, which is in reality often prohibited by the computational cost of the required calculations.

In this tutorial, you will learn how you can generate amorphous structures that can serve as representative examples and input configurations for further simulations, based on DFT or classical force field methods.

[5] J.-W. Jiang, H.S. Park, T. Rabczuk: Molecular dynamics simulations of single-layer molybdenum disulphide (MoS_2): Stillinger-Weber parameterization, mechanical properties and thermal conductivity J. Appl. Phys. 114, 064307 (2013)

You will learn how to use classical force field MD simulations to obtain a first starting structure of the amorphous material, which can in a second step be refined using semi-empirical or DFT molecular dynamics, when the focus is set on electronic properties.

As there is in general no universally valid recipe to create amorphous structures, the most suitable protocol for a given material depends on a lot of factors, such as the availability of a reliable empirical potential, the desired size of the supercell, the disposable computational resources, etc. This tutorial will therefore only provide a guideline, rather than a rigid protocol, to generate amorphous structures, which might have to be modified according to the particular application at hand.

You will simulate amorphous silicon dioxide, often referred to as *fused silica* or *fused quartz*, which is one of the most widely used amorphous materials, e.g. in conventional or specialized glass and also in the semiconductor industry.

You will also learn how to create interface structures between crystal and amorphous regions.

If you dont have much experience in MD simulations, you can start with the *How to Setup Basic Molecular Dynamics Simulations*, which will help you to become familiar with the MD functionality in **QuantumATK**.

Note: The simulations in this tutorial use a comparably small number of steps to demonstrate within a reasonable time the principle of generating amorphous structures. For actual production simulations, in order to achieve more reliable amorphous samples with a low amount of structural defects, you should use much longer simulation times. Typically, a melt-quench cycle comprises at least several hundred picoseconds, or even several nanoseconds. Take a look at the references of this tutorial to get an overview over common simulation protocols.

Amorphous Structure Generation with Classical MD Simulations

The most common way to generate amorphous structures is to mimic the experimental protocol of melting a crystal and cooling it down again by using MD simulations[1],[2]. Although cooling rates accessible in MD simulations can not be chosen as low as the experimental cooling rates, this procedure gives satisfactory results in many cases. The quality of the resulting amorphous structures of course depends much on the ability of the employed force field to describe the material, also further away from equilibrium conditions.

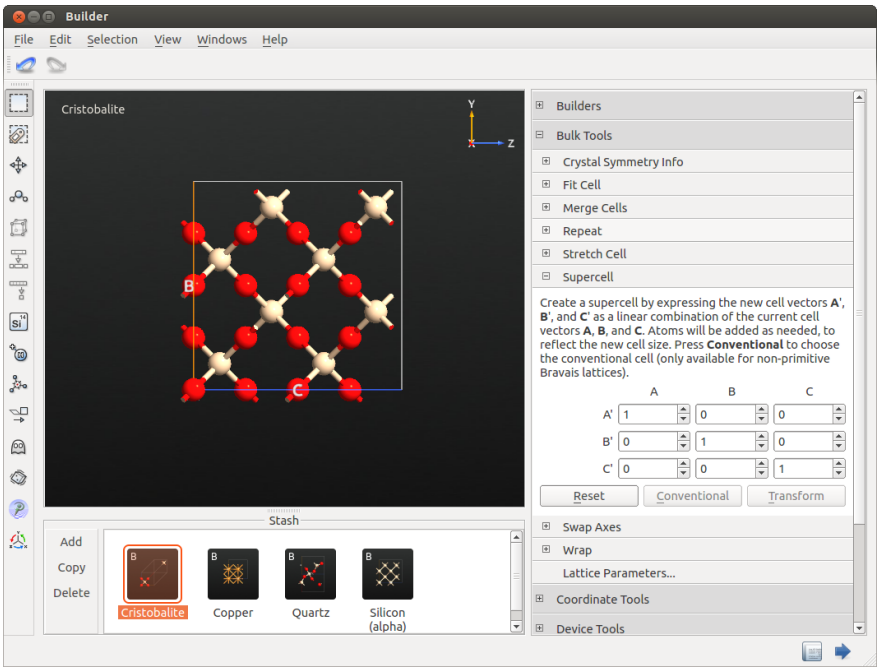
Setting up the Geometry

You will start from a crystalline SiO_2 system, specifically a cristobalite structure. In contrast to e.g. quartz, the cristobalite unit cell can easily be transformed into an orthogonal cell, which makes the simulation much more convenient. Moreover, the density of cristobalite is very similar to the experimental density of amorphous silica, being 2.2 g/cm^3.

Open the Builder and click *Add* → *From Database* and search for sio2. Select the cristobalite structure and add it to the **Stash**. To transform it into an orthogonal cell, click on *Bulk Tools* → *Supercell* on the right-hand-side panel. Click `Conventional` and then `Transform`. For cristobalite this will create a cubic supercell.

[1] E.ăA. Chagarov and A.ăC. Kummel. Generation of Realistic Amorphous Al2O3 And ZrO2 Samples By Hybrid Classical and First-Principle Molecular Dynamics Simulations. *ECS Transactions*, 16:773, 2008. URL: http://kummelgroup.ucsd.edu/pubs/papers/76.pdf.

[2] D.ăJ. Cole, M.ăC. Payne, G.ăCsányi, S.ăM. Spearing, and L.ăColombi Ciacchi. Development of a classical force field for the oxidized Si surface: Application to hydrophilic wafer bonding. *J. Chem. Phys.*, 127:204704, 2007. URL: http://dx.doi.org/10.1063/1.2799196.



The resulting cell is too small yet to be a useful container for an amorphous structure, as the periodic boundary conditions would still enforce a certain amount of crystallinity. Therefore, repeat the cell by opening *Bulk Tools* → *Repeat*, and apply a *3 x 3 x 3* repetition. This cell, containing 648 atoms, will be the starting structure for the MD simulations.

Setting up the Simulation

Send the configuration to the **ScriptGenerator**. Add a *NewCalculator* and an *Optimize* → *MolecularDynamics* block to the script panel.

Double-click on the **NewCalculator** block and select the **ATK-ForceField** calculator with the *Pedone_Fe2_2006* potential[3]. Uncheck `Print` and `Save`. Close the **NewCalculator** widget by clicking `Ok`.

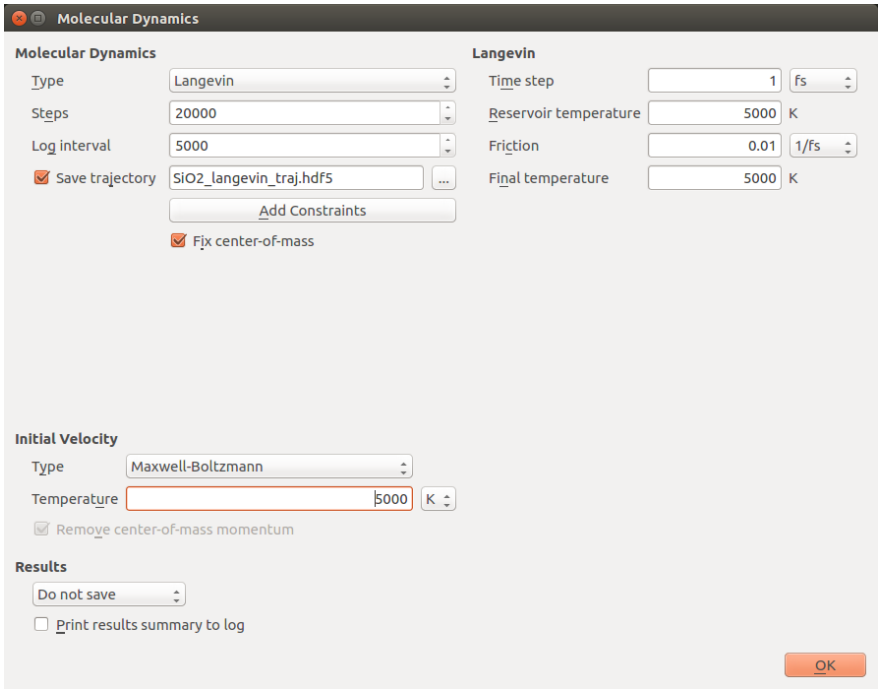
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

Now double-click on the **MolecularDynamics** block. To melt the crystal, you will start with simulations at a very high temperature of 5000 K. As the primary purpose of this initial simulation is to randomize the arrangement of the atoms, no pressure coupling is needed at this stage. Therefore you should select a `Type`, which gives you an *NVT* ensemble (take a look at the *How to Setup Basic Molecular Dynamics Simulations* for a more detailed documentation of MD simulation types in **ATK**). A good choice is the *Langevin* thermostat, as it produces a very tight coupling to the heat bath and therefore prevents single atoms from becoming too hot and possibly destabilizing the system.

Set the `Steps` to 20 000, the `Log interval` to 5000, and the filename to save the trajectory to `SiO2_5000K_langevin_traj.hdf5`. In the **Initial Velocity** group box, select the *Maxwell-Boltzmann* type at a temperature of 5000 K.

Enter the same temperature value of 5000 K in the **Langevin** group box in the `Reservoir temperature` field. Leave the remaining settings at their default values. Finally, select *Do not Save* under **Results** (as the trajectory is already being saved).

[3] Pedone Alfonso, Malavasi Gianluca, Menziani M.ăCristina, N.ăCormack Alastair, and Segre Ulderico. A New Self-Consistent Empirical Interatomic Potential Model for Oxides, Silicates, and Silica-Based Glasses. *The Journal of Physical Chemistry B*, 110(24):11780–11795, 2006. doi:10.1021/jp0611018.



Use the button to send the script to the **Editor** in order to make some additional changes.

Facilitating the Melting Process via Lowering the Density

In some cases, particularly when you attempt to melt comparably small supercells, you may sometimes observe that the crystal does not melt even at temperatures far beyond the experimental melting point. A possible reason for this behavior, apart from deficiencies of the employed classical potential, is that crystal structures are additionally stabilized by the periodic boundary conditions, and, generally, artifacts of the periodic boundary increases with decreasing size of the supercell. Another aspect that has to be kept in mind is that the fixed volume of the cell is likely to produce a large system pressure at high temperatures, which can additionally elevate the melting point temperature or in some cases thermodynamically favor another crystal phase over the liquid phase.

Note: If you dont expect or experience any problems with melting the structure, you can run the script as is, and skip the rest of this section.

To facilitate melting, you can slightly increase the cell volume to artificially lower the density. In the script this is achieved by multiplying the the lengths of the cell vectors by a factor of e.g. 1.1.

Modify the block in which the lattice is defined, as shown in the following script:

```
# Set up lattice
# Define a scale factor
scale_cell = 1.1
# Scale all 3 cell vectors by this factor
vector_a = [21.498*scale_cell, 0.0, 0.0]*Angstrom
vector_b = [0.0, 21.498*scale_cell, 0.0]*Angstrom
vector_c = [0.0, 0.0, 21.498*scale_cell]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)
```

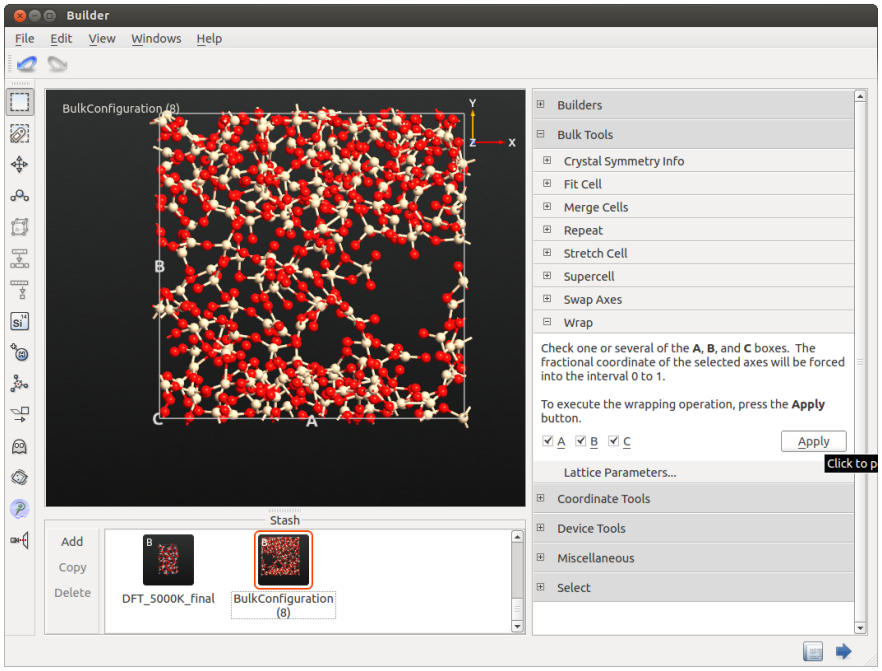
The exact value of the scaling factor does not play a crucial role, and can be increased as long as you still detect some residual crystallinity after annealing the system at 5000 K.

Send the modified script to the **Job Manager** and run the simulation.

It will take a couple of minutes to complete the simulation. If you select the produced trajectory file on the **LabFloor** and visualize it with the **Movie Tool**, you will see the atoms moving around in a chaotic fashion, as they frequently cross the periodic boundaries of the cell. To take a better look at the structure, send the last snapshot of the movie to the Builder via the button in the lower right-hand corner of the **Movie Tool**.

In the **Builder** you can fold all atoms that have crossed the periodic boundaries back into the simulation cell by clicking *Bulk Tools* → *Wrap* → *Apply*.

The final structure may look similar to the example below.



Due to the artificially decreased density the atoms cluster in certain regions, producing an inhomogeneous distribution. However, the purpose of the procedure, i.e. a complete randomization of the previously crystalline structure has been fulfilled very well.

To recover the original density, send this configuration to the **ScriptGenerator** and repeat exactly the same steps described above (i.e. prepare a script containing a **NewCalculator** and a **MolecularDynamics** block with the described settings and send it to the **Editor**), but now divide the cell vectors by the same scaling factor 1.1 (respectively the factor you have used to lower the density) instead of multiplying them. This procedure will compress the system and recover the original density of the cristobalite structure. Run this simulation at 5000 K for 20 000 steps to equilibrate the silica melt at the new density, send the final configuration from the **Movie Tool** to the **Builder**, and *Wrap* the configuration.

Cooling the Structure and Adjusting the Density

If you inspect the resulting structure with respect to the local atomic arrangements, you will encounter many structural defects, such as undercoordinated atoms (e.g. oxygen atoms with only one silicon neighbor).

This is because the resulting amorphous system represents a snapshot of the material in the liquid state at very high temperature. The ultimate goal is usually the generation of solid amorphous material, i.e. at temperatures well below the glass-transition temperature T_g. Therefore, you will perform a MD simulation where the temperature decreases linearly at each time step to cool the system to the desired temperature.

Although real amorphous structures always contain a certain amount of structural defects, the goal of such a cooling simulation is to perform the temperature change as slowly as possible to allow the system to reduce the number of defects as much as possible[Page 1202, 2]. Unfortunately, the limited accessible simulation time often requires the use of very high cooling rates compared to experimental settings. The parameters, suggested in the following paragraphs, have been chosen to allow for a rather quick demonstration of the principle of the structure generation, which may not be the optimal choice for production simulations. To obtain a more reliable amorphous structure, consider increasing the simulation time, which will decrease the cooling rate.

Another important parameter is the density of the amorphous structure. There are essentially two different possibilities for how to deal with this matter.

One way is to adjust the system volume a priori, to obtain the desired experimental amorphous density and leave the volume at this fixed density throughout the cooling simulation[Page 1202, 1]. This technique is the preferred method when the accessible simulation time is very limited, e.g. in DFT-MD simulations. However, it can lead to large residual stress values in the final systems.

The alternative way is to adjust the density continuously during the cooling simulation by coupling the system volume to a barostat at ambient pressure[Page 1202, 2]. Here, the resulting density will depend on the quality of the employed potential and you should check whether it is consistent with experimental values and, in case of large deviations, consider repeating the simulations at fixed volume.

In this example you will use the second technique, i.e. use a *barostat* to control the system pressure.

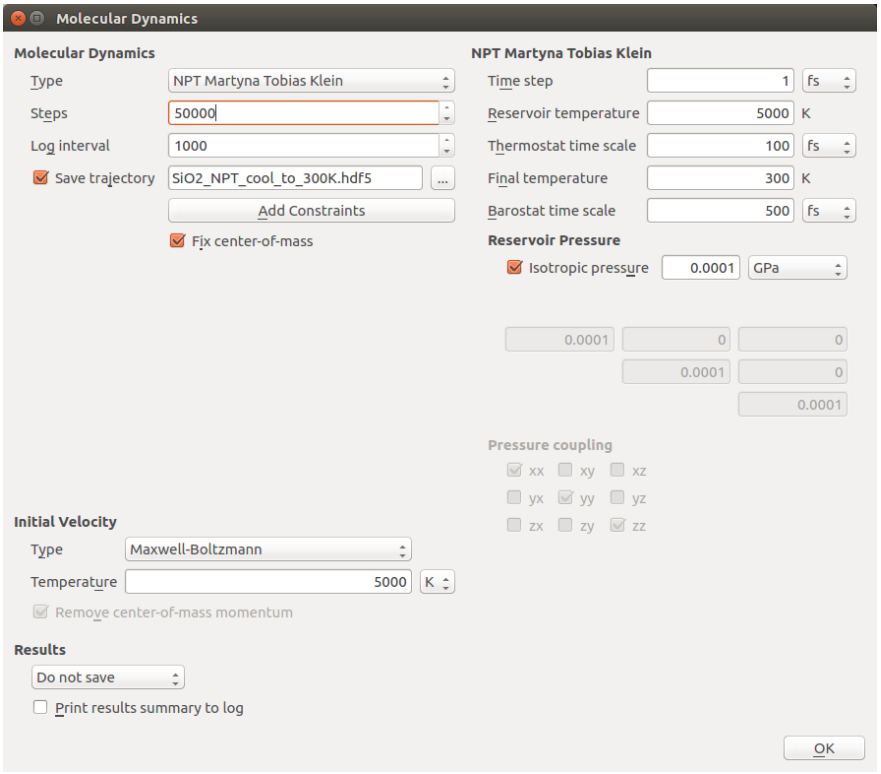
Prepare the cooling simulation by sending the last snapshot of the previous simulation to the **Builder** via the button.

Fold the coordinates back into the simulation cell by using the *Wrap* tool, as described above. Send the configuration to the **ScriptGenerator**. Add a *NewCalculator* and an *Optimize* → *MolecularDynamics* block. In the **NewCalculator** widget, use the same settings as in the previous simulations.

Open the **MolecularDynamics** settings. Select the *NPT Martyna Tobias Klein* type. Select 50 000 steps, a log interval of 1000, and set the trajectory name to `SiO2_NPT_cool_to_300K.hdf5`. Select *Maxwell-Boltzmann* initial velocities corresponding to 5000 K.

Change the `Reservoir temperature` to 5000 K. To cool the system down to room temperature you need to change the `Final temperature` to 300 K. This will result in a linear decrease of the current reservoir temperature from 5000 K to 300 K during the MD simulation.

Leave the remaining parameters at their default values.

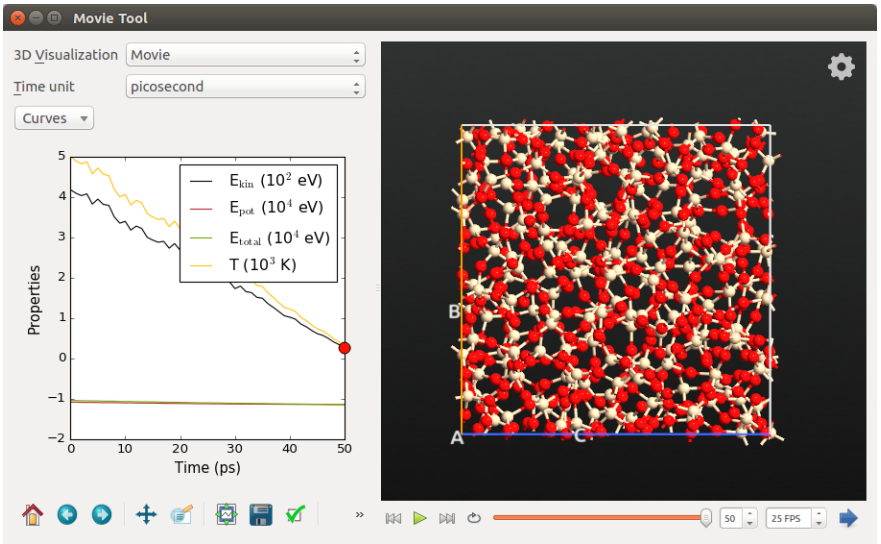


Send the simulation workflow to the **Job Manager** and execute it. It will take some time to complete.

The simulations should run without problems for this system. However, if you encounter extreme, non-stationary cell size fluctuations, or the formation of pronounced vacuum pores during the NPT runs at high temperatures, you can consider switching on the barostat coupling after reaching lower temperatures, e.g. roughly in the region of the suspected glass temperature. You could, for example, use the *NVT Nose Hoover* thermostat to cool the system from 5000 K to 2000 K at constant volume, and the *NPT Martyna Tobias Klein* barostat to cool it from 2000 K to 300 K.

Analyzing the Structure

After the simulations have finished, you can take a look at the trajectories using the **Movie Tool** or the **Viewer**.

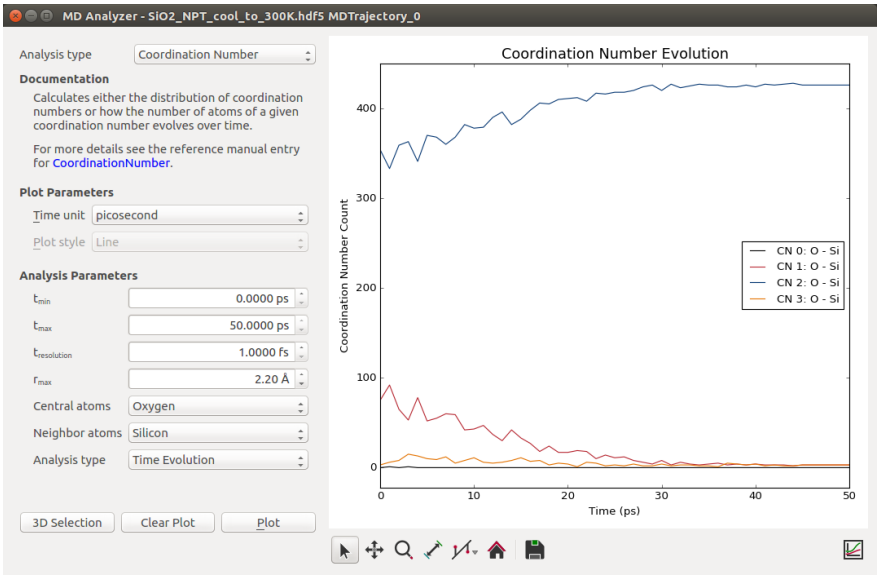


If you have a close look at the local atomic arrangements, you will notice that almost the entire structure is composed of slightly distorted, vertex- sharing tetrahedra formed by SiO_4 units. Additionally, you may encounter some few coordination faults.

For a quantitative assessment, select the `SiO2_300K_NPT_traj.hdf5` file on the **Lab Floor** and open it with the **MD Analyzer tool**. Here, you can perform various kinds of analyses of the MD trajectory. For amorphous systems, we are particularly interested in the *radial distribution function* (RDF). To perform such an analysis, increase the cutoff radius r:sub:*max* to 8.0 Å, select *Silicon* as `Element 1`, select *Oxygen* as `Element 2`, and click `Plot` to display the RDF between these two elements. To discard the high temperature part of the trajectory, you can set the t:sub:*min* parameter to 40 ps to analyze only the last 10 ps of the trajectory.

Instead of sharp single lines, as in a crystal structure, you will find a distribution of broadened peaks, which is characteristic for an amorphous structure. You can plot the RDF for different element pairs or between all elements.

The number of structural defects can be investigated by selecting the *Coordination number* analysis. Select, for example *Silicon* and *Oxygen* as `Element 1`, respectively `Element 2`, to calculate the coordination of silicon with oxygen atoms. Set the cutoff radius to 2.5 Å, according to the first minima in the RDF curve. Click `Plot` to create the distribution of coordination numbers. You will see that most of the silicon atoms have a coordination number of four, and only a tiny fraction have a lower coordination. Now switch the elements to calculate the coordination number of the oxygen atoms. Click `Plot` to add the distribution to the existing plot. Again, the majority of oxygen atoms exhibit its regular coordination number of two, whereas only few atoms are undercoordinated. Switch the `Analysis Type` to *Time Evolution* to plot how the number of atoms with a given coordination number changes during the cooling simulation. The following example shows that the number of one-fold coordinated oxygen atoms decreases during the simulation, while the number of two-fold coordinated atoms increase. This reduction of coordination defects could possibly be enhanced further by choosing a slower cooling rate (i.e. longer simulation time).



The **MD Analyzer** provides a large number of different analysis quantities that can be invoked with a few clicks. Feel free to explore its functions. For example, take a look at the *Angular distribution function* for *Silicon-Oxygen-Silicon* angles, or at the *Neutron scattering function*.

The density of the generated amorphous silica cell can be calculated from the volume of the cell, which can be found in the **Builder** under *Bulk Tools* → *Lattice Parameters*, and the numbers of the elements and their respective masses. For the present example, you should obtain a value around 2.6 g/cm^3. This value is a little higher than the experimental value of 2.2 g/cm^3, which must be considered a small shortcoming of the potential. To obtain exactly the experimental value, you may repeat the cooling simulation with a fixed volume, adjusted to reproduce the desired density. The *NVT Nose-Hoover* thermostat with a linear decrease in temperature might be the right method for such simulations.

The final result can be used as an input configuration for subsequent static calculations or MD simulations to calculate

additional properties of interest. However, you should keep in mind that the generated configuration represents only one possible snapshot out of the amorphous ensemble. This means that in order to obtain meaningful results, you should average over several different samples.

Refining Amorphous Structures

You can refine the structure obtained via classical MD to compensate some of the possible shortcomings of the force field. This becomes particularly beneficial if you intend to calculate additional properties of the amorphous material based on semi-empirical or quantum mechanical calculations, e.g. tight-binding (TB) or density-functional-theory (DFT).

The fastest and easiest way is to perform only a TB or DFT geometry optimization to bring the configuration to the nearest energy minimum. This will primarily improve locally distorted geometries, but is not expected to fix connectivity defects.

A more profound refinement can be achieved by a short TB/DFT MD annealing and cooling procedure[Page 1202, 1]. As these simulation are quite time-consuming they are suited predominantly for smaller systems comprising around 100-200 atoms.

In the following you will be guided through a brief example considering a small silica system with the ATK-DFT calculator. Since the accessible time in DFT-MD simulations is very limited, which makes a reliable constant pressure equilibration difficult and computationally expensive, you will perform the classical and DFT-MD simulations in this particular example at constant volume. The density will remain at the value of the cristobalite crystal, which is very close to the experimental density of amorphous silica.

Setting up the classical simulation

Before running the DFT simulation, you first have to prepare an amorphous cell using a classical MD simulation. These steps essentially follow the protocol outlined in the previous section.

Prepare a cristobalite cell in the same way as described in the previous section, but use only 2x2x1 repetitions in the *Repeat* tool of the **Builder**. This will create a cell containing 96 atoms. Due to this repetition the cell the box size is 14.33 x 14.33 x 7.17 Å3, i.e. the z-direction is thinner compared to the other directions, which might introduce an increased ordering in this direction. To avoid such effects, we will transform the cell to a cubic shape, which has the same volume as the original cell. The final cell size shall therefore be 11.375 x 11.375 x 11.375 $Å^3$. You will perform this shape transformation in two steps during the high temperature simulations where the system is in a liquid state.

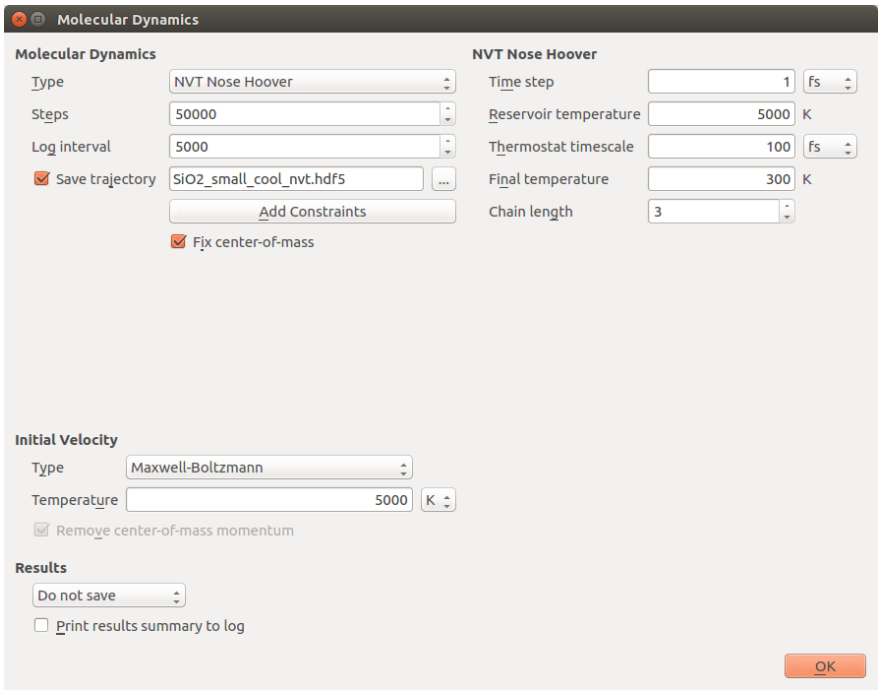
The first step comprises setting up a simulation, in exactly the same way as described above, i.e., adding the **ATK-ForceField** calculator with the *Pedone_Fe3_2006* potential and a **MolecularDyncamics** block with the *Langevin* type, a step number of 20 000, and a temperature of 5000 K. Send the resulting script to the **Editor**. To achieve a lower density and therefore facilitate the melting, increase the length of the *C*-vector of the box to its final value of 11.375 Å:

```
# Set up lattice
vector_a = [14.332, 0.0, 0.0]*Angstrom
vector_b = [0.0, 14.332, 0.0]*Angstrom
# Previous C-vector:
# vector_c = [0.0, 0.0, 7.166]*Angstrom
# New C-vector:
vector_c = [0.0, 0.0, 11.375]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)
```

Run the simulation and send the final snapshot from the **Movie Tool** to the **Builder**. Wrap the configuration, set up exactly the same simulation script as in the previous simulation, and send it again to the **Editor**.

The cell volume has to be reduced to restore the original cristobalite density. Instead of resetting the C-vector to the original value, you can now set the lengths of the A- and B-vectors to 11.375 Å, too, in order to obtain a cubic cell. Run the simulation.

Send the final snapshot to the **ScriptGenerator**, add the same **ATK- Classical** calculator as described above, and an *Optimize* → *MolecularDynamics* block. In the **MolecularDynamics** settings, select the *NVT Nose-Hoover* type, which allows to use a linear decrease in temperature. Adjust the settings as shown in the following screenshot.



These thermostat settings will invoke a linear decrease of the reservoir temperature from 5000 K to 300 K over 50 picoseconds.

Run the simulation. Send the final snapshot via *Movie Tool* → *Send To* → *Builder* → *Wrap* → *Send To* to the **ScriptGenerator**.

Setting up the DFT simulation

In the following you will perform an MD simulation using the **ATK-DFT** calculator in **ATK** to anneal the structure and cool it down to room temperature. MD simulations are rather expensive in terms of computational time. In order to generate amorphous structures, it is not absolutely crucial to obtain accurately converged energies. The suggested settings in the present example will therefore employ parameters, whose accuracy is slightly decreased with respect to the default values to accelerate the calculations. However, before you start the DFT simulations, be aware that these simulations can take very long time, up to several days, so it is recommended to run them on a cluster.

To set up the simulation script, add a *NewCalculator* and two *Optimize* > *MolecularDynamics* blocks. In the **NewCalculator** settings select **ATK-DFT** calculator. Open the calculator widget to adjust some of the settings. In the **Iteration control parameters** increase the `Tolerance` to 0.001 Hartree. In the **Basis set/exchange correlation** settings, change the exchange correlation `Type` to *GGA*. Next, change the basis sets from *DoubleZetaPolarized* to *SingleZetaPolarized*.

Depending on your computational resources you may of course choose more accurate settings here, in particular when aiming at production simulations.

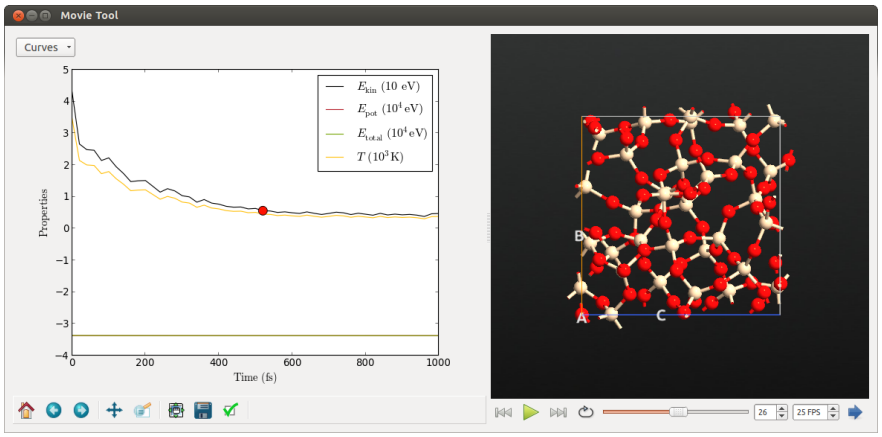
Close the **NewCalculator** settings by clicking `Ok`.

Add two *Optimize* → *MolecularDynamics* blocks to the script. Open the first block. Here, you will set up a short

annealing simulation at 3000 K. Therefore, select the Langevin type, set the number of steps to 500 and the log interval to 20. Choose an appropriate trajectory filename, e.g. `traj-sio2-DFT-MD-3000K.hdf5`, select *Maxwell-Boltzmann* initial velocities at 3000 K and a reservoir temperature of 3000 K, as well. Finally, uncheck `Save` and `Print` and close the settings.

Open the second **MolecularDynamics** block. In this simulation you will rapidly quench the system from 3000 K to 300 K. Use again the Langevin Type, set the number of steps to 1000 and the log interval to 20. Select again Maxwell-Boltzmann initial velocities at 3000 K. To cool the system during the simulation set the reservoir temperature to 300 K now, as if the system was transferred from a very hot reservoir to a heat bath at room temperature. Regarding the short simulation time this method is more efficient than using a linear decrease in the reservoir temperature via the NVT Nose-Hoover chain thermostat. Close the widget and run the simulation.

After the simulations have finished, the movie of the second phase, i.e. the quenching simulation, should look similar to the following.



The temperature decays from the initial value of 3000 K to room temperature. Of course, this rapid quenching within 1 ps is only a rough approximation to the experimental procedure.

Depending on the properties one is interested in calculating, one can continue from the final configuration and perform a DFT geometry optimization.

Creating Crystal/Amorphous Interfaces

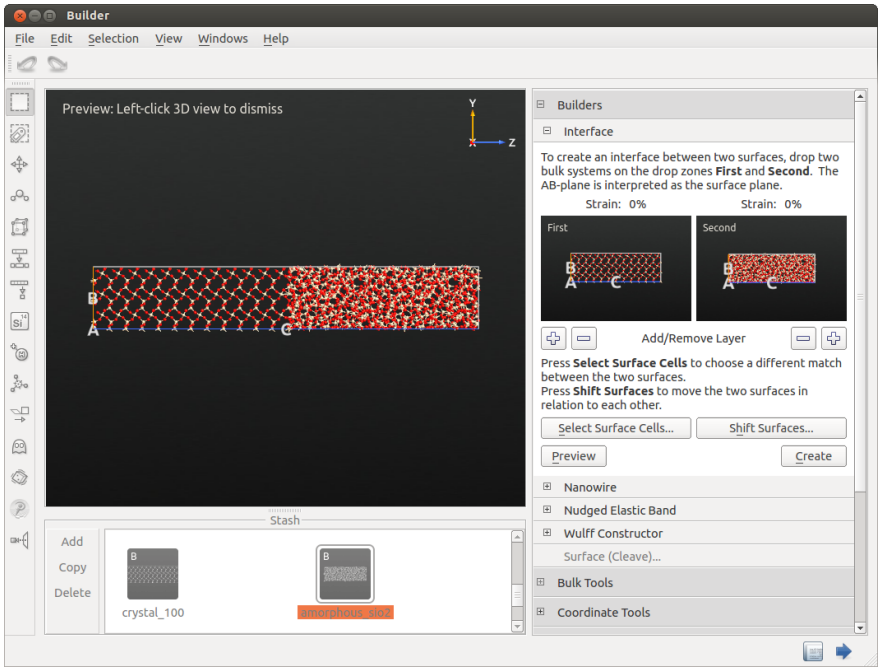
A particularly interesting application is given by interfaces between crystal and amorphous structures, as the local properties in proximity of the interface are largely unknown. Such systems can easily be created in the *QuantumATK*-Builder.

If the crystal is of the same material as the amorphous region, you can start by preparing the crystal configuration. The desired interface plane can be created using the *Surface (Cleaver)* tool in the Builder. Make a copy of the configuration. The latter system will serve as the basis to create the amorphous structure, as their surface areas already match exactly.

The following example considers the interface between quartz and amorphous silica. The initial structure is built in the *Surface (Cleaver)* tool by selecting the (10-10) surface and a 4x3 surface cell. Create a *Periodic (bulk-like)* supercell with a thickness of 12 layers. Perform a geometry optimization of the structure and the cell vectors, using again the **ATK-ForceField** *Pedone_2006_Fe3* calculator, and save the final configuration to a HDF5 file. Drag and drop the file to the **Builder**, and use it as the initial structure for the amorphous cell.

Prepare the amorphous cell via the same simulation protocol as described above, but apply the pressure coupling only to the Z-axis to preserve the lateral cell dimensions. Unidirectional pressure coupling can be selected in the **MolecularDynamics** settings, using *NPT Martyna Tobias Klein*. To couple the barostat only to the Z-component, first uncheck *Isotropic pressure* and then uncheck all boxes in the *Pressure coupling* group box, apart from the `zz` box.

After creating the amorphous supercell, you can join the optimized crystal and amorphous cell using the *Builders* → *Interface* tool in the **Builder** panel. Therefore, you can simply drag and drop the configurations from the **Stash** to *First* and *Second* field in the *Interface builder*. The builder should automatically detect suitable separations of both cells.



Click `Create` to finalize the build process and send the interface configuration to the **ScriptGenerator**.

Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

If you zoom into the interface, you will find many defects, such as dangling bonds. Most of these defects can often be healed by an annealing simulation using classical potentials, even if such potentials are not explicitly designed to include chemical reactions.

Add a *NewCalculator*, an *Optimize* → *OptimizeGeometry*, and an *Optimize* → *MolecularDynamics* block. After selecting the desired calculator settings, in this case the *Pedone_2006_Fe3*-potential, open the **OptimizeGeometry** widget.

After the interface creation there will probably be many artificially close atomic distances, which will cause a subsequent MD simulation to be unstable. To remove these large forces from the interface, you will first run a geometry optimization. The `Force tolerance` does not need to be extremely low, so you can set a value of 0.5 eV/Å. Increase the `Maximum number of steps` to 1000. Uncheck the `Print` box, rename the output file to `interface_optimized.hdf5`, and close the **OptimizeGeometry** settings.

Open the first **MolecularDynamics** settings. During the MD annealing you should fix the atoms in the central crystal region to prevent the crystal region from melting. To this aim, click on `Add constraints` to open the **Constraints Editor**. In the presenter window, draw a rectangle around the central atoms of the crystal region to select them, click `Add tag from selection`, and select `Fixed` atoms constraints for this group in the table on the side of the widget.

Close the **Constraints Editor**.

In the **MolecularDynamics** widget, select *NVT Nose-Hoover Chain*, set the number of steps to 50 000 and the log interval to 2000 steps. First, you should anneal your structure at a temperature high enough to allow for local rearrangements of atoms, thus facilitating the healing of interfacial coordination defects, but not too high, to prevent melting of the global structure. A temperature of 1500 K is suitable for this purpose. Enter this temperature into the initial velocity temperature and the reservoir temperature fields.

Now add another *Optimize* → *MolecularDynamics* block to the script for the cooling simulation. Double-click it and open the **Constraints Editor**. The previously selected group *Selection 0* should already be present in the table. Select `Fixed` atom constraints for this group and close the widget. In the **MolecularDynamics** widget, select the *NVT Nose-Hoover Chain* thermostat, 50 000 steps, a log interval of 2000 steps, and *Configuration velocities* as initial velocities. In the thermostat settings set a `Reservoir temperature` of 1500 K and a `Final temperature` of 300 K to achieve a temperature decrease during the simulation.

Run the simulation by sending the script to the Job Manager.

To finalize the structure, you should run another *NPT* simulation, this time without any constraints. Use the *NPT Martyna Tobias Klein* barostat and uncheck *Isotropic pressure* as well as all stress coupling boxes except for the `zz`-box to equilibrate only the cell height perpendicular to the interface plane. Run the simulation for 50 000 steps at 300 K to obtain a room temperature amorphous/crystal interface structure.

Further Examples

Finally, this section provides some further examples of generating amorphous structures of the technologically relevant materials *alumina*, *titania*, and *hafnia*.

Al_2O_3

For the creation of an amorphous alumina sample, you can essentially follow the protocol outlined in Refs.[Page 1203, 3] and[4]. To use the same classical potential[5], you can either select *Matsui_CaMgAlSiO_1994*, or, if you use an older version of QuantumATK where this potential is not available, download the Python script attached to this tutorial. If you copy and paste its contents to the top of your simulation script, the potential will be available in the simulation.

To set up the system, you need to import the crystal structure of α-Al_2O_3 into the **Builder**. To this aim, download the `Al2O3_corundum_AMS_DATA.cif` file. In the **Builder**, click *Add* → *From Files* and open the file. For a more convenient handling in MD simulations the trigonal cell should be converted into an orthogonal cell. The are several

[4] G.ăGutierrez and B.ăJohansson. Molecular dynamics study of structural properties of amorphous Al2O3. *Phys. Rev. B*, 65:104202, Feb 2002. doi:10.1103/PhysRevB.65.104202.

[5] M.ăMatsui. A transferable interatomic potential model for crystals and melts in the system CaO-MgO-Al2O3-SiO2. *Miner. Mag.*, 58A:571, 1994. URL: http://rruff.info/doclib/MinMag/Volume_58A/58A-2-571.pdf.

ways to achieve this. One possibility is to open the *Builders* → *Surface(Cleave)* tool. Select (10-10) Miller indices, click `Next` and select a 3x1 surface lattice. Click `Next` and select a *Periodic (bulk-like)* cell with a thickness of 4. Click `Finish` to obtain an orthogonal supercell containing 360 atoms.

Now, you should proceed similar to the protocol outlined above, with the exception that the density will be fixed to 3.175 g/cm^3 (as proposed in Ref.[4]) during the cooling simulations. The entire protocol reads as follows:

1. Run a high temperature MD simulation at 5000 K and artificially reduced density.
2. Cool the system to 3000 K using the *NVT Nose-Hoover* thermostat.
3. Compress the system to the desired amorphous density.
4. Anneal at 3000 K.
5. Slowly cool the system to room temperature.
6. Run a final simulation at room temperature to calculate observables.

Send the orthogonal cell to the ScriptGenerator to set up a basic molecular dynamics script. Add a *NewCalculator* and two *Optimize* → *MolecularDynamics* block.

In the **NewCalculator** block, select **ATK-ForceField** and the *Matsui_CaMgAlSiO_1994* potential.

In the first **MolecularDynamics** block select the *NVT Nose-Hoover* thermostat, 150 000 steps, a log interval of 5000 steps and uncheck the `Save trajectory` box. Set the initial velocities to a *Maxwell-Boltzmann- distribution* at 5000 K. Set the reservoir temperature to 5000 K and the final temperature to 3000 K. The remaining parameters should be kept at the default values. Finally, uncheck `Save` and `Print`, close the calculator widget, and send the script to the **Editor**.

The second **MolecularDynamics** block should have similar settings, but 300 000 steps, a reservoir temperature of 3000 K and a final temperature of 300 K.

To obtain an artificially reduced density you should again multiply all three lattice vectors with 1.1, as explained above.

This first **MolecularDynamics** sequence yields a liquid system at 3000 K and lowered density. Based on this system, the density should be adjusted to achieve the target density of 3.175 g/cm^3. As it is often cumbersome to calculate the macroscopic density of an atomic cell and to obtain the exact scaling factors for the cell vectors by hand, this can conveniently be achieved using the QuantumATK unit engine by inserting the following piece of code in the simulation script, right before the second **MolecularDynamics** block (i.e. after the line `bulk_configuration = md_trajectory.lastImage()`):

```
# Get the volume and the lattice vectors of the cell.
V = bulk_configuration.bravaisLattice().unitCellVolume()
lattice_vectors = bulk_configuration.bravaisLattice().primitiveVectors()
# Get the coordinates and the elements.
fractional_coordinates = bulk_configuration.fractionalCoordinates()
elements = bulk_configuration.elements()
# The total and species-resolved number of atoms.
N = len(elements)
N_Al = len(numpy.where(numpy.array(elements)==Aluminum)[0])
N_O  = len(numpy.where(numpy.array(elements)==Oxygen)[0])
# Calculate the current density
density = (N_Al*Aluminium.atomicMass() + N_O*Oxygen.atomicMass())/V
# Get the density in g/cm**3
density = density.inUnitsOf(kiloGram/Meter**3)/1000.0
target_density = 3.175
# Calculate the scaling factor for the lattice vectors.
scale_factor = (density/target_density)**(1.0/3)
# Scale the lattice vectors.
```

(continues on next page)

(continued from previous page)

```
lattice_vectors = lattice_vectors*scale_factor
# Define a new cell.
new_lattice = UnitCell(lattice_vectors[0], lattice_vectors[1], lattice_vectors[2])
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=new_lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
# The calculator is still defined and can readily be attached.
bulk_configuration.setCalculator(calculator)
```

These lines will extract the volume, calculate the density and the scaling factor required to obtain the desired target density of 3.175 g/cm^3. Finally, the lattice vectors are scaled accordingly and a new configuration is created, on which the second cooling sequence is performed.

Save the entire script to a Python file (an example for the final script can be found in the attachment), and run it via the **JobManager** or in a terminal using `atkpython`. It will take a couple of hours to complete. After the simulations have finished you can analyze the structure, as usual.

The complete script can be found in the file `script_al2o3_amorphous.py`.

TiO_2

To simulate amorphous titanium oxide, you will the use the potential of Matsui and Akaogi[6]. This potential has been used in many simulation studies of crystalline and amorphous titania systems, and yields properties in very good agreement with DFT-MD simulations. This potential is not yet available as a predefined potential in QuantumATK, but it can easily be set up in a script. Simply download the file `Matsui_TiO.py`, which is supplied as an attachment to this tutorial. Copy and paste its contents to the top of the script the you will create in the following.

To set up the system open the Builder and use *Add* → *From Database* to add the TiO_2 rutile crystal structure. Use the *Bulk tools* → *Repeat* function to create a 4x4x6 repetition of the system, which contains 576 atoms. Send the configuration to the **ScriptGenerator** and set up the same basic molecular dynamics script as in the alumina example. Send the final script to the **Editor**.

Locate the calculator definition as shown below:

```
potentialSet = Matsui_TiO_1991() # or any other potential class.
calculator = TremoloXCalculator(parameters=potentialSet)
```

Replace the line starting with `potentialSet = ...` with the entire content of the attached file `Matsui_TiO.py`.

When you adjust the volume to obtain the desired density, make sure that you adapt the script using the correct stoichiometry of TiO_2, the right element masses, and the right target density. For amorphous titania a density of 3.80 g/cm^3, as reported in Ref.[7], can be used.

The complete script can be found in the file `script_tio2_amorphous.py`.

[6] MăMatsui and M.ăAkaogi. Molecular dynamics simulations of the structural and physical properties of the four polymorphs of TiO2. *Mol. Sim.*, 6:239, 1991. URL: http://dx.doi.org/10.1080/08927029108022432.

[7] DăMergel, DăBuschendorf, SăEggert, RăGrammes, and BăSamset. Density and refractive index of TiO2 films prepared by reactive evaporation. *Thin Solid Films*, 371(1):218–224, 2000. URL: http://www.sciencedirect.com/science/article/pii/S0040609000010154.

HfO_2

For hafnia you can use the potential of Wang et al.[8], which is available in **QuantumATK** as *Wang_HfOZr_2012*.

To set up the system add the HfO_2 crystal structure from the database. Transform the *FCC* cell into a conventional unit cell, by opening the *Bulk tools* → *Supercell* plugin, and clicking `Conventional` and `Transform`. Repeat the system 3x3x3 times to obtain a supercell of suitable size.

Set up the same script as in the previous two examples. Make sure to adapt the system and element specific parameters accordingly and use an amorphous density of 7.97 g/cm^3 as suggested in[Page 1216, 8].

The complete script can be found in the file `script_hfo2_amorphous.py`.

3.8.7 Youngs modulus of a CNT with a defect

Modulus of elasticity

Youngs modulus is the relationship between stress, σ, and strain, ε, in a material in the elastic regime, where Hookes law applies: $E = \sigma/\varepsilon$

Carbon nanotubes are well known for their exceptional mechanical properties; Youngs moduli in the 1 TPa range have been measured[1], which is higher than any other known material. The theoretical Youngs modulus is fairly simple to calculate for a defect-free carbon nanotube, but this is not the case when defects are present. In that case, **molecular dynamics** (MD) is a suitable tool for calculation of the mechanical properties.

In this tutorial you will build a carbon nanotube (CNT) with a Stone–Wales defect, and use the **ATK-ForceField** engine to calculate the Youngs modulus.

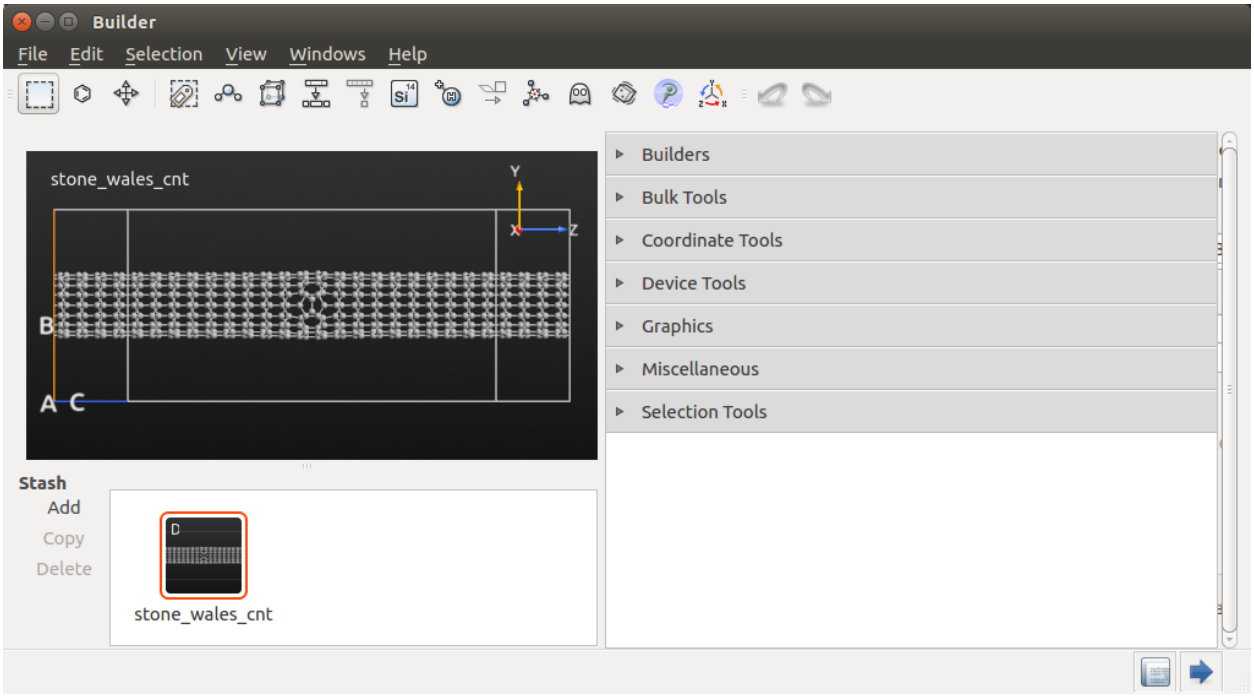
Important: This tutorial uses a CNT structure that is built in the tutorial *Stone–Wales Defects in Nanotubes*. If you have not already done so, please go that tutorial first and build the CNT device configuration, or simply download it as a Python script: `stone_wales_cnt.py`.

CNT bulk configuration

Open the QuantumATK **Builder** , and use *File* → *Add* → *From Files* to import the previously saved CNT configuration to the Stash. Note that the structure is a device configuration with the Stone–Wales defect in the middle of the central region.

[8] Y.ăWang, F.ăZahid, J.ăWang, and H.ăGuo. Structure and dielectric properties of amorphous high-k oxides: HfO2, ZrO2, and their alloys. *Phys. Rev. B*, 85:224110, 2012. URL: http://dx.doi.org/10.1103/PhysRevB.85.224110.

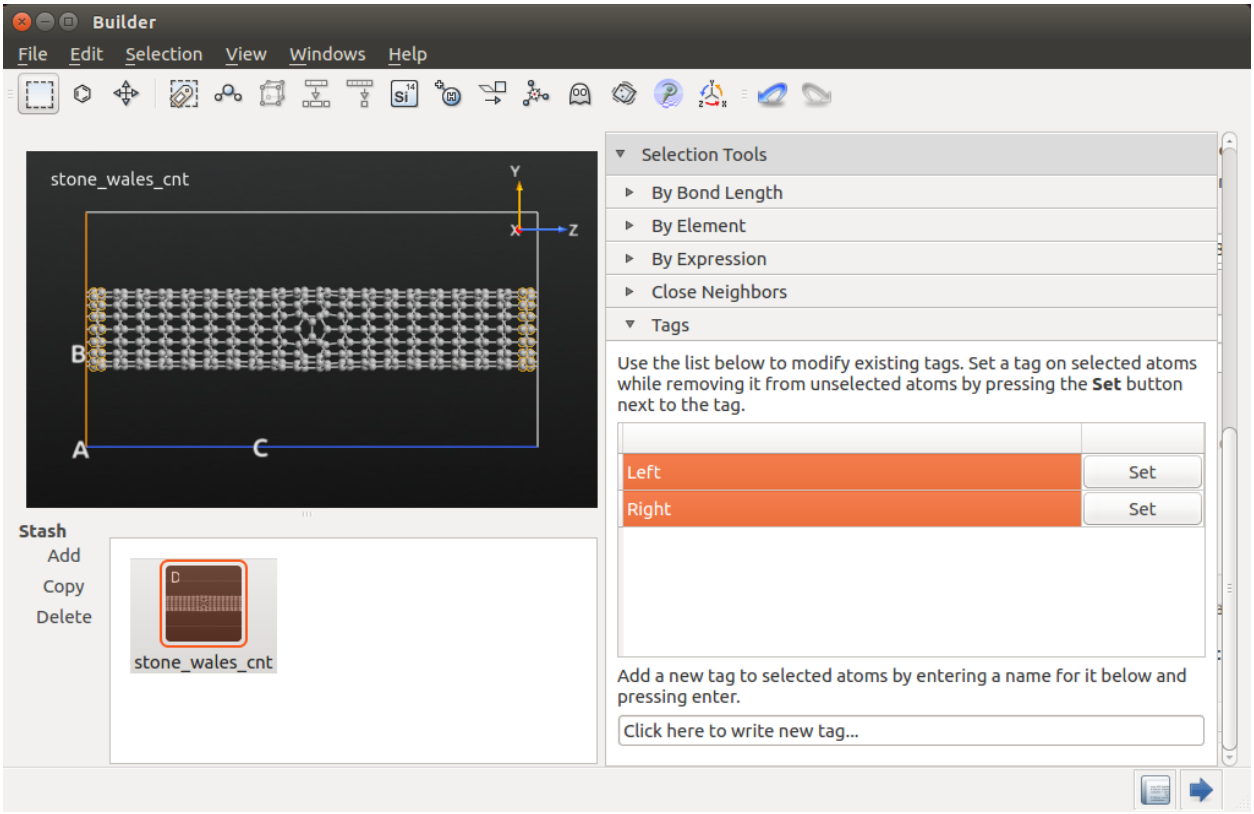
[1] A.ăKrishnan, E.ăDujardin, T.ăW. Ebbesen, P.ăN. Yianilos, and M.ăM.ăJ. Treacy. Youngs modulus of single-walled nanotubes. *Phys. Rev. B*, 58:14013–14019, Nov 1998. doi:10.1103/PhysRevB.58.14013.



Next, highlight the device Stash item, and click the icon to convert it to a bulk configuration (the device electrodes are simply removed), and use the *Selection Tools* → *Tags* tool to add tags to the atoms at both ends of the nanotube:

- Use the mouse to highlight the left-most carbon atoms and set the tag Left on them.
- Do the same for the right-most carbon atoms, as illustrated below.

These tags will be used to impose constraints on the structure during the MD simulation.



Configuring the MD simulation

Send the CNT bulk configuration with the Stone–Wales defect to the Script Generator and add the New Calculator and *Optimization* → *MolecularDynamics* blocks. Then edit both blocks to configure the calculation:

- First, change the default output file to `mdtrajectory_cnt.hdf5`.
- In the **New Calculation** block, select the *ATK-ForceField calculator*, and make sure to use the *Tersoff_C_2010* classical potential, which is designed for simulating lattice dynamics of graphene and nanotubes.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

- Make the following changes to the **MolecularDynamics** block:

–Molecular Dynamics options:

∗Set the Type to *NVT Berendsen*.

∗Set Steps to *20000*.

∗Set the Log interval to *200*.

–Initial Velocity options:

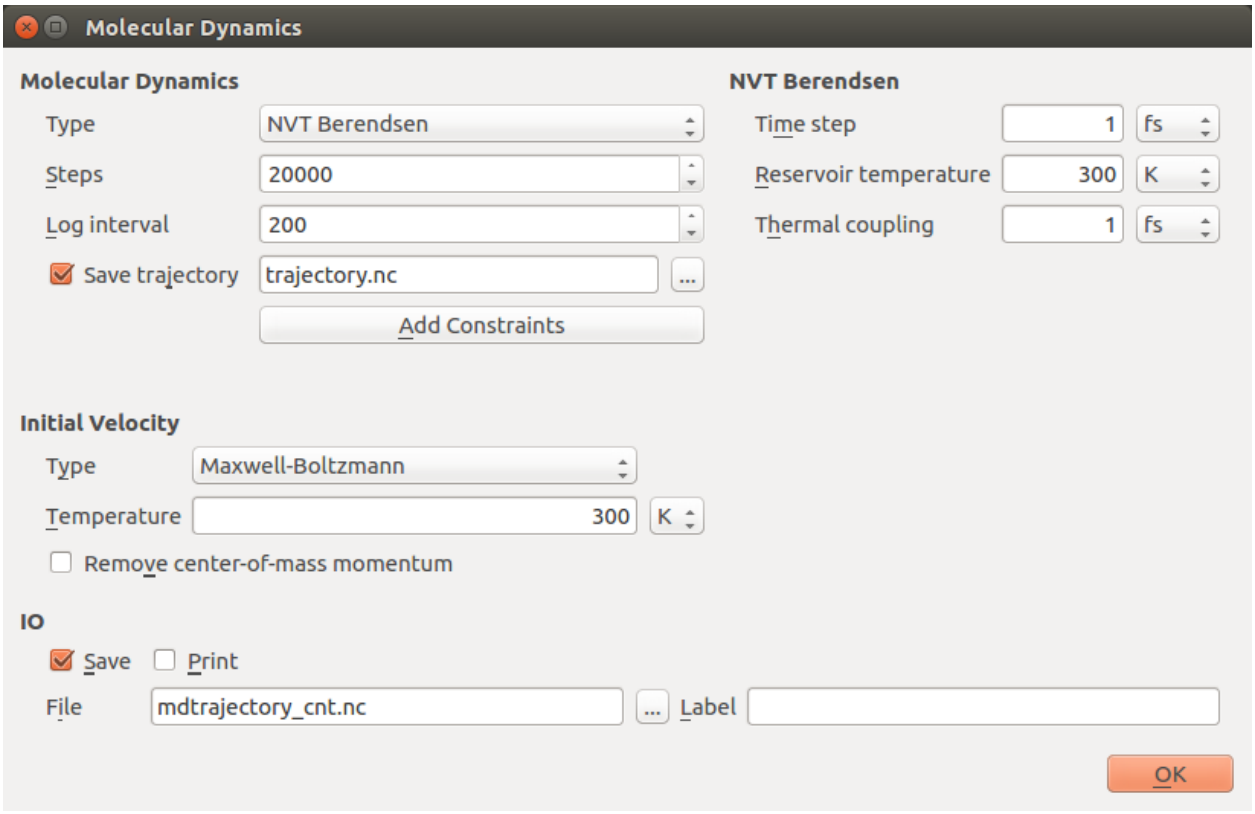
∗Set Type to *Maxwell-Boltzmann*.

∗Set Temperature to *300 K*.

∗Untick the *Remove center-of-mass momentum* option.

–**NVT Berendsen** options:

∗Set the Thermal coupling to *1 fs*.



Leave the remaining settings at defaults, like in the image above.

Save the script as `mdtrajectory_cnt.py` and send it to the **Editor** by using the icon.

Adding Python hooks

It is possible in QuantumATK to define hooks; Python functions or classes that are called before or after each MD step. These hooks can be used to provide customized on-the-fly analysis or monitoring of the system, or – as in this case – to modify the structure dynamically during the simulation.

You will now define a pre-step hook that will be used to strain the nanotube slightly for each step of the MD run:

•With the Python script open in the Editor, scroll down to the bottom of the script and locate the block defining the `md_trajectory`. The predefined tags Left and Right can now be used to apply constraints to the system. Change the line

```
constraints=[]
```

to

```
constraints=bulk_configuration.indicesFromTags(["Left","Right"])
```

•In the `md_trajectory` block, add the lines

```
pre_step_hook=strainer,
write_stresses=True,
```

and add a comma to the line immediately above it (`method=method,`). The first line will invoke the strainer hook function (defined below), and the second line will ensure that stresses are calculated and stored in the trajectory.

•You should also define what strainer is: Add the following line immediately before the `md_trajectory` block:

```
strainer = StressStrainUtility()
```

•The bottom of the script should now look like this:

```
# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=False
)

method = NVTBerendsen(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    initial_velocity=initial_velocity
)

strainer = StressStrainUtility()

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=bulk_configuration.indicesFromTags(["Left","Right"]),
    trajectory_filename='mdtrajectory_cnt.hdf5',
    steps=20000,
    log_interval=200,
    method=method,
    pre_step_hook=strainer,
    write_stresses=True
)
```

•Line 17 above initializes an instance of the class `StressStrainUtility`, which is a Python class that you now will define. Insert the following at the very top of the script:

```
# Define a prestep function which can expand the system

class StressStrainUtility:
    """ Perform a strain of a configuration over time. """
    def __init__(self, delta=0.0001, skip=0):
        """
        @param delta The strain to apply in each step
        @param  first steps to "skip" strain
        """
        self.__delta = [0.0,0.0,delta]*Angstrom
```

(continues on next page)

(continued from previous page)

```
11          self.__skip = skip
12          self.stresses = []
13          self.lengths = []
14
15      def __call__(self, step, time, configuration, forces, stress):
16          """
17          Strainer call.
18          """
19          # Get the lattice vectors.
20          u1, u2, u3 = configuration.primitiveVectors()
21
22          # Don't apply strain in the earliest MD steps
23          if step < self.__skip:
24              return
25
26          # Store the history
27          self.stresses.append( stress[2,2].inUnitsOf(eV/Angstrom**3) )
28          self.lengths.append( u3[2].inUnitsOf(Angstrom) )
29
30          # Increase the lattice vectors.
31          u3 = u3 + self.__delta
32
33          # Set the new lattice.
34          configuration.setBravaisLattice(UnitCell(u1,u2,u3))
35
36          # Move all the atoms that are constrained on the right side.
37          constraints = configuration.indicesFromTags("Right")
38          for c in constraints:
39              xyz = numpy.array(configuration.cartesianCoordinates()[c]) + numpy.
↪array(self.__delta)
40              configuration.setCartesianCoordinates(indices=[c], cartesian_
↪coordinates=[xyz]*Angstrom)
41
42      def length(self):
43          return numpy.array(self.lengths)*Angstrom
44
45      def stress(self):
46          return numpy.array(self.stresses)*eV/Angstrom**3
47
```

This class defines a method to enable straining of a unit cell along the z-direction. You will use it to study how a CNT with a defect responds to strain.

As you can see from the script, the class has two inputs, `delta` and `skip`:

delta
defines how much the u3 vector (the z-direction) is expanded on every step in the MD calculation.

skip
defines how many steps should pass before the configuration will be strained.

The actual straining is done in the for-loop beginning at line 38, by increasing the coordinates of the atoms tagged Right by the amount `delta` along the z-direction. On lines 27 and 28, the stress and total length of the configuration are calculated and appended to two separate lists. These will be used later for plotting.

Tip: By writing `pre_strain_hook=strainer` in the `md_trajectory` block, the class (technically, the method `__call__`) is called *before* the force and energy calculation in every MD step (for more details about hook functions see *MolecularDynamics()*).

Computing Youngs modulus

To properly calculate the Youngs modulus, it is necessary to take the total volume of the simulation domain into account. The stress reported by QuantumATK therefore has to be multiplied by the total volume of the cell and then divided by the volume of the nanotube.

Several definitions of the volume of a CNT. A common way of defining it is to say that the wall of the nanotube extends for half a π-bond in and out of the wall, so that it becomes a cylindrical shell with thickness equal to that of a π-bond, approximately 1.35 Å, so that the total cross-sectional area is 1.35 Å times the circumference of the nanotube. The circumference for a (5,5)-armchair nanotube is $5 \times (3 \times x)$, where x=1.42 Å is the carbon-carbon bond length. Finally, for the volume, you need the length of the tube, which is given in the `.hdf5` file saved by QuantumATK.

- At the bottom of the script, delete the line `nlsave('mdtrajectory_cnt.hdf5', md_trajectory)`.
- Add instead the following lines of code in order to invoke the proper stress calculation:

```
# ------------------------------------------------------------
# Finding Young's Modulus
# ------------------------------------------------------------

# Find center of cnt from mean x- and y-coordinates
coords = bulk_configuration.cartesianCoordinates()
x_coords = coords[:,0]
y_coords = coords[:,1]
x_0 = numpy.sum(x_coords)/len(x_coords)
y_0 = numpy.sum(y_coords)/len(y_coords)

# Find radius by mean distance from center to atoms' x- and y-coordinates
radius = numpy.sum( ( (x_coords - x_0)**2 + (y_coords - y_0)**2 )**(1/2.0) ) / len(x_
↪coords)

# Find cross-sectional area by multiplying circumference with length of a pi-bond
pi_bond = 1.35
circumference = radius * 2 * numpy.pi
area = circumference * pi_bond

# Find volumes of CNT and of whole cell
volume_cnt = area * strainer.lengths[-1]

u1, u2, u3 = bulk_configuration.bravaisLattice().primitiveVectors()
volume_cell = u1[0] * u2[1] * u3[2]

# Factor the stresses to the volume of the CNT
stresses_factored = numpy.array(strainer.stresses * (volume_cell / volume_cnt) )*eV/
↪Angstrom**3

# Calculate the strain
strain = (strainer.length() - strainer.length()[0] ) / strainer.length()[0]
```

(continues on next page)

(continued from previous page)

```
# Calculate Young's modulus in TPa
modulus = (stresses_factored[-1].inUnitsOf(Pa) - stresses_factored[0].inUnitsOf(Pa) ) /␣
↪float(strain[-1]) / 1e12
print("Young's Modulus: {} TPa".format(modulus))
```

•Finally, add a few lines of code that will save different parts of the calculation and plot the results using pylab.

```
# Save the result to a file.
print(strainer.length())
nlsave("mdtrajectory_cnt.hdf5", md_trajectory, "MD")
nlsave("mdtrajectory_cnt.hdf5",strainer.length(), "C length")
nlsave("mdtrajectory_cnt.hdf5",strainer.stress(), "C stress")
nlsave("mdtrajectory_cnt.hdf5",stresses_factored, "C stresses factored")

# Plot stress-strain curve.
import pylab
pylab.figure()
pylab.plot(strain * 100, stresses_factored.inUnitsOf(GPa))
pylab.title('CNT with Stone-Wales defect')
pylab.xlabel('Strain (%)')
pylab.ylabel('Stress (GPa)')
pylab.grid(True)
pylab.savefig("mdtrajectory_cnt_fig.png")
```

The script is now ready to be executed. You may download the script to make sure you did the manual editing correctly: `mdtrajectory_cnt.py`.

Send the script to the **Job Manager** by clicking the icon, then save it, and run it. The job should take around 3 minutes on a laptop.

Visualize and analyse the results

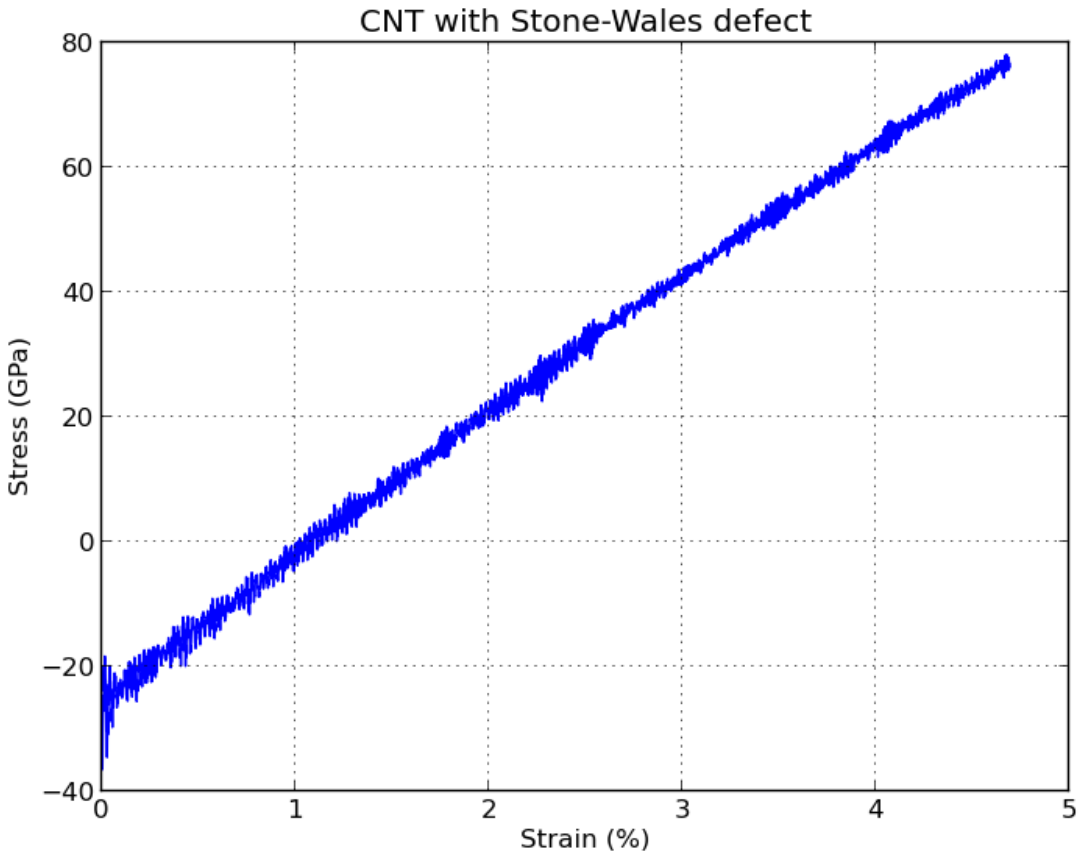
The nanotube lengths and stresses that occur during the MD simulation run are saved in the `.hdf5` file with different IDs (C length, C stress, and C stresses factored). These are used to produce the plot titled `mdtrajectory_cnt_fig.png`, which is automatically saved in the project folder. It shows the stress vs. strain relationship for the nanotube, which is also shown below. The linear (elastic) behavior for small strains make it possible to calculate Youngs modulus E for the nanotube, defined as $E = stress/strain$.

With this procedure, the calculated Youngs modulus of the carbon nanotube with one Stone–Wales defect is around 2.3 TPa (it may vary across different MD runs). This is pretty far from the calculated value of 1.28 TPa for defect-free nanotubes[2]. However, as shown in[1], nanotubes with defects have in general a higher Youngs modulus.

Note: The figure above shows that the stress is negative for small stress levels, which means that the nanotube is actually under compressive stress until around 1.1% strain.

This could be avoided by running a **geometry optimization** step before running the MD calculation, which can be done using the Optimization block available in the Script Generator. However, for the strains investigated here, the CNT

[2] Daniel Sanchez-Portal, Emilio Artacho, JoseăM. Soler, Angel Rubio, and Pablo Ordejon. Ab initio structural, elastic, and vibrational properties of carbon nanotubes. *Phys. Rev. B*, 59:12678–12688, May 1999. doi:10.1103/PhysRevB.59.12678.

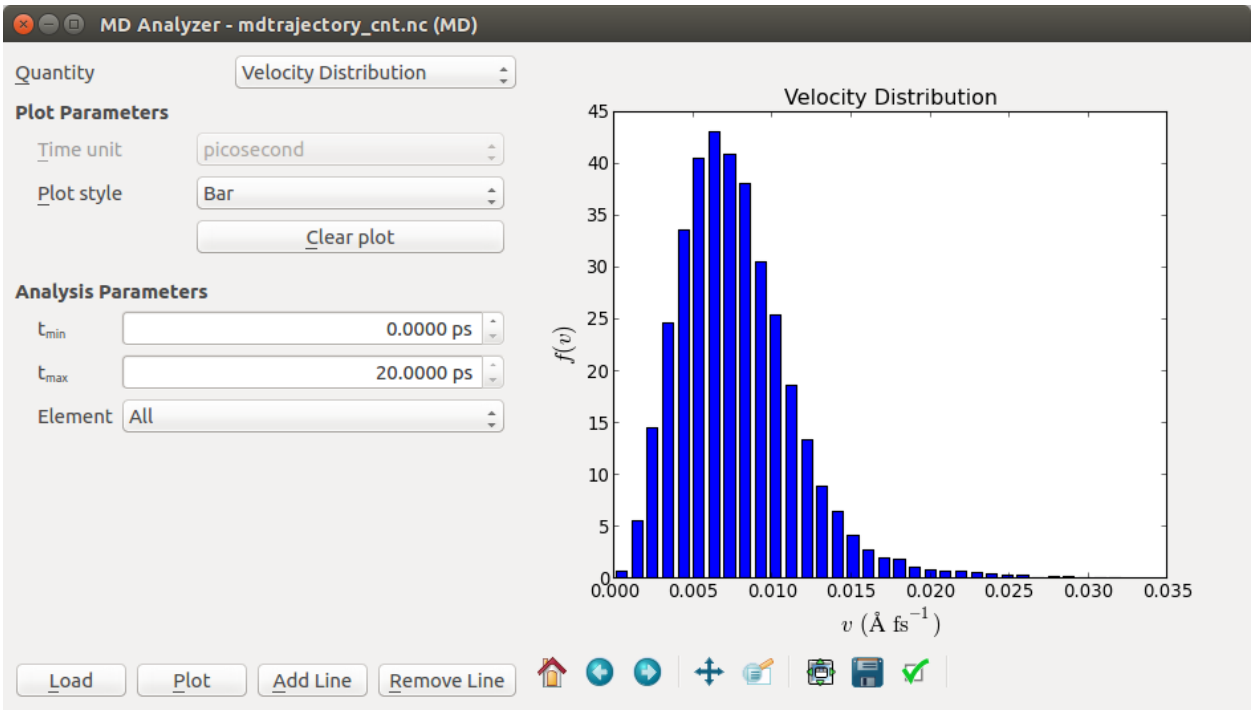

CNT with Stone-Wales defect
Stress (GPa)
Strain (%)
80
60
40
20
0
−20
−40
0
1
2
3
4
5

shows a linear stress–strain behavior, so the obtained Youngs modulus should not change significantly if a geometry optimization is performed first.

The data file `mdtrajectory_cnt.hdf5` should be shown on the **LabFloor**. Select the file and click the **Movie Tool** in the right-hand plugins panel. The Movie Tool can be used to inspect MD trajectories; it will show a movie of the simulation and data about energies and temperatures. The tool looks like illustrated below.

Tip: Right click the mouse and select *Show bonds* to visualize the bond connection between the carbon atoms.

Another useful plugin is the **MD Analyzer**, which can show the radial distribution, velocity distribution, and the kinetic energy distribution of the atoms in the system. A screenshot of the velocity distribution is shown below.



References

3.8.8 Diffusion in Liquids from Molecular Dynamics Simulations

Version: P-2019.03

In this tutorial, you will learn how to calculate the atomic diffusion coefficient in liquid metals using molecular dynamics (MD) simulations. In general, an accurate measurement of the diffusion coefficient in liquids is a challenging task. The present tutorial demonstrates how the **QuantumATK** tools allow you to efficiently study this fundamental phenomenon in a straightforward manner on a computer instead.

We adopt **classical molecular dynamics** to be able to generate rather long MD trajectories of liquid copper. This is a computationally efficient approach to obtain statistically meaningful averages of the physical quantities needed for calculating the diffusion coefficient of a single Cu atom in this liquid metal. The entire MD procedure is divided into three major steps:

1. *heating crystalline copper* to above its melting point,

2. *annealing liquid copper* at high temperature to bring the system to equilibrium,
3. *further annealing of this equilibrated liquid copper* to get more statistical data for accurate calculation of the atomic diffusion coefficient, using the time-dependent mean-square displacement derived from the corresponding MD trajectory.

Theory

Diffusion in liquid metals differs from atomic diffusion in solids, as it takes place at a much shorter time scale, allowing for a brute force approach based on molecular dynamics simulations to calculate an atomic diffusion coefficient, denoted D. Knowing this fundamental physical quantity is needed to understand liquid dynamics, as well as many other related phenomena, such as nucleation and crystal growth. Unfortunately, experimental diffusion data for liquids are often rather inaccurate for various reasons[1]. Instead, liquid diffusion can be rather accurately studied theoretically as demonstrated in this tutorial.

Tip: If you are interested in atomic diffusion in solids, please refer to the following two tutorials on *Boron diffusion in bulk silicon* and *Modeling Vacancy Diffusion in Si0.5 Ge0.5 with AKMC* as studying diffusion in solids usually requires a more sophisticated computational approach compared to the molecular dynamics approach.

Given that the MD simulation of the equilibrated liquid has been run for time T_{MD}, the diffusion coefficient, D, can then be expressed in terms of the mean-square displacement of atoms in this MD run

$$6tD = \left\langle X^2(t) \right\rangle ,$$

where $\left\langle X^2(t) \right\rangle$ is the mean-square displacement of Cu atoms in liquid copper, calculated at the observation time t

$$\left\langle X^2(t) \right\rangle = \lim_{T\to\infty} \frac{1}{T} \int_0^T dt' \frac{1}{N_{\mathrm{at}}} \sum_{j=1}^{N_{\mathrm{at}}} \left[\mathbf{r}_j(t'+t) - \mathbf{r}_j(t') \right]^2 \approx \frac{1}{T_{\mathrm{MD}} - t} \int_0^{T_{\mathrm{MD}}-t} dt' \frac{1}{N_{\mathrm{at}}} \sum_{j=1}^{N_{\mathrm{at}}} \left[\mathbf{r}_j(t'+t) - \mathbf{r}_j(t') \right]^2 ,$$

and N_{at} is the total number of Cu atoms in the supercell, and $\mathbf{r}_j(t'+t)$ and $\mathbf{r}_j(t')$, where $j \in [1, \ldots, N_{\mathrm{at}}]$, are atomic position coordinates at time $t'+t$ and t', respectively. In this tutorial, t is the time interval of observation that is adopted to extract the atomic diffusion coefficient from the time-dependent mean-square displacement, $\left\langle X^2(t) \right\rangle$, assuming that $\left\langle X^2(t) \right\rangle$ shows a linear behavior with respect to t. This observation time t cannot exceed the total time T_{MD} of the MD simulation of liquid. The mean-square displacement may become quite noisy, not showing a linear dependence with respect to the time interval t for the values of t close to T_{MD}. In some cases, only those values of t that are much smaller than the simulation time, i.e., when $t \ll T_{\mathrm{MD}}$, allow for accurate estimates of the diffusion coefficient from the linear fit of $\left\langle X^2(t) \right\rangle$.

Strictly speaking, the formula given in the previous paragraph defines the self-diffusion coefficient of liquid copper. That is the atomic diffusion coefficient averaged over all the copper atoms in the liquid. This ensemble averaging is possible in this case because all these copper atoms are assumed to be identical, having the same atomic mass and

[1] A.ăMeyer. Self-diffusion in liquid copper as seen by quasielastic neutron scattering. *Phys. Rev. B*, 81:012102, Jan 2010. doi:10.1103/PhysRevB.81.012102.

other physical characteristics that may affect diffusion of atoms. If a foreign (impurity) atom or a copper isotope with a different atomic mass was introduced in liquid copper, we would then have an additional diffusion process, which is diffusion of this foreign atom in liquid copper. To quantify this process, we can calculate the atomic diffusion coefficient, D_{imp}, of this single atom as follows

$$6tD_{\mathrm{imp}} = \left\langle X^2(t) \right\rangle \approx \frac{1}{T_{\mathrm{MD}} - t} \int_0^{T_{\mathrm{MD}}-t} dt' \left[\mathbf{r}_{\mathrm{imp}}(t'+t) - \mathbf{r}_{\mathrm{imp}}(t') \right]^2 ,$$

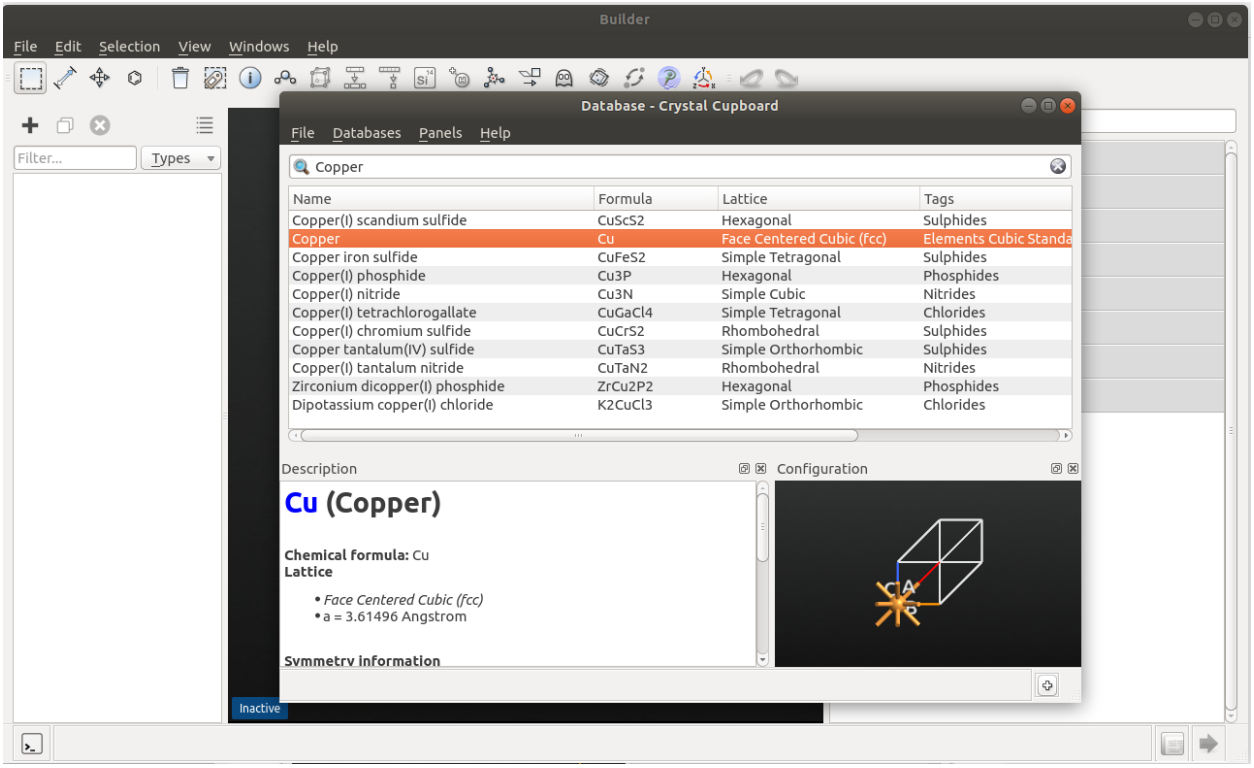
where the individual atom trajectory $r_{\mathrm{imp}}(t)$ is taken from the MD trajectory of the entire system of many atoms.

One should bear in mind that diffusion properties of materials depend on the temperature, and that also holds true for the atomic diffusion coefficient. In this tutorial, we will study liquid copper at T=2000 K, which is a temperature well above the melting point of crystalline copper.

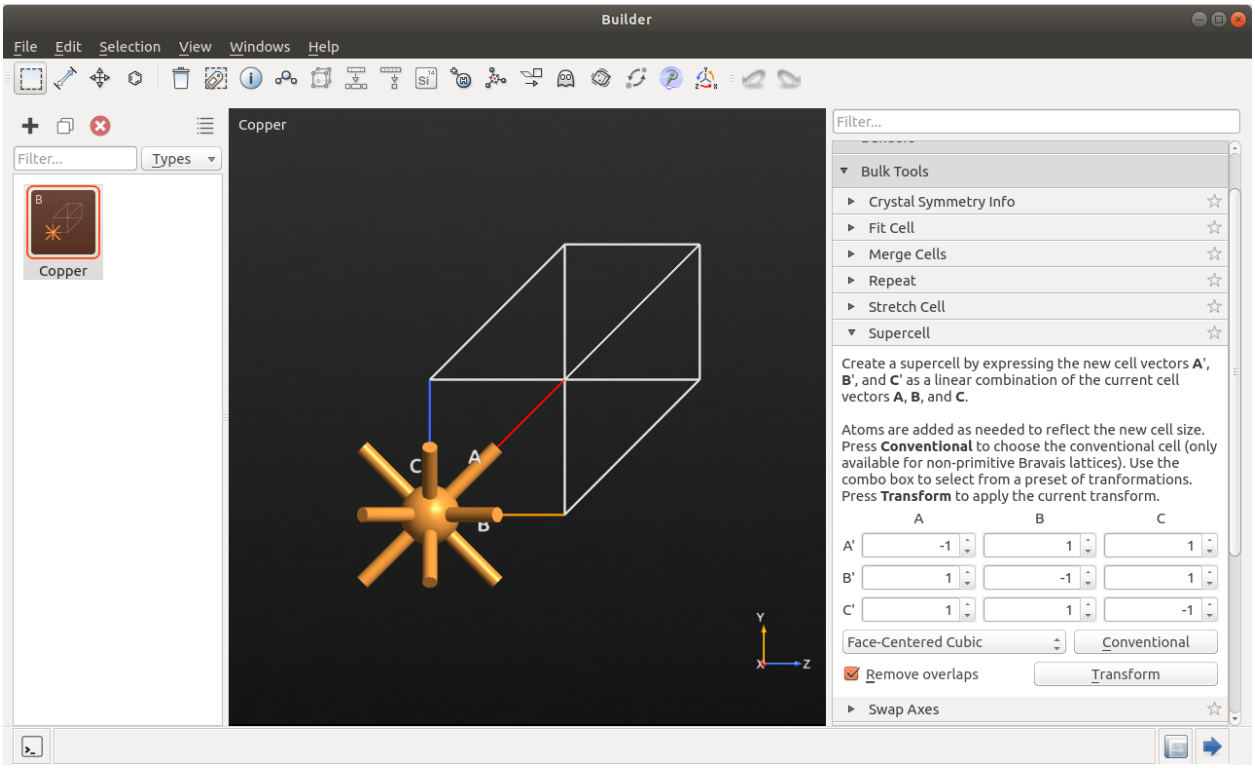
Computational Procedure

Building a Copper Supercell Structure

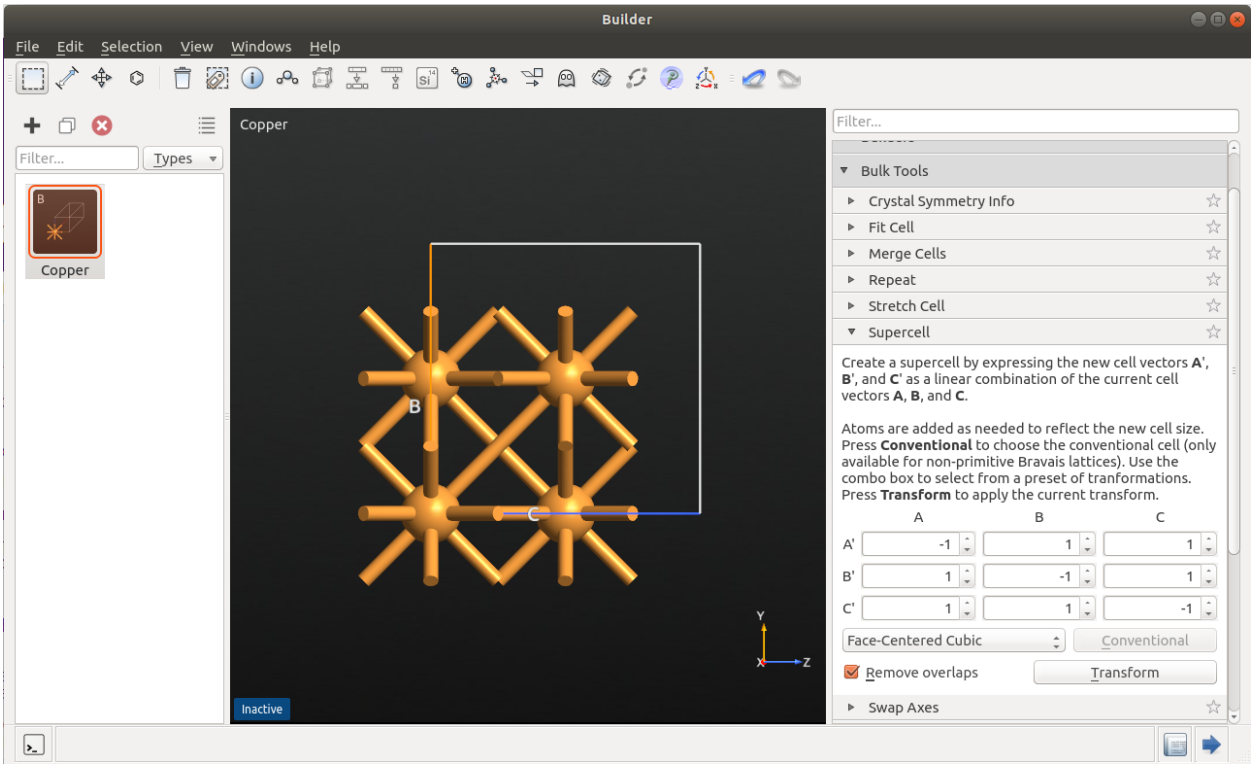
•Open the **Builder** and add a face-centered crystal structure of Copper from Database to the **Stash**.



•In **Bulk Tools**, choose *Supercell* and click on *Conventional*.



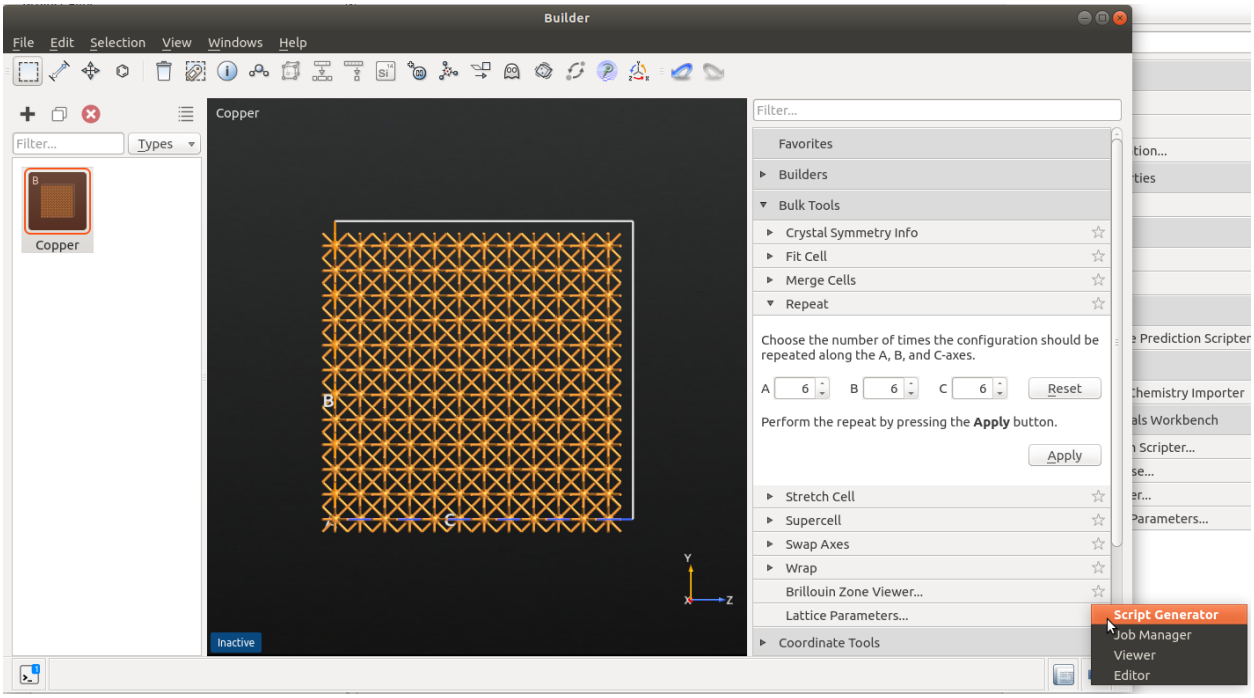
•Then click on *Transform* to convert the primitive cell of bulk copper with 1 Cu atom to the conventional unit cell with 4 Cu atoms.



•In **Bulk Tools**, choose *Repeat* and set the repetition of the conventional unit cell along the A, B, and C axes to 6.

•Click on *Apply* to create a 6x6x6 cubic supercell of bulk copper, which is the initial structure for MD simulations.

•Send the copper supercell structure to the **Script Generator** with the **Send To** icon .



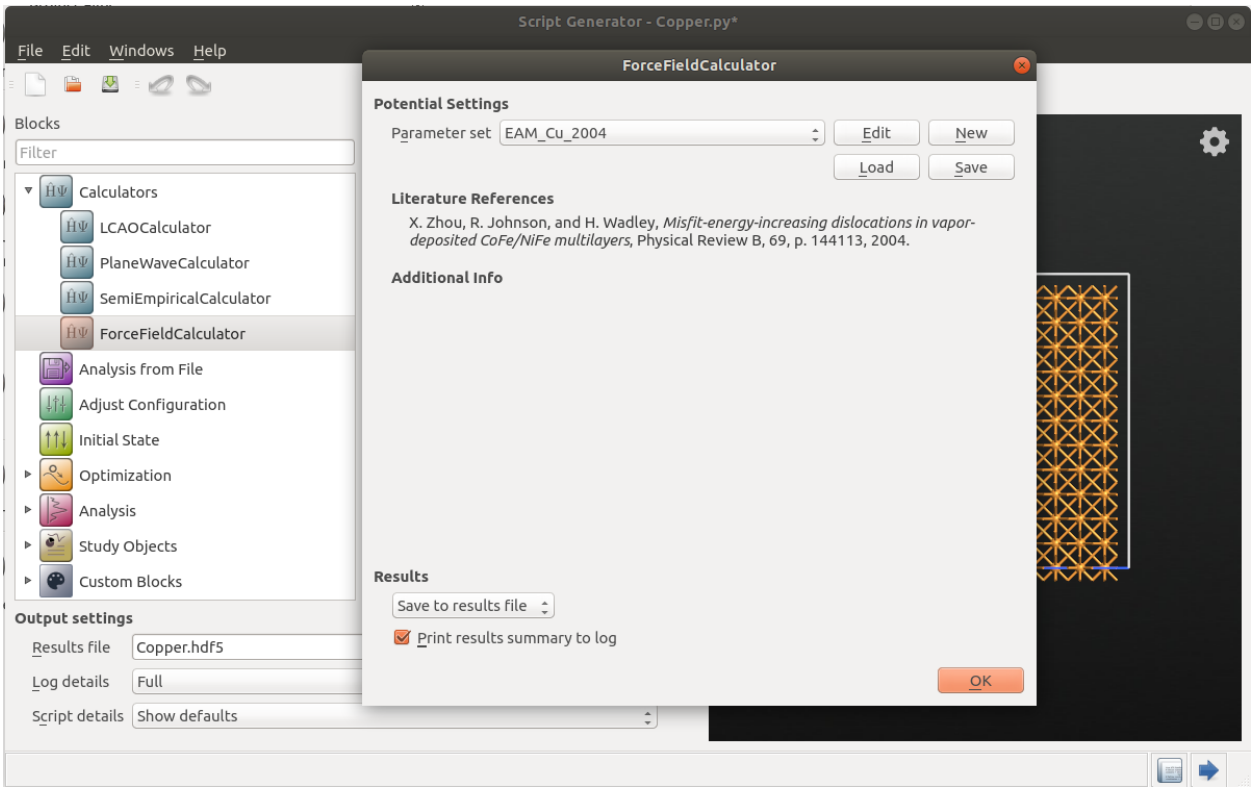
Setting up MD Simulations in the Scripter

•Add a *Calculators* → *ForceFieldCalculator* .

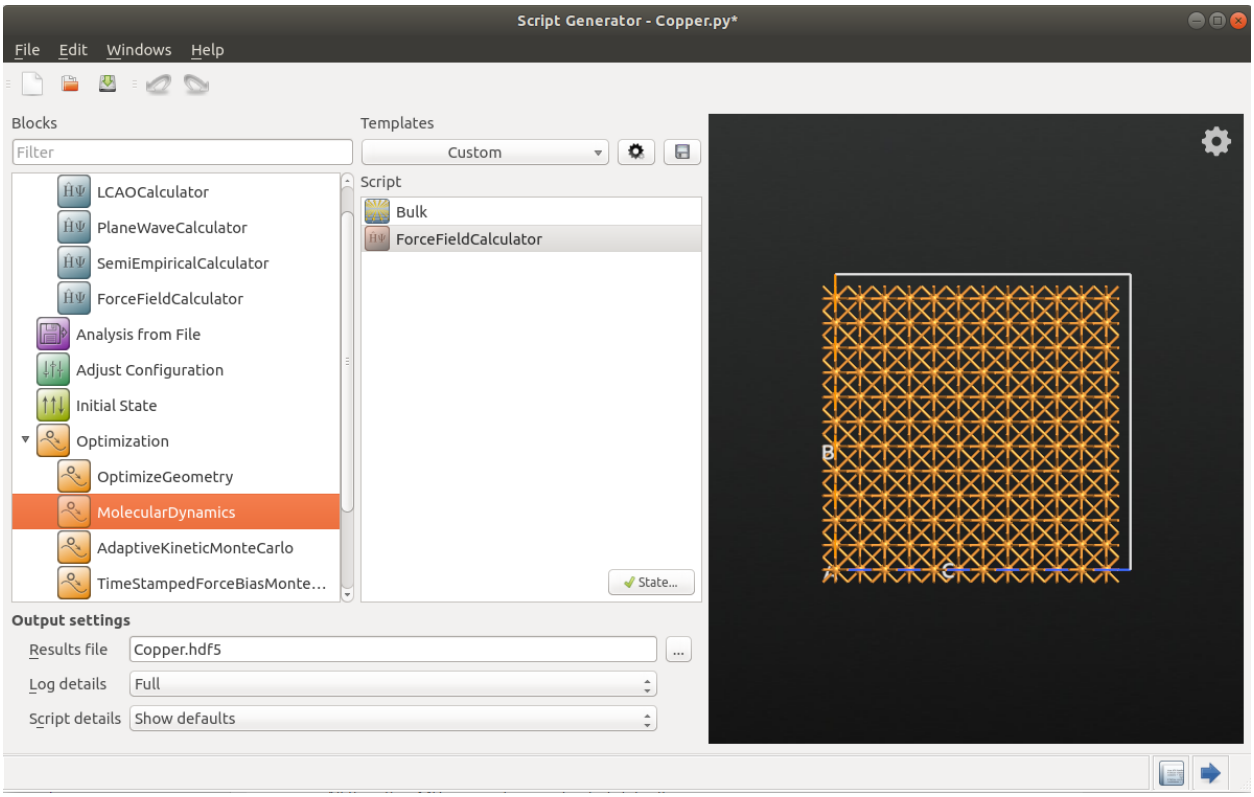
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

•In the Potential Settings, set the classical potential in Parameter set to EAM_Cu_2004[2].

[2] X.ăW. Zhou, R.ăA. Johnson, and H.ăN.ăG. Wadley. Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers. *Phys. Rev. B*, 69:144113, Apr 2004. doi:10.1103/PhysRevB.69.144113.



•Add an *Optimization → MolecularDynamics* object to the Script to set up the first step of the MD procedure, which is *heating crystalline copper* above its melting point.



Set the computational parameters for MolecularDynamics to heat up the supercell structure of crystalline copper from

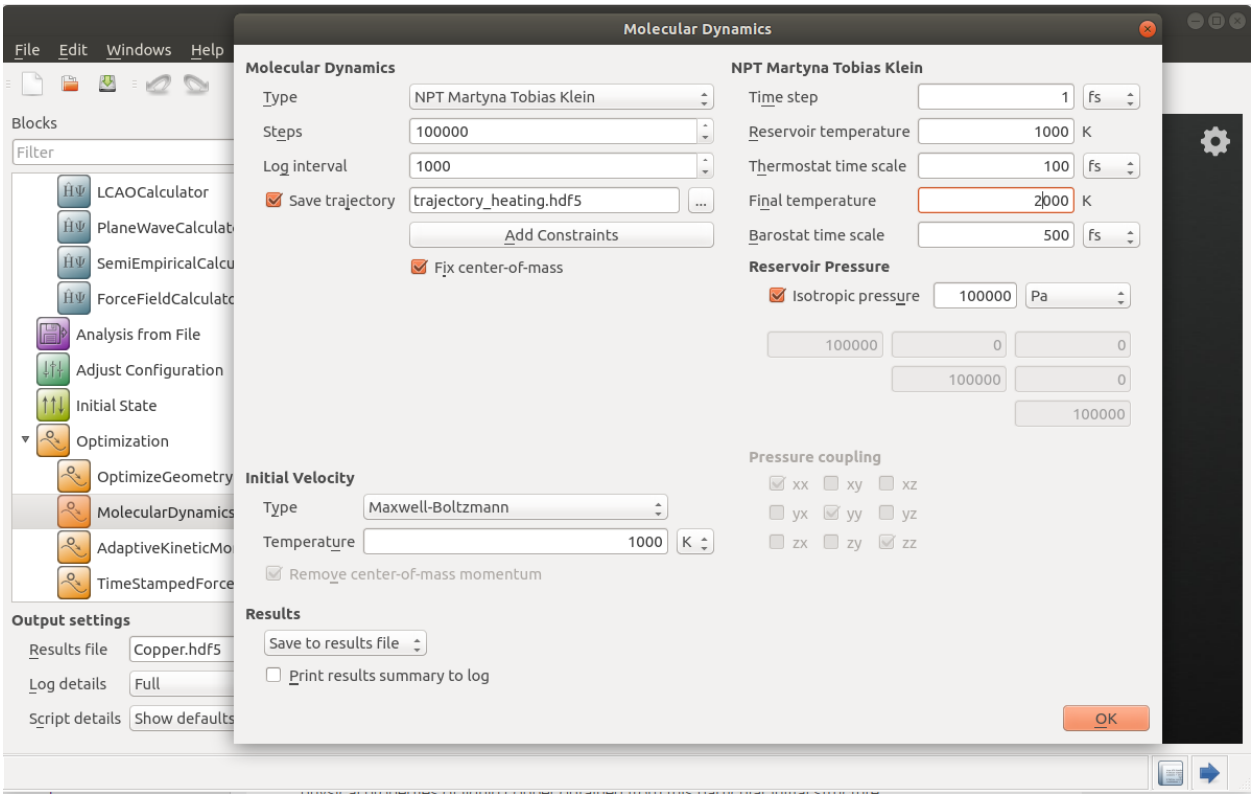
1000 to 2000 K, using an NPT barostat. This procedure allows us to increase the temperature of the system gradually above its melting point. It means that the ordered solid will eventually become a liquid, which only exhibits a short-range order as demonstrated using the radial distribution function (RDF) in the Analysis section.

•Set Molecular Dynamics Type to NPT Martyna Tobias Klein, Steps to 100000, Log interval to 1000, and Save trajectory to the `trajectory_heating.hdf5` file.

•Set Initial Velocity Type to Maxwell-Boltzmann, and Temperature to 1000 K.

•In the NPT Martyna Tobias Klein barostat settings, set Reservoir temperature to 1000 K, and Final temperature to 2000 K.

•All the other MD parameters are kept at default.



•Add a second *Optimization* → *MolecularDynamics* object to the script to set up the second step of the MD procedure, which is *annealing liquid copper* to bring the system to equilibrium.

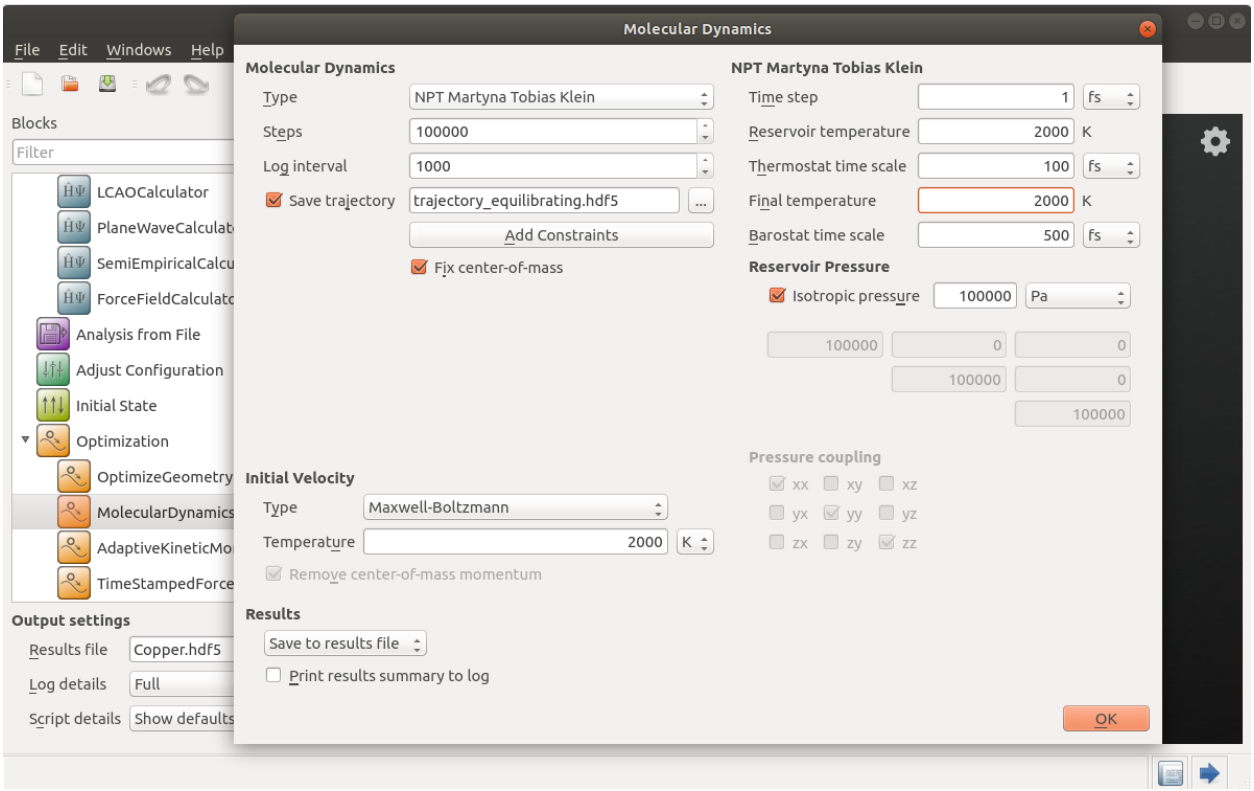
We should set the computational parameters for MolecularDynamics to equilibrate the supercell structure of liquid copper at the constant temperature 2000 K, using an NPT barostat. This step is needed to eliminate any memory effect of the initial copper structure (solid copper) on the physical properties of liquid copper obtained from this particular initial structure.

•Set Molecular Dynamics Type to NPT Martyna Tobias Klein, Steps to 100000, Log interval to 1000, and Save trajectory to the `trajectory_equilibrating.hdf5` file.

•Set Initial Velocity Type to Maxwell-Boltzmann, and Temperature to 2000 K.

•In the NPT Martyna Tobias Klein barostat settings, set Reservoir temperature and Final temperature to 2000 K.

•All the other MD parameters are kept at default.



•Add a third *Optimization* → *MolecularDynamics* object to the script to set up the third step of the MD procedure to collect sufficient statistical data for calculating the self-diffusion coefficient of liquid copper.

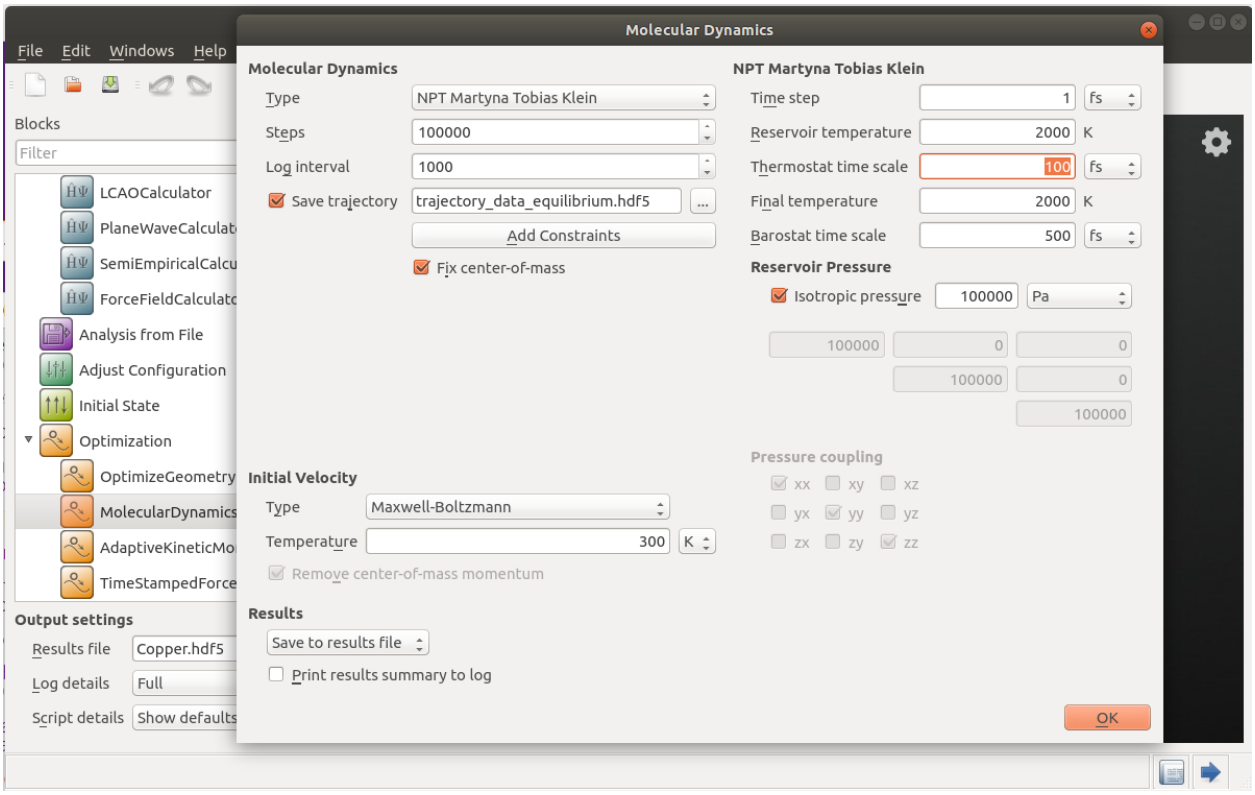
Set the computational parameters for MolecularDynamics to run the MD simulation for the supercell structure of equilibrated liquid copper at the constant temperature 2000 K, using an NPT barostat. In this step, we aim at collecting a good statistical data (a set of MD frames) for calculating atomic diffusion coefficient of liquid copper.

•Set Molecular Dynamics Type to NPT Martyna Tobias Klein, Steps to 100000, Log interval to 1000, and Save trajectory to the `trajectory_data_equilibrium.hdf5` file.

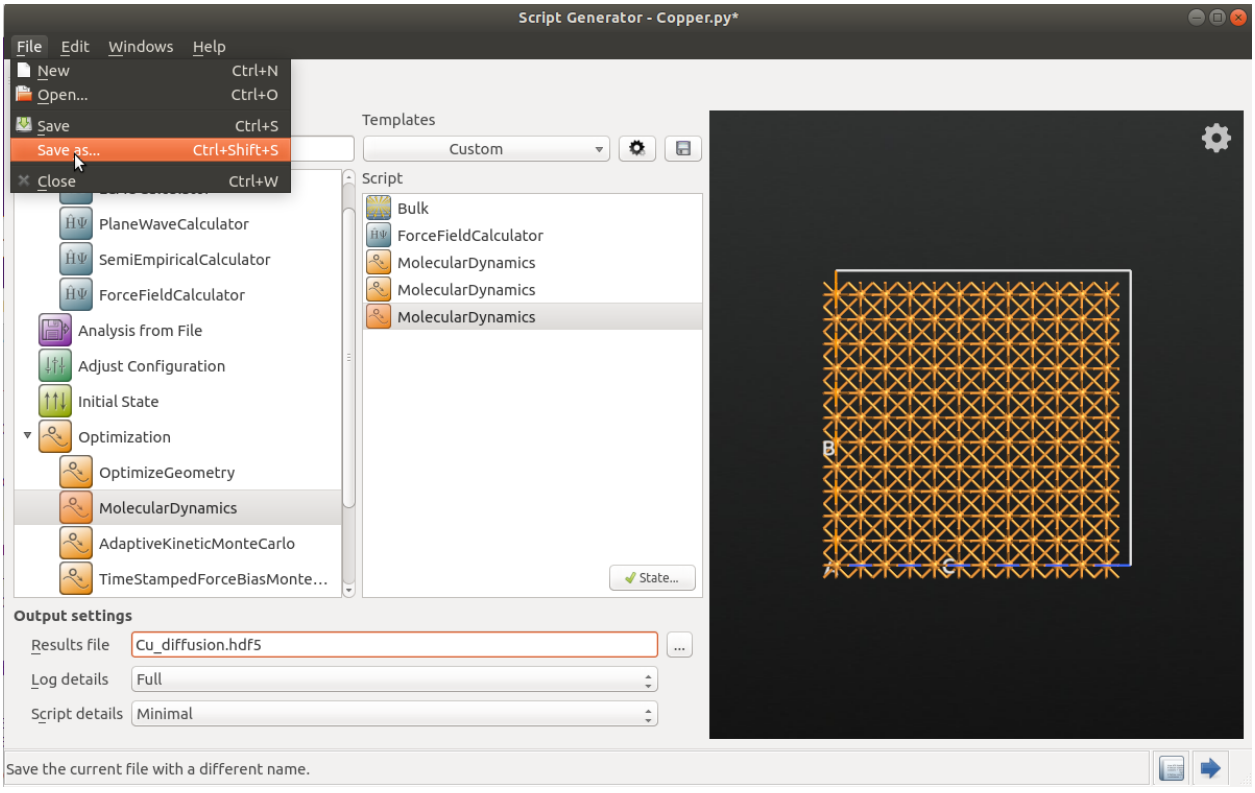
•Set Initial Velocity Type to ConfigurationVelocities, so that the velocities from the previous (second) MD run will be used as initial velocities for this (third) MD simulation.

•In the NPT Martyna Tobias Klein barostat settings, set Reservoir temperature and Final temperature to 2000 K.

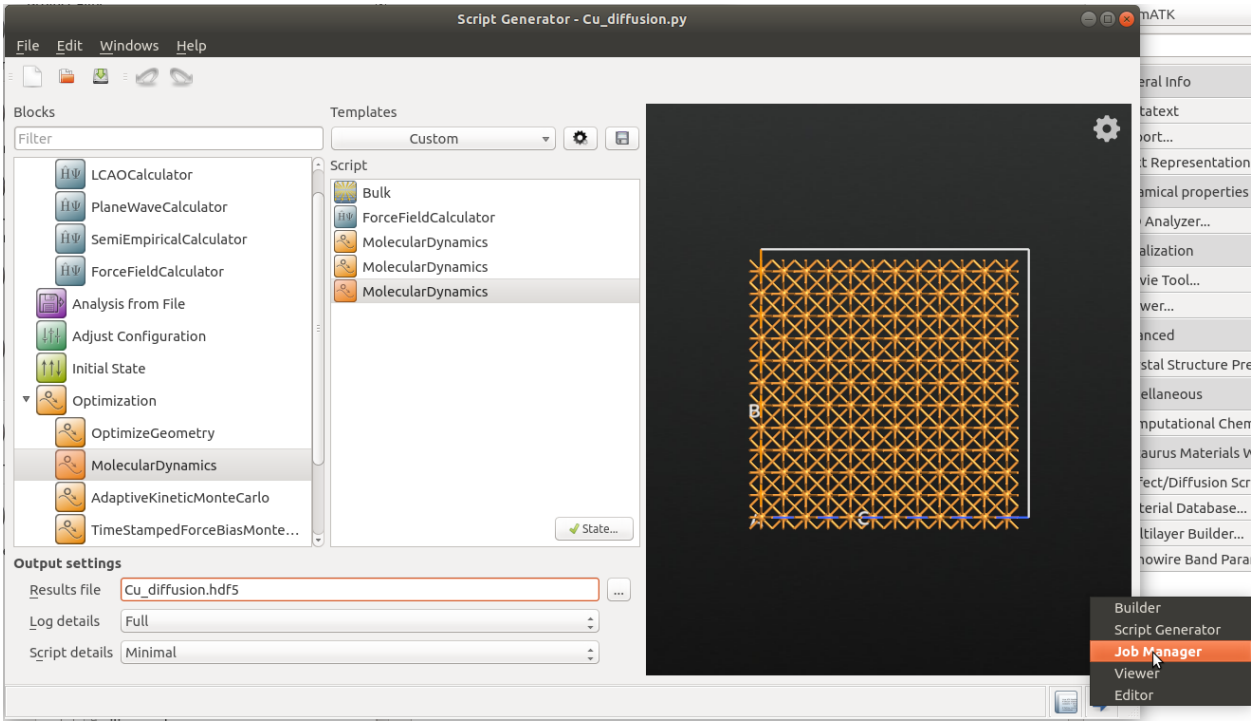
•All the other MD parameters are kept at default.



•In Output settings, rename Results file to `Cu_diffusion.hdf5`, and save the Script to *File* → *Save as* `Cu_diffusion.py`.



•Send the job to the **Job Manager** .



•Depending on your machine, it will take about half an hour to do the classical MD simulations for the supercell structure with 864 copper atoms adopted in this tutorial.

Download the scripts here: `Cu_diffusion.py` and `analysis.py`.

Analysis

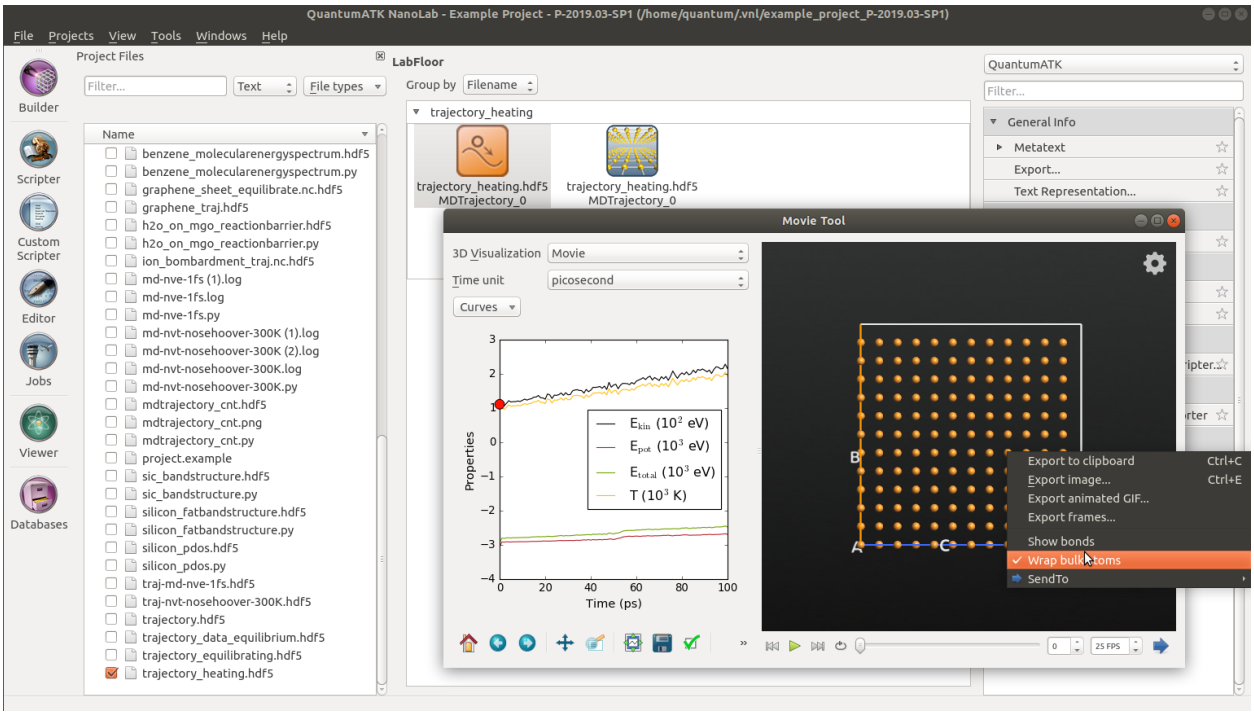
Self-diffusion of Liquid Copper

Once the calculations are done, the requested HDF5 data files should appear in the QuantumATK Project Files list, and their contents should be visible on the LabFloor.

•Select the trajectory named `trajectory_heating.hdf5` and open the **Movie Tool** from the right-hand plugins bar.

The left-hand plot shows the time-evolution of the MD quantities: kinetic energy, potential energy, total energy, and temperature, all as a function of MD simulation time. On the right you can see a movie of the MD trajectory.

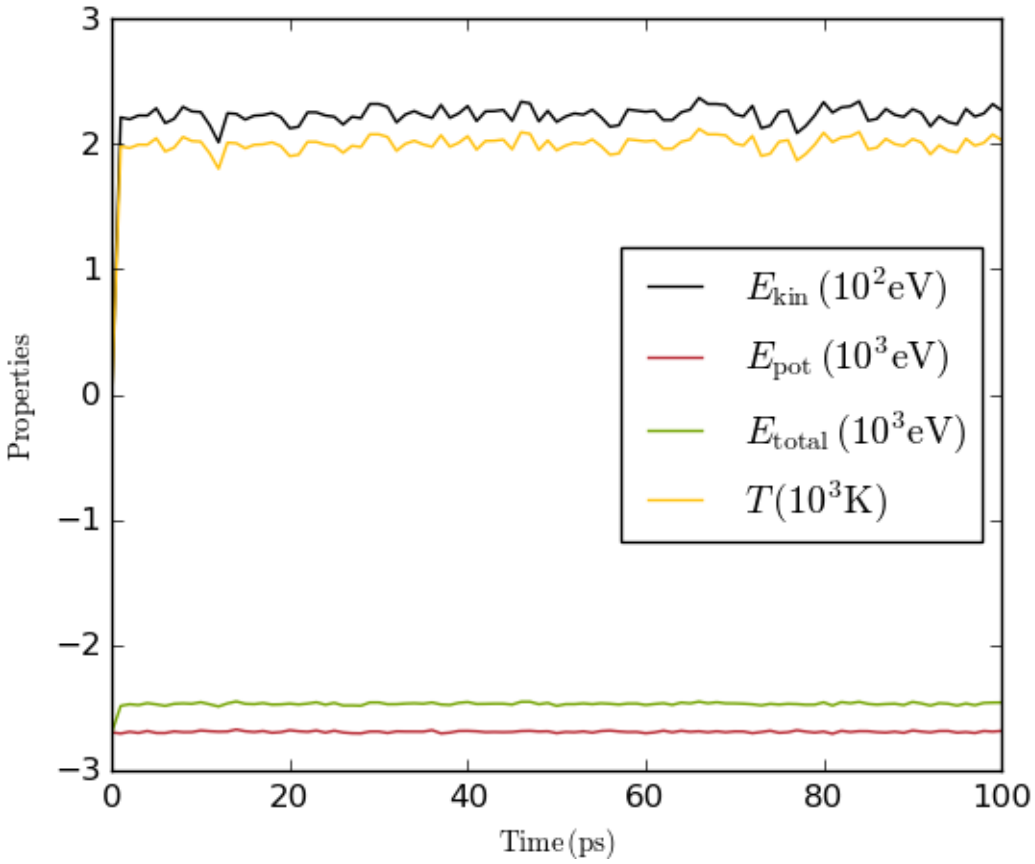
•Use mouse right-click to tick the option **Wrap bulk atoms**, and then click on the ▶ icon to start the movie.



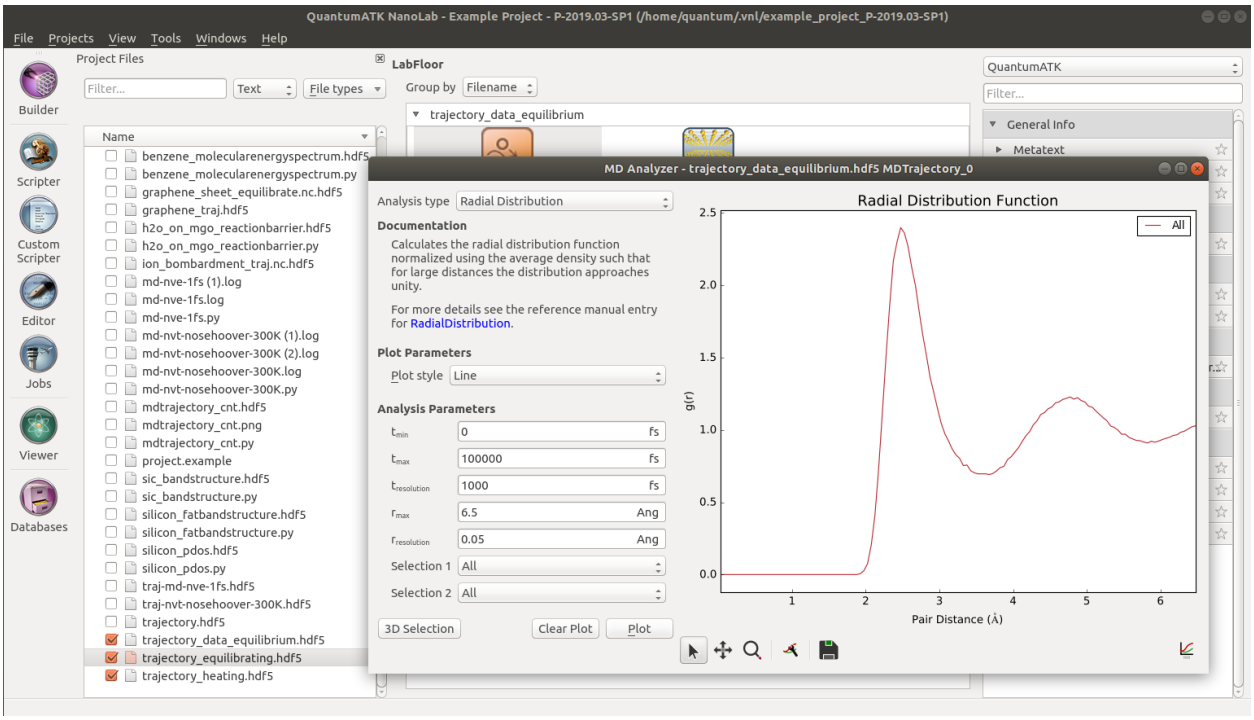
As shown in the animation of the MD trajectory, the average velocity of the copper atoms increases at elevated temperatures, and then the phase transition from the crystalline to liquid copper occurs about half-way through the heating MD run.

Tip: You can easily save such an animated GIF file yourself: Right-click the movie and select **Export animated GIF**.

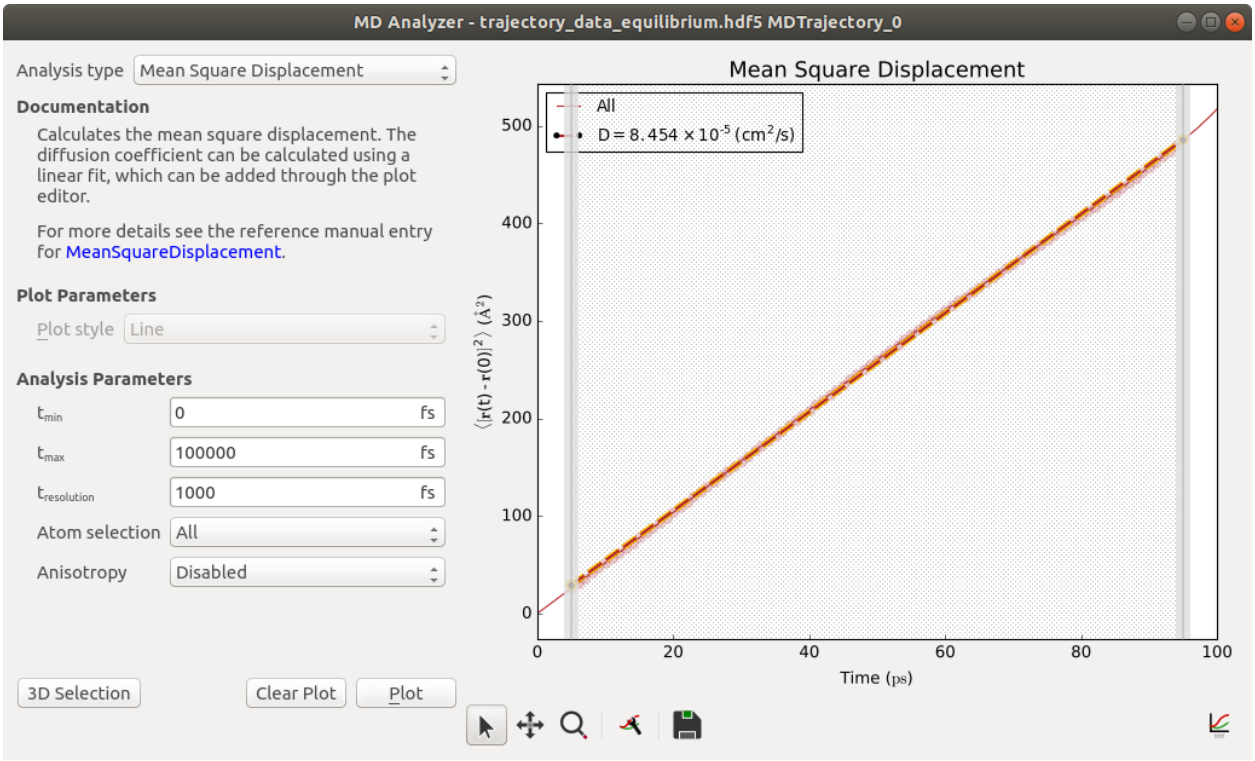
You can also use the Movie Tool to inspect the trajectory for the equilibration MD run, `trajectory_equilibrating.hdf5`. Note that the temperature and energy are fairly constant throughout the equilibration of the system:



As mentioned in the Computational procedure section, an ordered solid transforms into a liquid phase with a short-range order preserved upon melting. To demonstrate that liquid copper exhibits some structural ordering, select the final MD trajectory, `trajectory_data_equilibrium.hdf5`, and open the **MD Analyzer** from the right-hand plugins bar. Select *Radial Distribution* in the drop-down menu, and click on *Plot* to generate a plot of the radial distribution function $g(r)$ for the liquid copper. The distribution function shows a pronounced peak at 2.5 Å, indicating that this is the dominant nearest-neighbor distance between atoms in the liquid copper. This value is comparable to the nearest-neighbor bond distance of 2.55 Å in crystalline copper at room temperature, suggesting that there still exists some short-range order in the liquid phase of copper.



As described in the Theory section, the diffusion coefficient is related to the mean-square displacement (MSD) of the Cu atoms as the final MD simulation progresses. You can plot the MSD using the MD Analyzer: Select *Mean Square Displacement* as the quantity to be plotted, and then click on *Plot*.



The MSD increases linearly with simulation time, and all you need to compute the diffusion coefficient is the slope of the line. Right-click on the line, select Add fit, and click Polynomial fit. Close the Plot editor.

The atomic diffusion coefficient in liquid copper calculated at 2000 K is 0.85 Å^2/ps.

Diffusion of a Single Atom in Liquid Copper

To demonstrate how the diffusion coefficient, D_{imp}, of an individual atom in liquid can be obtained within the framework of the MD approach, we will look at the diffusion of a single copper atom in the liquid copper, pretending that this (randomly-chosen) atom differs from all the other copper atoms in the system. In this particular case, one would expect that the value of the atomic diffusion coefficient for this individual atom should be close to the self-diffusion coefficient value of liquid copper.

The MD trajectory that has already been obtained for liquid copper can be directly adopted for calculating atomic diffusion of a single copper atom. The MSD of a single atom also increases linearly with simulation time, at least on the short-time scale as seen in the figure given below, and all you need to compute the diffusion coefficient is the slope of the line.

Use the script `analysis_single_atom.py` for computing the diffusion coefficient. The content of the script is shown below. The script takes as an argument the name of the HDF5 file `trajectory_data_equilibrium.hdf5` from which it extracts the MD trajectory of a **single copper atom** (atom #7 as chosen in the script) in the equilibrated liquid copper, then uses the built-in function `MeanSquareDisplacement()` to calculate the MSD of the copper atom #7, and uses *NumPy* to do the linear fitting. Finally, a plot is produced using *pylab*.

```
# Get the name of the MD trajectory (hdf5) file given in the system arguments
filename = sys.argv[1]
#filename = 'trajectory_data_equilibrium.hdf5'

# Get the frames of the MD trajectory from the file
md_trajectory = nlread(filename)[-1]

# Get Mean Square Displacement (MSD) for a single Cu atom (#7) from the MD trajectory
msd = MeanSquareDisplacement(md_trajectory,atom_selection=[7])

# Get the times in ps and the MSD values in Ang**2.
t        = msd.times().inUnitsOf(ps)
msd_data = msd.data().inUnitsOf(Angstrom**2)

# Fit the slope of the MSD to estimate the diffusion coefficient.
# If you discover non-linear behavior at small times, you should then discard this␣
↪initial part in the fit.
for i in range(10,110,10):
  first_image = 0
  a = numpy.polyfit(t[first_image:i], msd_data[first_image:i], deg=1)
  print(i, a[0]/6)

# Setting the endpoint to 25 is based on a visual inspection of the plot for MSD(t).
# Note that this kind of plots may have some noise for large time-values.
endpoint = 25
a = numpy.polyfit(t[first_image:endpoint], msd_data[first_image:endpoint], deg=1)
line_fit = numpy.poly1d(a)

# Calculate the diffusion coefficient of the copper atom #7 in Ang**2/ps.
diffusion_coefficient = a[0]/6.0
print('Diffusion coefficient for a single atom in liquid copper at T=2000K is: ',
↪diffusion_coefficient, ' Ang**2/ps')
```

(continues on next page)

(continued from previous page)

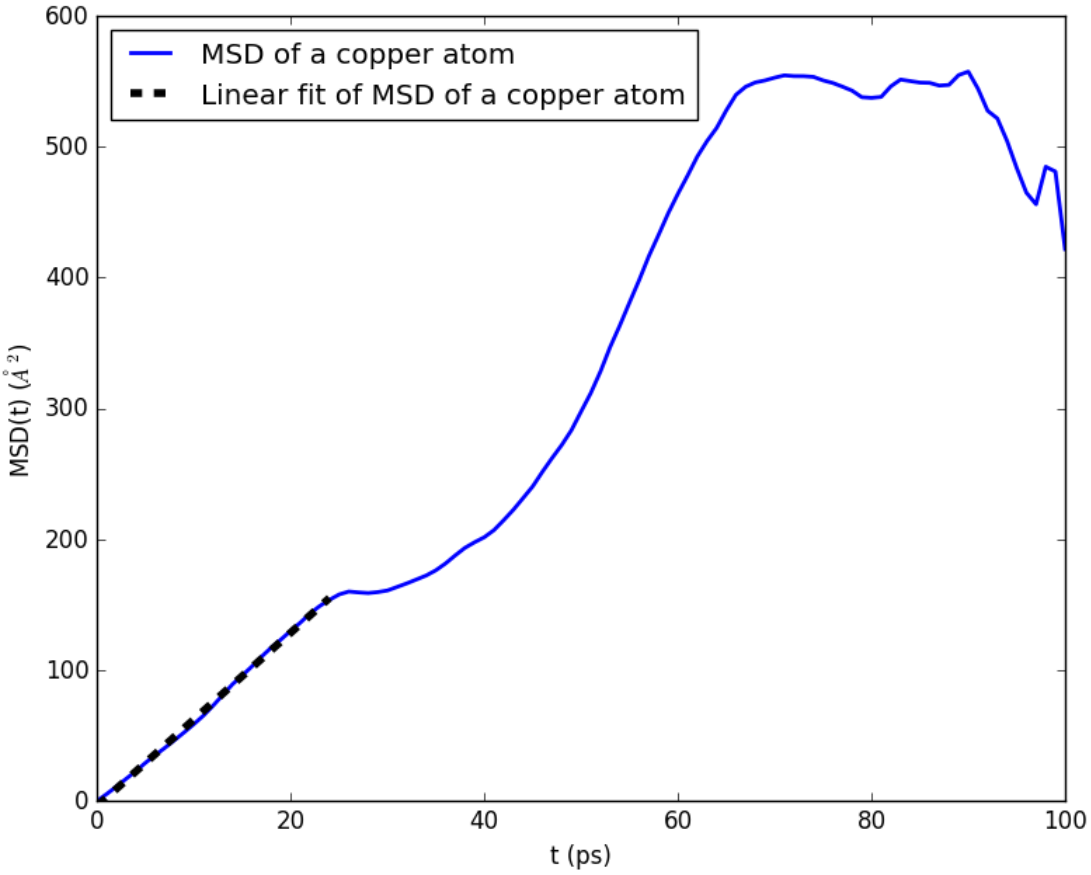
```
# Plot the data using pylab.
import pylab

pylab.plot(t, msd_data, linewidth=2,label='MSD of a copper atom')
pylab.plot(t[first_image:endpoint], line_fit(t[first_image:endpoint]), linewidth=4,color=
↪'k', linestyle='--', label='Linear fit of MSD of a copper atom')
pylab.xlabel('t (ps)')
pylab.ylabel('MSD(t) ($\AA^{2}$)')
pylab.ylim(0,600)
pylab.legend(loc='upper left')

pylab.show()
```

Download the script and execute it from the command line. The calculated diffusion coefficient of a single atom will be printed at the end of the logging output:

```
$ atkpython analysis_single_atom.py trajectory_data_equilium.hdf5
...
...
...
Diffusion coefficient for a single atom in liquid copper at T=2000K: 1.10459718032␣
↪Ang**2/ps
```

Tip: This python script used for the analysis can also be executed from the Job Manager, i.e., without using the command line. For that, replace filename = sysargv[1] with filename = trajectory_data_equilibrium.hdf5 in the downloaded *analysis_single_atom.py* script, save it, and send it to the Job Manager to execute it as a regular QuantumATK job.

The diffusion coefficient of a single atom in liquid copper calculated at 2000 K is found to be 1.10 Å^2/ps in this case. As expected, this value is comparable to that of the self-diffusion coefficient of liquid copper, 0.89 Å^2/ps. The difference between these two calculated coefficients originates from the fact that we have done ensemble-averaging over all the copper atoms in liquid to calculate the self-diffusion coefficient, D, whereas there is only one data point at a given time, t, for calculation of the atomic diffusion coefficient, D_{imp}. So, we may need to run MD simulations for much longer time (T_{MD}) to more accurately calculate the self-diffusion coefficient of liquid copper from the MD trajectory of a single copper atom. This longer MD trajectory should allow us to get better statistics and increase the observation time interval (t) where the mean-square displacement exhibits a linear time dependence. The linear region in the following figure, which shows the time-dependent mean-square displacement, is then expected to increase from $t = 25$ ps to a larger observation time value.



Note: When analyzing only a single atom for this amount of time, the statistics are not sufficient for obtaining the diffusion constant. If you want the diffusion constant from a single atom, the simulation needs to run much longer than if the averaging is done over many atoms.

Temperature Dependence of Diffusion Coefficient

As an exercise, you may repeat all the steps of the MD procedure to obtain the MD trajectory of liquid copper equilibrated at T=1620 K, and then calculate the self-diffusion coefficient from this MD trajectory. You will find that this physical quantity calculated without any adjustable parameters, i.e., calculated from first-principles, agrees very well with the experimental value of 0.52 Å^2/ps measured for liquid copper at T=1620 K. The smaller diffusion coefficient value at 1620 K compared to that at 2000 K suggests that the self-diffusion coefficient decreases upon lowering the temperature of liquid, as expected.

References

3.8.9 Simulating a creep experiment of polycrystalline copper

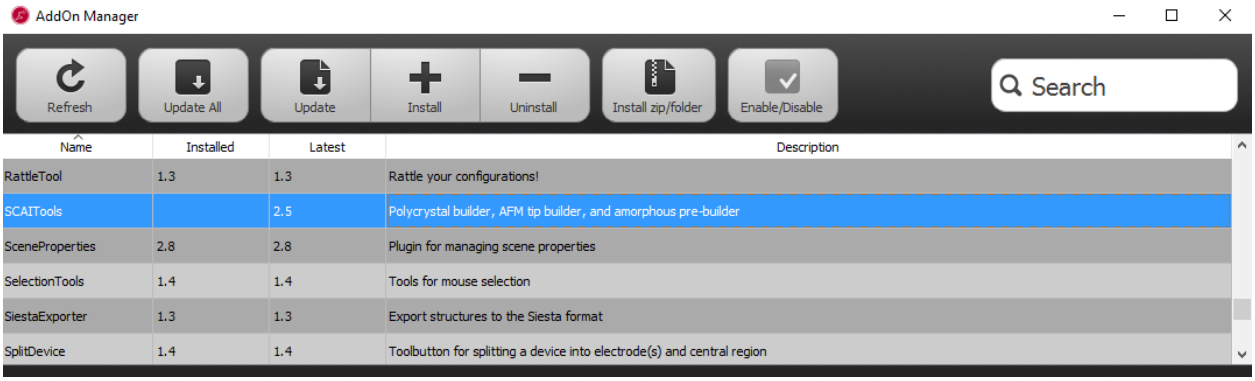
Version: 2016.3

This tutorial will show how to set up, run, and analyze a molecular dynamics (MD) simulation of a microscopic creep experiment using polycrystalline copper, *i.e.* the strain response to an externally imposed uniaxial stress[1]. You will get to know the **Polycrystal Builder** and the **Local Structure** analysis tool, and you will learn how to run **advanced MD simulations** at constant external stress.

Although the construction of realistic grain sizes of several nanometers is in principal possible, the computational cost of such simulations is relatively large, as the required system size can easily exceed a million atoms. Such calculations are certainly possible with QuantumATK, but for demonstration purposes you will here simulate only a small model system, to reduce the calculation time. Still, the presented methodology can be transferred in a straightforward manner to larger systems.

Installing the polycrystal builder plugin

First, make sure that the Polycrystal plugin is installed and available in the **Builder** – it should appear in the Builder panel under *Builders* → *Polycrystal (Voronoi)*. If the plugin is not listed, you have to install it via *Help* → *AddOn Manager*. Select the **SCAITools** package and install it.



You may have to restart QuantumATK to activate the plugin in the Builder.

Building the polycrystalline cell

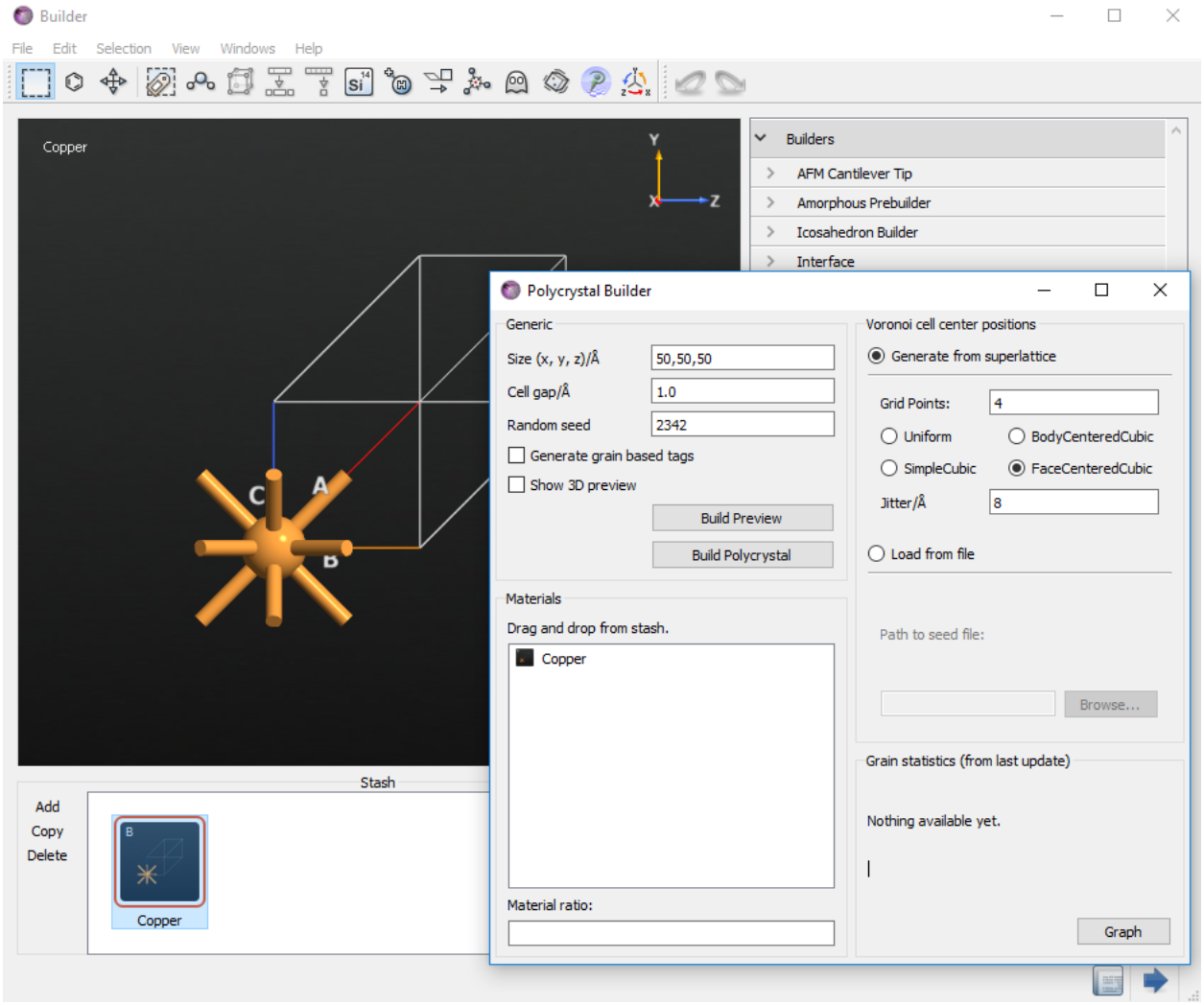
Create a new project in QuantumATK, call it for instance Polycrystalline Copper.

Open the **Builder** and add the fcc copper single crystal material to the **Stash** using *Add* → *From Database*.

To start building the polycrystalline structure, select *Bulk Tools* → *Polycrystal (Voronoi)*. Set the crystal size to 50, 50, 50 Å to obtain an cubic supercell. Set the gap between grains to a small value, like 0.1 Å, to allow for a compact packing of the grains. Drag and drop the cubic copper cell from the Stash to the **Materials** field. Make sure the *Generate grain based tags* option is unticked. As an advanced analysis feature, this option allows you to mark the atoms within each grain by different tags, so that the transformation of the grains during the simulation can be monitored. We will not use this feature in this tutorial.

[1] Yun-Jiang Wang, Akio Ishii, and Shigenobu Ogata. Grain Size Dependence of Creep in Nanocrystalline Copper by Molecular Dynamics. *MATERIALS TRANSACTIONS*, 53(1):156–160, 2012. doi:10.2320/matertrans.MD201122.

You can choose how the seeds for the grains are created. Select **Generate from superlattice** and set the number of grid points to 4. Select *FaceCenteredCubic* as the superlattice of the grain structure, and set the value of *Jitter* to 8 Å. These settings will create four initial grains on a distorted fcc lattice, with a distortion variance of 8 Å.



You can preview the grain structure by clicking **Build Preview**. Now, the 3D-window will show the only the seeds of the grains. You can additionally switch on the visualization of the grain boundaries by checking **Show 3D preview** box.

Click **Build Polycrystal** to initiate the structure generation.

After a few seconds the final polycrystalline cell will appear on the Stash.

Analyzing the grain structure

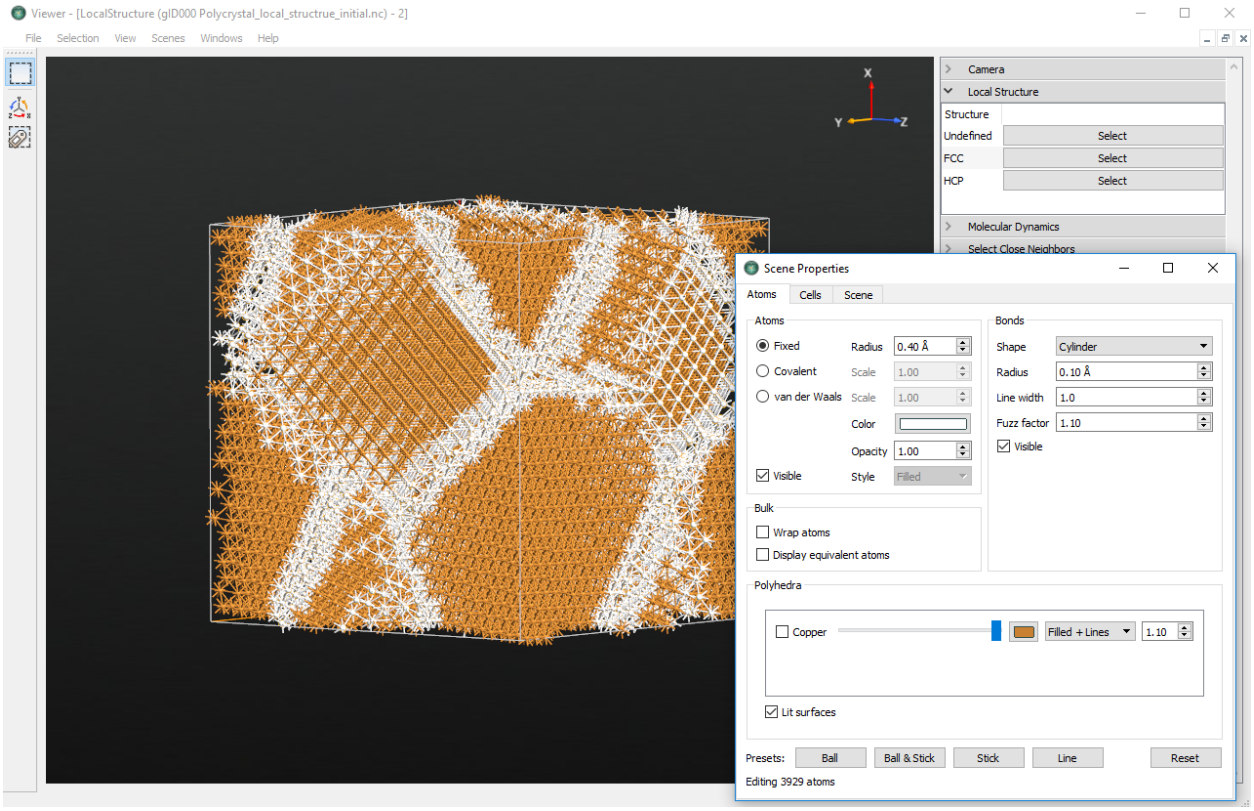
By rotating the cell in the Builder window, you can already see part of the grain structure. A better visualization in terms of the local crystal structure can be performed by using the **LocalStructure** analyzer.

For this, send the polycrystalline cell to **Scripter**, and just left double-click **Analysis** and add the **LocalStructure** (do not add any New Calculator, as you may be used to – we are not going to run any simulation, only analyze the real-space geometry). Change the output filename to something suitable, e.g. `local_structure_init.hdf5`.

Send the script to the **Job Manager** and run it.

After a few seconds, the results will appear on the **LabFloor**. Select the LocalStructure item and click the **Viewer**.

You can now use the **Local Structure** plugin in the Viewer to analyze the crystal structure. You will there find a list of the crystal structure types which have been detected in the polycrystalline cell.



In this example, the cell contains atoms in a local FCC, as well as HCP, or Undefined environment. Since the underlying basic copper crystal forms a fcc strucure, the grains will be of such struture, too. To identify the grain boundaries, click to select all Undefined and HCP atoms (the latter will be only a tiny fraction). Then open the **Properties** plugin and change the color of the selected atoms to e.g. white. Grains as well as grain boundaries will now be clearly visible, as illustrated above.

Tip: Alternatively, you could uncheck the *Visible* box to hide the selected grain-boundary atoms, or perform any other customized graphical modification of the scene.

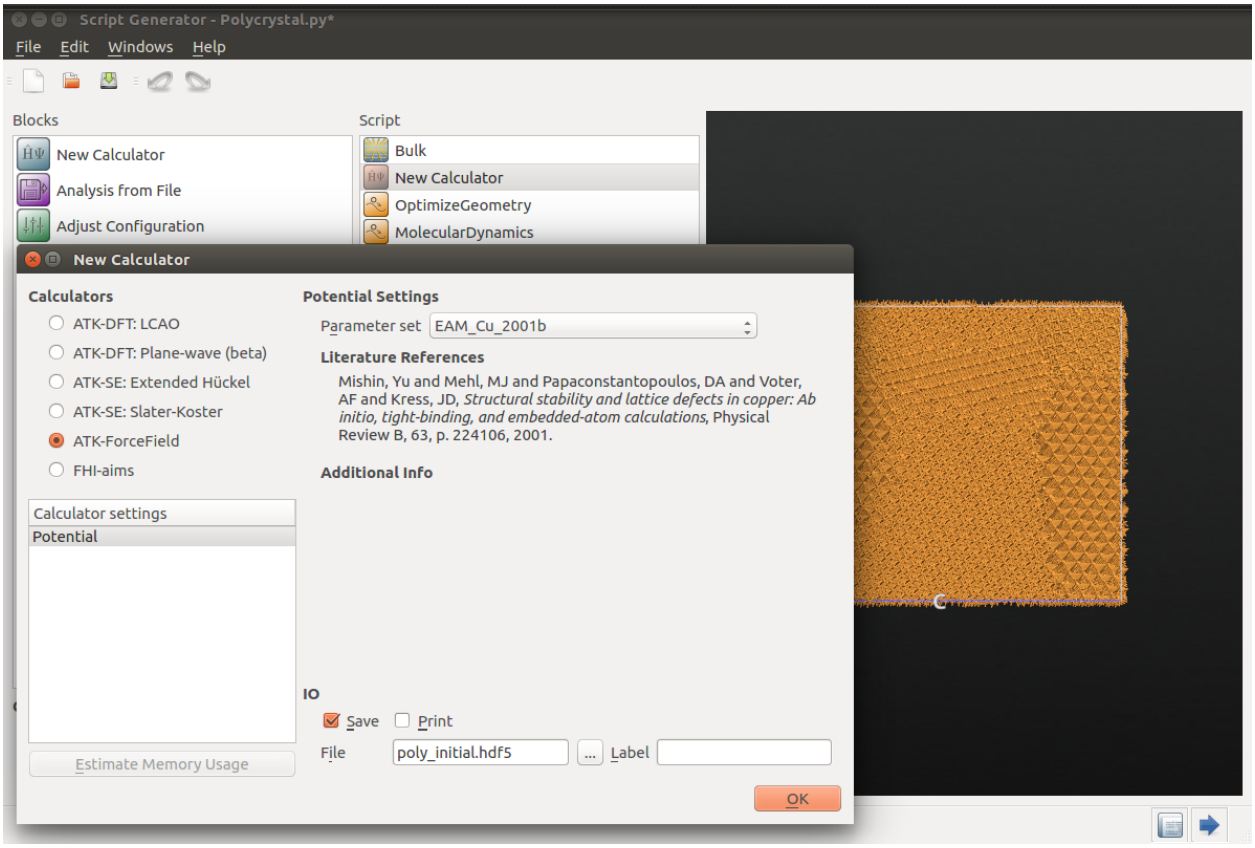
Note: The exact grain structure has been generated randomly and might be different from case to case.

Setting up the creep simulation

You will now set up the MD simulations; first equilibration and then the actual creep simulation. Send the bulk configuration from the **Builder** to the **Scripter**.

Add a **New Calculator** and open it (double-click). Select **ATK-ForceField** in the left panel; the suggested Embedded Atom Model (EAM) potential named *EAM_Cu_2001b*[2] is a suitable choice. Set the filename to `poly_initial.hdf5` to save the initial supercell in a separate HDF5 file – using separate data files can help organize the data. Uncheck the *Print* tick box at the bottom to not clutter the log file too much. Click *OK* to close the Calculator dialog.

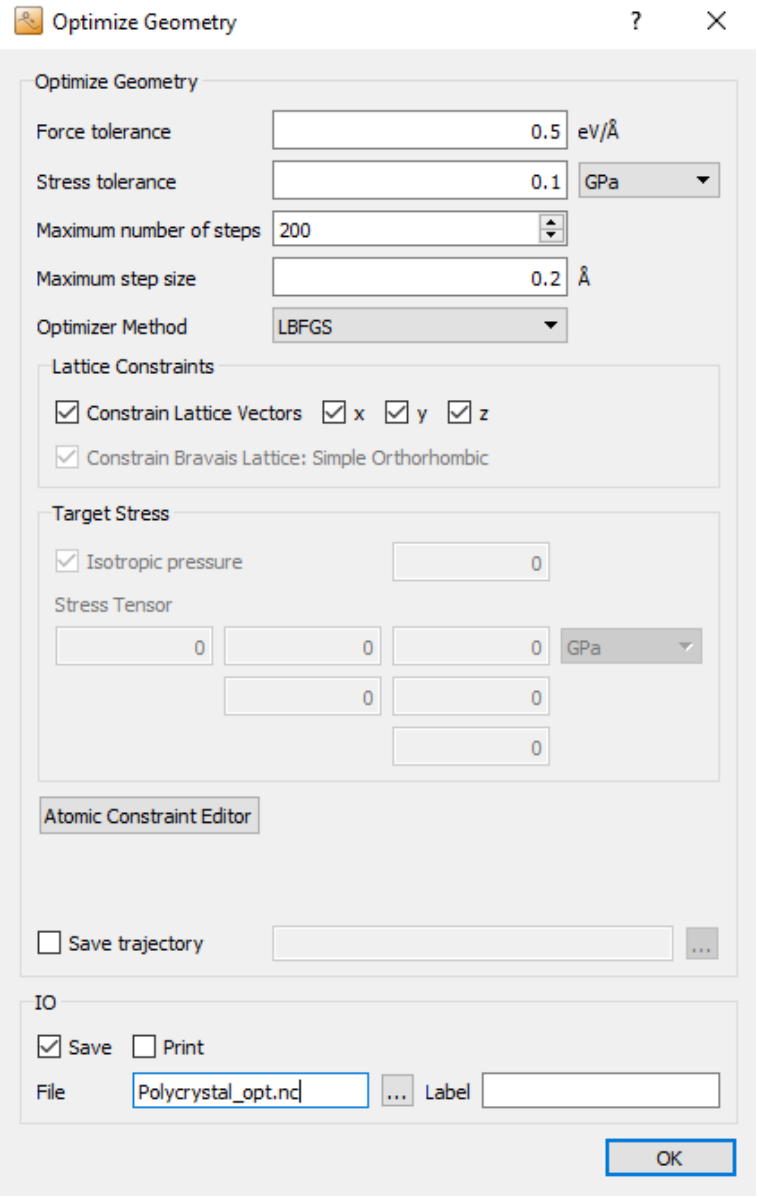
Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.



Next, add the **OptimizeGeometry** block (click *Optimize* → *OptimizeGeometry*). This step will remove possibly large inter-atomic forces due to the building procedure, before running the actual MD simulation. Double-click to open its settings. As we do not require a perfectly force-free structure, the force tolerance can be increased to 0.5 eV/Å, in

[2] Y.ăMishin, M.ăJ. Mehl, D.ăA. Papaconstantopoulos, A.ăF. Voter, and J.ăD. Kress. Structural stability and lattice defects in copper: Ab-initio, tight-binding, and embedded-atom calculations. *Phys. Rev. B*, 63:224106, May 2001. doi:10.1103/PhysRevB.63.224106.

order to speed up the calculation. The remaining parameters can be kept at their default values, although you could remove the checkmarks for *Save* and *Print*.

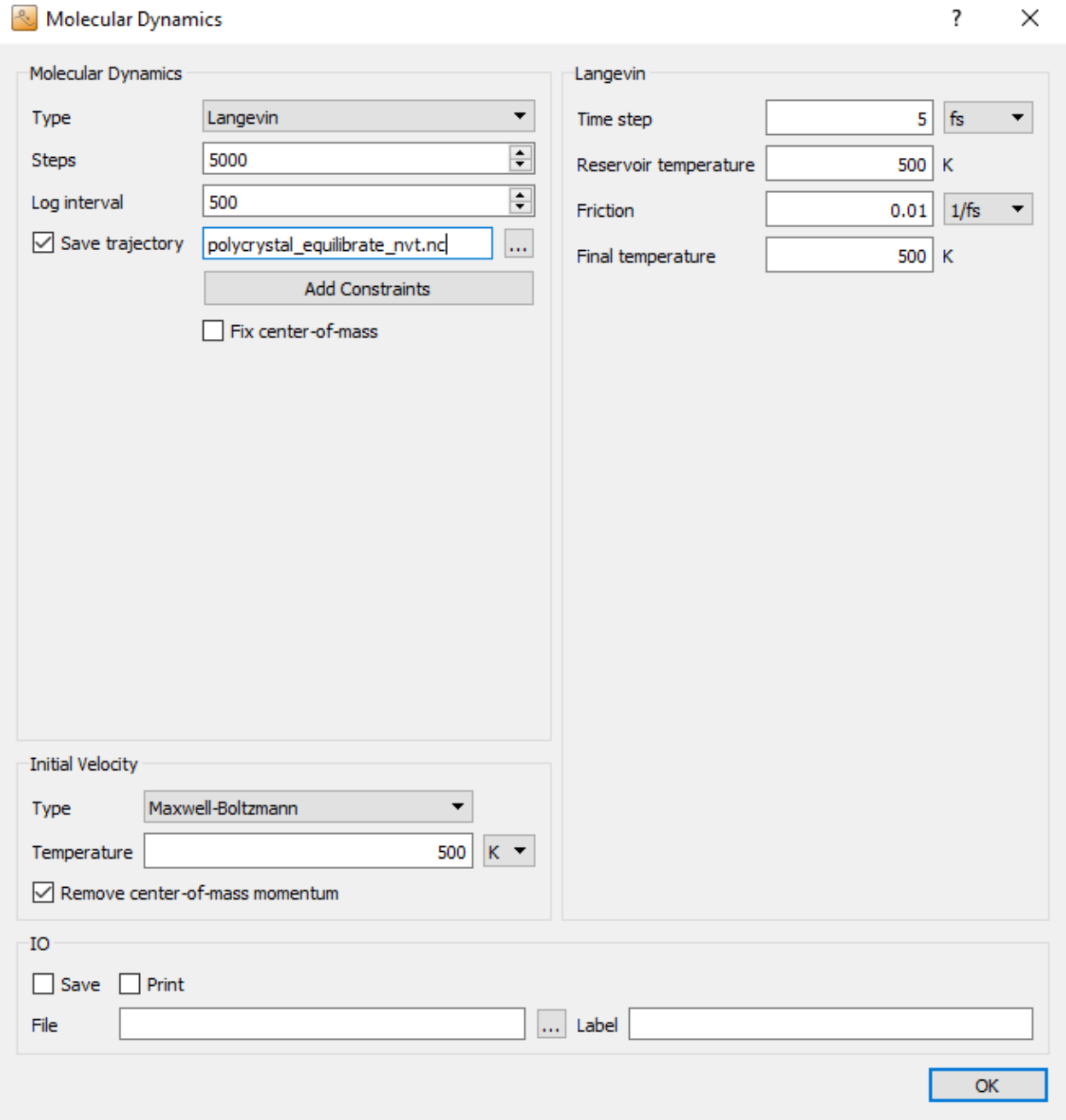


After the geometry optimization, you will want to run two short NVT (constant volume) and NPT (constant pressure) molecular dynamics simulations to get an equilibrated configuration. Therefore, add two **MolecularDynamics** blocks (click *Optimize* → *MolecularDynamics*). Later on, you will add one more MD block for the creep simulation.

In some cases, in particular if the initial configuration is far from equilibrium, directly starting with a constant pressure simulation can make the system to unstable and lead to unreasonable results. A safer approach is to equilibrate the system at constant volume first, before switching on the pressure coupling. To this aim, select Langevin for NVT simulation in the setting of the first MolecularDynamics block, choose the *Maxwell-Boltzmann* initial velocities at a *temperature* of 500 K. You can use a large *time step* of 5 fs for copper, as the atoms are quite heavy and thus the vibrational frequencies are small enough. Be aware, however, that a time step this large might in other cases affect the accuracy of the integration algorithms, so before simulating a new system one should carefully assess the chosen time step by monitoring the energy conservation in an NVE simulation at the desired temperature.

Set both the *reservoir temperature* and the *final temperature* to 500 K. We run the simulation at a slightly elevated temperature to accelerate possible thermally activated processes, to see at least the onset of the creep behaviour in a relatively short simulation time.

Save the trajectory to the file `polycrystal_equilibrate_nvt.hdf5`. Uncheck the *Save* and *Print* boxes at the bottom, as the trajectory is already being saved.

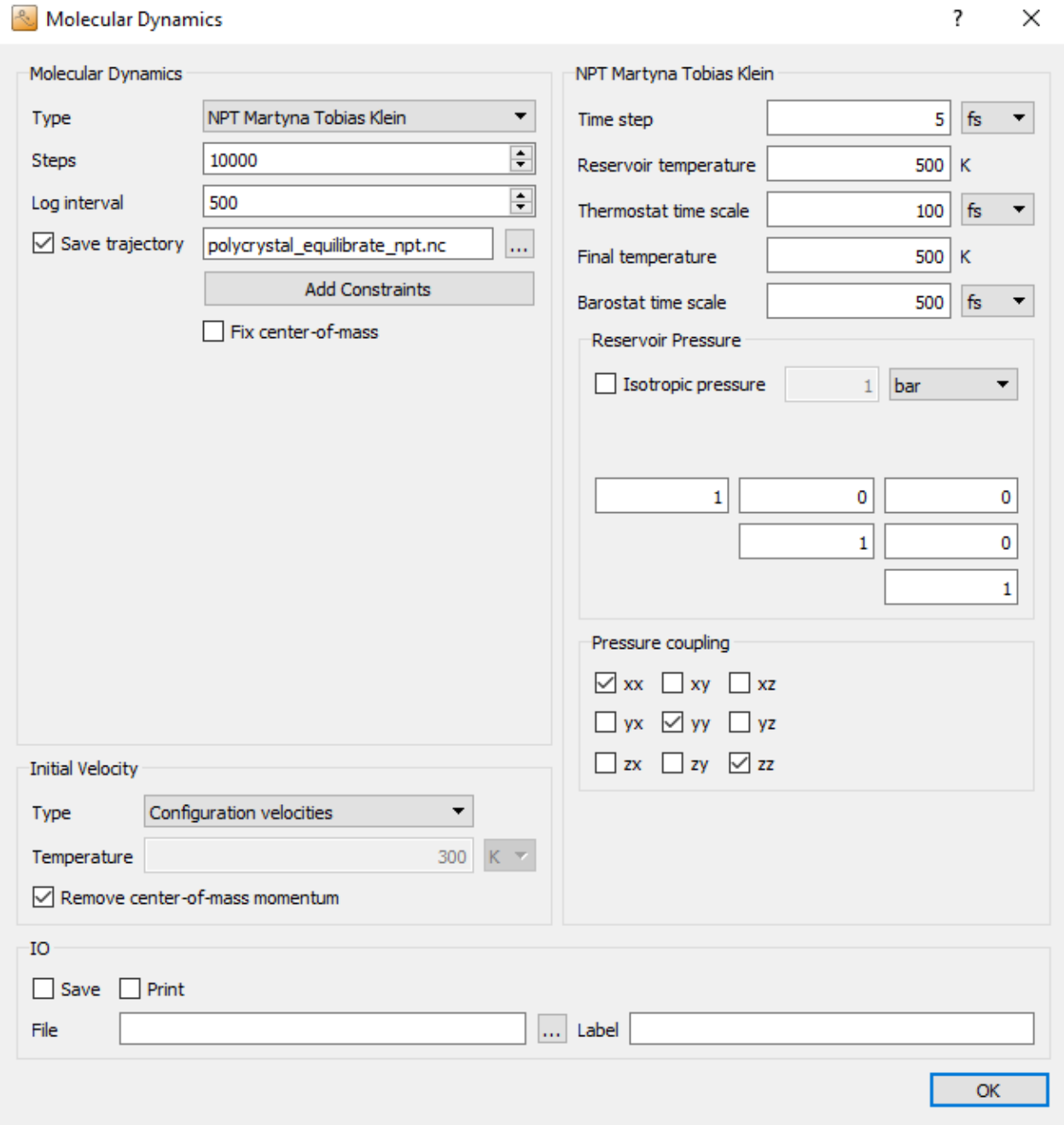


In the next **MolecularDynamics** block, you will perform another equilibration simulation, this time at constant pressure. Therefore, select *NPT Martyna Tobias Klein* MD type, as this method allows to impose an independent pressure coupling to each component of the stress tensor. Set the number of *steps* to 10,000 and the *log interval* to 500. Select an appropriate filename for the trajectory file, e.g. `polycrystal_equilibrate_npt.hdf5`.

Choose *Configuration velocities* as initial velocities.

Choose again a fairly large *time step* of 5 fs and 500 K for the *reservoir* and *final* temperatures. Moreover, make sure that the reservoir pressure is 1 bar.

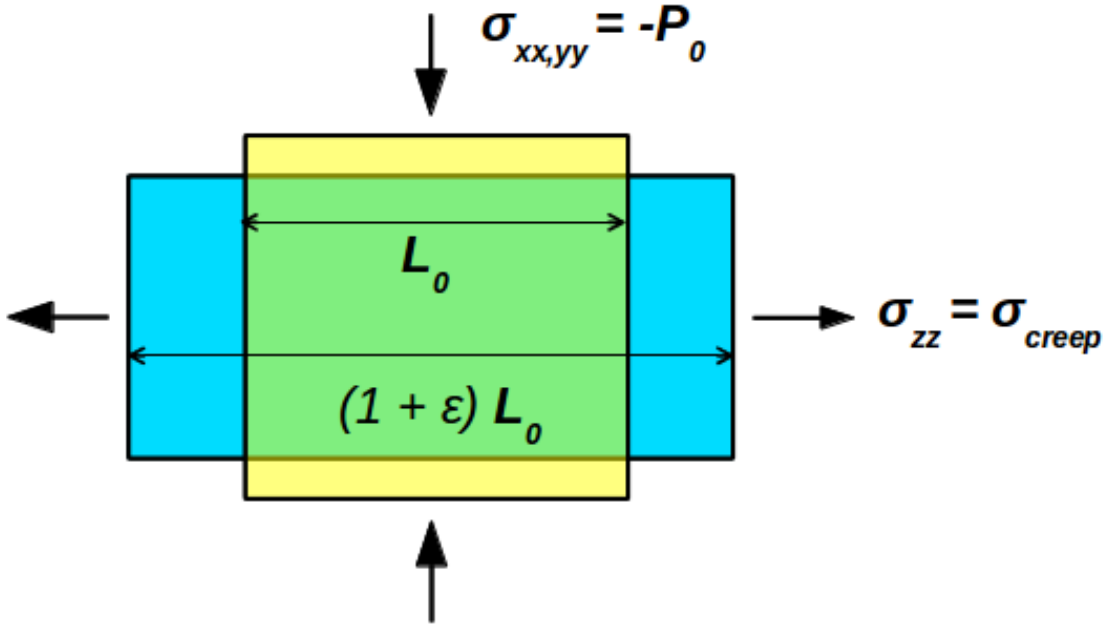
Uncheck **Isotropic pressure** to couple each cell vector independently to the barostat. In the field named *Pressure coupling* you can select which components of the stress tensor the barostat should couple to. Here, you should stick to the default settings, which activate only the hydrostatic stress, without any shear stress components.



Now, for the actual creep simulation, add a third **MolecularDynamics** block. Choose the same settings as in the previous (NPT) MD block, except for the *number of steps* and the *log interval*, which should be set to 40,000 and 250, respectively.

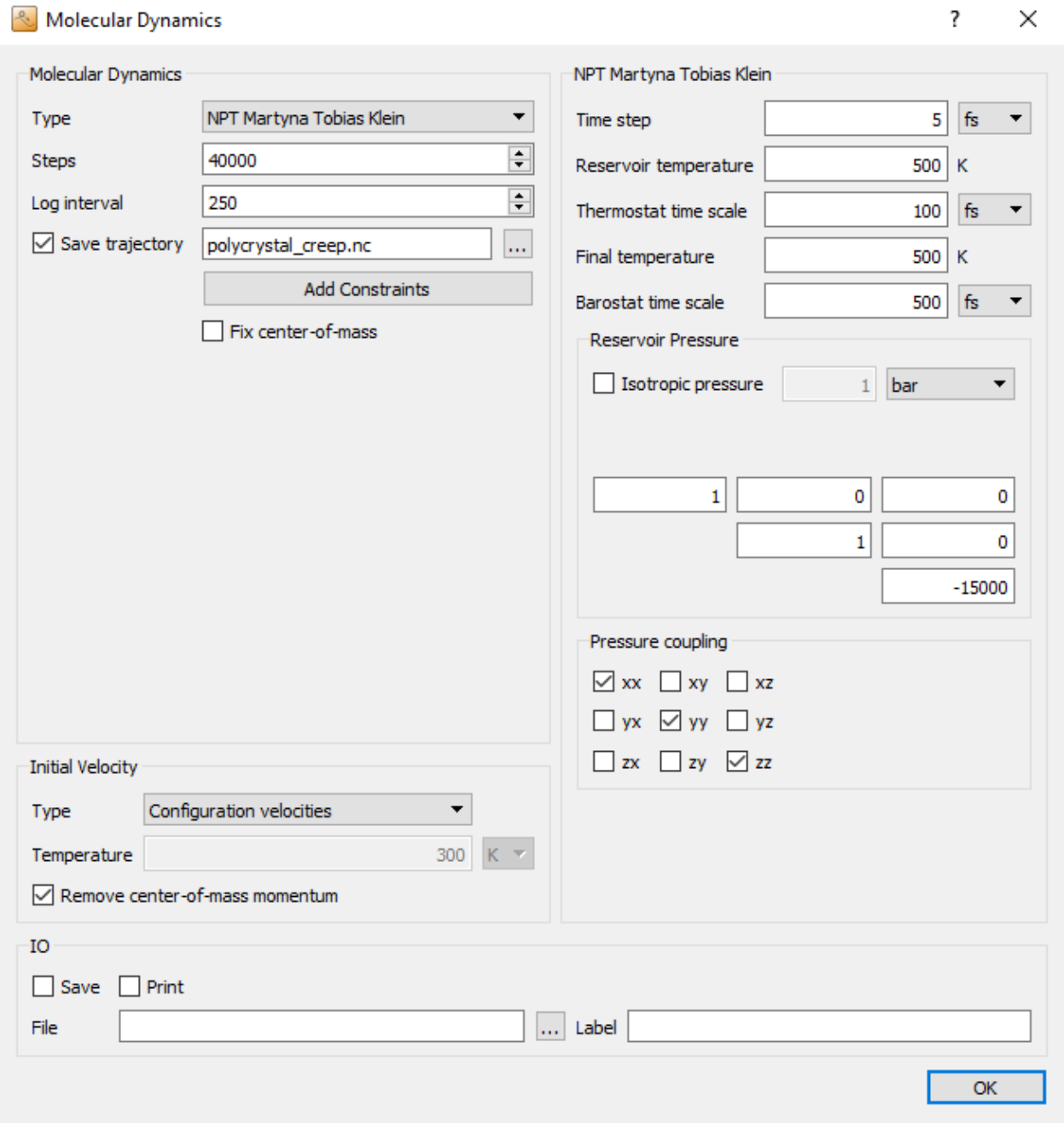
The essential point of a creep simulation is exposing the system to a large uniaxial stress σ_{creep} and monitoring the

strain response, as illustrated below:



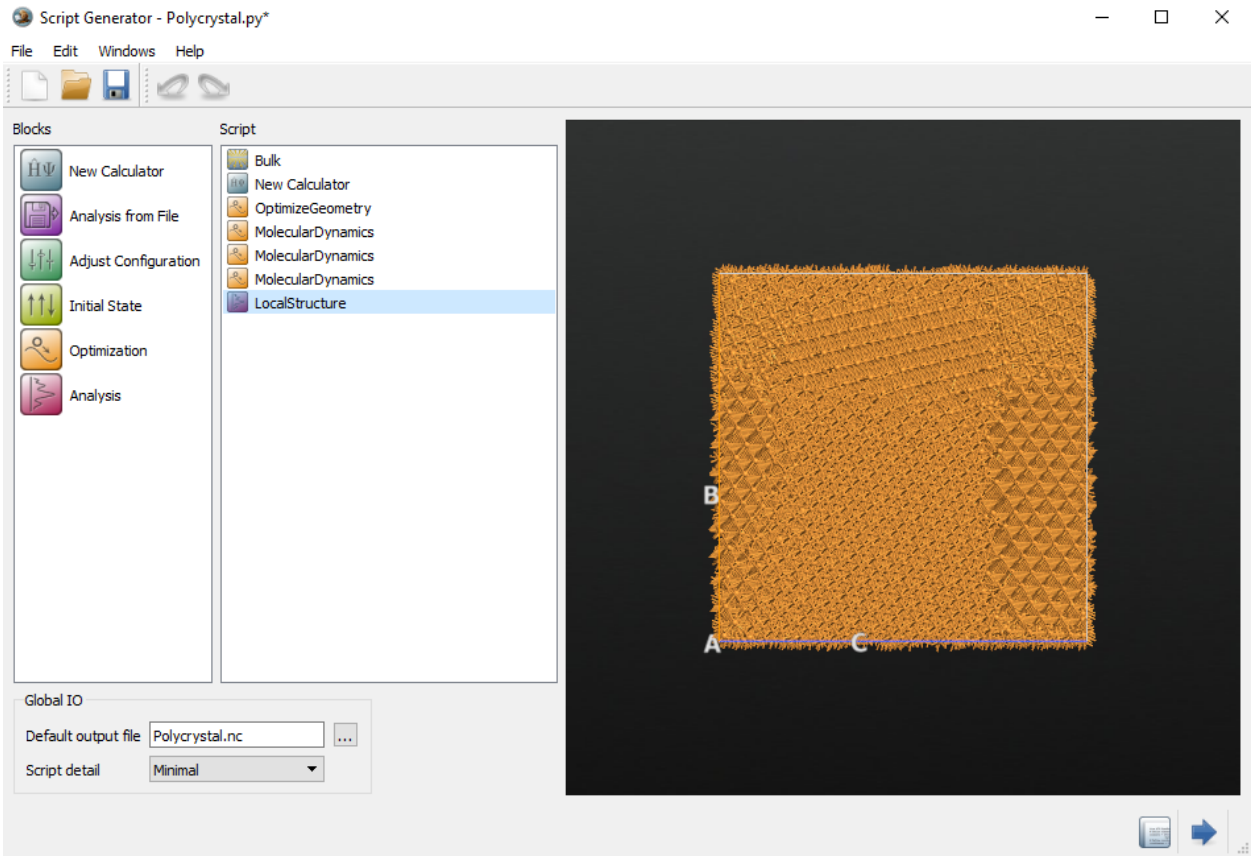
To this aim, you first need to uncheck the **Isotropic pressure** box, to achieve an independent pressure-coupling of the cell vectors. Then, you need to modfiy the Reservoir pressure tensor. The first two diagonal entries (xx and yy) define the ambient pressure perpendicular to the creep direction to allow for lateral relaxation, according the the poisson ratio of the material. For the actual creep stress, which is specified as the third diagonal entry (zz), we set a value of -15,000 bar, corresponding to -1.5 GPa. Note, that negative pressure values are interpreted as tensile external stress.

The off-diagonal entries denote the shear stresses (yz, zx, xy), which do not play a role here as we have de-ativated the coupling to these contributions.



Finally, add another **LocalStructure** analysis block in order to be able to analyze and visualize the final grain structure. Open the block and change the output filename to `local_structure_final.hdf5`.

The **Script Generator** should now look like below. Send the script to the **Editor** by using the botton.



Strain vs. time

A creep simulation is commonly analyzed in terms of the strain as a function of time, to measure the response to the imposed stress. To perform a similar analysis for our model simulation you have to add some more lines to bottom of the script you just produced.

These extra QuantumATK Python script lines are shown below and can be downloaded as a separate script: `analysis.py`. The relevant data (i.e. cell vectors, stress, potential energy) are extracted from the **md_trajectory** object by using the corresponding class methods. Plots of strain, stress and potential energy are then plotted using the pylab module, and saved as PNG files. This analysis can also be carried out separately, after loading a saved HDF5 trajectory using the **nlread()** function.

In the **Editor**, copy the analysis script lines into the bottom on the simulation script.

```
# ------------------------------------------------------------
# Custom Analysis
# ------------------------------------------------------------
md_trajectory = nlread('polycrystal_creep.nc')[-1]
# Extract the times, cell vectors, potential energies, and pressure tensors
times        = md_trajectory.times().inUnitsOf(picoSecond)
cell_vectors = md_trajectory.cells()
e_pot        = md_trajectory.potentialEnergies().inUnitsOf(eV)
pressure       = md_trajectory.pressureTensors().inUnitsOf(GPa)
# Convert the tensor elements into lists
strain_xx = cell_vectors[:, 0, 0]/cell_vectors[0, 0, 0] - 1.0
```

(continues on next page)

(continued from previous page)

```
strain_zz = cell_vectors[:, 2, 2]/cell_vectors[0, 2, 2] - 1.0
pressure_zz = pressure[:, 2, 2]
# Plot the data usig pylab
import pylab
# Plot the strain
pylab.figure()
pylab.plot(times,strain_zz,'b')
pylab.plot(times,strain_xx,'g')
pylab.title('Creep strain in Copper Polycrystal at 500K and a load of 1.5 GPa')
pylab.xlabel('MD time (ps)')
pylab.ylabel('Strain')
pylab.grid(True)
pylab.savefig("CreepStrain.png")
# Plot the pressure
pylab.figure()
pylab.plot(times,pressure_zz,'b')
pylab.title('Stress in Copper Polycrystal')
pylab.xlabel('MD time (ps)')
pylab.ylabel('Stress (GPa,bar)')
pylab.grid(True)
pylab.savefig("CreepStress.png")
# Plot the potential energy
pylab.figure()
pylab.plot(times,e_pot)
pylab.title('Potential energy during creep in copper polycrystal')
pylab.xlabel('MD time (ps)')
pylab.ylabel('Potential energy (eV)')
pylab.grid(True)
pylab.savefig("CreepEPot.png" )
```

Running the simulation

After editing the simulation script as desribed above, use the button to transfer the script to the **Job Manager**, then save the script as `run_polycrystal.py` and run it. The whole simulation takes approximately an hour, depending on your computing infrastructure, since the system is quite large (and yet small compared to the geometries studied in[Page 1241, 1]).

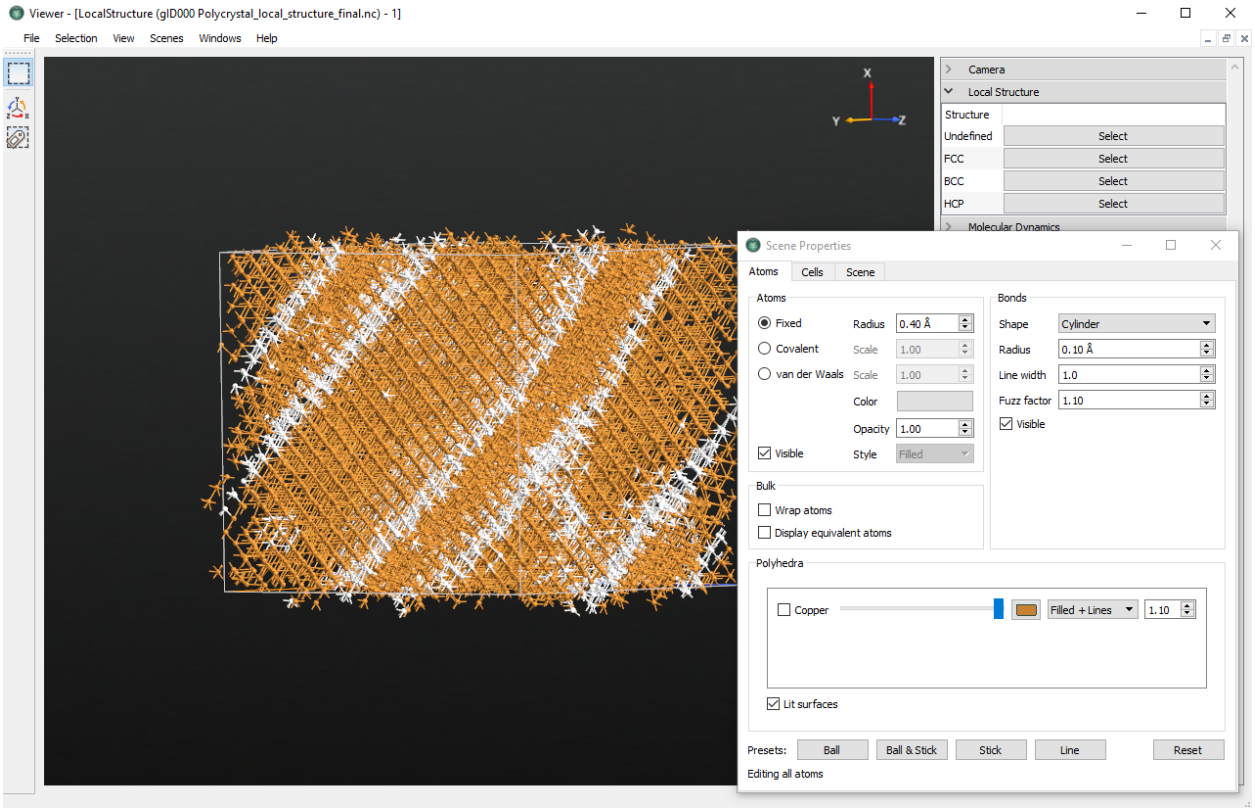
Analyzing the results

After the simulation has finished, let us at first take a look at the trajectory. Select the trajectory named `polycrystal_creep` on the **LabFloor** and open it using the **Movie Tool**. Start the movie by clicking the *Play* button.

You will immediately see an elongation of the simulation cell in the z-direction, accompanied by a slight compression in the perpendicular directions. The elongation proceeds as the simulation continues, but slows down. By rotating the cell with the right mouse button, you can identify the lattice planes of the grains – this allows you to get a visual impression of the grain dynamics.

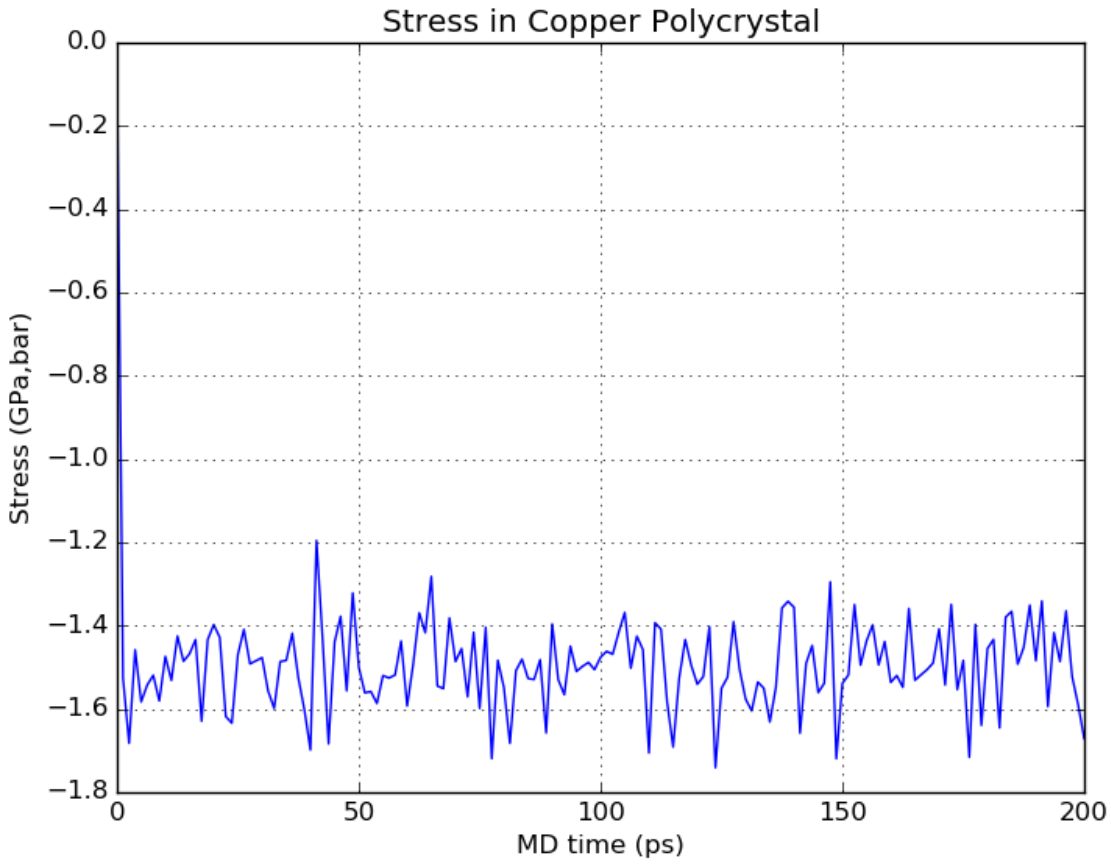
The final grain structure can be be analyzed in a more quantitative manner by using the **Viewer** to visualize the **LocalStructure** analysis item saved in the `local_structure_final.hdf5` file. Edit the coloring scheme just like

you did in the beginning of the tutorial. Probably (although it may depend on the initial structure) the major part of the cell has transformed into a planar grain, which extends across the periodic boundaries.



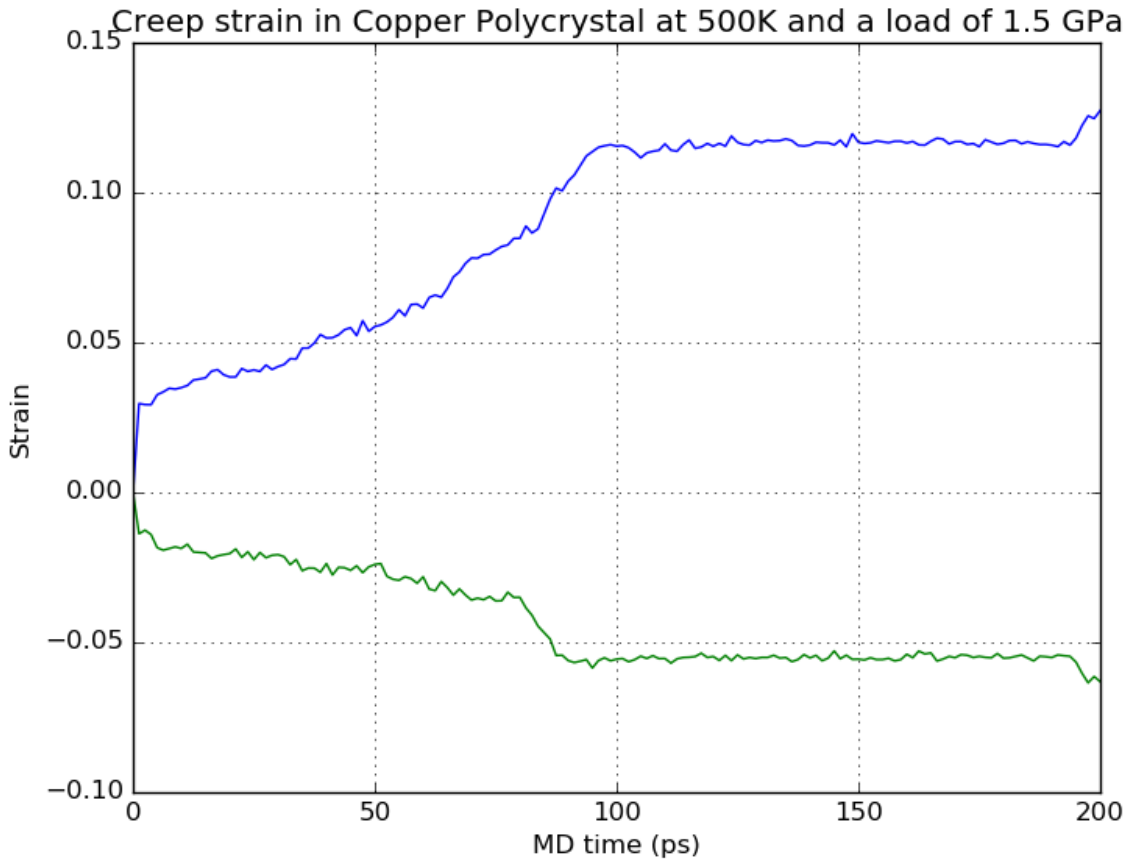
By rotating the cell, you will still see some grey regions, which are the remainders of the grain boundaries. The fast sliding and merging of the grains must be attributed to the small initial grain size, as this reduces the energy barrier upon changing their structure. The final structure appears locked by the periodic boundary conditions of the small cell, and little additional sliding to release energy is possible.

After the structural inspection you may want to have a look at the output observables, starting with the stress in the z-direction. The stress curve has been saved to the file `CreepStress.png`:



The graph shows that the stress oscillates at high frequency and stabilizes around the target value of -1.5 GPa. Generally, as long as the oscillations remain within reasonable bounds and do not increase in magnitude you dont have to worry about this behavior.

The time-dependent creep strain is plotted in the file `CreepStrain.png`. The curves denote the strain in the x (green) and z (blue) directions, respectively:

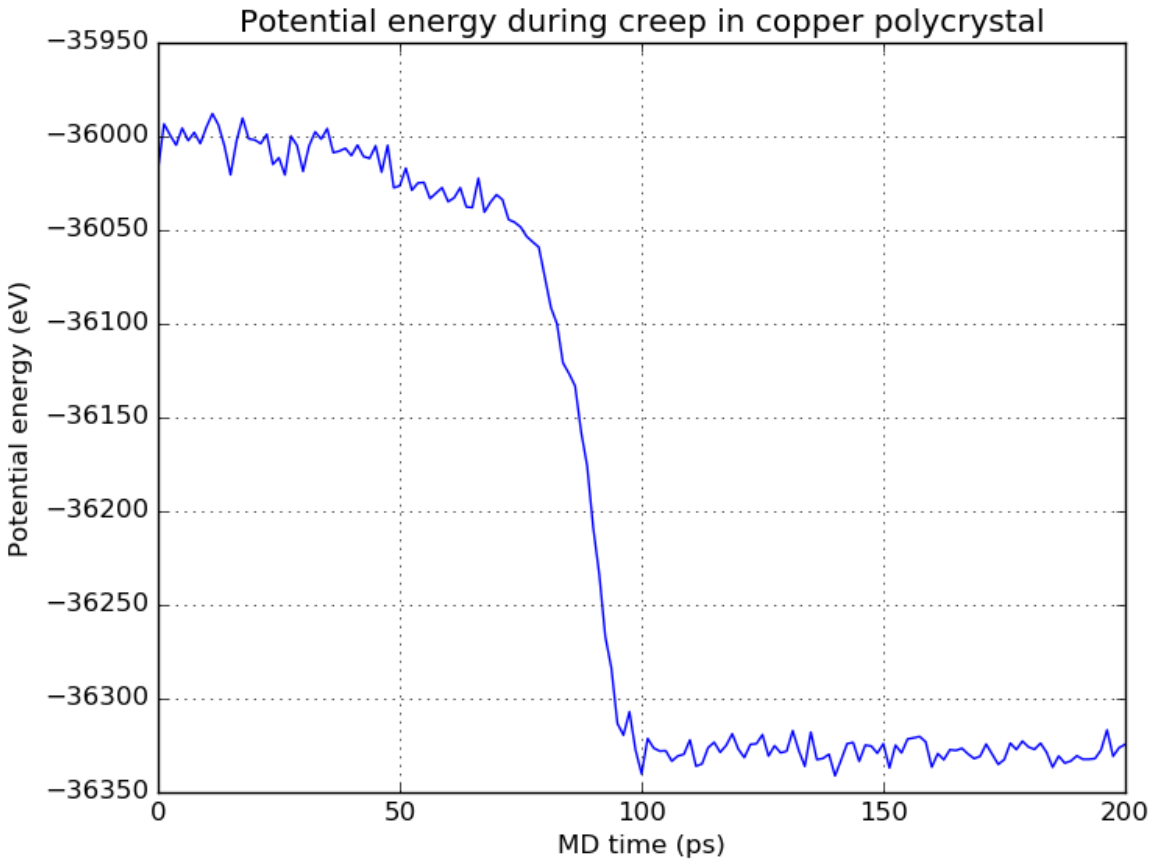


Although your example may look different from the results shown here (due to the random nature of the polycrystalline supercell building process) it should have some common features. In the beginning of the simulation a rapid increase of z-axis strain to about 2% is visible. This contribution arises from the immediate elastic deformation of the solid and it should be to a large extent reversible. Following that, a continuous – but much slower – increase in strain takes place. Here, the grain structure relaxes irreversibly by sliding and diffusion, or even structural transformation. Note that at around 90 ps a jump in the creep curve becomes visible, which must be attributed to a major (inelastic) structural rearrangement.

Important: Be aware that we are considering only the initial onset of creep behavior in a very small model system with very few grains and grain sizes in the order of less than one nanometer! The observed behavior might thus not be transferable in a straightforward way to real polycrystals with considerably larger grain sizes.

The elongation is accompanied by a transverse shrinking of the simulation box, as visualized by the x-strain, from which one can calculate the Poisson ratio of the polycrystal.

Finally, take a look at the time-dependent potential energy:



After the initial elastic deformation, which is of course accompanied by an increase of energy (it can be hard to see in the plot, it is very sudden), the creep process leads to a continuous decrease of potential energy caused by relaxation processes within the grain structure. As already observed in the creep strain, the same pronounced jump at 90 ps appears in the potential energy curve as well. From its magnitude, you can see that the corresponding relaxation process is accompanied by a large release of energy.

Outlook

Using the **Polycrystal Builder**, you can easily build huge polycrystal systems with a large number of grains. The following example shows a copper polycrystal of 300 x 300 x 300 $Å^3$. The system contains around 2.3 million atoms.

Simulations of such huge systems are perfectly possible with the **ATK- Classical** simulation engine, but will naturally take longer time, and one has to be careful with the size of the output files. The picture also nicely demonstrates the great graphical performance of QuantumATK – rotating and manipulating this crystal in 3D can be done completely fluently.

References

3.8.10 Metadynamics Simulation of Cu Vacancy Diffusion on Cu(111) - Using PLUMED

Version: W-2024.09

In this tutorial you will learn how to setup and run metadynamics simulations in **QuantumATK** using the `PlumedMetadynamics` class and the PLUMED plugin. It will cover setting up a simulation to explore the free-energy landscape associated with the diffusion of a Cu vacancy on Cu(111), how to set up a metadynamics calculation, and how to analyze the results.

Introduction

Metadynamics[1] is a powerful method based on molecular dynamics to explore multidimensional free energy surfaces (FESs). It allows one to reconstruct the FES of a system as a function of a number of collective variables (CVs) which are based on the microscopic coordinates of the system.

In metadynamics, an additional bias potential is added to the system at regular intervals during the simulation time. This allows the system to escape deep free-energy minima and overcome high-energy barriers, thereby exploring efficiently the entire free-energy surface. Within materials science, the method has been used to investigate crystal polymorphism, solid-liquid interfaces, and chemical reactions in solids and surfaces. Here, you will apply it to investigate the diffusion of a Cu vacancy on Cu(111).

QuantumATK implements metadynamics via the PLUMED plugin. In this tutorial, we will introduce the method briefly, explain how to setup a metadynamics script for the vacancy diffusion on Cu(111), and how to analyze the resulting free-energy surface.

For more information on the metadynamics method, you may look at Refs.[2] and[3].

Theoretical Background

In metadynamics, an external bias potential is constructed in the phase space of a few selected degrees of freedom, S, generally called collective variables (CVs)[Page 1257, 1]. S is a set of d functions of the microscopic coordinates R of the system.

$$S(R) = (S_1(R), ..., S_d(R)).$$

This potential is built as a sum of Gaussians deposited along the trajectory in the CVs space at time t. In its simplest version, the potential takes the form[2]

$$V(S,t) = \int_0^t dt' \frac{W}{\tau} \exp\left(-\sum_{i=1}^{d} \frac{(S_i(R) - S_i(R(t')))^2}{2\sigma_i^2} \right),$$

where τ is the Gaussian deposition stride, σ_i is the width of the Gaussian for the i-th CV, and W is the height of the Gaussian. The effect of the bias potential is to push the system away from local minima and allow it to visit new regions of the phase space.

[1] Alessandro Laio and Michele Parrinello. Escaping free-energy minima. *Proc. Natl. Acad. Sci.*, 99(20):12562–12566, 2002.

[2] Barducci, Alessandro and Bonomi, Massimiliano and Parrinello, Michele. Metadynamics. *WIREs: Comput. Mol. Sci.*, 1(5):826–843, 2011. doi:10.1002/wcms.31.

[3] KristofăM. Bal and ErikăC. Neyts. Merging Metadynamics into Hyperdynamics: Accelerated Molecular Simulations Reaching Time Scales from Microseconds to Seconds. *Journal of Chemical Theory and Computation*, 11(10):4545–4554, 2015.

Metadynamics Simulation of Cu Vacancy on Cu(111)

Creating a Vacancy on Cu(111)

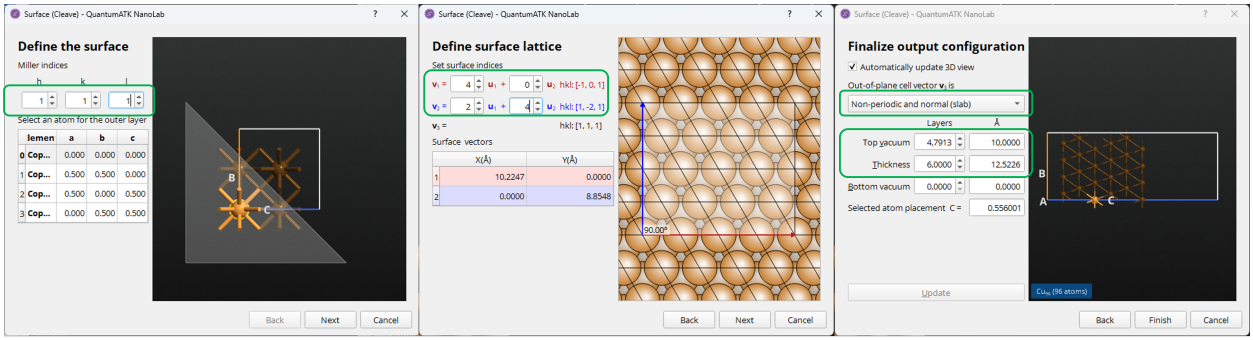
To create the Cu vacancy on Cu(111), you first need to make a Cu(111) surface.

In the **Builder**, click on the in the **Stash** and select *From Database*. In the database dialog, ensure you are on the **Crystals** tab and search for *Copper*. Add Copper to the Builder stash by double-clicking on the row. Next, find the **Surface (Cleave)** plugin in the plugins panel, either by using the filter or manually finding it under the **Builders** category. To create the Cu(111) surface, open the plugin and use the following settings:

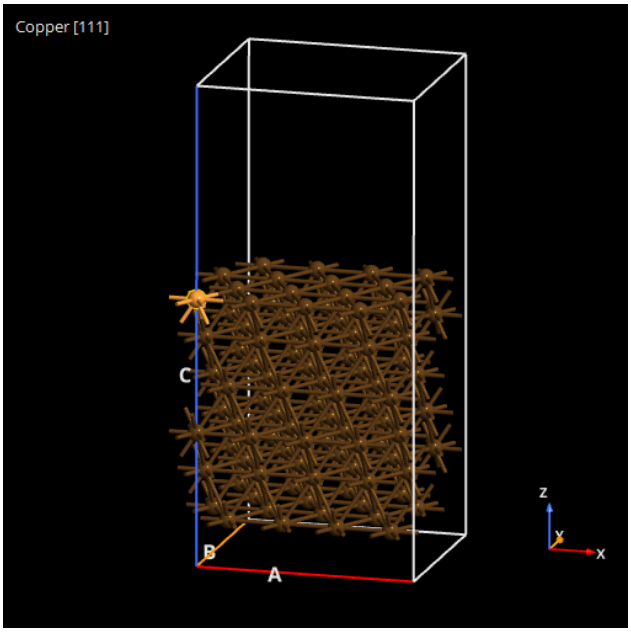
a. Set the Miller indices to $\langle 111 \rangle$.

b. Set the surface indices to $v_1 = (4, 0)$ and $v_2 = (2, 4)$.

c. Set the out-of-plane cell vector as *Non-periodic and normal (slab)*, the thickness to six layers and the (top) vacuum gap to 10 Å.

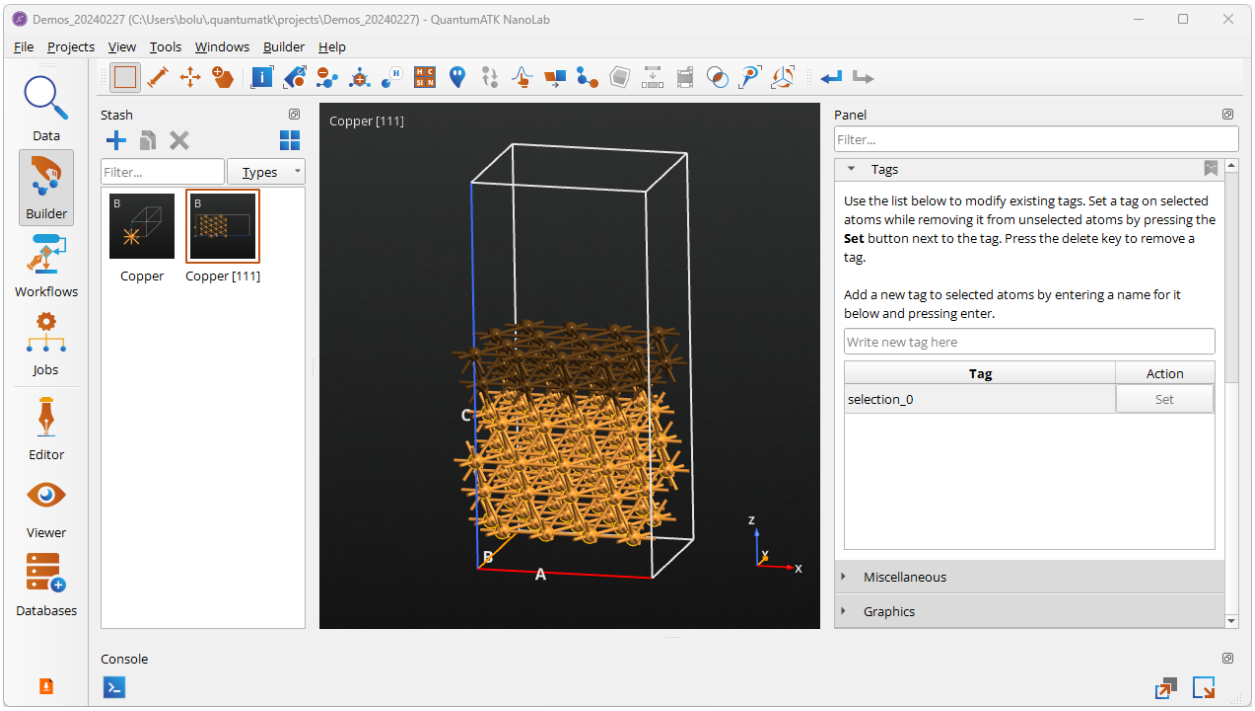


Next, create a vacancy by deleting the atom of the topmost layer with position $(x, y) = (0, 0)$ (shown highlighted in the figure below):



Finally, find the **Tags** plugin (**Selection Tools** category) and select the 4 lower-most Cu(111) layers. Add a new tag by entering a name, selection_0 or another name of your choosing, and hitting Enter. A successfully added tag will show

up in the list in the plugin and also be displayed in the tool-tip of any tagged atom. This tag will be used to fix them during the metadynamics simulation.



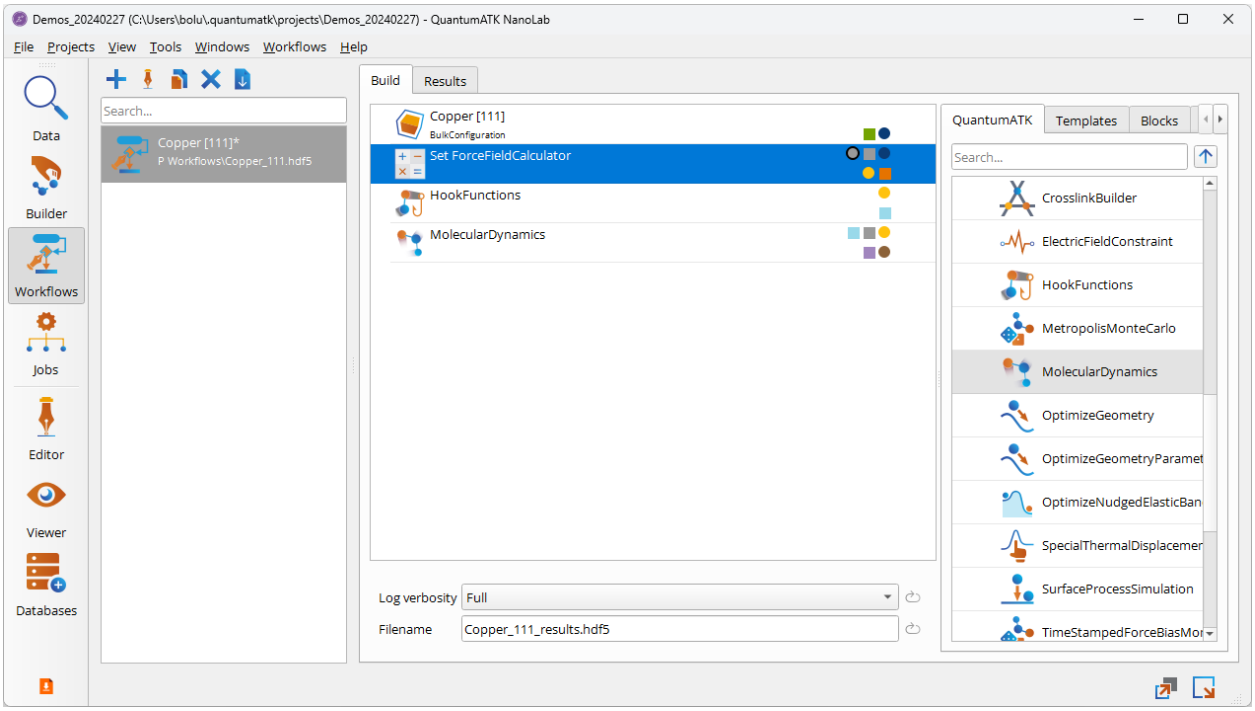
Now you are ready to create the workflow to perform the metadynamics simulation. Use the Send-to menu to send the structure to **Workflows**.

Creating the Metadynamics Workflow

Sending the configuration automatically creates a new workflow with the name of the configuration, and the configuration itself added in a Configurations block. To set up the remainder of the workflow, add the following blocks:

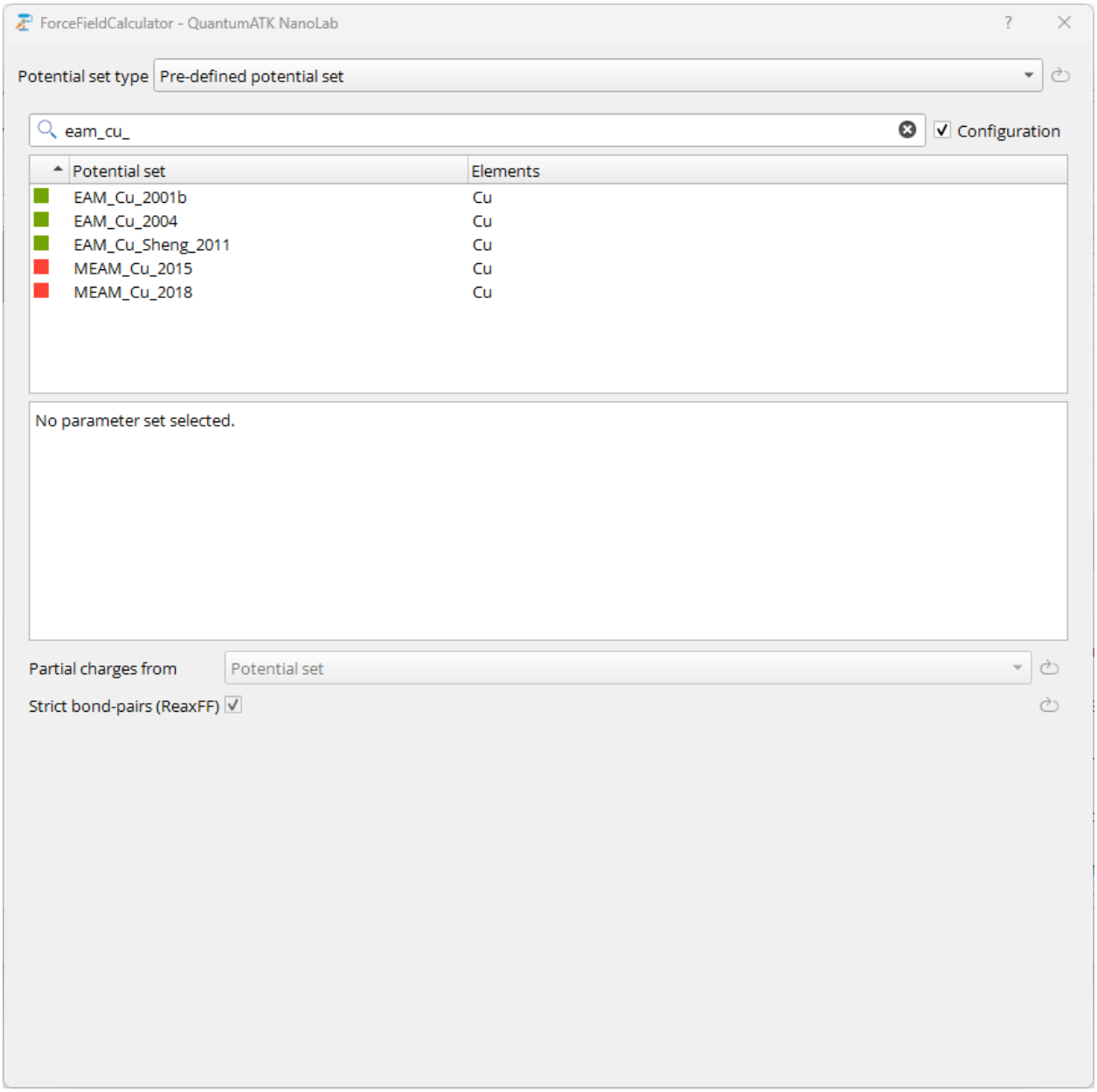
- *Calculators → Set ForceFieldCalculator*
- *Optimization and Dynamics → HookFunctions*
- *Optimization and Dynamics → MolecularDynamics*

Your workflow should now look like this:



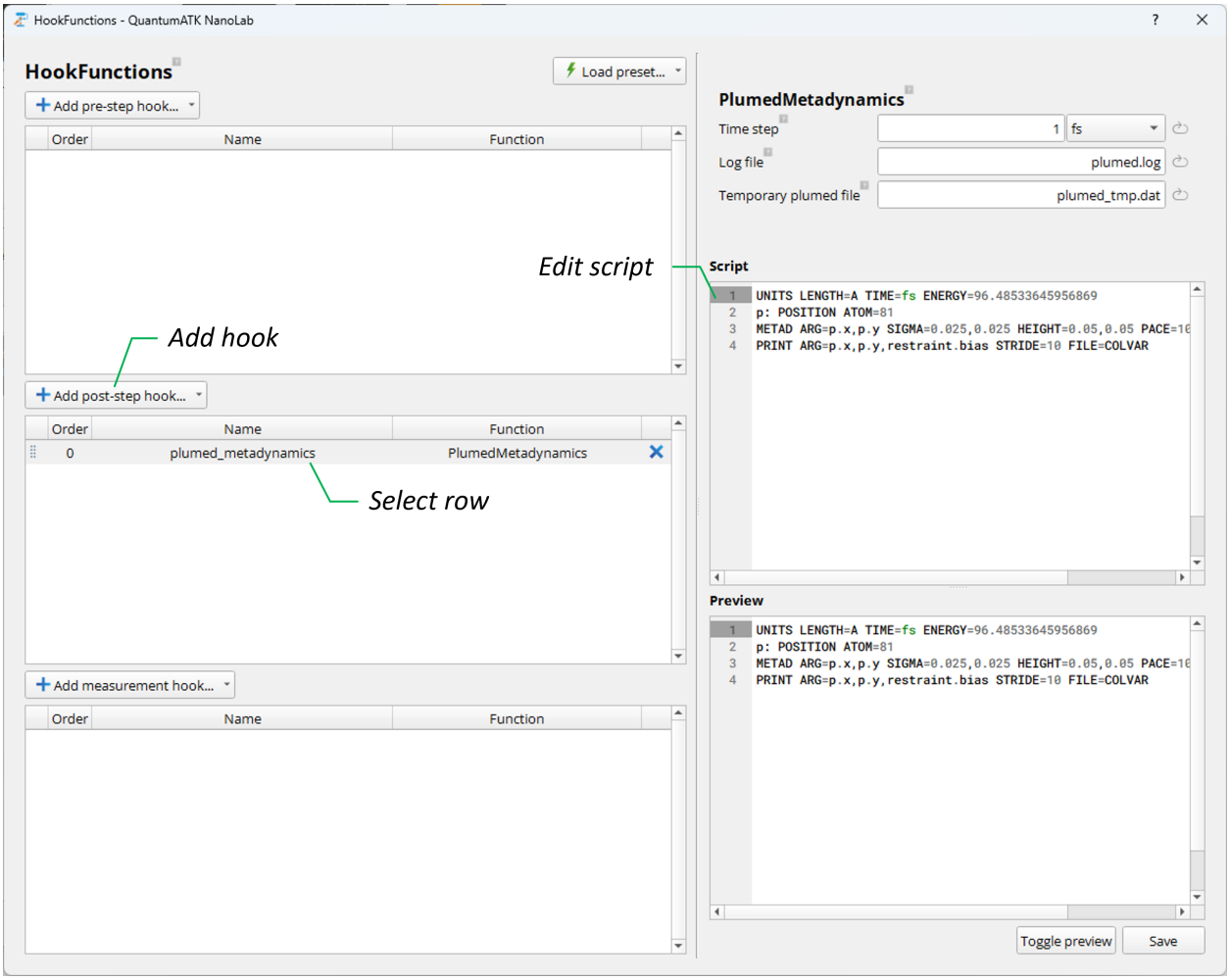
Next, double-click on each of the added blocks to edit them:

1. In the **Set ForceFieldCalculator** block, ensure that the **Potential set type** is set to Pre-defined potential set. Find and select EAM_Cu_2001b in the list of potential sets. To make it easier to find, use the search field to narrow down the list.



2. In the **HookFunctions** block, add a **PlumedMetadynamics** post-step hook. Click on the added row in the table to edit the hook settings. In the **Script** textbox, one can add any PLUMED[2] command. In this example we would like to add the following command:

```
UNITS LENGTH=A TIME=fs ENERGY=96.48533645956869
p: POSITION ATOM=81
METAD ARG=p.x,p.y SIGMA=0.025,0.025 HEIGHT=0.05,0.05 PACE=1000 LABEL=restraint
PRINT ARG=p.x,p.y,restraint.bias STRIDE=10 FILE=COLVAR
```



There are four command lines:

•`UNITS`: this line controls the units that will be used by PLUMED. Edit the lines as above, and the units of *Å*, *femtoseconds*, and *eV* will be used.

•p: This line is used to define the atoms, whose cartesian position should be used as collective variable. In this case, the atom with index 81.

Note: This value should be set according to the indexing used by PLUMED, where atom indexing starts from 1, and not from 0 as in **QuantumATK**. This value should therefore be set to *atom index* $+1$, where *atom index* is the index of the atom, which should be used in the collective variable.

We will look at the potential energy surface in the x- and y- axis using the Gaussian function of the *width* (0.025) and *height* (0.05) on each axis.

•`METAD`: This line controls the type of metadynamics and the shape of Gaussians functions that will be used during the simulations:

–`ARG`: this flag is used to set the collective variables. In this case, we will use as collective variables the position of the atom p along the Cartesian direction x $(p.x)$ and y $(p.y)$.

–`SIGMA`: this flag is used to set the value of the width of the Gaussian functions σ used for each collective variable. In this case, we will use $\sigma = 0.025$ for both collective variables.

–`HEIGHT`: this flag is used to set the height of the Gaussian functions used for each collective variable. In this case, we will use a height of 0.05 eV for both collective variables.

–`PACE`: this flag is used to set the intervals at which a new Gaussian function will be added. In this case, we will add a new Gaussian function every 1000 MD steps.

–`LABEL`: this flag is used to set the name of the collective variables.

•`PRINT`: This line controls the format of the PLUMED output.

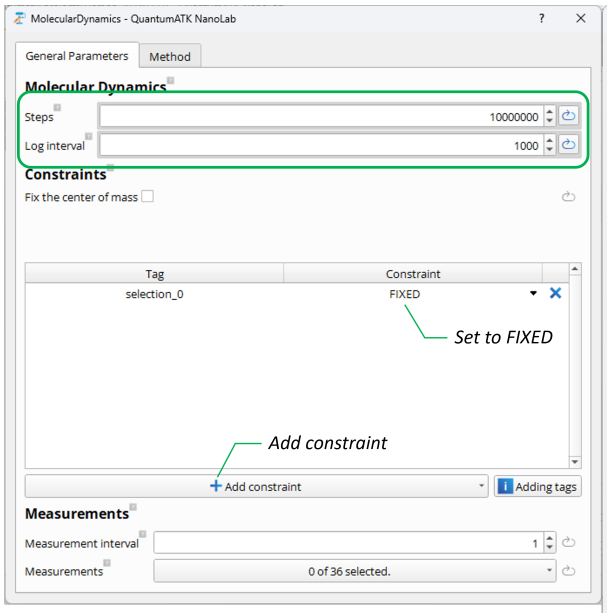
–`ARG`: this flag controls the variables that will be printed, in the present case $p.x$, $p.y$, and the bias.

–`STRIDE`: this indicates the printing interval. In the present case, 10 MD steps.

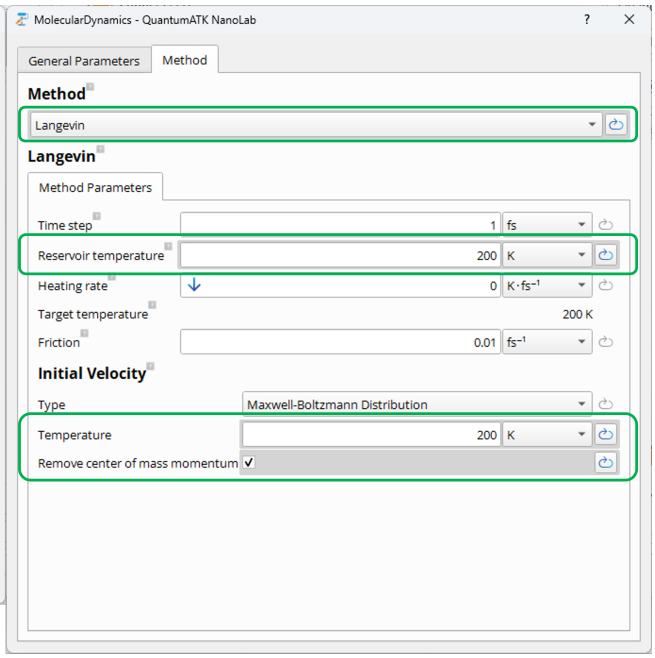
–`FILE`: this line indicates the name of the output file.

If you would like to know more detailed information, we recommend you to look in the PLUMED documentation. These commands are passed to the *`PlumedMetadynamics`* class. The latter is passed to the `MolecularDynamics` class using a **hook function**.

3. Open the **MolecularDynamics** block and set the simulation parameters on the two tabs as shown in the figures below. Also click on **Add constraint** and select for the previously added tag (by default, the tag is selection_0). Set the constraint type to *FIXED*.



a)



b)

For reference, the input files can be downloaded here:

•Worfklow `Cu111_vacancy_hopping.hdf5`

•Script `Cu111_vacancy_hopping.py`

Once you are done, send it to **Jobs** to run the simulation, which should take about 8 hours on 8 CPUs.

Analyzing the Results

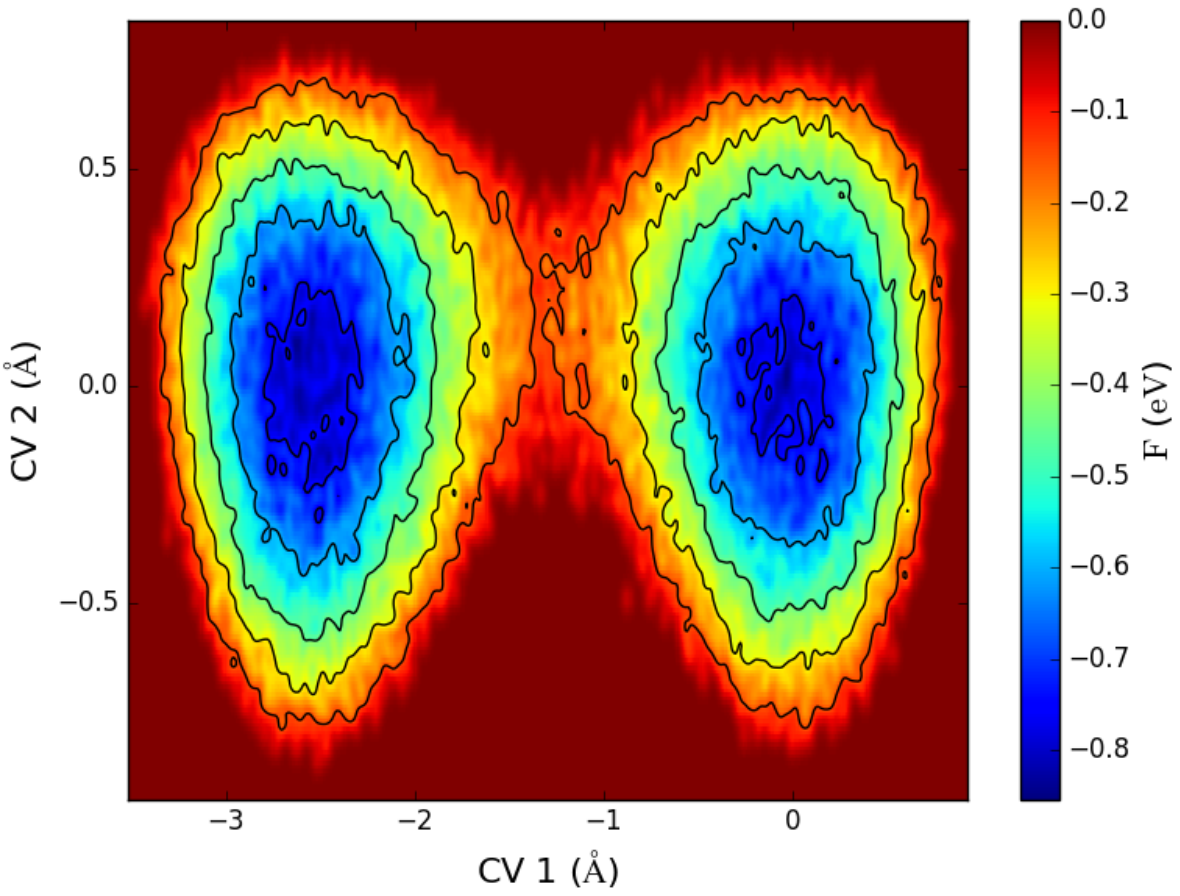
In addition to the standard **QuantumATK** output, at the end of the job you will obtain the files `COLVAR` and `HILLS`. These two files are the output of PLUMED and will be used to analyze the metadynamics simulation.

To plot the profile of the free energy F as a function of the collective variables CV 1 and CV 2, you first have to run the script `extract_F.py` as `atkpython extract_F.py`. Three output files will be produced:

- `F_vs_cv1.dat` contains the data for F vs. CV 1.
- `F_vs_cv2.dat` contains the data for F vs. CV 2.
- `F_vs_cv1_cv2.dat` contains the data for F vs. CV 1 and CV 2.

The map of the free energy profile F can be obtained by running the script `plot_F_vs_cv1_cv2.py` as:

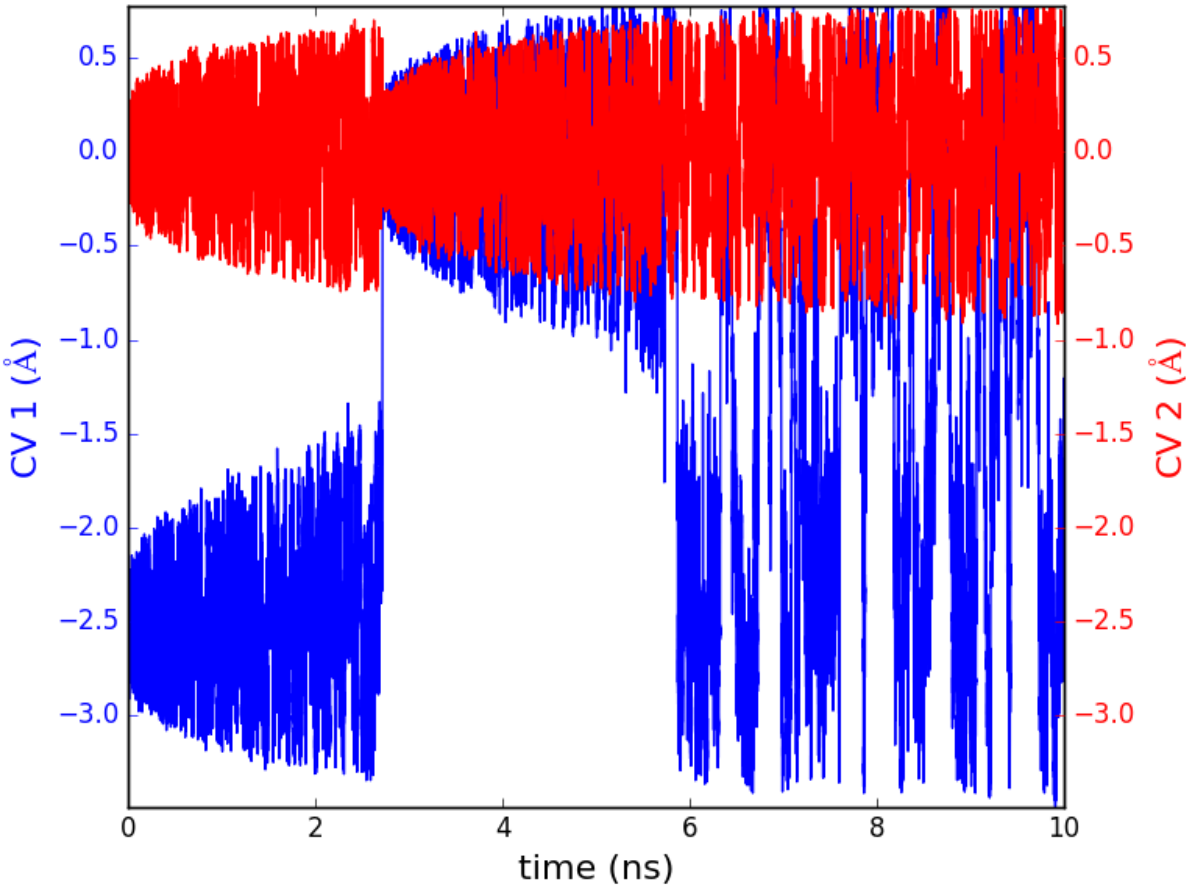
```
atkpython F_vs_cv1_cv2.py
```



The above plot shows a heat map of the free energy profile as a function of the collective variables CV 1 and CV 2. It can be seen that there are two minima on the free-energy surface, corresponding to the diffusion of the vacancy from one site to the neighboring one.

The evolution of the collective variables over the simulation time can be obtained by running the script `cv1_cv2_vs_time.py` as:

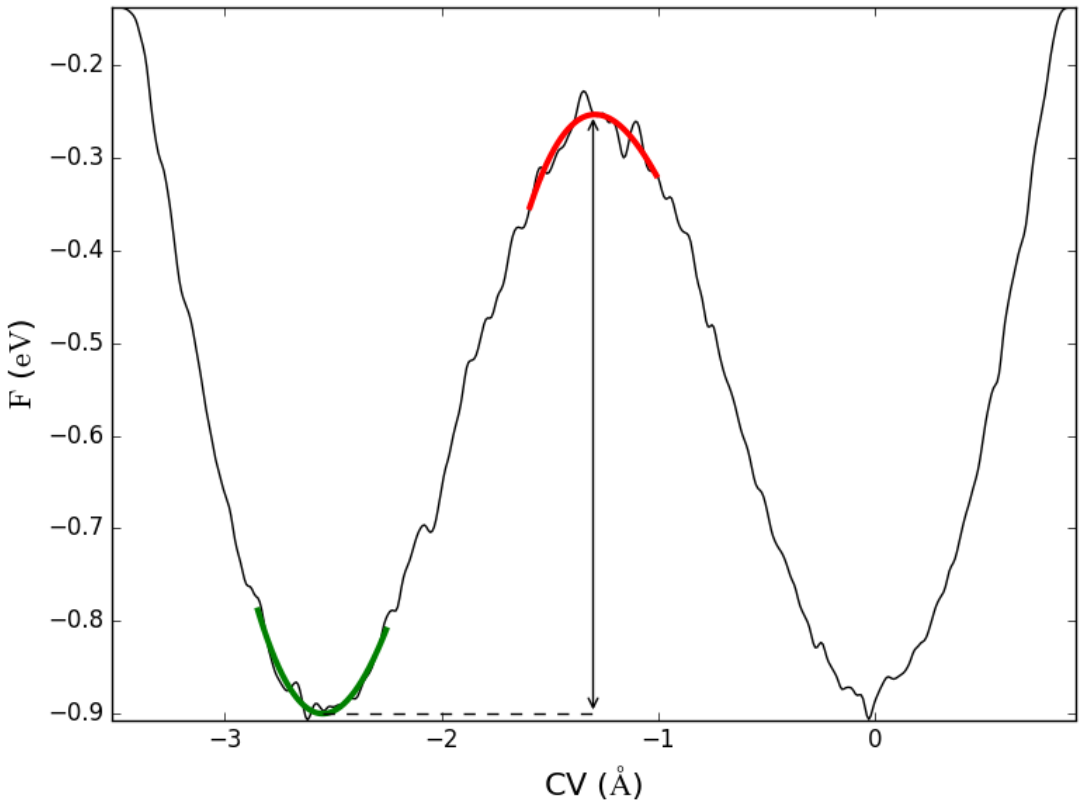
```
atkpython cv1_cv2_vs_time.py
```



The resulting plot shows the evolution of CV 1 (*red*) and CV 2 (*blue*) as a function of the simulation time. It can be seen that CV 2, which corresponds to the y Cartesian coordinate, oscillates around a constant value (CV 2 $= 0$), because both free energy minima occur at the same value of y. Conversely, the evolution of CV 1, which corresponds to the x Cartesian coordinate, indicates that the simulation starts in the minimum on the left (CV 1 $= -2.5$). The first minimum is filled until at approx. 2.2 *ns*, and then the system moves to the second minimum (CV 1$= 0$). After the second minimum is also filled, at around 6 *ns* the system oscillates with equivalent probabilities between the two minima, until the simulation is completed.

The free energy barrier can also be analyzed by running the script `barrier.py` as:

```
atkpython barrier.py
```



The results show that the barrier has a height of 0.647 *eV*, in good agreement with the results in[4].

References

3.8.11 Viscosity in liquids from molecular dynamics simulations

Version: P-2019.03

In this tutorial you will learn how to calculate the viscosity of different liquids using molecular dynamics (MD) simulations using the example of liquid methanol. Understanding viscosity is critical in designing a number of industrial chemical processes, as viscosity gives a description of how the liquid flows. This tutorial demonstrates how you can use the QuantumATK tools to simulate the viscosity of a simple fluid. This methodology can be applied to much more complicated fluids as well as mixtures.

To simulate the behavior of liquid methanol we use classical molecular dynamics to generate trajectories of the motion of molecules. These trajectories give the information necessary to statistically calculate the overall properties of the liquid. This tutorial also demonstrates how to set up and use specific bonded forcefields that have been created for these types of molecular problems.

Contents

•*Viscosity in liquids from molecular dynamics simulations*

–*Theory*

[4] Suresh KondatiăNatarajan and Jorg Behler. Self-Diffusion of Surface Defects at Copper-Water Interfaces. *The Journal of Physical Chemistry C*, 121(8):4368–4383, 2017.

*Molecular viscosity
*Force fields
–Computational procedure
*Building the initial methanol configuration
*Adding the forcefield to the calculation
*Relaxing the starting structure
*Equilibrating the structure
*Setting the production calculation
*Modifying the script
*Running the calculation
–Analyzing the results
*Viewing the change in pressure
*Calculating the viscosity from the Einstein relationship
*Calculating the viscosity from the Green-Kubo relationship
–Extending the results
*Running more independent trajectories
*Improving the accuracy

Theory

Molecular viscosity

Viscosity is a measure of the friction between molecules passing each other in the fluid. There are a number of ways of estimating this friction[1], but one that is commonly used is the Einstein relationship where the viscosity η can be stated as[2]:

$$\eta = \lim_{t\to\infty} \frac{V}{2tk_BT} \left\langle \left(\int_0^t \tau_{\alpha\beta}(t')dt' \right)^2 \right\rangle$$

Here the quantities being integrated are the off-diagonal elements of the pressure tensor. The labels α and β refer to these elements. The other quanities represented here are time t, temperature T, volume V, and the Boltzmann constant k_B.

An alternative method for calculating the viscosity is through the Green-Kubo relation[3]:

$$\eta = \frac{V}{k_BT} \int_0^\infty \langle \tau_{\alpha\beta}(t_0)\tau_{\alpha\beta}(t) \rangle \, dt$$

Here the quantity in the angle brackets is the autocorrelation function of the pressure tensor. As this auto-correlation function decays to zero at long timescales, the integral converges to give the fluid viscosity.

[1] B. Hess: Determining the shear viscosity of model liquids from molecular dynamics simulations. J. Chem. Phys. 116, 209 (2016)

[2] S. H. Jamali, R. Hartkamp, C. Bardas, J. Sohl, T. J. H. Vlugt, O. A. Moultos: Shear viscosity computed from the finite-size effects of self-diffusivity in equilibrium molecular dynamics. J. Chem. Theory. Comput. 14, 5959 (2018)

[3] D. Gonzalez-Salgado, C. Vega: A new intermolecular potential for simulations of methanol: The OPLS/2016 model. J. Chem. Phys. 145, 034508 (2016)

Both of these equations may produce similar results. For a trajectory with a large number of data points, the Green-Kubo expression can become computationally expensive. Calculating the autocorrelation function scales as $O(N^2)$ where N is the number of data points. The Einstein relationship scales as only $O(N)$. As calculating viscosity requires looking at a large number of data points, the focus will be placed on the Einstein relationship.

The pressure tensor itself is made up of contributions both from the motion of the atoms and also from how the atoms interact with the boundary conditions in the simulation. The pressure tensor can be given as:

$$\tau_{\alpha\beta} = \sum_i m_i v_{i,\alpha} v_{i,\beta} - \frac{dE}{d\varepsilon_{\alpha\beta}}$$

Here m_i references the mass of atom i, $v_{i,\alpha}$ references the velocity of the atom i in the cartesian direction α and E is the energy of the system which is being differentiated with respect to the strain $\varepsilon_{\alpha\beta}$

Force fields

Classical force fields work by replacing the computationally expensive calculation of electronic structure with simple functions designed to approximate the overall energy. This approach enables the effective simulation of much larger systems, well beyond what is possible with electronic structure calculations. One of the main challenges in this approach is fitting parameters for the replacement functions so they can accurately predict the energy.

One approach is to divide molecules up into a number of chemical types. Specific parameters can then be found for different types of interactions using these types. As an example, an sp^3 carbon in ethane will have different associated potentials to an sp^2 carbon in a benzene ring. As several combinations will be excluded based on how the types are defined, this limits the number of parameters that need to be found. Because parameters are fitted for specific atomic types and functions, these types of potentials can often be more accurate. As a trade-off these potentials are often more difficult to set up, as the chemical types for each atom need to be assigned for the structure. Potentials of this type also cannot model reactions, as the atom connectivity is needed to determine the chemical type.

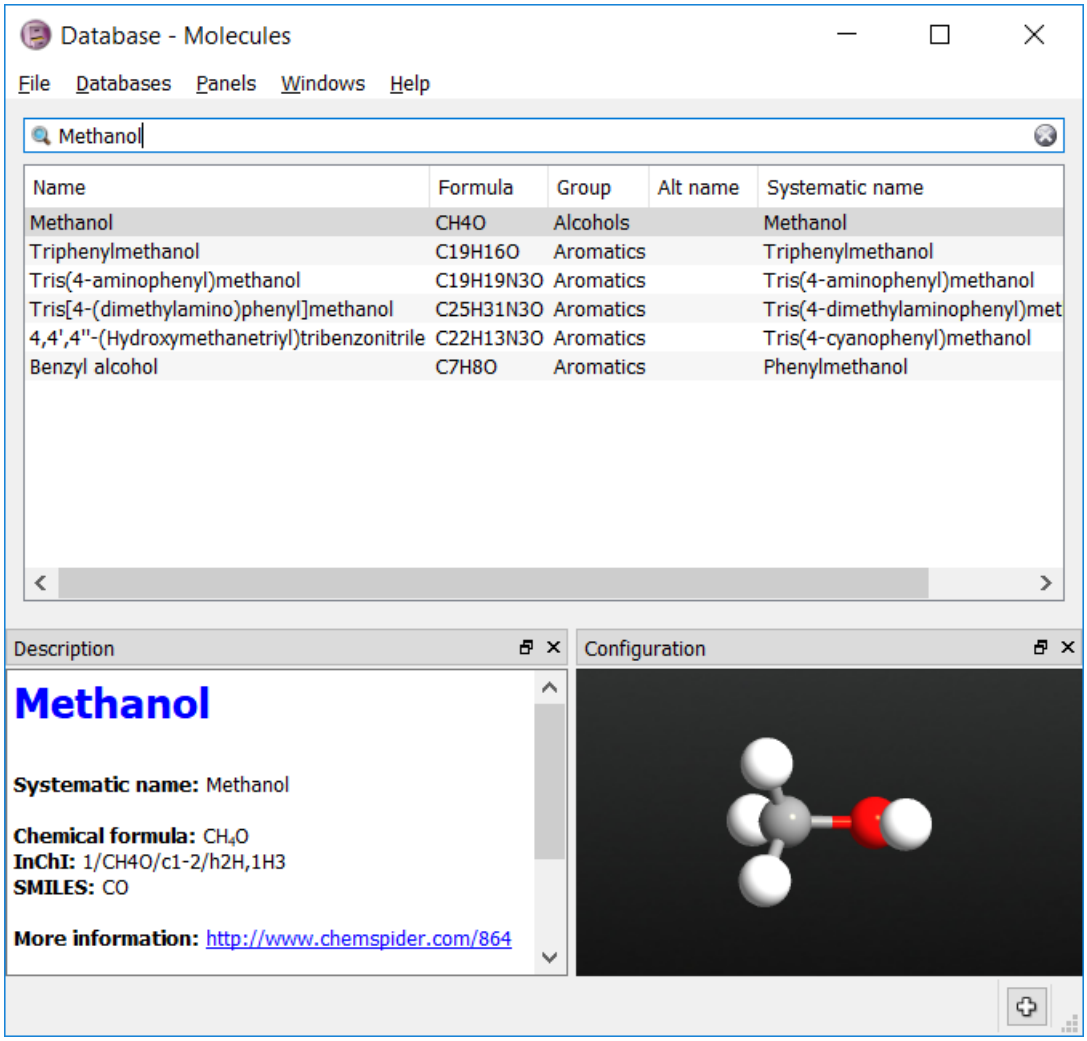
In this tutorial you will use the OPLS-AA forcefield to model methanol[4]. The OPLS-AA forcefield uses atom types with specifically fitted potentials for those types. This tutorial covers how to set up calculations using these types of forcefields.

Computational procedure

Building the initial methanol configuration

Open the **Builder** and select *Add* → *From Database*. Add the molecular structure of methanol from Database to the **Stash**, making sure to choose the molecule database by selecting *Databases* → *Molecules*.

[4] W. L. Jorgensen, D. S. Maxwell, J. Tirado-Rives: Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids. J. Am. Chem. Soc. 118, 11225 (1996) 118,



Now the chemical types for each of the atoms needs to be assigned. OPLS methanol has 4 types:

•CT for the tetrahedral carbon

•HC for the hydrogens attached to the carbon

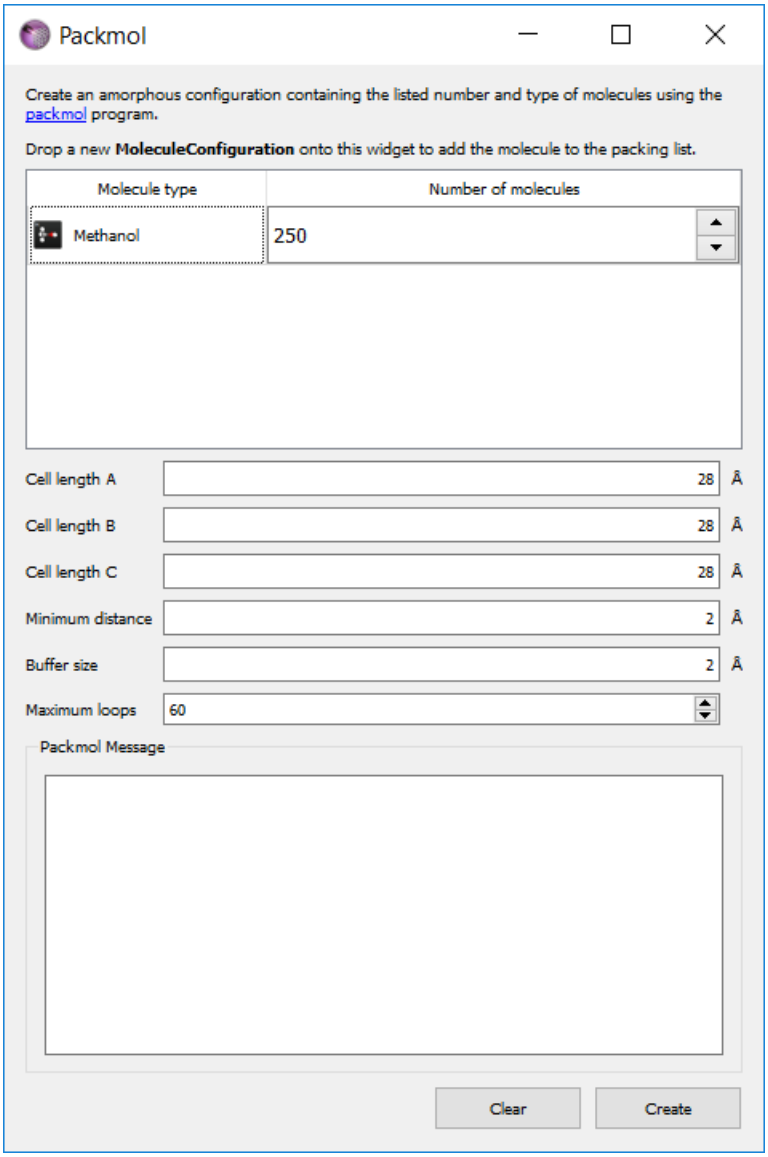
•OH for the alcohol oxygen

•HO for the alcohol hydrogen.

Select the three hydrogen atoms attached to the carbon. Set their types to HC by going to *Selection Tools* → *Tags* and typing in the tag HC. You should now be able to select these atoms by using the tag. Repeat the process for the other three atom types.

Tip: When selecting atoms, holding down the *Ctrl* key adds atoms to the current selection, while holding down the *Shift* key removes them.

Next create a bulk model of liquid methanol by selecting *Builders* → *Packmol*. Drag the molecule of methanol from the stash into the molecule type column, and select 250 molecules. Set each cell length to 28 Ångstroms. Click Create.

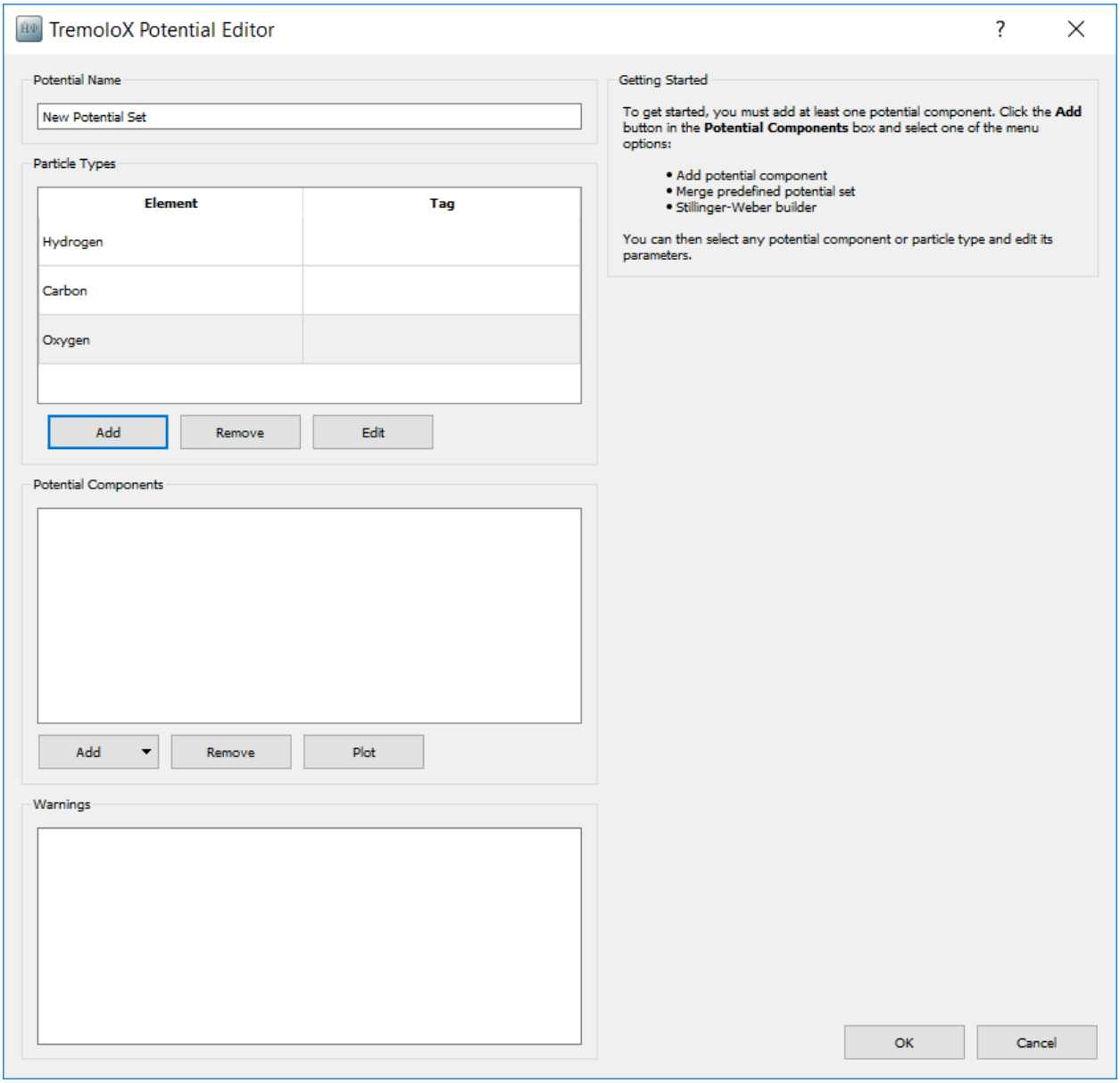


Mouse over the atoms in the new structure to check that the tags reflect the type of each atom.

Adding the forcefield to the calculation

One you are happy with the atom tag assignment, send the bulk methanol structure to the **Scripter** using the **Send To** icon .

In the **Scripter** first add in a **ForceFieldCalculator** block. In this block the forcefield needs to be defined. To do this a new forcefield definition needs to be added to the script. Open the **ForceFieldCalculator** and select *New*. This opens up the **Potential Editor** dialogue to enter in new forcefield terms.



Give the new potential the name `Methanol OPLS-AA`. When opening the **Potential Editor** particle types are added for each atom. These need to be deleted and replaced with particle types for the 4 atom tags. Select each element and press *Remove*. Then press *Add* to add each atom term.

Atom Tags

Pressing *Add* opens the **Particle Type Editor**. In this the type and basic information about the non-bonding parameters for the type are added.

The OPLS-AA forcefield used the expression for the non-bonding energy E_{nb}:

$$E_{nb} = \sum_{ij} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_i j} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}$$

The terms ϵ_{ij} and σ_{ij} are calculed from the Lorentz-Berthelot combination rules:

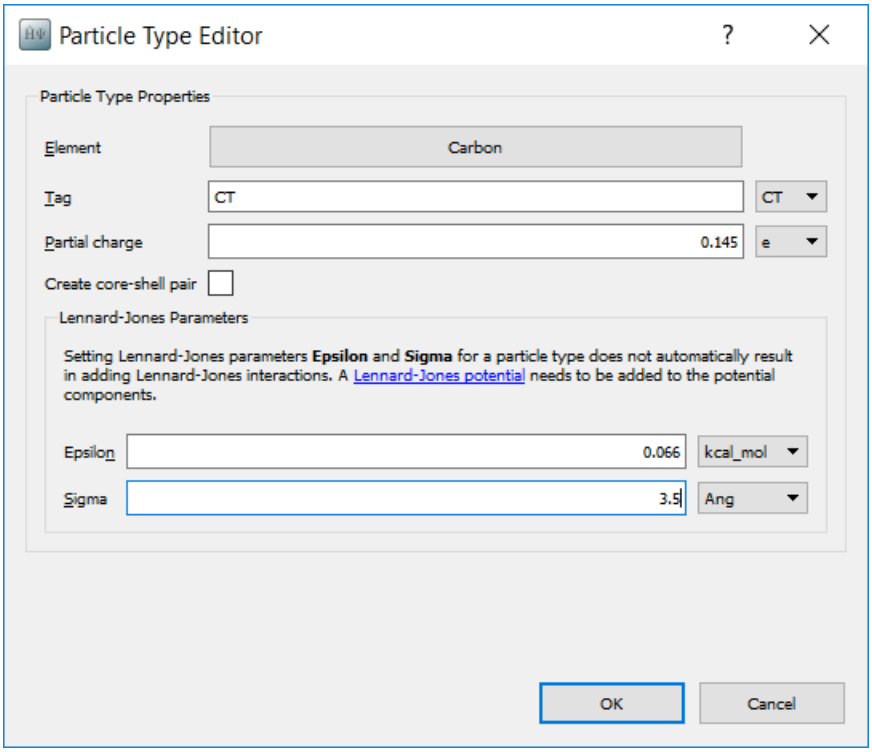
$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j}$$
$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}$$

The nonbonding parameters for the 4 atom types in methanol are:

Type	ϵ_i / kcal mol^{-1}	σ_i / Å	q_i / e
CT	0.066	3.50	0.145
OH	0.170	3.12	-0.683
HC	0.030	2.50	0.04
HO	0.000	0.00	0.418

Note: The HO type is not a van der Waals site in the OPLS-AA potential, and so the parameters for this atom are zero. It is however an electrostatic site, and so carries a charge.

Select the element from the pop-up periodic table, and then select the tag from the dropdown list. Add the corresponding nonbonding parameters. As an example, the input for the CT type is shown below.



Bonding Terms

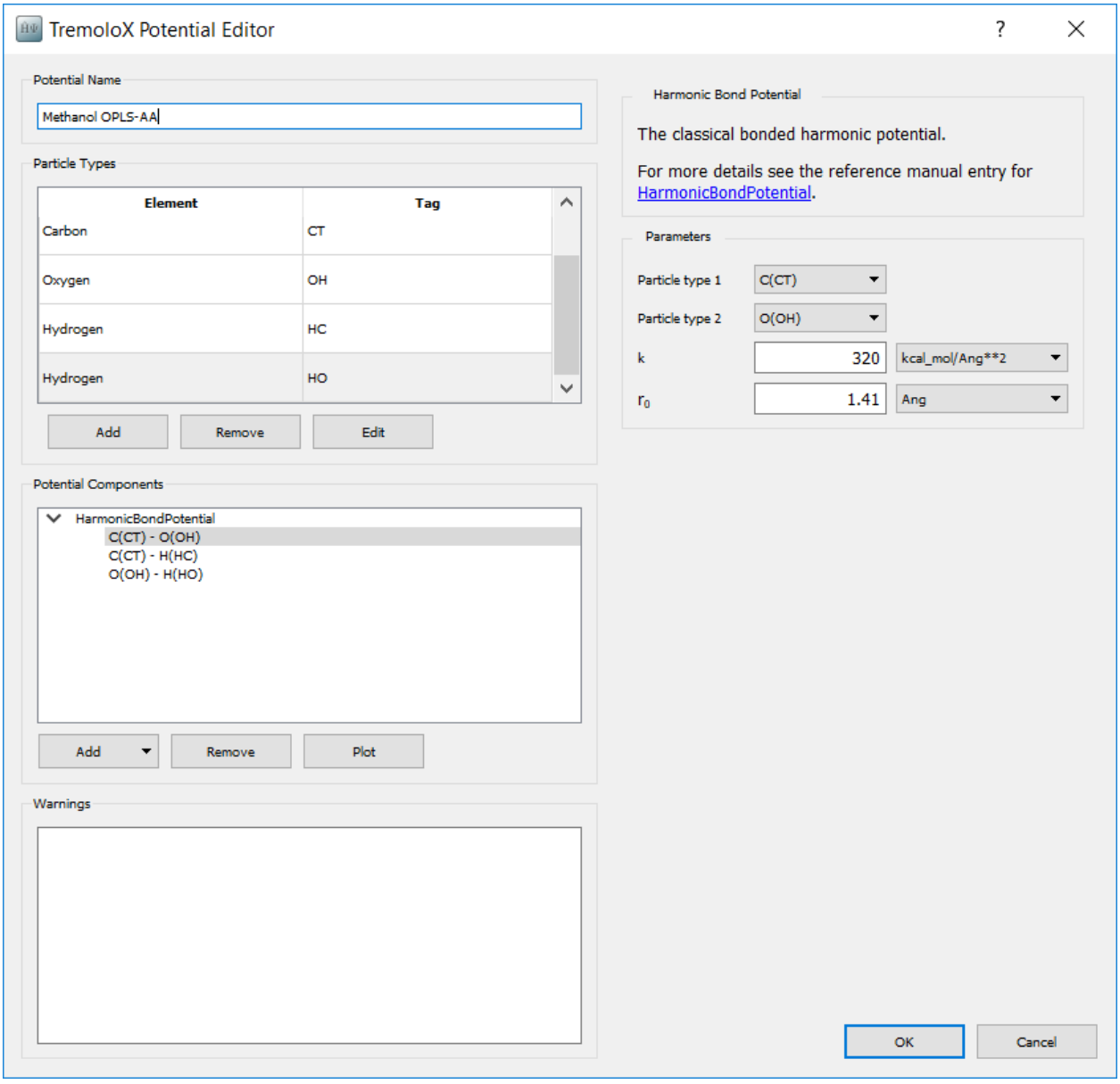
Bond stretching energy E_{bond} is approximated by a simple harmonic spring equation:

$$E_{bond} = \sum_{ij} K_{ij} \left(r_{ij} - r_0\right)^2$$

There are three different types of bonds in methanol. The parameters for these bonds are given in the following table:

Type I	Type J	K_{ij} / kcal mol^{-1} Å^{-2}	r_0 / Å
CT	OH	320.0	1.41
CT	HC	340.0	1.09
OH	HO	553.0	0.945

To add the bonding potentials, in the *Potential Components* box select *Add* → *Add potential component*. From the list of potential types select *HarmonicBondPotential*. Selecting a potential displays the parameters for it one the right hand side. Add three potentials and enter the parameters for each bond type. The input for the `CT - OH` bond type is shown below.



Angle Terms

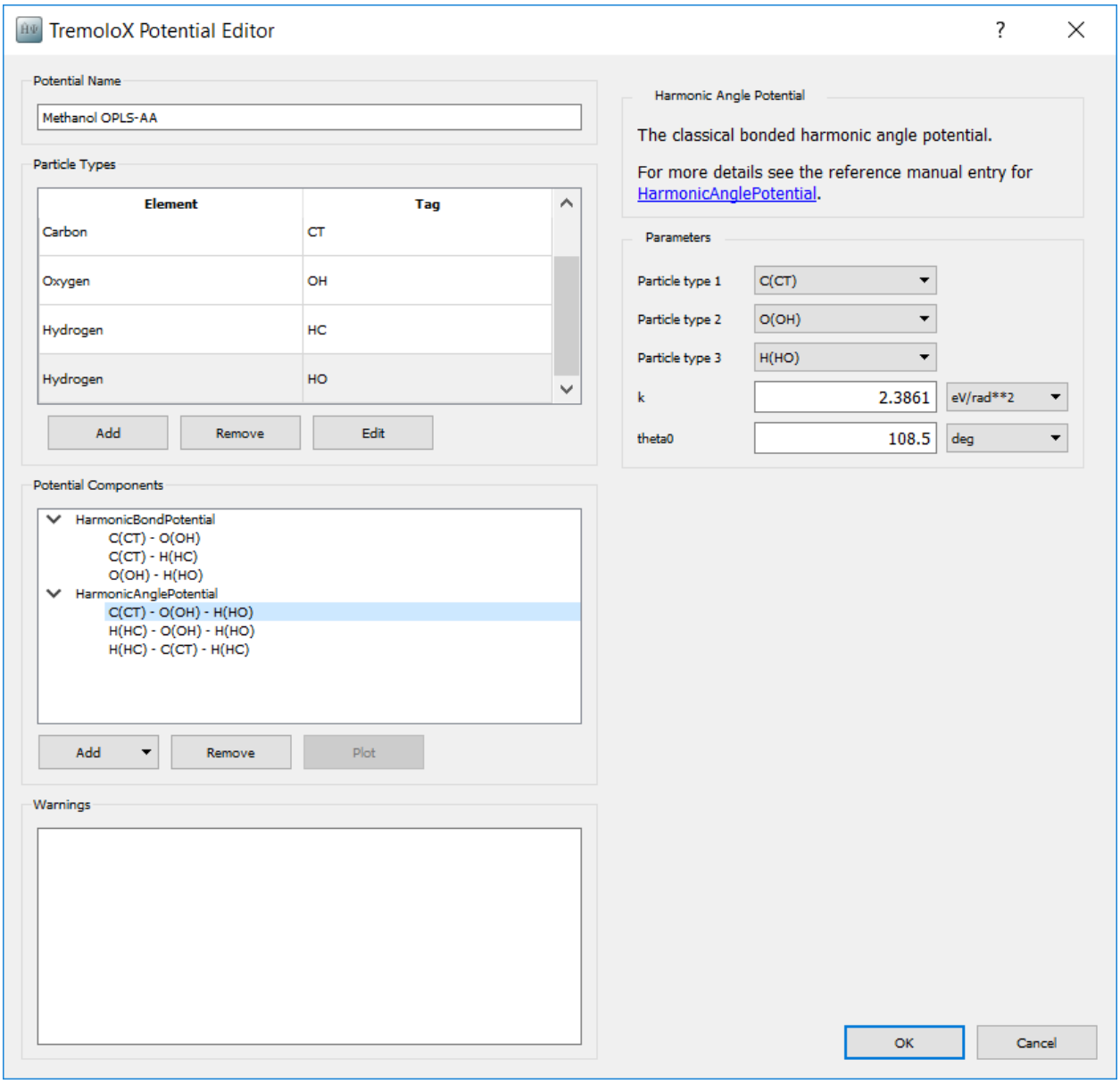
Angle stretching energy E_{angle} is approximated by another simple harmonic spring equation:

$$E_{angle} = \sum_{ijk} K_{ijk} \left(\theta_{ijk} - \theta_0\right)^2$$

There are three different types of angles in methanol. The parameters for these angles are given in the following table:

Type I	Type J	Type K	K_{ijk} / eV rad^{-2}	θ_0 / degree
CT	OH	HO	2.3861	108.5
HC	CT	OH	1.5184	109.5
HC	CT	HC	1.4317	107.8

To add the angle potentials, in the *Potential Components* box select *Add* → *Add potential component*. From the list of potential types select *HarmonicAnglePotential*. Selecting a potential displays the parameters for it one the right hand side. Add three potentials and enter the parameters for each bond type. The input for the `CT - OH - HO` angle type is shown below.



Torsion Terms

In methanol, there is only one bond torsion term. This extends across the type `HC - CT - OH - HO`. The bond twisting energy $E_{torsion}$ in this term is approximated by a Fourier expansion:

$$E_{torsion} = K(1 + \cos(3\phi))$$

The general torsion term in the OPLS-AA potential is actually a more complicated 3-term Fourier expansion. This potential cannot be added in directly in the interface, but can be easily set up with a small modification to the script. To create a place for the potential, in the *Potential Components* box select *Add* → *Add potential component*. From the list of potential types select *CosineTorsionPotential*. We will set the parameters later when editing the final script.

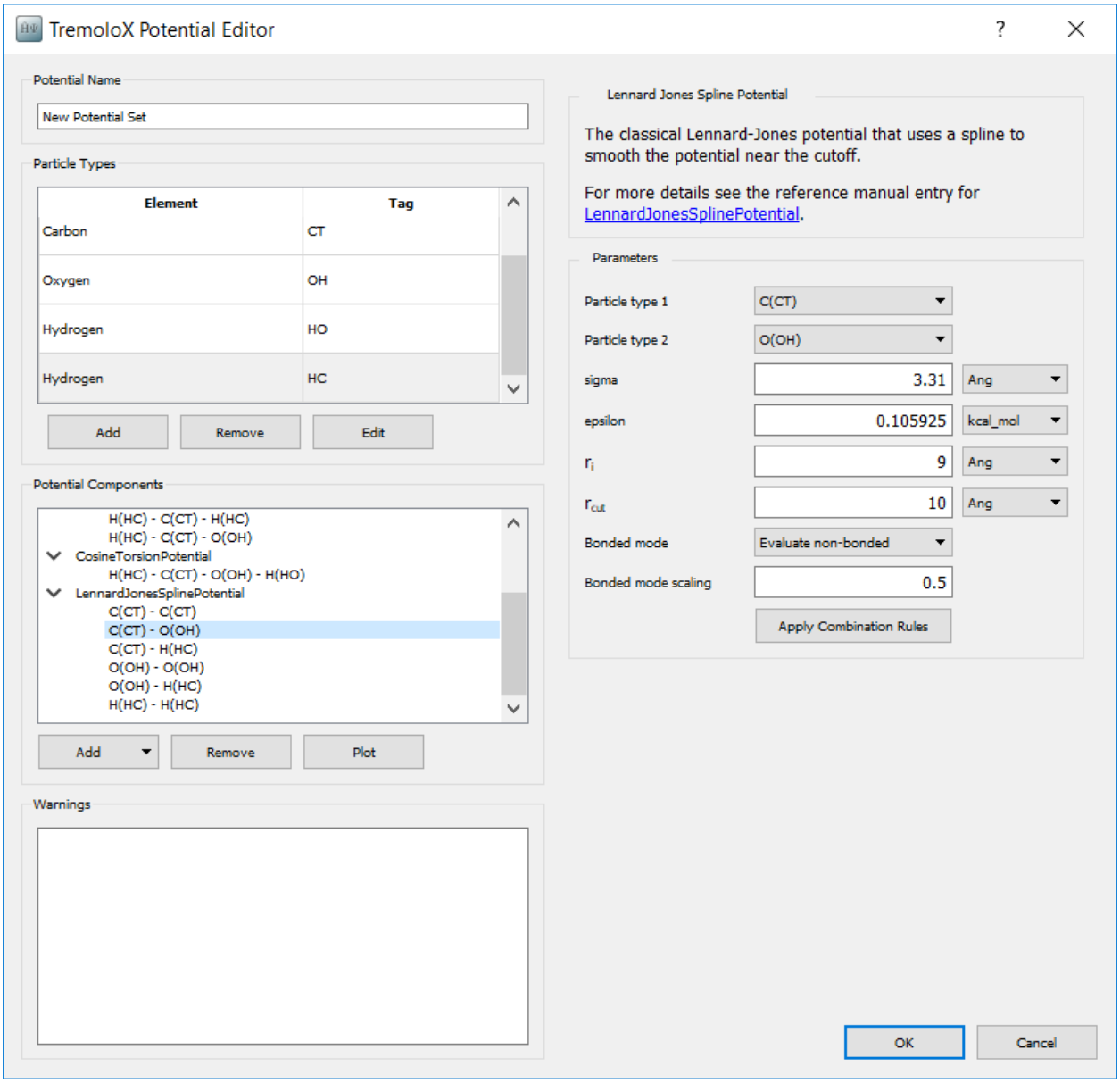
Van der Waals Terms

While parameters have been added for single atom types in the particle type definition, it is also necessary to add terms for pairs of types. As the atom type *HO* is not a van der Waals site, pair terms involving this atom do not need to be added. The remaining 3 types then gives 6 unique combinations.

For each combination in the *Potential Components* box, select *Add* → *Add potential component*. From the list of potential types select *LennardJonesSplinePotential*. The parameters for each can now be set on the right hand side.

- Sigma and epsilon can be filled in from the single atom type parmameters by pushing *Apply Combination Rules*
- Set r_{cut} to `10` Å. This sets the distance beyond which potential terms are ignored
- Set r_i to `9` Å. This sets the distance beyond which potential terms are scaled down by a spline function, so that they are zero at the cutoff. This is necessary so that there is not a sudden increase in forces as atoms come within range of each other.
- Set *Bonded mode scaling* to `0.5`. This scaling is for atoms connected through 3 bonds and is specified by the OPLS-AA potential.

An example for the `CT - OH` nonbonding potential is shown below.



Electrostatic Terms

Atomic charges were added to the model in the definition of particle types. A summation method for the electrostatic terms still needs to be added.

In the *Potential Components* box select *Add* → *Add potential component*. From the list of potential types select *CoulombSPME*. This selects *Smooth Particle Mesh Ewald* summation of electrostatic terms. This method uses Fourier transforms to correctly estimate the long-range electrostatic interactions.

In the parameters section on the right hand side:

- Set r_{cut} to 9 Å.
- Set Accuracy to `0.001`.

The other default parameters are suitable for the calculation.

Having added all of these potentials terms, click *OK* on the **Potential Editor** and on the **ForceFieldCalculator**

Relaxing the starting structure

Optimizing the structure

Now that the forcefield has been defined, calculations can be added to the script.

Note: This tutorial follows the general methodology for running molecular dynamics simulations. For more information on specifics of the method and settings see the tutorials:

- *How to Setup Basic Molecular Dynamics Simulations*
- *Diffusion in Liquids from Molecular Dynamics Simulations*

The first step is to optimise the geometry to remove any large forces from the starting configuration. Large initial forces can cause integration problems in the following molecular dynamics calculations. It is not important that the structure is completely optimised, only that any atom-atom overlaps are removed, which usually happens in the first few steps of a geometry optimisation.

In the **Scripter** add a *Optimization* → *OptimizeGeometry* block. The default parameters for this block are sufficient.

Obtaining the right density

When the structure was built, the density was too low to make it easier to add methanol molecules into the bulk structure. Now the cell needs to be contracted to approximately the right density. This can be done with a simple molecular dynamics calculation.

Add a molecular dynamics block using the *NPT Berendsen* barostat. This barostat is more stable but also less accurate than the MTK barostat, making it useful for creating relaxed structures from inital guesses. A run of 100ps is typically sufficient for the system to condense to the liquid density.

Equilibrating the structure

Add a molecular dynamics block using the *NPT Martyna Tobias Klein* barostat. In this calculation 100ps is an adequate time for the simulation to equilibrate. Equilibration time can vary greatly however depending on the system being investigated. The simulation should also be started using the velocities from the previous calculation, which can be done by setting the *Initial Velocity* type to `Configuration Velocities`.

Setting the production calculation

Add a molecular dynamics block using the *NPT Martyna Tobias Klein* barostat. Set the dynamics to run for 500ps. The appropriate length of the production simulation depends largely on the system being studied and also the desired accuracy of the final result. Two specific things to that should be set in this part are:

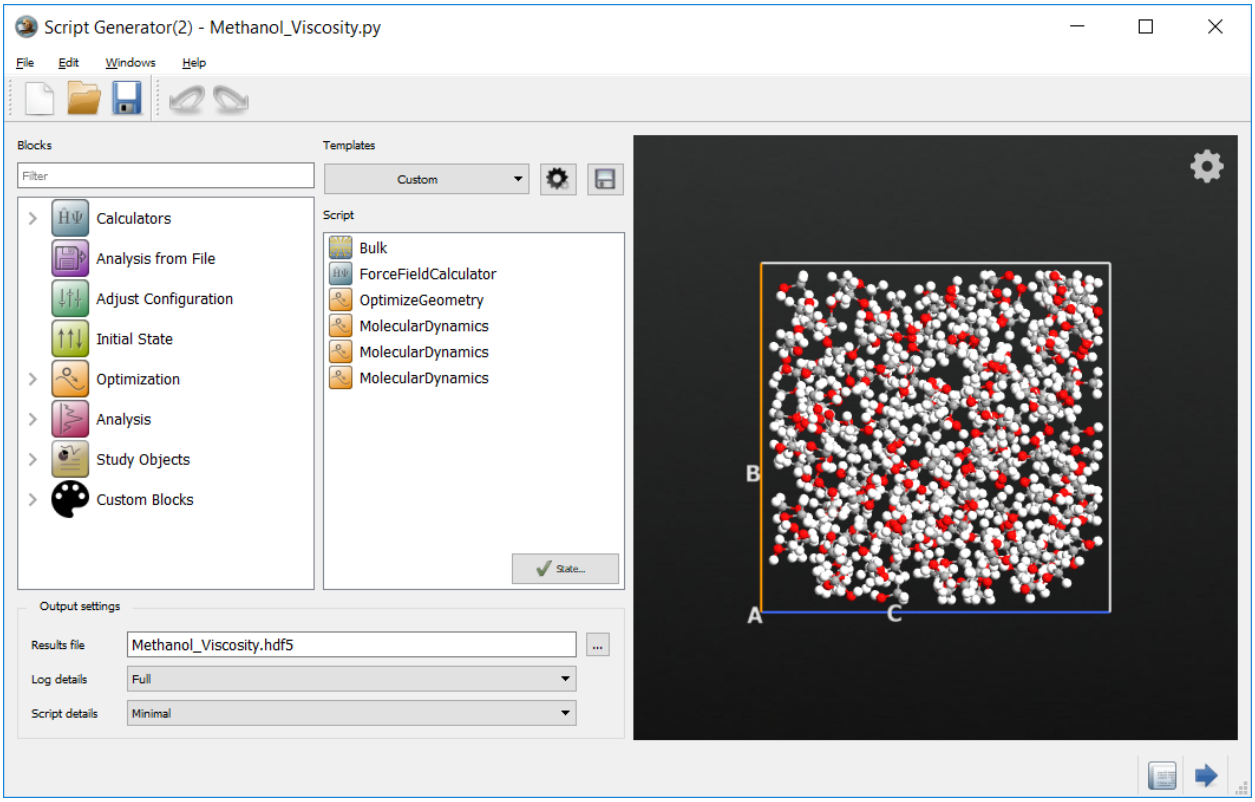
- Set the *Save Trajectory* box to `Production_Trajectory.hdf5`
- Set the *Log interval* box to 1000

Both of these details will be used in the analysis part of the tutorial.

Modifying the script

Adding the torsion terms

Now that the basic calculation is created, **Scripter** set the output file to `Methanol_Viscosity.hdf5`. The script generator should look like:



The script can now be sent to the **Editor** using the **Send To** icon . The torsion parameters can now be set. The torsion potential is defined in the section:

```
_potential = CosineTorsionPotential(
   particleType1 = ParticleIdentifier('H', ['HC']),
   particleType2 = ParticleIdentifier('C', ['CT']),
   particleType3 = ParticleIdentifier('O', ['OH']),
   particleType4 = ParticleIdentifier('H', ['HO']),
   k = [0.0*eV, ],
   n = [0, ],
   delta = [0.0*Radians, ],
)
potentialSet.addPotential(_potential)
```

Replace this block with the block

```
_potential = CosineTorsionPotential(
   particleType1 = ParticleIdentifier('H', ['HC']),
   particleType2 = ParticleIdentifier('C', ['CT']),
   particleType3 = ParticleIdentifier('O', ['OH']),
```

(continues on next page)

(continued from previous page)

```
    particleType4 = ParticleIdentifier('H', ['HO']),
    k = [0.176*kiloCaloriePerMol, ],
    n = [3, ],
    delta = [0.0*Radians, ],
)
potentialSet.addPotential(_potential)
```

Save the edited script.

Defining the hook function

To calculate the viscosity the values of the pressure tensor during the simulation are required. As the pressure changes rapidly during the simulation, a high resolution of data is necessary. While saving structures into the trajectory also saves the pressure tensor, saving trajectory files with short timesteps quickly results in unreasonably large files. What is required is a method to save just the pressure information at each step.

In **QuantumATK** this can be easily achieved with hook functions. These are functions that are run before or after each molecular dynamics integration. Hook functions are designed for either controlling the simulation, or as in this case, recording and analysing various quantities during the simulation.

Note: For an example on how to use hook functions to modify the simulation, see the tutorial: *Youngs modulus of a CNT with a defect*

To use a hook function a Python object that implements the __call__ function must first be added. The arguments passed to this function are:

- *step*: The step number that the simulation is up to when the function is called
- *time*: The amount of time that has elapsed in the simulation so far
- *configuration*: A *BulkConfiguration* object containing the current structure
- *forces*: An array of the forces on each atom
- *stress*: The stress tensor for the current structure

Using these values it is possible to create many custom analysis and data recording methods. In the present case the pressure tensor needs to be recorded. This can be done by the *PressureRecordingUtility* class defined in the script below.

```
# Define the class that does the analysis for the MD simulation
class PressureRecordingUtility:
    ''' Calculate and record the volume and pressure tensor at each step '''

    def __init__(self):
        ''' Initialse the arrays used to store the data '''
        self._time_list = []
        self._volume_list = []
        self._pressure_list = []

    def __call__(self, step, time, configuration, forces, stress):
        ''' Function called at each step that calculates and records the pressure tensor '''

        # Add the current time onto the list
```

(continues on next page)

(continued from previous page)

```
        self._time_list.append(time)

        # Get the cell volume and add that to the list too
        volume = configuration.bravaisLattice().unitCellVolume()
        self._volume_list.append(volume)

        # Get the velocity component of the pressure tensor
        velocity = configuration.velocities()
        masses = configuration.atomicMasses()
        velocity_tensor = ((velocity.reshape(-1,3,1) * velocity.reshape(-1,1,3)) *␣
↪masses.reshape(-1,1,1)).sum(axis=0)

        # Construct the pressure tensor
        pressure_tensor = (velocity_tensor/volume) - stress

        # Add the pressure tensor to the list
        self._pressure_list.append(pressure_tensor)

    def times(self):
        ''' Function that returns the list of times saved during the simulation '''
        return Units.PhysicalQuantity(self._time_list)

    def volumes(self):
        ''' Function that returns the list of volumes saved during the simulation '''
        return Units.PhysicalQuantity(self._volume_list)

    def pressure_tensors(self):
        ''' Function that returns the list of pressure tensors saved during the simulation '
↪'
        return Units.PhysicalQuantity(self._pressure_list)
```

Paste this code at the top of the simulation script and then save the edited script.

Using the hook function in the molecular dynamics simulation

Now that an appropriate hook function is defined, **QuantumATK** needs to be told how to use this function during the simulation.

Find the section of the script where the final production phase of the molecular dynamics calculation is carried out. At the top of this section add the following line:

```
pressureRecorder = PressureRecordingUtility()
```

This line declares an instance of the class used to record the pressure data, allocating a memory space in which to store it.

Now the **MolecularDynamics** call needs to be changed to add the hook function. Edit this function call to include the argument:

```
post_step_hook=pressureRecorder
```

This tells **QuantumATK** to use the *pressureRecorder* instance of the *PressureRecordingUtility* to run and store the data at each step.

Note: The **MolecularDynamics** function supports arguments *pre_step_hook* and *post_step_hook* for running the hook function before or after the molecular dynamics integration step respectively.

Finally the data collected by the *pressureRecorder* object needs to be written out to a file for later processing. This can be done by writing the data to a hdf5 file. To do this add the following code to the end of the script.

```
data_file = 'md_methanol_data.hdf5'
temperature = 300 * Kelvin

nlsave(data_file, pressureRecorder.times(), "Time")
nlsave(data_file, pressureRecorder.pressure_tensors(), "Pressure")
nlsave(data_file, pressureRecorder.volumes(), "Volume")
nlsave(data_file, pressureRecorder.volumes().mean(), "Volume_Average")
nlsave(data_file, temperature, "Temperature")
nlsave(data_file, md_trajectory.timeStep(), "Time_Step")
```

Save the edited script.

Running the calculation

All of the necessary parts of the script are now in place. The script can be sent to the **Job Manager** and run by using the icon.

Analyzing the results

Viewing the change in pressure

When using the NPT ensemble, it is important to check how the pressure was controlled during the simulation. Ideally the pressure should average out to the target value over the course of the simulation. If this does not happen a longer equilibration or simulation time may be needed.

When the NPT ensemble is used the pressure is written to the HDF5 file by default. To visualise this pressure a script can be used to read it from the HDF5 file and plot it. To do this open the **Editor** using the and paste in the following script.

```
import pylab

# Read in the existing trajectory to ge the data
trajectory_file = "Production_Trajectory.hdf5"
trajectory = nlread(trajectory_file, MDTrajectory)[-1]
time = trajectory.times()
pressure = trajectory.pressures()

# Set the average and target pressures
p_tot_avg = numpy.cumsum(pressure.inUnitsOf(GPa)) * GPa / (numpy.arange(pressure.
↪shape[0])+1)
p_target = numpy.ones(pressure.shape[0]) * Pa

# Plot the resulting pressure and average pressure
```

(continues on next page)

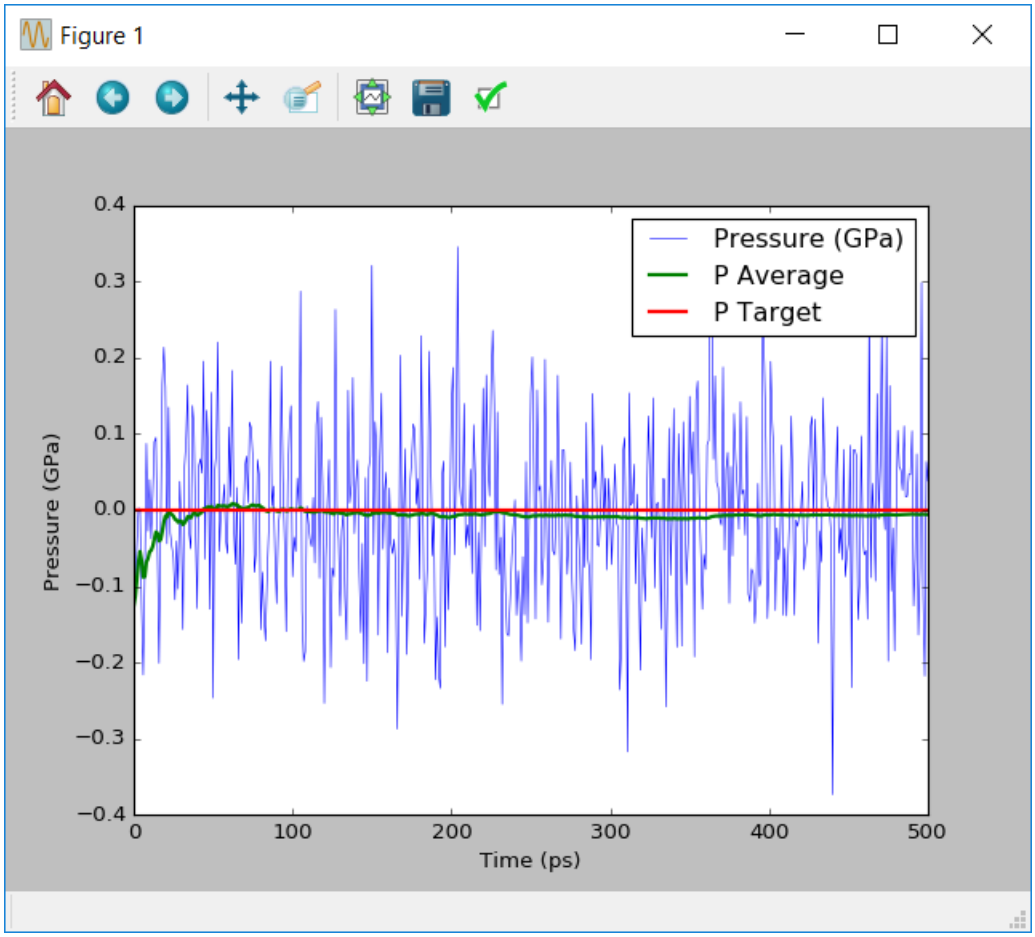
(continued from previous page)

```
pylab.figure()
pylab.plot(time.inUnitsOf(picosecond), pressure.inUnitsOf(GPa), label='Pressure (GPa)',␣
↪linewidth=0.5)
pylab.plot(time.inUnitsOf(picosecond), p_tot_avg.inUnitsOf(GPa), label='P Average',␣
↪linewidth=2.0)
pylab.plot(time.inUnitsOf(picosecond), p_target.inUnitsOf(GPa), label='P Target',␣
↪linewidth=2.0)
pylab.xlabel('Time (ps)')
pylab.ylabel('Pressure (GPa)')
pylab.legend()
pylab.show()
```

The script can also be directly downloaded here. `pressure_analysis.py`

Run this script by sending it to the **Job Manager**. The script should display a graph similar to the one below.

Tip: Analysis scripts like this can also be run from the command line using the **atkpython** command



Note that during the dynamics simulation the pressures swing by approximately 0.2GPa. These large pressure changes are typical for molecular dynamics simulations. What is important is that as the simultion runs the average pressure converges to the target pressure of 0.1MPa.

Calculating the viscosity from the Einstein relationship

Using the data collected by the hook function during the molecular dynamics simulation, it is possible to calculate and visualise an estimate of the viscosity.

This can be done by using an **ATKPython** script. Open a new script in the **Editor** and paste in the following script.

```
import pylab
import scipy

# Read the data that is stored in the HDF5 file from the simulation
data_file = 'md_methanol_data.hdf5'
time = nlread(data_file, object_id="Time_8")[-1]
pressure_tensor = nlread(data_file, object_id="Pressure_8")[-1]
volume_avg = nlread(data_file, object_id="Volume_Average_8")[-1]
temperature = nlread(data_file, object_id="Temperature_8")[-1]
time_step = nlread(data_file, object_id="Time_Step_8")[-1]

# Set up the calculation to create 100 time based estimates of the viscosity
N = pressure_tensor.shape[0]
N_steps = 101
skip = int(N/100)
time_skip = time[::skip]

# Calculate the off-diagonal elements of the pressure tensor
P_shear = numpy.zeros((5,N), dtype=float) * Joule / Meter**3
P_shear[0] = pressure_tensor[:,0,1]
P_shear[1] = pressure_tensor[:,0,2]
P_shear[2] = pressure_tensor[:,1,2]
P_shear[3] = (pressure_tensor[:,0,0] - pressure_tensor[:,1,1]) / 2
P_shear[4] = (pressure_tensor[:,1,1] - pressure_tensor[:,2,2]) / 2

# At increasing time lengths, calculate the viscosity based on that part of the␣
↪simulatino
pressure_integral = numpy.zeros(N_steps, dtype=numpy.float) * (Second * Joule /␣
↪Meter**3)**2
for t in range(1,N_steps):
    total_step = t*skip

    for i in range(5):
        integral = scipy.integrate.trapz(
            y = P_shear[i][:total_step].inUnitsOf(Joule/Meter**3),
            dx=time_step.inUnitsOf(Second)
        )
        integral *= Second * Joule / Meter**3
        pressure_integral[t] += integral**2 / 5

# Finally calculate the overall viscosity
# Note that here the first step is skipped to avoid divide by zero issues
kbT = boltzmann_constant * temperature
viscosity = pressure_integral[1:] * volume_avg / (2*kbT*time_skip[1:])

# Print the final viscosity
print("Viscosity is {} cP".format(viscosity[-1].inUnitsOf(millisecond*Pa)))
```

(continues on next page)

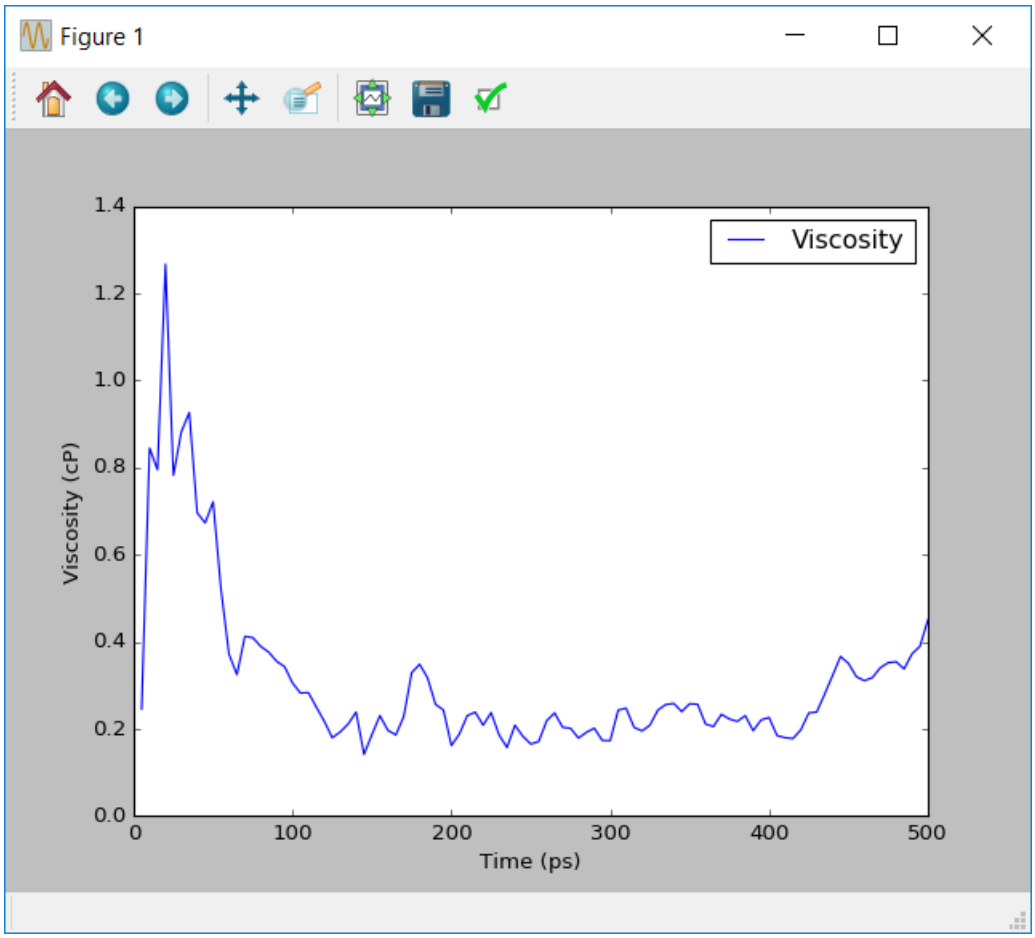
(continued from previous page)

```
# Display the evolution of the viscosity in time
pylab.figure()
pylab.plot(time_skip[1:].inUnitsOf(ps), viscosity.inUnitsOf(millisecond*Pa), label=
↪'Viscosity')
pylab.xlabel('Time (ps)')
pylab.ylabel('Viscosity (cP)')
pylab.legend()
pylab.show()
```

The script can also be directly downloaded here. `viscosity_analysis.py`

This script reads in the saved data from the HDF5 file saved from the hook function. It then calculates the viscosity using the Einstein relationship given in the theory section. Note that the integral is averaged over the 5 unique off-diagonal components of the pressure tensor, P_{xy}, P_{yz}, P_{xz}, $\frac{P_{xx}-P_{yy}}{2}$ and $\frac{P_{yy}-P_{zz}}{2}$. The script then prints the final viscosity at the simulation time limit, as well as showing how the estimate of the viscosity changed over the course of the simulation.

Below is a typical example of the viscosity calculated by the simulation. For comparison the experimental value of the density of methanol at 298K is 0.54cP. A rigid OPLS-AA model which only includes non-bonding interaction and keeps the configuration of each molecule fixed gives a viscosity of 0.43cP[Page 1267, 3]. Although the example gives similar results, there is still a reasonable amount of noise in the calcuated viscosity. Ideally this viscosity should converge to a distinct value. This indicates that more sampling of the pressure is required to improve the accuracy of the simulation.



Calculating the viscosity from the Green-Kubo relationship

Similar results for the viscosity can be obtained using the Green-Kubo relationship. A script outlining how this is achieved can be downloaded here. `green_kubo_analysis.py`

Advanced users of Python are encouraged to view and run the script to understand how it works. This script also demonstrates some of the power and flexibility of the **ATKPython** platform for not only running, but also analyzing simulations.

Extending the results

Running more independent trajectories

In the previous section it was seen that more simulations need to be done in order improve the accuracy of the estimated viscosity.

In **QuantumATK** simulations can be easily repeated by placing them within Python loops. In the case of methanol one strategy is to loop over both the equilibration and production phases of the simulation. As the velocities are re-randomised at the start of the equilibration phase this generates a series of independant trajectories. From these not only the average viscosity but also the error in the average viscosity can be easily estimated using standard statistical methods.

To loop over calculations in Python the code to be repeated can be placed in a *for* loop. To loop the viscosity calculation 30 times to produce different trajectories the simulation code needs to be placed in a *for* loop such as:

```
for loop in range(30):
    # Molecular dynamics simulation commands

# Futher commands outside the loop
```

Note: One of the important features of Python is that it uses indentation to determine the extent of code blocks. This determines which lines are inside or outside loops and logical statements. The indentation in the above example is how you distinguish code from inside or outside a Python loop.

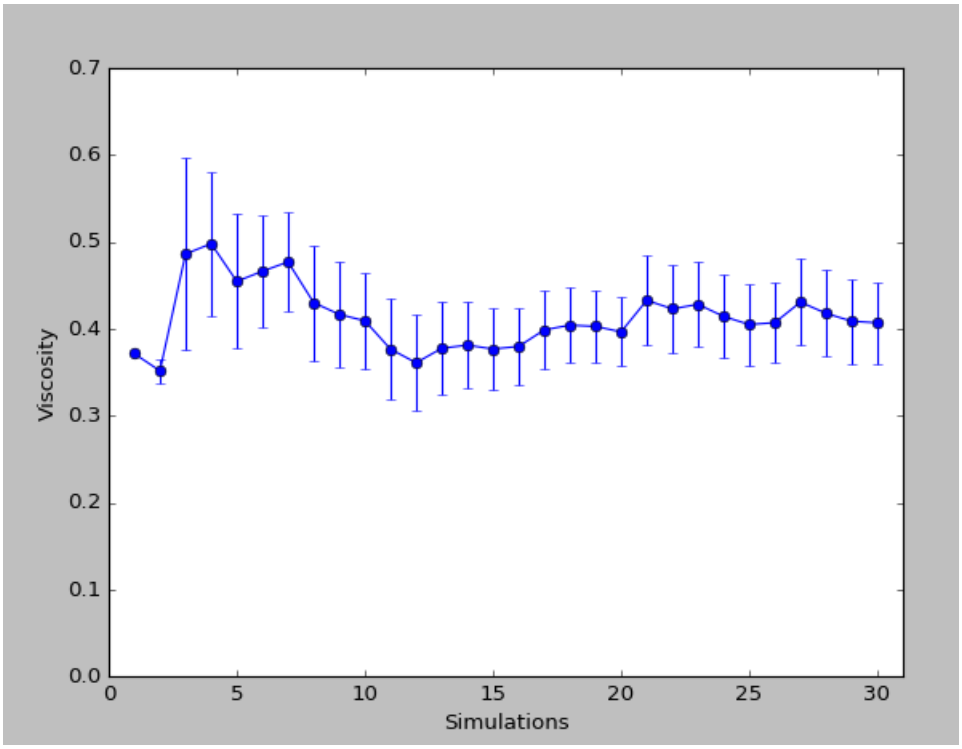
The only other necessary change in the script to create several trajectories is to modify how the output is done. In HDF5, files writing data with the same text descriptor overwrites the previous data. Unique descriptors are therefore needed. This is simply done by including the looping variable (in the above case the variable *loop*) into the name. This is done by changing the file-writing code to

```
data_file = 'md_methanol_data.hdf5'
temperature = 300

nlsave(data_file, pressureRecorder.times(), "Time_"+str(loop))
nlsave(data_file, pressureRecorder.pressure_tensors(), "Pressure_"+str(loop))
nlsave(data_file, pressureRecorder.volumes(), "Volume_"+str(loop))
nlsave(data_file, pressureRecorder.volumes().mean(), "Volume_Average_"+str(loop))
nlsave(data_file, temperature, "Temperature_"+str(loop))
nlsave(data_file, md_trajectory.timeStep(), "Time_Step_"+str(loop))
```

To read in the data from each trajectory for analysis the names need to be changed in the `nlread` commands in the same way.

Improving the accuracy

To demonstrate how the accuracy of the simulation can be improved using multiple runs, the simulation was repeated 30 times. The average obtained from these simulations was 0.41cP, with a standard error of 0.05cP. The graph below shows the change in the average viscosity and the standard error with the number of simulations.



References

3.9 Molecular Electronics

3.9.1 Building molecule–surface systems: Benzene on Au(111)

Version: 2017.0

In this tutorial you will learn how to use **QuantumATK** (**QuantumATK**) to construct two different adsorption configurations of a benzene molecule adsorbed on the gold (111) surface, here denoted Bz@Au(111). In particular, you will:

- Create the structure of bulk Au using the value of the lattice constant reported by Lui *et al.* in Ref.[1];
- Construct a slab model for the Au(111) surface according to the parameters reported in[Page 1287, 1];
- Attach a benzene molecule to the Au(111) surface slab model;
- Modify the atomic coordinates of the Bz molecule to construct configurations of Bz@Au(111) using the advanced building tools in the QuantumATK **Builder**.

[1] Wei Liu, VictorăG Ruiz, Guo-Xu Zhang, Biswajit Santra, Xinguo Ren, Matthias Scheffler, and Alexandre Tkatchenko. Structure and energetics of benzene adsorbed on transition-metal surfaces: density-functional theory with van der Waals interactions including collective substrate response. *New Journal of Physics*, 15(5):53046, 2013. URL: http://stacks.iop.org/1367-2630/15/i=5/a=053046.

Tip: Once you have constructed these geometries, you can optimize them using one of the computational methods implemented in QuantumATK, and perform further analysis.

Note: Many of the QuantumATK functions described in this tutorial are rather general, and can be used to build complex geometries other than molecules on surfaces, see for example the tutorial on *Building a molecular junction*.

Summary of workflow

The following is a summary of the main steps needed for building the Bz@Au(111) configurations. These steps are also shown in the short **video** available below. If needed, more detailed instructions are given in the sections after the video.

1. Create a new QuantumATK project and open the **Builder**.
2. Use the **Builder Database** to import the bulk gold primitive unit cell.
3. Use the **Lattice Parameters** plugin to change the gold lattice parameter to $a = 4.159$ Å, as reported in Ref.[1] using DFT and the PBE density functional.
4. Open the **Surface (Cleave)** plugin, and create the 3×3 Au(111) slab with 6 metal layers and 20 Å vacuum above it.
5. Use the **Molecular Builder** tool to add the benzene molecule to the slab configuration.
6. Select the Bz atoms, and use the **Rotate** plugin to rotate the molecule 90° around the y-axis such that the molecule can adsorb parallel to the slab surface plane.
7. Click the icon to add a new atom in at the geometrical center of the molecule.
8. Use the **Move** tool to move the molecule into the hcp-30° adsorption site. The extra atom added in the previous step is used to snap the molecule into position, and is afterwards removed from the configuration.
9. **Rename** the configuration and create a **copy** of it in the Stash.
10. **Rename** the copy, and use the **Rotate** plugin to rotate the Bz 30° around the z-axis to create the hcp-0° adsorption geometry.

Many more **video tutorials** are available on the QuantumATK YouTube channel.

Detailed instructions

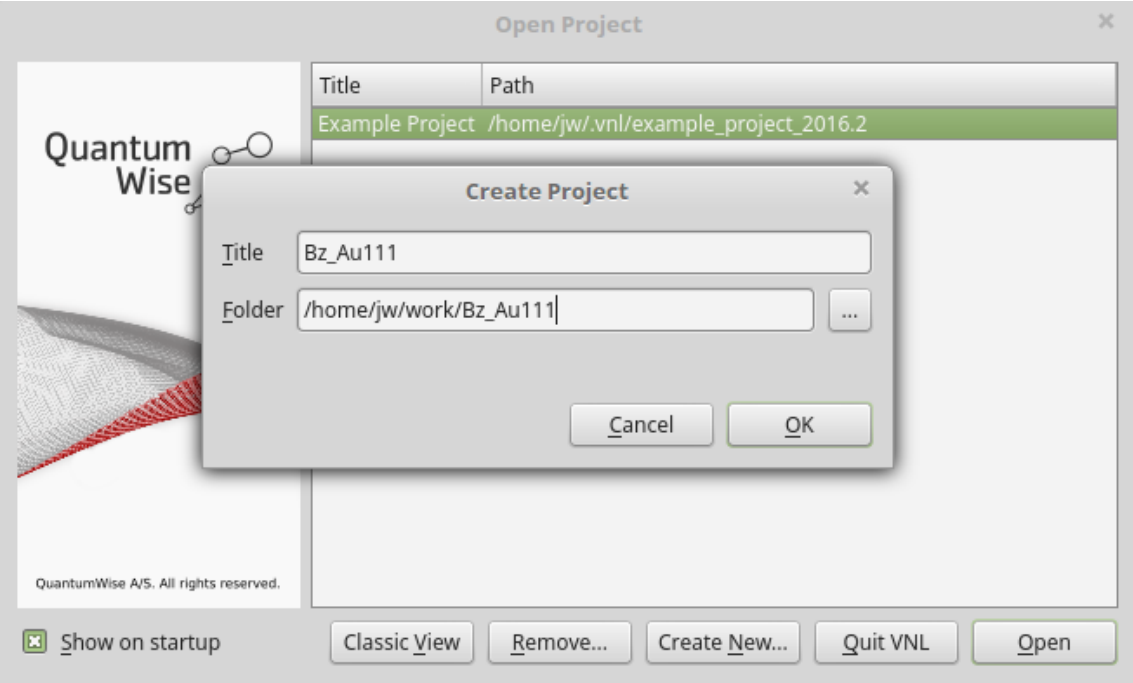
Building the Au(111) surface

You will here construct a slab model for the Au(111) surface starting from the gold primitive unitcell.

Bulk gold

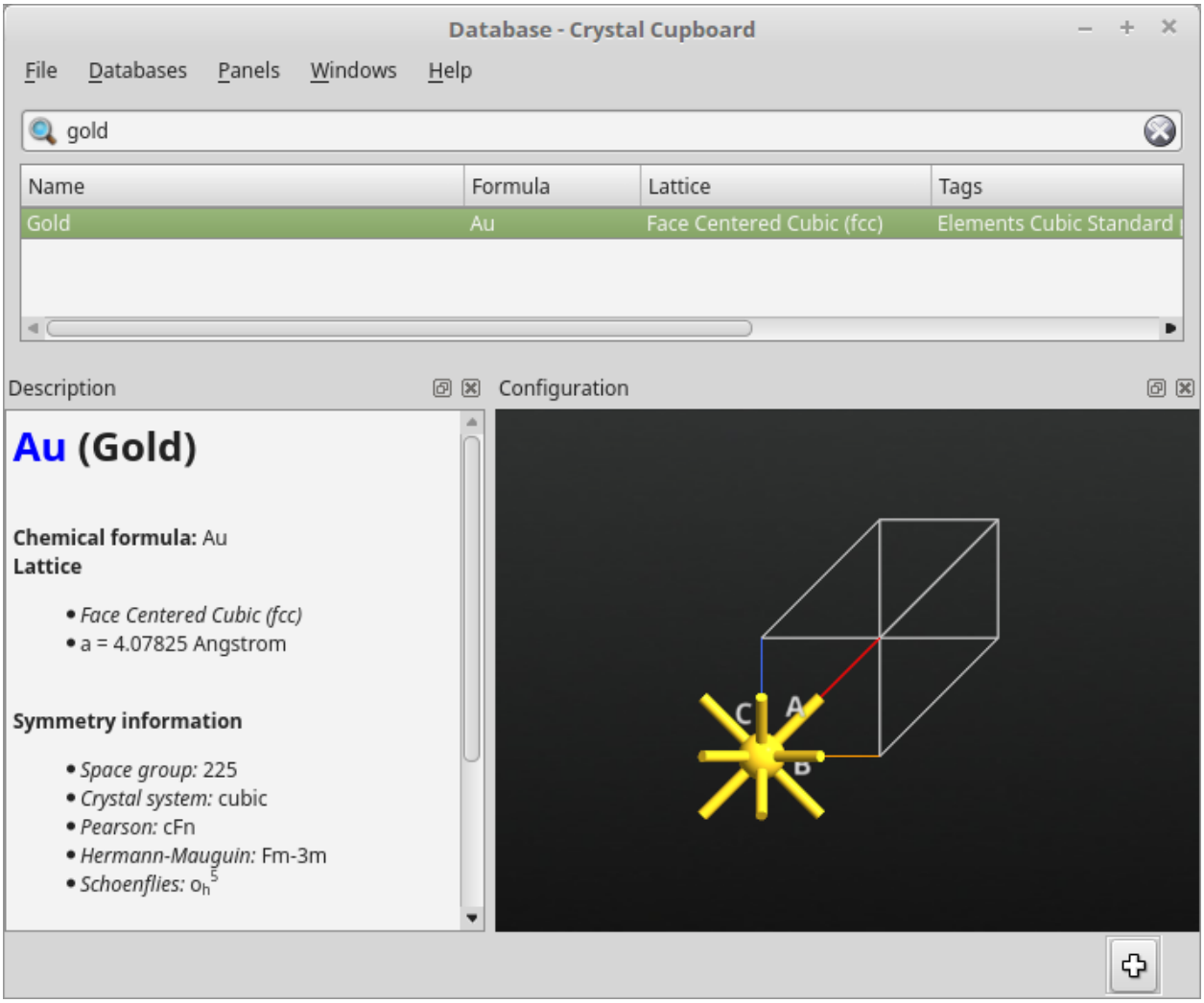
Open **QuantumATK** and create a new project. Give the project a title (here: *Bz_Au111*), select the folder where the project files should be stored (scripts, data files, etc.), and click **OK** to create the project.

click **Open** to open the project and start your QuantumATK session.



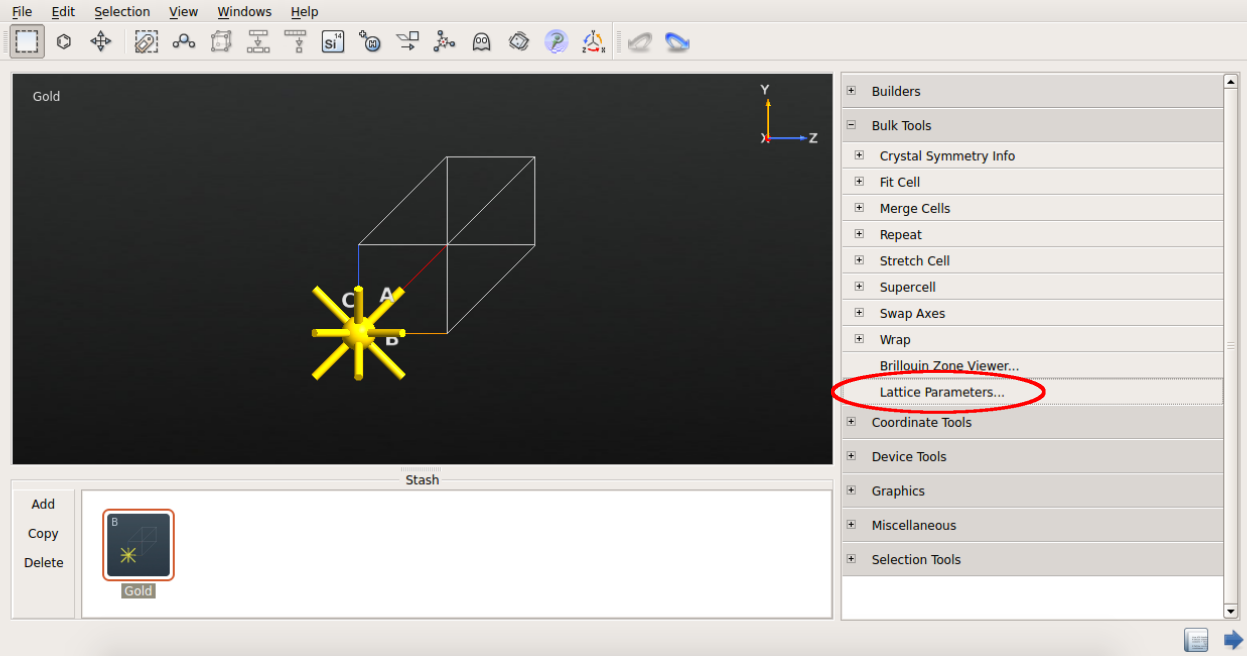
In the QuantumATK main window, open the **Builder**, and click *Add* → *From Database* next to the **Stash** to open a database of common materials in their experimental structures.

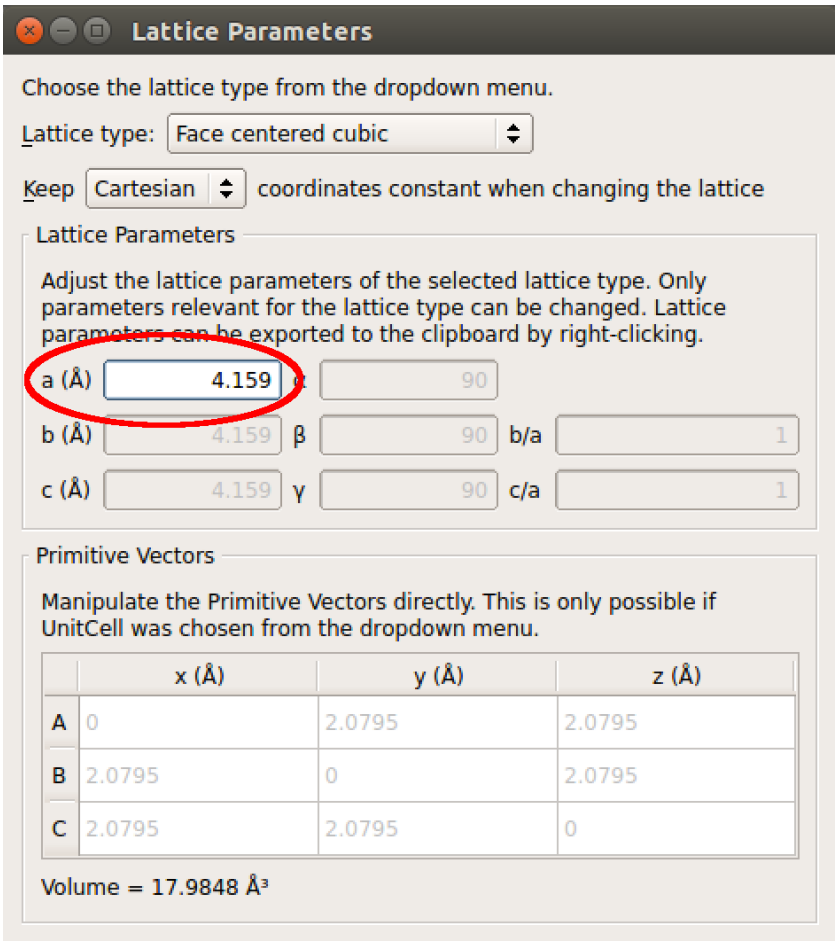
Search for gold using the search bar, and import the structure to the **Stash** by clicking the search result and then clicking the icon:



The gold primitive cell has been added as a **Stash** item, and may now be modified using the **Builder plugins**.

In this case, use the *Bulk Tools* → *Lattice Parameters* plugin to change the lattice parameter to that reported in Ref.[Page 1287, 1] using DFT and the PBE functional, $a = 4.159$ Å.



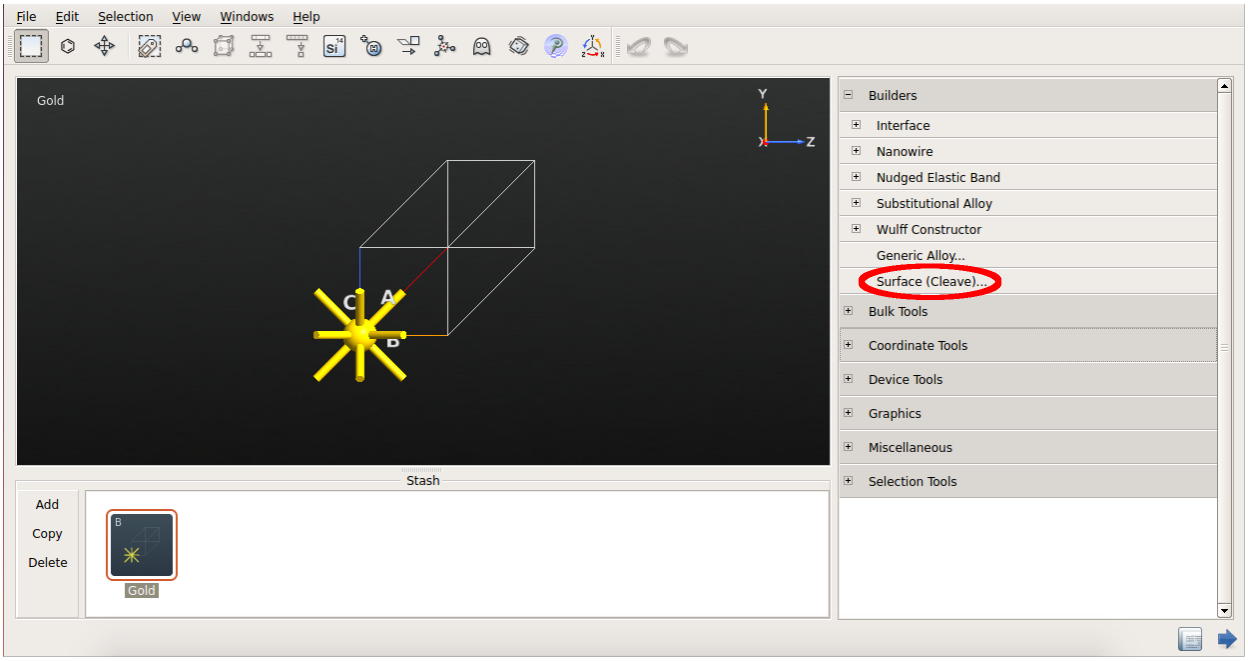


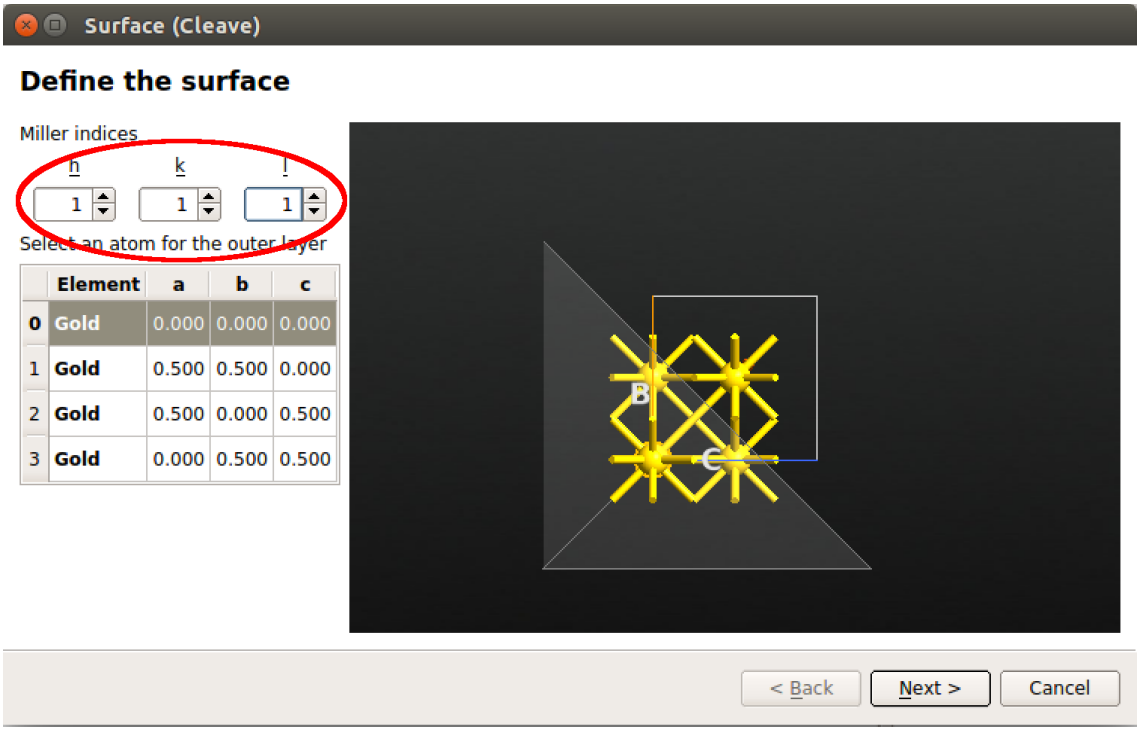
When you are done, close the plugin widget and go back to the Builder.

Au(111) slab

Next step is to cleave the bulk gold along the (111) direction in order to create a slab model of the Au(111) surface.

In the **Builder**, use the *Builders* → *Surface (Cleave)* plugin to cleave the Au crystal using the Miller indices $(h, k, l) = (1, 1, 1)$:



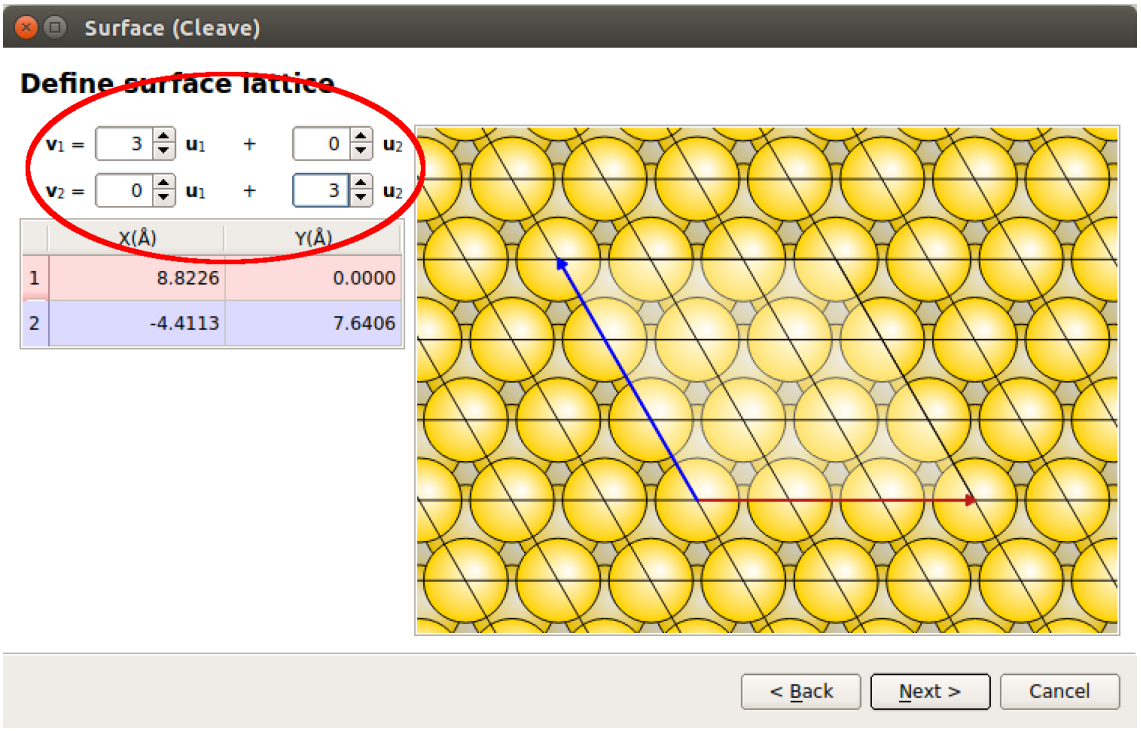


The default supercell lateral size is 1×1. This is sometimes sufficient, but in the present case we aim for a 3×3 slab in

order to minimize in-plane interactions between neighboring Bz molecules.

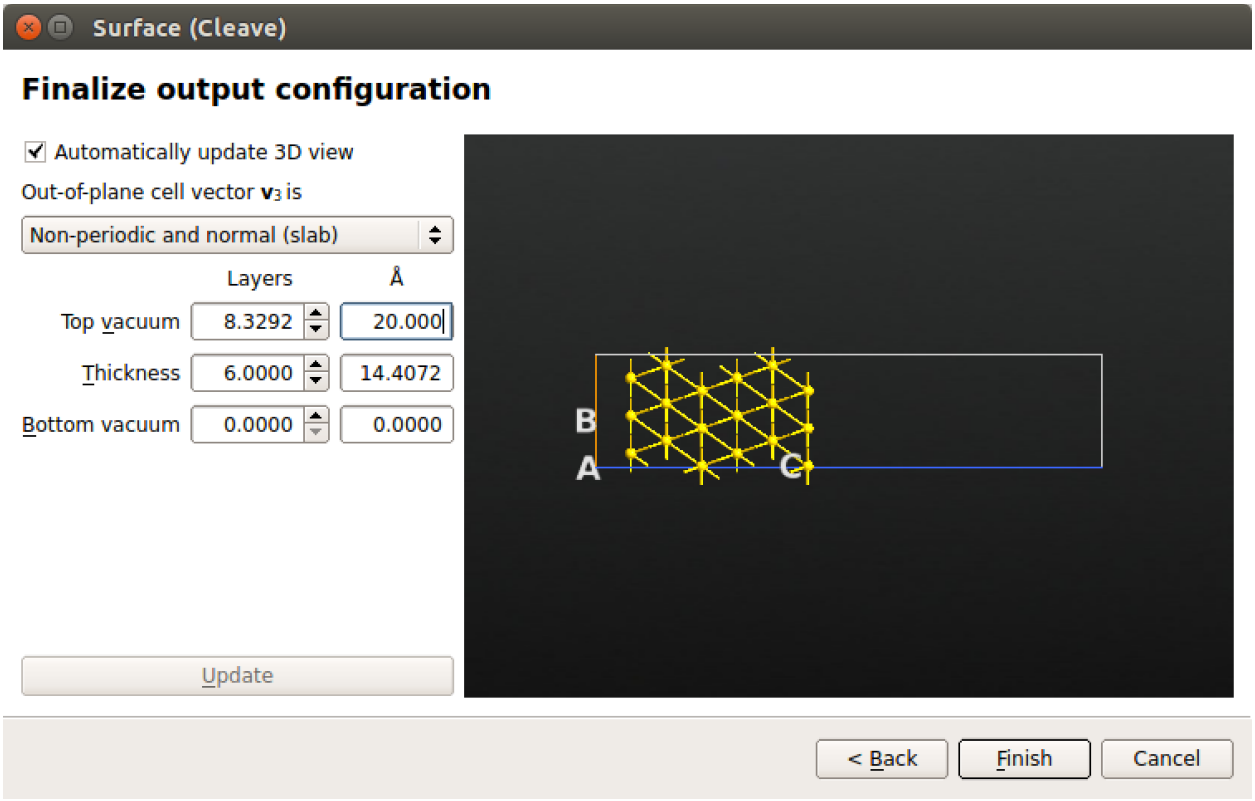
Therefore, increase the lengths of the lattice vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, such that

- $v_1 = 3u_1 + 0u_2$
- $v_2 = 0u_1 + 3u_2$

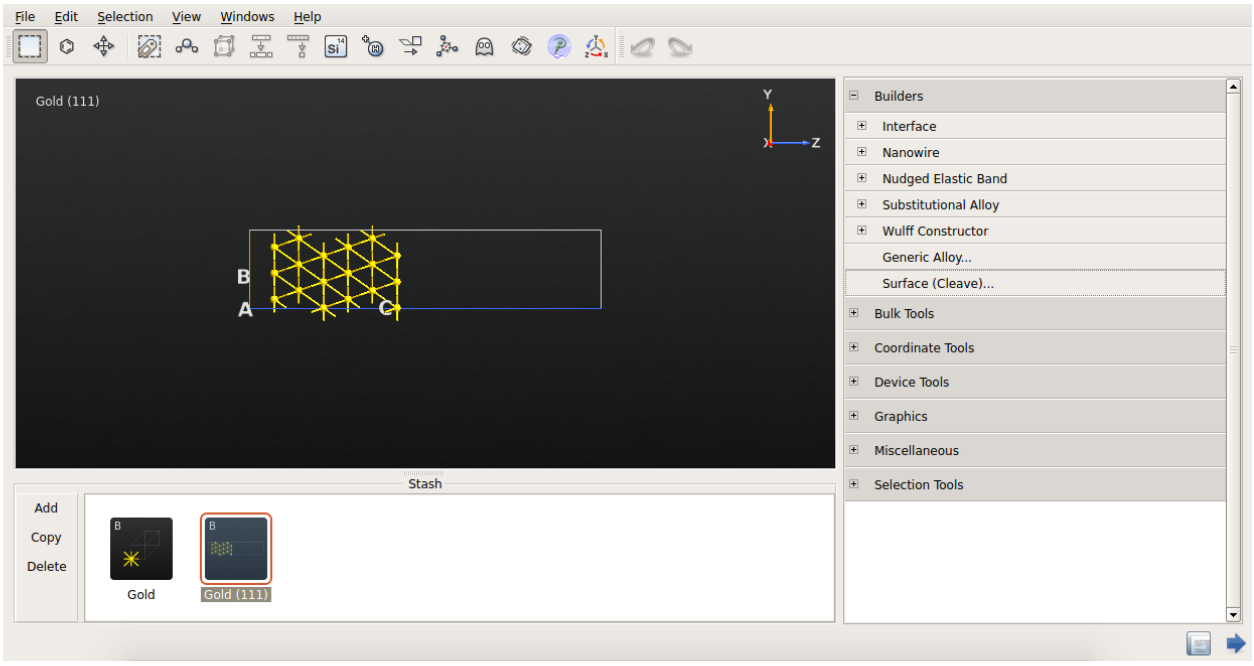


Tip: You can also modify the lattice vectors **interactively** by using the mouse to manually displace the red and blue vectors in the right-hand window showing the in-plane cell.

Finally, use the drop-down menu to select the out-of-plane cell vector $\mathbf{v}_3$ to be **Non-periodic and slab-like** such as to model a surface with vacuum above and below it. Also increase the **Top vacuum** to 20 Å and the slab **Thickness** to 6 layers, as shown in the figure below.



Once you are done, click **Finish** to add the Au(111) slab configuration to the **Stash** and close the plugin widget.

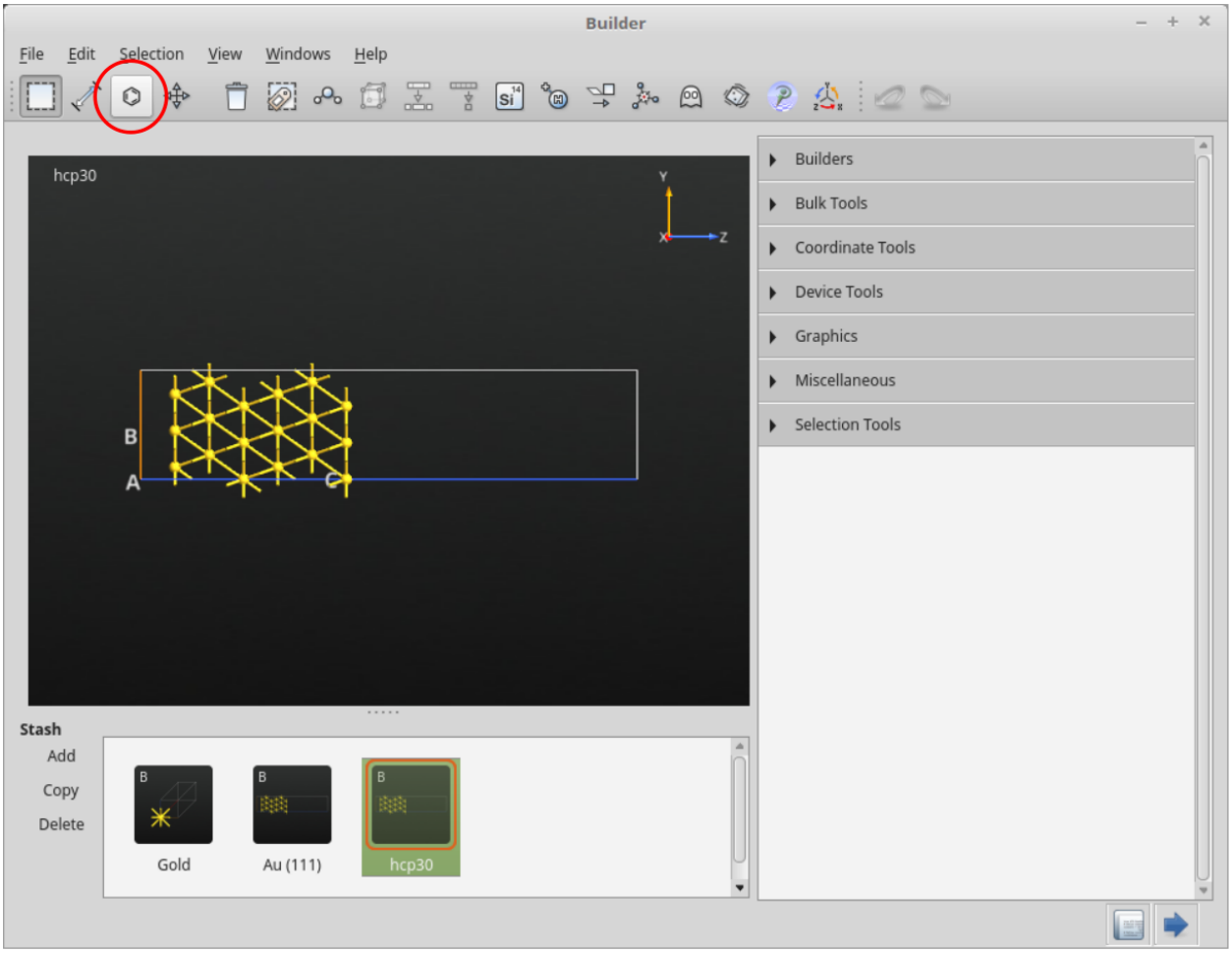


Attaching benzene to Au(111)

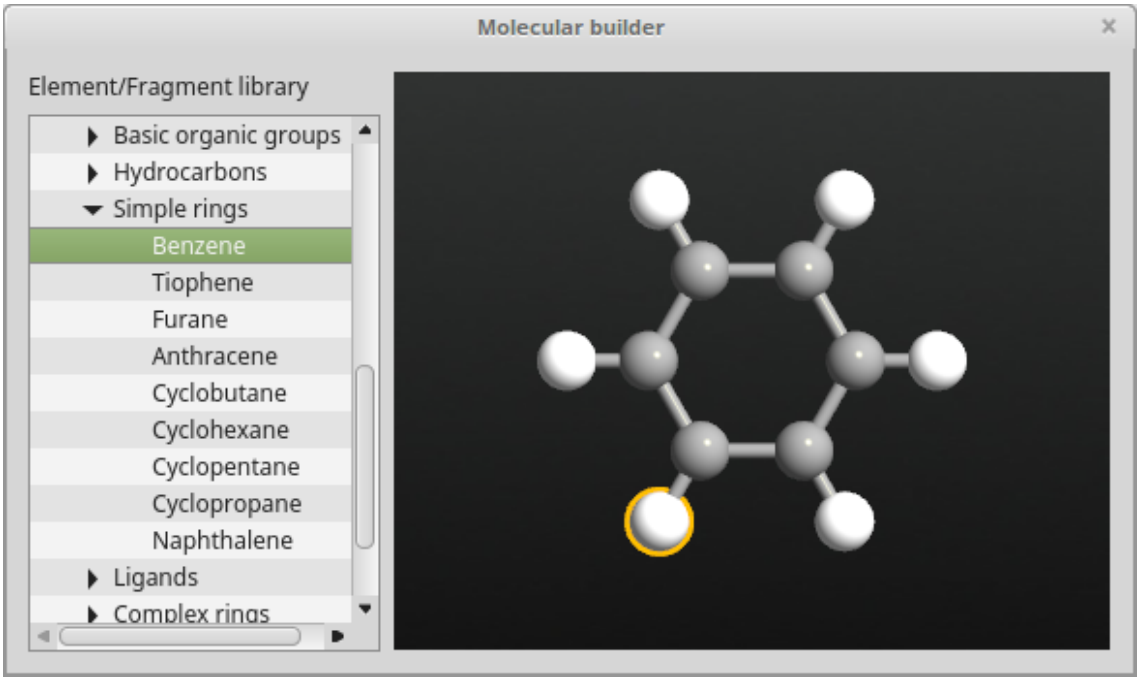
The final step is of course to attach a benzene molecule to the Au(111) slab. We will first aim for the hcp-30° configuration reported in[Page 1287, 1].

Select the *Gold(111)* **Stash** item and rename it to *Au(111)* by using mouse right-click or F2 on the keyboard. Then create a copy of the item, again using mouse right-click. Rename the copy *hcp30*.

In the **Builder**, click the icon to open the **Molecular Builder**.



In the **Molecular Builder**, select *Fragments* → *Simple rings* → *Benzene* from the drop-down menu.

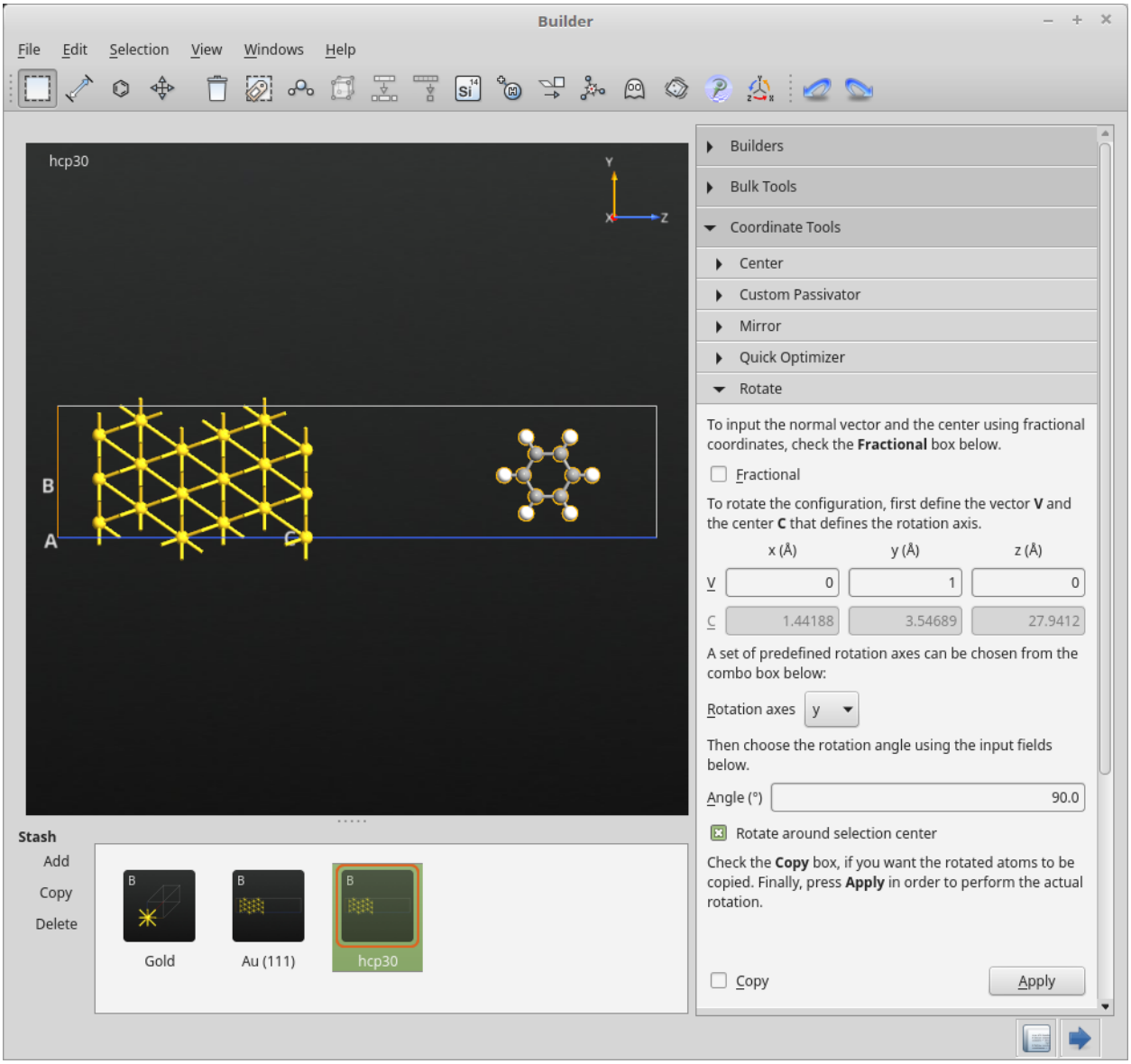


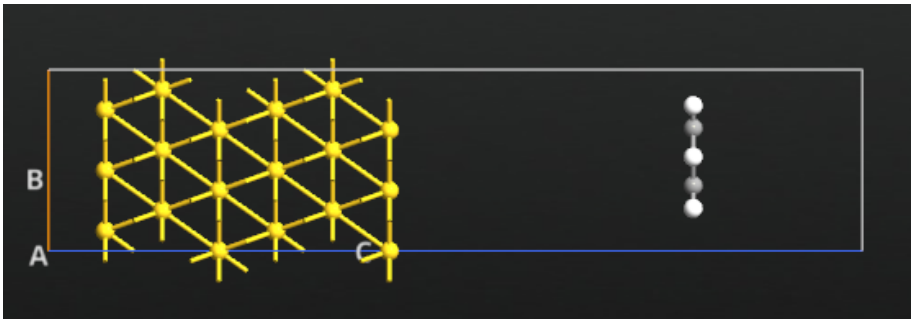
You can now insert the benzene molecule into the Au(111) configuration: Click somewhere in the Au(111) vacuum region to place the molecule there. Then close the Molecular Builder widget.

Warning: Take care to click only once to add the Bz molecule to Au(111). For each click, a new molecule will be added at exactly the same position, so that its atoms will overlap with those of the already existing molecule!

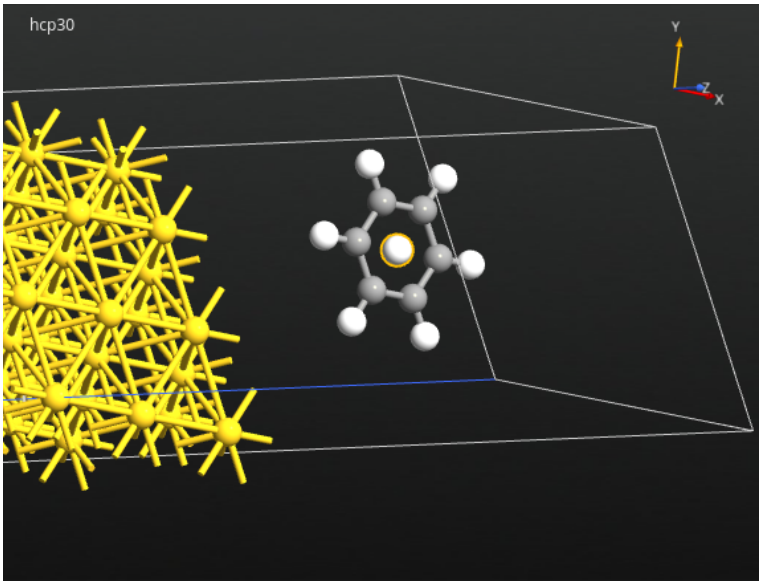
Remember that you can always check for the presence of overlapping atoms by using the *Selection Tools* → *Close Neighbors* plugin and choosing a relatively small threshold (for example, 0.001 Å).

You will then need to rotate the molecule 90°. First select all Bz atoms by using the mouse to draw a rectangle around the molecule. Then open the *Coordinate Tools* → *Rotate* plugin. Set the rotation axis to y and the rotation angle to 90°, and make sure the check-box *Rotate around selection center* is ticked. Click **Apply** to perform the rotation.





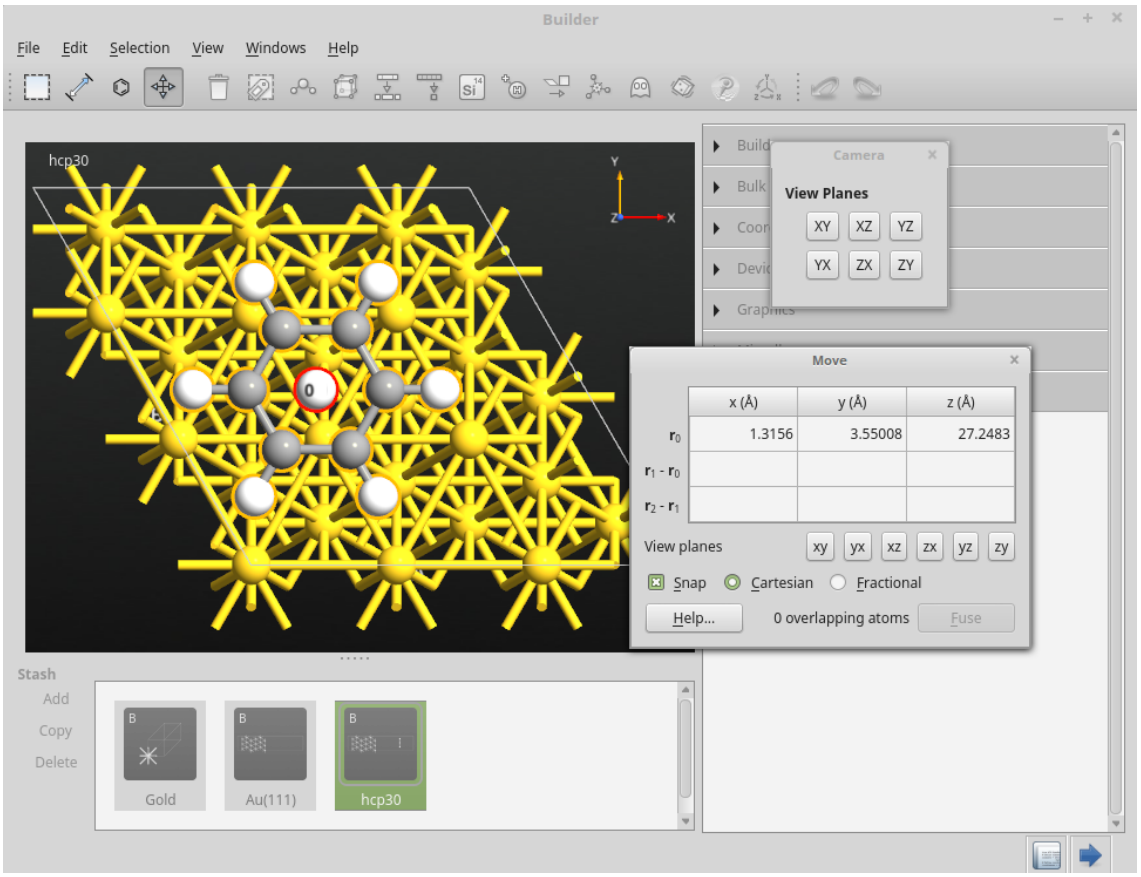
Next step is to move the Bz molecule into position in the hcp-30° adsorption site. First, make sure all Bz atoms are selected. Then click the icon to add a new atom in at the geometrical center of the molecule. You will use this anchor atom to snap the molecule into position above the surface.



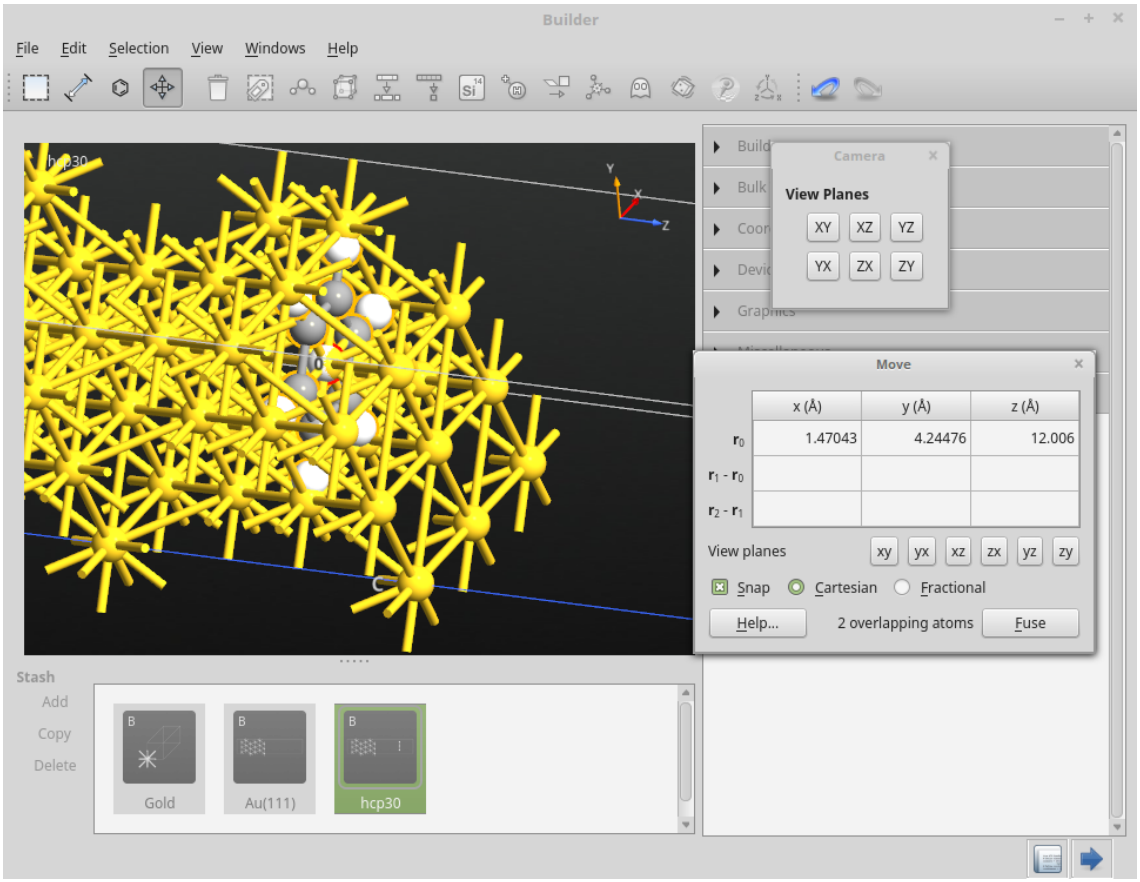
Select again all benzene atoms, including the extra atom, and rotate the view such that you see a top-view of the configuration.

Tip: The **Camera** tool may be useful: Select the **XY** view plane to get a perfect view down along the z-axis.

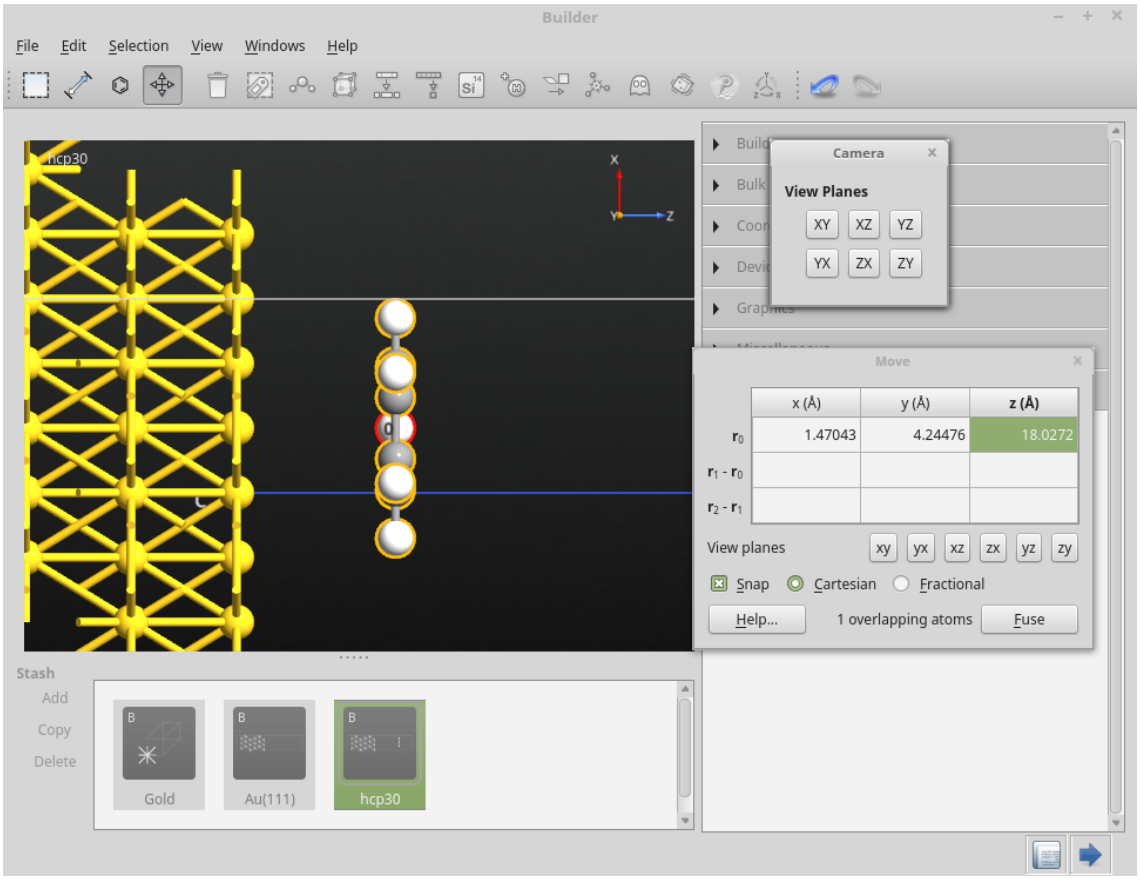
Click the icon to open the **Move** tool, and then click the extra snap-atom in the middle of the molecule to select it as an anchor atom. The selection shade should change color to red.



In the **Move** panel, make sure that the **Snap** option is selected. Use then the mouse to drag the molecule by the anchor atom, and drop it such that the anchor atom snaps onto a gold atom in the second slab layer.

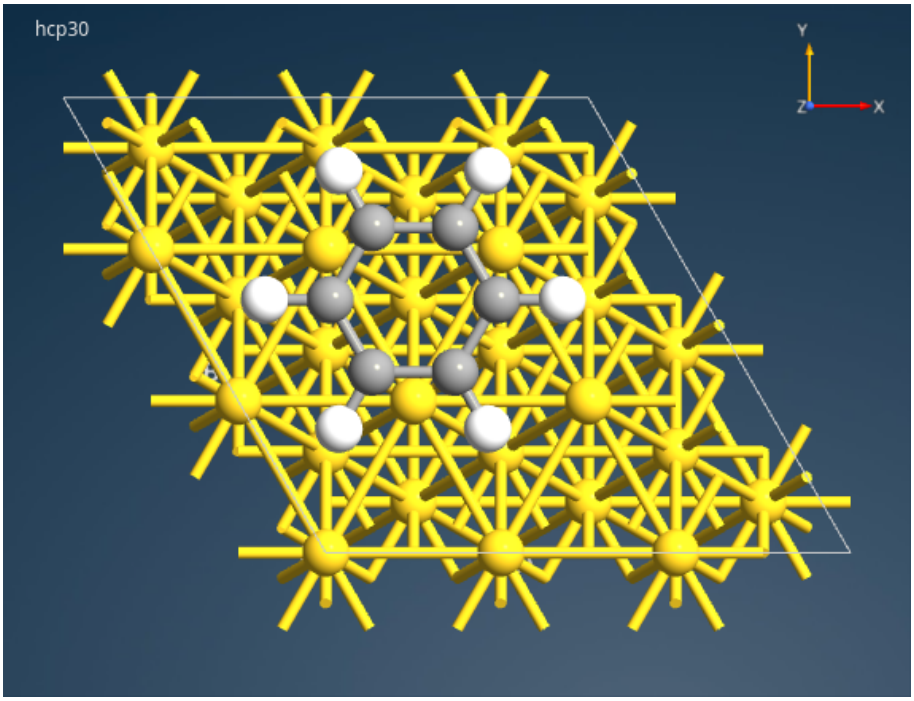


Finally, adjust the molecule–surface distance such that it matches the 3.62 Å reported in[Page 1287, 1]: In the **Move** panel, enter 18.0272 Å for the z-coordinate of the anchor atom position r_0, as shown in the figure below.



Note: In general, to set the desired molecule–surface distance, you of course need to first know the cartesian coordinate Z of the Au topmost layer. This can be done by simply hovering the pointer above one of the atoms of the Au(111) topmost layer.

Finally, remove the anchor atom by selecting it and deleting it. Your *hcp30* configuration should now look like this:



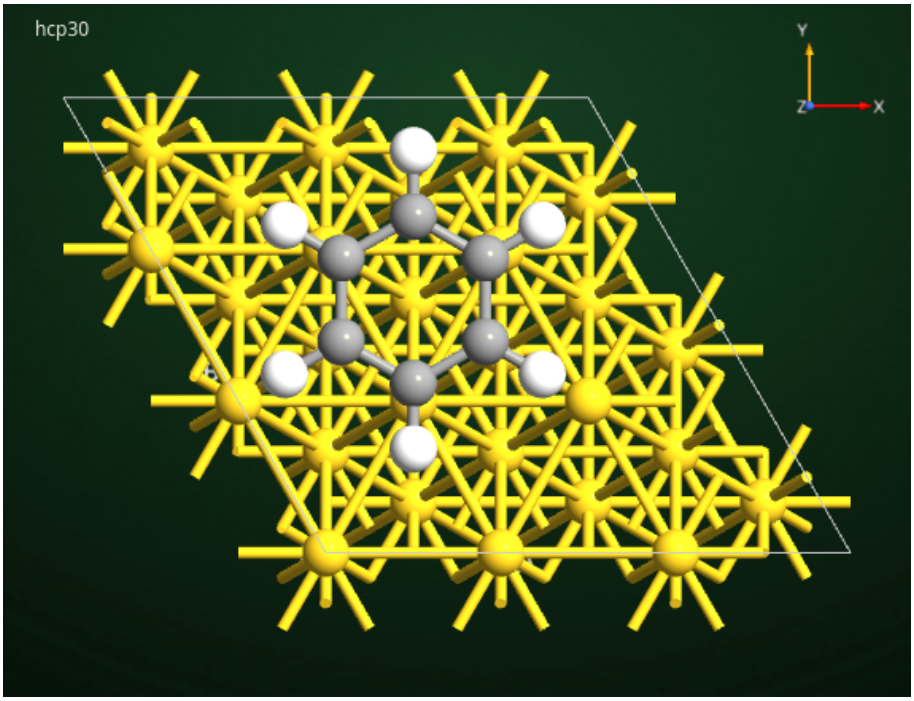
Tip: You may save the configuration as an ATKPython script, or **export** it to a range of popuar file formats.

Creating the hcp-0° configuration

The hcp-0° adsorption configuration is simply related to the hcp-30° one by a 30° rotation of the benzene adsorbate (assuming constant Bz–Au(111) distance).

First, **copy** the *hcp30* Stash item and **rename** the copy to *hcp0*. Then select all benzene atoms in the newly created copy, and open once again the *Coordinate Tools* → *Rotate* plugin.

Apply a rotation of 30° around the z-axis, which is normal to the surface plane. The *hcp0* configuration should now look like this:



References

3.9.2 Building a molecular junction

Note: The work flow described below will walk you through a rather broad overview of the various tools in the Builder. The individual operations are not specific for this surface and this molecule, but can be applied to any other molecule or similar structure. This is the idea of the Builder - by applying a specific sequence of generic operations, you can easily build even highly specialized structures.

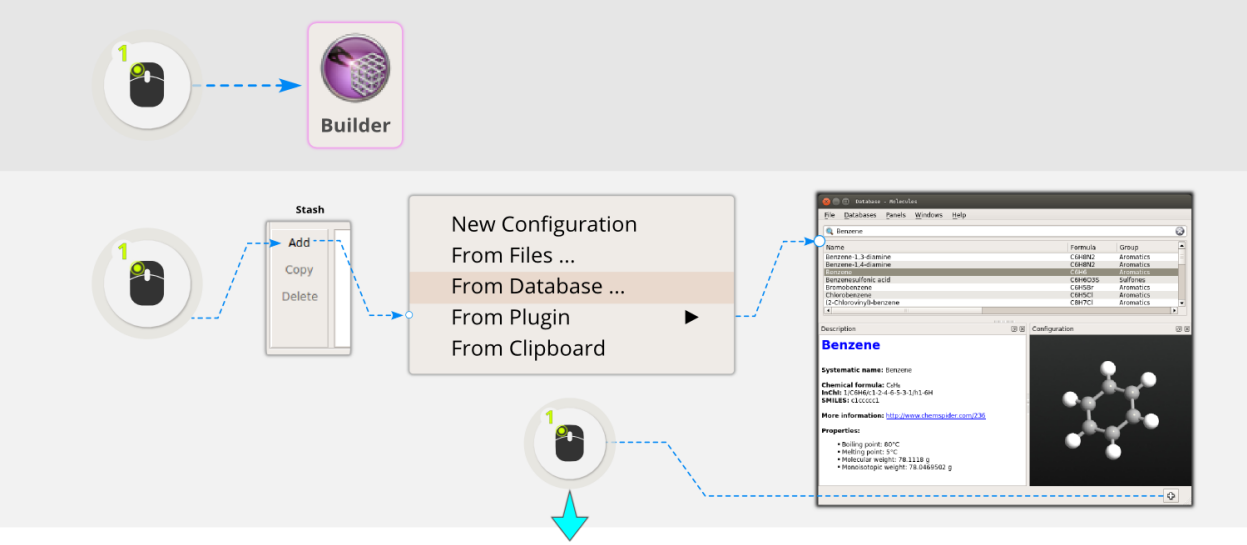
The recipe is quite simple and includes the following simple steps:

Construct the molecule. (Sometimes trivial, if you have it ready in the Database or as an external file; in our case we will modify benzene to make DTB.) Construct the surfaces, by cleaving the bulk crystal and creating a copy, to make two electrodes. Place the molecule on the surface of the left electrode, and connect the right electrode to the molecule. Finalize the device geometry.

The instructions below may appear a bit complicated, since each step contains many operations, but with a bit of training the whole process can be carried out in under 5 minutes. Also remember - if you make a mistake anywhere along the line, use Undo (Ctrl+Z)!

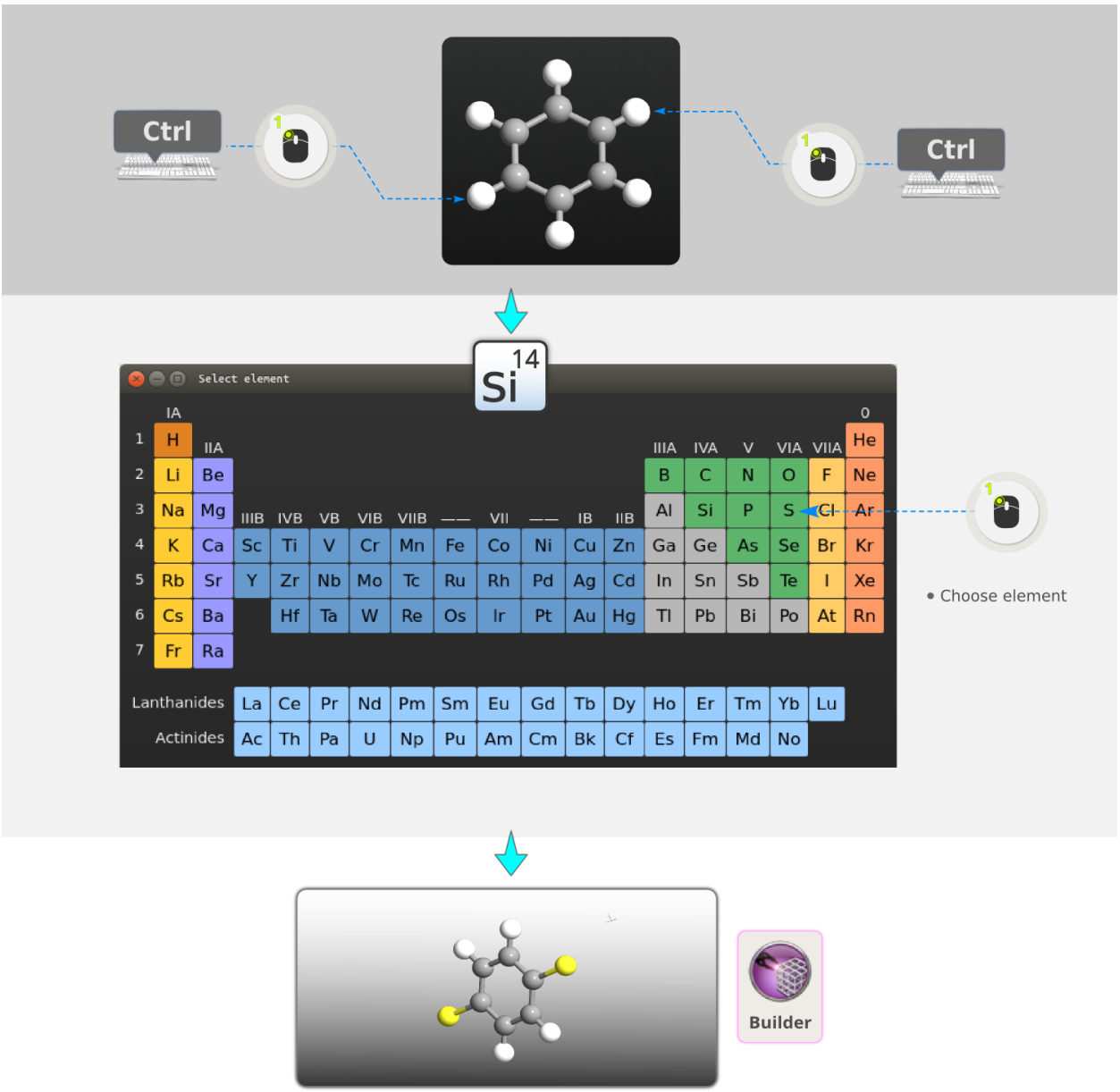
Benzene to DTB: Building the molecule

•**Step 1:** Click *Add* → *Add From Database* and from the menu select *Databases* → *Molecules*. Locate **benzene**, and click the **+** icon in the lower right-hand corner to add the molecule to the stash.



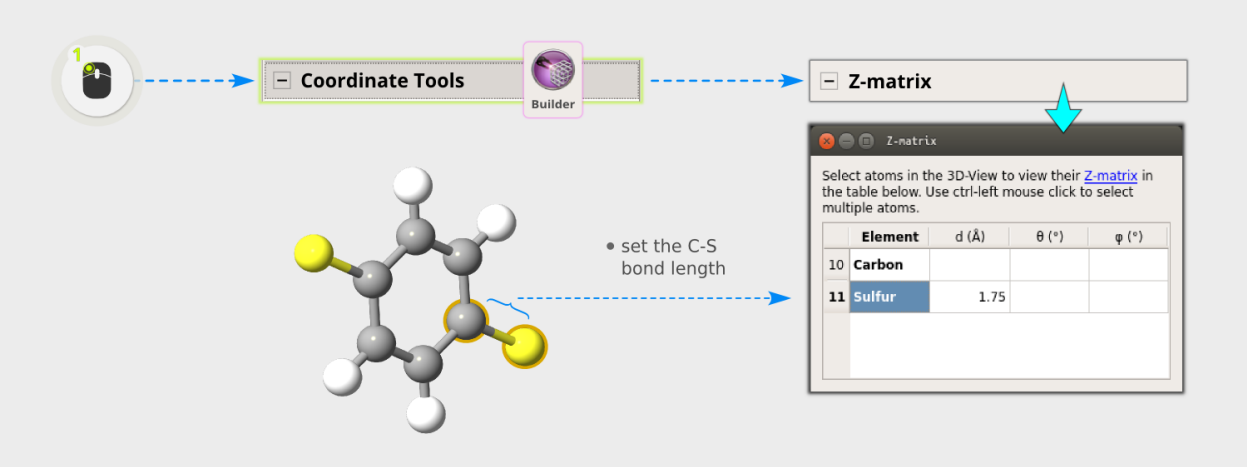
•**Step 2:** To convert the molecule to DTB, select two hydrogen atoms on opposite sides of the molecule by clicking them while holding down Ctrl. Then open the periodic table tool

, and click **Sulfur** (S) element:



•**Step 3:** Deselect all atoms by pressing Esc or clicking the background.

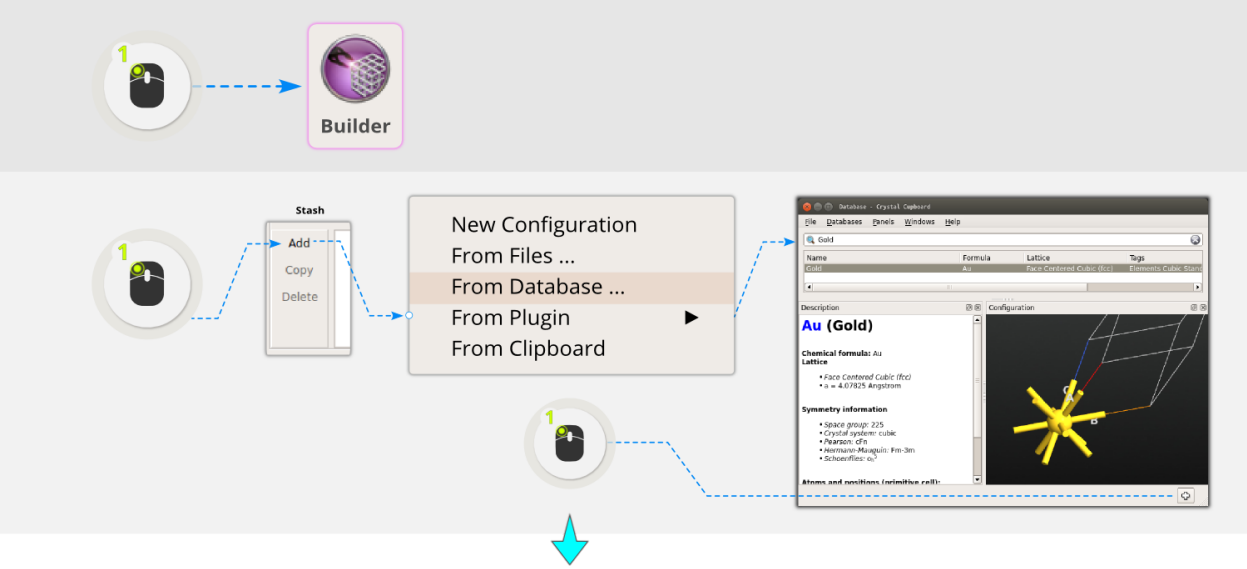
•**Step 4:** The C-S bond lengths need to be adjusted. Select one pair of bonded C-S atoms, taking care to **select the carbon atom first**. So, Open *Coordinate Tools* → *Z-matrix* and set the bond length to 1.75 Å. Note that the second selected atom moves, i.e. the S atom.



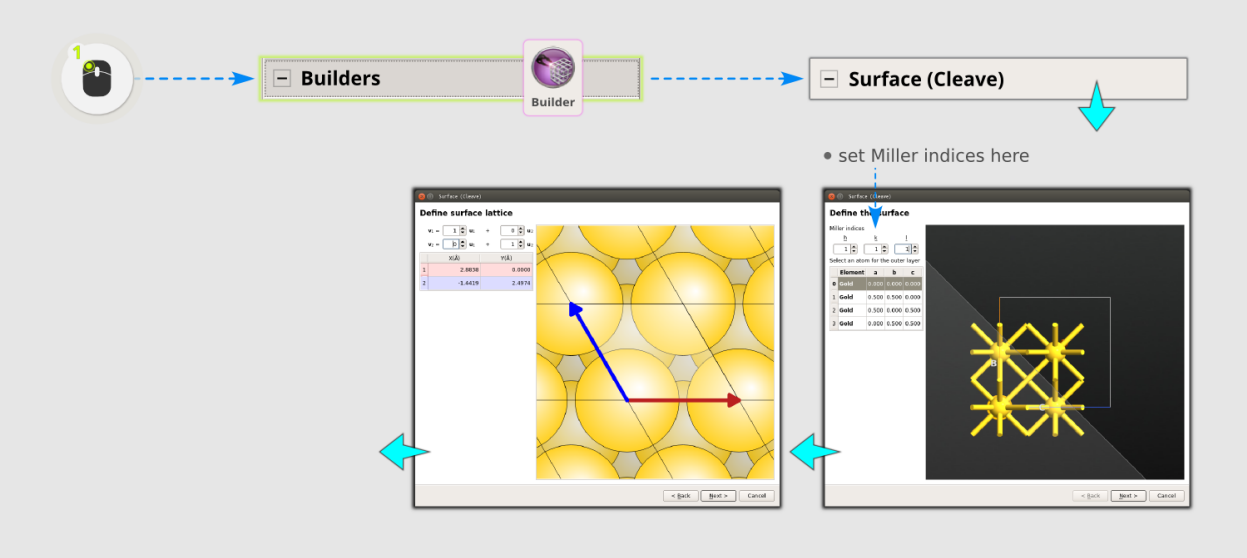
•**Step 5:** Do the same for the other Carbon-Sulfur bond!

Cleaving gold into two surfaces

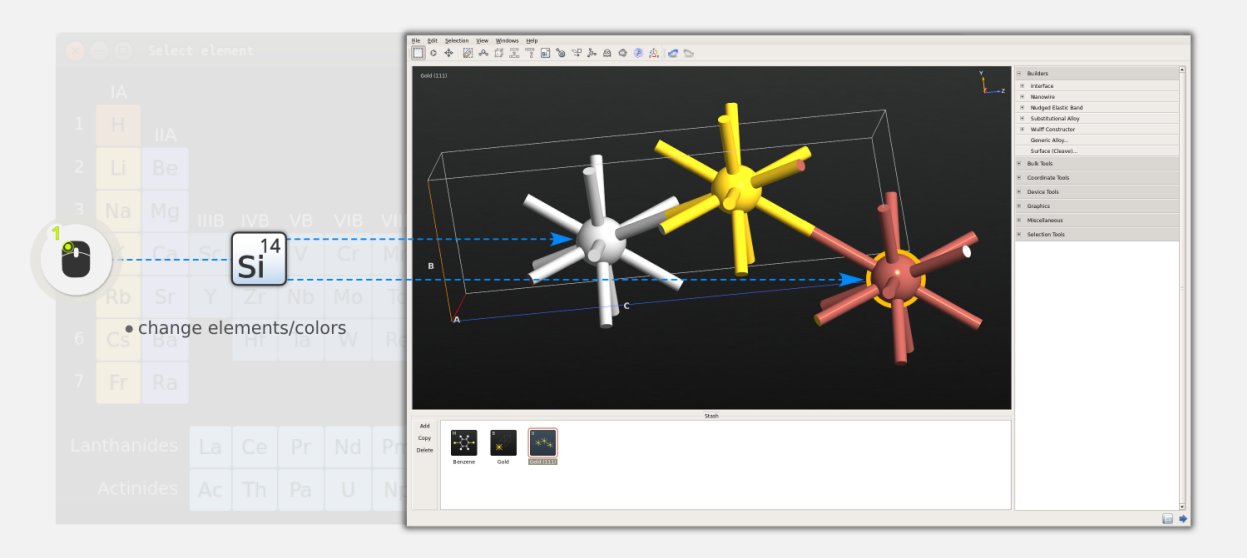
•**Step 6:** Click *Add* → *Add from Database* and this time select *Databases* → *Crystals* from the menu. Locate **Gold** and add it to the stash.



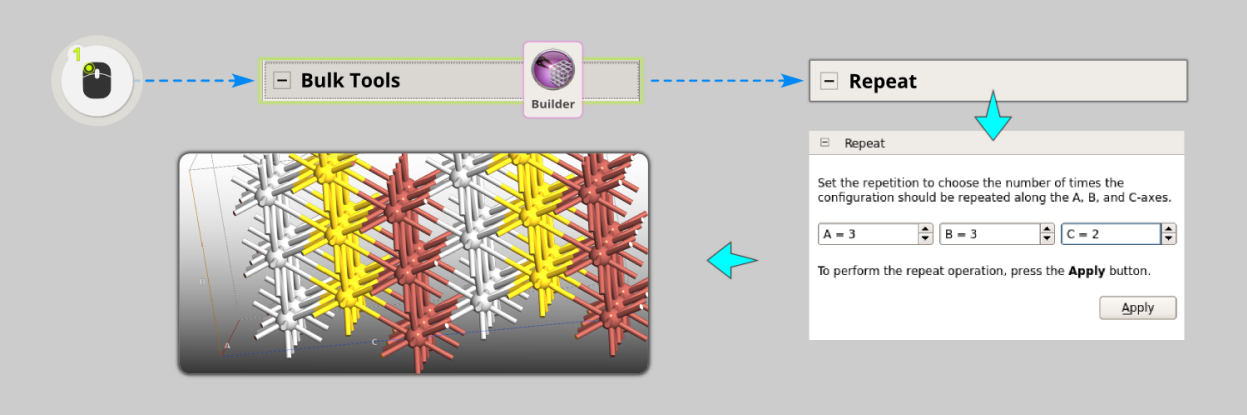
•**Step 7:** To cleave the gold crystal, open *Builders* → *Surface (Cleave)*. Enter Miller indices h=1, k=1, l=1 and click *Next* → *Next* → *Finish*.



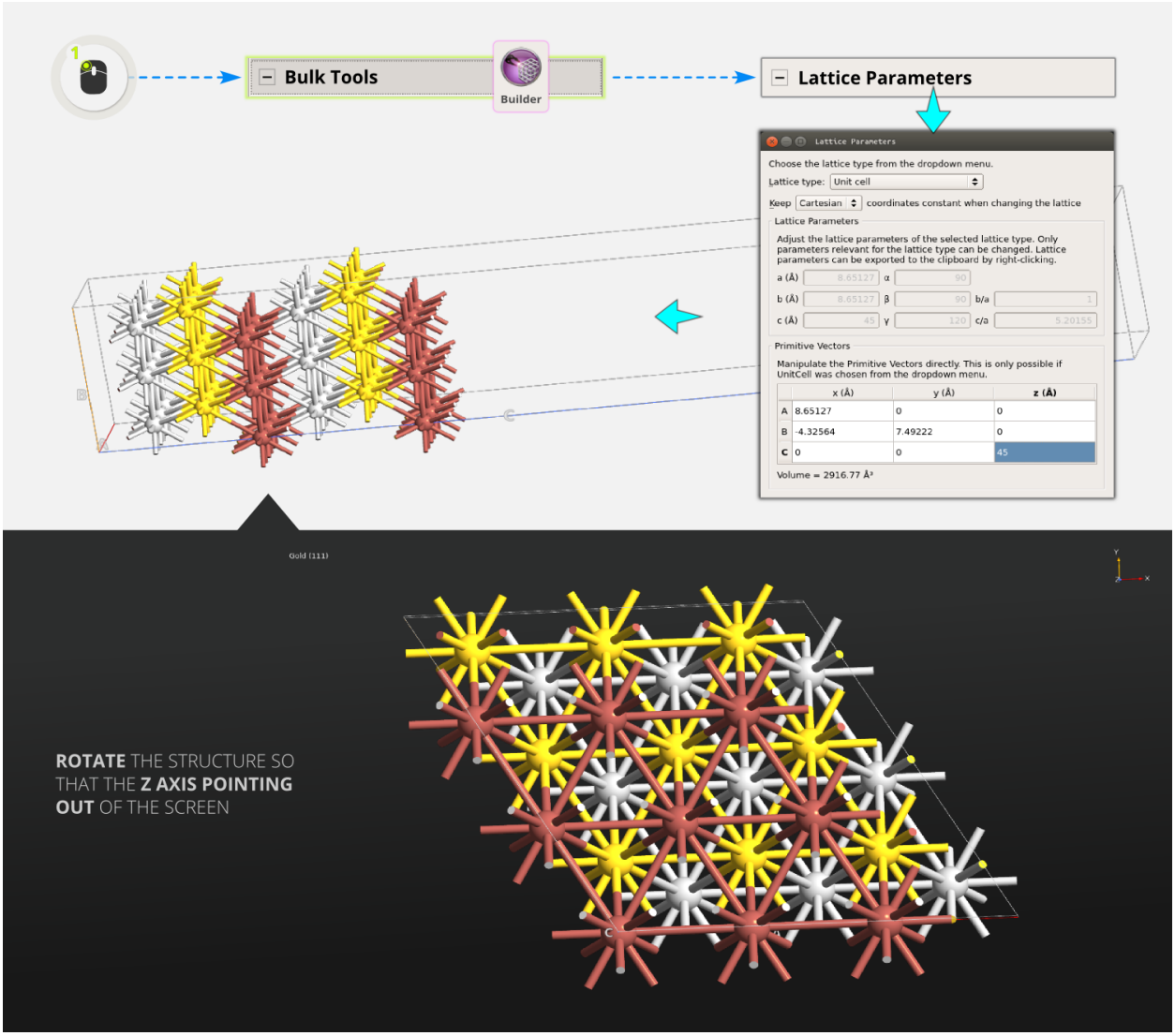
•**Step 8:** To make it easier to visually identify the three layers (ABC) in the fcc stacking sequence, change the gold atom in layer A to Ag and the one in layer C to Tl (these elements are just chosen to give nice colors).



•**Step 9:** To make a large enough surface cell for the molecule, repeat the structure 3x3x2 times using *Bulk Tools* → *Repeat*. The 2x repetition in the C direction is needed so that we later can extract the electrode from the central region.



•**Step 10:** You need to make some room for the molecule on the surface by adding vacuum. Open *Bulk Tools* → *Lattice Parameters* , select to keep Cartesian coordinates, and increase the C vector to 45 Å (still pointing along Z, as is necessary for a device).

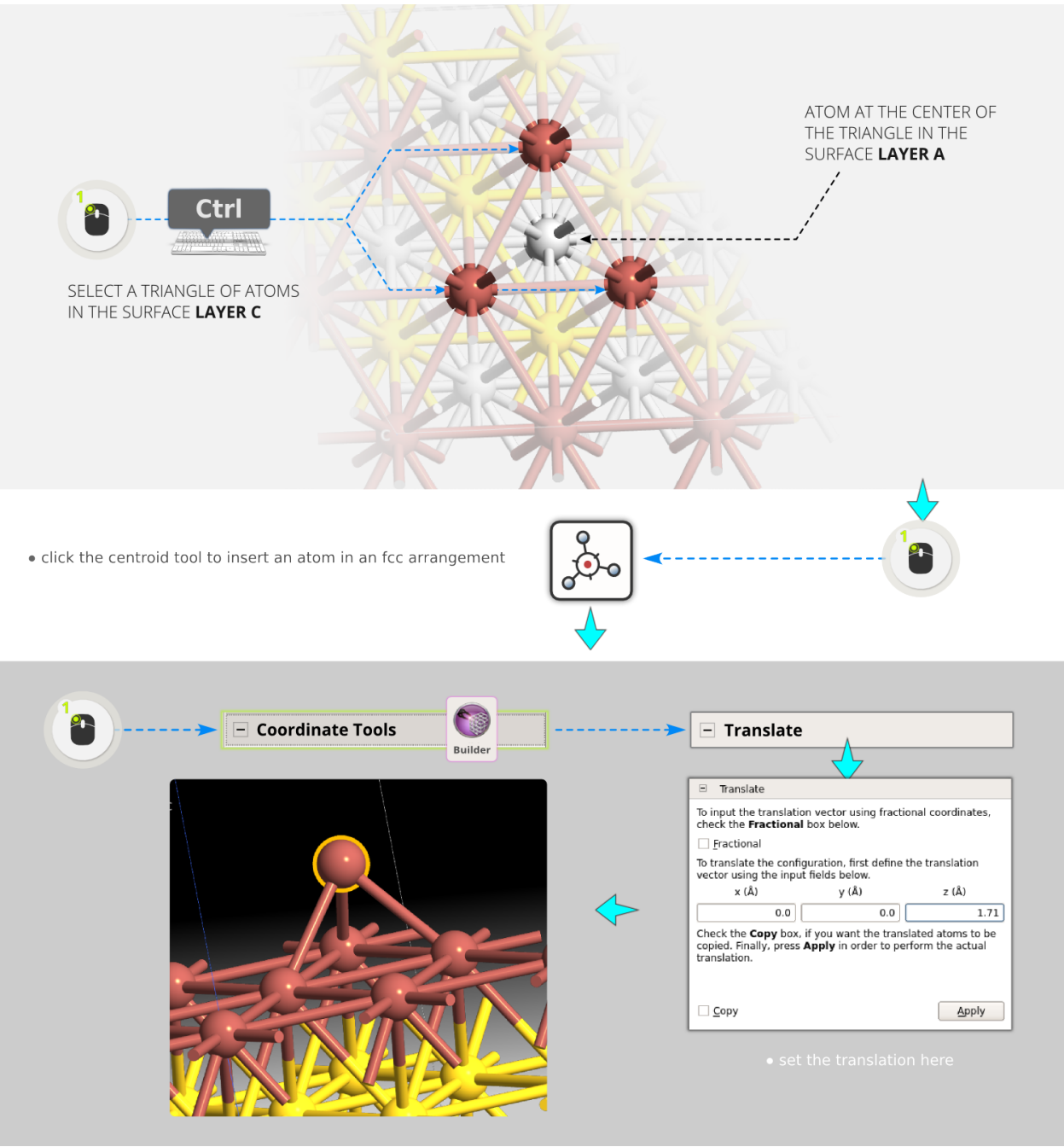


You will now create an alignment point on the surface where we can attach the molecule.

•**Step 11:** Select three atoms in a triangle in the surface layer C (the Tl atoms), such that the center of the triangle is above a silver atom in the A layer. Press the centroid tool to insert another atom in the average position of the three selected atoms. By choosing these particular atoms, the inserted atom sits in an fcc site, which is where we want to attach the molecule.

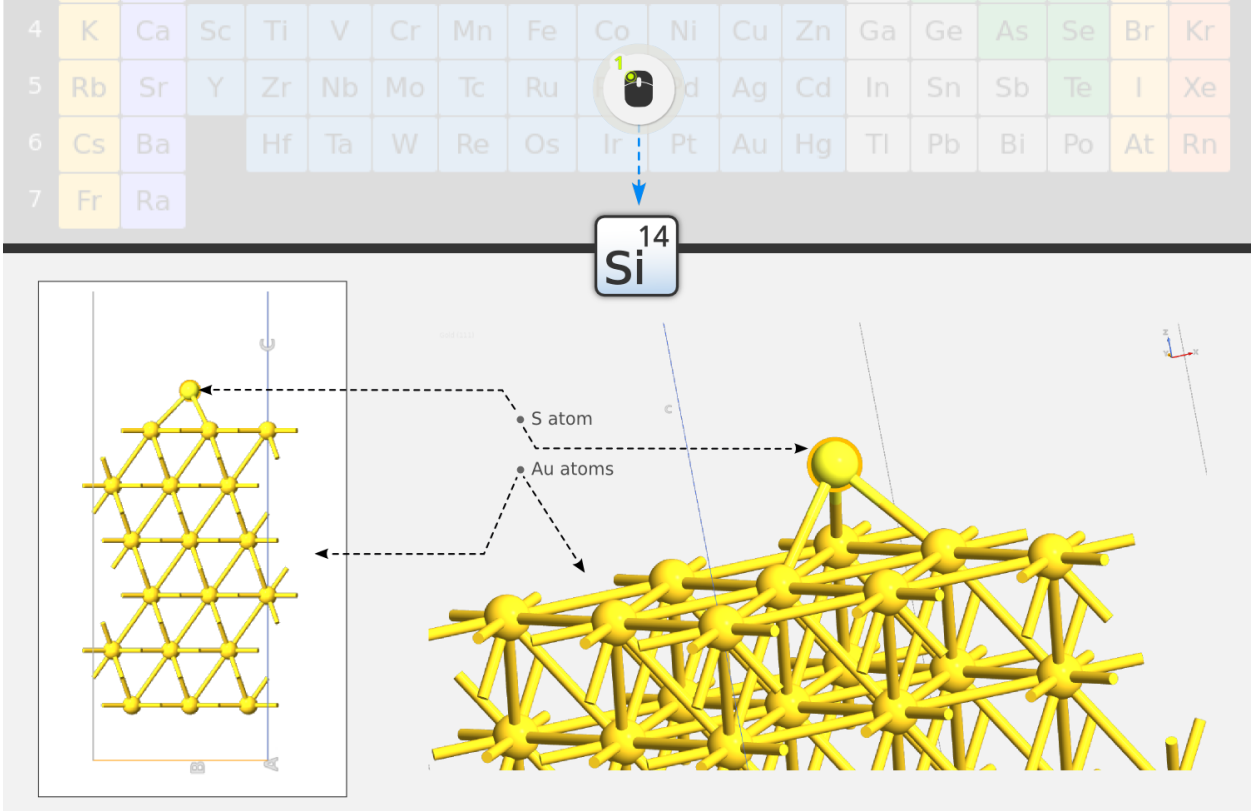
•**Step 12:** Use *Coordinate Tools* → *Translate* to move this atom 1.71 Å in the Z direction. (The value 1.71 Å gives an Au-S distance of 2.39 Å, taken from[1]).

[1] M.ăBrandbyge, J.-L. Mozos, P.ăOrdejon, J.ăTaylor, and K.ăStokbro. Density-functional method for nonequilibrium electron transport. *Phys. Rev. B*, 65:165401, Mar 2002. doi:10.1103/PhysRevB.65.165401.

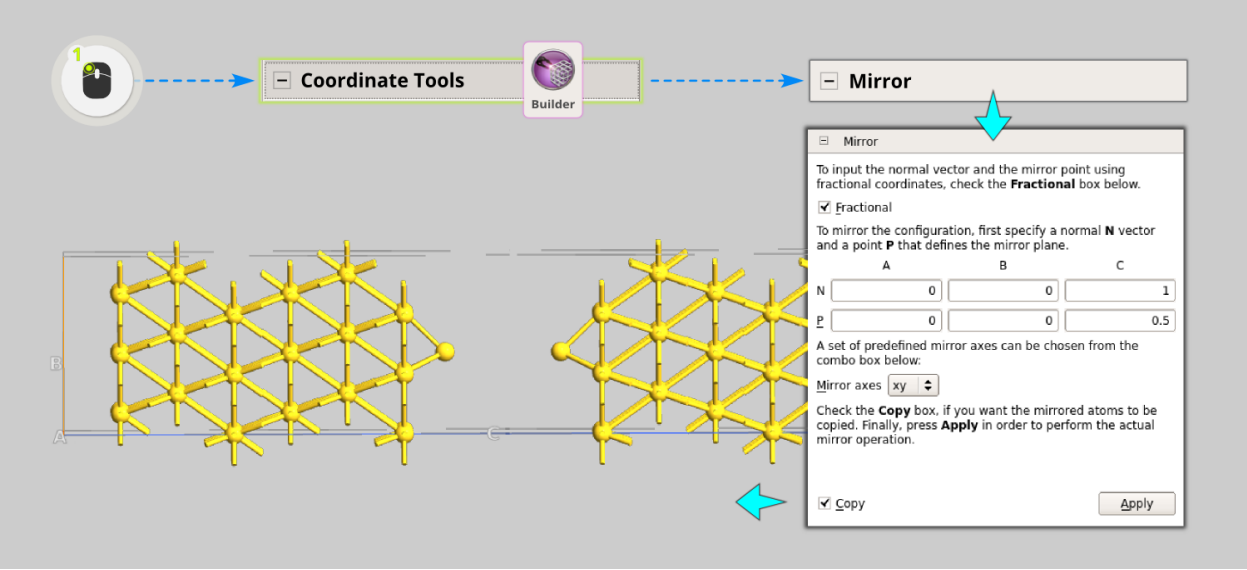


•**Step 13:** Press Ctrl+A to select all atoms, and switch them back to gold.

•**Step 14:** Also, to make it easier to handle the alignment atom later on, convert it to sulfur.



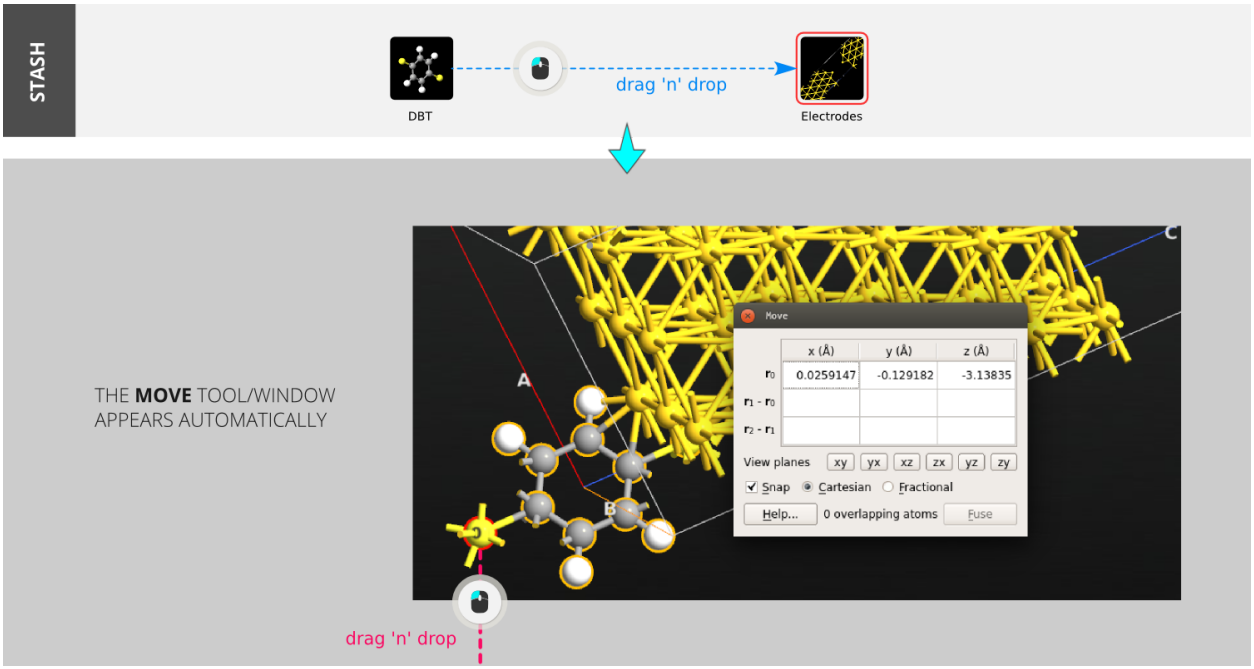
•**Step 15:** So far there is only one gold surface - there should be two. Therefore, press Ctrl+A again and open *Coordinate Tools* → *Mirror* . Select Fractional coordinates and set P=[0,0,0.5]. Tick Copy, then press Apply.



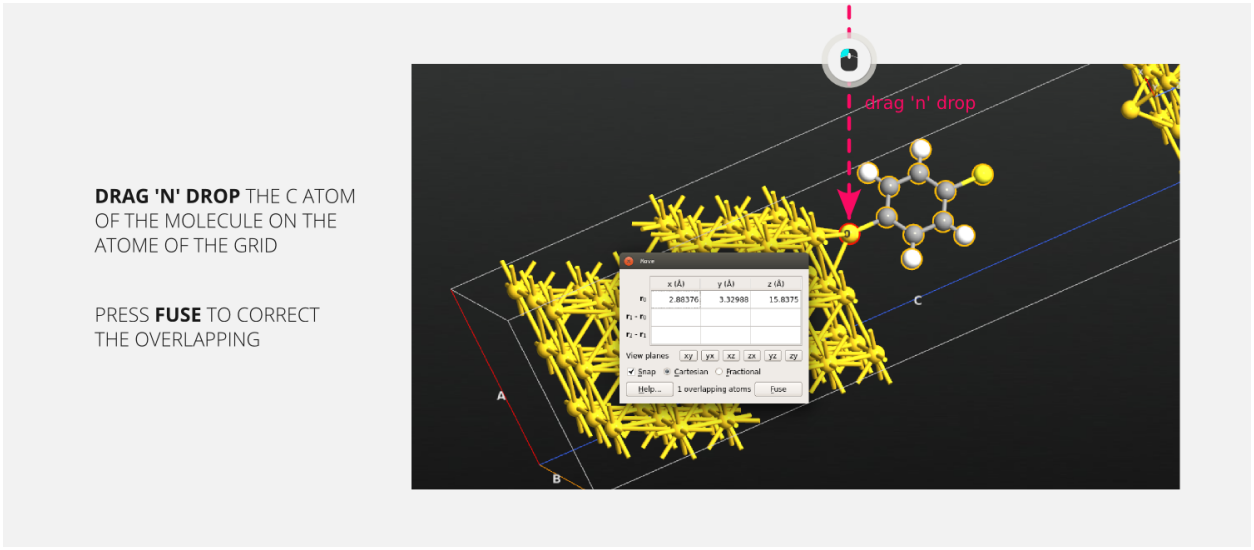
Combining the molecule and the surfaces

•**Step 16:** Add the benzene to the gold structure by dragging it from the stash onto the main window (where the gold surfaces are displayed). When the molecule is added, the Builder automatically activates **Move mode**.

Note: All atoms in the molecule are selected (indicated by the yellow circles around the atoms); in **Move mode**, you cannot change this selection, but you can appoint so-called **anchor atoms** which are used to translate and rotate the selection. **Anchor atoms will be indicated by red circles**.

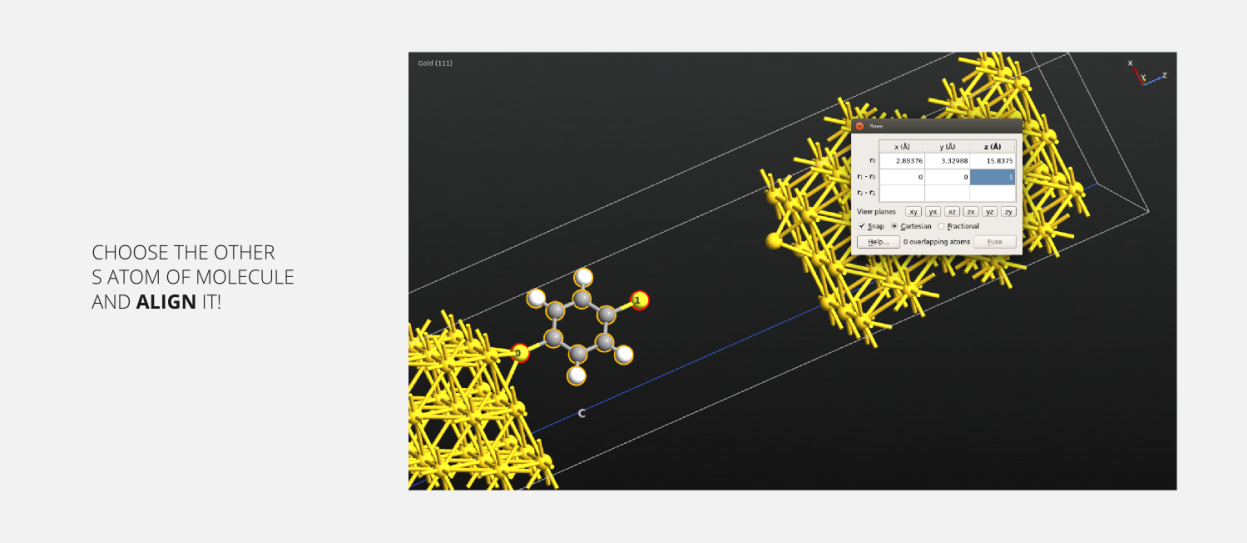


•**Step 17:** Click one of the sulfur atoms - it will become an anchor atom. Then drag it so that it ends up on top of the sulfur alignment atom on the left surface. Since Snap is on, it will latch on to the exact position of the alignment atom, indicated by the selection marker turning blue when the two atoms are on top of each other.



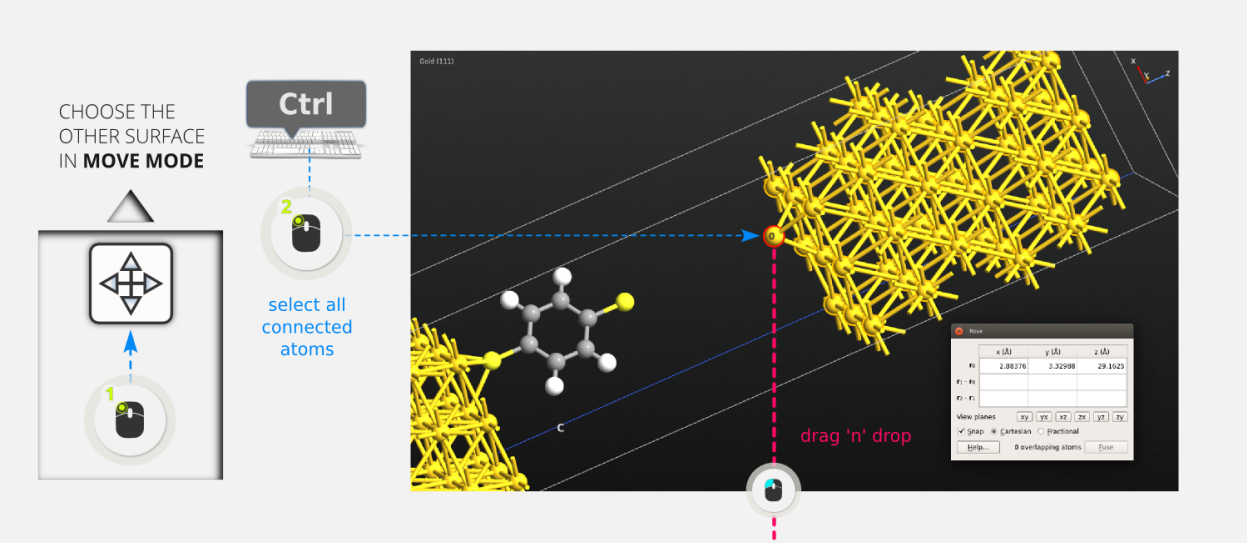
•**Step 18:** Press Fuse - this removes one of the two overlapping sulfur atoms (the one in the surface is removed; atoms in the active selection are always kept when fusing).

•**Step 19:** Now click the other S atom (the first one will remain selected). Then enter [0,0,1] on the **r1-r0** row of the Move tool widget. This aligns the two atoms - and hence the molecule along the Z axis. Close the Move tool widget.



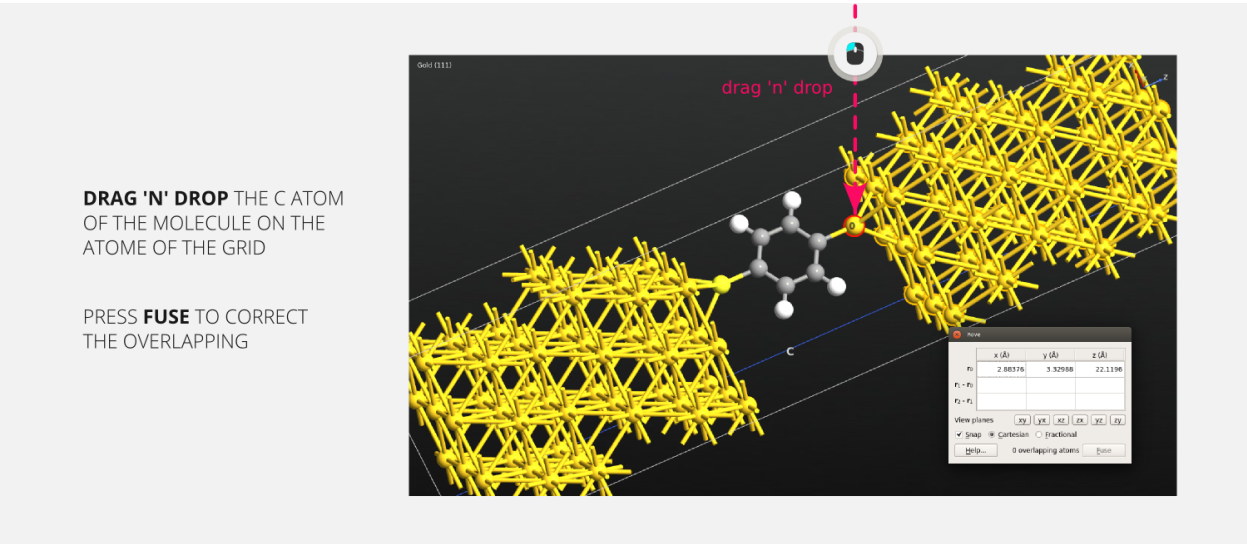
•**Step 20:** Select the second surface by holding down Ctrl while double-clicking one of the atoms in it. This selects all connected atoms.

•**Step 21:** Enter **Move mode** again and click the sulfur alignment atom in the second surface to make it an anchor atom.



•**Step 22:** Drag the right surface so that the anchor/alignment atom ends up on top of the sulfur atom on the free end of the DTB molecule.

•**Step 23:** Press **Fuse** to remove one of the two overlapping sulfur atoms (the one in the molecule, this time). Close the Move tool widget.



You now have a complete central region, and the only thing remaining is to convert it to a device configuration.

Converting the central region to a device configuration

•**Step 24** Open *Devices Tools → Device from Bulk*. The tool automatically figures out the size of the electrodes, since we repeated the gold (111) surface cell 2x in C.

•**Step 25:** This is however a very large central region, which will take a long time to calculate. Therefore, reduce the left and right surfaces to four layers by clicking twice for each surface (left and right). Press OK and you are done!

Note: In order to optimize the geometry of the device you can read the *Advanced device relaxation - manual workflow* tutorial.

References

3.9.3 Molecular Device

Version: 2016.3

This tutorial focuses on the calculation and analysis of a molecular device. The device consists of a dithiol benzene (DTB) molecule (sometimes also called benzene dithiol, BDT) in contact with two gold (111) surfaces. This is a classic molecular device, sometimes called the fruit fly of molecular electronics, and it has been studied in a large number of publications, see for instance[1][2] and references therein.

Some of the calculations in this tutorial are quite time-consuming; for a quicker introduction to quantum transport studies with **ATK**, see *Transport calculations with QuantumATK*. The example structure you will use is a molecular junction with a benzene ring between two gold electrodes. You can find the instruction of how to build it in Molecular Junction.

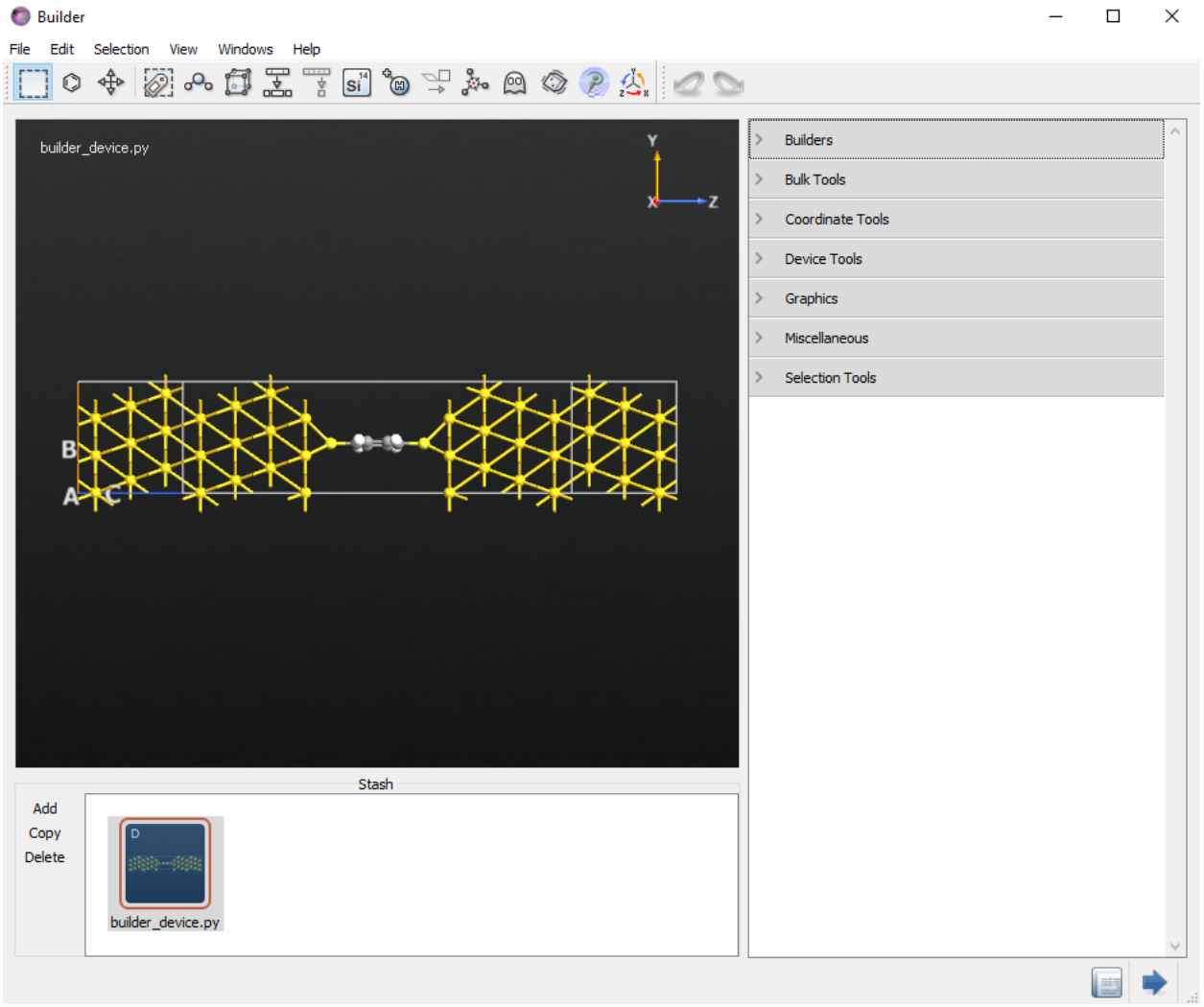
[1] K.ăStokbro, J.ăTaylor, M.ăBrandbyge, J.-L. Mozos, and P.ăOrdejón. Theoretical study of the nonlinear conductance of Di-thiol benzene coupled to Au(111) surfaces via thiol and thiolate bonds. *Computational Materials Science*, 27(1-2):151–160, 2003. doi:10.1016/S0927-0256(02)00439-1.

[2] K.S. Thygesen and K.W. Jacobsen. Molecular transport calculations with Wannier functions. *Chemical Physics*, 319(1-3):111–125, 2005. doi:10.1016/j.chemphys.2005.05.032.

The DFT model in QuantumATK is the underlying calculation engine used in this tutorial. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK reference manual.

Zero-bias calculation

Following the instruction of Molecular Junction, or download the script `builder_device.py`, you will get the following structure.

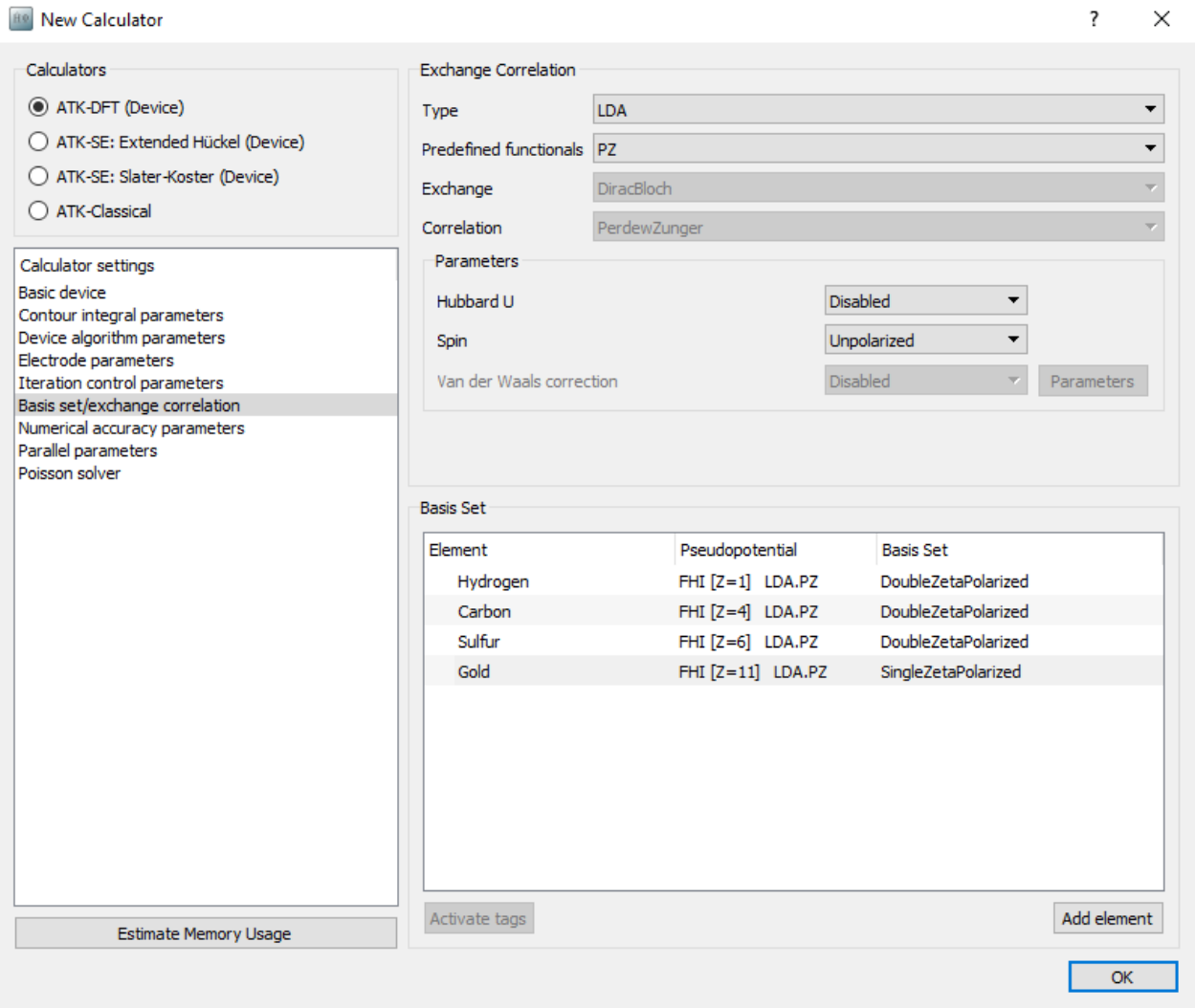


Attention: The geometry was not relaxed. This is not critical for the concepts we will discuss in this tutorial, but it is an important step to consider for any real calculations with **ATK**. To learn how to relax a device geometry, see the *Relaxation of devices using the OptimizeDeviceConfiguration study object*.

Setting up the calculation with the Script Generator

Now send the geometry to the **Script Generator** by pressing the **Send to** button at the lower right-hand corner of the **Builder** and select **Script Generator** from the pop-up menu.

- In the Script Generator, add three blocks:
  - a **New Calculator**
  - an *Analysis* → *DeviceDensityOfStates*
  - an *Analysis* → *TransmissionSpectrum*
- Change the default output filename to `au-dtb-au.hdf5`.
- Open the **New Calculator** block and change the following settings:
  - Set the k-point sampling to 3 x 3 x 50 points.
  - In *Basis set/exchange correlation*, reduce the basis set to *SingleZetaPolarized* for Gold to gain some speed.



- Open the **DeviceDensityOfStates** block by double-clicking it and change the following settings:
  - set the k-point sampling to 3 x 3 points
  - set the energy range to (-3, 3) eV
  - set points from 101 to 401
- Open the **TransmissionSpectrum** block and make the same modifications, i.e.
  - set the k-point sampling to 3 x 3 points.
  - set the energy range to (-3, 3) eV.
  - set points from 101 to 401
- Finally, save the calculation script to the file `au_dtb_au.py` in your working directory for future reference. You can also download the file `au-dtb-au.py` for comparison.

To run the script on your local machine, send it to the **Job Manager** by pressing the **Send to** button at the lower right-hand corner of the **Script Generator** and select **Job Manager** from the pop-up menu. In the **Job Manager**, press **Run Queue** to launch the job. On a modern, single-node computer the script should take around an hour to execute.

Note: The calculation speed can be increased greatly by using a parallel computer. For information about how to run jobs in parallel, see the *Parallelization of QuantumATK calculations*.

Analysis of the zero-bias results

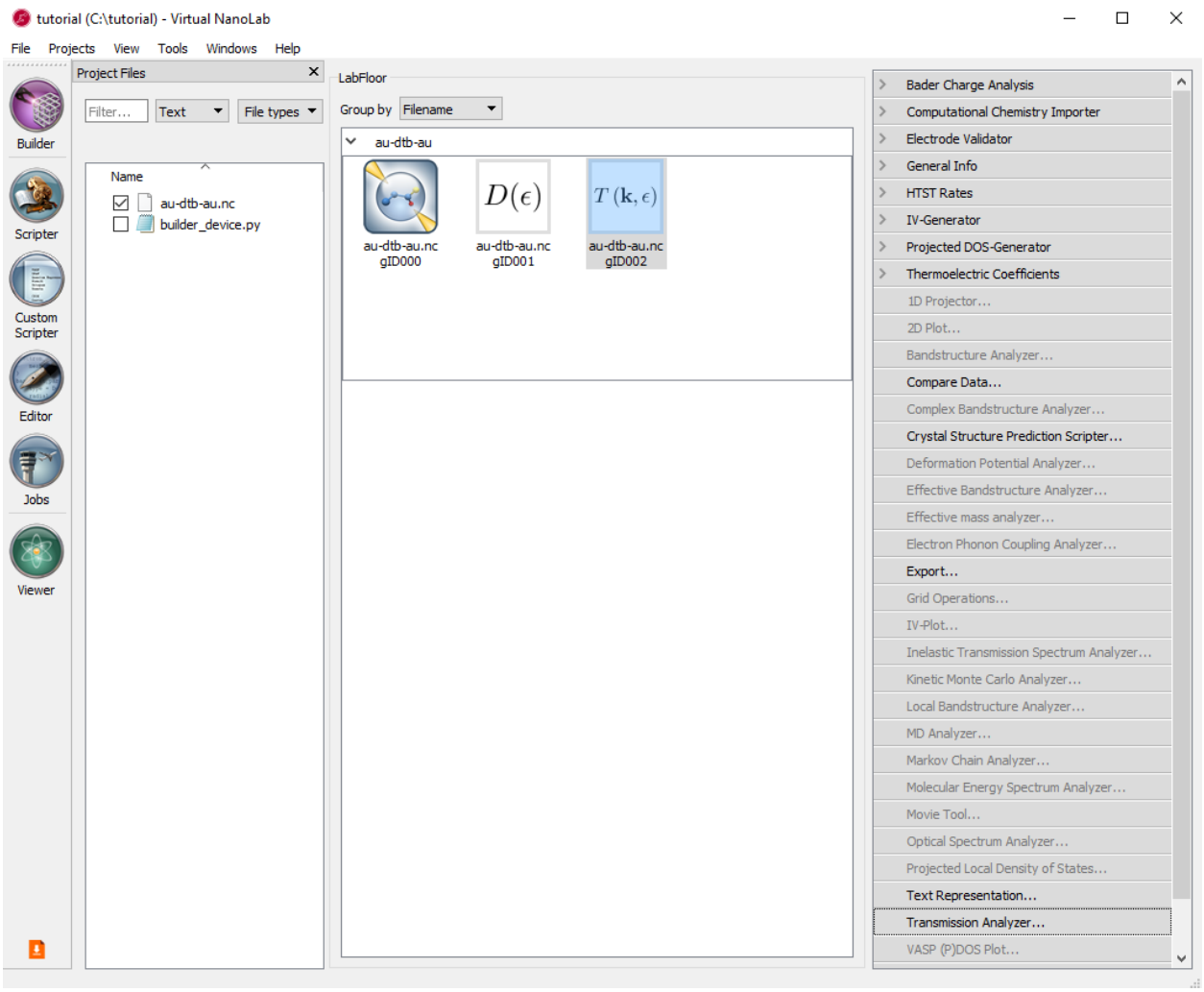
In this chapter you will extract data from the zero-bias calculation. You will do the following analyses:

- Compare the transmission spectrum with the Partial Device Density Of States (PDDOS) of the phenylene ring.
- Calculate the Molecular Projected Self-Consistent Hamiltonian (MPSH), and rationalize the transmission in terms of the MPSH states.
- Calculate the transmission eigenstates.
- Calculate the energy-dependent Local Density of States (LDOS), in order to obtain a spatial view of the energy levels.

The transmission spectrum and density of states

Return to the main **QuantumATK** window. The `au-dtb-au.hdf5` file should now be visible under **Project Files**, and its contents displayed on the **LabFloor**. It contains three objects:

- **DeviceConfiguration** contains the two-probe device configuration and the self-consistent state of the attached calculator.
- $T(\mathbf{k}, \epsilon)$ **TransmissionSpectrum** contains all information about the computed transmission spectrum, including settings and results.
- $D(\epsilon)$ **DeviceDensityOfState** contains all informations about the Density state of the device region.



Select the TransmissionSpectrum object, and click the **Transmission Analyzer** in the right-hand panel bar.

The Transmission Analyzer plugin opens. On the left, named **Spectrum**, is shown the k-point averaged transmission spectrum as function of energy. The average Fermi energy of the two electrodes is set to zero and indicated with a dashed line.

The transmission and energy values in the point marked by the red dot are shown below the plot. You can use the mouse to move the dot along the $T(E)$ curve and read off the values. The *Curves* drop-down menu lets you select the spin components to be plotted. Note that the transmission for spin types X, Y, and Z are only non-zero for calculations with noncollinear spin.

At any particular energy (defined by the red dot), the total transmission is computed as an average over the transmission coefficients in the sampled k-points. The right-hand plot, named **Coefficients**, shows an interpolated contour plot of the transmission coefficients vs. reciprocal vectors k_A and k_B. For example, at the energy, E = -1.4 eV, where you find a spectrum peak at the transmission, the contribution is minimum around the Γ point, as shown in the figure below.

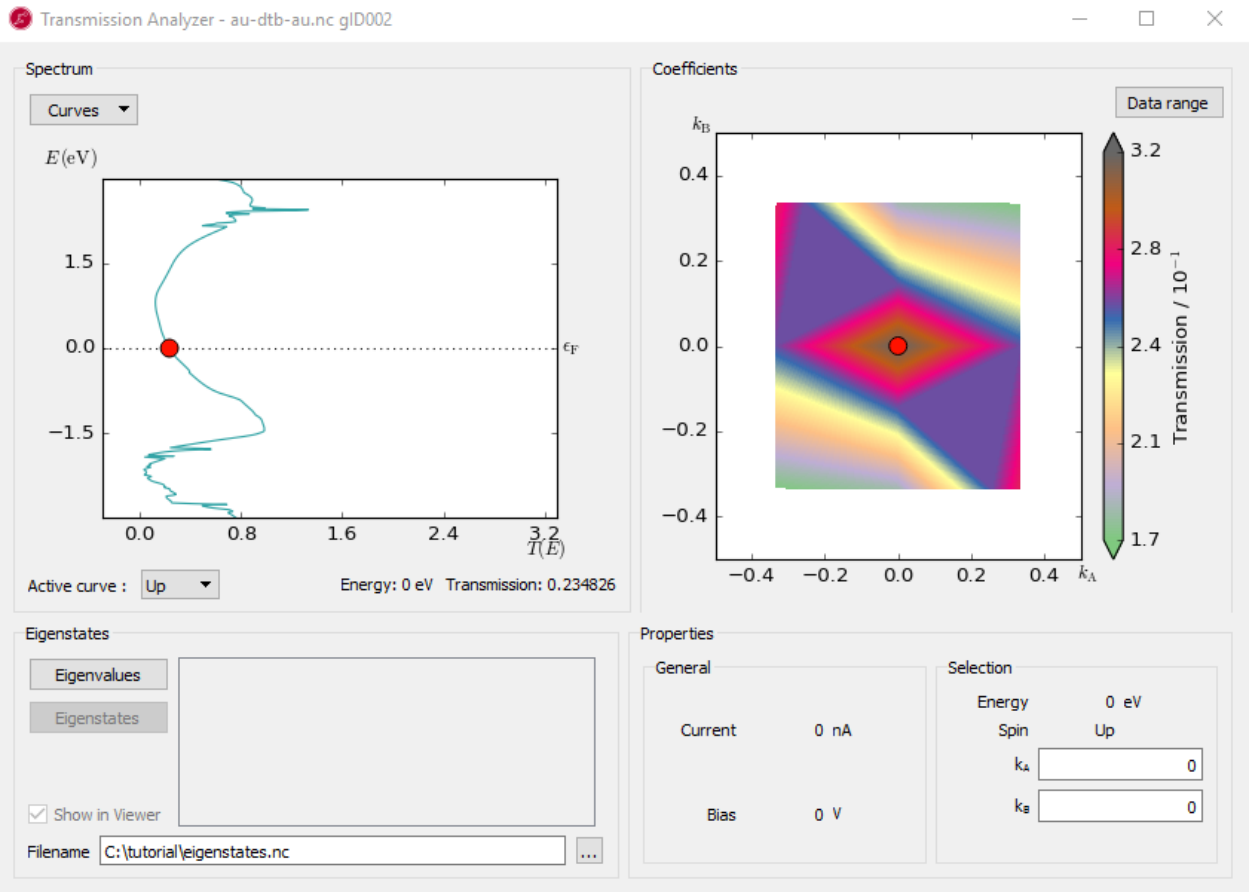
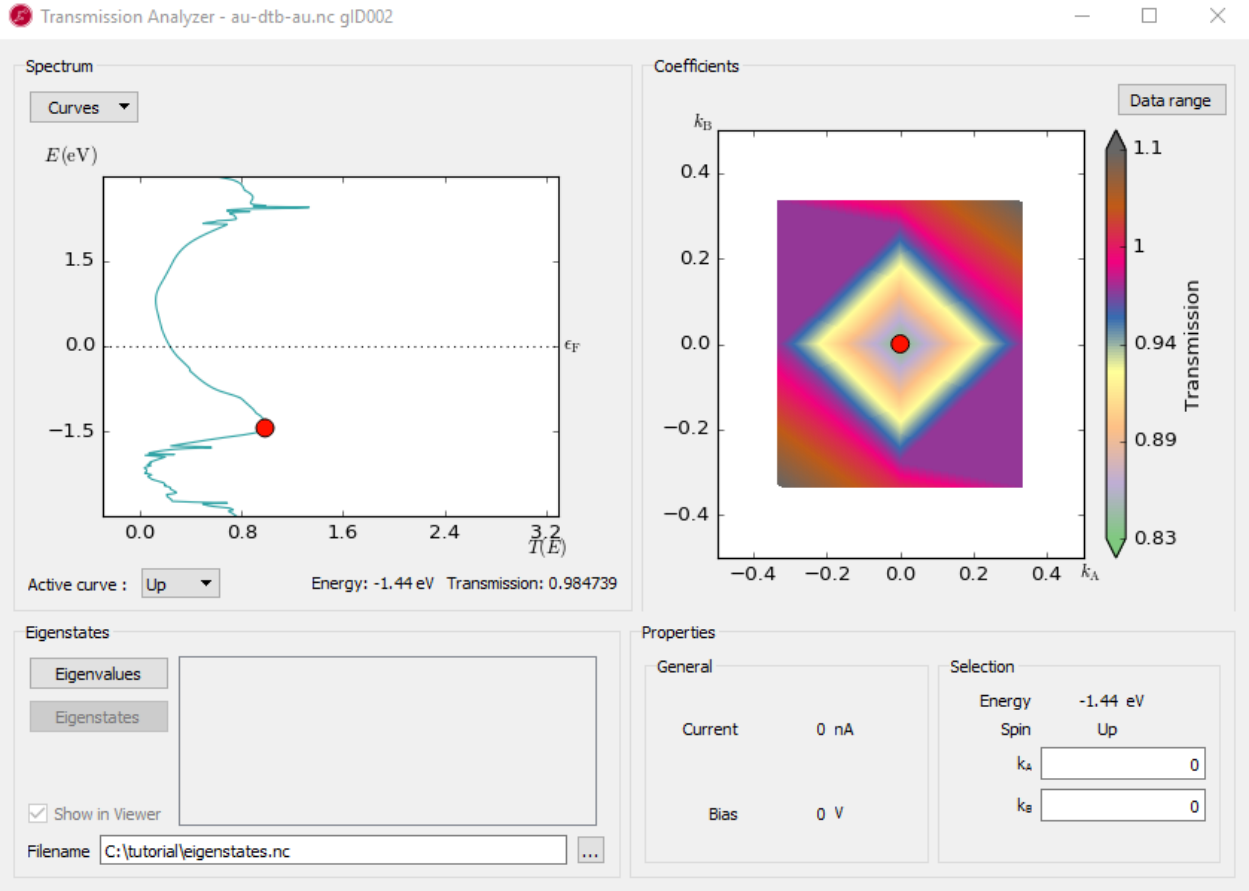
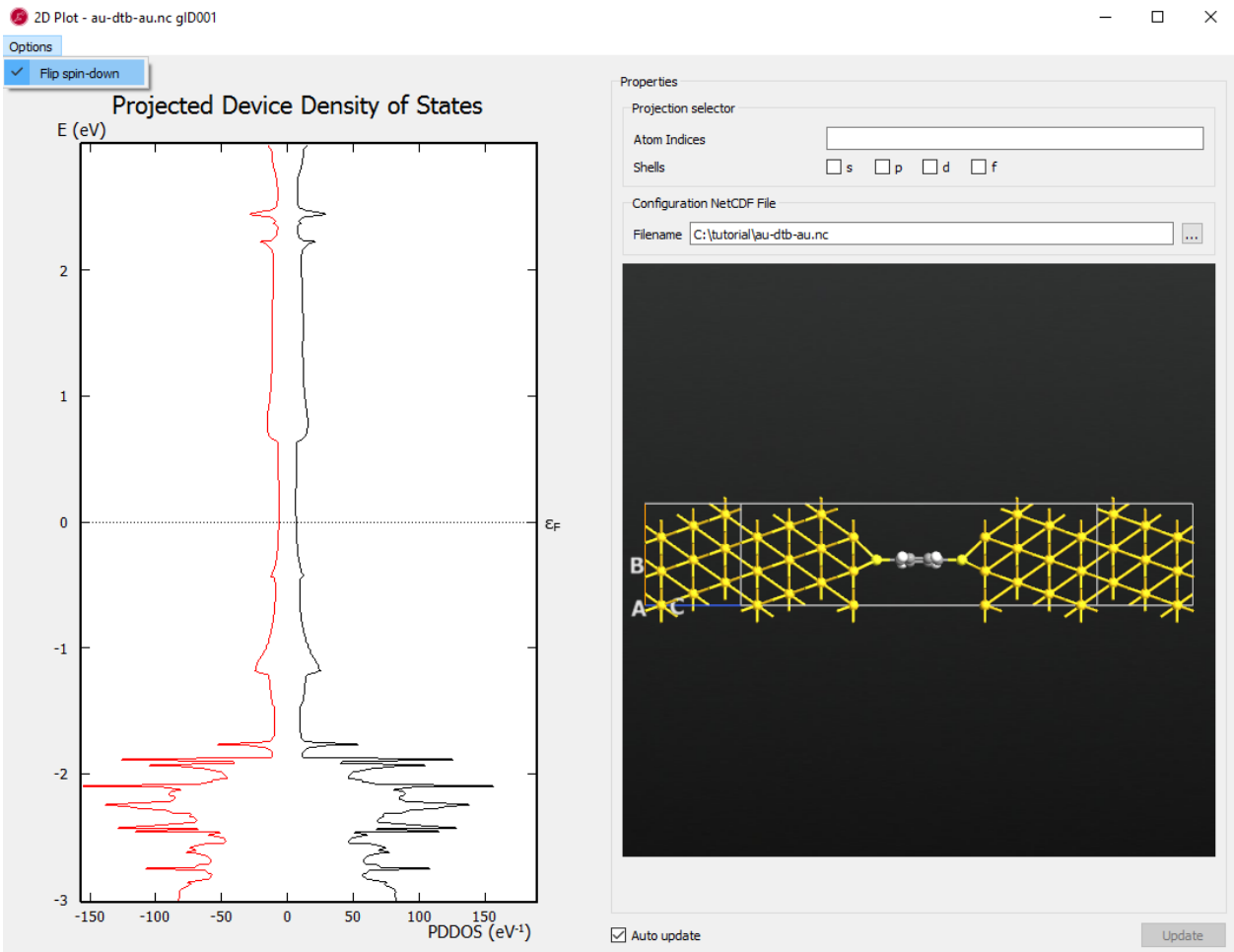


Fig. 3.94: The k-point averaged transmission spectrum as function of energy

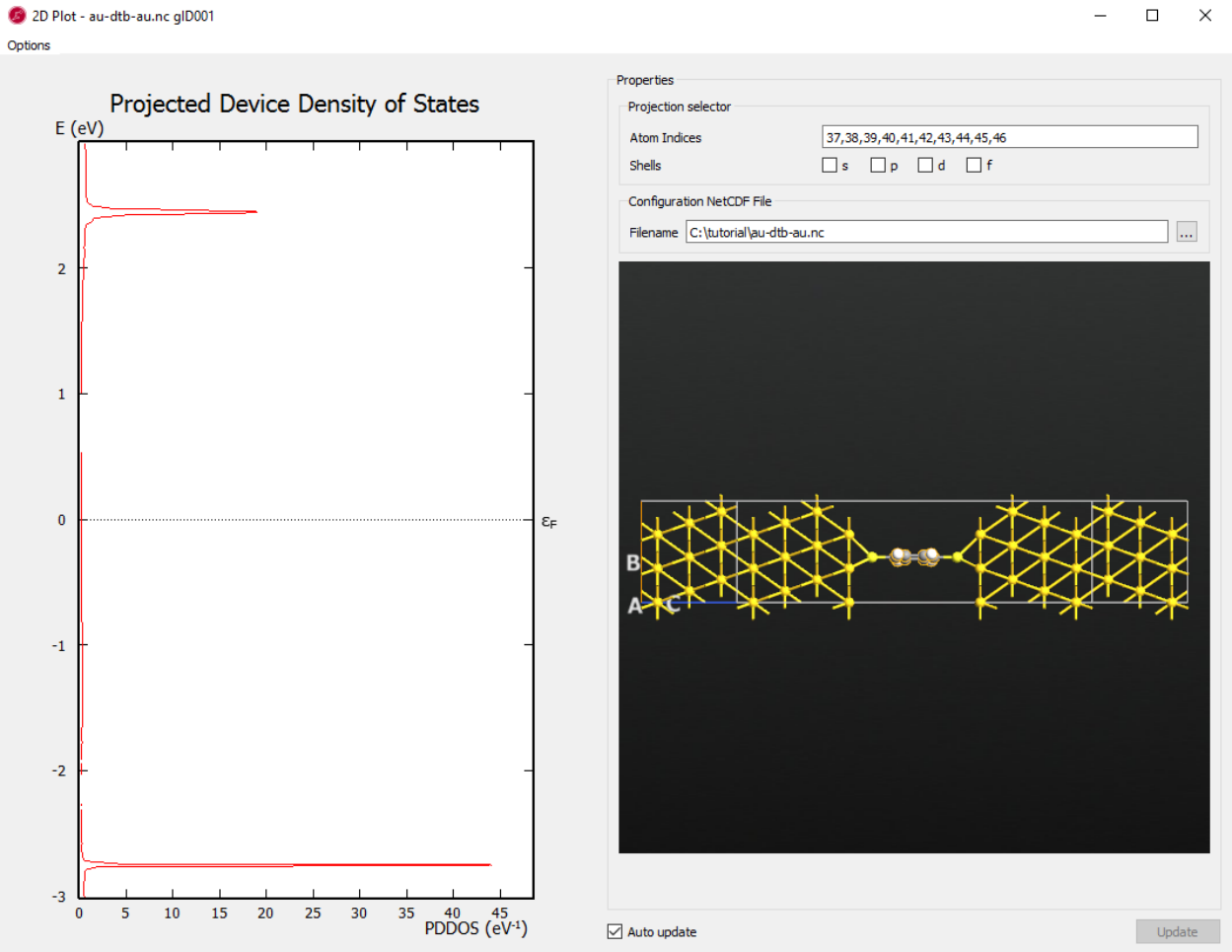


There are three broad peaks, at around -3, -1.4 and 2.5 eV. Also note the narrow peak around 2.4 eV which rises above T(E)=1. In the following you will investigate the origin of these peaks.

Next select the **DeviceDensityOfStates** object $D(\epsilon)$ and click the *Show 2D Plot* button to open the PDDOS analyzer window.To merge both PDDOS lines into one, click the **Options** button on the upper left corner and turning off *flipping*.



Select all carbon and hydrogen atoms by enclosing them with a rectangle using left mouse button (or select the individual atoms with Ctrl left-click).



Tip: Copy the *Atom Indices* here as you will need to input the indices in the next step.

Note: The similarity in the peak structure of the PDDOS and the transmission spectrum indicates a clear correspondence between the energy levels on the phenylene ring and the transmission spectrum.

The Molecular Projected Self-Consistent Hamiltonian (MPSH)

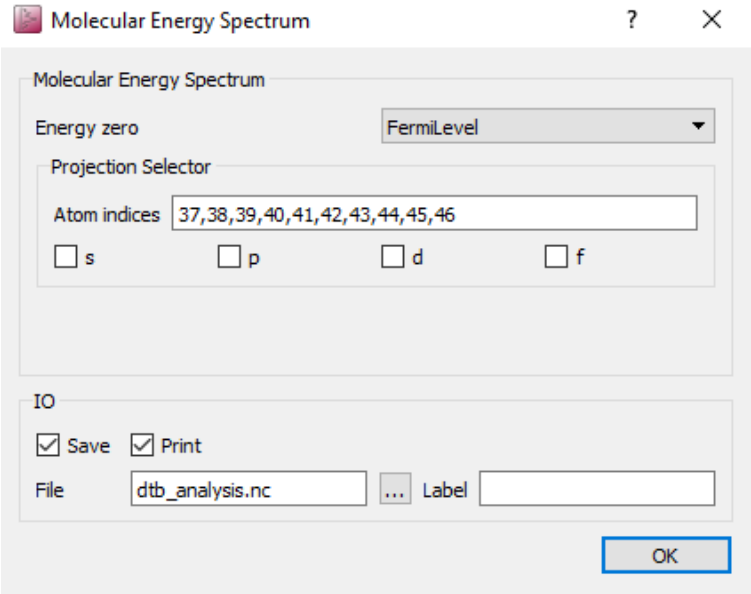
To understand the energy levels of the phenylene ring in the presence of the surrounding electrodes, you will now calculated the MPSH states. The MPSH states are obtained by diagonalizing the molecular part of the full self-consistent Hamiltonian[Page 1313, 1].

Open the **Script Generator** tool and

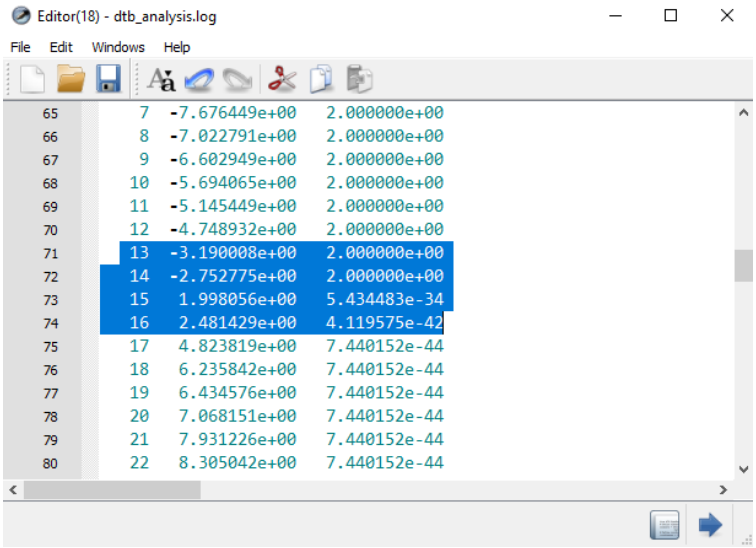
- Double-click **Analysis from File** to add the block .
- Select the file `au-dtb-au.hdf5`, then select *DeviceConfiguration object (gID000)*.
- Add *Analysis* → *MolecularEnergySpectrum*.

•Change the output filename to `dtb_analysis.hdf5`.

•Open the **MolecularEnergySpectrum** block and input the indices of carbon and hydrogen atoms to *Atom indices* (37,38,39,40,41,42,43,44,45,46).



•Send the script to the **Job Manager** and execute it (the execution only takes a few seconds). The MPSH energies are then printed in the log file.



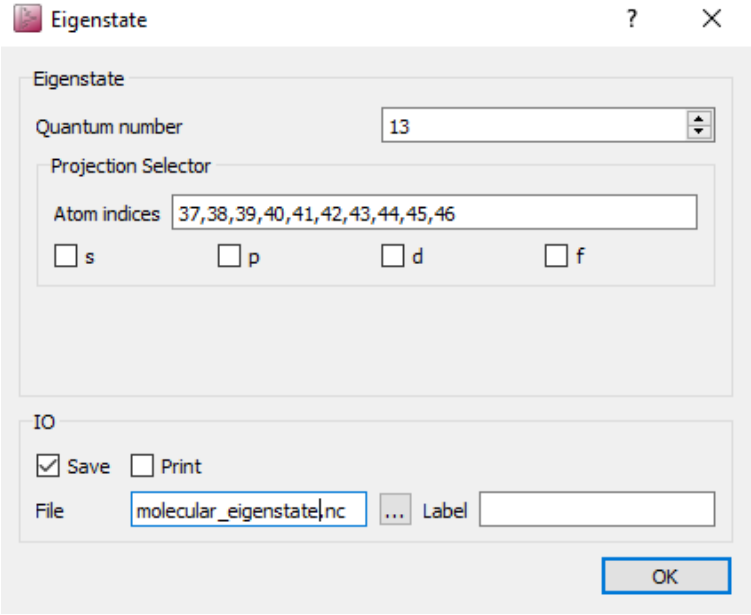
To calculate the eigenstates of the MPSH levels, reopen the last **Script Generator** (it is available from the windows menu at the top of each **QuantumATK** window, if you didnt close it).

•Change the name of the output file to `molecular_eigenstate.hdf5`.

•Add four *Analysis* → *EigenState* blocks.

•Open each EigenState script block and select projection on carbon and hydrogen Set the Quantum number to 13, 14,

15, 16 in each respective block.



•You may delete the **MolecularEnergySpectrum** block , it is no longer needed (select it and press the Delete key).

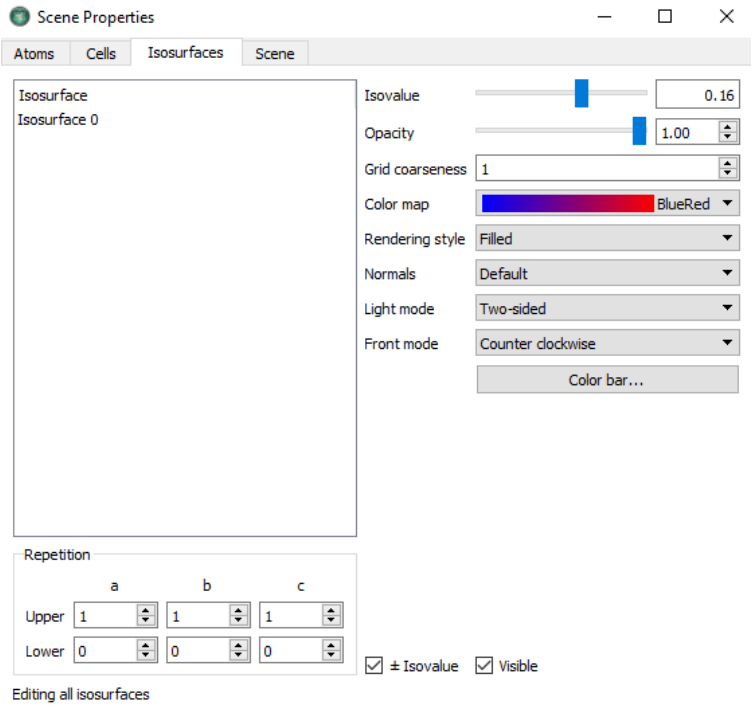
•Send the script to the **Job Manager** and execute the script.

When the calculation is done, the `molecular_eigenstate.hdf5` file should already be ticked under the Project Files, so its objects will appear on the **LabFloor**. Select the first eigenstate object. Click the button *Viewer* in the tool bar on the right-hand side and select *Isosurface*. In the **Viewer**, open the **Properties** menu, select the *Isosurface* tab and do the following settings.

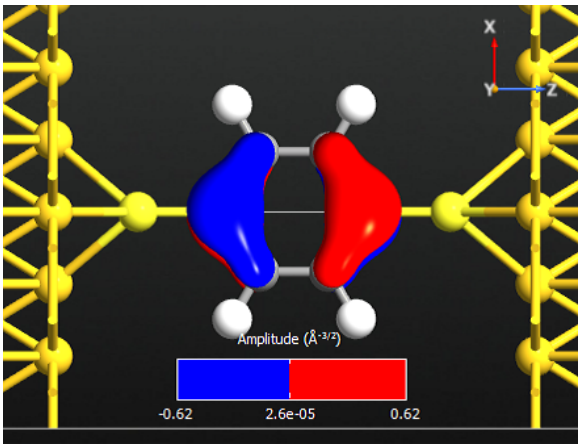
•Set the Isovalue to 0.16.

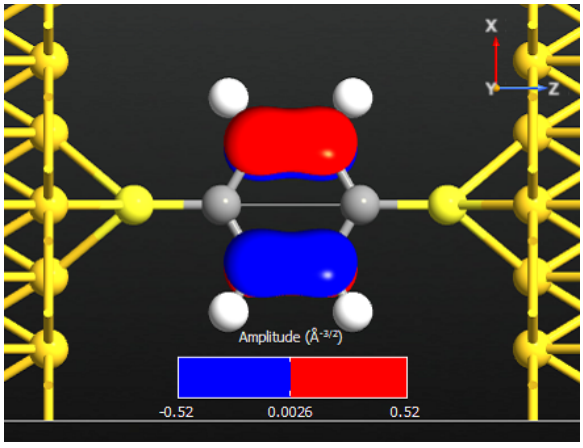
•Set the Color map to BlueRed

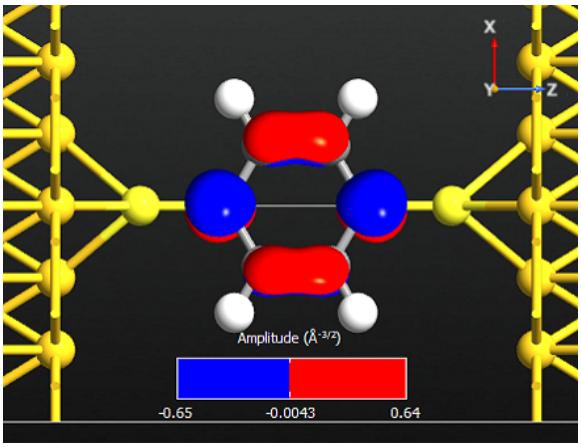
•Tick ± Isovalue

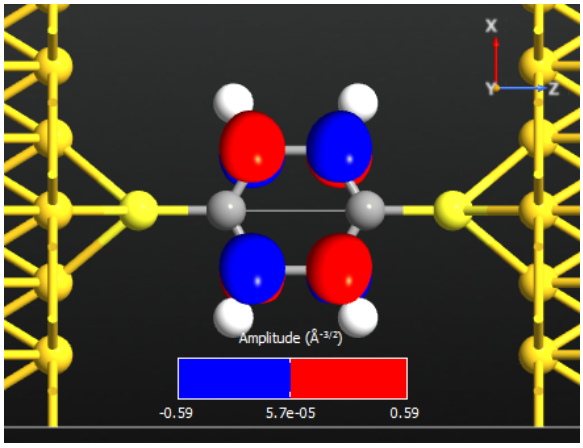


To visualize the eigenstate together with geometry, drag and drop the file `builder_device.py` onto the **Viewer**. Do the same to the other eigenstates and you will get the figures below:





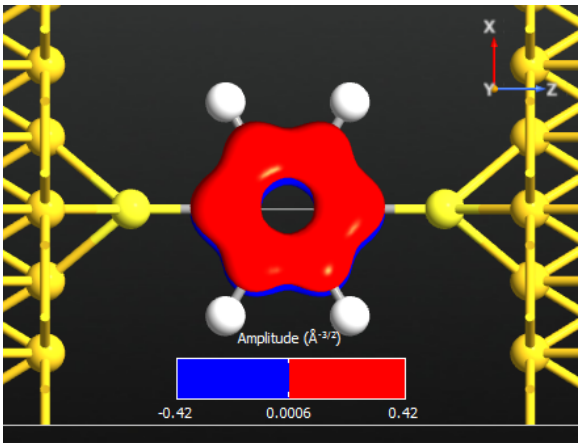


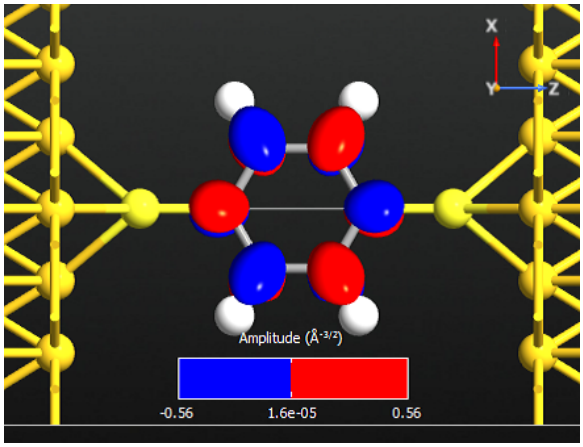


Note that all the states are anti-symmetric in the plane of the benzene ring. This means that they are the π-orbitals of the benzene ring, i.e. they are linear combinations of the p-orbitals on each carbon atom which are perpendicular to the plane of the molecule. There are 6 such p-orbitals which means that there are 6 π-orbitals.

By visually inspecting the symmetry of the different eigenstates the remaining two π-orbitals are identified as state 10 and 18. Use **Editor** to open the previous script, delete two *Eigenstate* blocks and change the *quantum_number* in the remaining two *Eigenstate* blocks to be 10 and 18 respectively. These are visualized below with an iso-value of 0.16.

Note: There are 6 electrons in the π-band, corresponding to that lowest 3 states are occupied. We may therefore denote state 14 the HOMO and state 15 the LUMO.





Transmission eigenvalues and eigenstates

In this chapter you will analyze the transmission eigenstates at the peak positions in the transmission spectrum, i.e. at energies -3, -1.5 and 2.4 eV.

- Re-open the **Script Generator** and delete the **Eigenstate** blocks.
- Instead, add three *Analysis* → *TransmissionEigenvalues* blocks.
- Set the energies to -3, -1.5 and 2.4 eV, into the three **TransmissionEigenvalues** blocks respectively.
- Execute the script using the **Job Manager** and inspect the log file.

```
+------------------------------------------------------------------------+

Eigenvalues (Up):
7.806646e-01
2.244804e-02


Eigenvalues (Up):
7.900386e-01
2.194993e-02


Eigenvalues (Up):
6.175970e-01
1.270894e-02

+------------------------------------------------------------------------+
```

The transmission eigenvalues are obtained by diagonalizing the transmission matrix. The number of eigenvalues indicates the number of individual channels through the molecule, and the eigenvalue shows the strength of each channel. The eigenvalues are the true transmission probabilities, and thus lie in the interval [0,1]. If several channels are available at a particular energy, their sum - and hence the transmission coefficient at this energy - may however be larger than 1.

Most of the eigenvalues are very small and can be neglected. The two largest eigenvalues of each energy are the most important ones.

Note: At the first two energies there is only one dominating eigenstate, while at the last energy there are two dominating eigenvalues. The reason that they do not sum up to the total transmission spectrum T(E) will be discussed in the next section.

The next step is to visualize the transmission eigenstates for the first two peak energies, where there is only one significant eigenvalue.

- Re-open the **Script Generator** and delete the **TransmissionEigenvalues** blocks .
- Add two *Analysis* → *TransmissionEigenstates* blocks.
- Enter the energies -3 and -1.5 into the *Analysis* → *TransmissionEigenstates* blocks . Leave the quantum number at 0, corresponding to the highest eigenvalue.
- Execute the script via the **Job Manager** .

The objects will again be saved in the file `dtb_analysis.hdf5` and can be visualized using the same approach as for the eigenstates. Below are shown isosurfaces of the two transmission eigenstates.

Tip: The transmission eigenstate is a complex wave function. The isosurface shows the absolute value of the wave function while the color of the isosurface indicates the phase. Since the phase is periodic, it is best to use a periodic color map.

The transmission eigenstate at -3.0 eV has a clear resemblance with MPSH state 13, i.e. the HOMO-1 state. The transmission eigenstate at -1.5 eV is also a π-state.

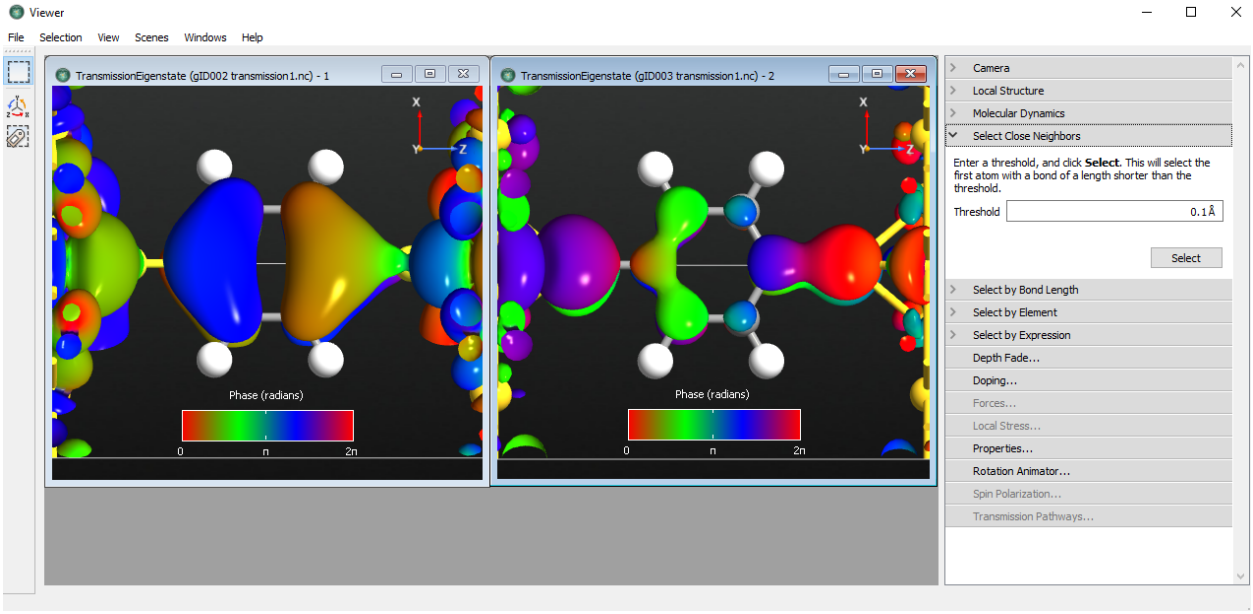


Fig. 3.95: Transmission eigenstates of energy -3 and -1.5 eV

k-dependent transmission and eigenchannels

You will now analyze the transmission at 2.4 eV. Previously, you found that there is an important transmission eigenstate at this energy with transmission probability 0.618 which is different from the value 1.45 obtained from inspecting Fig. 3.94. This is because Fig. 3.94 shows the k-point averaged transmission. You will now calculate the k-dependent transmission coefficients at this energy.

•Re-open the **Script Generator** and delete the **TransmissionEigenstates** blocks.

•Add an *Analysis* → *TransmissionSpectrum* block .

•Open the **TransmissionSpectrum** block and set

–Energy start and end equally to 2.4 eV,

–Number of energy points to 1,

–k-point sampling to 21 x 21 points,

•Execute the script using the **Job Manager** .

When the calculation has finished you can visualize the k-dependent using *Transmission Analyzer*.

The contour plot of the k-dependent transmission above has a pronounced peak at $(k_A, k_B) = (0.0, 0.38)$, with a transmission coefficient of around 1. In the computation of the transmission spectrum earlier, we did not have a dense enough k-point sampling, and hence the transmission was overestimated, i.e. we obtained 1.45 while the 21x21 mesh gives around 1.1.

Attention: This shows the importance of carefully checking the convergence of the results in the k-point sampling of the transmission spectrum. There is no reason to assume that the same number of points used to get a correct

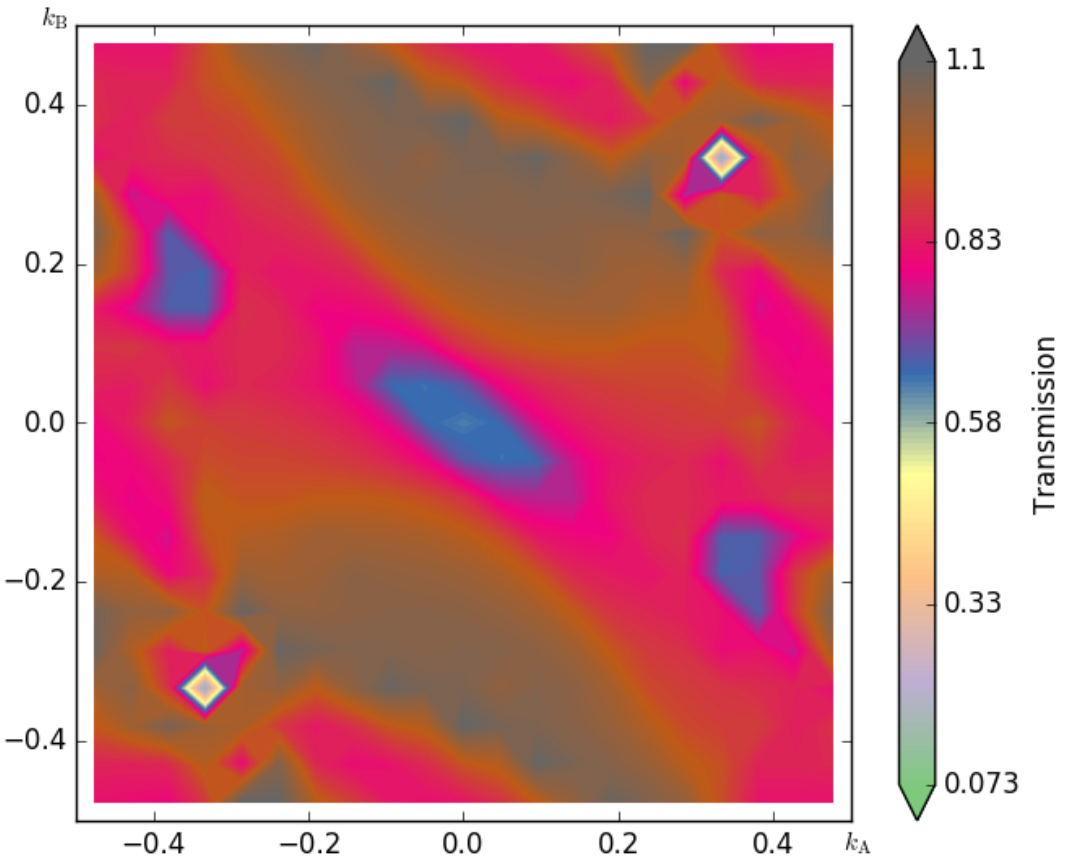


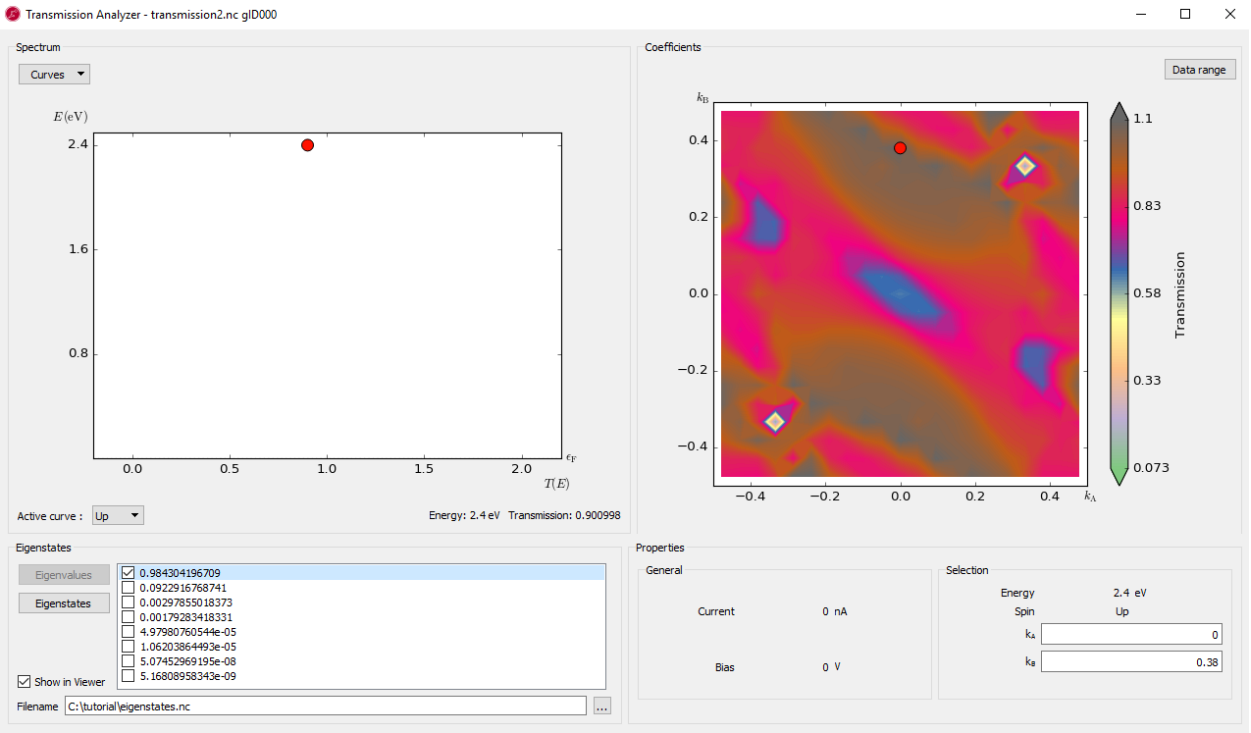
Fig. 3.96: The k-dependence of transmission at energy point 2.4eV

electron density (i.e. the k-points used in the self-consistent calculation) also give an accurate transmission.

You will now calculate the transmission eigenstates at the k-point with the strongest transmission. We can choose either point (0, -0.38) or (0, 0.38), they are equivalent by time-reversal symmetry.

•Click the point (0, -0.38) or (0, 0.38) on the transmission contour plot.

•Click *Eigenvalues* in the **Eigenstates** block on the lower left side. You will get a list of eigenvalues.



•Select the first eigenvalue only, and press the button *Eigenstates*. The first eigenstate will be calculated and the **Viewer** window will pop up with a figure. Tune the isovalue in *Properties* to display the eigenvalue image nicely.

Projected local DOS

In this section you will calculate the local device density of states throughout the device, projected onto the real-space transport direction (the Z-axis).

•Re-open the **Script Generator** and delete the previous analysis blocks.

•Add a *Analysis → ProjectedLocalDensityOfStates* block.

•Open the analysis block and edit it: Increase the energy window to -3 eV to +3 eV, and set the k-point sampling to 3x3.

•Change the output filename to `pldos_0.0V.hdf5` and save the QuantumATK Python script as `pldos_0.0V.py`.

Execute the script – the calculations may take up to an hour to run. The PLDOS item will be added to the QuantumATK LabFloor. Plot the results using the **Projected Local Density of States** plugin in the right-hand plugins menu. Adjust the data range to get a plot like the one shown below.

The PLDOS shows a high density in the metallic gold electrodes (pink regions), while the black color in the middle corresponds to vanishing DOS in the vacuum gap between the electrodes. In the middle, where the molecule is, there

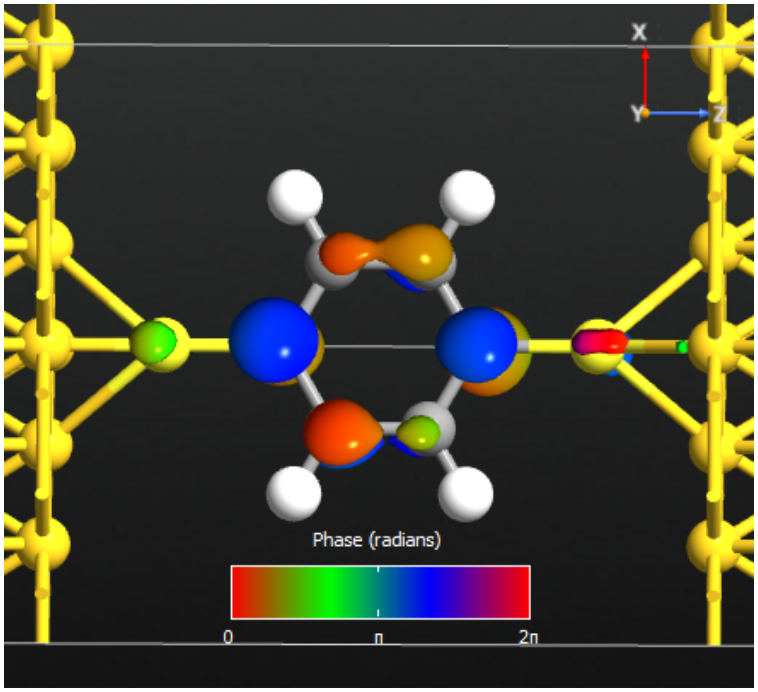


Fig. 3.97: The first eigenstate contributes to the transmission at energy point 2.4 eV.

are three peaks in the local DOS as a function of energy, positioned around -3, -1.4, and +2.5 eV, in agreement with the peaks in the transmission spectrum.

I-V characteristics

In this chapter you will calculate the I-V characteristics of the Au-DTB-Au molecular device, and analyze the device for an applied bias of 1.6 V.

Setting up the calculation

Open a new **Script Generator** window, and do the following:

•Add the Analysis from File block.

•Add the *Analysis* → *IVCurve* block.

•Change the default output filename to `IV-curve.hdf5`.

Edit the Analysis from File block:

•Set *HDF5* file to `au-dtb-au.hdf5`, and use *gID000* to restart from the zero-bias calculation.

Also edit the IVCurve block:

•Set the voltage bias range to end at V_0 = -2V and V_1 = 2 eV.

•Set the number of voltage points to 11.

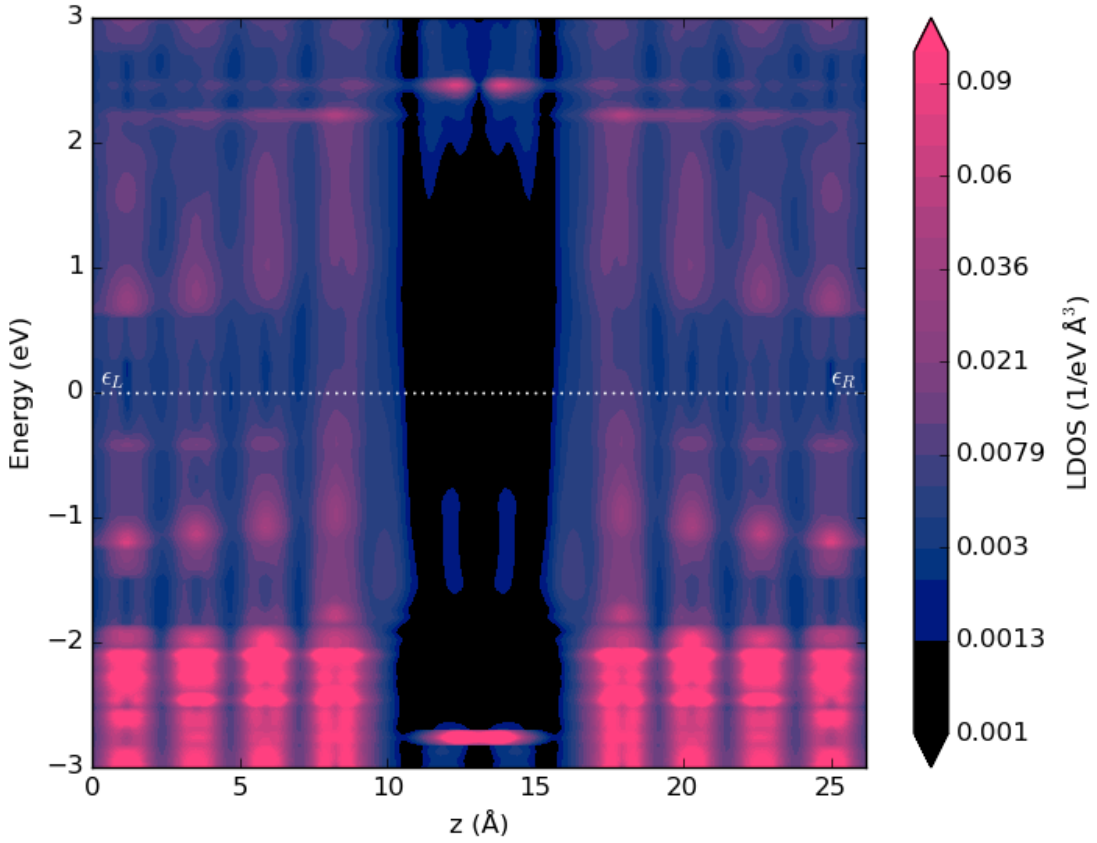
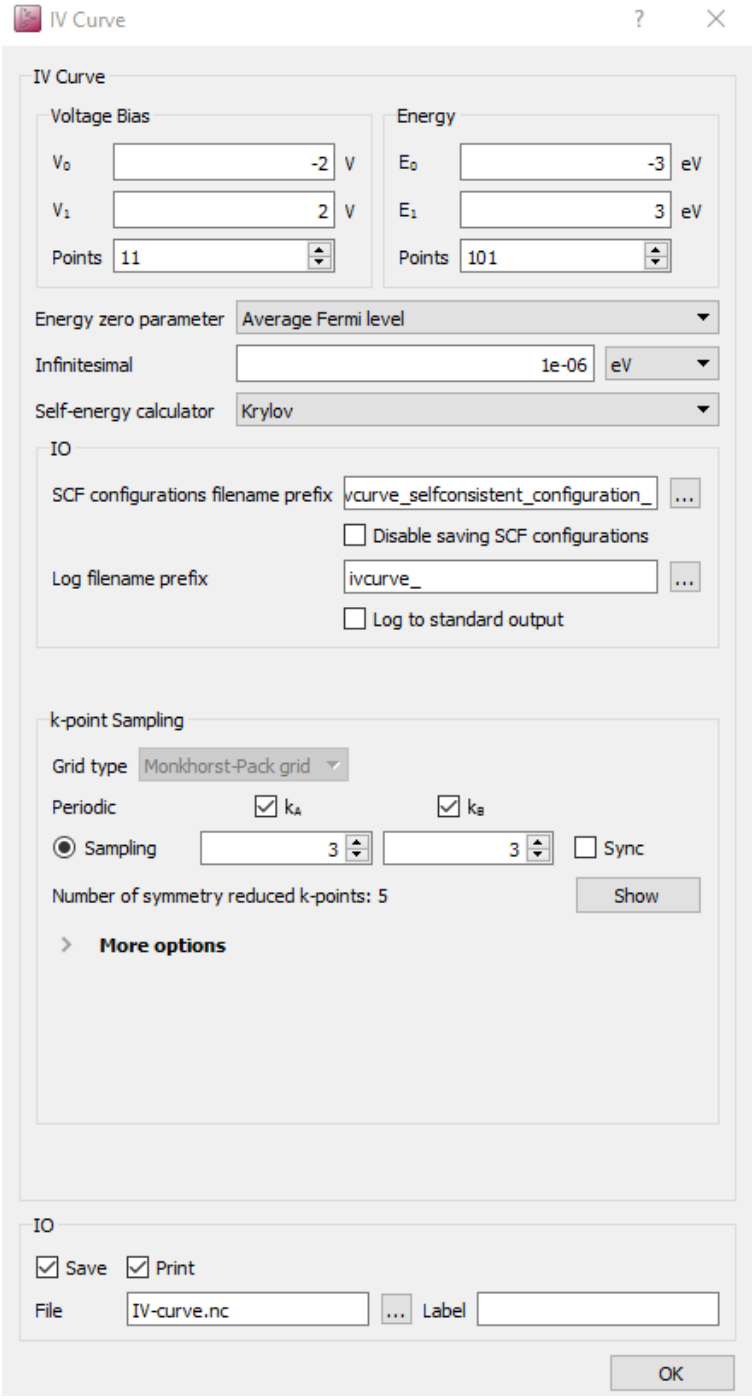


Fig. 3.98: Local device density of states projected onto the device Z-axis.

•Set the energy range as (-3,3).

•Use 3x3 transverse k-points.

•Choose the *Krylov* self-energy calculator (for speed).

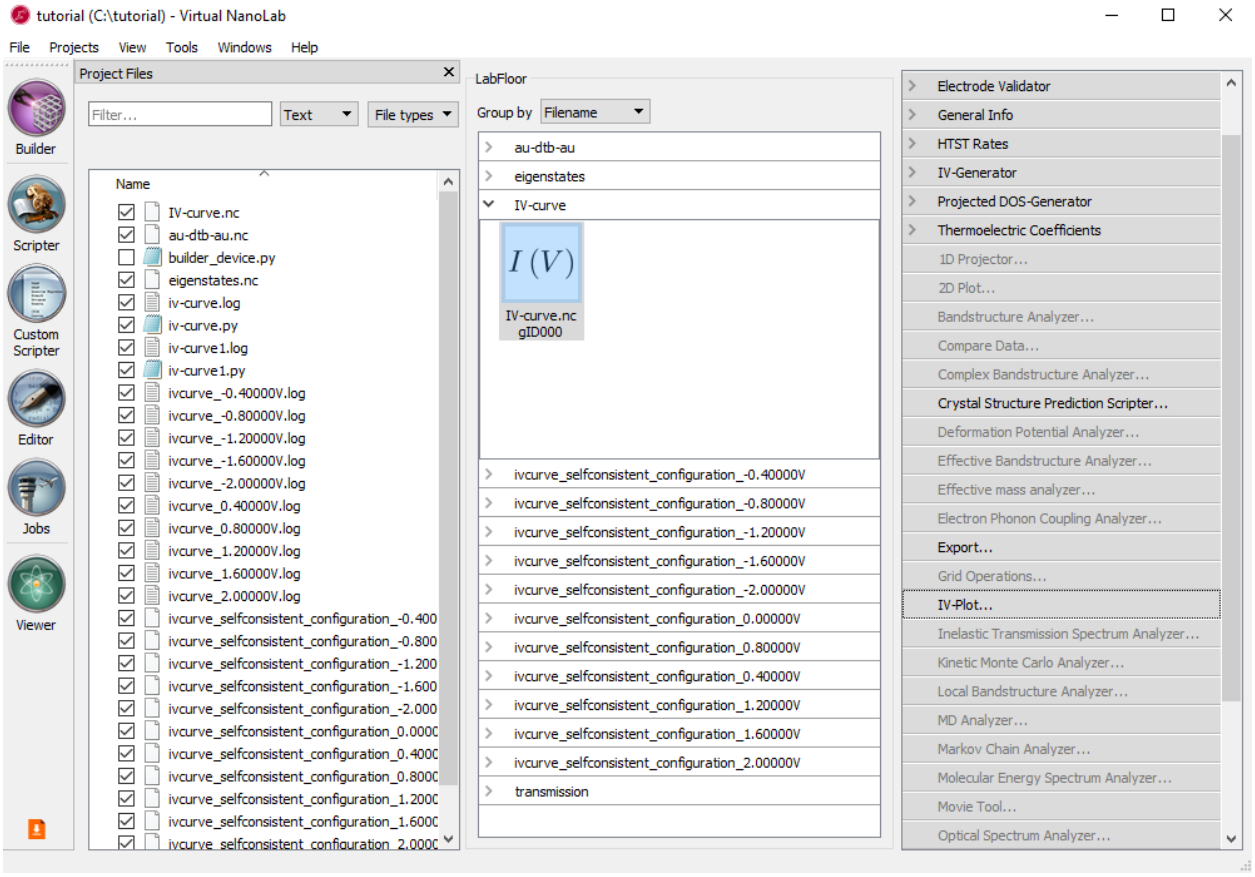


The script is now done. Save it as `iv-curve.py` and execute it using the **Job Manager** or from the command line.

This calculation will take some time, because a self-consistent SCF calculation and a transmission spectrum analysis must be done at each bias point. It takes six hours to finish in a 12-core machine.

IV characteristic analyzation

Once the calculation is done, the file `IV-curve.hdf5` appears in the folder. Eleven new output files are also created, prefixed `ivcurve`; they contain the converged SCF state at each bias point.



Note: The **IVCurve** object contains the calculated I–V curve, including the transmission spectra computed $I(V)$ at each bias voltage.

Select the IV-curve and click the **IV-Plot** analysis tool in the right-hand plugins bar. The I–V curve is plotted in the window that pops up.

Check the box *Additional plots*. Three additional panels appear, plotting the differential conductance (dI/dV), the calculated transmission spectra, and the spectral current.

Attention: Pay special attention to the spectral current: All possible contributions to the total current must be included inside the specified energy window, in order to get accurate $I(V)$ points. If this is not the case, you should consider redoing the analysis using a wider energy window.

Note: The peak in the dI/dV around 1.6 V arises from a resonance in the transmission spectrum entering the bias

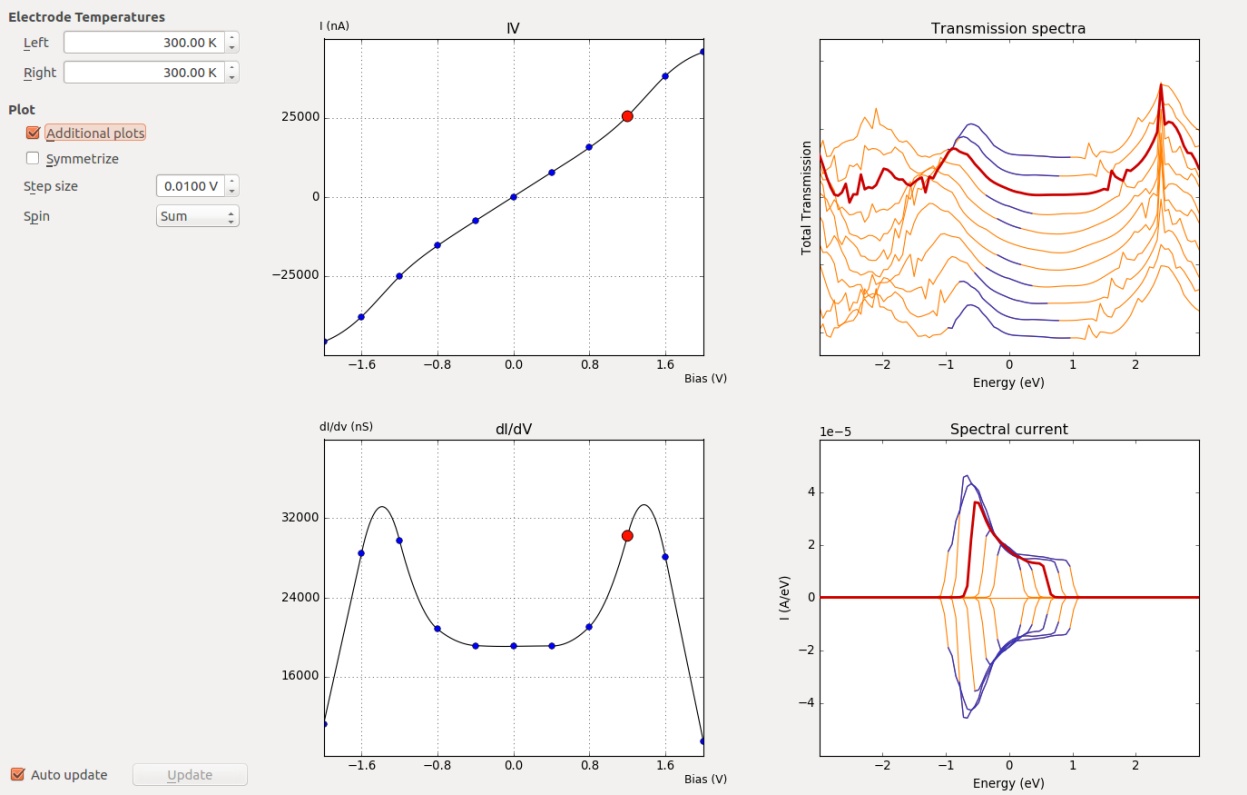


Fig. 3.99: Each transmission spectrum and spectral current correspond to a specific bias voltage point. The energy range corresponding to the bias window is indicated in blue. By moving the mouse cursor to one of the marker points on either the IV or dI/dV plot, the corresponding transmission spectrum and spectral current will be highlighted.

window at this voltage.

PLDOS at 1.6 V bias

To investigate in details the electronic features of the molecular device under bias of 1.6V, you should compute the projected local density of states, just like you did above for zero bias, in section *Projected local DOS*.

•Open a new script by cliking **Script Generator** and add two blocks:

– *Analysis from file*

– *Analysis → ProjectedLocalDensityOfStates*

•Select file `ivcurve_selfconsistent_configuration_1.60000V.hdf5` from the previous calculation after opening the *Analysis from file* block.

•In the PLDOS analysis block, increase the energy window to -3 eV to +3 eV, and set the k-point sampling to 3x3.

•Change the output filename to `pldos_1.6V.hdf5` and save the QuantumATK Python script as `pldos_1.6V.py`.

Run the script and use agin the **Projected Local Density of States** plugin to visualize the results.

Compared with Fig. 3.100, the resonance at -1.5 eV has split into two resonances, at -0.8 and -2 eV, as we saw already in the transmission spectrum above. We also see that the resonance at -0.8 is located towards the right electrode, while the resonance at -2 eV is located towards the left electrode. The splitting arises from the drop in the electrostatic potential through the molecule (due to the applied bias), as will be investigated in the next section.

Calculating the voltage drop

You will now calculate the voltage drop and induced density for an applied bias of 1.6 V. At first, we will calculate the ElectrostaticDifferencePotential and ElectronDifferenceDensity between 1.6 V and 0 V.

•In the **Script Generator** , add three blocks:

–the Analysis from File

–an *Analysis → ElectrostaticDifferencePotential*

–an *Analysis → ElectronDifferenceDensity*

•Open the Analysis from File to load the `ivcurve_selfconsistent_configuration_1.60000V.hdf5`.

•Change the default output filename to `voltage_drop_16V.hdf5`.

•Send it to the **Job Manager** by pressing the **Send to** button.

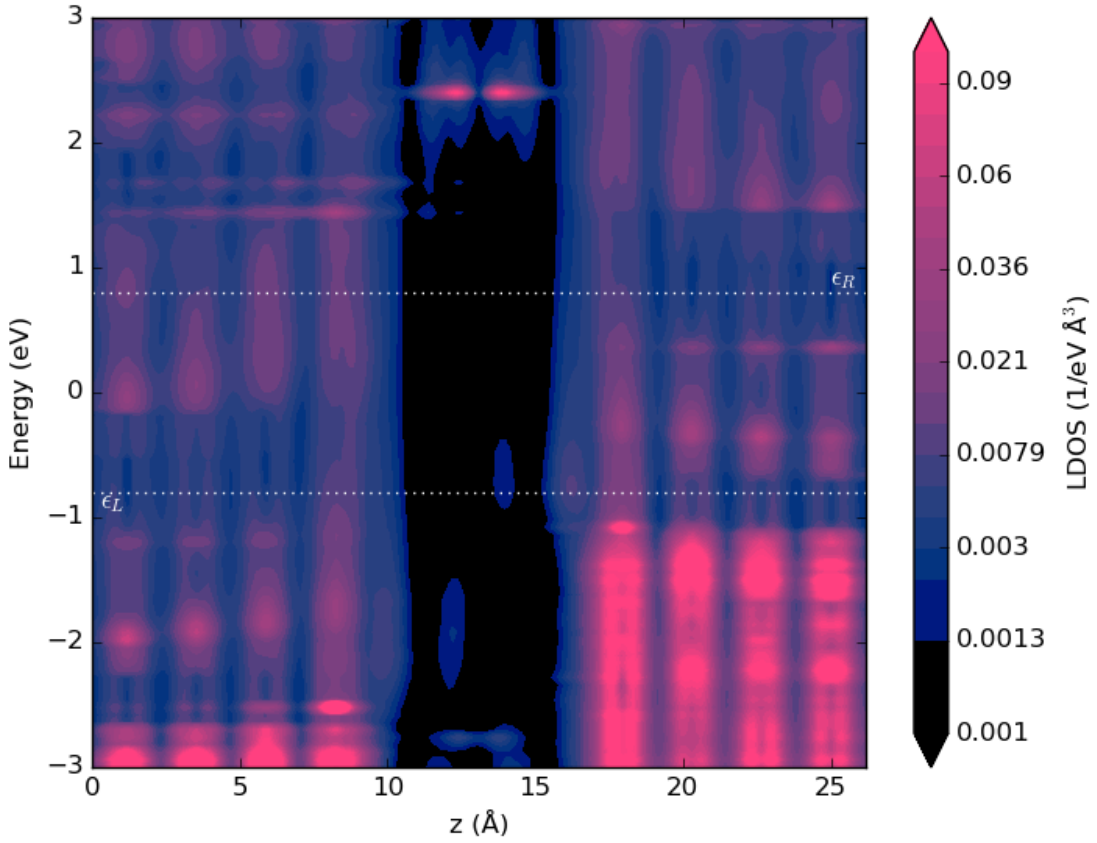
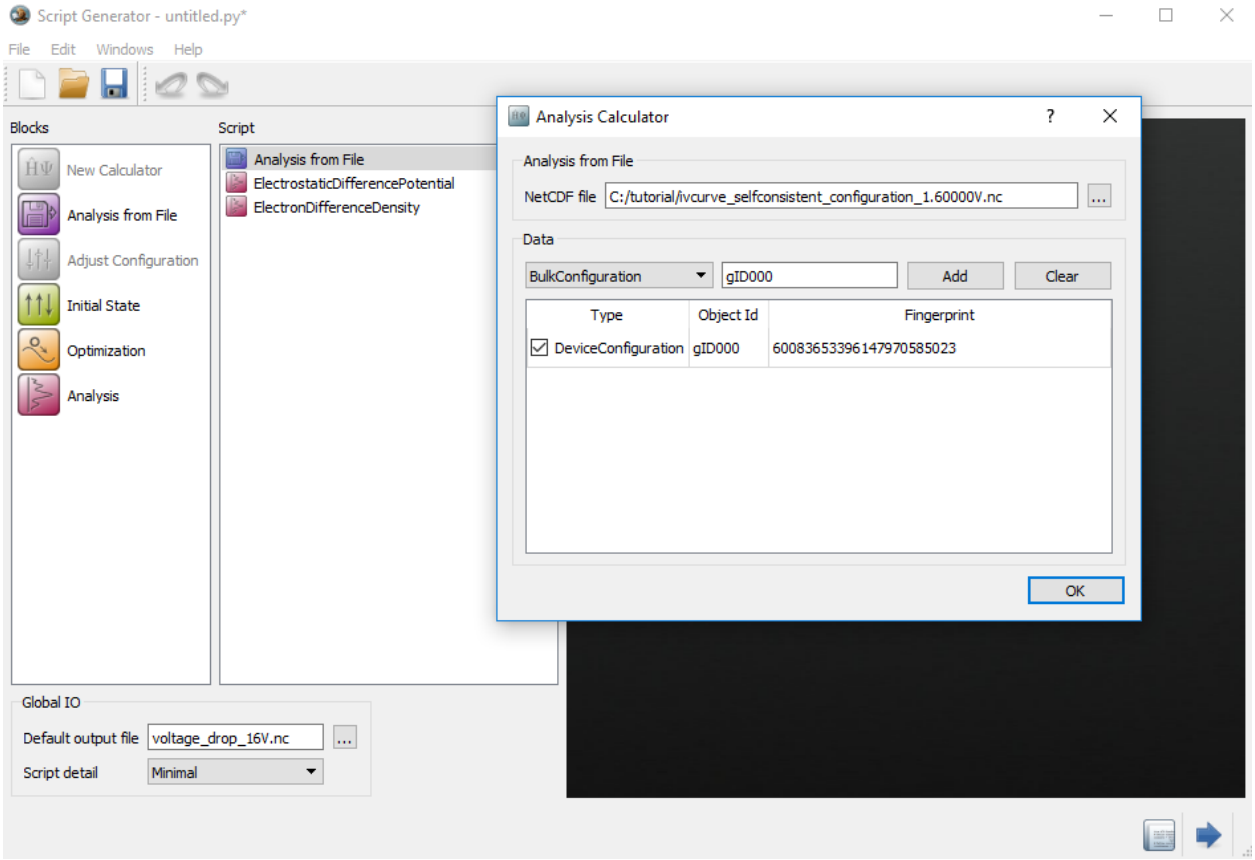


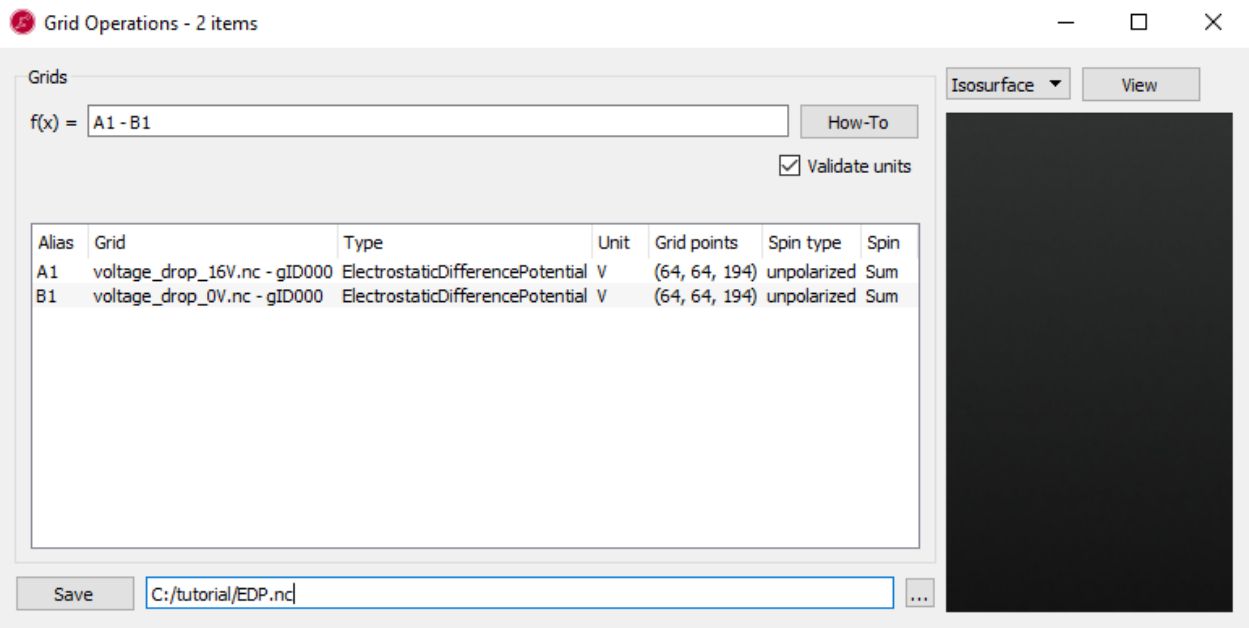
Fig. 3.100: Projected local device DOS at 1.6 V applied bias.



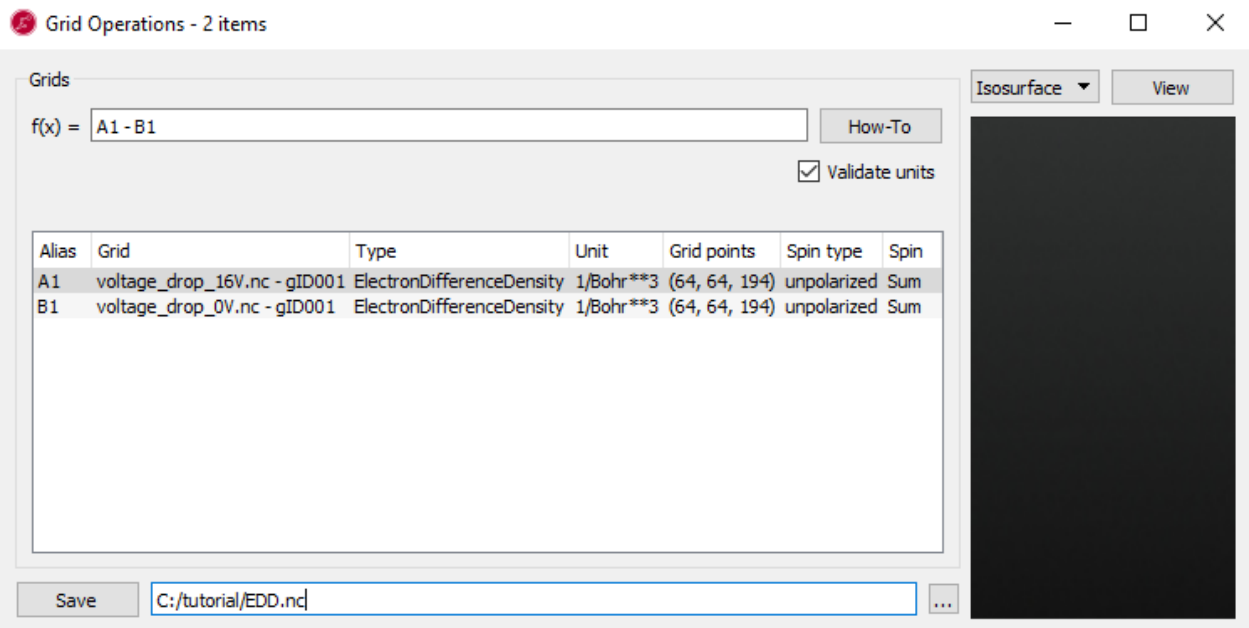
Repeat the above recipe to calculate the ElectrostaticDifferencePotential and ElectronDifferenceDensity at 0 V. Finally, you will have the files `voltage_drop_16V.hdf5` and `voltage_drop_0V.hdf5` under *Project files*.

To calculate the voltage drop using the ElectrostaticDifferencePotential and ElectronDifferenceDensity at 1.6 V and 0 V, we use GridOperations in the right-hand panel bar.

- Select the two objects by holding the `ctrl` key while clicking on each $\delta V_E(\mathbf{r})$ icon in the `voltage_drop_16V` and `voltage_drop_0V` files on the **LabFloor**.
- Open the **GridOperations** widget.
- Make a function such as A1 - B1.
- Save as `EDP.hdf5`.



Repeat the above process for the density to get the voltage-induced density.



You should now have the calculated grids in the files `EDP.hdf5` and `EDD.hdf5`. To visualize the ElectrostaticDifferencePotential and ElectronDifferenceDensity induced by the 1.6 voltage, do the following:

•Select the file `EDP.hdf5` and open the **Viewer** in the panel.

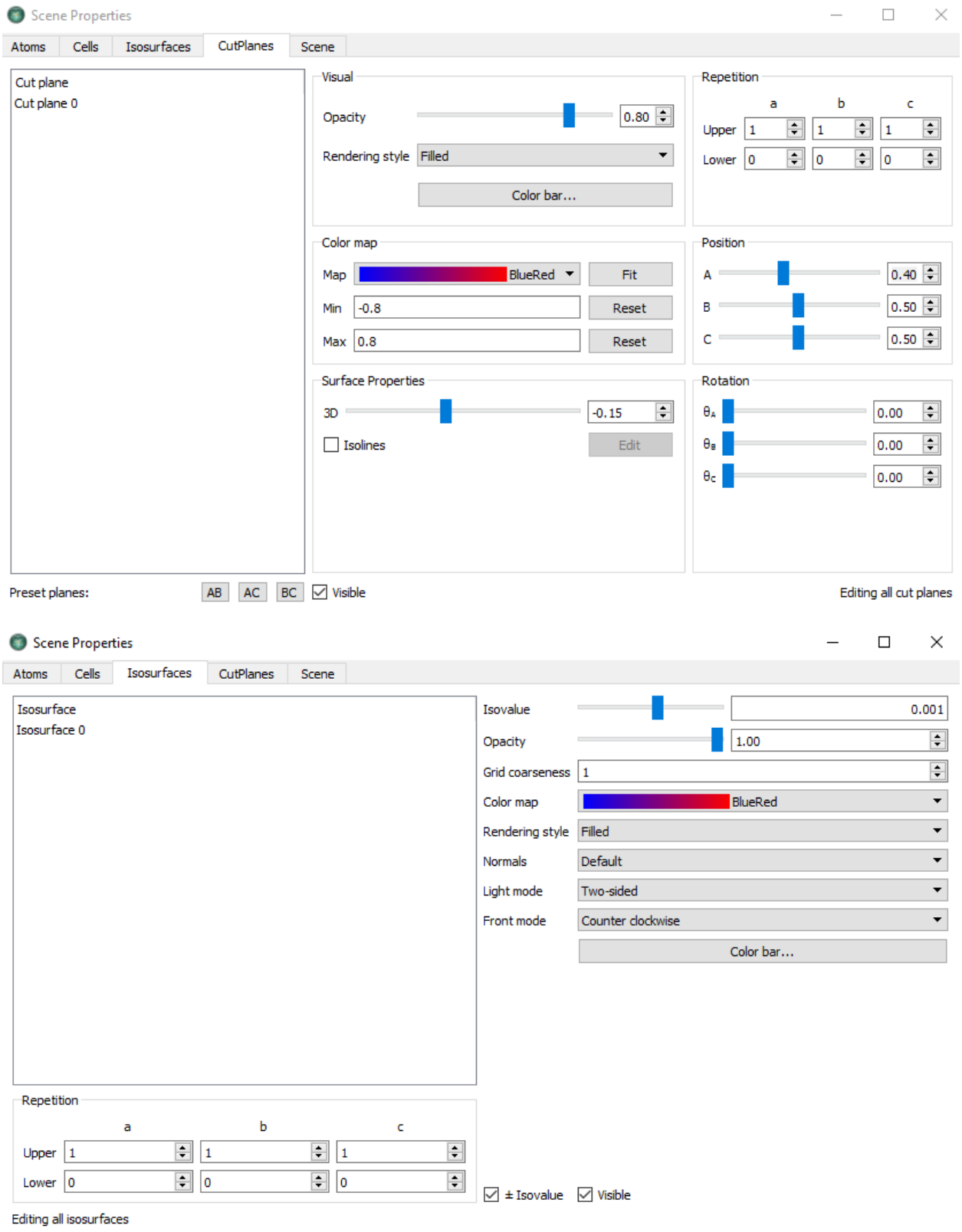
•Change the plot type in the grid visualization selection to the CutPlanes.

•Drag the `builder_device.py` on the window.

•Next, select the file `EDD.hdf5` and drag into the same window.

•Keep the default, Isosurface for the ElectronDifferenceDensity.

You can tune the CutPlanes and Isosurface in the properties. In the case of CutPlanes, you can adjust the 3D and position of the CutPlane.



If you apply the settings shown above, you should now have the following figure:

It is useful to make a 1D plot of the voltage drop. To this end, select the $f(x)$ of the *EDP* in the **LabFloor** and click

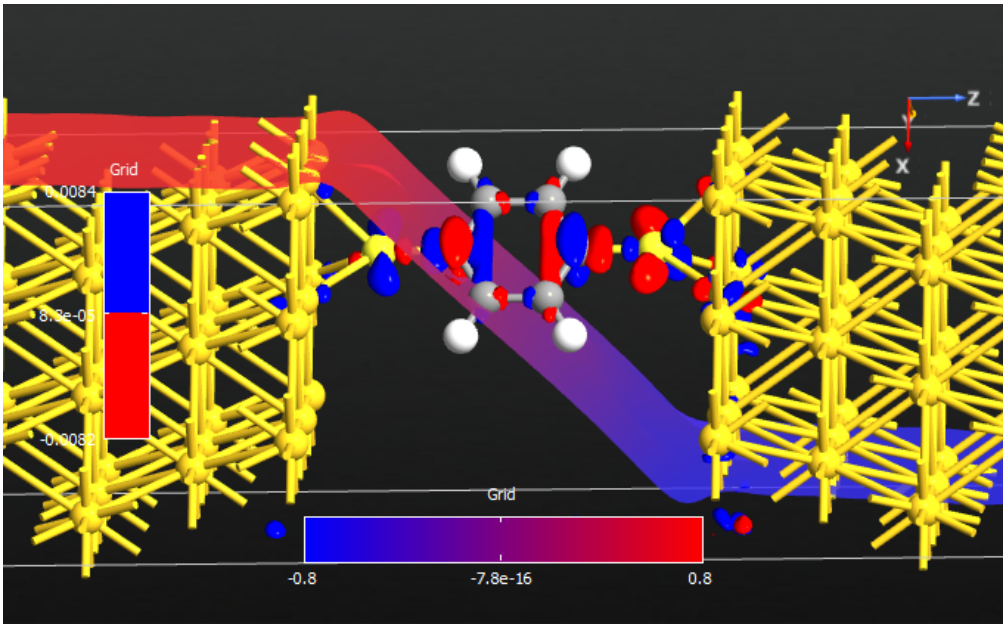


Fig. 3.101: Electrostatic potential drop and electron density difference induced by a bias of 1.6V.

the **1D Projector** in the right hand panel bar. Then change the projection type to *Average* and click *Add line*.

In the case of 1D projection plot, c-axis shows the relative length between 0 and 1 corresponding to the real z distance.

Note: Both Fig. 3.101 and Fig. 3.102 show the total voltage drop of 1.6 V across the device. It is also clear from both figures that most of the voltage drop takes place around the molecule, not in the electrodes. A drop in voltage corresponds to a quantum resistance, and this explains the observed finite-bias splitting of the zero-bias resonance.

References

3.10 Spintronics

3.10.1 Spin Transfer Torque

Version: 2016.3

In this tutorial you will learn about the spin transfer torque (STT) and how to calculate the STT in magnetic tunnel junctions.

We will use a simple model of such a tunnel junction; a magnetic carbon-chain device. The QuantumATK package offers a convenient analysis object for calculating STT linear-response coefficients at zero bias, but you will also calculate the STT using finite-bias calculations and compare results to those obtained using linear response.

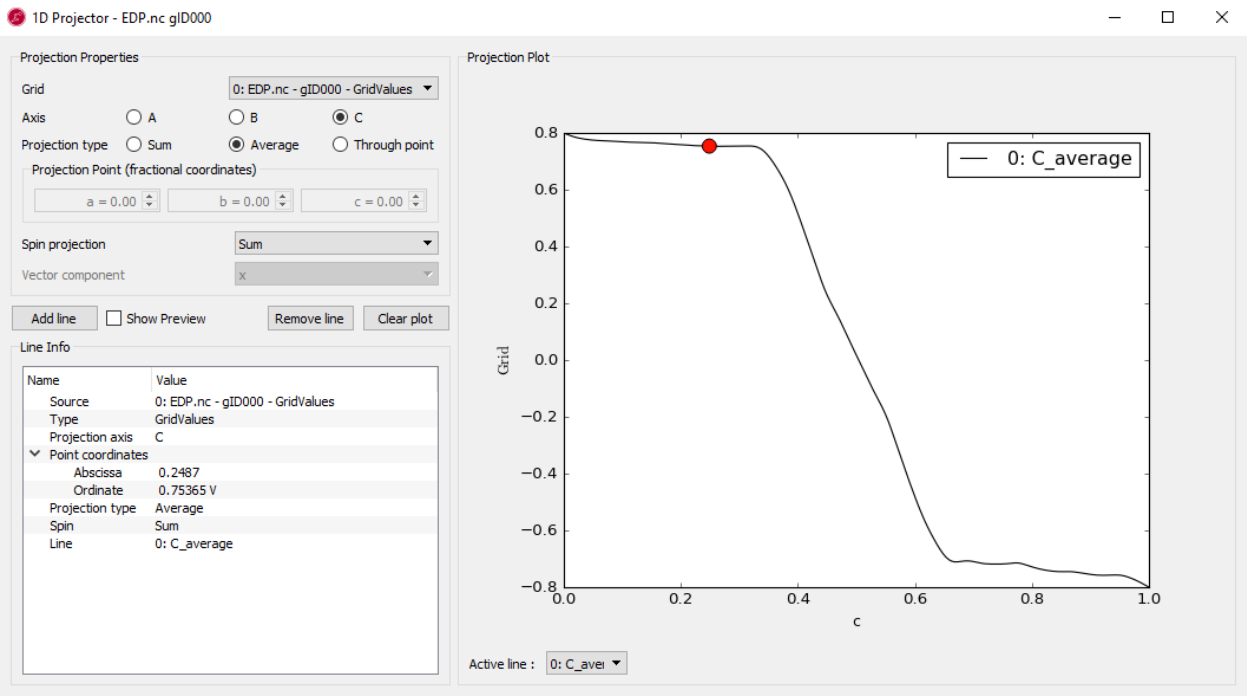


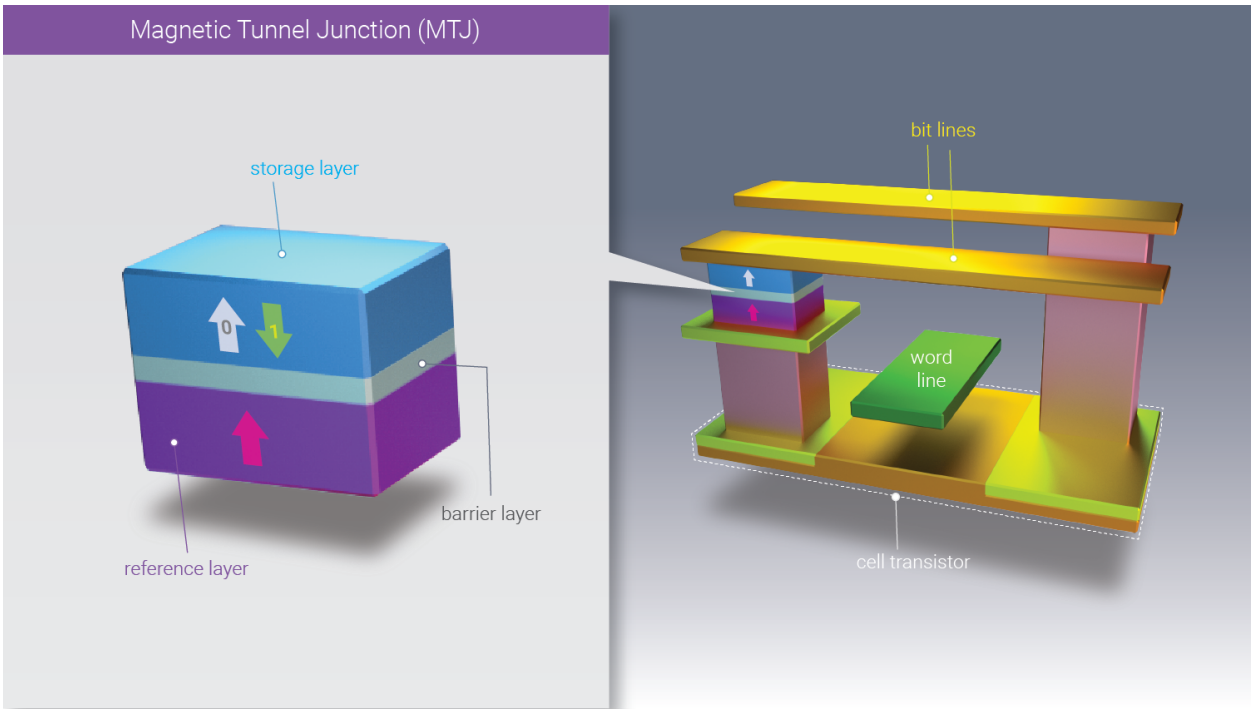
Fig. 3.102: Voltage drop along a line through the molecular axis.

Introduction

Spin transfer torque occurs in situations where a current of spin-polarized carriers from the left part of a device with a particular polarization (given by the unit vector S_1) enters the right part of the device with a different magnetization direction (given by the unit vector S_2). When the electrons incoming from the left side enter the right magnetic part, they will eventually be polarized along S_2, meaning that the right magnetic domain has **exerted a torque** on the electrons in order to rotate their spin angular momentum. However, due to conservation of angular momentum, the electrons exert an **equal but opposite torque** on the right magnetic domain – the spin transfer torque. Note that a noncollinear description of electron spin is needed to study this effect!

Application in Random Access Memory (RAM)

The spin transfer torque can be used to modify the orientation of a magnetic layer in a magnetic tunnel junction (MTJ) by passing a spin-polarized current through it, and can therefore be used to flip the active elements in magnetic random-access memory (MRAM).



Theory

There are two principally different ways to calculate STT from atomic-scale models:

1. The STT can be found from the divergence of the spin current density, $\nabla \cdot I_s$, which in QuantumATK can be directly calculated using Greens function methods.

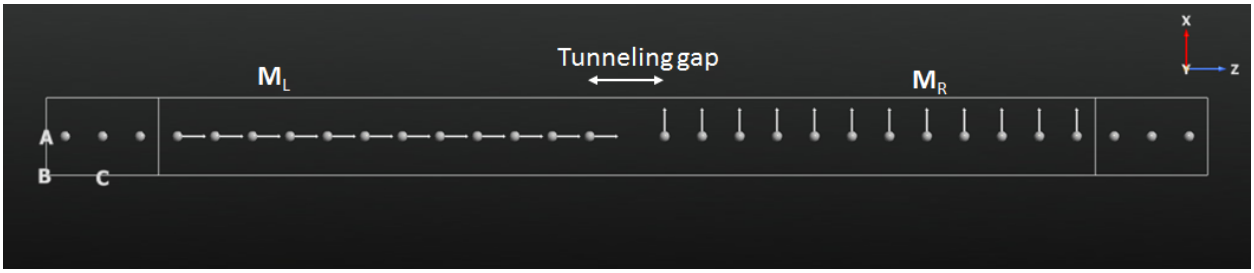
2. Another way to calculate the STT, here denoted $\mathbf{T}$, is from the expression $\mathbf{T} = \mathbf{Tr}(\delta\rho_{\mathrm{neq}}\sigma \times \mathbf{B}_{\mathrm{xc}})$, where $\delta\rho_{\mathrm{neq}}$ is the non-equilibrium contribution to the density matrix, σ is a vector of Pauli matrices, and $\mathbf{B}_{\mathrm{xc}}$ is the exchange-correlation magnetic field.

   Method 2 will be used in this tutorial. We can either calculate $\delta\rho_{\mathrm{neq}}$ from the difference between finite-bias and zero-bias calculations, $\delta\rho_{\mathrm{neq}} = \rho_{\mathrm{neq}} - \rho_{\mathrm{eq}}$, or we can estimate it using linear response theory, $\delta\rho_{\mathrm{neq}} = (G^{\mathrm{r}}\Gamma_{\mathrm{L}}G^{\mathrm{a}})qU$, where $G^{\mathrm{r/a}}$ is the retarded/advanced Greens function, Γ_{L} is the left-electrode coupling operator, U is the bias voltage, and q is the electron charge. The expression for the STT given in method 2 above is then evaluated in real space.

   ATK implements a convenient **analysis object** for calculating the spin transfer torque using method 2 in the linear-response approximation, where the non-equilibrium density response depends linearly on the bias.

Getting Started

You will here consider a device configuration consisting of a carbon chain with a gap in the middle, which the electrons have to tunnel through. The system is highly artificial, but serves as a simple model of a magnetic tunnelling junction.



Note: There is no need for k-point sampling in the A- and B-directions in this 1D device. However, for commonly studied magnetic tunnelling junctions, like Fe|MgO|Fe, a very fine k-point sampling along A and B is often needed.

Collinear Initial State

The 1D carbon-chain device illustrated above is similar to the one used in the tutorial *Introduction to noncollinear spin*, but not identical to it. Download the device configuration as an QuantumATK Python script: `device.py`.

As already mentioned, a noncollinear representation of the electron spin is needed when calculating STT. We will use a collinear, spin-parallel ground state as initial guess for the noncollinear state. First step is therefore a collinear calculation for the 1D device. Send the geometry to the **Script Generator** and set up the ATK-DFT calculation:

- Set the **default output file** to `para.nc`.
- Add a New Calculator block and set the **Spin** to *Polarized*. The **exchange-correlation** functional will automatically switch to *LSDA*. Change the carbon basis set to *SingleZetaPolarized* in order to speed up calculations.
- Add an Initial State block and select **User spin** in order to initialize the calculation with all carbon atoms maximally polarized.

Save the script as `para.py` (it should look roughly like this: `para.py`), and run the calculation using the **Job Manager**.

Noncollinear State: 90ř Spin Rotation

Next step is a noncollinear calculation where the spins on the left side of the tunnelling gap point in a different direction than the spins on the right side of the gap. The basic recipe to do this is fairly simple:

1. The ATK-DFT calculator settings used in the collinear calculation, saved in `para.nc`, are loaded from file and slightly adapted for the noncollinear calculation.
2. The spins in the right-hand part of the device are rotated 90ř around the polar angle θ.
3. The spin-parallel collinear ground state is used as initial state for the noncollinear calculation.

The QuantumATK Python script shown below implements the basic steps given above, but also runs a Mulliken population analysis for visualizing the resulting spin directions throughout the device, and computes the electrostatic difference potential (EDP) and the transmission spectrum.

```
# Read in the collinear calculation
device_configuration = nlread('para.hdf5', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True),
)

# Get the calculator and modify it for noncollinear LDA
calculator = device_configuration.calculator()
calculator = calculator(
    exchange_correlation=NCLDA.PZ,
    iteration_control_parameters=iteration_control_parameters,
)

# Define the spin rotation in polar coordinates
theta = 90*Degrees
left_spins  = [(i, 1, 0*Degrees, 0*Degrees) for i in range(12)]
right_spins = [(i, 1, theta,     0*Degrees) for i in range(12,24)]
spin_list   = left_spins + right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

# Setup the initial state as a rotated collinear state
device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
    initial_state=device_configuration,
)

# Calculate and save
device_configuration.update()
nlsave('theta-90.hdf5', device_configuration)

# ------------------------------------------------------------------
# Mulliken Population
# ------------------------------------------------------------------
mulliken_population = MullikenPopulation(device_configuration)
nlsave('theta-90.hdf5', mulliken_population)
nlprint(mulliken_population)

# ------------------------------------------------------------------
# Electrostatic Difference Potential
# ------------------------------------------------------------------
electrostatic_difference_potential = ElectrostaticDifferencePotential(device_
↪configuration)
nlsave('theta-90.hdf5', electrostatic_difference_potential)

# ------------------------------------------------------------------
# Transmission Spectrum
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
kpoint_grid = MonkhorstPackGrid(
    force_timereversal=False
)

transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(-2,2,101)*eV,
    kpoints=kpoint_grid,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )
nlsave('theta-90.hdf5', transmission_spectrum)
```

A special density mixer for noncollinear calculations is employed, and the exchange-correlation is changed to `NCLDA.PZ`. Note how the *InitialSpin* object is used to define the initial spin directions on the calculator.

Save the script as `theta-90.py` and run it using the **Job Manager** – it should not take long to finish.

Visualizations

When the calculation finishes, the file `theta-90.nc` should appear in the QuantumATK Project Files list and the contents of it should be available on the LabFloor:

- DeviceConfiguration

- MullikenPopulation

$\mathrm{Tr}(\boldsymbol{\rho}\mathbf{S})$

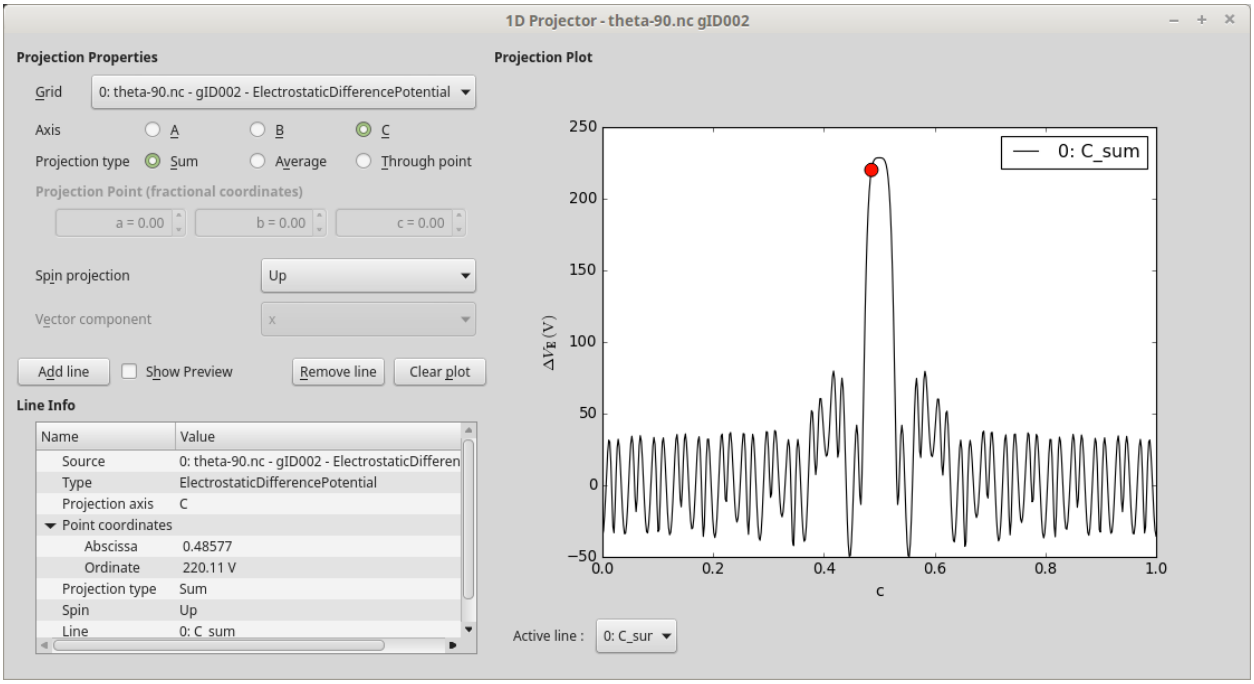
- ElectrostaticDifferencePotential

$\delta V_{\mathbf{E}}(\mathbf{r})$

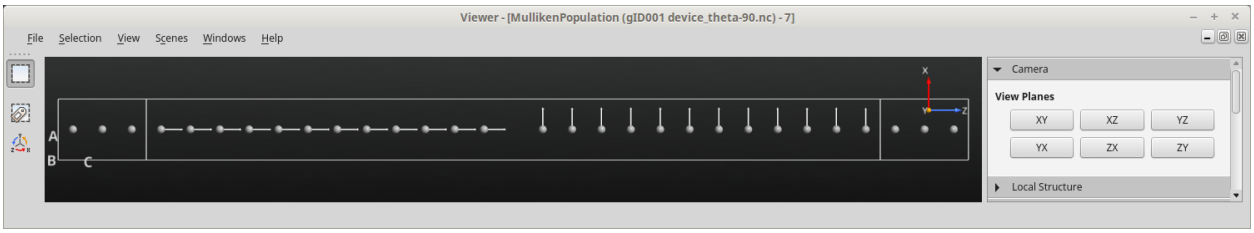
- TransmissionSpectrum

$T\left(\mathbf{k},\epsilon\right)$

First of all, select the **electrostatic difference potential** and plot it using the **1D Projector** in order to check that the NEGF calculation converged to a well-behaved ground state: Project the average EDP onto the C axis, and observe that the ground state electrostatic potential is nicely periodic in both ends of the device toward the electrodes, as it should be, and has the expected non-periodic feature around the gap in the middle of the device.

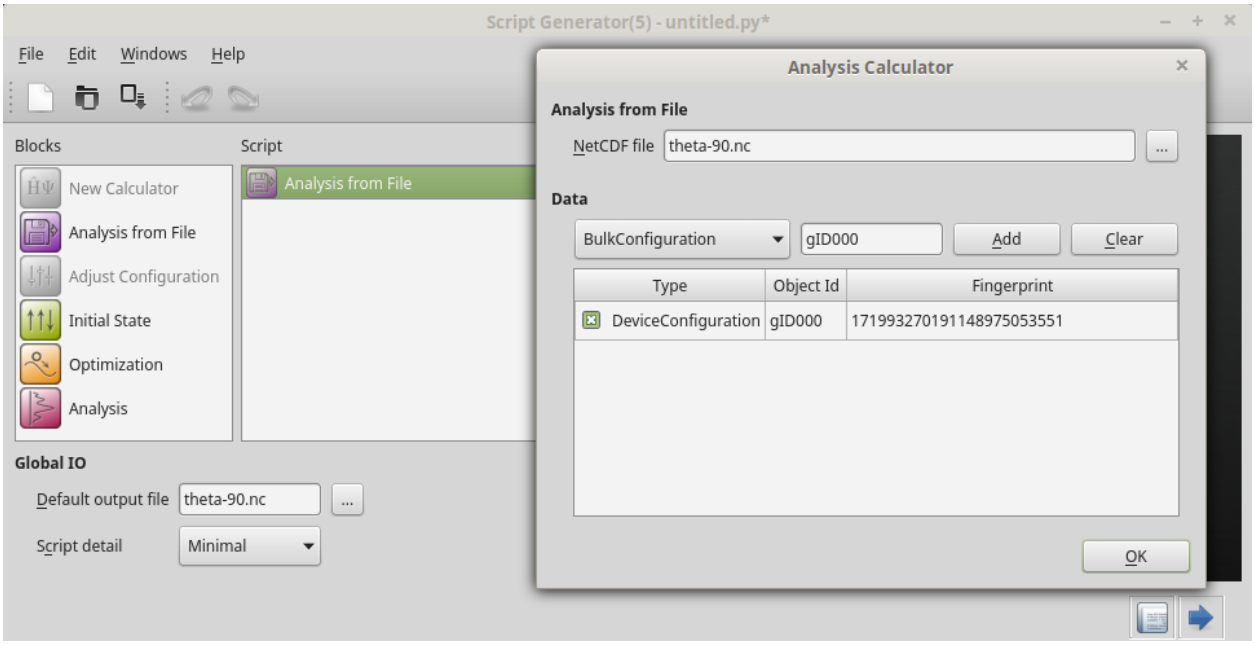


Next, select the **Mulliken population** analysis item and open it in the **Viewer**. Use the *Camera* options (or click) to select the **ZX plane** for a clear view of the spin orientations.



Calculate the STT

You are now ready to use linear response (LR) theory to calculate the spin transfer torque. Open a new **Script Generator** window and add the **Analysis from File** block. Open the block and select the device configuration saved in `theta-90.nc` for post-SCF analysis. Then change the Script Generator default output file to `theta-90.nc` in order to save all calculated quantities in that file.

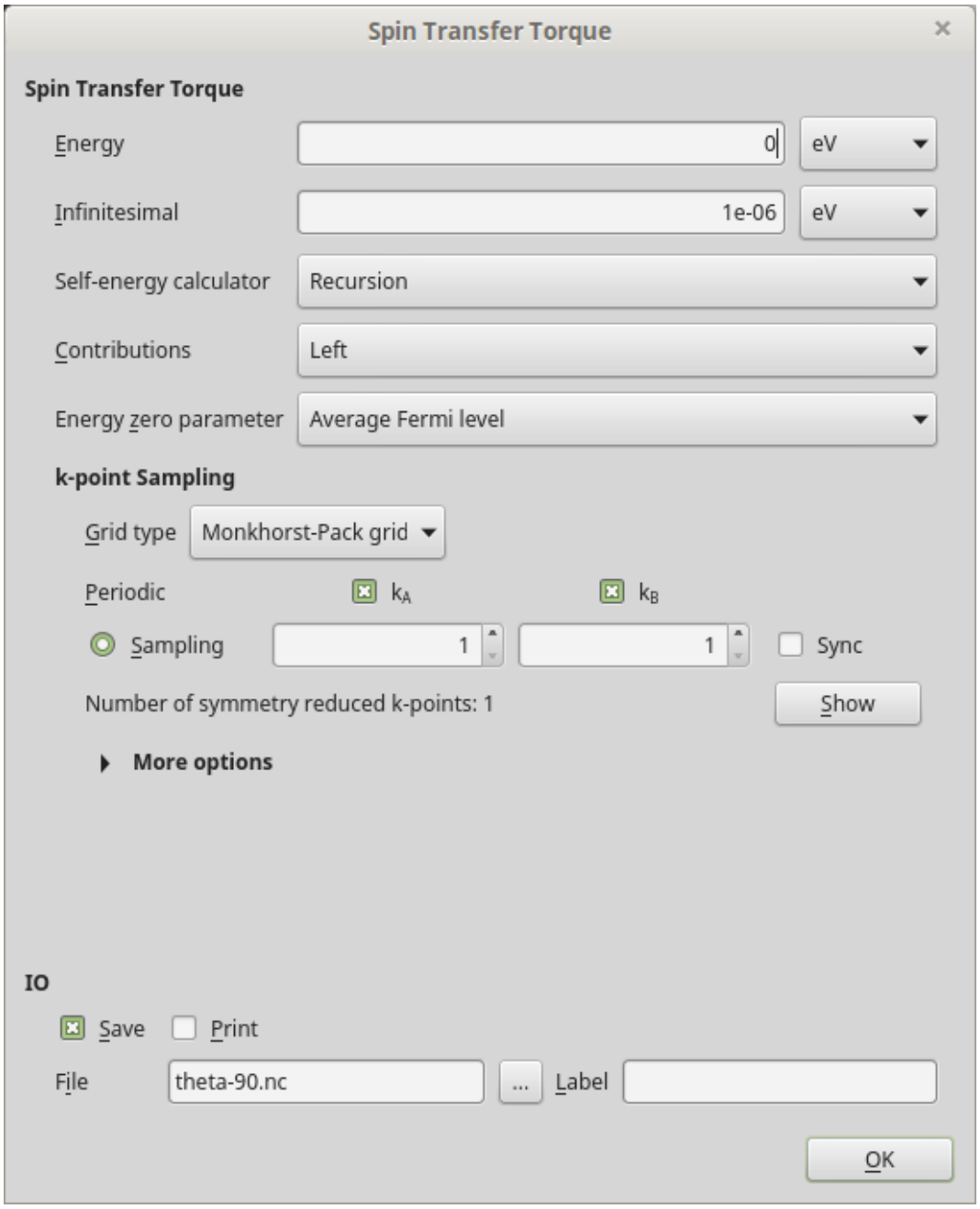


Add also the *Analysis* → *SpinTransferTorque* analysis block, and open it to see the available options. Note in particular 3 important settings:

Energy
The electron energy, with respect to the Fermi level, for which the STT will be calculated.

Contributions
The STT is by default calculated for the left –> right current.

k-points
Fairly dense k-point grids are sometimes needed along periodic directions orthogonal to the transport direction.



Tip: See the QuantumATK Manual entry *SpinTransferTorque* for a full description of all input parameters for the STT calculation and all QuantumATK Python methods available on the object.

Leave all the STT options at defaults and save the script as `stt.py`. The script should look roughly like this: `stt.py`. Run the calculation using the **Job Manager** – it should finish in less than a minute.

The STT analysis object should now have been added to the file `theta-90.nc` and be available on the QuantumATK LabFloor:

- SpinTransferTorque

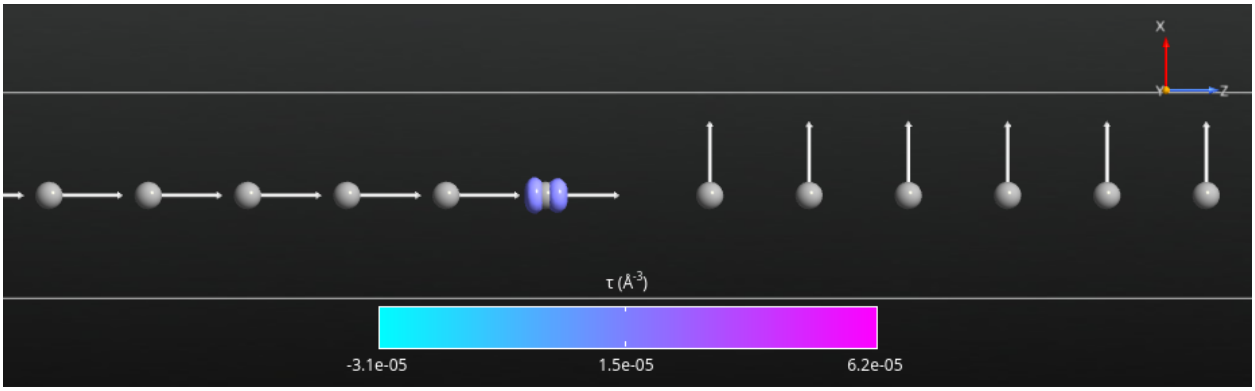
$\tau_i(\mathbf{r})$

Important: The spin transfer torque depends on the bias voltage across the electrodes, and is of course zero at zero bias, since no current flows. The linear-response spin transfer torque (LR-STT) assumes a linear relationship between the STT and the bias voltage V,

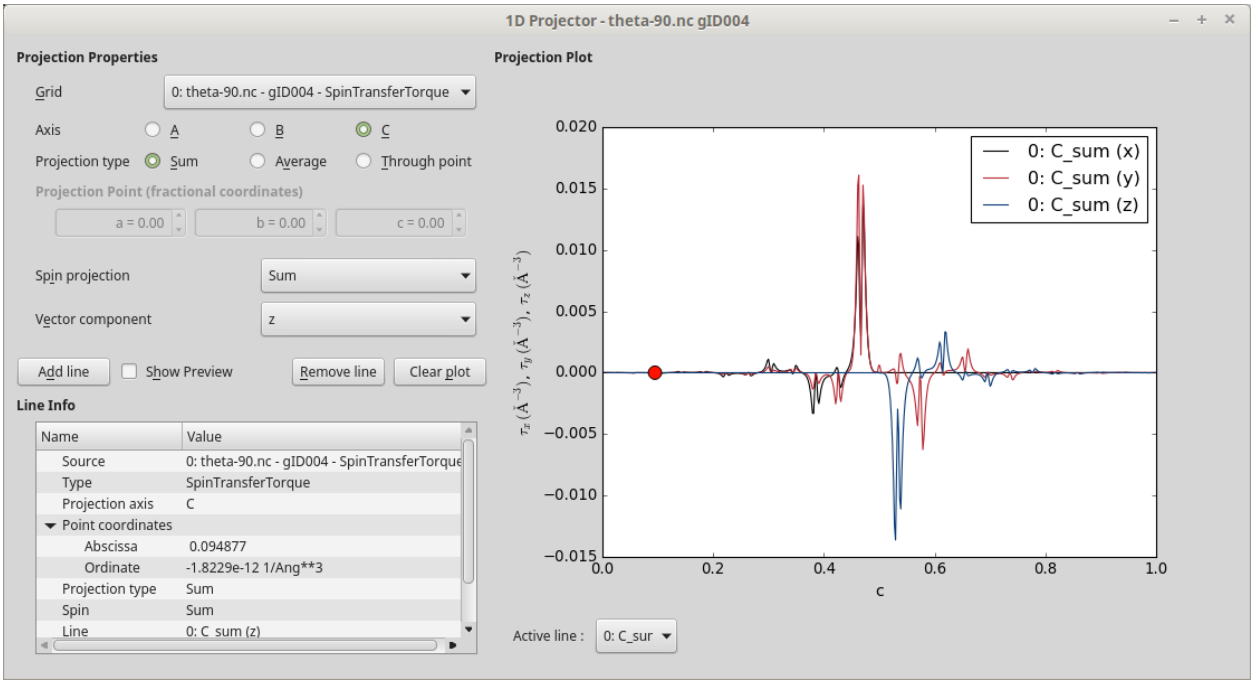
$$\boldsymbol{T}(V) = V\frac{\partial \boldsymbol{T}}{\partial V}\bigg|_{V=0} = V\boldsymbol{\tau}$$

where τ is a **linear-response coefficient**. The QuantumATK SpinTransferTorque analysis object computes the local components of the LR-STT coefficient τ, from which the small-bias STT may be calculated.

Use first the **Viewer** to get a 3D visualization of the values of the LR-STT coefficient for θ=90ř: Open the Mulliken population in the Viewer, and then drag-and-drop the STT item onto the Viewer window. The coefficient τ is represented on a **3D vector grid**, each vector consisting of the local x-, y-, and z-components, so you need to choose which component to visualize. Choose in this case to visualize the y-component as an isosurface:



You can also use the **1D Projector** to visualize the STT components. Select the STT item on the LabFloor, and click the 1D Projector plugin. Plot the x, y, and z vector components in the same plot, projected on the C axis.



It is common to divide the STT into **out-of-plane** and **in-plane** contributions. In the present case, the magnetizations in the left and right electrodes point in the z- and x-directions, respectively, thus defining the in-plane contribution as being in the XZ-plane, while the y-component gives the out-of-plane contribution.

The two figures below show the out-of-plane and in-plane components of the LR-STT coefficient. The spin of the current is oriented along z when entering the right part of the device, and is then turned such as to be aligned along x. The x-component of τ is therefore non-zero only in the left-hand part of the device, while the z-component is non-zero only in the right-hand part. The two components are in this case mirror images of each other.

Tip: Both figures above were generated using the 1D Projector. If you right-click the plot and select **Customize**, a dialog appears where you can customize plot details such as title, legend, colors, etc.

Angle Dependence

Let us next investigate how the LR-STT coefficient depends on the angle θ in the interval 0ř to 180ř. What is needed is a range of range of zero-bias NEGF calculations for different values of θ, each followed by calculation of the SpinTransferTorque analysis object for that particular spin configuration. This is most easily achieved by combining the main parts of the scripts `theta-90.py` and `stt.py` used above:

```
thetas = [0,10,20,30,40,50,60,70,80,100,110,120,130,140,150,160,170,180]

for theta in thetas:
    # Output data file
    filename = 'theta-%i.hdf5' % theta

    # Read in the collinear calculation
    device_configuration = nlread('para.hdf5', DeviceConfiguration)[0]

```

(continues on next page)

(continued from previous page)

```
    # Use the special noncollinear mixing scheme
    iteration_control_parameters = IterationControlParameters(
        algorithm=PulayMixer(noncollinear_mixing=True),
    )

    # Get the calculator and modify it for noncollinear LDA
    calculator = device_configuration.calculator()
    calculator = calculator(
        exchange_correlation=NCLDA.PZ,
        iteration_control_parameters=iteration_control_parameters,
    )

    # Define the spin rotation in polar coordinates
    left_spins  = [(i, 1, 0*Degrees,    0*Degrees) for i in range(12)]
    right_spins = [(i, 1, theta*Degrees, 0*Degrees) for i in range(12,24)]
    spin_list   = left_spins + right_spins
    initial_spin = InitialSpin(scaled_spins=spin_list)

    # Setup the initial state as a rotated collinear state
    device_configuration.setCalculator(
        calculator,
        initial_spin=initial_spin,
        initial_state=device_configuration,
    )

    # Calculate and save
    device_configuration.update()
    nlsave(filename, device_configuration)

    # ------------------------------------------------------------
    # Mulliken Population
    # ------------------------------------------------------------
    mulliken_population = MullikenPopulation(device_configuration)
    nlsave(filename, mulliken_population)
    nlprint(mulliken_population)

    # ------------------------------------------------------------
    # Transmission Spectrum
    # ------------------------------------------------------------
    kpoint_grid = MonkhorstPackGrid(
        force_timereversal=False
    )

    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,101)*eV,
        kpoints=kpoint_grid,
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=RecursionSelfEnergy(),
        )
    nlsave(filename, transmission_spectrum)
```

(continues on next page)

(continued from previous page)

```
    # ------------------------------------------------------------
    # Spin Transfer Torque
    # ------------------------------------------------------------
    kpoint_grid = MonkhorstPackGrid(
        force_timereversal=False,
        )

    spin_transfer_torque = SpinTransferTorque(
        configuration=device_configuration,
        energy=0*eV,
        kpoints=kpoint_grid,
        contributions=Left,
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=RecursionSelfEnergy(),
        )
    nlsave(filename, spin_transfer_torque)
```

The Mulliken population and transmission spectrum are not strictly needed, but will prove useful for further analysis later on. Notice that the case θ=90ř is skipped because that calculation was already done above.

Download the script, `angles.py`, and run it using the **Job Manager** – it may take up to 30 minutes to complete.

When the job finishes you have a number of new nc-files on the QuantumATK LabFloor, each for a particular spin angle. You may visualize the Mulliken populations to check for yourself that the spin angle varies correctly.

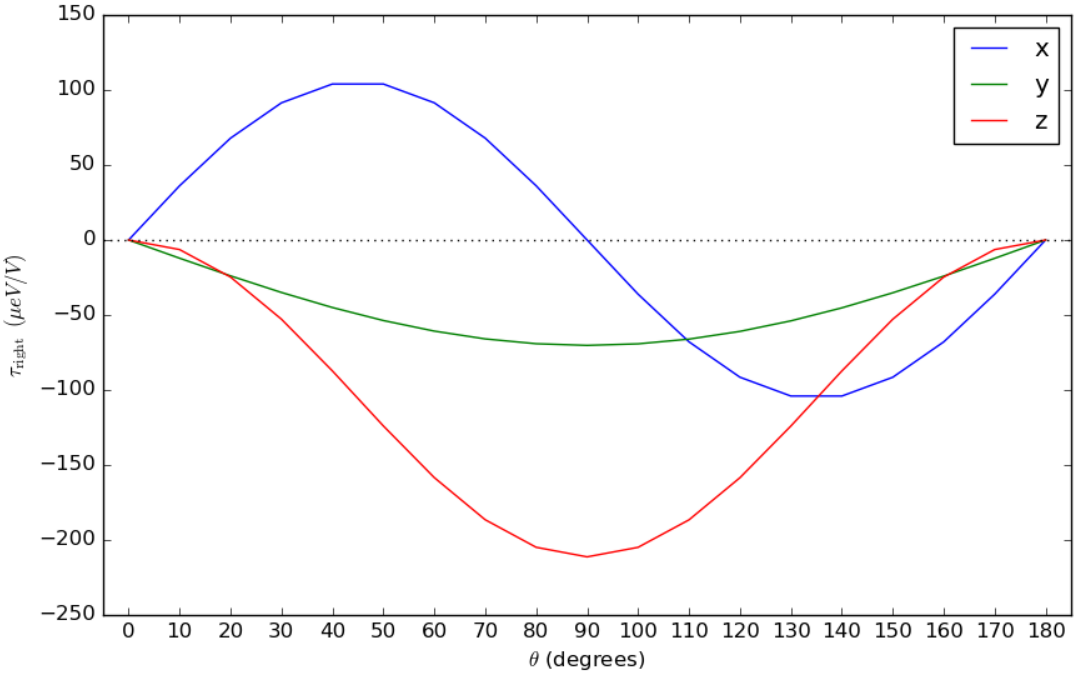
Use the script `angles_plot.py` to analyze and plot the results. For each angle, it sums the x-, y- and z-components of τ in the right-hand part of the device, and plots them against the angle:

```
# Prepare lists
thetas = [0,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,180]
x = []
y = []
z = []

# Read data
for theta in thetas:
    # Data file
    filename = 'theta-%i.hdf5' % theta

    # Read in the STT analysis object
    stt = nlread(filename, SpinTransferTorque)[0]

    # Get the STT 3D vector grid (units Bohr**-3)
    array = stt.toArray()

    # Get the index of middle position along C
    sh = numpy.shape(array)
    k = sh[2]/2

    # Get the volume element of the STT grid.
    dX, dY, dZ = stt.volumeElement()
```

(continues on next page)

(continued from previous page)

```
    volume = numpy.abs(numpy.dot(dX, numpy.cross(dY, dZ)))

    # Integrate the vector components in the right-hand part of the device
    stt_x = numpy.sum(array[:,:,k:,:,0])*volume*stt.unit()
    stt_y = numpy.sum(array[:,:,k:,:,1])*volume*stt.unit()
    stt_z = numpy.sum(array[:,:,k:,:,2])*volume*stt.unit()

    # append to lists
    x.append(stt_x)
    y.append(stt_y)
    z.append(stt_z)

# Convert lists to arrays
x = numpy.array(x)
y = numpy.array(y)
z = numpy.array(z)

# Save data for future convenience
import pickle
f = open('angles_plot.pckl', 'wb')
pickle.dump((thetas,x,y,z), f)
f.close()

# Plot results
import pylab
pylab.figure(figsize=(10,6))
pylab.plot(thetas, x*1e6, label='x')
pylab.plot(thetas, y*1e6, label='y')
pylab.plot(thetas, z*1e6, label='z')
pylab.axhline(0, color='k', linestyle=':')
pylab.legend()
pylab.xlabel(r'$\theta$ (degrees)')
pylab.ylabel(r'$\tau_\mathrm{right} \, \, \, (\mu eV/V)$')
ax = pylab.gca()
ax.set_xlim((-5,185))
ax.set_xticks(thetas)
pylab.savefig('angles_plot.png')
pylab.show()
```



The in-plane STT may also be compared to an analytic expression from Ref.[1]:

$$\tau_{\parallel}(\theta) = 0.5 \cdot [T_z(180^\circ) - T_z(0^\circ)]\, \mathbf{M}_L \times (\mathbf{M}_R \times \mathbf{M}_L),$$

where $T_z(180^\circ) = h/(4\pi e)[T_\uparrow(180^\circ) - T_\downarrow(180^\circ)]$ is the spin transmission for the anti-parallel configuration, and likewise for $T_z(0^\circ)$. The vector $\mathbf{M}_{L/R}$ is the direction of magnetization in the left/right part of the carbon chain.

Use the script `analytical.py` to do the analysis. It computes the in-plane ($\tau_\parallel = \sqrt{\tau_x^2 + \tau_z^2}$) and out-of-plane ($\tau_\perp = \tau_y$) components in the right-hand part of the device, and also computes $\tau_\parallel$ from the analytical expression using the obtained transmission spectrum at each angle:

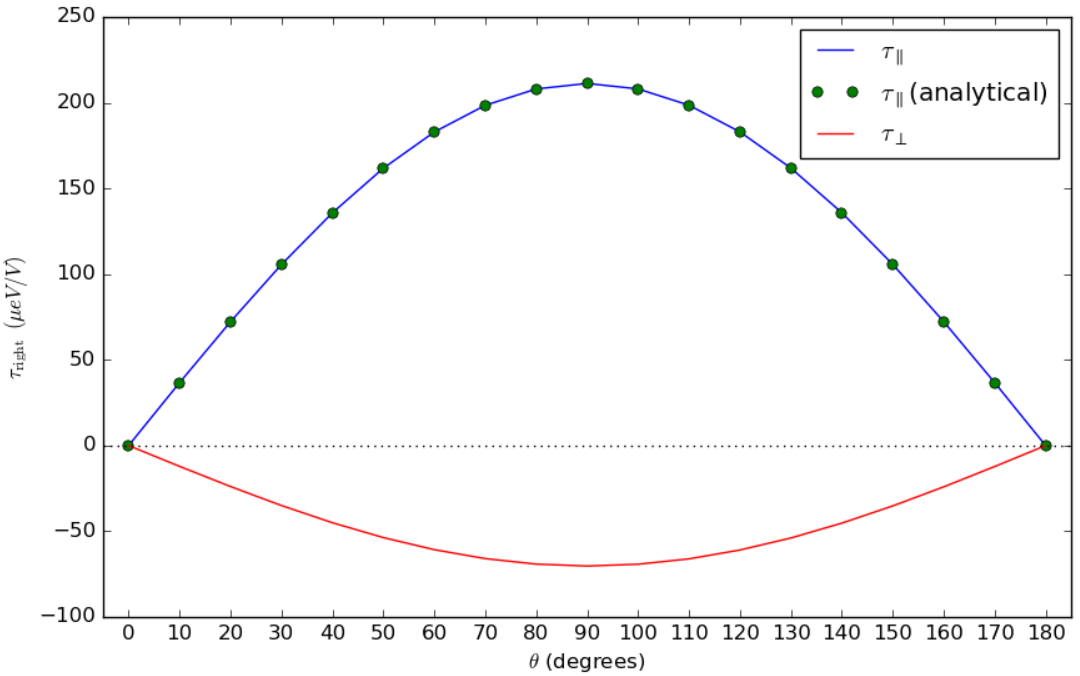
```
# Load STT data
import pickle
f = open('angles_plot.pckl', 'rb')
(thetas,x,y,z) = pickle.load(f)
f.close()

# Process STT data
inplane = (x**2 + z**2)**0.5
outplane = y

# Analytical in-plane STT
transmissions = numpy.zeros(2)
for j,theta in enumerate([0,180]):
    # Data file
    filename = 'theta-%i.hdf5' % theta
    # Read the transmission spectrum
    transmission = nlread(filename, TransmissionSpectrum)[0]
```

(continues on next page)

[1] I.ăTheodonis, N.ăKioussis, A.ăKalitsov, M.ăChshiev, and W.ăH. Butler. Anomalous Bias Dependence of Spin Torque in Magnetic Tunnel Junctions. *Phys. Rev. Lett.*, 97:237205, Dec 2006. doi:10.1103/PhysRevLett.97.237205.

(continued from previous page)

```
    # Get the energies
    energies = transmission.energies().inUnitsOf(eV)
    # Find the index of the Fermi level
    i_Ef = numpy.argmin(abs(energies))
    # Calculate the spin-transmission (Up - Down)
    T = transmission.evaluate(spin=Spin.Up)[i_Ef]-transmission.evaluate(spin=Spin.
↪Down)[i_Ef]
    # Append to list
    transmissions[j] = T
analytical = abs(transmissions[0]-transmissions[1])/2*numpy.sin(numpy.
↪array(thetas)*numpy.pi/180)/(2*numpy.pi)

# Plot results
import pylab
pylab.figure(figsize=(10,6))
pylab.plot(thetas, inplane*1e6,          label=r'$\tau_\parallel$')
pylab.plot(thetas, analytical*1e6, 'o', label=r'$\tau_\parallel$'+'(analytical)')
pylab.plot(thetas, outplane*1e6,         label=r'$\tau_\perp$')
pylab.axhline(0, color='k', linestyle=':')
pylab.legend()
pylab.xlabel(r'$\theta$ (degrees)')
pylab.ylabel(r'$\tau_\mathrm{right} \, \, \, (\mu eV/V)$')
ax = pylab.gca()
ax.set_xlim((-5,185))
ax.set_xticks(thetas)
pylab.savefig('analytical.png')
pylab.show()
```



The agreement between the angle-dependent calculation (blue line) and the analytical results (green dots), obtained

only from the spin transmissions in the parallel and anti-parallel configurations, is excellent.

References

3.10.2 Spin transport in magnetic tunnel junctions

Version: 2016.0

Downloads & Links

Transport calculations with QuantumATK `AdaptiveGrid`

`central_region.py` `mgo_para.py` `mgo_relax.py` `tmr.py` `adaptive_grid.py` `zoom.py` `mgo_nonco_dc.py` `mgo_nonco_dc_analysis.py`

Introduction

This tutorial shows you how to simulate and analyze the electronic transport properties of magnetic tunnel junctions (MTJs), commonly studied for spintronic applications. You will consider a Fe|MgO|Fe MTJ, which is a rather complex spin polarized system, initially studied by MacLaren and co-workers[1].

You will use QuantumATK to study the colinear and non-colinear spin-dependent transport properties, including electronic transmission, tunnel magnetoresistance, and spin-transfer torque. You will also become acquainted with the AdaptiveGrid method, which is an adaptive alternative to the conventional Monkhorst–Pack type grid for k-point sampling.

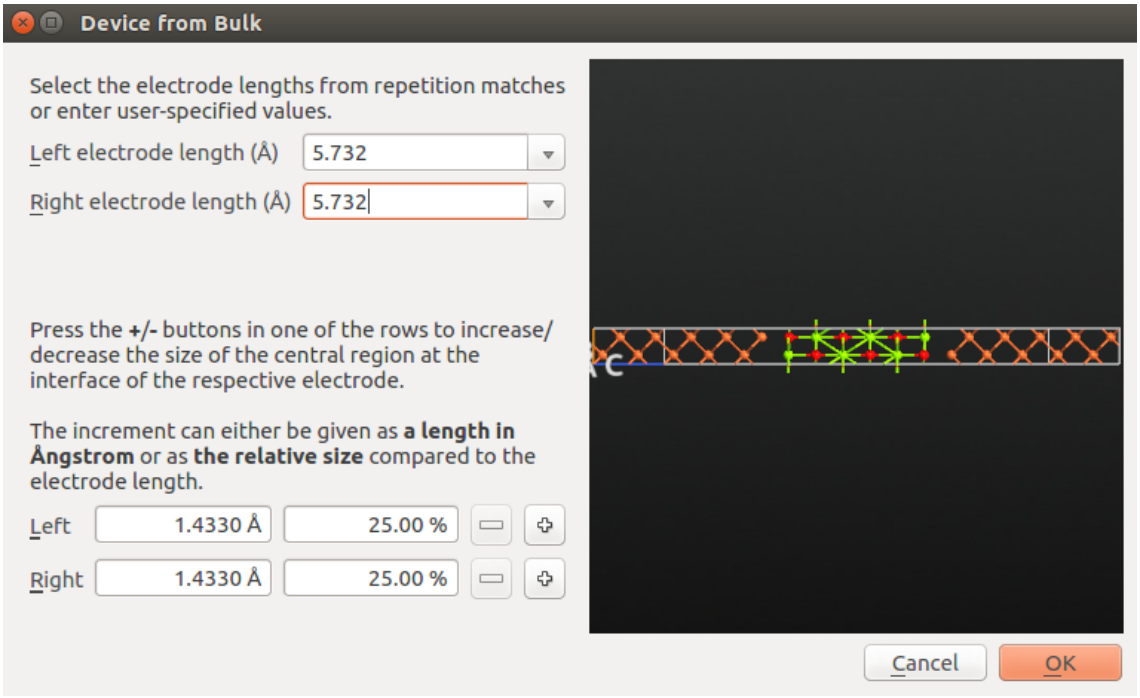
Getting started

It is very easy to create a Fe|MgO|Fe tunnel junction using the *Magnetic Tunnel Junction* Builder plugin. However, the device configuration will then need to be geometry optimized, which is not the main objective of this tutorial. We therefore provide an QuantumATK Python script with the optimized device central region, `central_region.py`, which you should use for building the MTJ device. You can also jump to section *Relaxing the device central region* if you want to learn how the geometry relaxation was done.

For now, just start QuantumATK, create a new project and give it a name, then select it and click *Open*. Download `central_region.py` and save it in the Project Folder. Then drop the script on the QuantumATK **Builder**, and use the **Device from Bulk** plugin to attach electrodes to the central region, and thereby create the Fe|MgO|Fe device.

The default electrode length of 5.732 Å will work fine for the present purposes. This corresponds to two times the iron lattice constant.

[1] W.ăH. Butler, X.-G. Zhang, T.ăC. Schulthess, and J.ăM. MacLaren. Spin-dependent tunneling conductance of Fe/MgO/Fe sandwiches. *Physical Review B*, 63(5):054416, 2001. doi:10.1103/PhysRevB.63.054416.



Note: The Device from Bulk plugin will suggest a few different electrode lengths. These are detected by searching for periodicity in your central region. Always check and carefully choose these important parameters. See the tutorial *Transport calculations with QuantumATK* for more information.

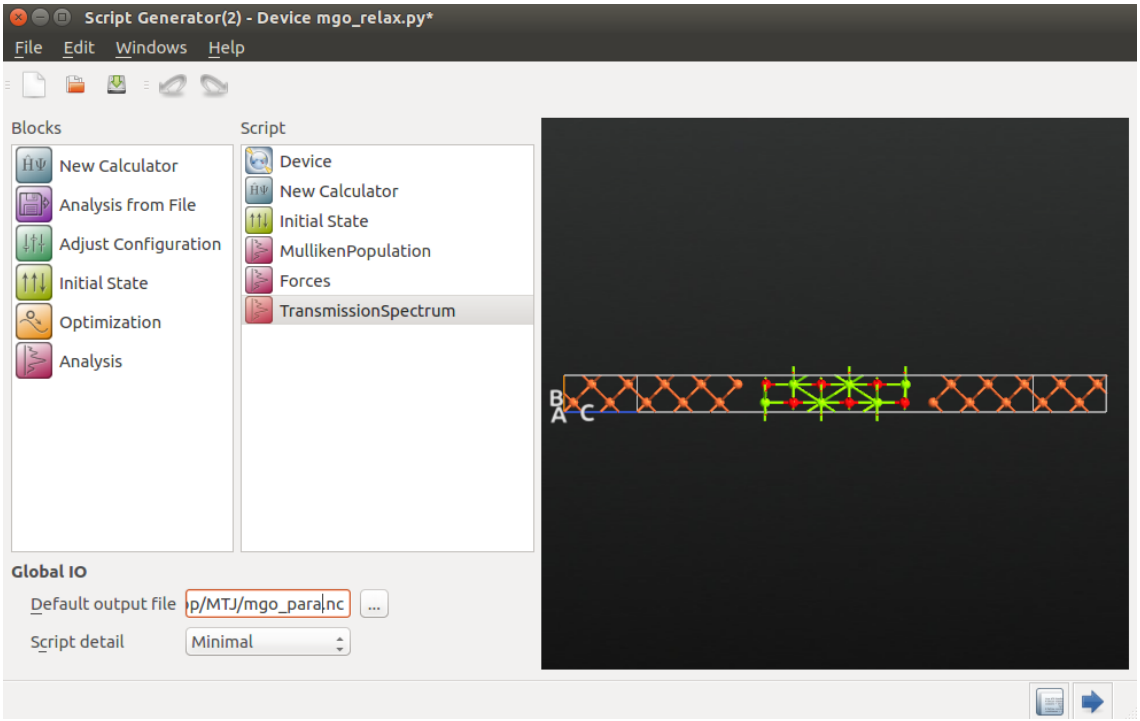
Parallel spin

You will in the following perform a self-consistent NEGF-DFT calculation for the case of parallel spin polarization in the left and right iron parts of the MTJ. You will also compute the transmission spectrum as a function of the k-vector parallel to the transport direction.

Send the device configuration to the **Script Generator** (use the icon for this), and add the following blocks to the script:

- New Calculator
- Initial State
- MullikenPopulation
- Forces
- TransmissionSpectrum

Then change the default output file name to `mgo_para.nc`.



Next, adjust the settings in a some of the script blocks:

•**Calculator**:

–Select the SGGA exchange-correlation method, to perform a spin polarized GGA calculation using the PBE functional.

–Increase the electron temperature to 1200 Kelvin, to ease the self-consistent convergence.

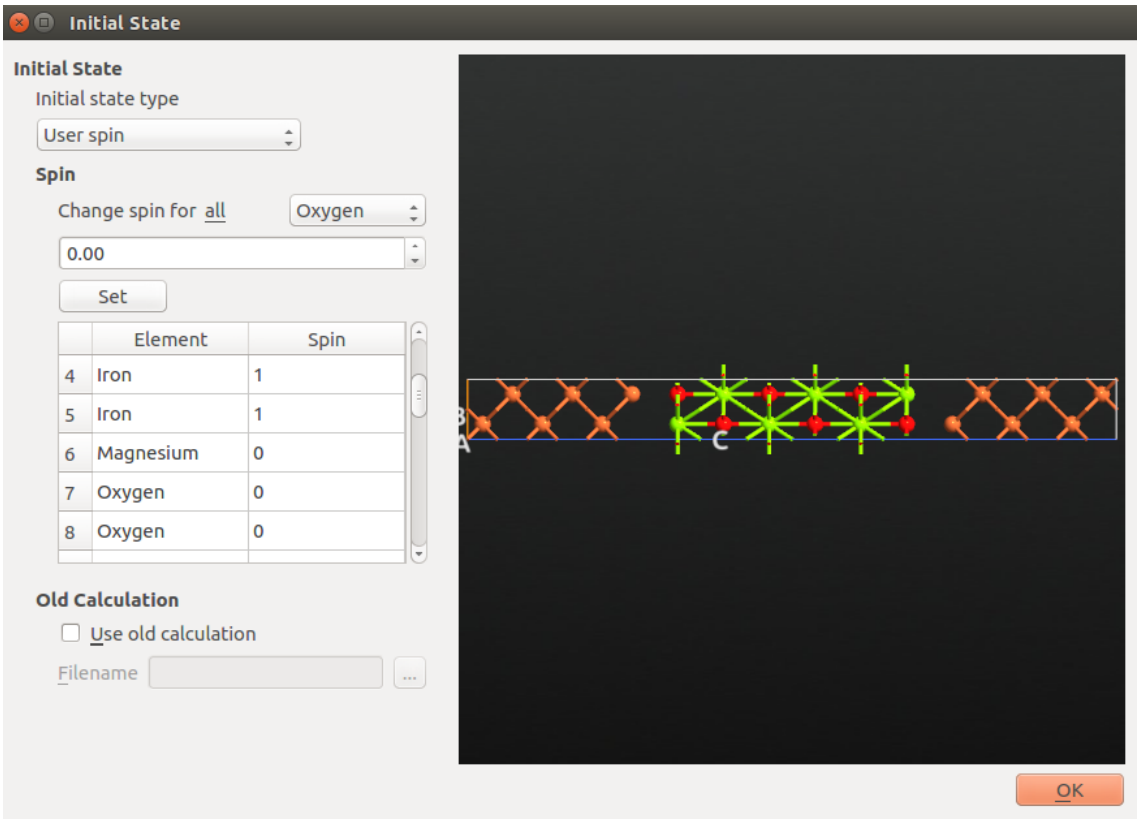
–Use a 7x7x100 k-point grid.

–Select the SingleZetaPolarized basis set for iron.

•**Initial State**:

–Choose *User spin* for as the type of initial state.

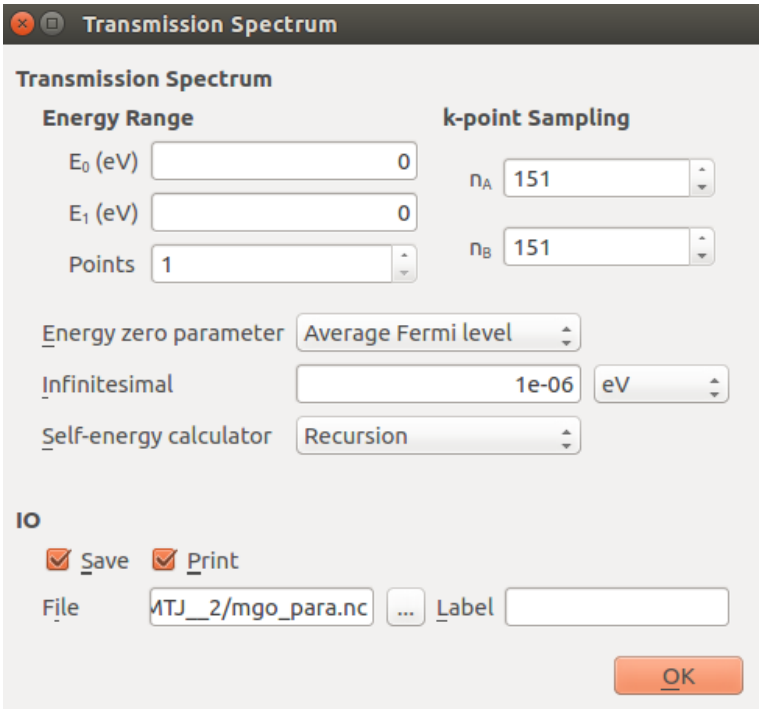
–Set the relative spins on O and Mg atoms to 0.



•**TransmissionSpectrum**:

–You will in this example compute the transmission only at the Fermi level, to save time. Use the settings $E_0 = E_1 = 0.0$ eV and just a single energy point to achieve this.

–151x151 grid for k-point sampling.

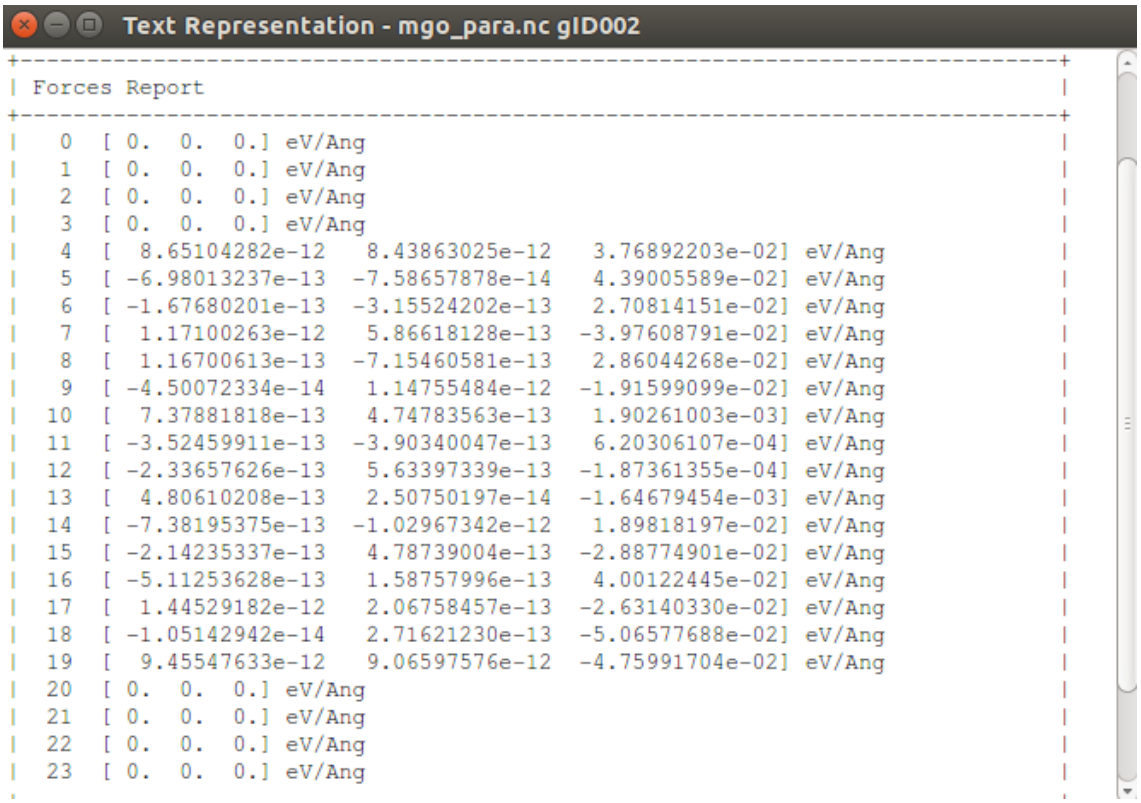


Save the script as `mgo_para.py` and run the calculation, e.g. using the **Job Manager**. The calculations may take an hour on a single core, but it is possible to obtain a very good speedup by executing in parallel, reducing the computational time down to roughly 20 minutes with 4 MPI processes. If needed you can also download the script: `mgo_para.py`.

Important: Do not close the Script Generator window. You will need it again later on for setting up calculations for the anti-parallel spin configuration.

Inspecting the forces

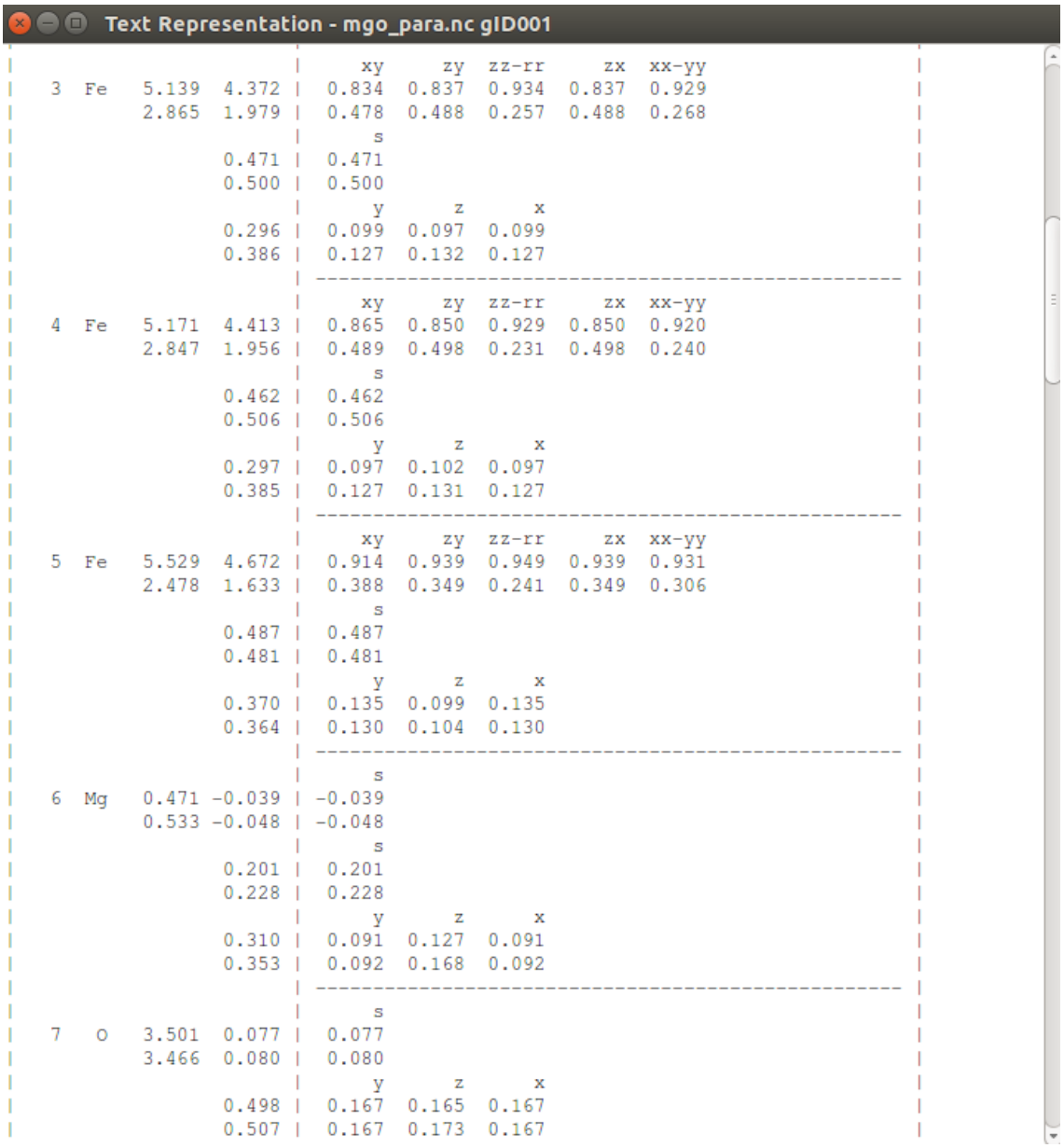
The output data file `mgo_para.nc` should now have appeared in the Project Files list. Make sure it is selected, such that the data items are visible on the LabFloor. Then select the **Forces** item and use the **Text Representation** tool in the right-hand Panel Bar to see a report of the forces:



The forces on the atoms belonging to the electrode extension regions are not calculated, and these are therefore set to zero. For the other atoms, the forces are small.

Mulliken population

You can also use the Text Representation tool to inspect the calculated **Mulliken populations**:



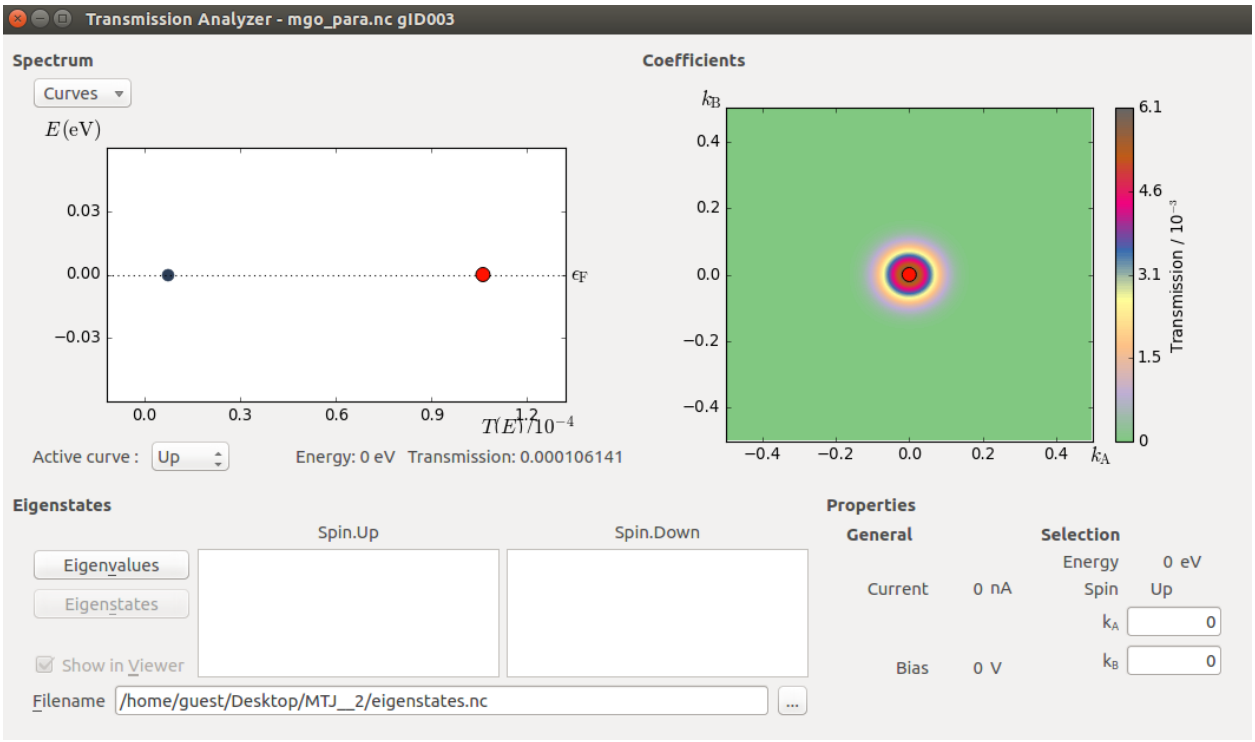
The leftmost column lists the total Mulliken population per spin channel for each atom. It is important to inspect this result in order to check that each atom is correctly spin polarized. Note that all Fe atoms are polarized with their majority spin pointing up, while the O atoms are essentially unpolarized. The Mg atoms next to the interfaces (on both sides) couple anti-parallel with the Fe atoms, while the Mg atoms further away from the interfaces are unpolarized.

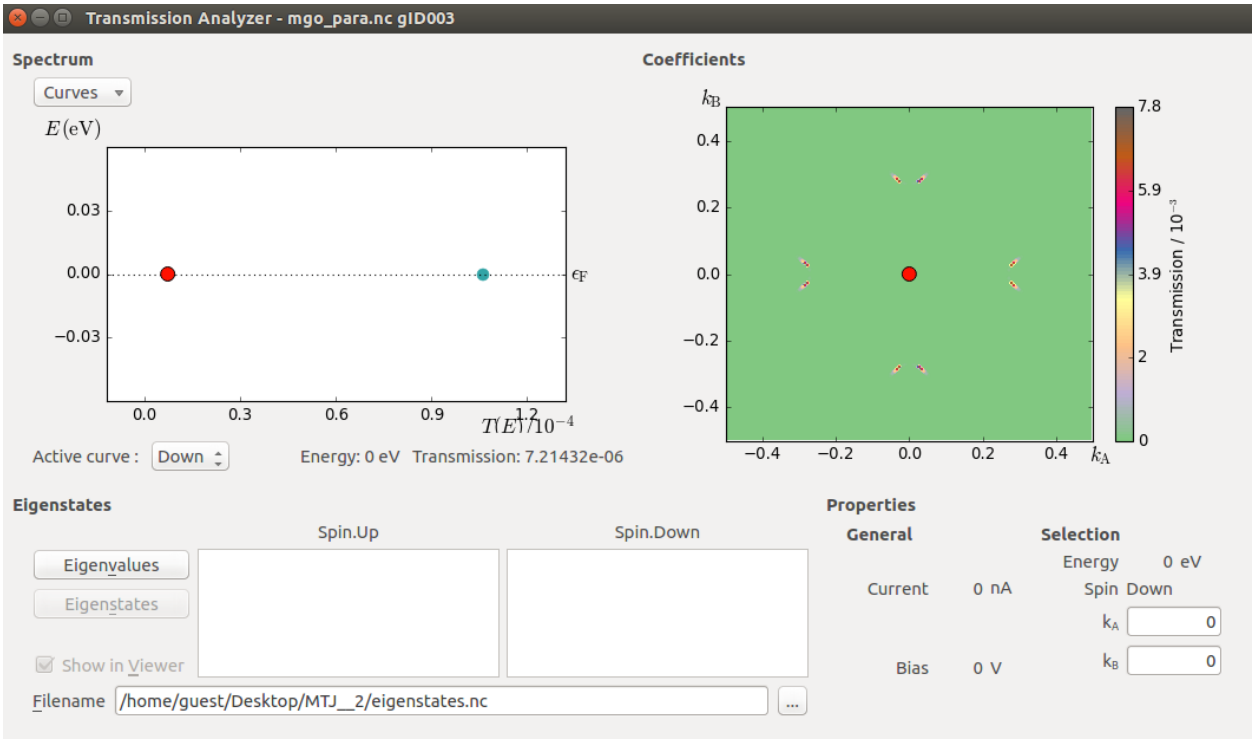
The report also shows the population for each shell and resolves it on individual orbitals.

Electron transmission

Use the **Transmission Analyzer** plugin to analyze the calculated transmission spectrum. A few modifications of the plotting options make this spectrum easier to analyze:

- Right-click the right-hand plot of the k-dependent transmission coefficients, and select the color map named *Spectral*.
- In the **Curves** menu, tick off Up and Down.
- Using the **Active Spin** menu, you can now choose between the k-dependent transmission for spin-up and spin-down electrons:



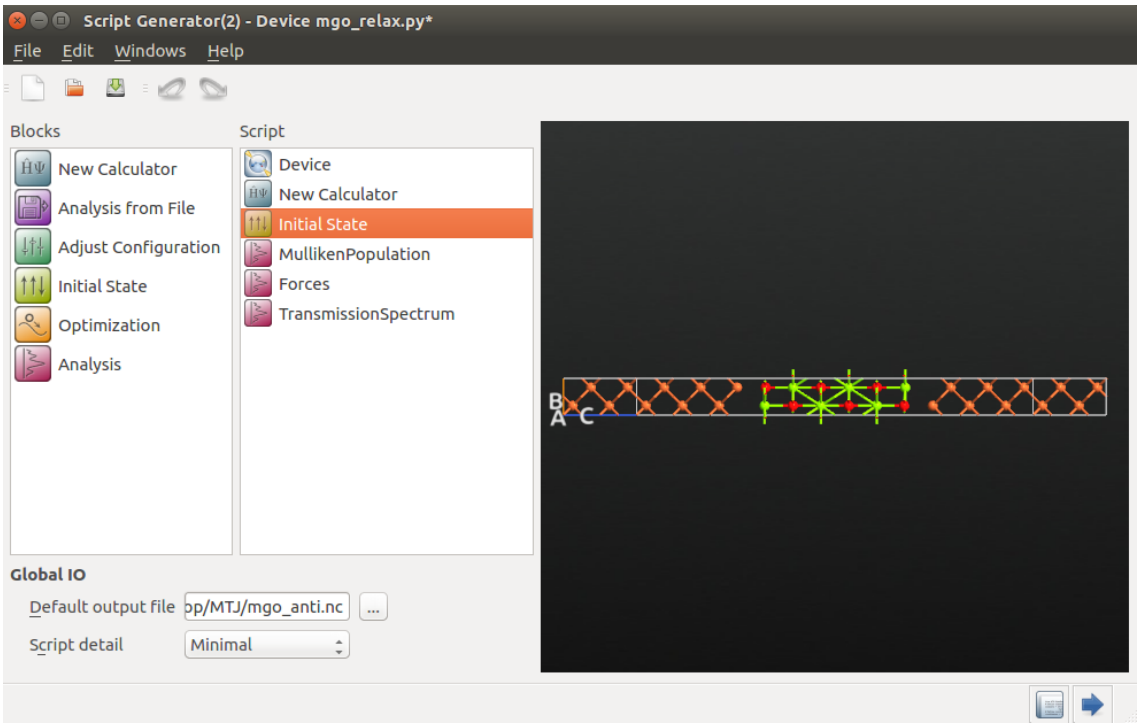


As for the results of the calculation, it is clear from the plots above that the spin-down (minority) transmission is strongly suppressed in magnitude, and exhibits spikes at certain points in the Brillouin zone, corresponding to resonant tunneling, whereas the spin-up (majority) transmission indicates regular barrier tunneling. This is all in agreement with the accepted picture of the tunneling mechanism in the Fe|MgO|Fe tunneling junction (compare to Figures 6 and 10 in[Page 1358, 1]).

Anti-parallel spin

We now turn to the case of anti-parallel alignment of the spins in the left and right iron parts of the Fe|MgO|Fe junction. The self-consistent state of the parallel configuration computed above will be used as an initial guess for the anti-parallel state. This speeds up SCF convergence as compared to starting the anti-parallel calculation completely from scratch.

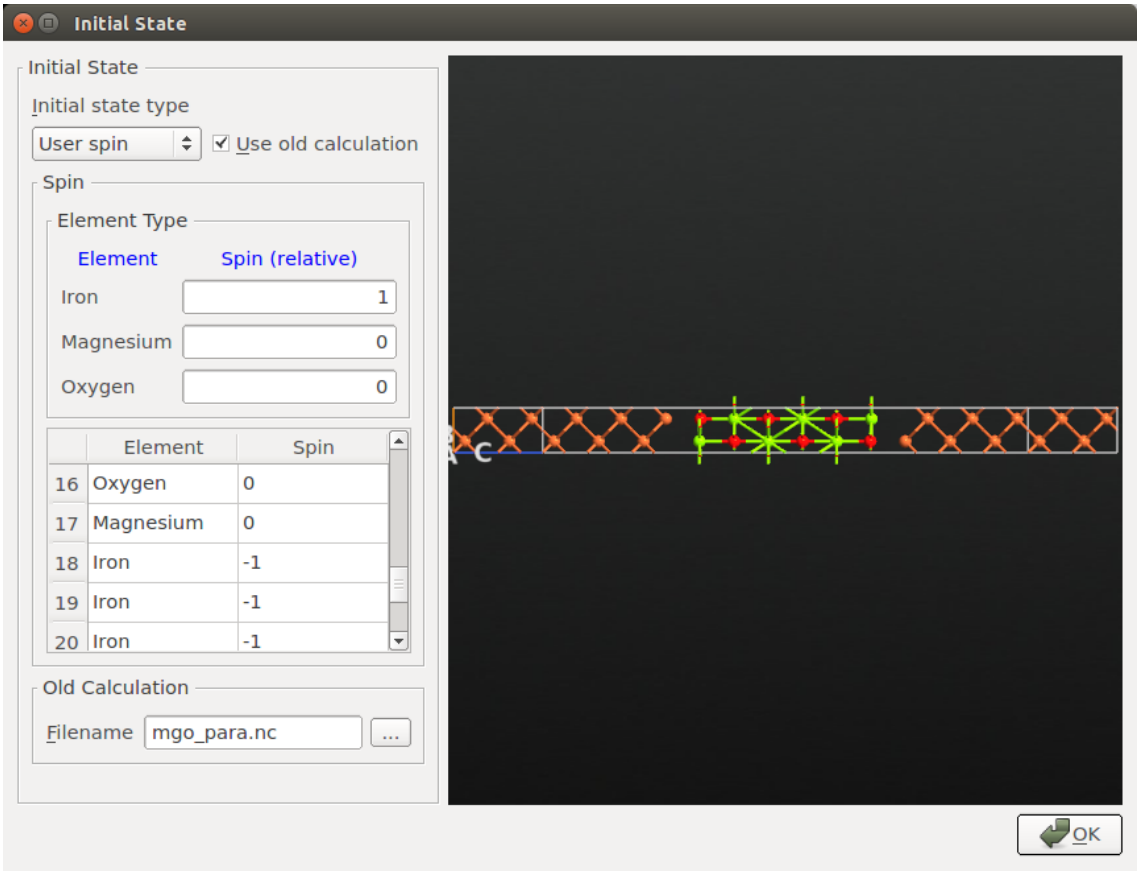
Return to the **Script Generator**. If you did not close it after setting up the parallel calculation, only a few modifications are needed. Otherwise, drop the script `mgo_para.py` on the Scripter and start by setting up the script identically to what you did above for the parallel calculations.



Setting the initial state

Open the **Initial State** block, and do the following to set up the anti-parallel spin configuration and use the parallel calculation as an initial guess for the device density matrix:

- Set again the relative spin for all iron atoms to 1 and set it to 0 for magnesium and oxygen atoms.
- Change the initial spin to -1 for all iron atoms on the right-hand side of the junction.
- Check the *Use old calculation* tick box and enter `mgo_para.nc` as the filename from which the old calculation should be loaded from.



Note: The method ouitlined above works like this: The converged density matrix from the parallel-spin calculation is used as initial state. However, the spin densities for atoms in the right-hand side of the device are flipped, resulting in an anti-parallel initial density matrix for the full device.

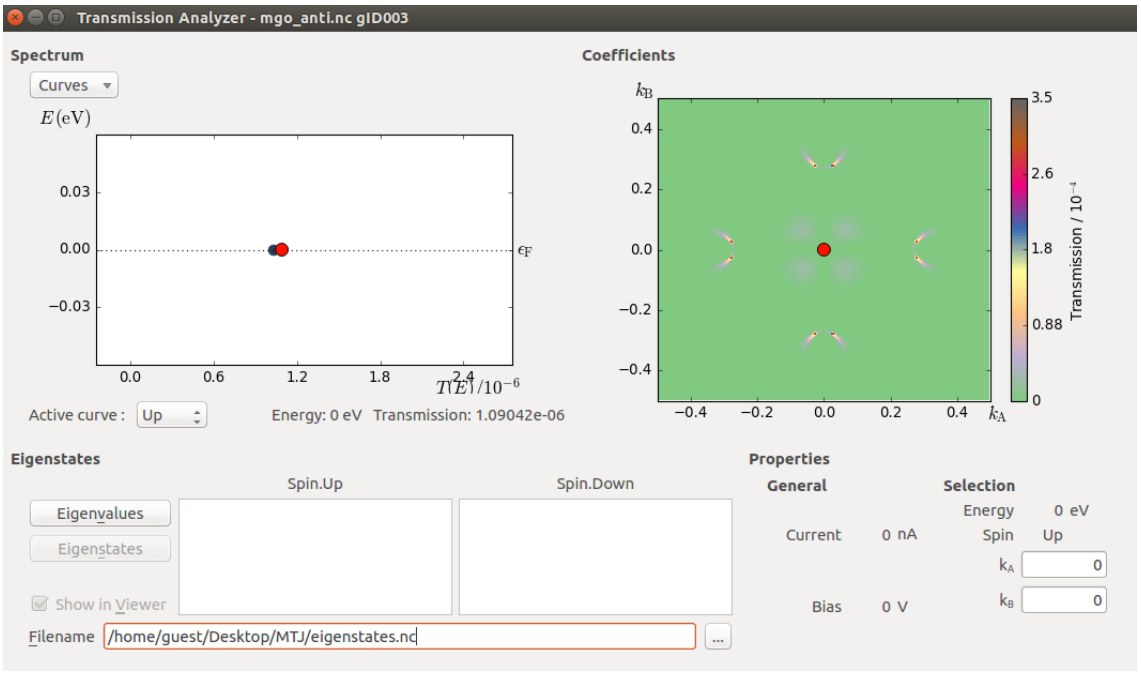
Finally, change the default output file name to `mgo_anti.nc` and make sure the **TransmissionSpectrum** analysis block is identical to the one used for the spin-parallel case.

Save the QuantumATK Python script as `mgo_anti.py` and run it.

Analyzing the results

Check that the forces on each atom are still negligible. Also check the Mulliken populations to ensure that each atom has the expected spin polarization.

Use again the **Transmission Analyzer** to analyze the electronic transmission. In this case, the transmission for up and down (majority and minority) spins are identical because of the mirror symmetry of the system. We find again that the k-dpendent transmission coefficients exhibit highly localized peaks surrounded by regions of zero transmission.



To understand why the transmission is identical for spin-up and -down, first note that due to time inversion symmetry, propagation from left to right is always identical to propagation from right to left. The up component of the transmission spectrum corresponds to up electrons from the left electrode propagation into the right electrode. The down component of the transmission spectrum corresponds to down electrons in the right electrode propagating into the left electrode.

Due to the anti-symmetric spin, the up electrons in the left electrode and the down electrons in the right electrode are both majority channels. Because of the mirror symmetry of the device, the propagation of the left majority channels into the right minority channels (the first transmission coefficient spin component) and the propagation of the right majority channels into the left minority channels (the second transmission coefficient spin component) are the same.

Thus, for symmetric devices, the equivalence of the two spin channels is an important check for anti-symmetric calculations at zero bias.

Tunneling magnetoresistance

The tunneling magnetoresistance (TMR) is defined as

$$\mathrm{TMR} = \frac{G_P - G_{AP}}{G_{AP}},$$

where G_P is the conductance through the junction with parallel spin alignment and G_{AP} the conductance for anti-parallel spin alignment.

The conductances can be calculated from their respective transmission spectra. The script below performs this calculation. You can download it here: `tmr.py`.

```
# Calculate conductance for parallel spin
transmission_para = nlread('mgo_para.hdf5', TransmissionSpectrum)[0]
conductance_para_uu = transmission_para.conductance(spin=Spin.Up)
conductance_para_dd = transmission_para.conductance(spin=Spin.Down)
conductance_para = conductance_para_uu + conductance_para_dd

# Calculate conductance for anti-parallel spin
```

(continues on next page)

(continued from previous page)

```
transmission_anti = nlread('mgo_anti.hdf5', TransmissionSpectrum)[0]
conductance_anti_uu = transmission_anti.conductance(spin=Spin.Up)
conductance_anti_dd = transmission_anti.conductance(spin=Spin.Down)
conductance_anti = conductance_anti_uu + conductance_anti_dd

print('Conductance Parallel Spin (Siemens)')
print('Up=%8.2e, Down=%8.2e' % (conductance_para_uu.inUnitsOf(Siemens),
                                conductance_para_dd.inUnitsOf(Siemens)))
print('Total = %8.2e' % (conductance_para.inUnitsOf(Siemens)))
print()

print('Conductance Anti-Parallel Spin (Siemens)')
print('Up=%8.2e, Down=%8.2e' % (conductance_anti_uu.inUnitsOf(Siemens),
                                conductance_anti_dd.inUnitsOf(Siemens)))
print('Total = %8.2e' % (conductance_anti.inUnitsOf(Siemens)))
print()

print('TMR (optimistic)  = %8.2f percent' % \
      (100.*(conductance_para-conductance_anti)/conductance_anti))
print('TMR (pessimistic) = %8.2f percent' % \
      (100.*(conductance_para-conductance_anti)/(conductance_para+conductance_anti)))
```

Execute the script, and find the TMR:

```
Conductance Parallel Spin (Siemens)
Up=4.11e-09, Down=2.79e-10
Total = 4.39e-09

Conductance Anti-Parallel Spin (Siemens)
Up=4.22e-11, Down=4.00e-11
Total = 8.22e-11

TMR (optimistic)  =  5240.12 percent
TMR (pessimistic) =    96.32 percent
```

Adaptive k-point grid

The **AdaptiveGrid** class implements an alternative to the traditional Monkhorst–Pack (MP) grid for k-point sampling. As described in great detail in the QuantumATK Reference Manual entry for *AdaptiveGrid*, an adaptive k-point grid can be used to automatically zoom in on the significant features of the electron transmission spectrum. This is particularly useful when the k-dependent transmission is dominated by localized peaks, like for the Fe|MgO|Fe MTJ, where the total computed transmission may depend critically on resolving those peaks.

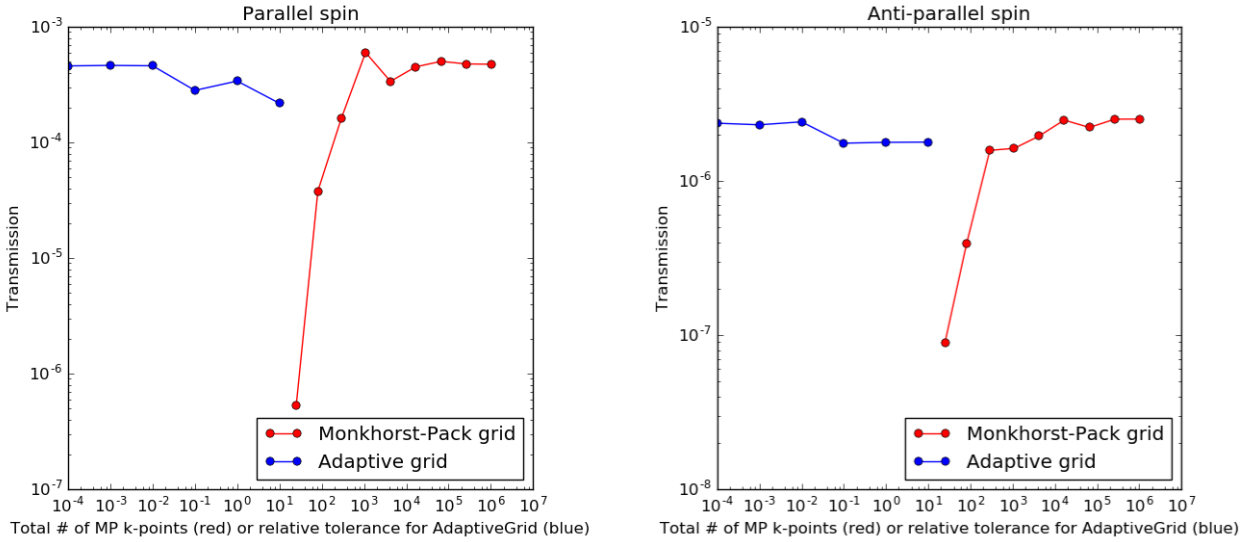
The adaptive algorithm essentially divides the Brillouin zone (BZ) into triangles and integrates over each triangle. Each triangle becomes increasingly small in an iterative manner until convergence is reached, but stops if the number of refinement levels reach the `maximum_number_of_levels`, which is 20 by default. Two types of error measures are available:

Absolute
Converged when the *absolute* value of the BZ integral in all triangles is lower than the chosen **tolerance**.

Relative
Converged when the *relative* value of the BZ integral in all triangles is lower than the chosen **tolerance**.

Note: The AdaptiveGrid functionality is available from QuantumATK 2016 an onwards.

The figures below show how the total transmission through the Fe|MgO|Fe MTJ depends on the k-point resolution used for sampling the k-dependent transmission function for the parallel and anti-parallel cases. Results using the MP k-point grid are compared to those from using the adaptive k-point grid.



Well converged calculations of the total transmission through the MTJ considered here are obtained with no less than 10^3 Monkhorst–Pack k-points, roughly corresponding to a 31x31 grid. With the adaptive k-point grid, a **relative tolerance** of 10^{-2} seems to be quite sufficient.

Note: The computational load for producing the data shown in the figures above increase from right to left with AdaptiveGrid (blue curves), and from left to right with MonkhorstPack (red curves).

Parallel transmission

The script `adaptive_grid.py` calculates the transmission for the Fe|MgO|Fe device with parallel spin orientation using both a dense MP grid and the AdaptiveGrid method:

```
# ------------------------------------------------------------
# Load device configuration
# ------------------------------------------------------------
device_configuration = nlread('mgo_para.nc', DeviceConfiguration)[0]

# ------------------------------------------------------------
# Transmission Spectrum using a dense Monkhorst-Pack grid.
# ------------------------------------------------------------
transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(0,0,1)*eV,
    kpoints=MonkhorstPackGrid(500,500),
    )
```

(continues on next page)

(continued from previous page)

```
nlsave('adaptive_grid.nc', transmission_spectrum)

# ------------------------------------------------------------
# AdaptiveGrid.
# ------------------------------------------------------------
adaptive_grid = AdaptiveGrid(
    tolerance=0.01,
    error_measure=Relative)

# ------------------------------------------------------------
# Transmission Spectrum using the AdaptiveGrid.
# ------------------------------------------------------------
transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(0,0,1)*eV,
    kpoints=adaptive_grid,
    )
nlsave('adaptive_grid.nc', transmission_spectrum)
```

The Monkhorst–Pack grid has 500x500 k-points, while the adaptive grid is set up with a relative tolerance of 10^{-2}. Download and run the script. It should finish in roughly 25 minutes if executed in parallel with 4 MPI processes on a modern laptop.

Spin-up transmission

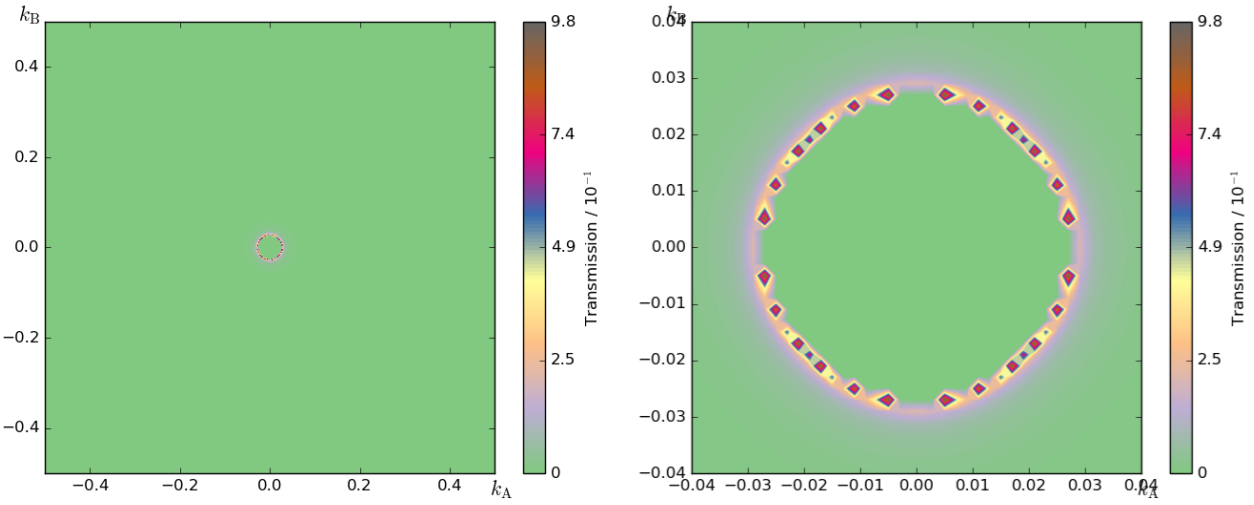


Fig. 3.103: Spin-up transmission spectrum evaluated using a 500x500 Monkhorst–Pack grid. The total k-point averaged spin-up transmission is 4.6 $\cdot 10^{-4}$. Left: Full BZ view. Right: Zoom-in around the point.

The spin-up spectrum obtained with the dense Monkhorst–Pack grid is different from the one obtained in section *Electron transmission* using a less dense grid: We now see that the feature in the middle of the spectrum is actually circular, with zero transmission immediately around the point. Clearly, a very dense grid is needed to resolve this fact. Note, however, that the improved k-point sampling has only increased the total k-point averaged spin-up transmission by a factor of 4.

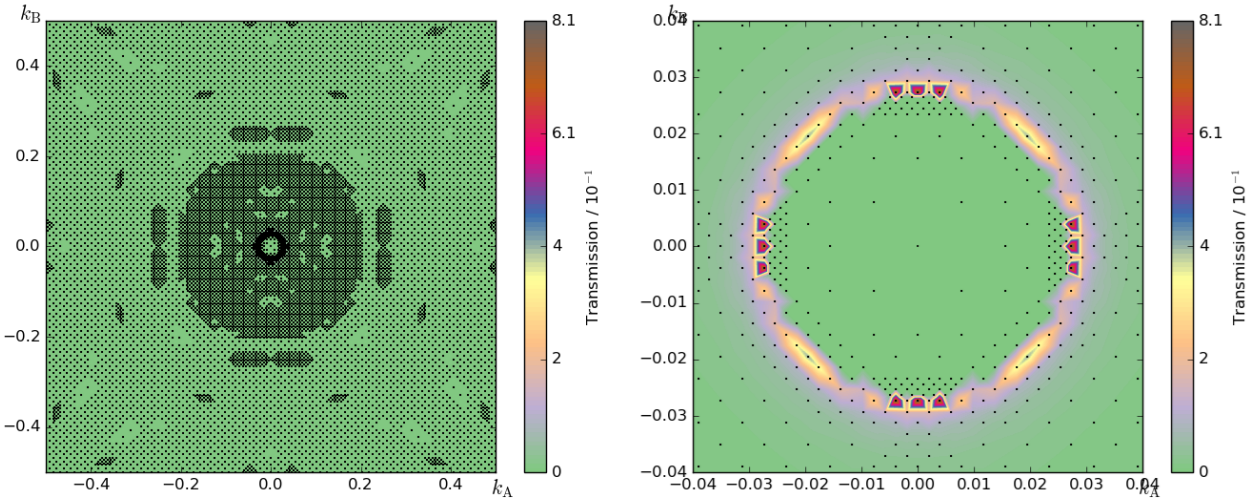


Fig. 3.104: Spin-up transmission spectrum evaluated using an adaptive grid using a relative tolerance of 10^{-2}. The black dots indicate the sampled k-points. The total k-point averaged spin-up transmission is $4.2 \cdot 10^{-4}$. Left: Full BZ view. Right: Zoom-in around the point.

The spin-up transmission spectrum evaluated using an adaptive grid is shown in the figure above, where the black dots indicate the sampled k-points. The adaptive algorithm has clearly zoomed in on the circular feature of the k-dependent transmission, causing the k-point density to be largest around the circular transmission feature.

Spin-down transmission

The 4 peaks in the densely sampled spin-down transmission are hardly visible before zooming in on one of them. Comparing the transmission obtained using the MP grid (above) to that obtained using the adaptive grid (below), it is again clear that the adaptive algorithm has zoomed in on the transmission peaks. However, the total k-point averaged spin-down transmission has changed only 5%.

Restricting the grid range

The AdaptiveGrid class can also be used to resolve only a specific part of the k-dependent transmission. The script shown below (`zoom.py`) gives an example of this, where the adaptive grid is restricted to specific k_A, k_B ranges, and thereby zooms in on specific features of the spin-up and spin-down k-dependent transmissions, respectively. A relative tolerance of 10^{-3} is used to fully resolve the transmission features.

The resulting spin-up and spin-down transmissions are illustrated below. Note the different scales on the figure axes.

```
# ------------------------------------------------------------
# Load device configuration
# ------------------------------------------------------------
device_configuration = nlread('mgo_para.nc', DeviceConfiguration)[0]

# ------------------------------------------------------------
# Transmission Spectrum zoom #1.
# ------------------------------------------------------------
adaptive_grid = AdaptiveGrid(
```

(continues on next page)

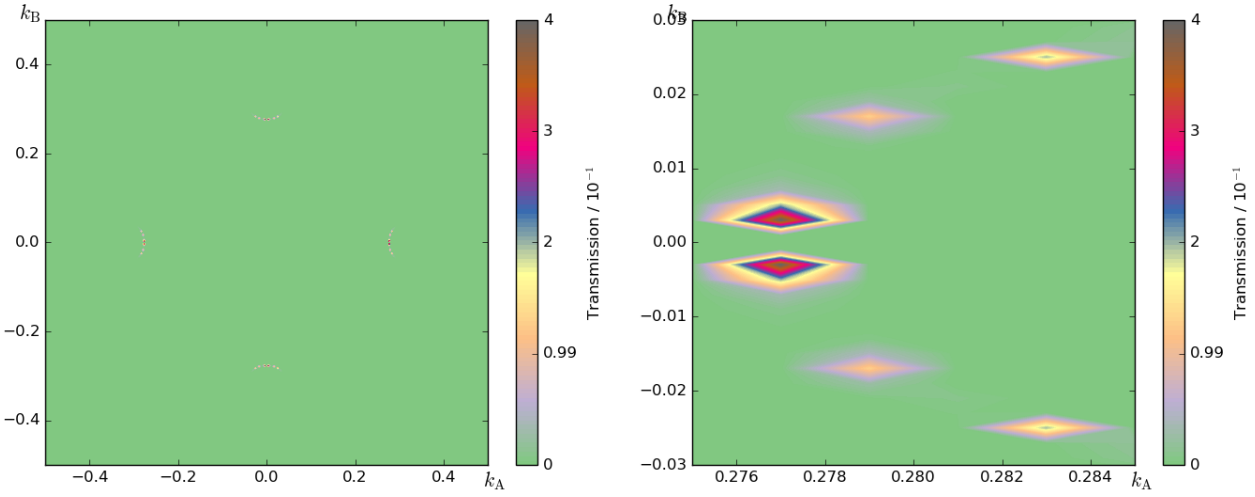


Fig. 3.105: Spin-down transmission spectrum evaluated using a 500x500 Monkhorst–Pack grid. The total k-point averaged spin-down transmission is 3.7 $\cdot 10^{-5}$. Left: Full BZ view. Right: Zoom-in around the right-hand transmission peak.

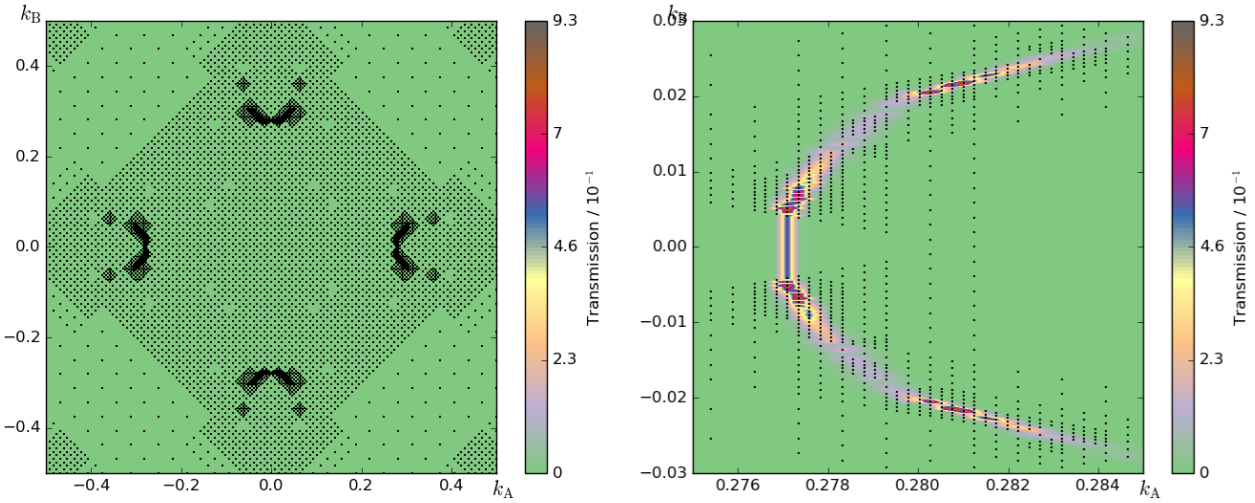
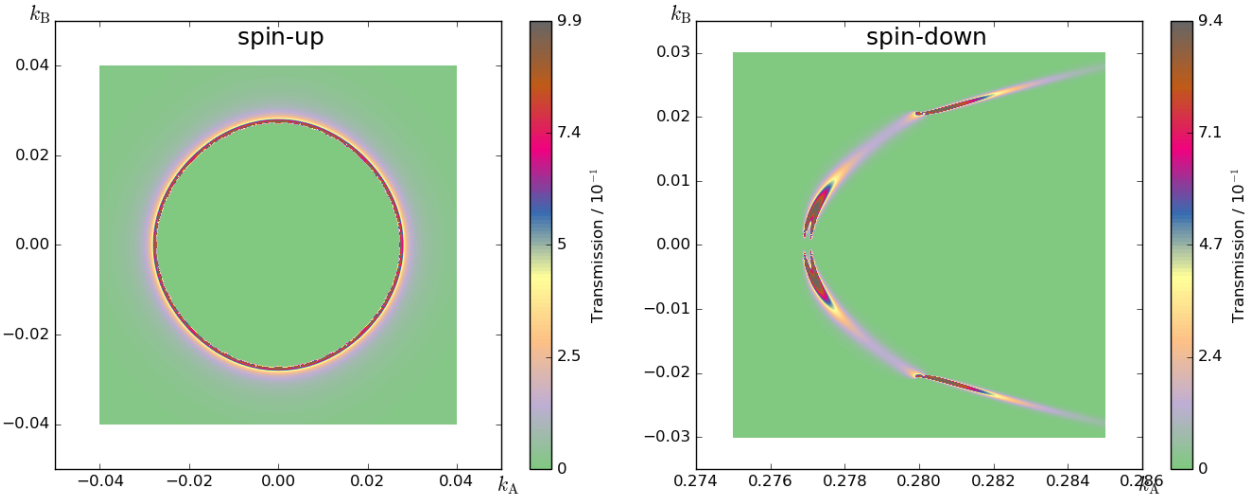


Fig. 3.106: Spin-down transmission spectrum evaluated using an adaptive grid using a relative tolerance of 10^{-2}. The black dots indicate the sampled k-points. The total k-point averaged spin-down transmission is 3.5 $\cdot 10^{-5}$. Left: Full BZ view. Right: Zoom-in around the right-hand transmission peak.

(continued from previous page)

```
    kA_range=[-0.04, 0.04],
    kB_range=[-0.04, 0.04],
    tolerance=0.001,
    error_measure=Relative)

transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(0,0,1)*eV,
    kpoints=adaptive_grid,
    )
nlsave('zoom.nc', transmission_spectrum)

# ------------------------------------------------------------
# Transmission Spectrum zoom #2.
# ------------------------------------------------------------
adaptive_grid = AdaptiveGrid(
    kA_range=[0.275, 0.285],
    kB_range=[-0.03, 0.03],
    tolerance=0.001,
    error_measure=Relative)

transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(0,0,1)*eV,
    kpoints=adaptive_grid,
    )
nlsave('zoom.nc', transmission_spectrum)
```



Warning: The total (k-point averaged) transmission is most likely not correct if you choose to sample only a specific part of the Brillouin zone. Use full-zone sampling to compute the total transmission through the device, which is also the default setting.

Spin-transfer torque

You will in this section compute the spin transfer torque (STT), which requires non-colinear spin. The tutorial *Introduction to noncollinear spin* gives for more information on how to perform STT calculations and non-colinear spin.

In the following, you will:

1. calculate the non-colinear device configuration;
2. calculate the Mulliken population and spin transfer torque.

Computing the non-colinear device configuration

The script `mgo_nonco_dc.py` calculates the non-colinear ground state of the device configuration by using the colinear ferromagnetic calculation as starting point and a density mixing scheme which diagonalizes the density matrix before mixing it. Please, note that this script replaces the spin polarized exchage-correlation (SGGA.PBE) by the non-colinear one (NCGGA.PBE).

```
# Read in the collinear calculation
device_configuration = nlread('mgo_para.nc', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )
# Get the calculator
calculator = device_configuration.calculator()
new_calculator = calculator(
       exchange_correlation=NCGGA.PBE,
       iteration_control_parameters = iteration_control_parameters
       )
# Define the spin rotation
theta = 120*Degrees
left_spins = [(i, 1, 0*Degrees, 0*Degrees) for i in range(3)]
center_spins = [(i+3, 1, theta*i/5, 0*Degrees) for i in range(6)]
right_spins = [(i+9, 1, theta, 0*Degrees) for i in range(3)]
spin_list = left_spins+center_spins+right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

# Setup the initial state
device_configuration.setCalculator(
    calculator=new_calculator,
    initial_spin=initial_spin,
    initial_state=device_configuration)

# Calculate and save
device_configuration.update()
nlsave("mgo_nonco_dc.nc", device_configuration)
```

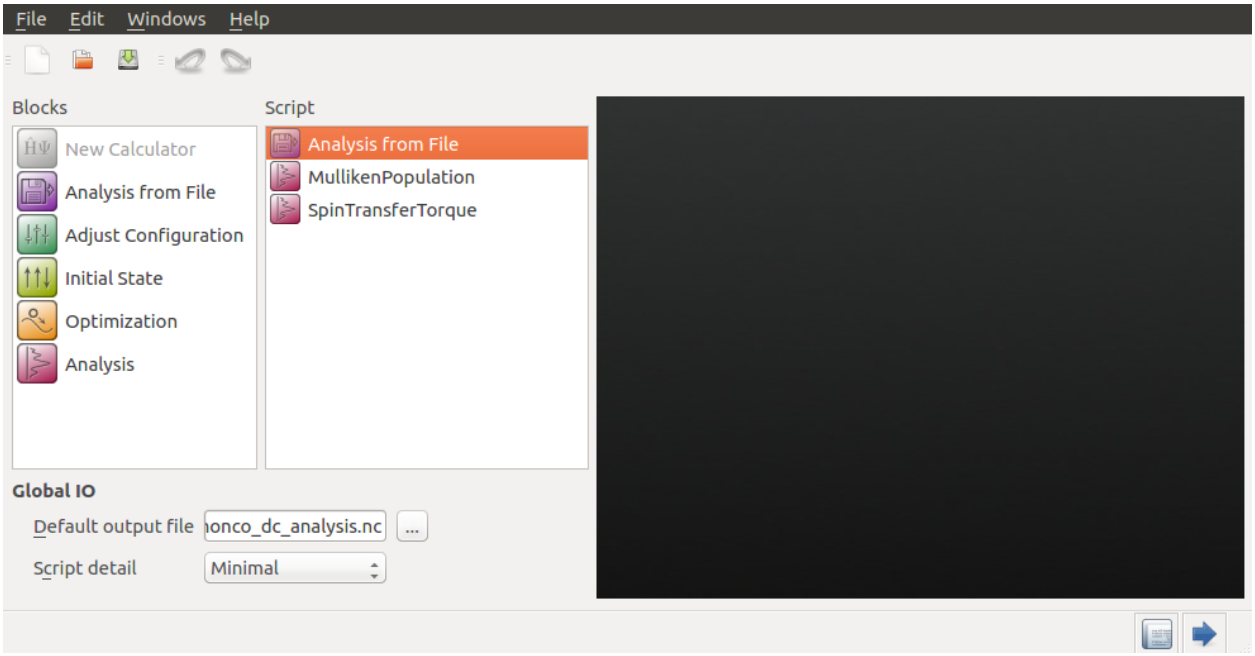
The spin setup corresponds to the spin polarization of the atoms in the left electrode pointing along the transport axis Z, while in the right electrode the polarization is rotated 120 degrees. In the central region, the angle is interpolated between these two values. Note that this is just the initial spin configuration – the actual spin polarization vectors will be computed selfconsistently and may therefore change (you can see the results in the next section).

Calculate the Mulliken population and spin transfer torque

Once the calculation is finished you can find the non-colinear device configuration in the LabFloor. In order to calculate the Mulliken population and spin transfer torque, go to the **Script Generator** and add the following blocks to the script:

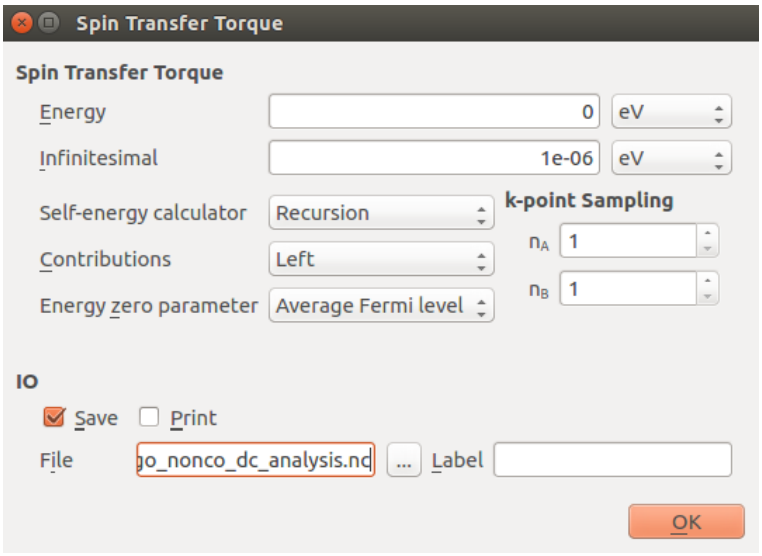
- Analysis from File
- MullikenPopulation
- SpinTransferTorque

Then, change the default output file name to `mgo_nonco_dc_analysis.nc`.



Adjust the settings in some of the blocks:

- **Analysis From File**:
  - select the `mgo_nonco_dc.nc` file.
- **SpinTransferTorque**:
  - Select **Left** for the contributions. This way, you will calculate the **left –> right** linear response current.

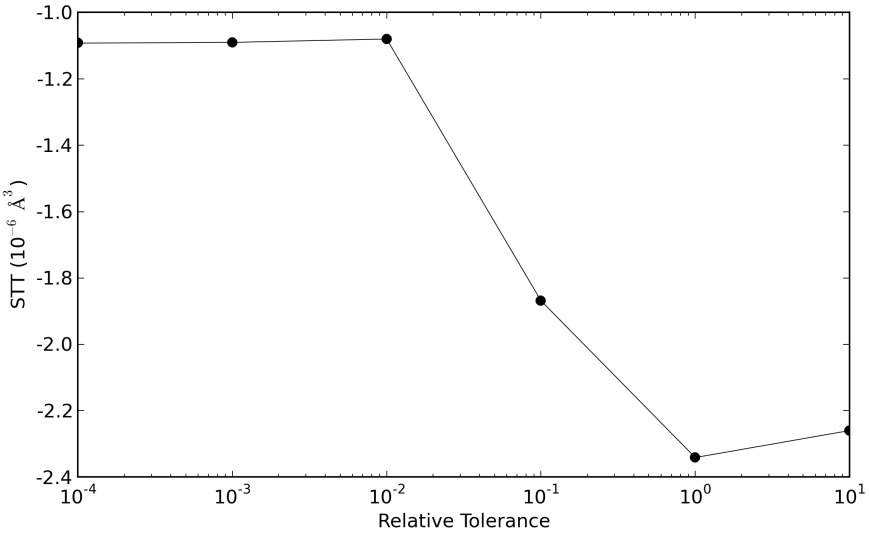


Send the script to the editor using the send to button (the icon) and replace the Monkhorst-Pack grid with the Adaptive grid as shown below.

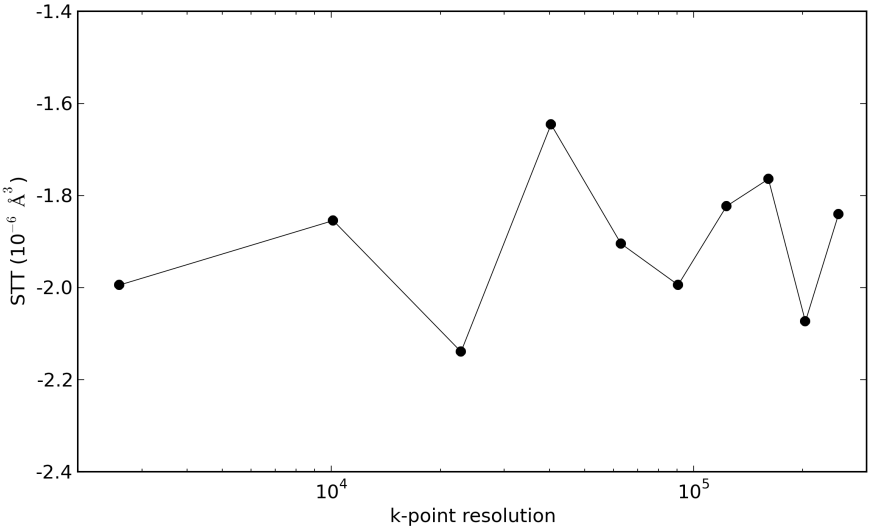
```
# Setup adaptive grid object.
adaptive_grid = AdaptiveGrid(
        kA_range=[-0.5, 0.5],
        kB_range=[-0.5, 0.5],
        tolerance=1e-2,
        error_measure=Relative,
        )
# ------------------------------------------------------------
# Spin Transfer Torque
# ------------------------------------------------------------
spin_transfer_torque = SpinTransferTorque(
    configuration=device_configuration,
    energy=0*eV,
    kpoints=adaptive_grid,
    contributions=Left,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )
nlsave('mgo_nonco_dc_analysis.nc', spin_transfer_torque)
```

If you wish, you can download the script here (`mgo_nonco_dc_analysis.py`).

The STT should be computed using a k-point sampling that is dense enough to get a well-converged value for the total torque on the right-hand side of the device. In this script we are using the adaptive grid with a relative tolerance of 10^{-2}. The below figure shows the total spin transfer torque transfered at the right side through Fe|MgO|Fe MTJ for different relative tolerances.



This plot shows that the use of a relative tolerance of 10^{-2} seems to be sufficient to obtain converged calculations of the MTJ considered in this tutorial. We have also computed the total spintransfertorque at the right side using a increasingly denser Monkhorst-Pack grid.



It can be seen that using a Monkhorst-Pack grid the total STT does not reach full convergece.

Analysis of the results

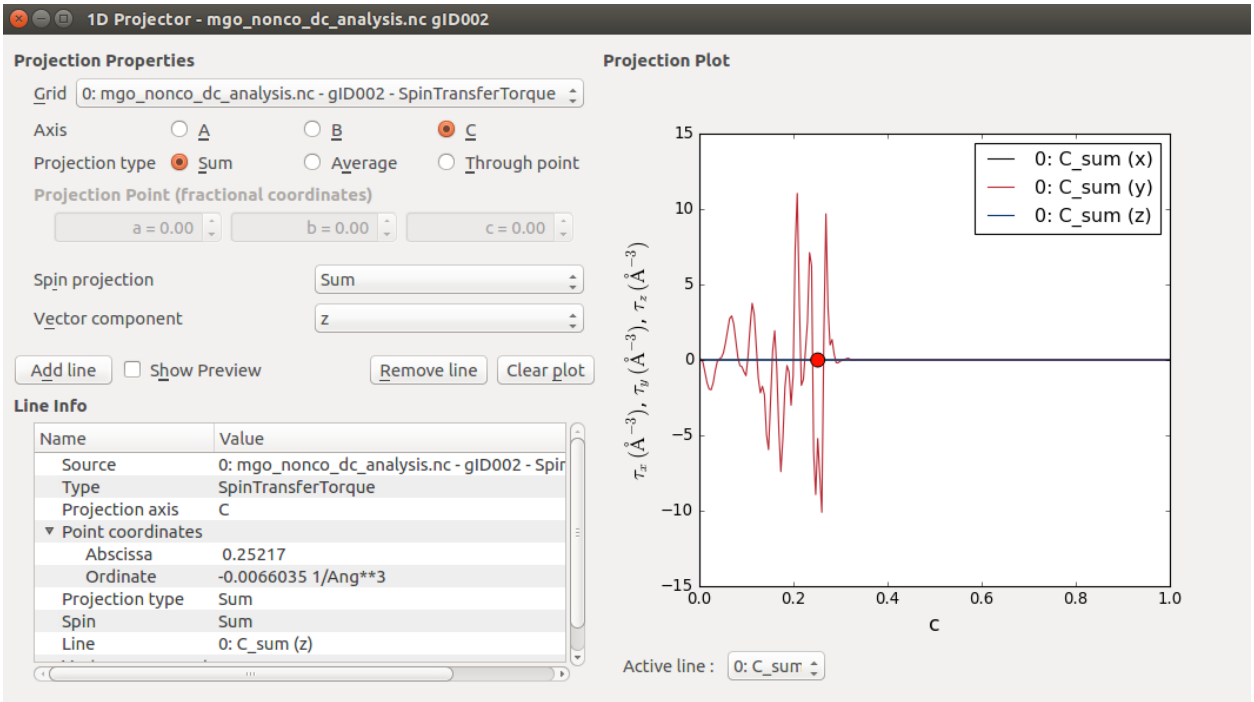
Once the calculation is finished, you will get the **MullikenPopulation** and **SpinTransferTorque** objects in the LabFloor. You can inspect the Mulliken charge and the component of the spin polarization vector using the text representation.

Text Representation - mgo_nonco_dc_analysis.nc gID001

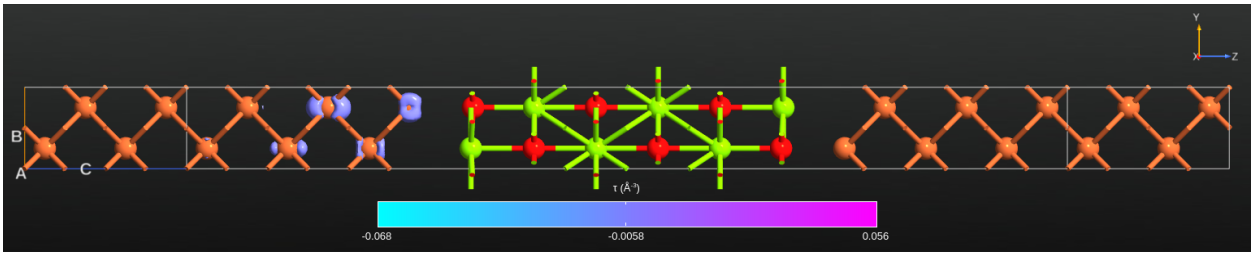
Element	Up	Down	Theta (Degrees)	Phi (Degrees)
0 Fe	5.0867	2.9217	0.0677	359.7676
1 Fe	5.0633	2.9370	0.1560	180.1813
2 Fe	5.0576	2.9400	0.0348	358.7518
3 Fe	5.1583	2.8468	0.5838	180.1091
4 Fe	5.1849	2.8335	24.8154	0.0034
5 Fe	5.5418	2.4686	49.8978	359.9971
6 Mg	0.5278	0.4751	135.7503	180.0182
7 O	3.4988	3.4673	65.4680	0.0035
8 O	3.5211	3.5194	59.5990	0.0220
9 Mg	0.4820	0.4752	37.2865	359.6952
10 Mg	0.4789	0.4786	52.7842	1.2203
11 O	3.5218	3.5216	30.5324	0.2408
12 O	3.5218	3.5216	2.1668	0.0697
13 Mg	0.4788	0.4787	11.8829	358.7890
14 Mg	0.4819	0.4753	0.0557	0.7770
15 O	3.5210	3.5195	0.0599	359.8667
16 O	3.5008	3.4657	0.0004	359.9098
17 Mg	0.5326	0.4722	179.9986	0.1708
18 Fe	5.5300	2.4769	0.0000	179.5913
19 Fe	5.1796	2.8391	0.0000	359.9876
20 Fe	5.1499	2.8555	0.0000	180.0096
21 Fe	5.0520	2.9460	0.0000	0.0127
22 Fe	5.0695	2.9308	0.0000	359.6984
23 Fe	5.0823	2.9116	0.0000	0.0180

In the noncolinear calculations, the sum of the up and down populations corresponds to the usual Mulliken charge (the number of electrons), and their difference – combined with the two angles – forms the spin polarization vector. This vector can be visualized using the **Viewer**:

You can also plot the spatial components of the STT using the **1D Projector**:



The figure below shows the MTJ configuration and the y component of the STT.



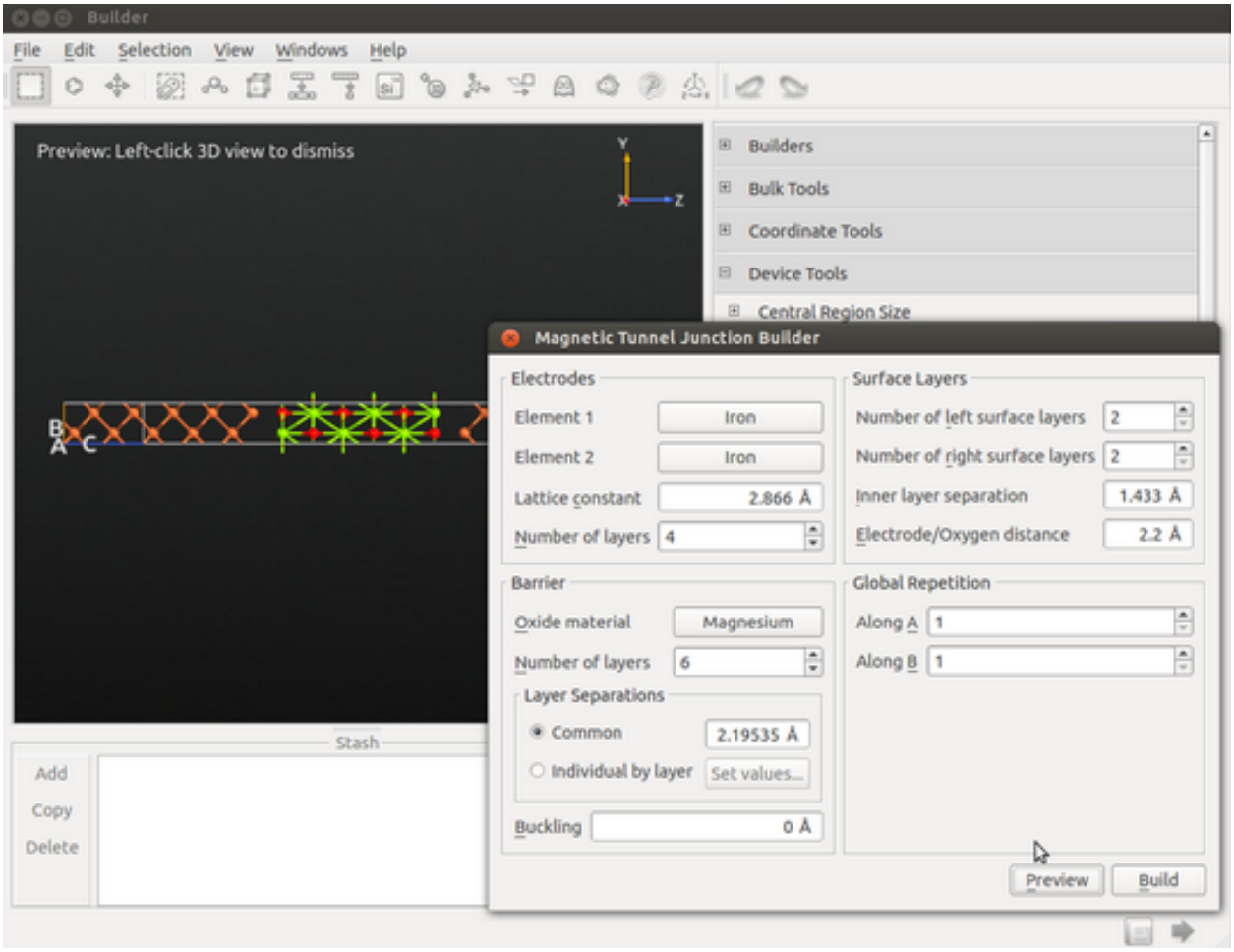
Relaxing the device central region

This section shows you how to create and geometry optimize the central region in the Fe|MgO|Fe junction. The tutorial *Advanced device relaxation - manual workflow* gives more detailed instructions for full device optimizations, including relaxed electrodes and optimum central region length. For the sake of simplicitly, we here perform force minimization for the device central region only.

Initial configuration

First, open the Builder. To set up the initial geometry of the Fe|MgO|Fe device you will use the **MTJ Builder** plugin, which is designed specifically for constructing this type of geometry. Locate it like this: *Add → From Plugin → Magnetic Tunnel Junction (FeMgO-style)*.

The MTJ builder can build a range of different magnetic tunnel junction geometries. For the present purpose, change the number of barrier layers to 6, and the number of surface layers to 2 on both the left and right side, but leave the other parameters as they are by default. Click the **Preview** button to see the geometry, then click **Build** to add it to the Stash.



Tip: Hover the mouse over each input field to get a description of the input parameters.

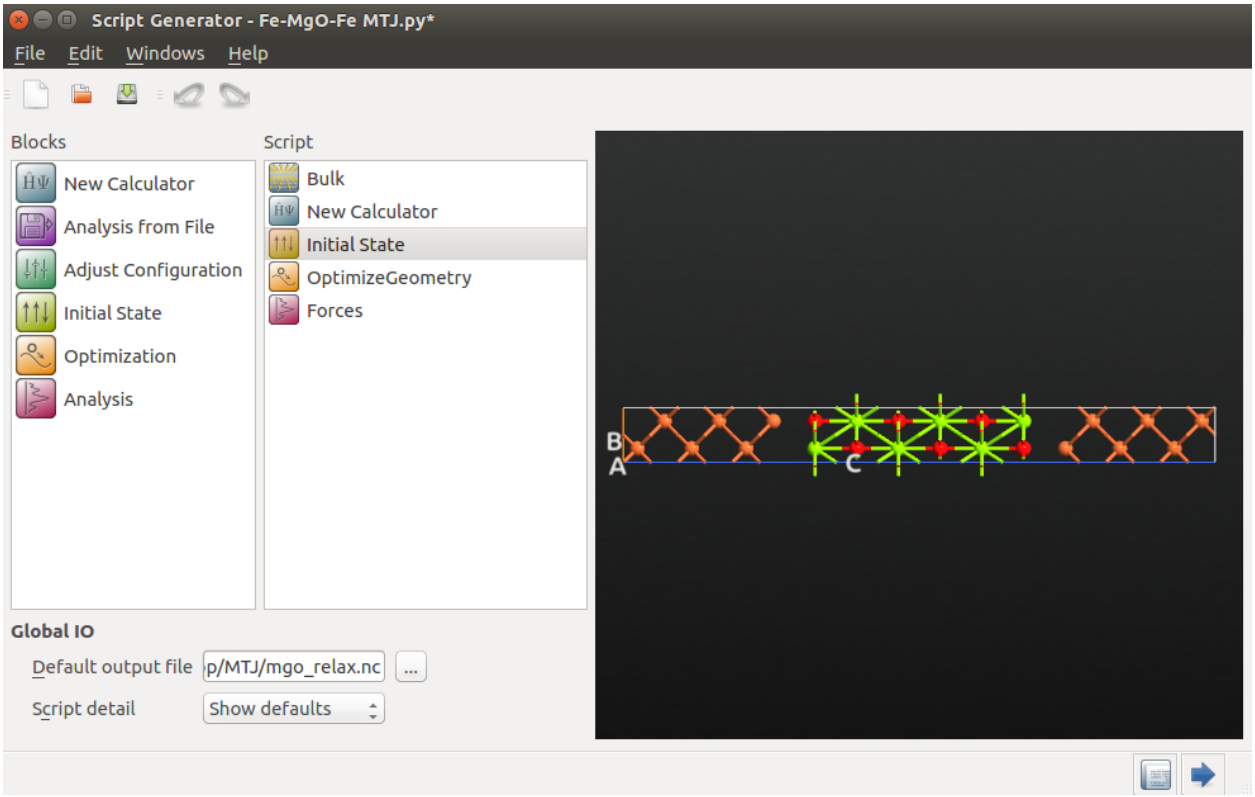
Convert the device geometry to a bulk configuration by clicking the **Bulk** tool on the plugins toolbar. This simply removes the electrodes and leaves you with the bulk central region.

Geometry optimization

Send the device central region to the **Script Generator** and add the following blocks to the script:

- New Calculator
- Initial State
- OptimizeGeometry
- Forces

Also, change the default output filename to `mgo_relax.nc`



The Calculator and InitialState settings should be very similar to those used in section *Parallel spin*:

•**Calculator**:

–Select the SGGA exchange-correlation method.

–Increase the electron temperature to 1200 Kelvin.

–Set the k-point grid to 7x7x2. This choice gives a reasonable k-point sampling in the x and y directions. In the z direction, you would like to model a device configuration, so the iron atoms at the edges must be bulk-like. Selecting 2 k-points in the z-direction improves the bulk like description of the Fe edge atoms.

–Select the SingleZetaPolarized basis set for iron.

•**Initial State**:

–Choose *User spin* for as the type of initial state.

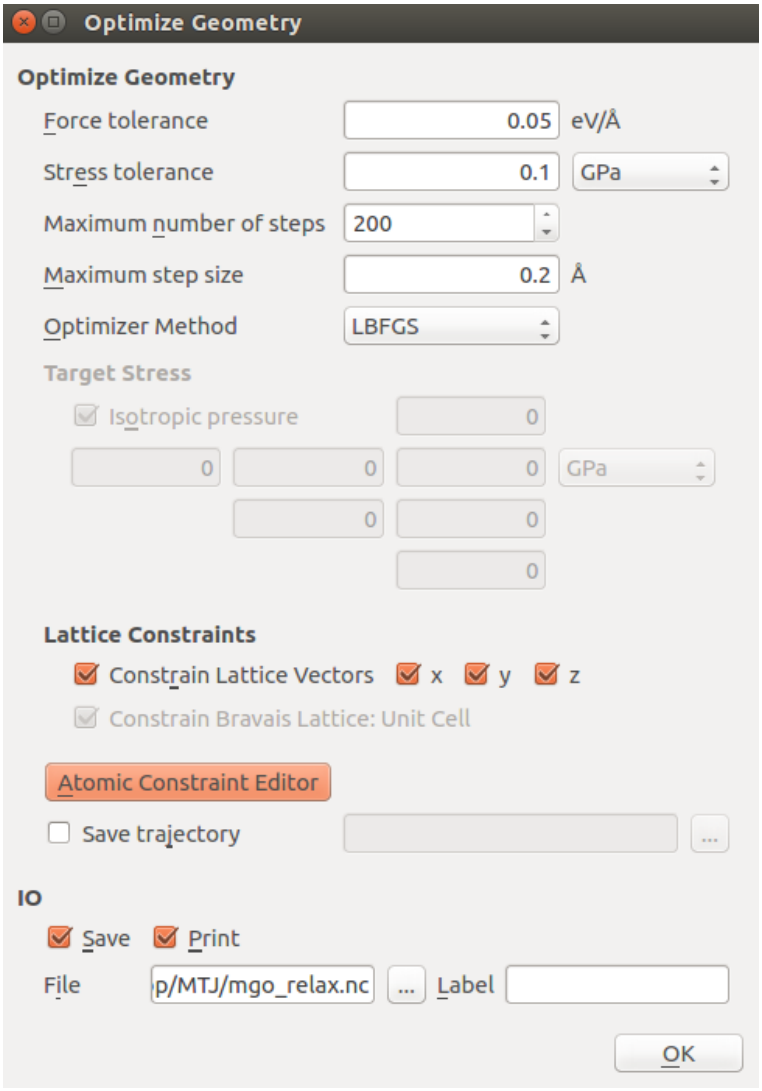
–Set the relative spins on O and Mg atoms to 0.

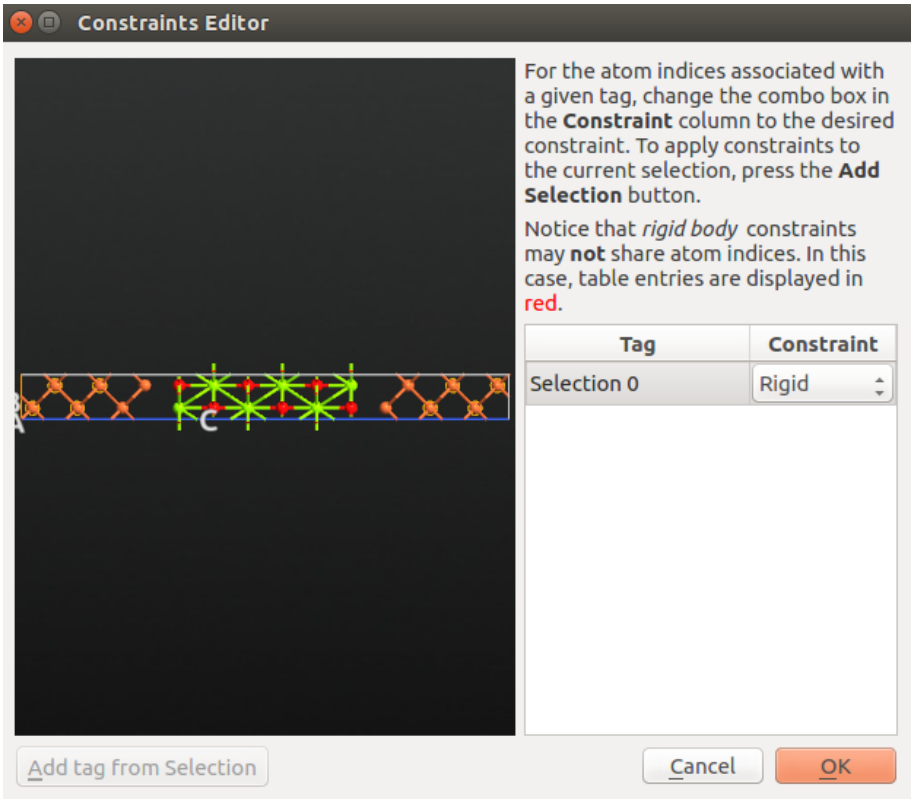
Next, open the **Optimization** block to specify that the first and last four atoms in the central region (the atoms in the electrode extensions) should be kept fixed during the relaxation. This is very important – if any of these atoms move you no longer have a periodic structure at the edges of the central cell, which is required for generating and attaching device electrodes later on.

Click *Add Constraints* to open the **Constraints Editor**. Then do the following:

•Select the atoms indicated in the image below by holding down the Ctrl key and drawing a rectangle around them. Then click *Add tag from Selection* to assign the tag Selection 0 to those atoms. Change the constraint type to Rigid for this tag group.

•Do the same for the right-hand electrode extension atoms, but leave the constraint type as Fixed.


Optimize Geometry
Optimize Geometry
Force tolerance
0.05
eV/Å
Stress tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer Method
LBFGS
Target Stress
Isotropic pressure
0
0
0
0
GPa
0
0
0
0
Lattice Constraints
Constrain Lattice Vectors
x
y
z
Constrain Bravais Lattice: Unit Cell
Atomic Constraint Editor
Save trajectory
...
IO
Save
Print
File
p/MTJ/mgo_relax.nc
...
Label
OK



Save the script as `mgo_relax.py` and run it using the Job Manager or from the command line:

```
atkpython mgo_relax.py > mgo_relax.log
```

If needed, you can also download the script here: `mgo_relax.py`. This calculation will take a 1-2 hours on a standard PC, but may finish in 20 minutes if distributed over 4 parallel MPI processes.

Inspecting the results

Locate the calculation data items on the **LabFloor**. Select the **Forces** item (*gID002*), and click the **Text Representation** plugin. You will now see a report of the final forces on all atoms.

Text Representation - mgo_relax.nc gID002

```
# Item:  0
# File:  /home/guest/Desktop/MTJ/mgo_relax.nc
# Title: mgo_relax.nc - gID002
# Type:  Forces
+-------------------------------------------------------------------------------+
| Forces Report                                                                 |
+-------------------------------------------------------------------------------+
|   0  [ -5.65166433e-13  -2.98611144e-13  -9.01817935e-02] eV/Ang               |
|   1  [  6.64493502e-12   6.52651480e-12   2.94255336e-02] eV/Ang               |
|   2  [  1.36823030e-12   1.95106286e-12  -1.00136659e-01] eV/Ang               |
|   3  [ -8.80667107e-12  -8.91005512e-12  -1.28493796e-01] eV/Ang               |
|   4  [  6.51485018e-12   6.03326508e-12   4.26430559e-02] eV/Ang               |
|   5  [  2.04540092e-12   1.95944348e-12   4.62198920e-02] eV/Ang               |
|   6  [ -2.37574737e-13   1.70103226e-13   2.44565333e-02] eV/Ang               |
|   7  [  5.65371777e-13   4.78715809e-13  -1.87839167e-02] eV/Ang               |
|   8  [  6.44396562e-13   3.09377807e-14   4.22671554e-02] eV/Ang               |
|   9  [ -3.17920036e-13   1.90817595e-13  -1.78024014e-02] eV/Ang               |
|  10  [  2.39036696e-13  -1.42687211e-13   6.38323635e-03] eV/Ang               |
|  11  [ -9.01442166e-14   3.67834264e-13   5.32441803e-03] eV/Ang               |
|  12  [  4.81927740e-13   6.67437624e-14  -6.05958779e-03] eV/Ang               |
|  13  [  8.34988325e-14   1.70593147e-13  -5.67429277e-03] eV/Ang               |
|  14  [ -2.10890956e-14   3.37287204e-14   1.85028336e-02] eV/Ang               |
|  15  [  4.25436429e-13   7.27451603e-14  -4.28807194e-02] eV/Ang               |
|  16  [  1.86917685e-13   1.50087159e-13   1.85504931e-02] eV/Ang               |
|  17  [  6.00559901e-13   2.32819142e-13  -2.46488340e-02] eV/Ang               |
|  18  [  1.37458279e-12   1.99913731e-12  -4.16245073e-02] eV/Ang               |
|  19  [  4.42454411e-12   5.54030364e-12  -3.40699334e-02] eV/Ang               |
|  20  [ -5.65094164e-12  -7.22871690e-12   1.17937925e-01] eV/Ang               |
|  21  [  8.76830526e-12   1.10265548e-11   9.98868244e-02] eV/Ang               |
|  22  [ -1.87565300e-11  -2.07851072e-11  -3.01908210e-02] eV/Ang               |
|  23  [ -7.04845969e-14  -2.85051189e-13   9.13089459e-02] eV/Ang               |
+-------------------------------------------------------------------------------+
```

Note that all forces in the x-y direction are for all practical purposes zero. For the atoms which have been relaxed, the forces are less than the optimization criterion of 0.05 eV/Ang. For the electrode extension atoms, the forces are slightly larger. Moreover, the force vectors point out of the cell, indicating that the cell is under compressive strain. Adding more surface layers to increase the cell length in the z-direction would lower the strain, but would have very little effect on results in the present case.

You can now use the optimized device central region for calculations of transport properties of the device, c.f. section *Getting started*.

References

3.10.3 Relativistic effects in bulk gold

Gold is one of the more heavy transition metals, and the electronic structure of bulk gold is therefore significantly affected by relativistic effects, including spin-orbit coupling (SOC (spin-orbit coupling)).

Relativistic *ab initio* band structure calculations for gold were first reported in the early 1970s, see e.g. Refs.[1],[2], and[3]. Those studies, and later ones, found that relativistic effects cause a significant reorganization of the electronic energy bands in gold. In particular, inclusion of relativistic effects removes a large part of the band crossings in the deeper valence bands found in standard calculations. In this tutorial, you will demonstrate this using ATK-DFT for standard scalar-relativistic (GGA) and relativistic (SOGGA) calculations of the gold band structure.

[1] M.ăG. Ramchandani. Energy band structure of gold. *Journal of Physics C: Solid State Physics*, 3(1S):S1, 1970. URL: http://stacks.iop.org/0022-3719/3/i=1S/a=301.

[2] N.ăEgede Christensen and B.ăO. Seraphin. Relativistic Band Calculation and the Optical Properties of Gold. *Phys. Rev. B*, 4:3321–3344, Nov 1971. doi:10.1103/PhysRevB.4.3321.

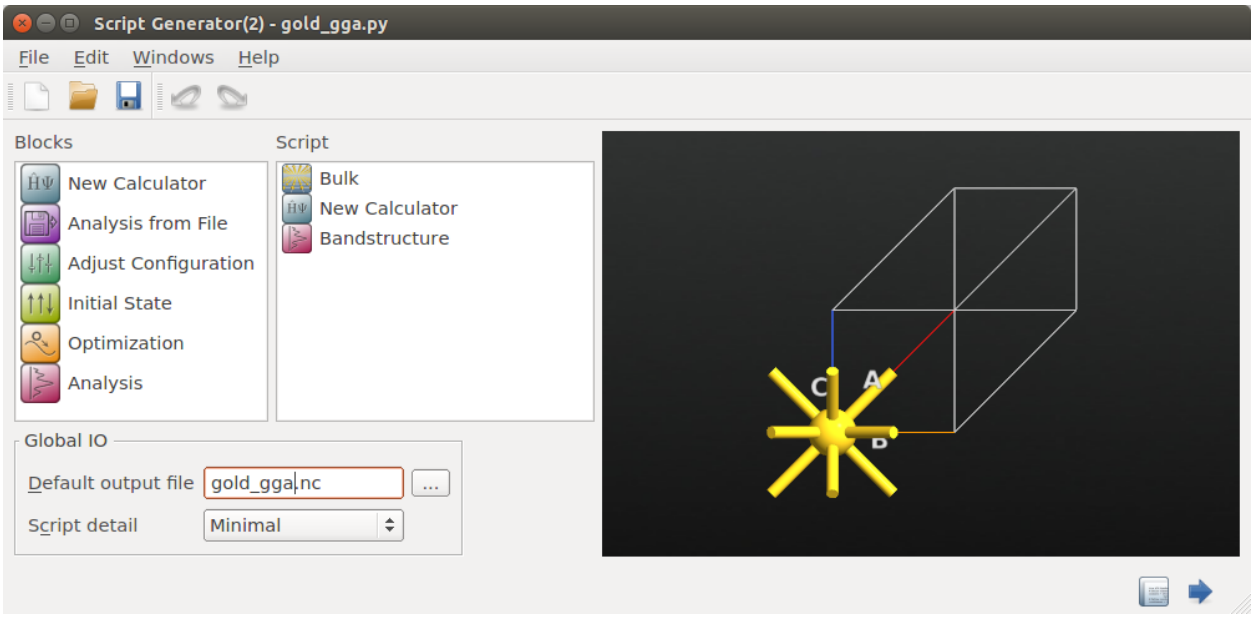
[3] N.ăEgede Christensen. High-energy band structure of gold. *Phys. Rev. B*, 13(6):2698–2701, 1976. doi:10.1103/PhysRevB.13.2698.

GGA band structure

Use the QuantumATK Builder to create the gold bulk configuration. This is most easily done using the *Add* → *From Database* tool. Keep the gold lattice constant at the experimental value (4.078 Å).

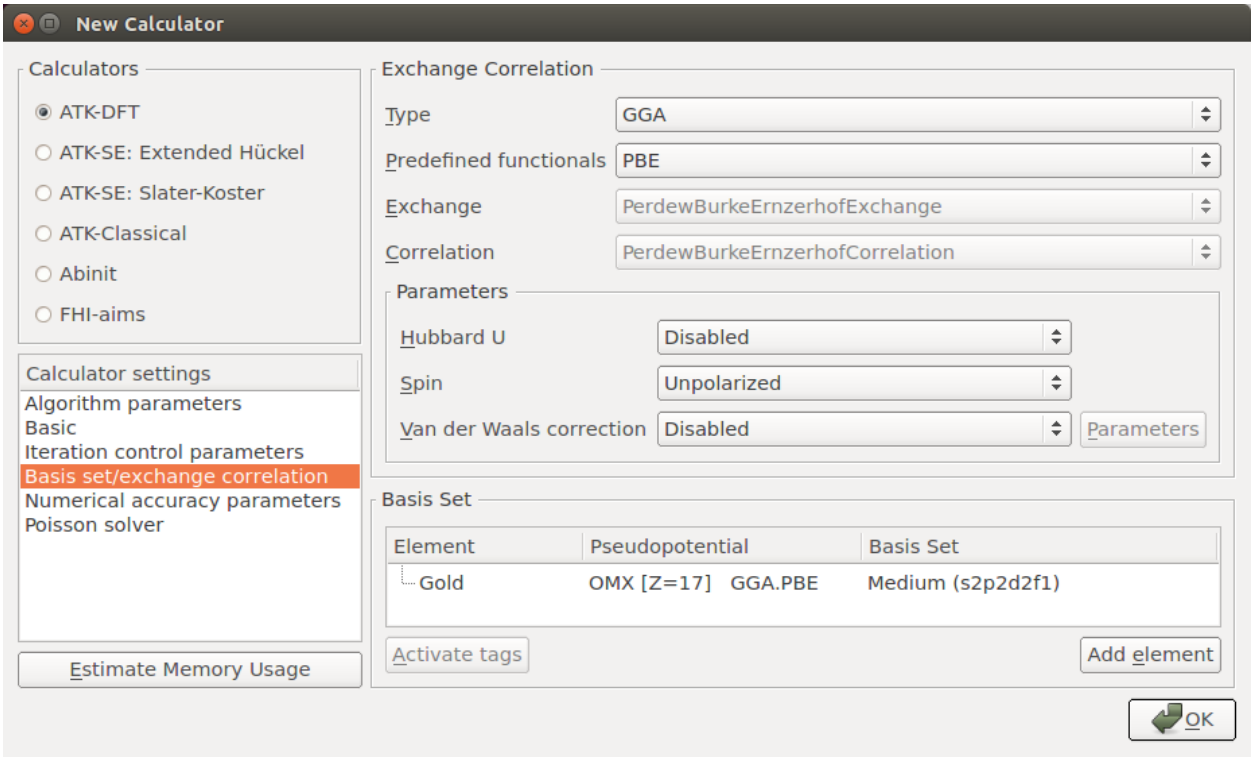
Then transfer the configuration to the Script Generator and set up a standard GGA-PBE band structure calculation using an OMX pseudopotential.

First, add a New Calculator block and a Bandstructure analysis block, and set `gold_gga.nc` as the output file.

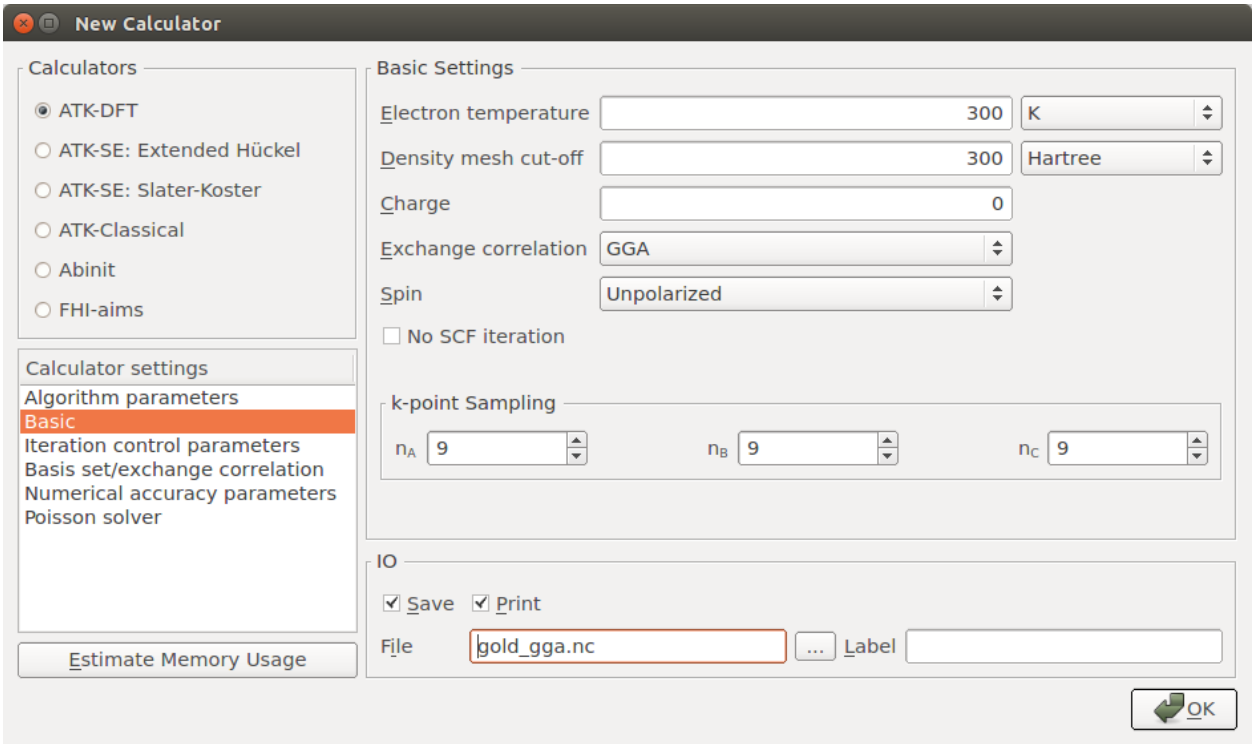


Edit the New Calculator settings:

- In the **Basis set/exchange correlation** panel, choose the GGA-PBE exchange-correlation functional and the OMX pseudopotential with a *Medium* accuracy basis set.

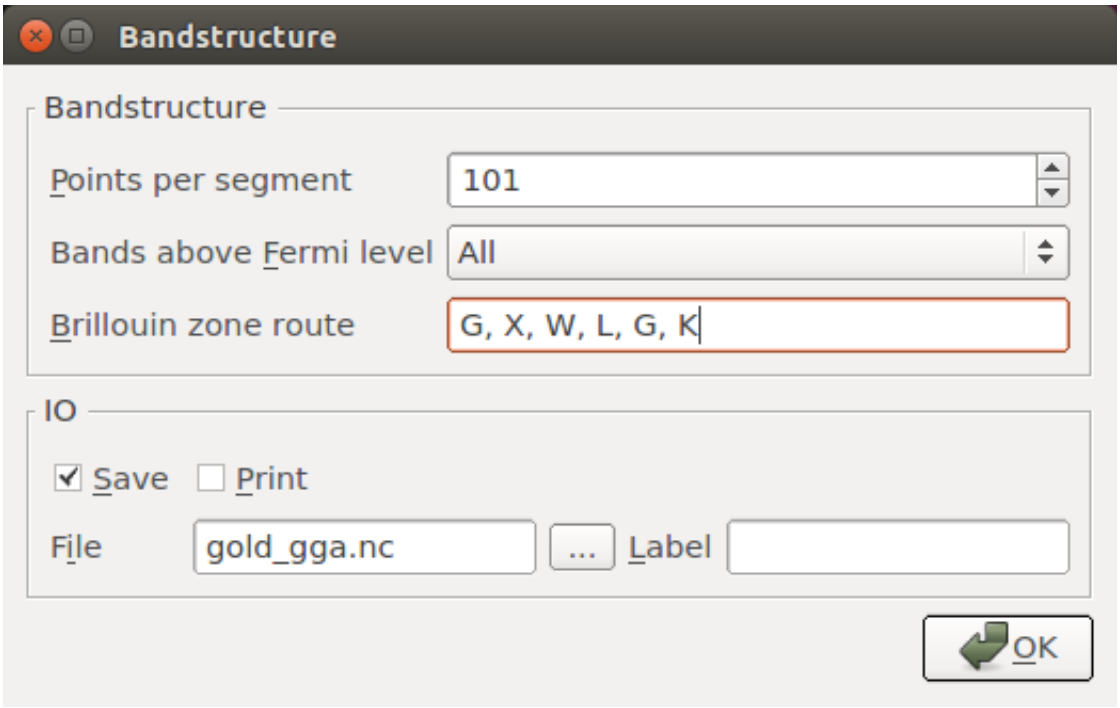


•In the **Basic** panel, set the *Density mesh-cutoff* to 300 Hartree and increase the *k-point Sampling* to 9x9x9.



Open the Bandstructure block and edit it:

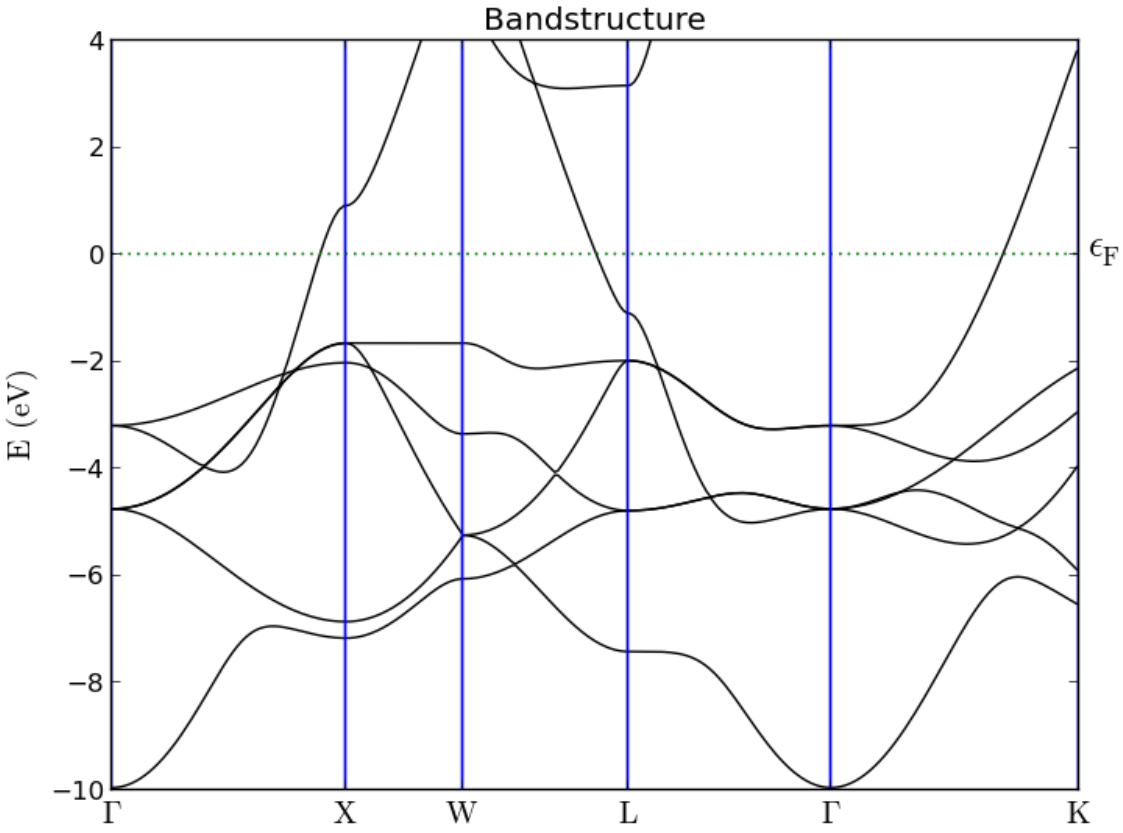
•Increase *Points per segment* to 101 in order to finely sample the band structure between each high-symmetry point, and trim the default *Brillouin zone route* such that it reads G, X, W, L, G, K.



Save the script as `gold_gga.py` and run it. You can use the Job Manager for this. It should take around a minute if executed on a single processor.

Results

Find the Bandstructure object in the output file on the QuantumATK LabFloor and use the **Bandstructure Analyzer** to plot it. The results are shown below and are very similar to those reported in Refs.[Page 1385, 2] and[Page 1385, 3]. Gold is clearly not an insulator, since there are plenty of bands crossing the Fermi level. Also note the frequent band crossing in the deeper valence bands, from ca. -2 to -5 eV below the Fermi energy. As we shall see, most of these crossings disappear in a relativistic calculation including SO interactions.



Spin-orbit GGA band structure

You will here set up and execute a GGA calculation that includes the SOC. This must be done in a noncollinear representation of electron spin, which is computationally more expensive than the GGA calculation and can exhibit poor SCF convergence. You should therefore apply two tricks: a) start the noncollinear calculation from a converged spin-polarized state (SGGA), and b) use a special electron density mixing scheme.

Note: We know that gold is not magnetic, so it may seem a bit silly to do a spin-polarized calculation for gold. After all, the results should not differ from those found in the collinear spin-compensated calculation done in the previous section.

Indeed, they do not, but the converged spin-polarized state offers a much better initial guess for the self-consistent noncollinear+SO state than the spin-compensated state does: Fewer noncollinear SCF steps are required if starting from the spin-polarized state, and computational time is therefore saved.

SGGA initial state

You have two options for setting up a script for the SGGA calculation:

a. Use the **Scripter**: Set up the script exactly like in the previous section, but choose Polarized for the *Spin* parameter, remove the Bandstructure analysis object, set the output file to `gold_sgga.nc`, and save the new script as `gold_sgga.py`.

b. Use the **Editor**: Copy `gold_gga.py` to `gold_sgga.py` and open the new script in the Editor and edit it a bit. Set the exchange-correlation to `SGGA.PBE`, remove the bottom lines that execute the bandstructure calculation, and rename the output file to `gold_sgga.nc`.

In any case, the script should look like this:

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(4.07825*Angstrom)

# Define elements
elements = [Gold]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#----------------------------------------
# Basis Set
#----------------------------------------

GoldBasis = OpenMXBasisSet(
    element=PeriodicTable.Gold,
    filename="openmx/pao/Au7.0.pao.zip",
    atomic_species="s2p2d2f1",
    pseudopotential=NormConservingPseudoPotential("normconserving/upf2/Au_PBE13.upf.zip
↪"),
    )

basis_set = [
    GoldBasis,
    ]

#----------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Exchange-Correlation
#-------------------------------------
exchange_correlation = SGGA.PBE

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(9, 9, 9),
    density_mesh_cutoff=300.0*Hartree,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('gold_sgga.nc', bulk_configuration)
```

Run the script, it should be very fast. Note that the output file `gold_sgga.nc` is generated and that it contains only a single object; the gold bulk configuration including the self-consistent SGGA electronic state, which will be used in the next step.

SOGGA band structure

Use the script below to perform the following work-flow:

1. Read the `bulk_configuration` from the file `gold_sgga.nc`.
2. Get the `calculator` used in the SGGA calculation and modify it slightly; use the SOGGA exchange-correlation and a special mixing scheme for noncollinear calculations.
3. Use the converged SGGA state as initial guess for the SOGGA electron density.
4. Do the SOGGA ground state calculation followed by band structure analysis.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('gold_sgga.nc', BulkConfiguration)[0]

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )
```

(continues on next page)

(continued from previous page)

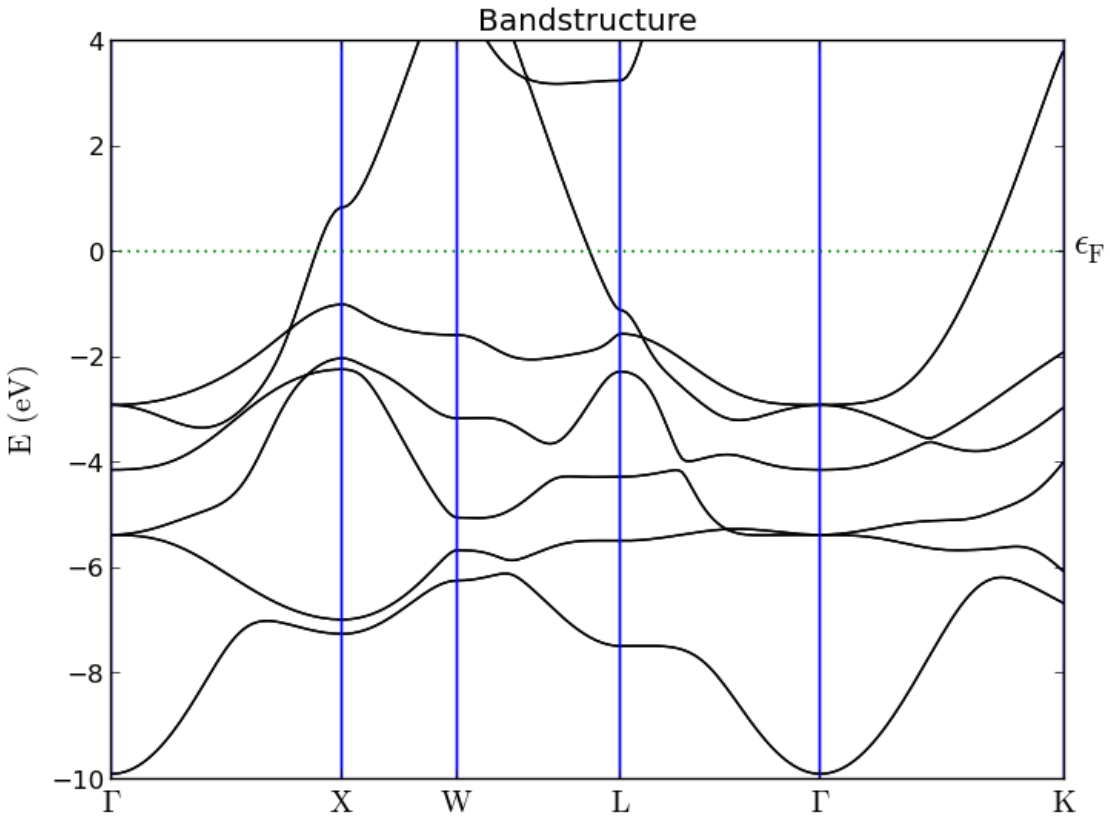
```
# Get the calculator and modify it for spin-orbit GGA
calculator = bulk_configuration.calculator()
calculator = calculator(
    exchange_correlation = SOGGA.PBE,
    iteration_control_parameters = iteration_control_parameters
    )

# Setup the initial state from the GGA calculation
bulk_configuration.setCalculator(
    calculator,
    initial_state = bulk_configuration
    )
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('gold_sogga.nc', bulk_configuration)

# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'X', 'W', 'L', 'G', 'K'],
    points_per_segment=101,
    bands_above_fermi_level=All
    )
nlsave('gold_sogga.nc', bandstructure)
```

Save the script as `gold_sogga.py` and run the calculation. It should take roughly 6 minutes to finish if executed in parallel on 4 CPUs.

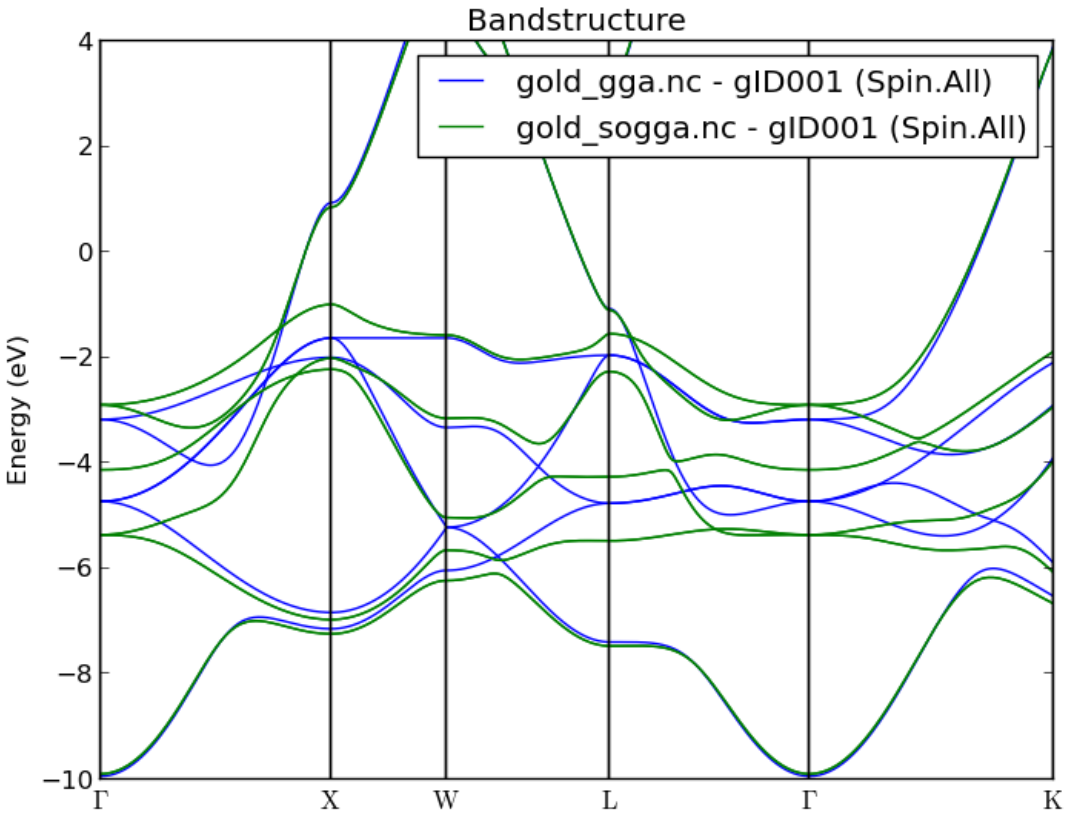
Results

The plot below shows the band structure obtained from the SOGGA calculation. It follows very closely the literature results in Refs.[Page 1385, 2] and[Page 1385, 3] and is virtually identical to the SOGGA band structure of gold recently published in Ref.[4]. Note how most of the valence band crossings found in the standard GGA calculation have now been lifted.

[4] T.ăRangel, D.ăKecik, P.ăE. Trevisanutto, G.-M. Rignanese, H.ăVanăSwygenhoven, and V.ăOlevano. Band structure of gold from many-body perturbation theory. *Phys. Rev. B*, 86:125125, Sep 2012. doi:10.1103/PhysRevB.86.125125.



You can also use the **Compare Data** plugin for a more direct comparison of the two band structures, as illustrated below. Simply highlight both the GGA and the SOGGA bandstructure objects, and click the plugin. It is quite clear that relativistic effects have a significant impact on the gold valence bands. The data illustrated below is essentially identical to that in Figure 6 of Ref.[4].



References

3.10.4 Noncollinear calculations for metallic nanowires

Version: 2016.3

In the tutorial *Introduction to noncollinear spin* you learned how to perform a simple noncollinear calculation. In this tutorial you will apply the same procedure to more advanced systems. In a first example, you will consider a metal nanowire connecting two electrodes of the same material. In the second example, you will consider an infinite wire and study the effect of spin-orbit coupling and spin-orientation on the electronic structure. The calculations will be very similar to the two works published by Czerner et al.:[1][2].

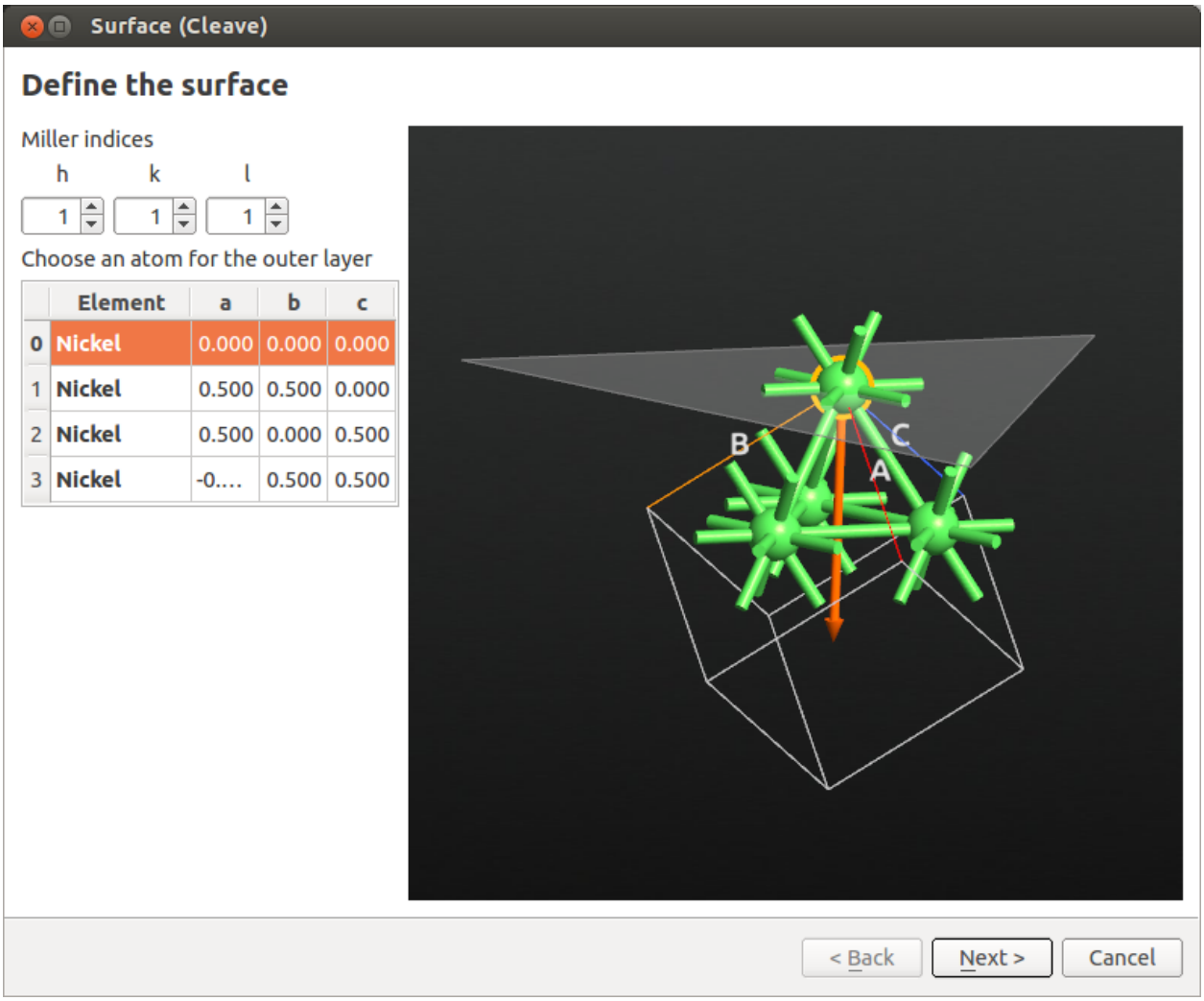
[1] Michael Czerner, BogdanăYu. Yavorsky, and Ingrid Mertig. Fully relaxed magnetic structure of transition metal nanowires: First-principles calculations. *Phys. Rev. B*, 77:104411, 2008. doi:10.1103/PhysRevB.77.104411.

[2] Michael Czerner, BogdanăYu. Yavorsky, and Ingrid Mertig. The role of noncollinear magnetic order and magnetic anisotropy for the transport properties through nanowires. *Phys. Stat. Sol. B*, 247:2594, 2010. doi:10.1002/pssb.201046190.

Building the device

Set up the Ni(111) surface

Open the **Builder**, click on *Add* → *From Database*, search for the Nickel *fcc* bulk structure and add it to the Stash. Open *Builders* → *Surface (Cleave)*, enter Miller indices (111) as shown in the figure below, and click Next.

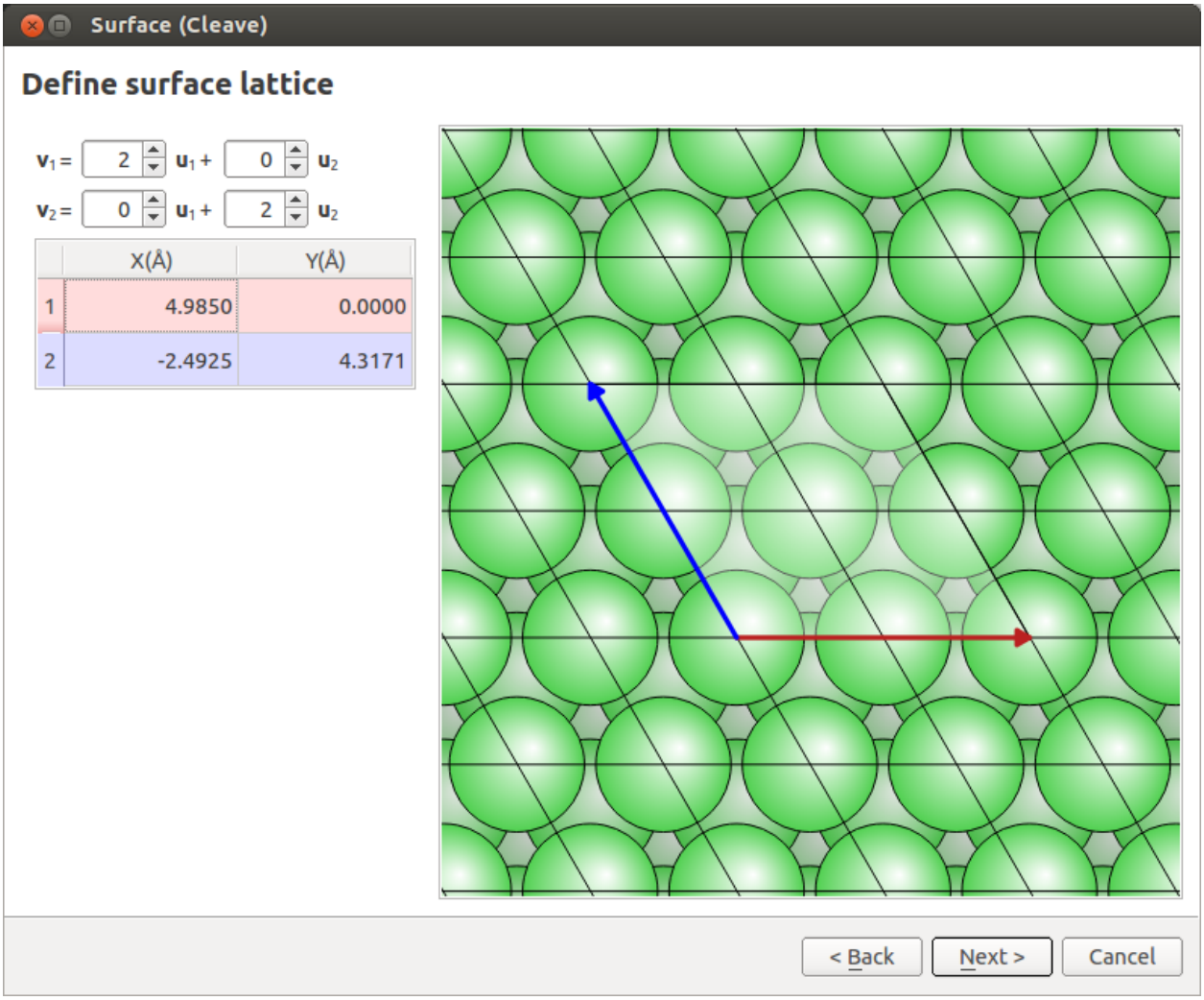


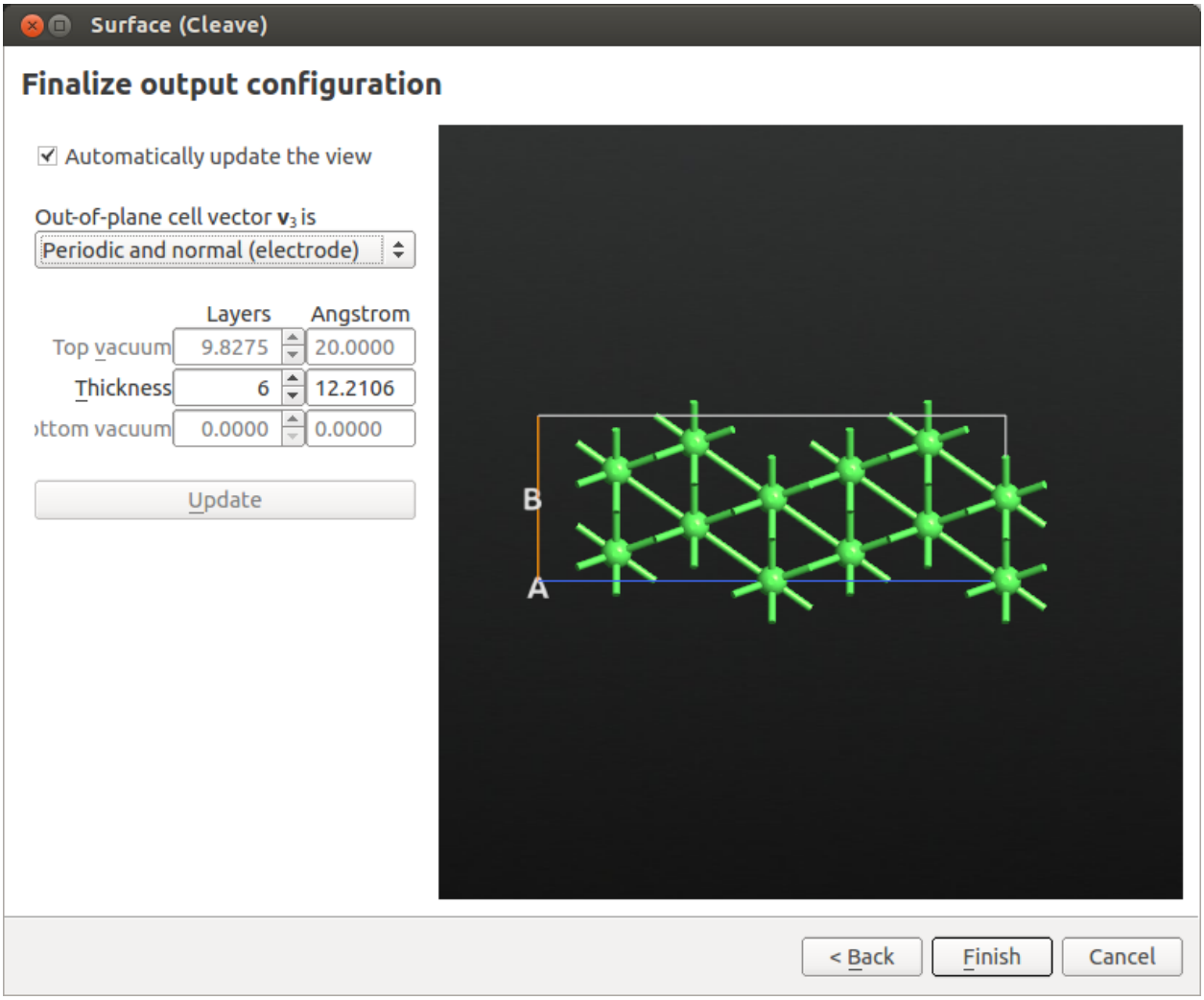
In the next window, define a 2×2 surface supercell as shown below, then click Next.

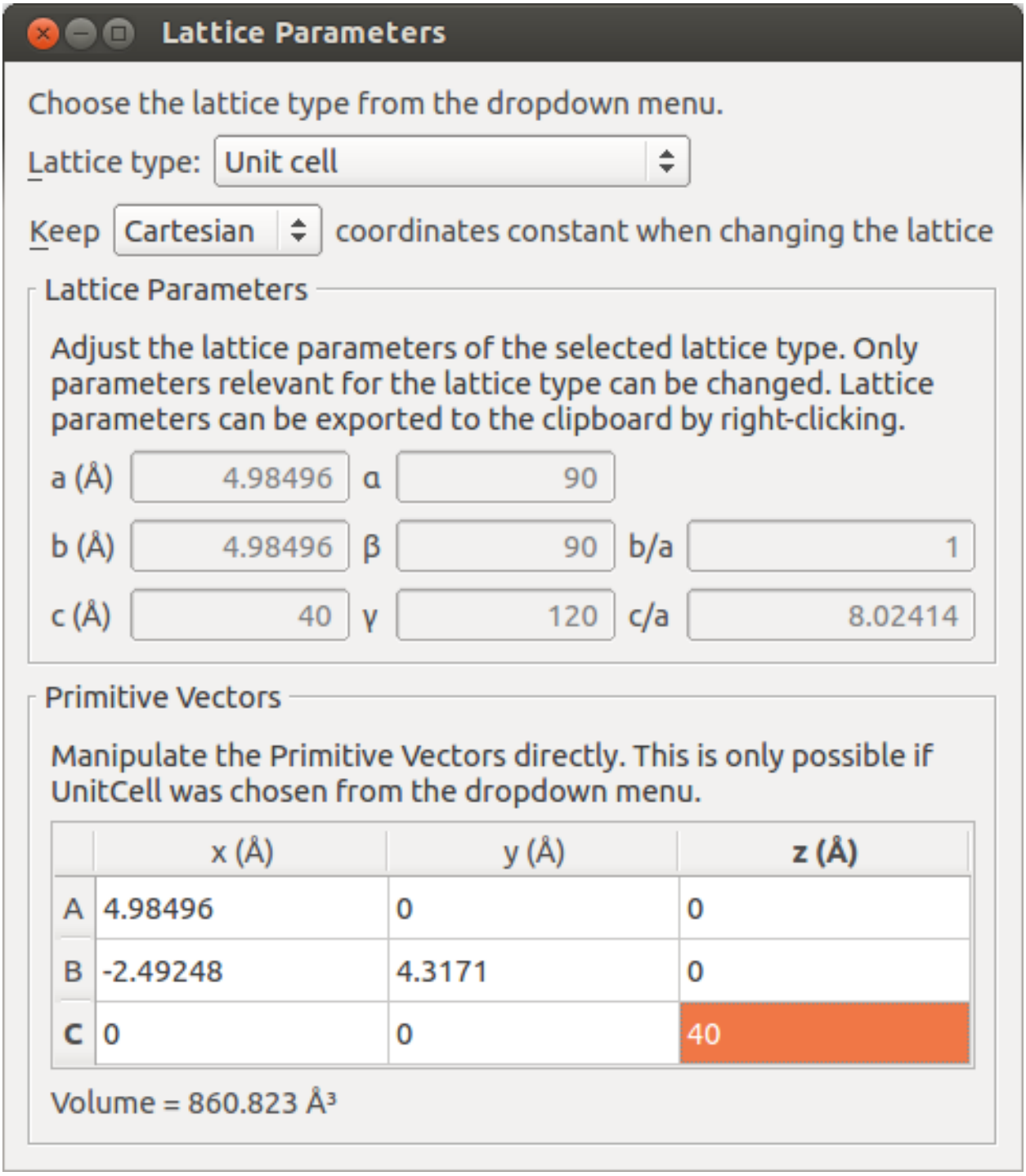
Set the out-of-plane direction to *Periodic and normal (electrode)*, set the thickness to 6 layers (see image below) and click Finish.

Finally, add some vacuum above the (111) surface. Go to *Bulk Tools* → *Lattice Parameters* and set the C vector length to 40 Å. Be careful to select to *Keep Cartesian coordinates constant when changing the lattice*.

Note: In principle, you can also create the surface by adding the vacuum in the **Surface (Cleave)** tool by choosing a slab configuration. The procedure followed here will make the visualization of the next steps easier.

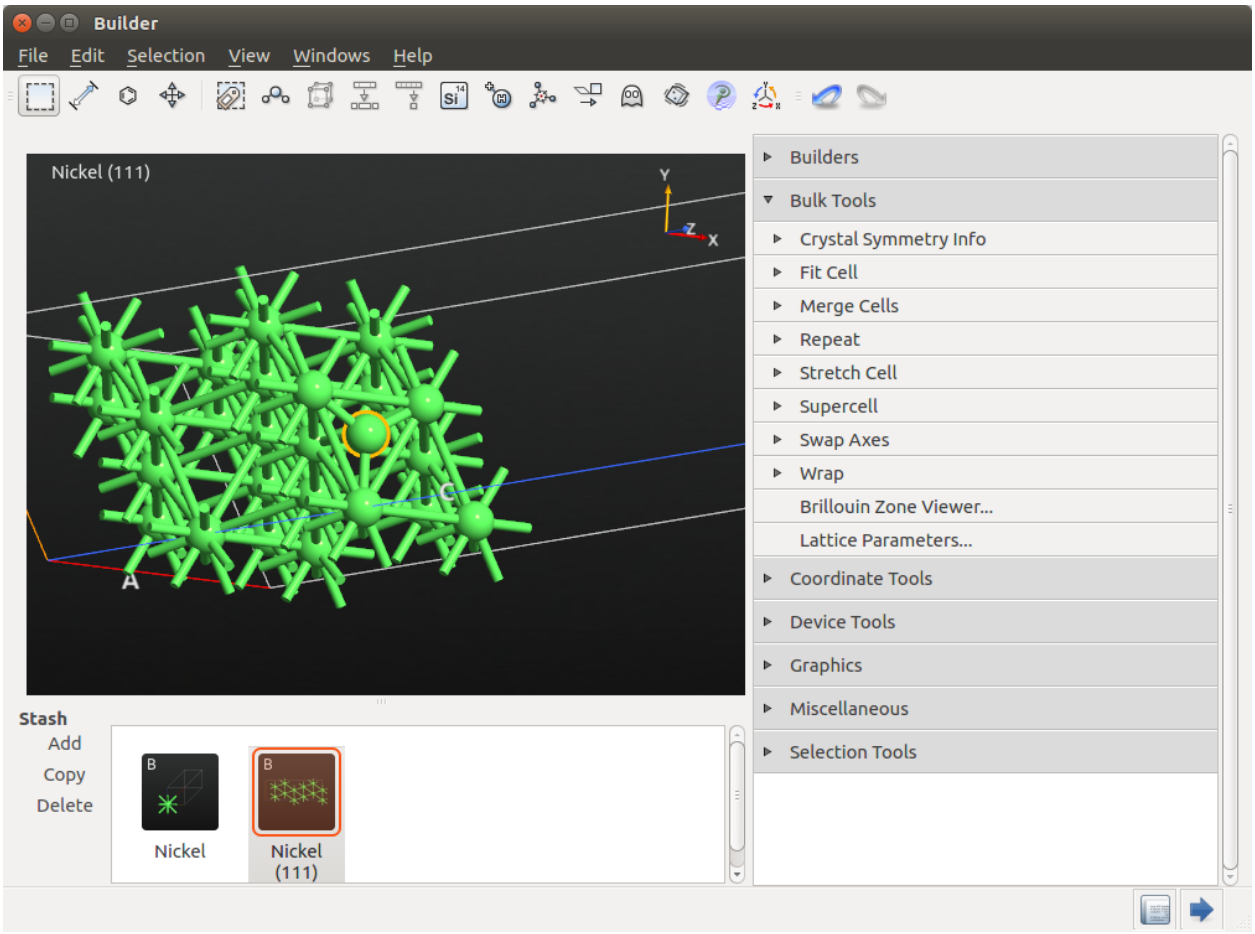

Surface (Cleave)
Define surface lattice
v1 = 2 u1 + 0 u2
v2 = 0 u1 + 2 u2
X(Å)
Y(Å)
1
4.9850
0.0000
2
-2.4925
4.3171
< Back
Next >
Cancel


Surface (Cleave)
Finalize output configuration
Automatically update the view
Out-of-plane cell vector $\mathbf{v}_3$ is
Periodic and normal (electrode)
Layers
Angstrom
Top vacuum
9.8275
20.0000
Thickness
6
12.2106
ttom vacuum
0.0000
0.0000
Update
B
A
< Back
Finish
Cancel


Lattice Parameters
Choose the lattice type from the dropdown menu.
Lattice type: Unit cell
Keep Cartesian coordinates constant when changing the lattice
Lattice Parameters
Adjust the lattice parameters of the selected lattice type. Only parameters relevant for the lattice type can be changed. Lattice parameters can be exported to the clipboard by right-clicking.
a (Å) 4.98496 α 90
b (Å) 4.98496 β 90 b/a 1
c (Å) 40 γ 120 c/a 8.02414
Primitive Vectors
Manipulate the Primitive Vectors directly. This is only possible if UnitCell was chosen from the dropdown menu.
x (Å) y (Å) z (Å)
A 4.98496 0 0
B -2.49248 4.3171 0
C 0 0 40
Volume = 860.823 Å³

Set up the Ni wire between two Ni(111) surfaces

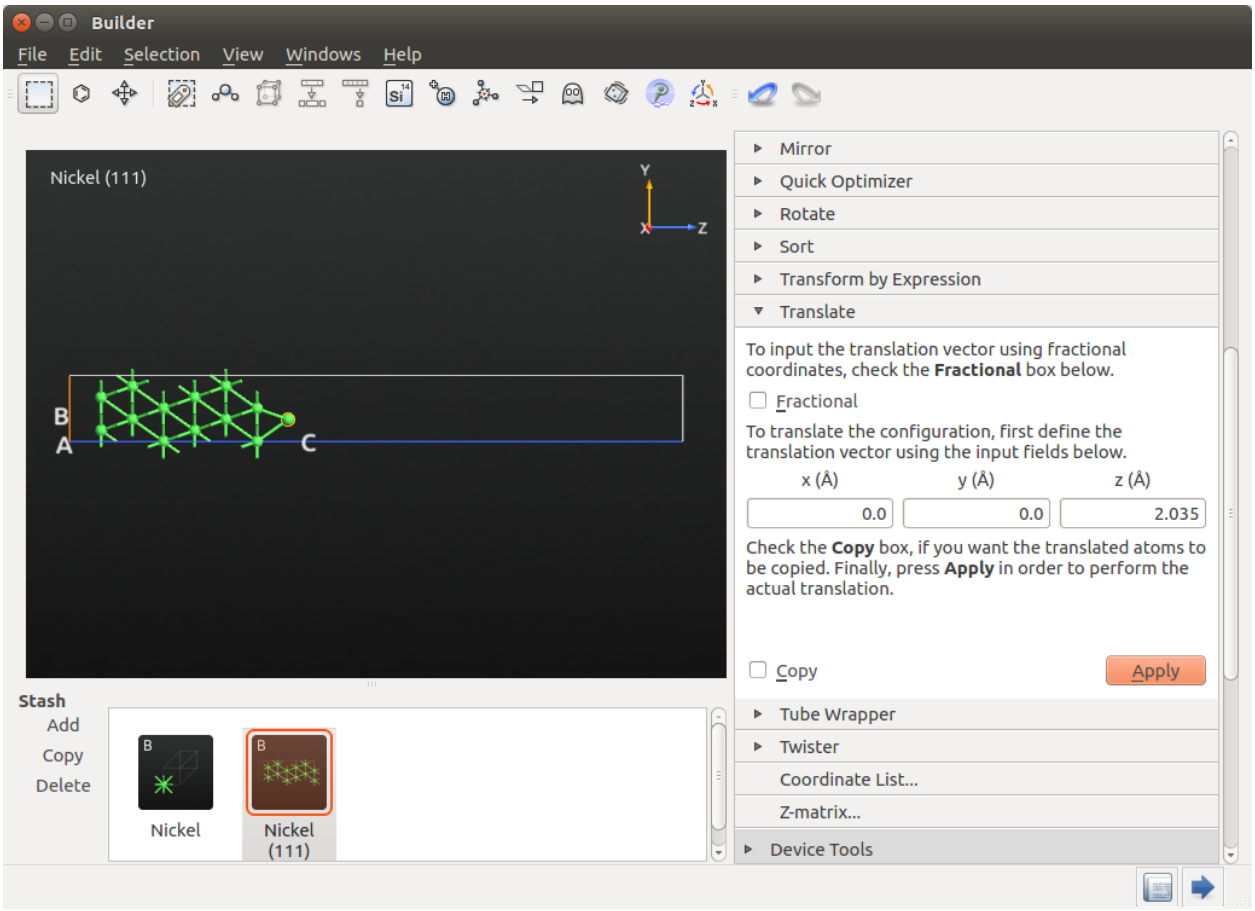
Select three Ni atoms at the surface and click the centroid plugin on the toolbar at the top of the **Builder** to add a Ni atom in the geometric center of the selected atoms.

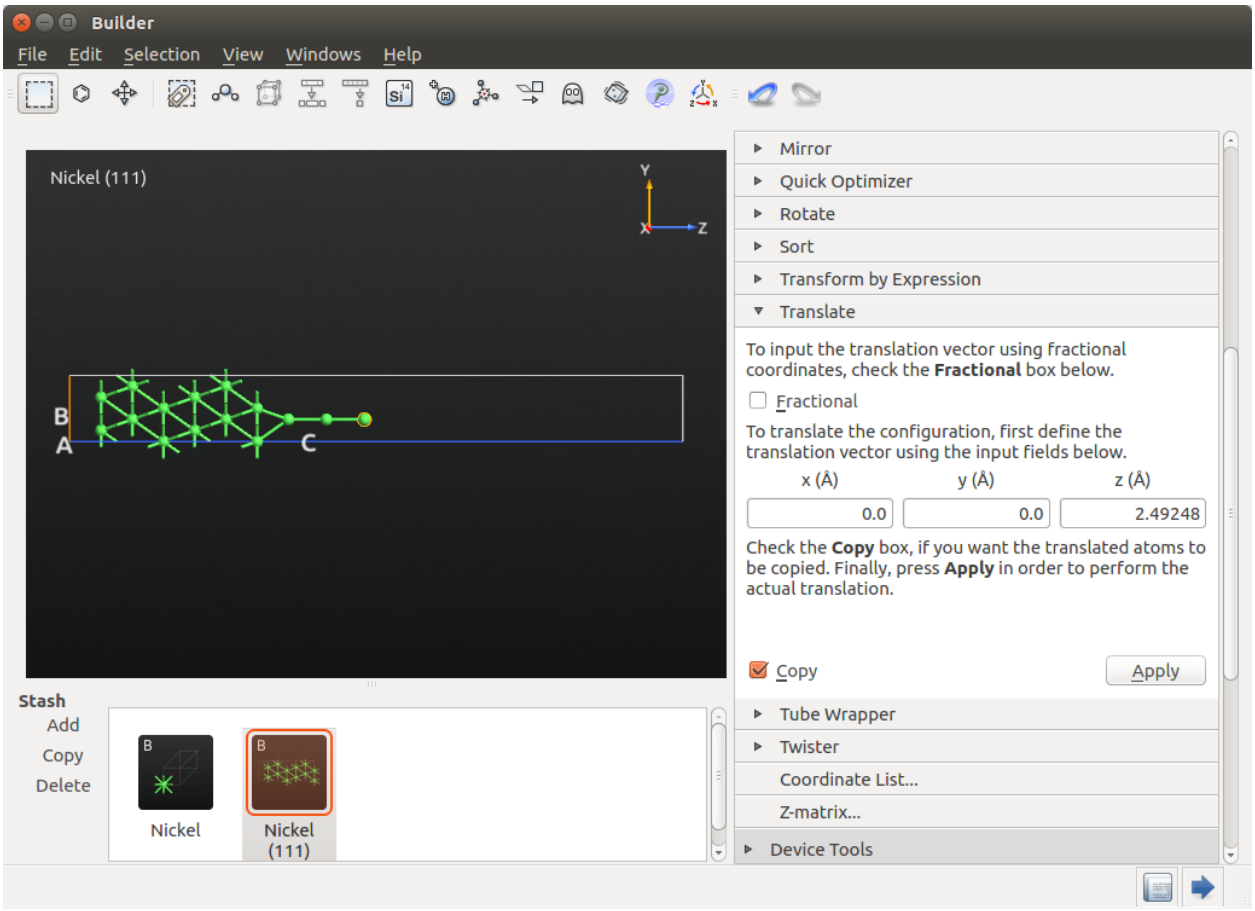


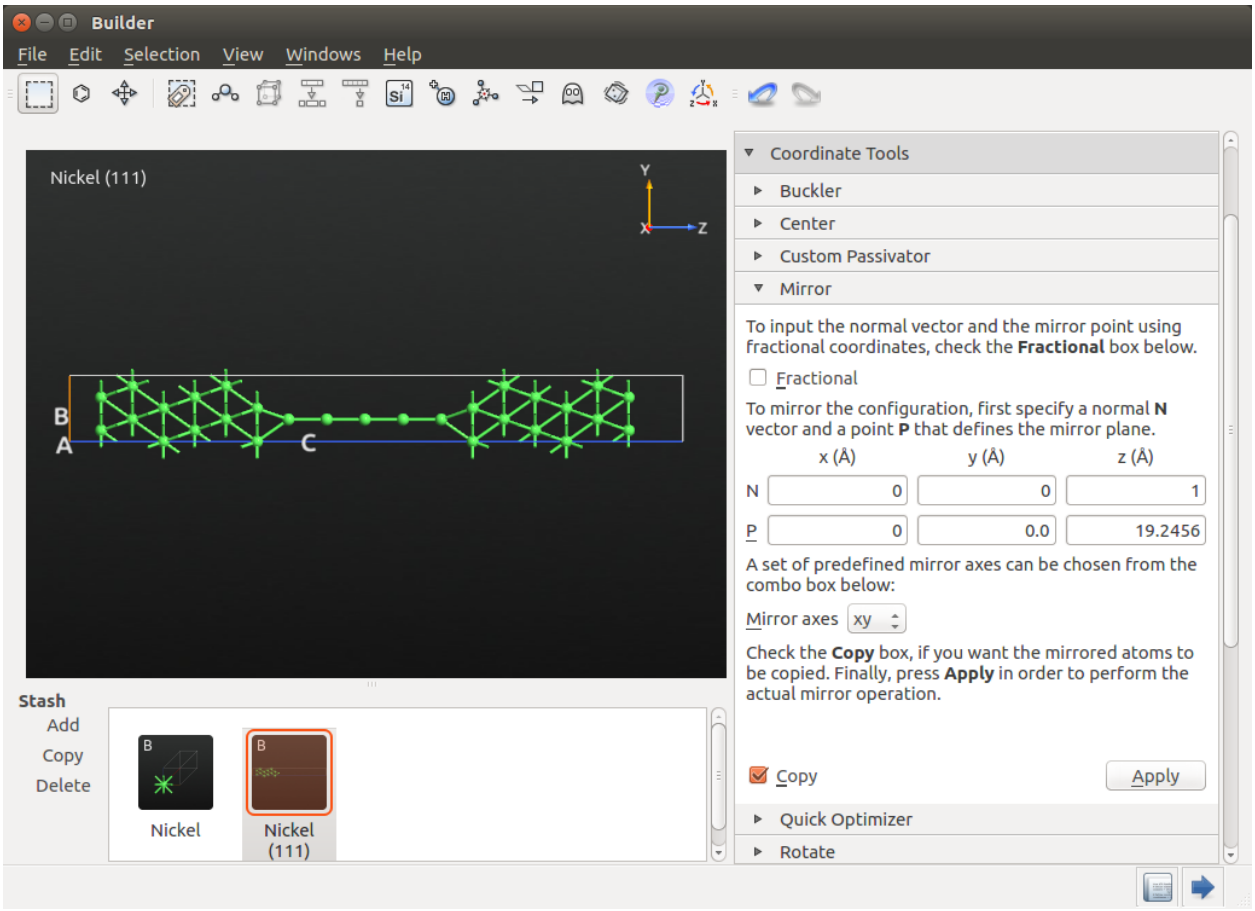
Then, go to *Coordinate Tools* → *Translate tool* to move this atom by 2.035 Å above the surface. This distance corresponds approximately to the Ni(111) interlayer distance.

To create the wire, use again *Coordinate Tools* → *Translate* but this time select the *Copy* option and translate by 2.49248 Å, which corresponds to the Ni-Ni distance. Do it twice and you are halfway to creating a 5-atom wire embedded between two Ni electrodes.

Take note of the Z coordinate of the last Ni atom added to the chain (19.2306 Å) and select all the Ni atoms in your system except this last Ni atom. Then, go to *Coordinate Tools* → *Mirror*, select the predefined xy mirror axes and enter 19.2306 Å as the Z coordinate of the mirror point P. Remember to check the Copy box to make a copy of the mirrored object, and press Apply and you will obtain the structure shown below.

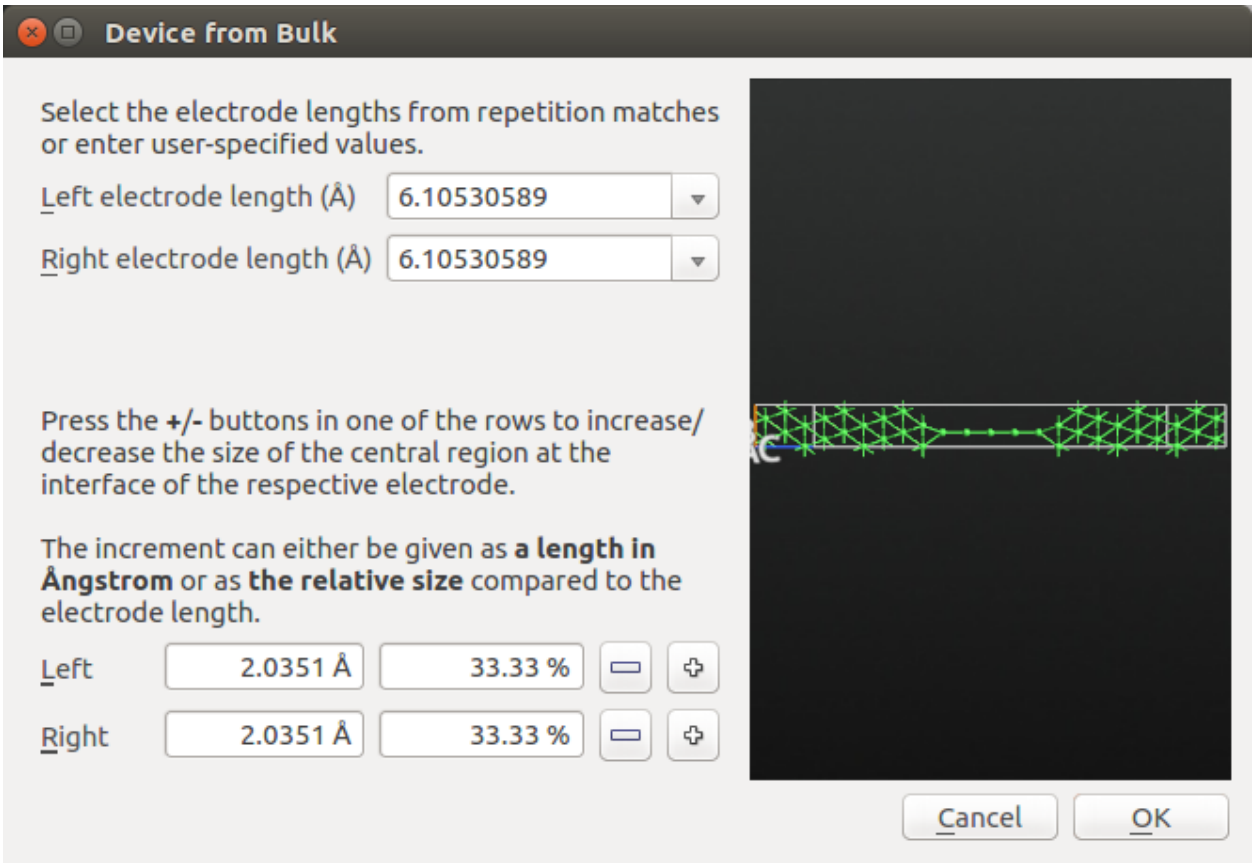

Builder
File
Edit
Selection
View
Windows
Help
Nickel (111)
Y
X
Z
B
A
C
Mirror
Quick Optimizer
Rotate
Sort
Transform by Expression
Translate
To input the translation vector using fractional coordinates, check the Fractional box below.
Fractional
To translate the configuration, first define the translation vector using the input fields below.
x (Å)
y (Å)
z (Å)
0.0
0.0
2.035
Check the Copy box, if you want the translated atoms to be copied. Finally, press Apply in order to perform the actual translation.
Copy
Apply
Tube Wrapper
Twister
Coordinate List...
Z-matrix...
Device Tools
Stash
Add
Copy
Delete
Nickel
Nickel (111)


Builder
File Edit Selection View Windows Help
Nickel (111)
Y
X
Z
B
A
C
Mirror
Quick Optimizer
Rotate
Sort
Transform by Expression
Translate
To input the translation vector using fractional coordinates, check the Fractional box below.
Fractional
To translate the configuration, first define the translation vector using the input fields below.
x (Å)
y (Å)
z (Å)
0.0
0.0
2.49248
Check the Copy box, if you want the translated atoms to be copied. Finally, press Apply in order to perform the actual translation.
Copy
Apply
Tube Wrapper
Twister
Coordinate List...
Z-matrix...
Device Tools
Stash
Add
Copy
Delete
Nickel
Nickel (111)


Builder
File Edit Selection View Windows Help
Nickel (111)
Y
X
Z
B
A
C
Coordinate Tools
Buckler
Center
Custom Passivator
Mirror
To input the normal vector and the mirror point using fractional coordinates, check the Fractional box below.
Fractional
To mirror the configuration, first specify a normal N vector and a point P that defines the mirror plane.
x (Å) y (Å) z (Å)
N 0 0 1
P 0 0.0 19.2456
A set of predefined mirror axes can be chosen from the combo box below:
Mirror axes xy
Check the Copy box, if you want the mirrored atoms to be copied. Finally, press Apply in order to perform the actual mirror operation.
Copy
Apply
Quick Optimizer
Rotate
Stash
Add
Copy
Delete
Nickel
Nickel (111)

Create the device

Click on *Device Tools* → *Device From Bulk* and keep the predefined electrode lengths corresponding to three Ni layers.



Setting up the collinear calculation and analyzing the results

In this step you will run a collinear calculation and save the corresponding state in a HDF5 file. In the next step, you will run a noncollinear calculation by using the collinear state as an initial guess. In this way, you will save a lot of computational time, since the convergence of a noncollinear calculation can be hard to achieve otherwise.

Send the device structure you have previously created to the **Script Generator** and add the following blocks:

- Add a New Calculator with the following parameters:
  - *LDA* exchange-correlation functional
  - Spin-polarized calculation
  - $4 \times 4 \times 100$ grid for k-points sampling
  - *SingleZetaPolarized* basis set
  - Electron temperature set to *2400 K*
- Add an Initial State with the following options:

–Select *User spin* and check that *Spin* for Nickel is set to 1.

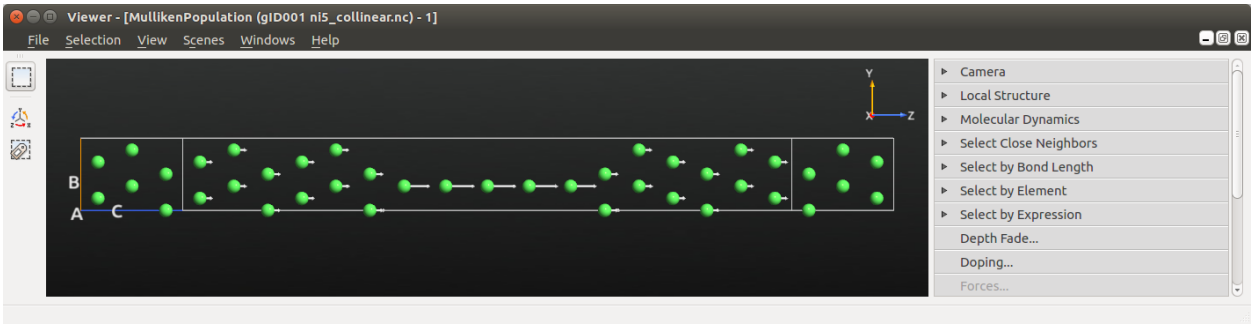
•Add an Analysis > MullikenPopulation

•Change the output file name to *Ni5_collinear.hdf5*.

Note: The high electron temperature considerably improves convergence.

Send the script to the **Job Manager** and run the calculation. Once the job is done (in serial it can take up to three hours), the LabFloor will be populated by the DeviceConfiguration and MullikenPopulation objects.

Select the MullikenPopulation and drag and drop it on the **Viewer** to visualize the converged collinear spin state.



The directions of the arrows is not so interesting or surprising - this is a collinear calculation with the electrodes polarized parallel to each other. We do however see a strong net spin polarization in the atomic wire compared to the surface layers.

Setting up the noncollinear calculation

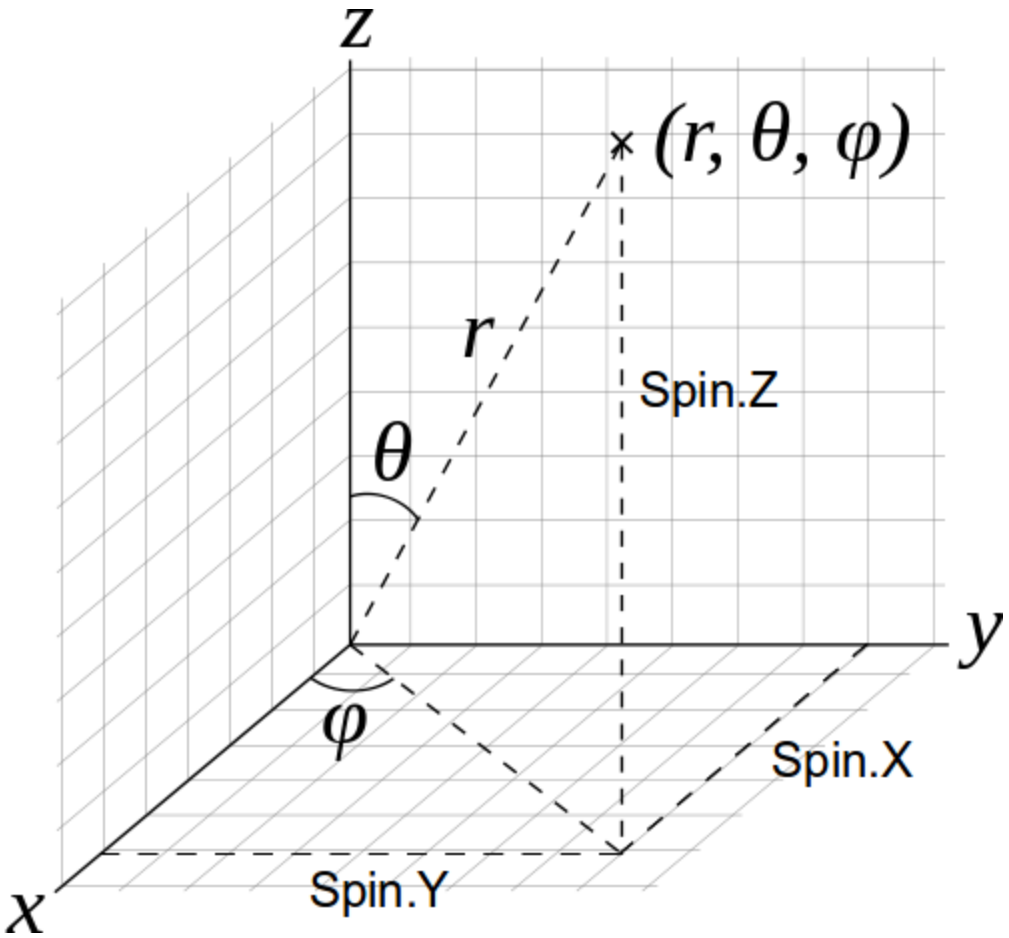
In the tutorial *Introduction to noncollinear spin* you have learned how to set up a noncollinear calculation. Here you will learn how to set up the initial spins of a noncollinear calculation based on the collinear converged result.

In order to set the initial spin state you need to specify the spin direction in physical spherical coordinates (r, ,), with the following important definitions (see also the figure below):

• is the angle with the z axis

• the polar angle in the xy plane relative to the x-axis

Warning: If you start your job by reading a previously converged spin state, as described below in this tutorial, the reference axis (Z in the picture above) for each spin of each atom is the axis of the corresponding converged spin state.


z
(r, θ, φ)
r
Spin.Z
θ
y
φ
Spin.X
Spin.Y
x

Drag and drop the DeviceConfiguration from the LabFloor to the **Script Generator** and modify the following parameters:

- In the existing New Calculator, set the spin option to *Noncollinear*
- Add an Initial State with the following options:
  - Select *User spin* and check that *Spin* for Nickel is set to 1
  - Check the option *Use old calculation* and use as filename *ni5_collinear.hdf5*
- Add Analysis > MullikenPopulation

Send the script to the **Editor** and modify the Initial State block as follows:

```
# ------------------------------------------------------------
# Initial State
# ------------------------------------------------------------

# Define the spin rotation
theta = 180*Degrees
left_spins = [(i, 1, 90*Degrees, 0*Degrees) for i in range(24)]
center_spins = [(i+24, 1, 90*Degrees-theta*(i+1)/6, 0*Degrees) for i in range(5)]
right_spins = [(i+29, 1, 90*Degrees-theta, 0*Degrees) for i in range(24)]
spin_list = left_spins+center_spins+right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

old_calculation = nlread('ni5_collinear.hdf5', DeviceConfiguration)[0]

device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
    initial_state=old_calculation,
)
device_configuration.update()
nlsave('ni5_noncollinear_out-plane.hdf5', device_configuration)
nlprint(device_configuration)
```

The setup above corresponds to an out-of-plane spin rotation between two anti-ferromagnetic electrodes, as reported in Refs.[Page 1394, 1] and[Page 1394, 2] (cf. Figure 1 in Ref.[Page 1394, 1]). In a later section you will also consider an in-plane rotation.

Note: The scaled_spin argument follows the format (atom index, initial scaled spin, ,) as documented in the *InitialSpin* entry in the Reference Manual.

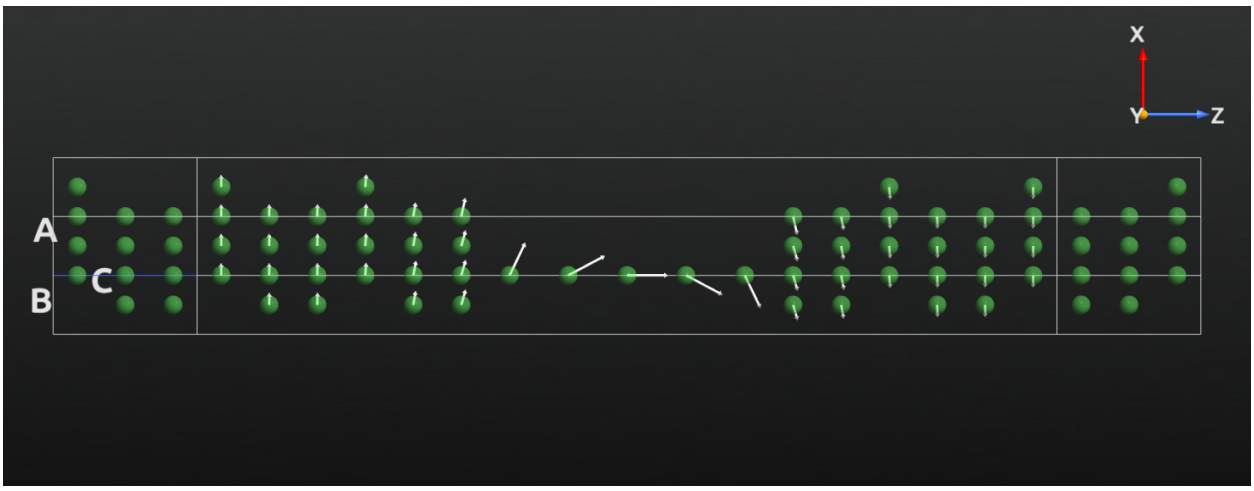
Remember that the spin orientation you enter here is relative to the collinear spin state read before. In this case, the collinear state is ferromagnetic, which corresponds to =0 and =0.

Save the script in your project directory, check that the *ni5_collinear.hdf5* file is present in the same directory and run the script.

Analyzing the results

Antiparallel configuration, out-of-plane rotation

From the LabFloor drag and drop the $\mathrm{Tr}(\rho\mathbf{S})$ MullikenPopulation object contained in *Ni5_non-collinear_out-plane.hdf5* file into the **Viewer** to visualize the converged noncollinear spin state.



For a more detailed analysis, open the text representation of $\mathrm{Tr}(\rho\mathbf{S})$ MullikenPopulation. Here you can read the spin-up, spin-down, and components for each atom.

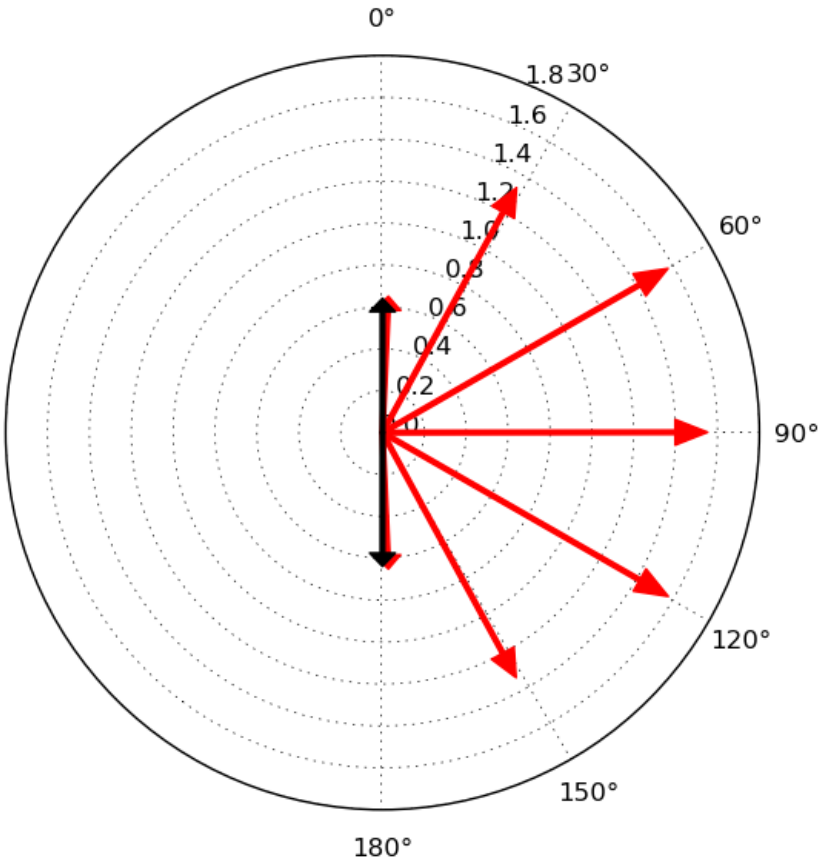
The figure below shows the direction and size of the spin state of the atoms in the wire (red arrows) and also of two atoms in the electrode extension regions (black arrows) for comparison. You can compare this plot to the results reported in Refs.[Page 1394, 1] and[Page 1394, 2].

Antiparallel configuration, in-plane rotation

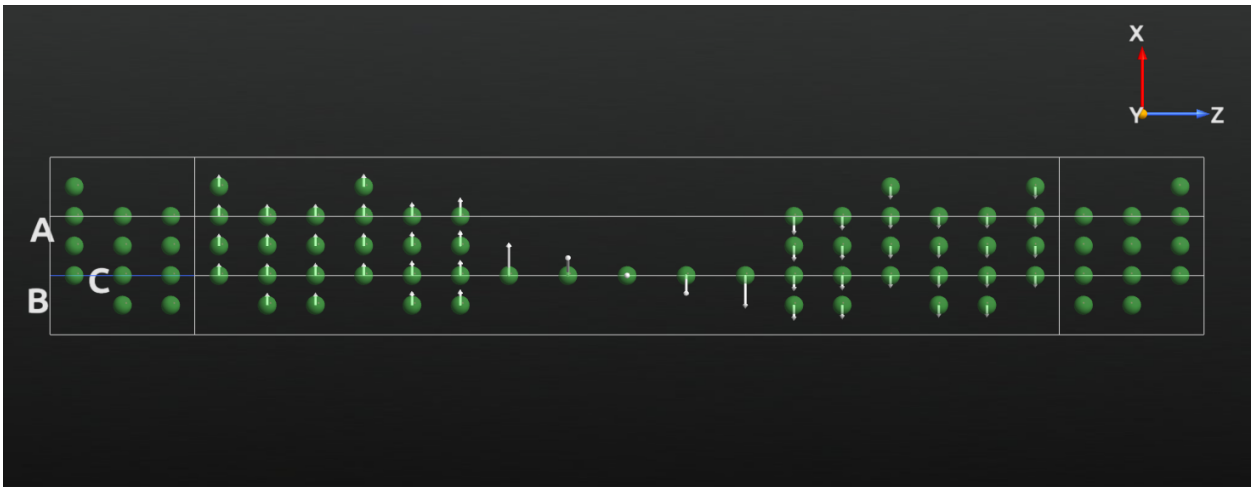
Another possibility is to rotate the spins in th xy plane by changing the angle. To achieve this, modify the script above as indicated below:

```
# Define the spin rotation
phi = 180*Degrees
left_spins = [(i, 1, 90*Degrees, 0*Degrees) for i in range(24)]
center_spins = [(i+24, 1, 90*Degrees, 0*Degrees+phi*(i+1)/6) for i in range(5)]
right_spins = [(i+29, 1, 90*Degrees, 0*Degrees-phi) for i in range(24)]
spin_list = left_spins+center_spins+right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)
```

Run the calculation for this initial spin configuration and display the corresponding $\mathrm{Tr}(\rho\mathbf{S})$ MullikenPopulation in


0°
30°
60°
90°
120°
150°
180°
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0

the **Viewer**:



Including spin-orbit coupling in noncollinear calculations

ATK also allows you to perform noncollinear calculations including spin-orbit coupling (SOC). In this example, you will investigate the effects of noncollinear spin and spin-orbit coupling to the simple case of an infinite Ni chain[Page 1394, 2]. In order to improve the convergence of the calculation, you will first set up a spin polarized calculation and only afterwards perform a noncollinear and finally a calculation with SOC, starting from the previously converged state.

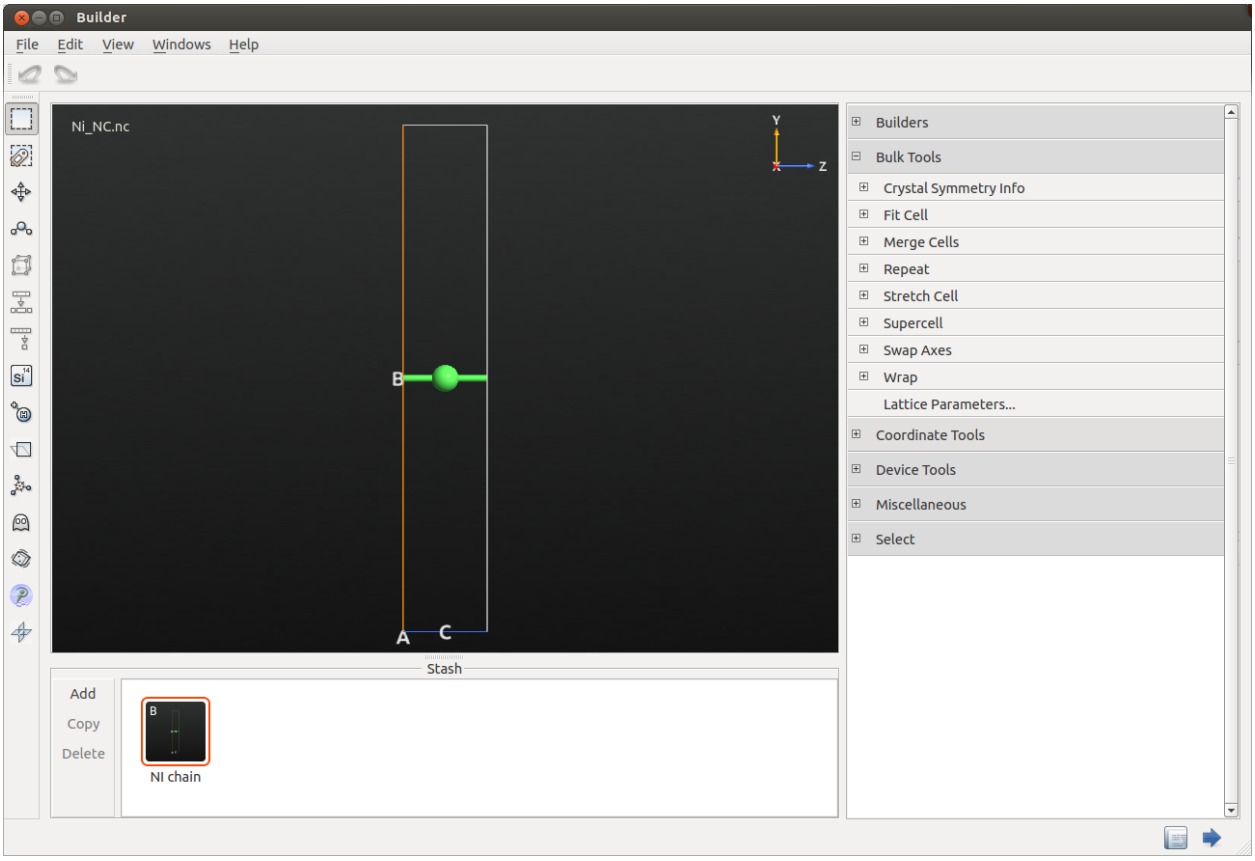
Building the Ni wire

Open the **Builder** and create a bulk configuration corresponding to an infinite chain of Ni atoms with an interatomic distance of 2.49248 Å (Ni-Ni distance).

Collinear calculation

Send the structure to the **Script Generator**, and setup the calculation as follows:

- Add a New Calculator with the following parameters
  - *PBE* exchange-correlation functional
  - Spin-polarized calculation
  - $1 \times 1 \times 13$ k-points sampling
  - *SG15-Medium* basis set
- Initial State
  - Select *User spin* and check that *Spin* for Nickel is set to 1.
- Change the output file name to *ni_collinear.hdf5*.



Note: In order to run a calculation including spin-orbit later on you need fully relativistic pseudopotentials. In this case you will use the *SG15 Pseudopotentials and basis sets* .

Once you are ready, run the calculation.

Noncollinear calculation with spin-orbit coupling

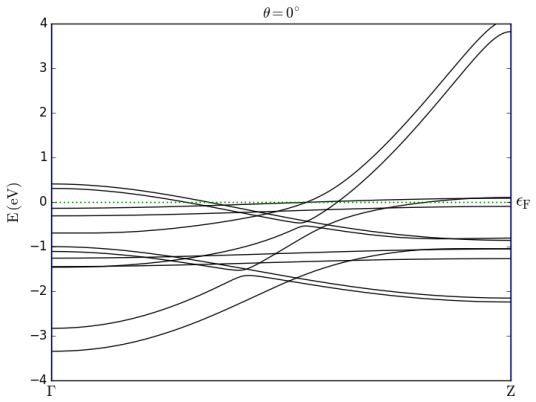
Drag and drop the DeviceConfiguration contained in the *ni_collinear.hdf5* file from the LabFloor to the **Script Generator** and modify the following parameters:

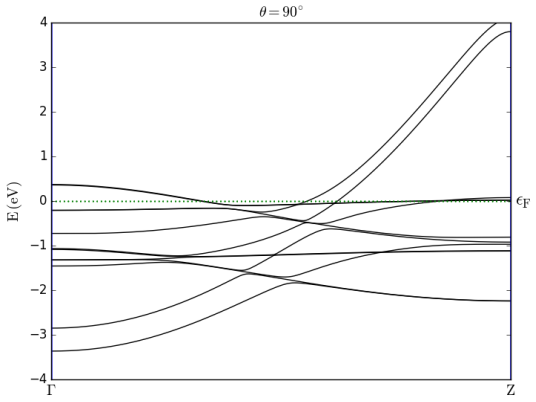
- In the existing New Calculator, set the spin option to *Noncollinear Spin-Orbit*
- Add an Initial State
  - Select *User spin* and check that *Spin* for Nickel is set to 1.
  - Check the option *Use old calculation* and use as filename *ni_collinear.hdf5*
- Add Analysis > BandStructure
  - 200 points
  - *G,Z* path

Send the script to the **Editor** and modify the Initial State block as follows:

```
# ------------------------------------------------------------
# Initial State
# ------------------------------------------------------------

# Define the spin rotation
theta = 0*Degrees
left_spins = [(0, 1, theta, 0*Degrees) ]
initial_spin = InitialSpin(scaled_spins=left_spins)

old_calculation = nlread('Ni_collinear.hdf5', BulkConfiguration)[0]

bulk_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
    initial_state=old_calculation,
)
bulk_configuration.update()
nlsave('Ni_noncollinear_phi0.hdf5', bulk_configuration)
nlprint(bulk_configuration)
```

Then run the calculation. You can notice that in the calculation the direction of the spin is parallel to the C axis (`theta=0*Degrees`). Once the calculation is done, repeat the calculation by setting the spin direction perpendicular to C (`theta=90*Degrees`), and compare the two resulting band structures (see figure below).





The results are in good agreement with the observations reported in[Page 1394, 2]. In particular, the effect of the SOC is clearly visible from the band splitting when the spin is oriented parallel to the wire axis. For both spin orientations you can also observe anticrossing of bands.

References

3.10.5 Electronic structure of NiO with DFT+U

Version: 2017

By tuning the empirical Hubbard parameter U, one can sometimes obtain the correct band gap for semiconductors even with LDA or GGA. This tutorial shows how to approach this type of calculations by taking NiO as an example, and at the same time it also introduces the density of states (DOS) functionality in QuantumATK.

Introduction

The self-interaction error is probably the most serious drawback of the LDA and GGA approximations to the exchange-correlation energy. This self-interaction can be described as the spurious interaction of an electron with itself. It has two main consequences:

- Electrons are over-delocalized;
- Band gaps in semiconductors and insulators are predicted to be much lower than their real counterpart.

The mean-field Hubbard correction, popularly called DFT+U, is a semi-empirical correction which tries to improve on these deficiencies.

In the DFT+U an additional energy term,

$$E_U = \frac{1}{2} \sum_{\mu} U_{\mu} \left(n_{\mu} - n_{\mu}^2 \right),$$

is added to the *Exchange-correlation energy*[1]. In this equation, n_{μ} is the projection onto an atomic shell and U_{μ} is the value of the *Hubbard U* for that shell. The E_U energy term is zero for a fully occupied or unoccupied shell, while positive for a fractionally occupied shell.

The energy is thereby lowered if states become fully occupied. This may happen if the energy levels move away from the Fermi Level, i.e. increasing the band gap, or if the broadening of the states is decreased, i.e. the electrons are localized. In this way, the Hubbard U improves on the deficiencies of LDA and GGA.

[1] M.ăCococcioni and S.ădeăGironcoli. Linear response approach to the calculation of the effective interaction parameters in the LDA+U method. *Phys. Rev. B*, 71:035105, Jan 2005. doi:10.1103/PhysRevB.71.035105.

The NiO crystal has a too low band gap in LDA and GGA and is one of the standard examples of how the DFT+U approximation can be used to improve the description of the electronic structure of solids[2]. In this tutorial you will compare the DFT and DFT+U models for this system using the GGA.

Further details of the Hubbard U implementation in QuantumATK can be found in the ATK Reference Manual, particularly the section *XC+U mean-field Hubbard term*.

The electronic structure of NiO calculated with DFT

NiO has a fcc crystal structure with two atoms in the unit cell. The Ni atoms have a net magnetic moment and form an anti-ferromagnetic arrangement in the (111) direction of the fcc cell. The structure can be described by a rhombohedral unit cell with 4 atoms in the basis[1]. The structure is given below in the QuantumATK *ATK-Python* format:

```
# Set up lattice
lattice = Rhombohedral(5.138*Angstrom, 33.5573*Degrees)

# Define elements
elements = [Nickel, Oxygen, Nickel, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.5 ],
                          [ 0.75,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

Copy the script and save it as `NiO.py`, or download it directly: `NiO.py`.

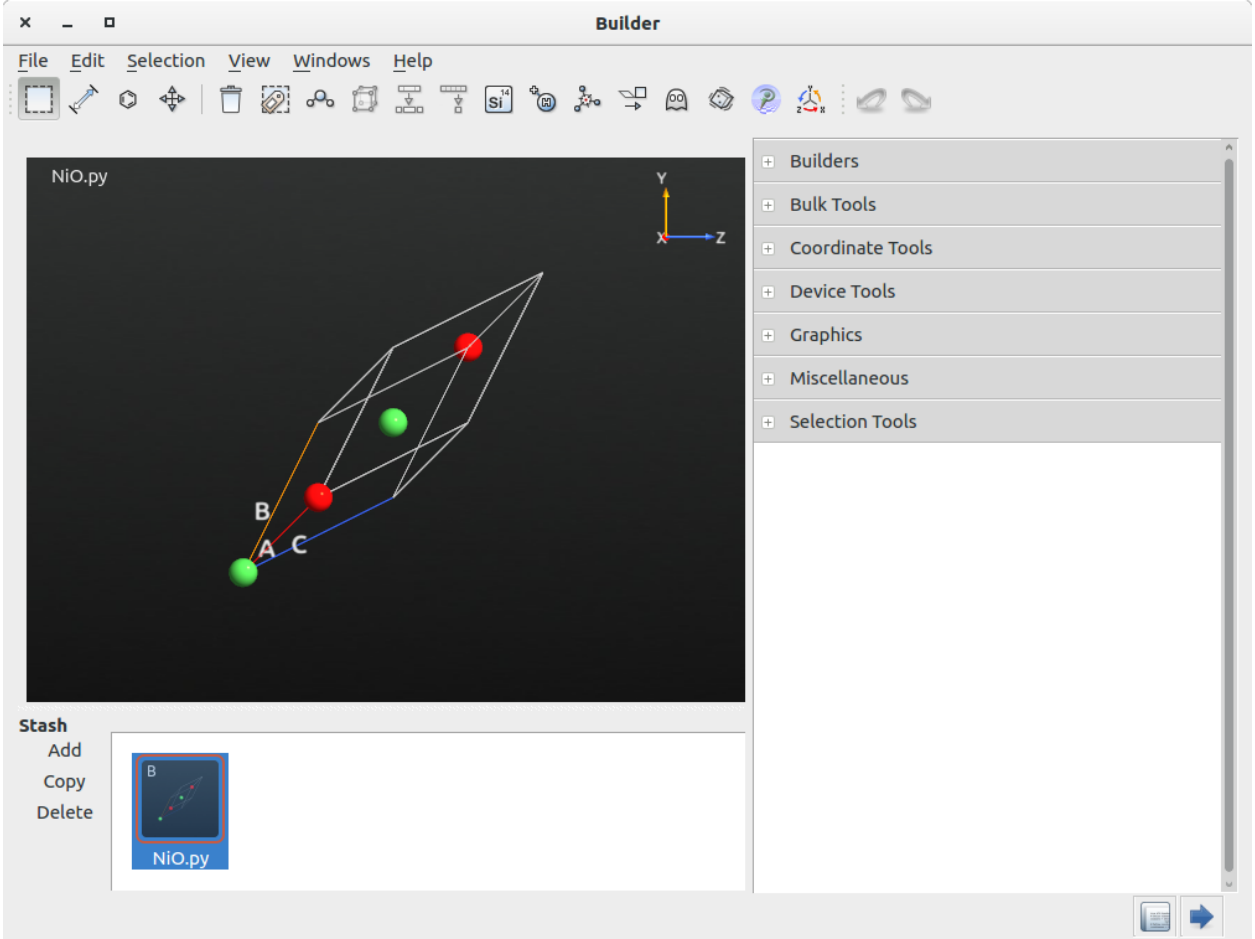
Note: The rhombohedral unit cell vectors are given as $(1, \frac{1}{2}, \frac{1}{2})a$, where a is the fcc lattice constant. The length of the rhombohedral unit cell vectors are therefore given by $\sqrt{\frac{3}{2}}a$, and are in accordance with the experimental fcc lattice constant of 4.19 Å.

Setting up the calculation

You will in this section set up a spin-polarized DFT calculation using the spin-polarized version of the generalized gradient approximation (SGGA) for the NiO electronic structure and calculate the Mulliken population and density of states. If you are not familiar with the workflow of QuantumATK you are recommended to first go through the Basic QuantumATK Tutorial.

Start up QuantumATK and drag the script `NiO.py` onto the **Builder**. The NiO crystal will be added to the Stash.

[2] A.ăB. Shick, A.ăI. Liechtenstein, and W.ăE. Pickett. Implementation of the LDA+U method using the full-potential linearized augmented plane-wave basis. *Phys. Rev. B*, 60:10763–10769, Oct 1999. doi:10.1103/PhysRevB.60.10763.

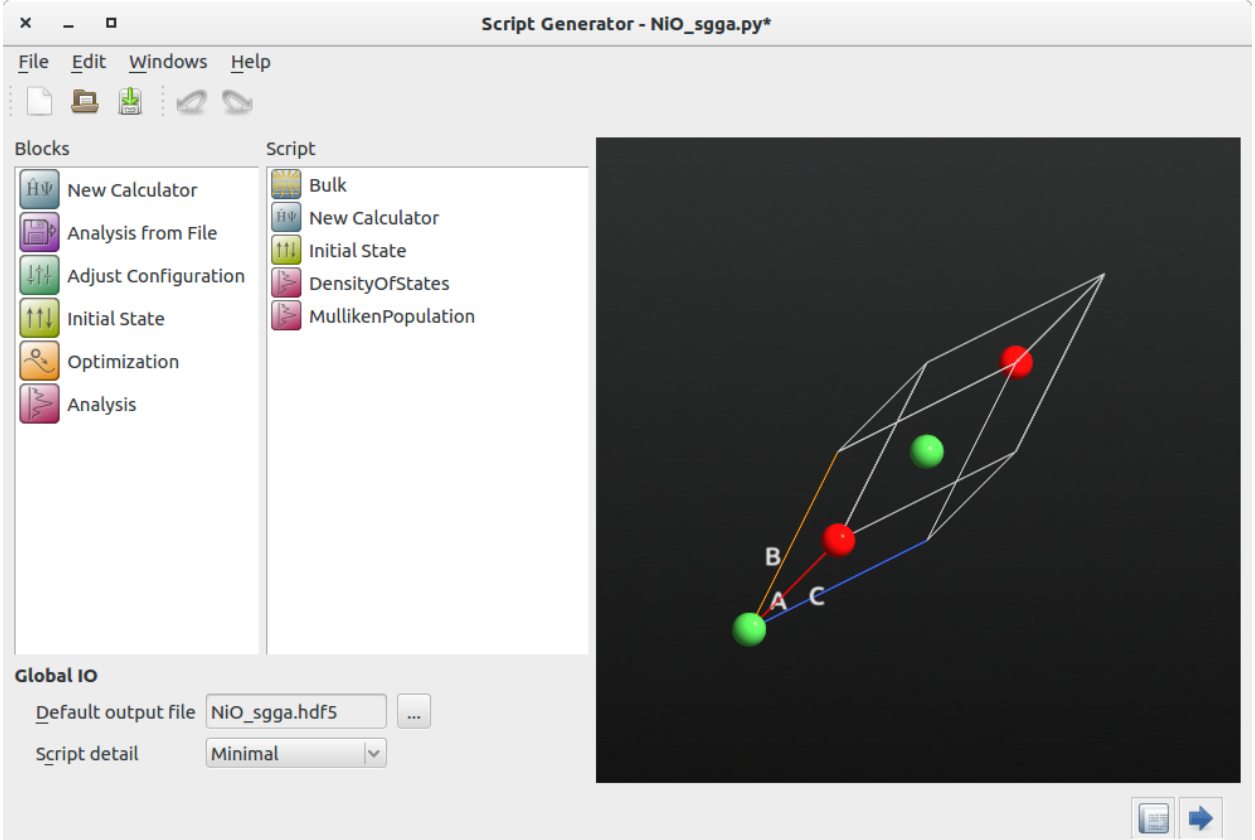


Next, click the the button and select the **Script Generator**.

Tip: Alternatively you can drag the script `NiO.py` directly onto the **Scripter** from the QuantumATK Project Files list.

Change the default output file name to `NiO_sgga.hdf5` and add the following blocks to the script:

- *New Calculator*
- *Initial State*
- *Analysis → DensityOfStates*
- *Analysis → MullikenPopulation*

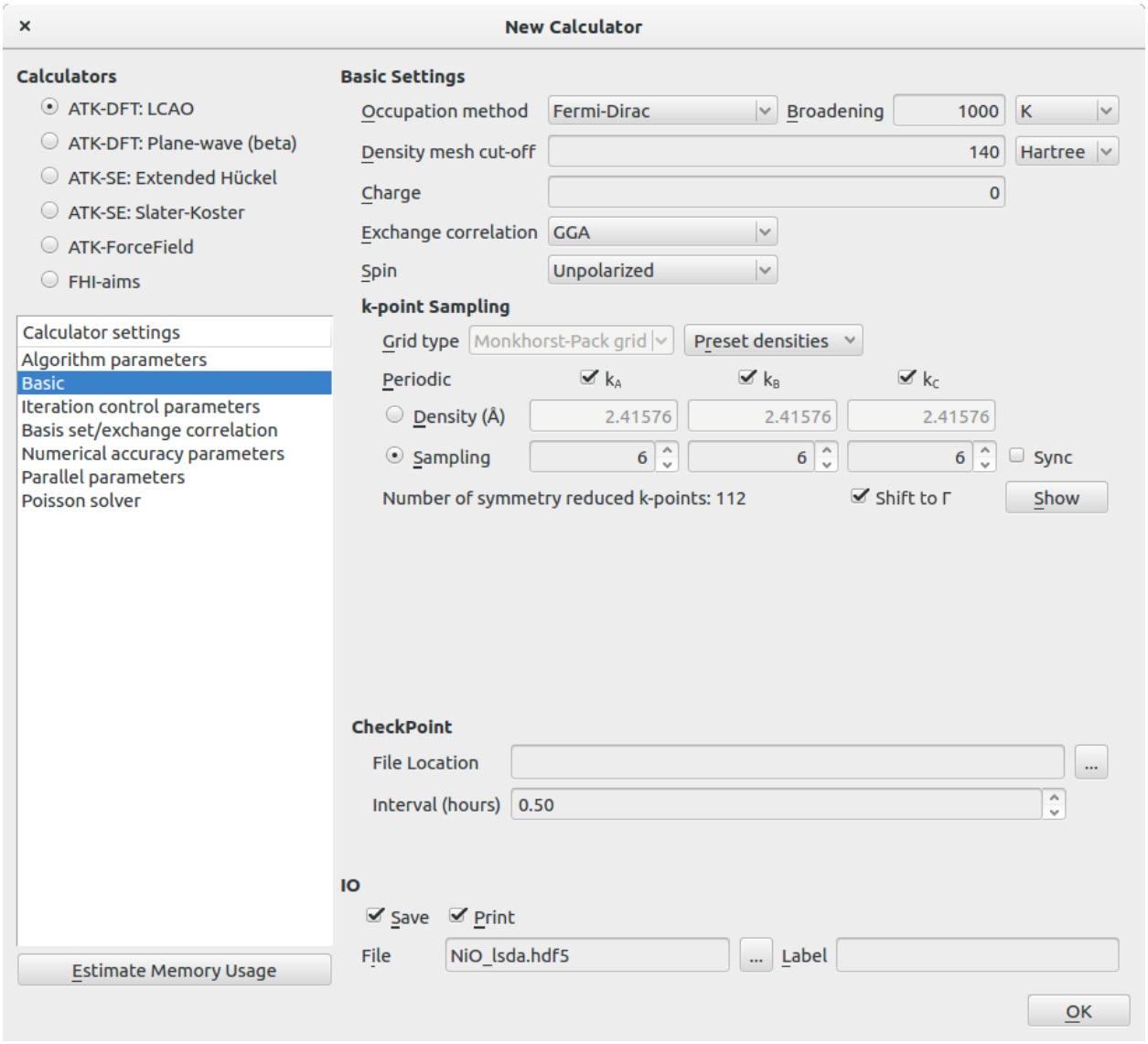


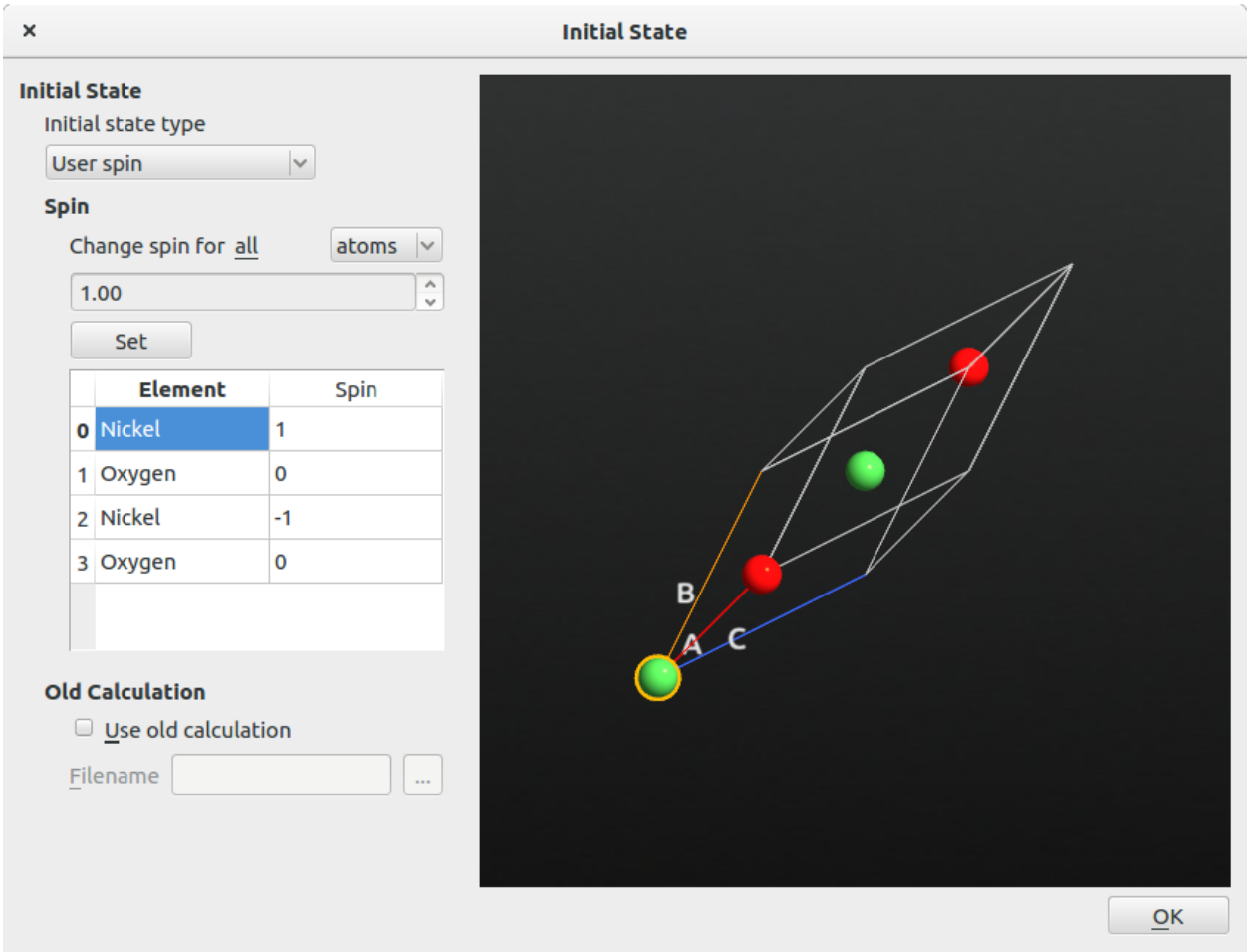
Now double-click the **New Calculator** to open the calculator widget, and set the k-points sampling to 6x6x6.

Note: When an even sampling grid is used, the grid is automatically shifted to make sure that the Γ-point is included. The automatic shift can be avoided by unticking the option *Shift to* Γ.

The next step is to set up the anti-ferromagnetic NiO spin arrangement. For this purpose, open the **Initial State** block.

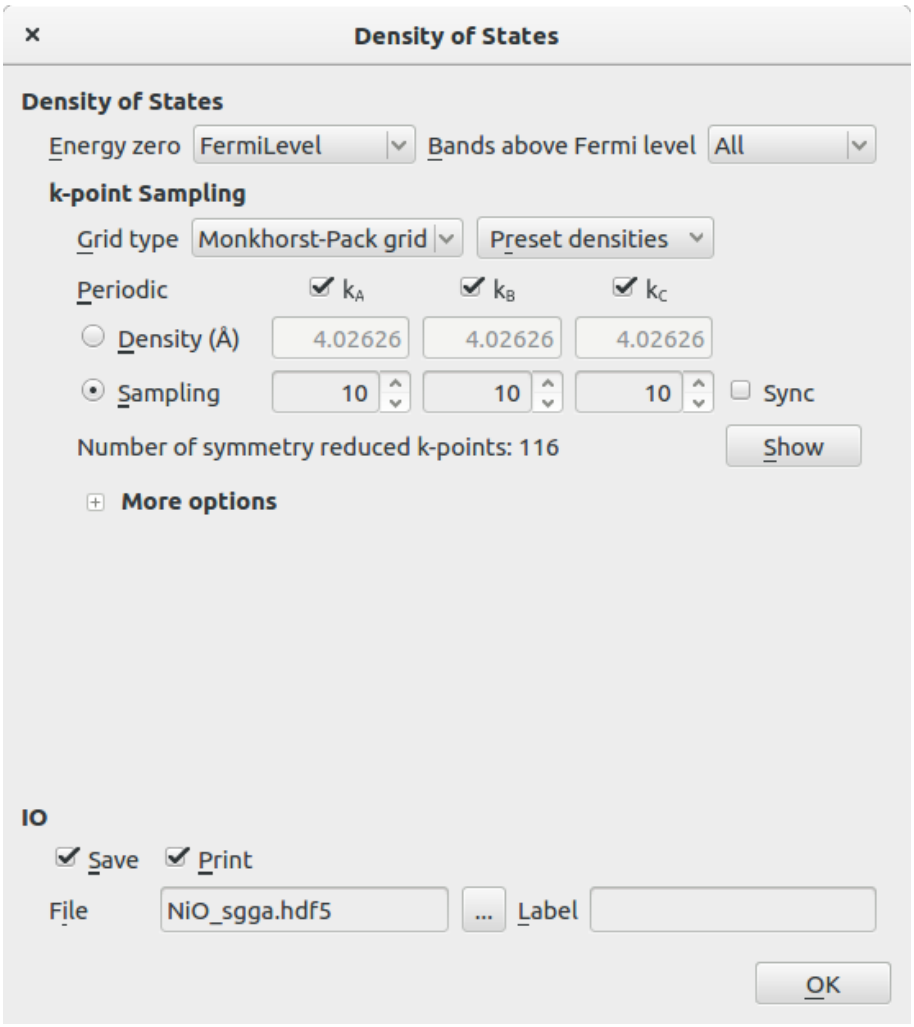
Select *User spin* – this will allow you to individually set the spin on each atom. Set opposite spins on the two nickel atoms and no spin on the oxygen atoms as illustrated below.


New Calculator
Calculators
ATK-DFT: LCAO
ATK-DFT: Plane-wave (beta)
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-ForceField
FHI-aims
Calculator settings
Algorithm parameters
Basic
Iteration control parameters
Basis set/exchange correlation
Numerical accuracy parameters
Parallel parameters
Poisson solver
Estimate Memory Usage
Basic Settings
Occupation method
Fermi-Dirac
Broadening
1000
K
Density mesh cut-off
140
Hartree
Charge
0
Exchange correlation
GGA
Spin
Unpolarized
k-point Sampling
Grid type
Monkhorst-Pack grid
Preset densities
Periodic
Density (Å)
2.41576
Sampling
6
Sync
Number of symmetry reduced k-points: 112
Shift to Γ
Show
CheckPoint
File Location
Interval (hours)
0.50
IO
Save
Print
File
NiO_lsda.hdf5
Label
OK



Important: The initial spin on each atom is given relative to the atomic spin of the element as obtained by Hunds rule. For nickel the electronic configuration of the atom is $[Ar]3d^8 4s^2$ (see periodic table in the ATK Reference Manual.). The 3d shell is fractionally occupied, and only this shell will contribute to the spin of the atom. According to Hunds rule the 3d shell has 5 electrons in the up direction and 3 electrons in the down direction, giving a total atomic spin of $2\mu_B$ for nickel.

Finally, open the **DensityOfStates** block and set the k-points sampling to 10x10x10. In general, one should choose a quite dense k-points grid for DOS analyses, in order to capture possible sharp features in the density of states.



Save the script as `NiO_sgga.py`, but do not close the Scripter window – you will need it again later.

Performing the calculation

To start the calculations, send the script to the **Job Manager** by using the button.

Choose a *Local* machine for executing the job and click start. The job will finish after 1-2 minutes and you can inspect the results.

Analysing the results

Mulliken Population

To inspect the Mulliken population reported in the calculation log file, scroll down to the end of the log file and you will find a report as shown below.

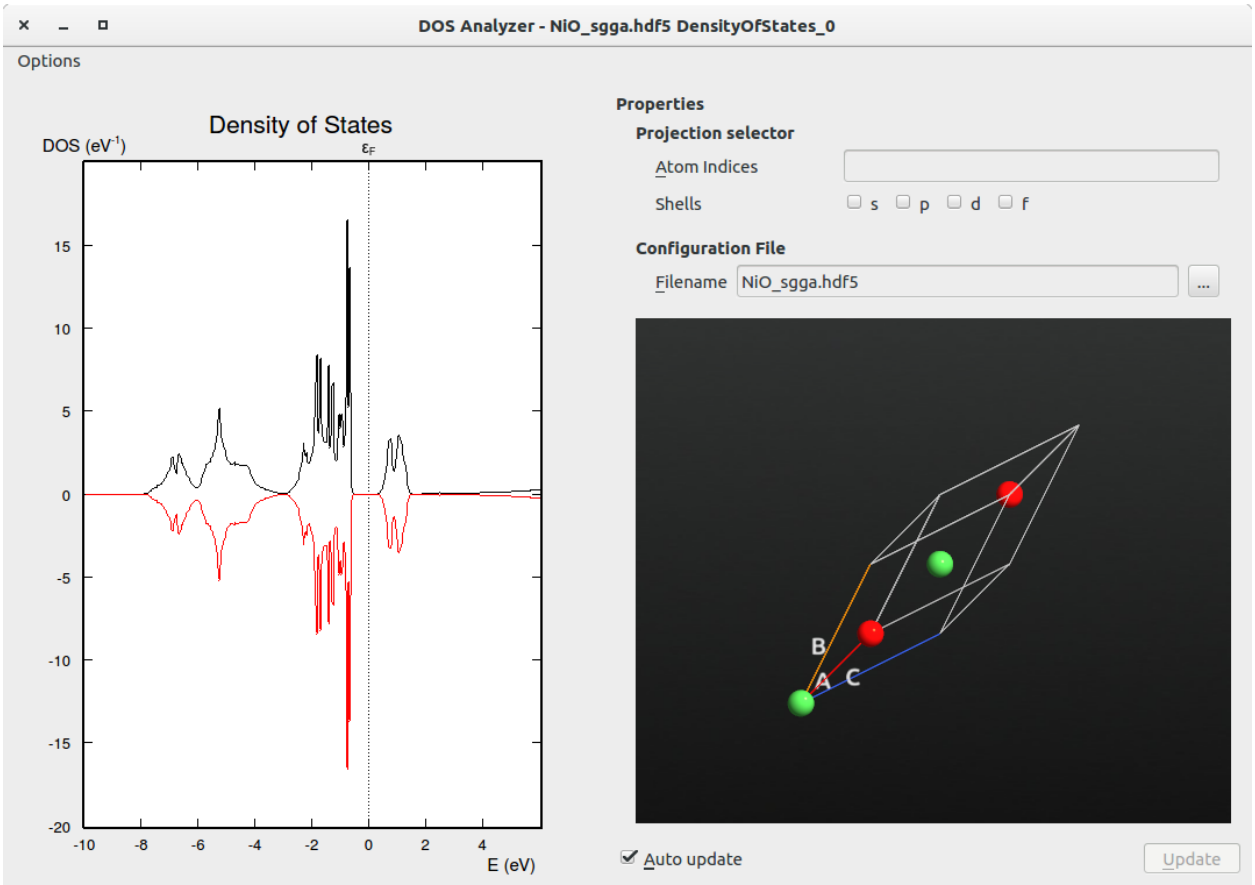
```
+-------------------------------------------------------------------------------+
|                                                                               |
| Mulliken Population Report                                                    |
|                                                                               |
| ----------------------------------------------------------------------------- |
|                          |                                                    |
| Element   Total  Shell   | Orbitals                                           |
|                          |                                                    |
|                          |      s                                             |
|   0  Ni   9.244  0.994   |  0.994                                             |
|           7.892  0.994   |  0.994                                             |
|                          |      y      z      x                               |
|                  2.997   |  0.999  0.999  0.999                               |
|                  2.996   |  0.999  0.999  0.999                               |
|                          |      s                                             |
|                  0.127   |  0.127                                             |
|                  0.127   |  0.127                                             |
|                          |     xy     zy  zz-rr     zx  xx-yy                 |
|                  4.887   |  0.986  0.986  0.964  0.986  0.964                 |
|                  3.533   |  0.980  0.980  0.297  0.980  0.297                 |
|                          |      y      z      x                               |
|                  0.042   |  0.014  0.014  0.014                               |
|                  0.044   |  0.015  0.015  0.015                               |
|                          |     xy     zy  zz-rr     zx  xx-yy                 |
|                  0.056   |  0.007  0.007  0.018  0.007  0.018                 |
|                  0.053   |  0.007  0.007  0.016  0.007  0.016                 |
|                          |      y      z      x                               |
|                  0.140   |  0.047  0.047  0.047                               |
|                  0.145   |  0.048  0.048  0.048                               |
|                          | ------------------------------------------------- |
```

Tip: The Mulliken population can also be inspected by selecting the $\mathrm{Tr}(\boldsymbol{\rho}\mathbf{S})$ Mulliken Population item on the QuantumATK LabFloor. Use the **Text Representation** tool in the right-hand panel to see the results.

The Mulliken population reports the numbers of electrons per spin and orbital, as well as the orbital sum for each atom. Note that oxygen atoms are not polarized while the two nickel atoms are polarized in opposite directions, thus forming an anti-ferromagnetic arrangement. The polarization can be calculated from the difference between the number of electrons in the spin-up channel (9.244) and that in the spin-down channel (7.892). The resulting value of $1.35\mu_B$ is in good agreement with other DFT calculations[Page 1412, 1].

Projected density of states

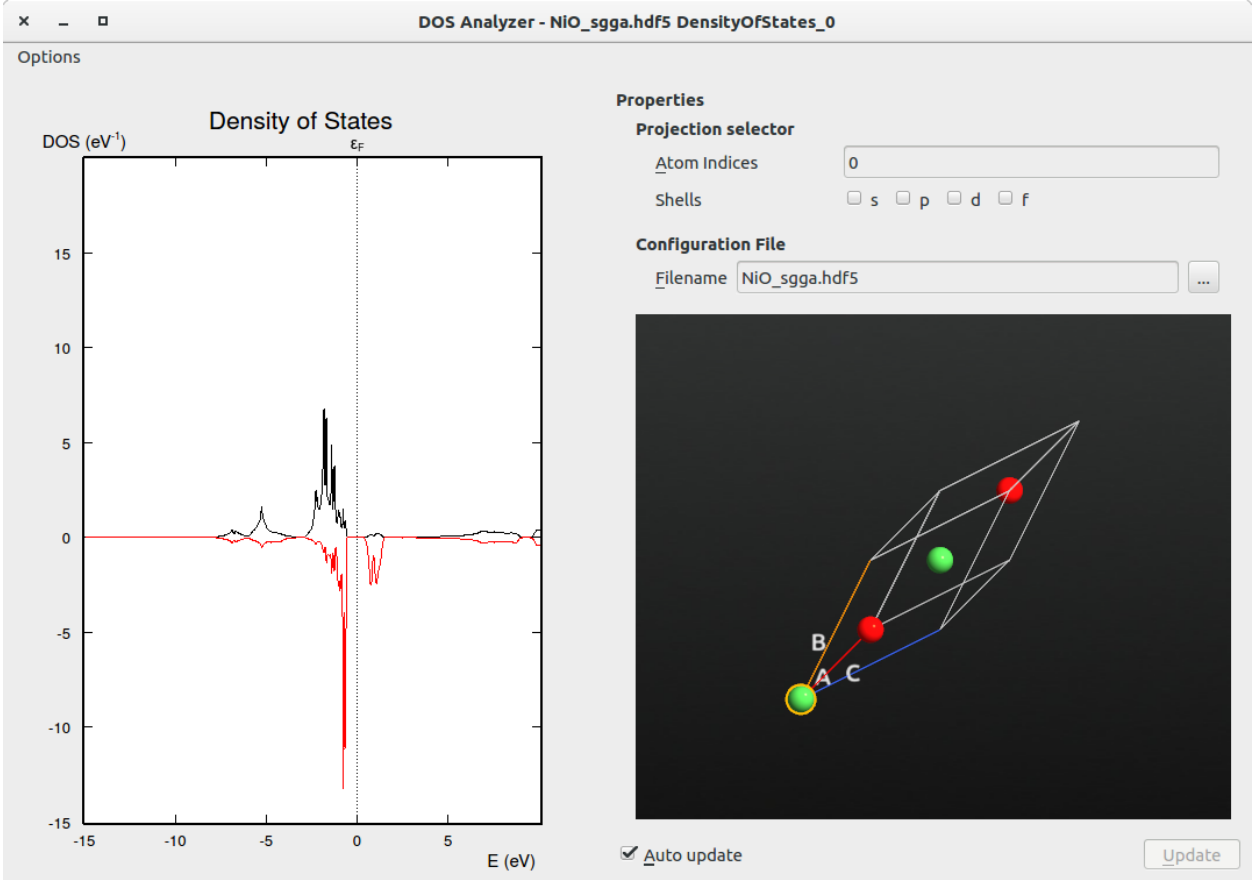
To investigate the NiO density of states (PDOS), select the **DensityOfStates** item on the LabFloor, click the $D(\epsilon)$ **2D Plot** tool in the right-hand panel. You may need to zoom in a little on the plot; use the left mouse button for this.



Tip: The plot shows the total density of states of the spin-up channel with a black line and minus the result for the spin-down channel with a red line. If you de-select Flip spin-down in the Options menu, the density of states of the spin-down channel will also be plotted on the positive axis.

The total DOS shows no difference between the two spin channels. However, you saw from the Mulliken population that the nickel atoms are spin polarized!

To inspect the projected DOS corresponding to just one nickel atom, select a nickel atom with the left-hand button of the mouse, as illustrated below.



The PDOS is simply the total DOS projected onto the selected nickel atom. The expected difference between the spin-up and spin-down DOS channels is now apparent.

Tip: You can also create combined projections by selecting multiple atoms (use the left-hand button of the mouse while holding `Ctrl`) and more than one shell.

The calculation predicts a band gap of ~0.8 eV, which is much smaller than the experimental value of 4.0 eV[3]. In the next chapter you will see how the description of the band gap is improved with the DFT+U approximation.

[3] VladimirăI Anisimov, FăAryasetiawan, and AăI Lichtenstein. First-principles calculations of the electronic structure and spectra of strongly correlated systems: the LDA+U method. *Journal of Physics: Condensed Matter*, 9(4):767, 1997. doi:10.1088/0953-8984/9/4/002.

DFT+U calculation for the NiO crystal

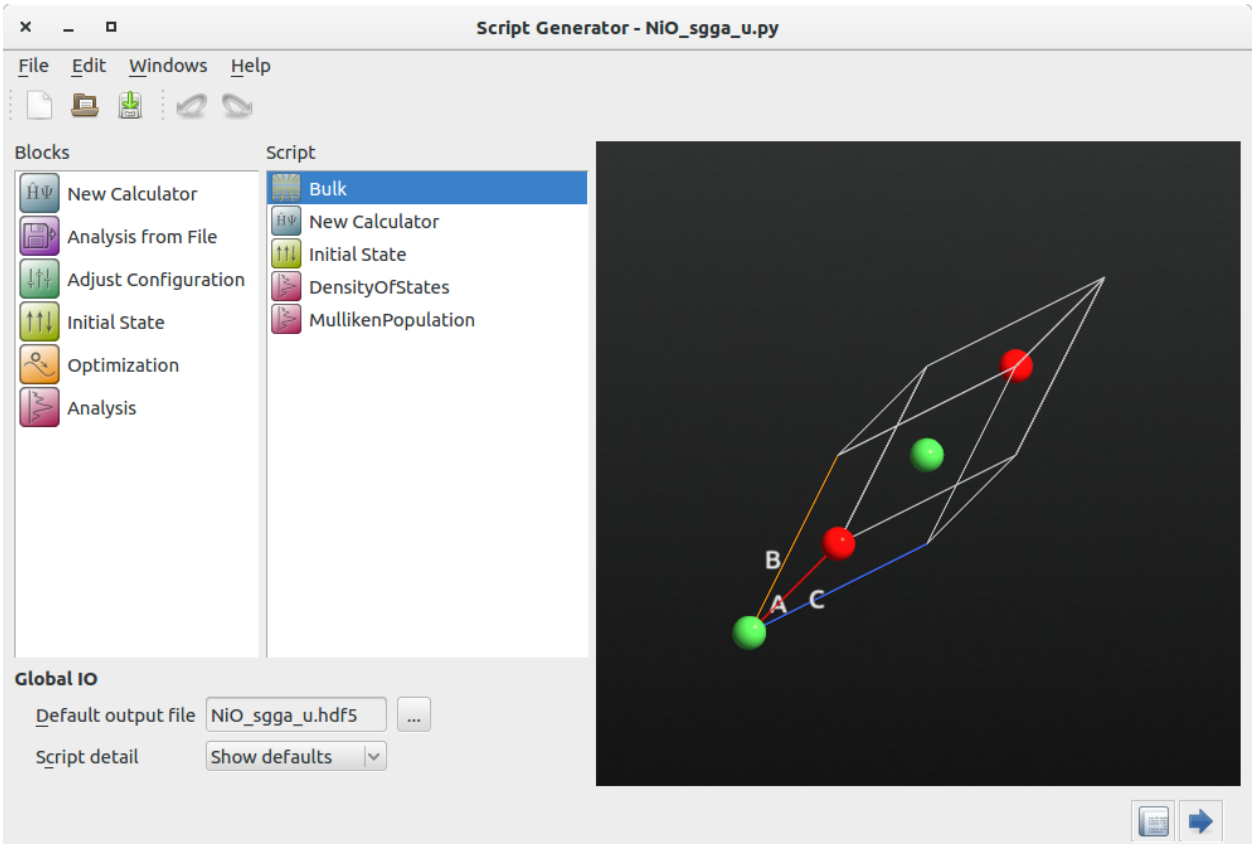
You will now perform a DFT+U calculation of the NiO crystal, using U = 4.6 eV for the nickel *d*-states, as proposed in[Page 1412, 1].

Calculations

You will need to modify the script generated above for the SGGA calculation. Open the **Scripter** used in the previous chapter and make the following modifications:

•Change the default output file name to `NiO_sgga_u.nc`.

•Change the **Script detail** to *Show defaults*.

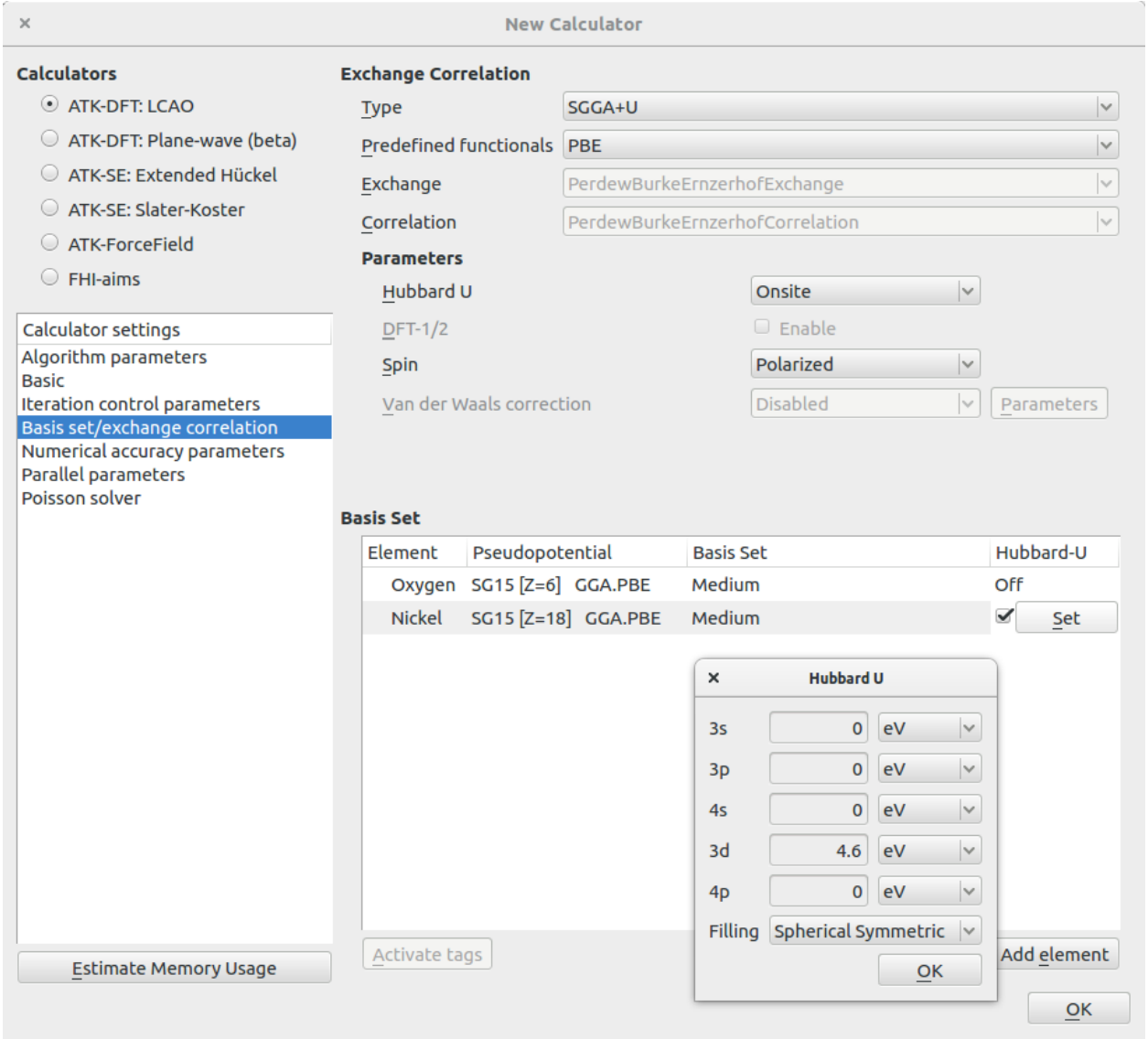


Open the **New Calculator**:

•Switch to the **Basis set/exchange correlation** tab.

•Change the option **Hubbard U** from *Disabled* to *Onsite*.

•In the **Basis set** section, click **Set** in the column on the right-hand side on the row corresponding to nickel, and set the Hubbard U parameter for the 3d orbital to 4.6 eV.



Next transfer the script to the **Editor** using the send to button, in order to check that all the parameters are properly set.

In the editor locate the line where the nickel basis is defined and check that the hubbard U parameter is set to 4.6 eV for the `nickel_3d` and `nickel_3d_0` orbitals.

```
NickelBasis = BasisSet(
    element=PeriodicTable.Nickel,
    orbitals=[nickel_3s, nickel_3p, nickel_4s, nickel_3d, nickel_3p_0, nickel_3d_0,␣
↪nickel_4p],
    occupations=[2.0, 6.0, 0.0, 8.0, 1.0, 1.0, 0.0],
    hubbard_u=[0.0, 0.0, 0.0, 4.6, 0.0, 4.6, 0.0]*eV,
    dft_half_parameters=Automatic,
    filling_method=SphericalSymmetric,
    onsite_spin_orbit_split=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]*eV,
    pseudopotential=NormConservingPseudoPotential("normconserving/sg15/gga/28_Ni.upf"),
    )
```

Now save the script as `NiO_sgga_u.py` and execute the job using the **Job Manager**.

Analyzing the results

First inspect the Mulliken population in the log file. You should find a magnetic moment of $1.80\mu_B$ on a nickel atom, which is in good agreement with the experimental result of $1.64 - 1.9\mu_B$[Page 1421, 3].

To determine the band gap, inspect the printed density of states in the log file. You should find that the DOS is zero in the range -1.69 to 1.69 eV, corresponding to a band gap of 3.38 eV. This is much higher than the SGGA value of 0.8 eV, and in better agreement with the experimental value of 4.0 eV[Page 1421, 3].

Comparing the DFT and DFT+U projected DOS

The final step is to compare the nickel PDOS obtained with DFT and DFT+U. You will here use Python scripting to perform the analysis.

The script `dos-comparision.py` performs the analysis. It is shown below:

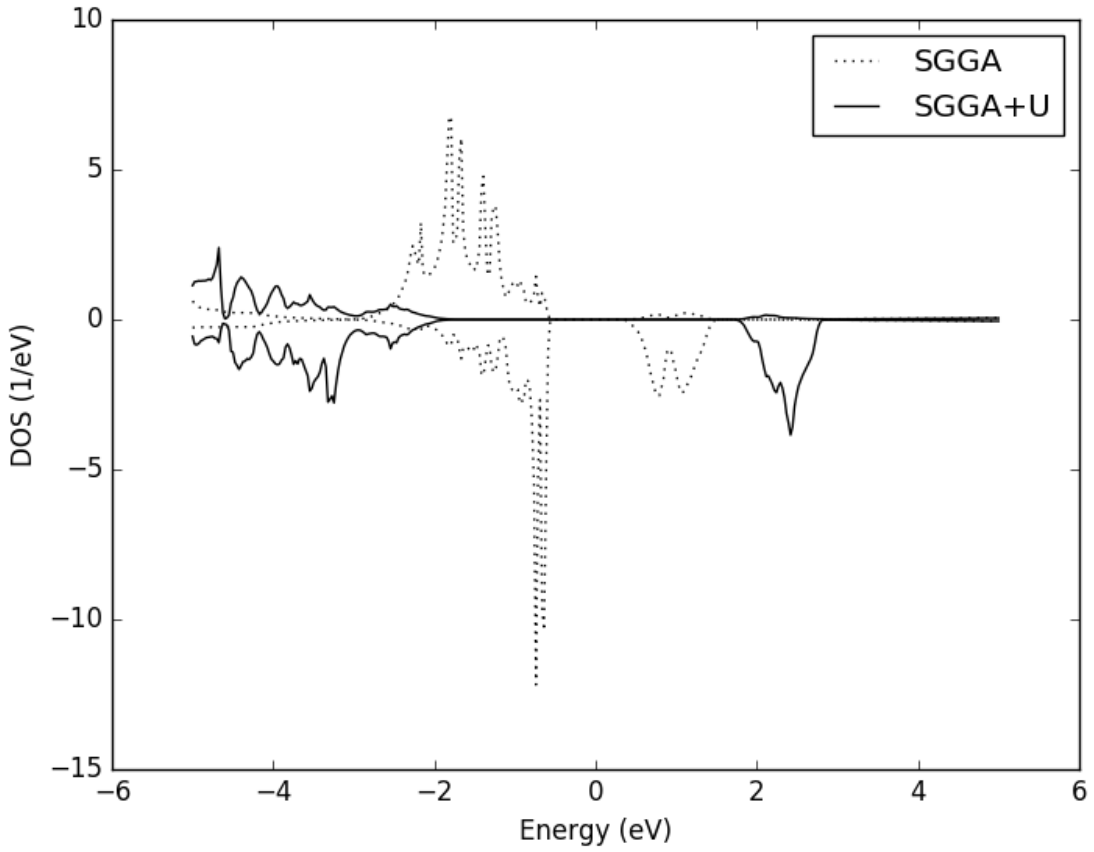
```
#read in the dos object
dos = nlread('NiO_sgga.hdf5',DensityOfStates)[0]
#generate some energies
energies = numpy.linspace(-5,5,400)*eV
#calculate the spectrum
n0_up = dos.tetrahedronSpectrum(energies=energies,
                                spin=Spin.Up,
                                projection_list = ProjectionList([0]))

n0_down = dos.tetrahedronSpectrum(energies=energies,
                                  spin=Spin.Down,
                                  projection_list = ProjectionList([0]))
e = dos.energies()

#do the same for GGA+U
dos_u = nlread('NiO_sgga_u.hdf5',DensityOfStates)[0]
n0_up_u = dos_u.tetrahedronSpectrum(energies=energies,
                                    spin=Spin.Up,
                                    projection_list = ProjectionList([0]))

n0_down_u = dos_u.tetrahedronSpectrum(energies=energies,
                                      spin=Spin.Down,
                                    projection_list = ProjectionList([0]))

#plot the spectrum using pylab
import pylab
#first plot the up component with dots
pylab.plot(e.inUnitsOf(eV), n0_up.inUnitsOf(eV**-1), 'k:',label = 'SGGA')
#now plot the down component with negative values and dots
pylab.plot(e.inUnitsOf(eV), -1.*n0_down.inUnitsOf(eV**-1), 'k:')
#now plot the GGA+U up  components with solid
pylab.plot(e.inUnitsOf(eV), n0_up_u.inUnitsOf(eV**-1),'k',label = 'SGGA+U')
#now plot the  GGA+U down component with negative values and solid
```

(continues on next page)

(continued from previous page)

```
pylab.plot(e.inUnitsOf(eV), -1.*n0_down_u.inUnitsOf(eV**-1),'k')
#show legends
pylab.legend()
pylab.xlabel("Energy (eV)")
pylab.ylabel("DOS (1/eV)")
pylab.show()
```

Download the script and execute it using the **Job Manager**. The following plot is produced, illustrating the projected DOS for the nickel atom obtained using SGGA and SGGA+U. Notice the large difference in band gap between the two calculations (region of zero DOS around the Fermi level, at 0 eV energy).



Note: The plotting is based on the matplotlib package which is part of QuantumATK, see *Plotting using pylab* for more information.

References

3.11 Materials, Surfaces and Chemistry

3.11.1 Polymer Builder

Version: Q-2019.12

Downloads & Links

`sio2_nanoparticle.hdf5` `sio2_surface.hdf5` *`ForceCappedEquilibration`* *`PolymerSequence`* *`OPLSPotentialBuilder`* *`DreidingPotentialBuilder`* More About Polymer Simulations

In this tutorial you will learn how to use the **Polymer Builder** to build polymers such as homopolymers and polymer blends, and how to include embedding of molecules, nanoparticles, or a surface, all using the Nanolab GUI. The Polymer Builder is introduced with the Q-2019.12 version of QuantumATK. After building the inital polymer model, you can proceed to perform realistic atomistic simulations for studies of the polymer properties such as polymer melting, glass transition temperature, mechanical properties, miscibilty, and so on.

This tutuorial guides how to build an initial polymer using the **PolymerBuilder** script block located in the **Script Generator**.

Contents

•*Polymer Builder*

–*Procedure of the Polymer Builder*

∗*Homopolymer*

∗*Polymer blend*

∗*Polymer with embedded molecules*

∗*Nanoparticles in a melted polymer*

∗*Polymers around a surface*

∗*Customized monomers*

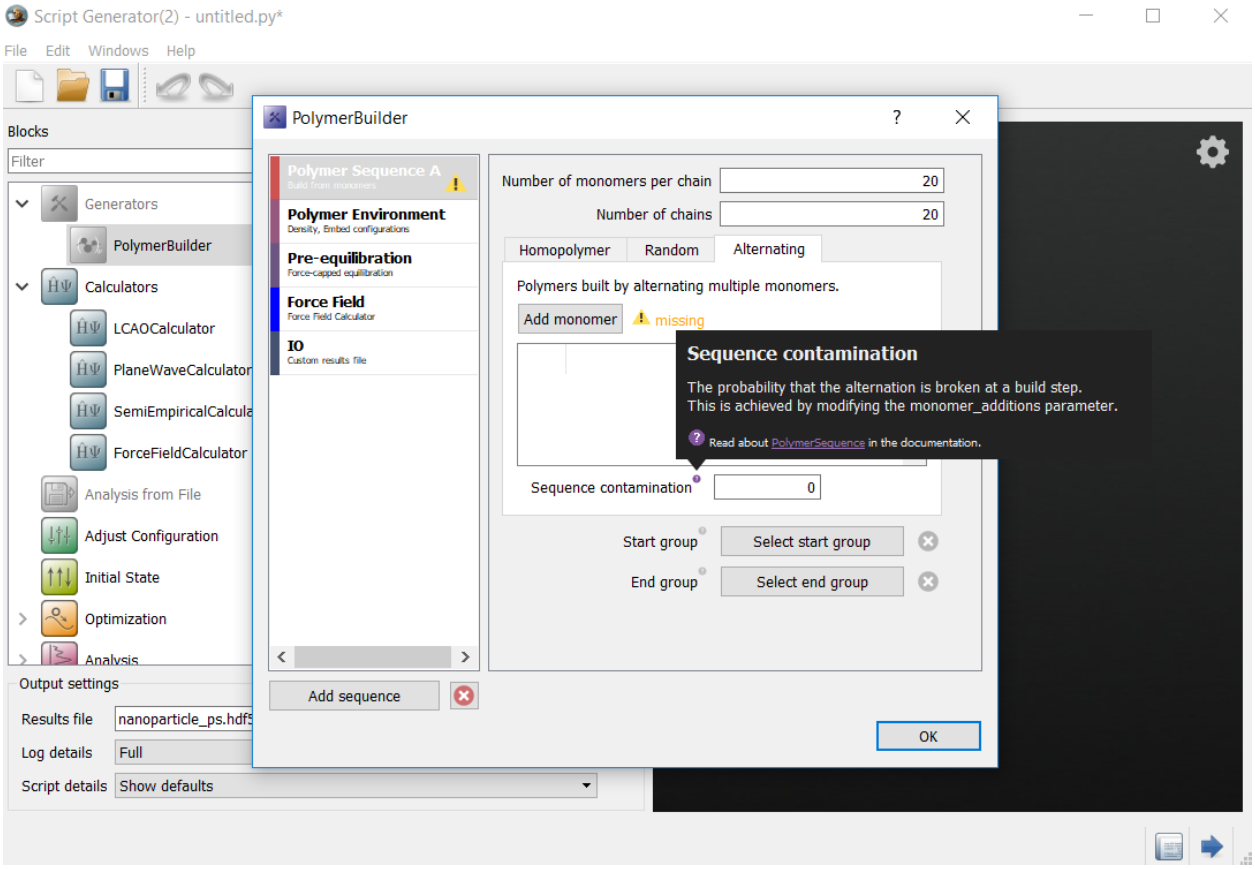
Procedure of the Polymer Builder

1. Open the **Script Generator**.
2. Add the **PolymerBuilder** block under the **Generator** into the the templates.

Note: The **PolymerBuilder** object is located in **Script Generator** not in **Builder**. Monomers of polymer sequences are stored in the **Monomer Database** which is newly implemented in Q-2019.12. You can also create and

add customized monomers into the **Monomer Database**. You will learn it in the *Customized monomer* section of tutorial.

3. Decide the polymer sequences. In the **Polymer Sequence A**, you see `Homopolymer`, `Random`, `Alternating`. The `Homopolymer` builds one main monomer chain, while `Random` and `Alternating` can build the polymer chains consisting more than two monomers randomly or alternatively. When you click the small question sign, you get a short description such as the below figure.



Tacticity or Sequence contamination is in a range of 0 and 1. And the number of monomer per chain and the number of chains are each 20 as a default. You can also add the different polymer sequence clicking the button of `Add Sequence`. When you select the `monomer`, `start group`, or `end group`, **Monomer Database** will be pop-up automatically.

4. In the **Polymer Enviroment**, you can set up the polymer density and embeded host configuration and molecules. PolymerBuilder builds melted polymer with a defined density controlling a periodic box size. But in the embedded case with a host configuraiton, polymers are packed in the periodic box of host configuration. And many applications include not only polymer, but also nanoparticles, surfaces or molecules. You can setup to build polymer with embedded other material types.

5. **Polymer Monte Carlo** *`PolymerMonteCarloBuilder`* runs with above defined conditions.

6. **Pre-equilibration** is performed after Polymer Monte Carlo. It is using the **Force-Capped Molecular Dyanmics** `ForceCappedMolecularDynamics`. Initial configuration by Polymer Monte Carlo includes overlapped atoms or artifically closed atoms. Optimization and Force-capped molecular dynamics run to make a reasonable configuration.

7. PolymerBuilder runs by `Dreiding` or `OPLS` force field. The desired polymer system should be parameterized in `Dreiding DreidingPotentialBuilder` or `OPLS`*`OPLSPotentialBuilder`* force field types.

8. Run it using the **Job manager**. Polymer is usually much larger than molecular/bulk system. Due to the number of

atoms and scale of size, it takes a bit longer time, especially for the pre-equilibrium step.

Homopolymer

You will build *polyvinylidene fluoride (PVDF)* polymer as an example.

•Open the **PolymerBuilder**.

•In the **Polymer Sequance A**, see `Homopolymer`. You can create a polymer model from a single monomer in this section.

•Click the `Select monomer` to load the monomer. It is pop-up the **Monomer Database** which includes common monomers.

•Select `tetrafluroethylene` to build the *polyvinylidene fluoride (PVDF)*. There is an option to define the tactic ratio. `Isotactic` is `0` and `Syndiotactic` is `1`. Default is `Atatic` with `0.5` ratio of isotactic and syndiotactic. You can also customize the atatic ratio in the `Custom`.

•Move to the **Polymer Environment**. You can set the desired density with a unit of g/cm ^{3}. Change the density into `1.78` g/cm ^{3} in experiment for *PVDF*.

You can also see the **Pre-equilibration** section. After generating an initial polymer model by Polymer Monte Carlo, PolymerBuilder follows the pre-equilibration using the Force-capped equilibration method. It is for preventing the overlapping or artifically closed atoms. Detailed information is documented in the manual page, *ForceCappedEquilibration*. You will keep the default condition for others.

•Change the output file as `pvdf.hdf5` and open the **Editor** to confirm the settings as an input script.

You can see the following information in the input python file.

The first part of script shows the definition of monomer configuration as *MoleculeConfiguration* with information of tags and bonds using the `Dreiding` or `OPLS` force field. QuantumATK PolymerBuilder includes the `Dreiding` or `OPLS` force field. The `Dreiding` force field is much simpler than `OPLS` force field. PolymerBuilder can automatically indicate the types of `Dreiding` force field. Therefore, when you use `Dreiding` force field, you dont need to add the tags manually. But it is too simple not to describe the tacticity or partial charge. On the other hand, the `OPLS` force field is able to consider the tacticity and partial charge of atomic types. It is more complex but more accuracy. You can use it adding tags connected to the `OPLS` parameters for monomer configuration.

PolymerBuilder uses the `Dreiding` and `OPLS` force field parameters to build a polymer model using the Polymer Monte Carlo and Force capped equilibration.

In the following section as shown in the below script, polymer Monte Carlo builder is defined. After building an initial polymer, pre-equilibration will be performed using the force capped equilibarion method. Force capped equilibration will run 4 cycles. Each simulation runs 40,000 times with 0.5 fs timestep as a default. The `OPLS` potential is used in Force capped equilibration.

```
# ------------------------------------------------------------
# Polymer Monte Carlo Builder
# ------------------------------------------------------------
polymer_builder = PolymerMonteCarloBuilder(
    polymer_sequence=polymer_sequences,
    density=1.7800*gram/cm**3,
    monte_carlo_temperature=300.0*Kelvin,
    angle_sampling_points=20,
    repack_molecules=False,
    )
```

(continues on next page)

(continued from previous page)

```
bulk_configuration = polymer_builder.buildPolymerConfiguration()

# ----------------------------------------------------------------
# Calculator for Force Capped Equilibration
# ----------------------------------------------------------------
potential_builder = DreidingPotentialBuilder(
    include_electrostatic=False,
    )
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ----------------------------------------------------------------
# Force Capped Equilibration
# ----------------------------------------------------------------
equilibration_method = ForceCappedEquilibration(
    temperature=300.00*Kelvin,
    md_steps_per_force_capped_simulation=40000,
    md_time_step=0.50*fs,
    force_capped_simulations=4,
    starting_factor=1.040,
    ending_factor=0.800,
    fixed_indices=None,
    )
bulk_configuration = equilibration_method.runEquilibration(bulk_configuration)
nlsave('pvdf.hdf5', bulk_configuration)

# ----------------------------------------------------------------
# Calculator
# ----------------------------------------------------------------
potential_builder = OPLSPotentialBuilder(
    lennard_jones_cutoff=10.0*Angstrom,
    electrostatic_accuracy=0.0001,
    )
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)
nlsave('pvdf.hdf5', bulk_configuration)
```

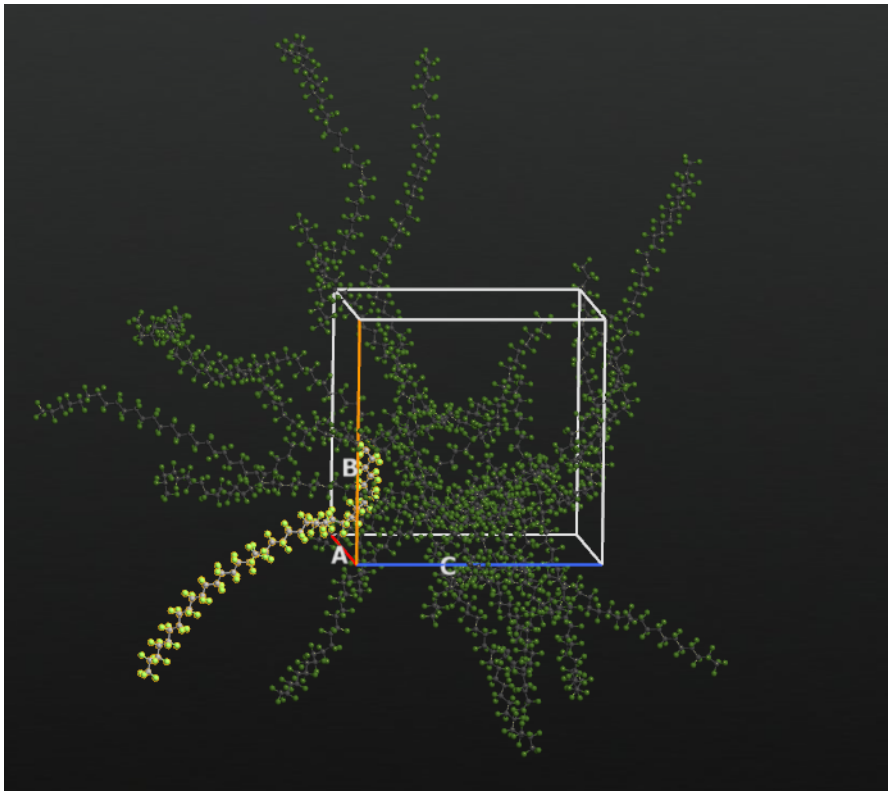
Now you are ready to run the script. Go to the **Job manager** after saving the script as `pvdf.py`. It takes around 15 minutes on your laptop.

After the job finishes, you can visualize the results using the **Viewer**. .. , when you select the `pvdf.hdf5`, .. you can see 2 objects of `BulkConfiguration_0` and `BulkConfiguration_1` .. on the **labfloor**. .. `BulkConfiguration_0` is the configuration after polymer Monte Carlo .. and `BulkConfiguration_1` is the configuration after pre-equilibration process. .. You can compare them in the **Compare Configurations** plug-in. .. You can also visualize it using the **Viewer**.

As shown in the below figure, you can see the 20 chains with 20 monomers per each chain of *PVDF* with `1.78` g/cm ^{3} density. The periodic box is automatically controlled to fit the requested density.

You may need to increase the *Fuzz factor* (ex. `1.15`) in the **Graphics Properties** of **Viewer** to see reasonable polymer chains. And the default configuration shows unwrapped configuration. When you check the *wrap atom*, you can see

the wrapped configuration in a periodic box.



Polymer blend

Pactically, polymer blend is important to mix different polymers. You can easily build the polymer blend using the **PolymerBuilder**. We will blend *Poly Methyl Methacrylate (PMMA)* and *Poly Vinyl Chloride (PVC)* in a periodic box as an example.

•Open the **PolymerBuilder**.

•Click `Add sequence` in the botton side of **PolymerBuilder** widget.

Now you can make two different polymer sequences. If you need more different polymer sequence, you can add it more from here.

•In the **Polymer Sequance A**, select the `Methyl Methacrylate` in the Monomer Database.

•In the **Polymer Sequance B**, select the `Vinyl Chloride` in the Monomer Database.

To demonstrate it in a short time, you will blend simply two different polymers of 5 chains possessing 20 monomers per chain.

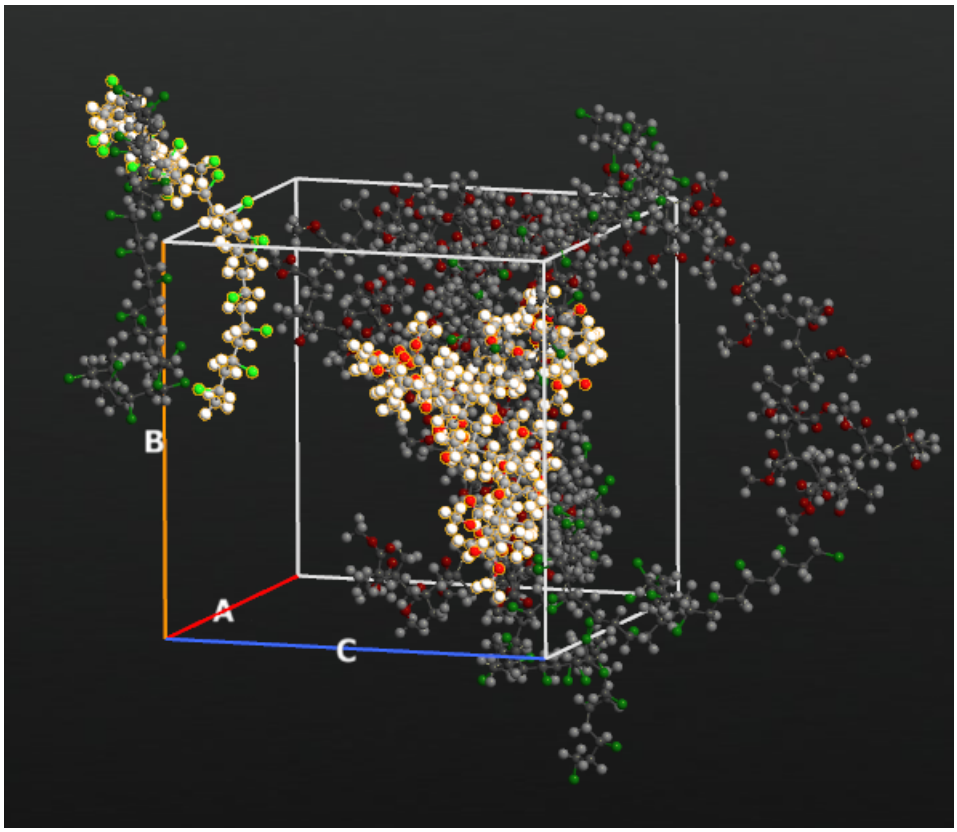
•Change the number of chains into 5 in the Polymer Sequance A and B.

•Change the output file name as `pmma-pvc.hdf5`. Other options are default.

•Open the **Editor**. You can see molecular configuration and parameters of 2 monomers from polymer sequence a and b in the script.

•Save the script as `pmma-pvc.py` and run it in the **Job Manager**.

It takes around 20 minutes in your laptop. As shown in the following figure, you get the polymer blend system including 5 *PMMA* and 5 *PVC* chains.



Polymer with embedded molecules

In many cases, polymer is melted in liquid. The tutorial guides how to embed molecules into the polymer, as building the *PMMA* polymers with methane molecules as a simple example.

It starts with one monomer and one molecule.

•In the **Polymer Sequance A**, select the `Methyl Methacrylate` in the Monomer Database. And change the number of chain as `5`.

•Open the **Builder** and import `Methane` **From Database** of Molecules.

•In the **Polymer Environment**, click `Add molecule from file` button.

You see saved configurations in your project and also see *MoleculeConfiguration Methane* in the `Builder_Stash.hdf5`.

•Select *MoleculeConfiguration Methane* and modify `Count` as `10`.

•In the **Force Field**, change `OPLS` into `Dreiding` force field. Because the new embedded methane will run using the Dreiding force field without assigning the OPLS tag for methane manually.

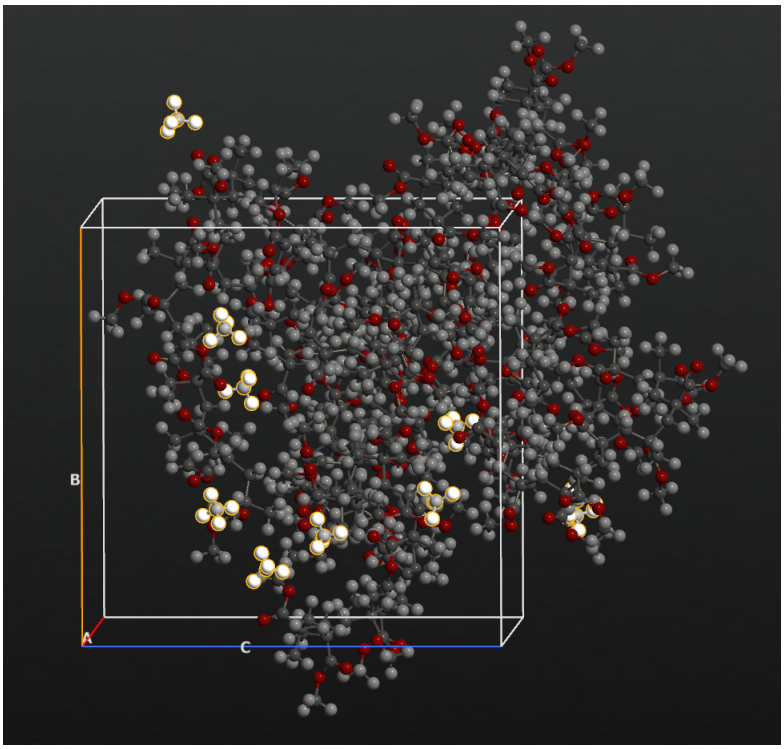
•Save the output file as `pmma_methane.hdf5`.

•Open the **Editor** and confirm the script.

Embedded molecules are defined as called *Included Molecule 1* with configuration and bonds information in the polymer Monte Carlo builder. In this condition, 5 *PMMA* chains will be generated by embedded 10 *Methane* molecules.

•Run the job in the **Job Manager**.

It takes around 15 minutes on the laptop. You can see 5 chains of *PMMA* and 10 methanes in a periodic box as shown in the below figure.



Nanoparticles in a melted polymer

There are two ways to embed nanoparticles in a melted polymer system.

1. Embed nanoparticles as a host configuration. The host configuration will be fixed during the polymer Monte Carlo and force capped molecular dynamics. Therefore polymer will be positioned in space defined by the host configuration.
2. Embed nanoparticles as molecules. Smaller sized nanoparticle can be assumed as molecules. You can load a nanoparticle from `Add molecule from file`. Such as embedded molecules *Polymer with embedded molecules*, you can also add how many particles will be embedded. In this case, embedded nanoparticle configuration should be a type of `MoleculeConfiguration`. And it is randomly positioned with polymers by Polymer Monte Carlo.

In this tutorial, you will embed about `15` Å sized *Silica* nanoparticle with *poly-stylene* polymer.

You will firstly embed a nanoparticle using the host configuration.

Note: The host configuration is fixed, while polymers are generated into the host configuration by polymer Monte Carlo builder. You can import and embed any kinds of host configuration with polymers.

Polymer Monte Carlo is parameterized by `Dreiding` or `OPLS` force fields. Considering atomic types should be able to use in `Dreiding` or `OPLS` force field parameter.

•Open the **PolymerBuilder**.

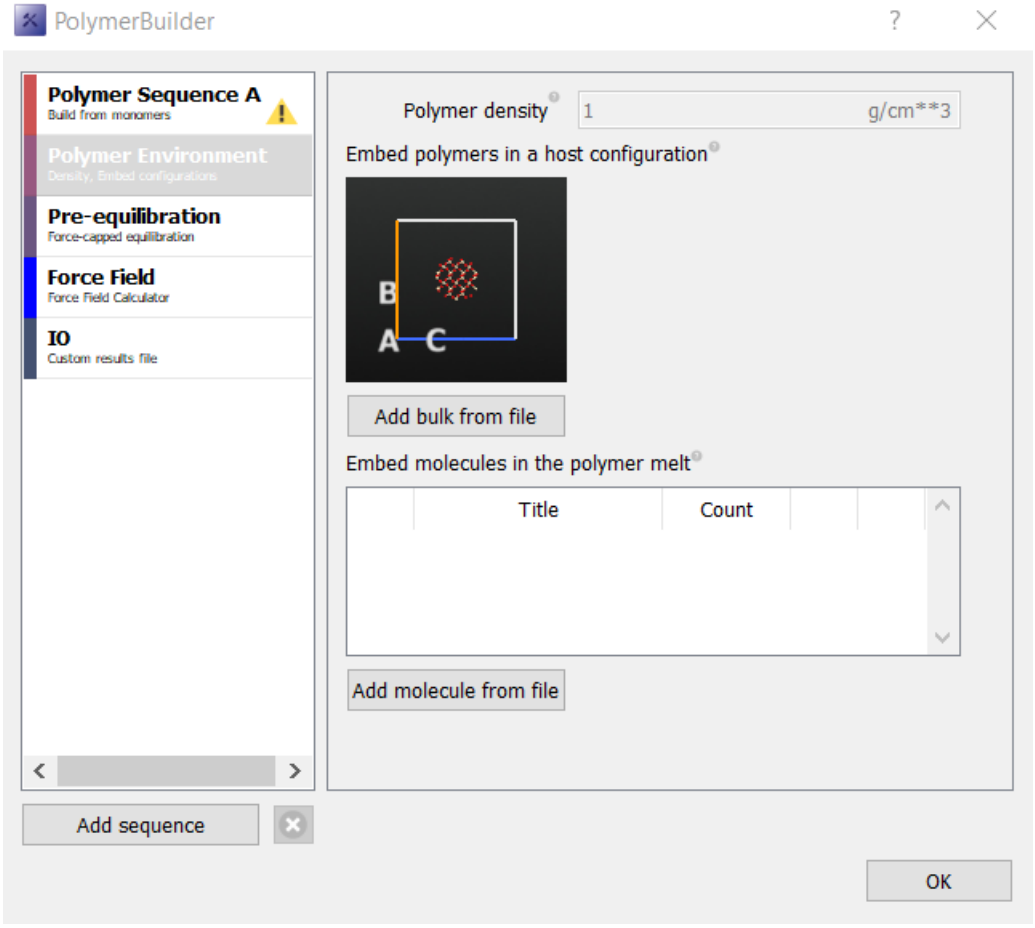
•In the **Polymer Sequance A**, select the `Stylene` in the Monomer Database. And change the number of chains as `10`.

•In the **Polymer Environment**, click `Add bulk from file` button.

You see saved configurations in your project. Import the host configuration here. In this tutorial, download the `15` Å sized SiO_2 nanoparticle configuration in a `35` Å x `35` Å x `35` Å periodic box, `sio2_nanoparticle.hdf5` and use it.

Note: Once you use the host configuration selecting the `Add bulk from file`, `polymer density` is inactive because polymer is packed in the defined periodic box of host configuration.

•Select the `BulkConfiguration_0` of `sio2_nanoparticle.hdf5`. You can see the host configuration on the widget as the below figure.



•In the **Force Field**, change `OPLS` into `Dreiding` force field. Because the new embedded nanoparticle will run using the Dreiding force field without assigning the OPLS force field manually and in the case of SiO_2 nanoparticle, OPLS will be in a trouble not to make a neutral charge in a total configuration.

•Save the output file as `host_nanoparticle_10ps.hdf5`.

•Open the **Editor** and check the script.

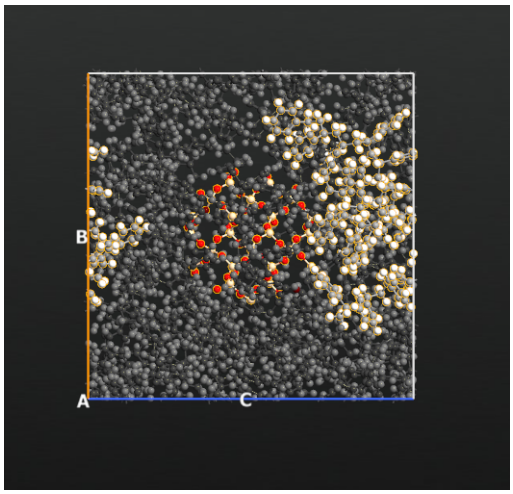
You can see *Host configuration information*.

•Run the job after saving the input file as `host_nanoparticle_10ps.py`.

It takes around 30 minutes on the laptop.

Note: Once the host configuation is large, it takes time to build it even in the MPI process. Polymer builder by polymer Monte Carlo takes a few minutes in your laptop, but force capped molecular dynamics takes time depending on the number of atoms. Such a large system probably needs more steps than default in pre-equilibration to have a reasonable configuration.

The configuration of SiO_2 nanoparticle with 10 *PS* is shown in the below figure.



You will now build nanoparticles in polymer as the embed molecules.

•Open the **PolymerBuilder**.

•In the **Polymer Sequance A**, select the `Stylene` in the Monomer Database. And change the number of chains as `10`.

•In the **Polymer Environment**, click `Add molecule from file` button.

You see saved configurations in your project. But unlike the host configuration, you need a molecular type of configuration.

•Open the downloaded `sio2_nanoparticle.hdf5` in the **Builder**.

It is the `15` Å sized SiO_2 nanoparticle configuration in a `35` Å x `35` Å x `35` Å periodic box. We will remove the periodic box and save as a MolecularConfiguration.

•Change the current configuration into a molecule configuration clicking the icon.

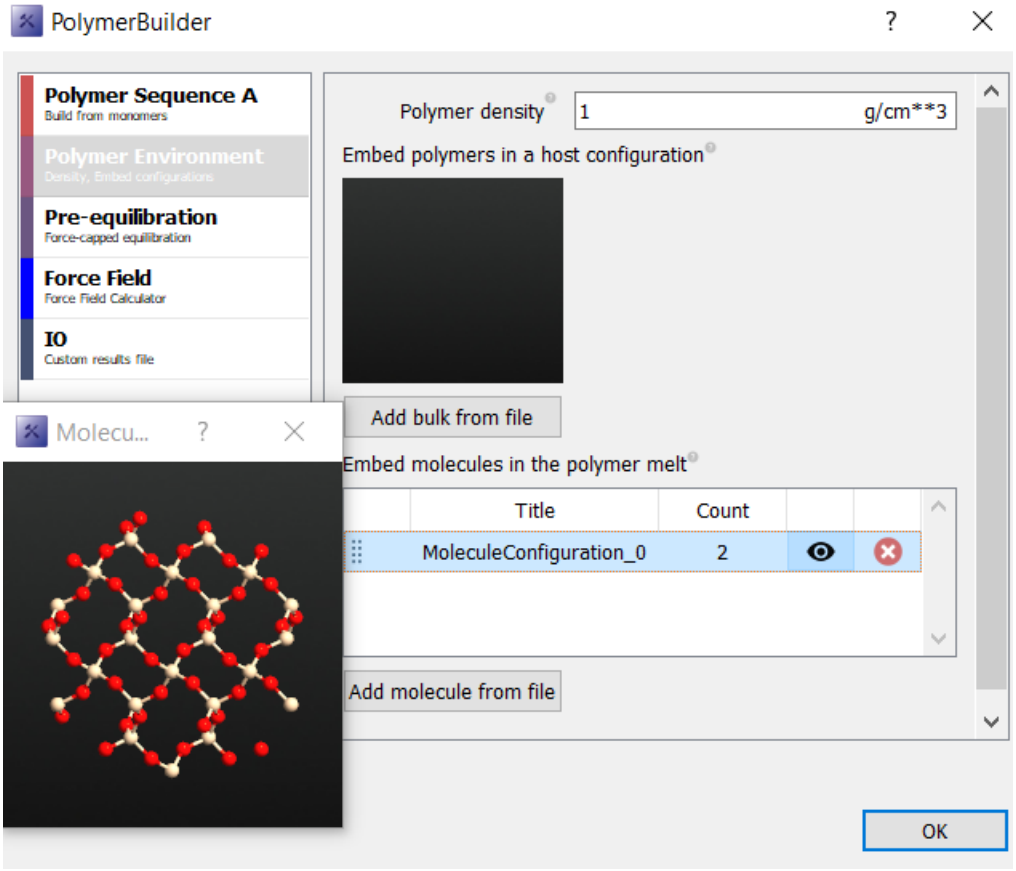
•Save the configuration as `sio2_nanoparticle_molecule.hdf5`.

•Now when you click `Add molecule from file` button again, you see it including `MoleculeConfiguration`.

•Load the `MoleculeConfiguration` in `sio2_nanoparticle_molecule.hdf5`.

•In this case, you will embed 2 nanoparticles. Change the `Count` into `2`.

As the below figure, you can visually check the embed nanaparticle configuration and the number of particles.



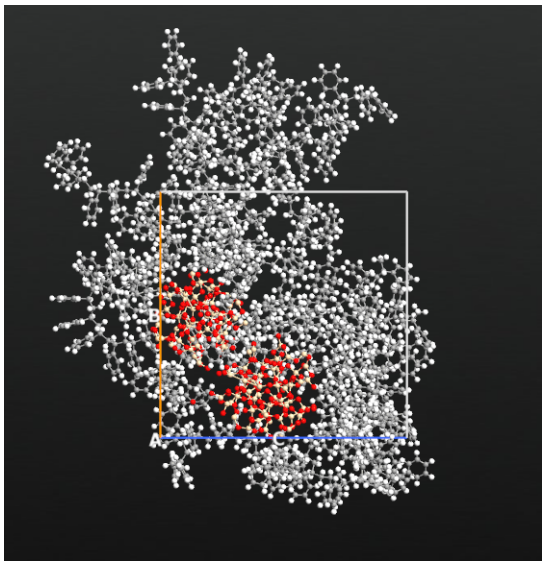
•In the **Force Field**, change `OPLS` into `Dreiding` force field. Because you will not assign the OPLS force field on the nanoparticle manually.

•Save the output file as `2nanoparticle_10ps.hdf5`.

•Open the **Editor** and check the script.

•Run the job after saving the input file as `2nanoparticle_10ps.py`.

It takes around 30 minutes on the laptop. As shown in the following figure, 2 nanoparticles are randomly positioned with polymers. In this case, it is fit to the defined polymer density controlling the periodic box. And nanoparticles and polymers are relaxed by pre-equilibration process.



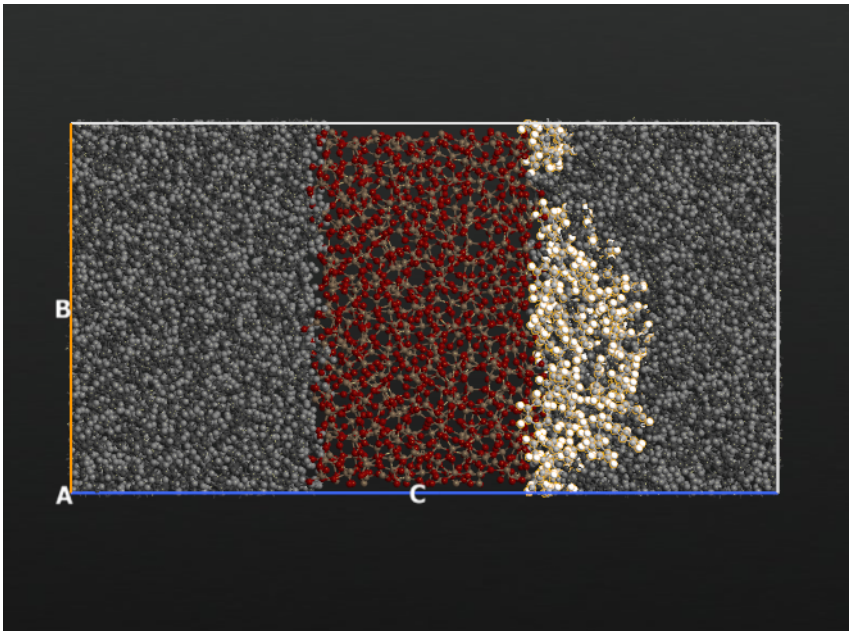
Polymers around a surface

Now you will see an advanced example using the *Poly-stylene* melt with amorphous OH-terminated SiO_2 surface. It is a same procedure to use a host configuration to import the surface configuration as shown in the nanoparticle example. Download the OH-terminated amorphous SiO_2 surface configuration, `sio2_surface.hdf5` and use as a host configuration.

You will build 20 *poly-stylene* chains of 40 monomers per chain with embedded OH-terminated SiO_2 amorphous surface configuration with `25` Å thickness of SiO_2 in about `40` Å x `40` Å x `80` Å sized periodic box as a host configuration.

During the polymer Monte Carlo, the host configuration is fixed to generate 20 *poly-stylene* in a periodic box. After that, Force-capped molecular dynamics will be performed by force field parameters.

It has more than 16,000 atoms. Generation of random polymers by polymer Monte Carlo takes a few minutes on the laptop. But the force-capped molecular dynamics takes a quite long time even in the default condition. Probably you need more steps to get a reasonable configuration in the pre-equilbrium. The below figure shows the 20 *Poly-stylene* melt with amorphous OH-terminated SiO_2 surface with highligted one chain of *Poly-stylene*.



Customized monomers

The Monomer Database of QuantumATK includes several types of monomers. But you may need to design and save customized monomers. You can add customized monomer into the Monomer Database in `/.vnl/databases/monomers` of user home directory.

Original shipped QuantumATK Monomer Database is located in the `/share/nanolab/monomers` of installer directory. You will see some grouping directories such as `Condensation_Monomers`, `Endgroups`, or `Vinyl_Monomers`. When you open one of monomer database (ex. `Tert-Butyl Methacrylate.py`), you see the configuration information with tags. *Add tags* include the information of atom type in a monomer such as `END`, `HEAD`, `TAIL`, `CONNECT`, `GROUP_A` and so on. You can design your own monomer. And add tags and bonds with a correct format and save the python file in `/.vnl/databases/monomers` of user home directory.

We guide the simple way to modify the monomer. This tutorial demonstrates to add `OH` functional group in the `Tert-Butyl Methacrylate` monomer.

In the `/share/nanolab/monomers/Vinyl_Monomers`,

•Drag the `Tert-Butyl Methacrylate.py` into **Builder**.

You see the `Tert-Butyl Methacrylate` monomer on the window.

•Open the *Tags* in the **Selection Tools** plug-in.

You can see the existed tags of `Tert-Butyl Methacrylate` monomer. When you choose one of tags, atom selection will be together. Acoording to this tags, for instance, `HEAD_GROUP_A` and `TAIL_GROUP_A` will finally overlap to formate the polymer. You will modify one of hydrogens not to be effect on the formation of main polymer chain.

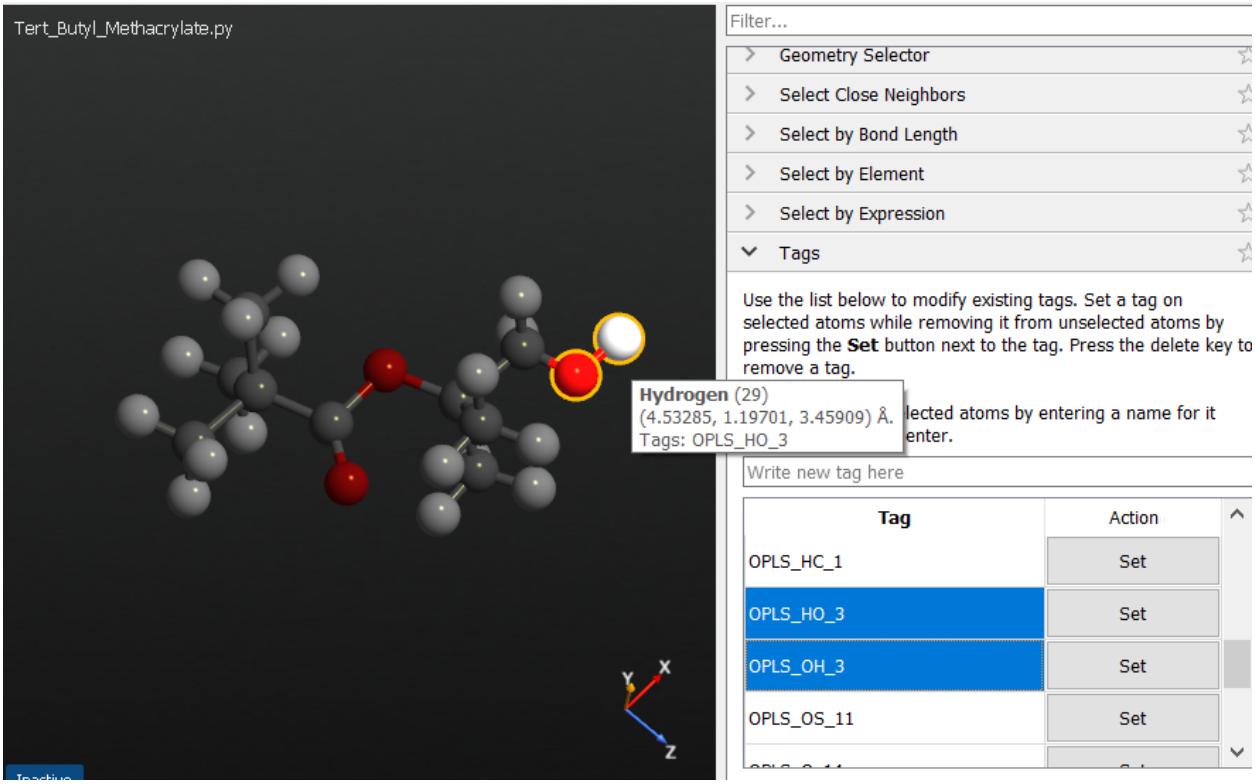
•Change `Hydrogen` of atom number `23` into `Oxygen` atom.

•Click the icon of passivate the configuration with hydrogen.

You will see the `OH` group instead of hydrogen. You need to define the atomic parameter using the `Dreiding` or `OPLS` force field.

•Write new tag in Tags plug-on tool such as O (atom number 23) into OPLS_OH_3, new H (atom number 29) into OPLS_HO_3. And remove or modify the previous tags for OH functional group. For instance, remove the changed hydrogen (atomic number 23) in OPLS_HC_1 tags and modify the carbon (atom number 11) attached OH group into OPLS_CT_11 tag.

It is shown in the below figure.



Add tags should be very careful because you need to consider the polymerization from the monomer. Otherwise force field type is not correct. You can see in detail the Dreiding and OPLS potential parameters of QuantumATK in the *DreidingPotentialBuilder* and *OPLSPotentialBuilder*.

Note: ATKPython has functions of *DreidingPotentialBuilder* and *OPLSPotentialBuilder* that tell the types that you have, and gives a short description of them. Try the function `printTotalTypes()` on the *DreidingPotentialBuilder* and *OPLSPotentialBuilder*. It gives a table for all of the available types. Monomers should have the right tags, then they work with it.

•Open the script using the **Editor** in the Send to.

You will see new tags with OPLS_OH_3 for atom number 23, OPLS_HO_3 for atom number 29, OPLS_CT_11 for atom number 11. And you also add bond information. Atom number 29 needs to make a new bond with atom number 23. You should add it with the format of *Add bonds*.

```
# Add tags
monomer_1.addTags('ENDH_OPLS_CT_4', [0])
monomer_1.addTags('HEAD_CONNECT',   [0])
monomer_1.addTags('HEAD_GROUP_A',   [1])
monomer_1.addTags('H_C',            [20, 21, 22, 24, 25, 26, 27, 28])
monomer_1.addTags('OPLS_CT_11',     [11])
```

(continues on next page)

(continued from previous page)

```
monomer_1.addTags('OPLS_CT_4',      [9, 10, 12, 16])
monomer_1.addTags('OPLS_CT_5',      [0])
monomer_1.addTags('OPLS_CT_8',      [4])
monomer_1.addTags('OPLS_CT_95',     [8])
monomer_1.addTags('OPLS_C_19',      [5])
monomer_1.addTags('OPLS_HC_1',      [ 1,  2,  3, 13, 14, 15, 17, 18, 19, 20, 21, 22, 24,
                                         25, 26, 27, 28])
monomer_1.addTags('OPLS_HO_3',      [29])
monomer_1.addTags('OPLS_OH_3',      [23])
monomer_1.addTags('OPLS_OS_11',     [6])
monomer_1.addTags('OPLS_O_14',      [7])
monomer_1.addTags('TAIL_CONNECT',   [4])
monomer_1.addTags('TAIL_GROUP_A',   [16])
monomer_1.addTags('TAIL_GROUP_B',   [12])

# Add bonds
bonds = [[ 0,  1],
         [ 0,  2],
         [ 0,  3],
         [ 0,  4],
         [ 4,  5],
         [ 4, 12],
         [ 4, 16],
         [ 5,  6],
         [ 5,  7],
         [ 6,  8],
         [ 8,  9],
         [ 8, 10],
         [ 8, 11],
         [ 9, 26],
         [ 9, 27],
         [ 9, 28],
         [10, 20],
         [10, 21],
         [10, 22],
         [11, 23],
         [11, 24],
         [11, 25],
         [12, 13],
         [12, 14],
         [12, 15],
         [16, 17],
         [16, 18],
         [16, 19],
         [23, 29]]
monomer_1.setBonds(bonds)
```

•Save the new monomer with new tags as `Custom_Tert_Butyl_Methacrylate.py` creating the new directory as `Custom` in the `/.vnl/databases/monomers` of user home directory.

Note: If you create a new directory in `/.vnl/databases/monomers`, it will make a grouping with the directory

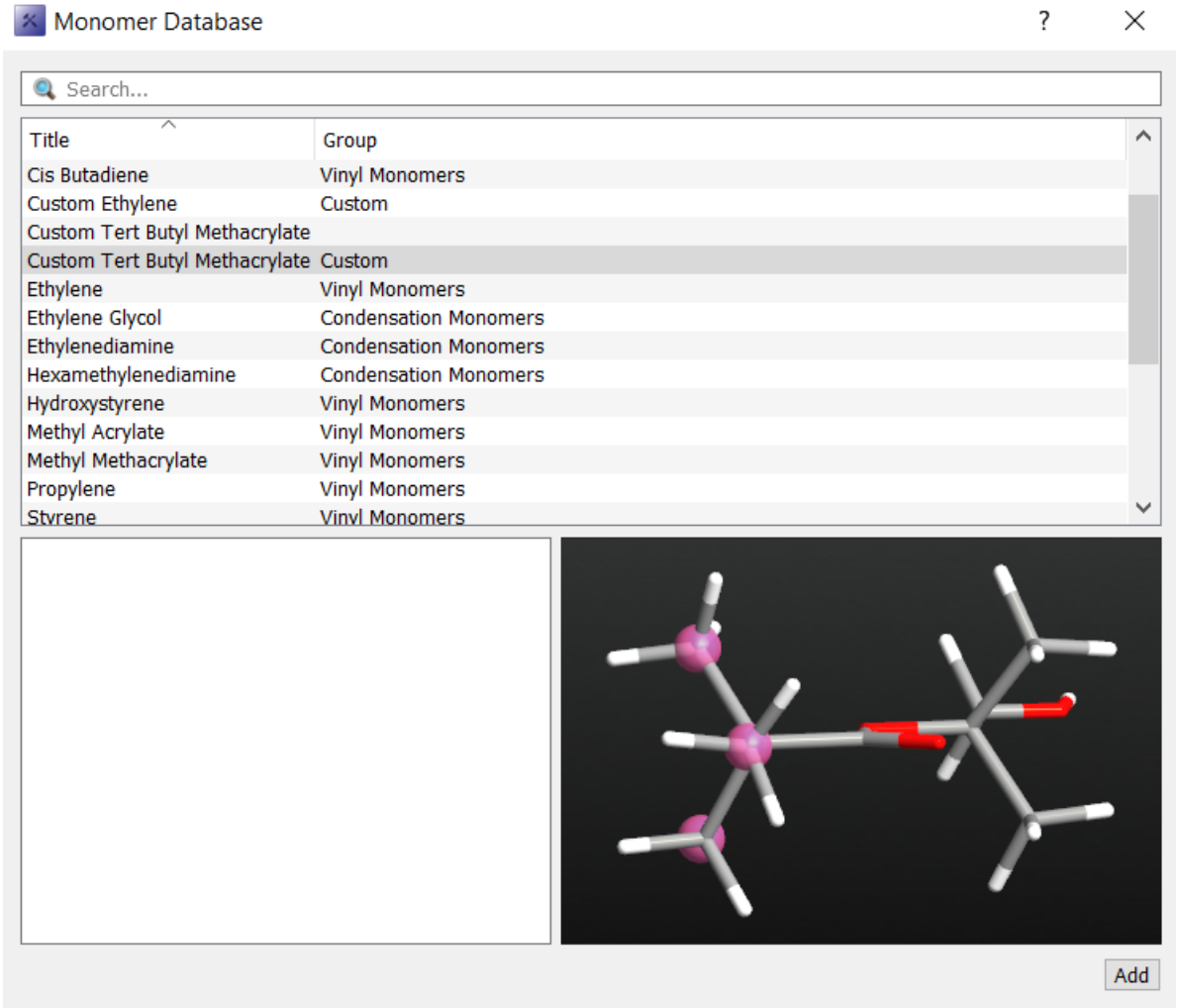
name. After creating a new monomer by user, you need to restart the **Script Generator**.

Now you can load the customized monomer from the Monomer Database in the PolymerBuilder GUI.

•Open the **PolymerBuilder**.

•Click the `Select monomer`.

Now you see the `Custom Tert Butyl Methacrylate` in the momoner database as the below image. When you click it, you also see the configuration indicating how to connect the main chain and `OH` functional group on the widget.



3.11.2 Greens function surface calculations

Version: 2016.0

This tutorial introduces you to a very novel method for computing surface properties: Greens function surface calculations. The central difference to traditional slab calculations is that the surface electronic structure is coupled to the bulk electronic structure through the non-equilibrium Greens function (NEGF) method.

The system you will investigate is a silver (100) surface, for which you will compute the work function. In particular, you will:

•construct the surface configuration;

•set-up and run the Greens function surface calculation;

•analyze the work function and compare to results from traditional slab calculations.

Atomistic models of a surface

This section aims to introduce the Greens function surface configuration, and compare it to the conventional slab model. The two figures shown below illustrate the differences between the two.

The slab consists of a finite number of silver layers, surrounded be vacuum in both directions along the C-axis, so it really has two surface–vacuum interfaces. The Hartree difference potential, ΔV_{H}, is bulk-like in the middle of the slab, but changes near the slab–vacuum interfaces, and becomes constant in the vacuum regions.

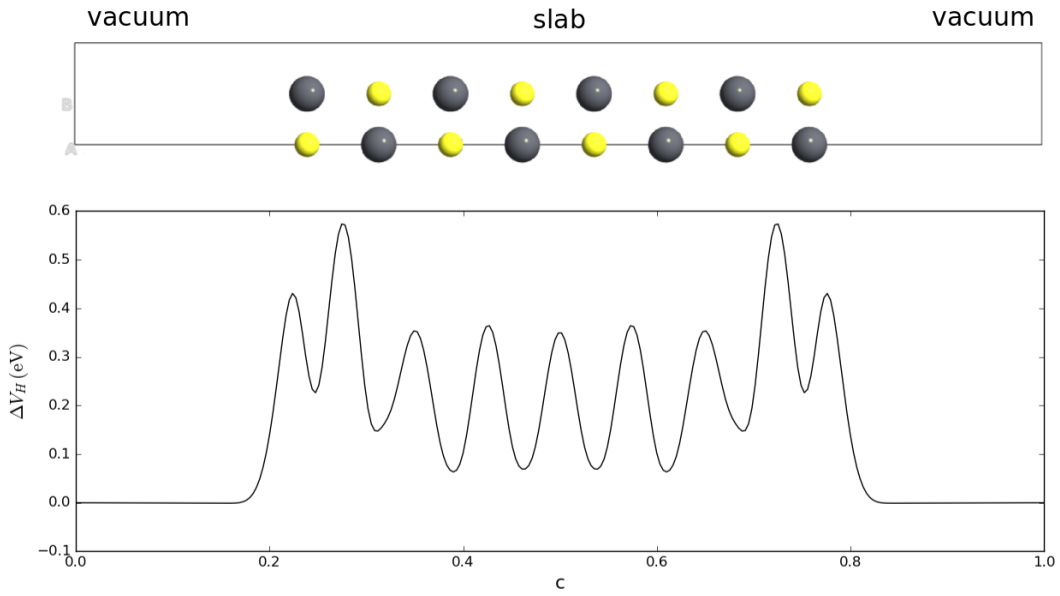


Fig. 3.107: The traditional slab model of a surface is surrounded by vacuum on both sides along the C-direction.

In contrast, the Greens function surface model consists of a single surface region attached to an electrode (a fully periodic bulk). The electronic structure is matched at the electrode–surface boundary using Greens functions. The surface is therefore electronically connected to a bulk substrate, and the surface chemical potential is fixed to that of the electrode.

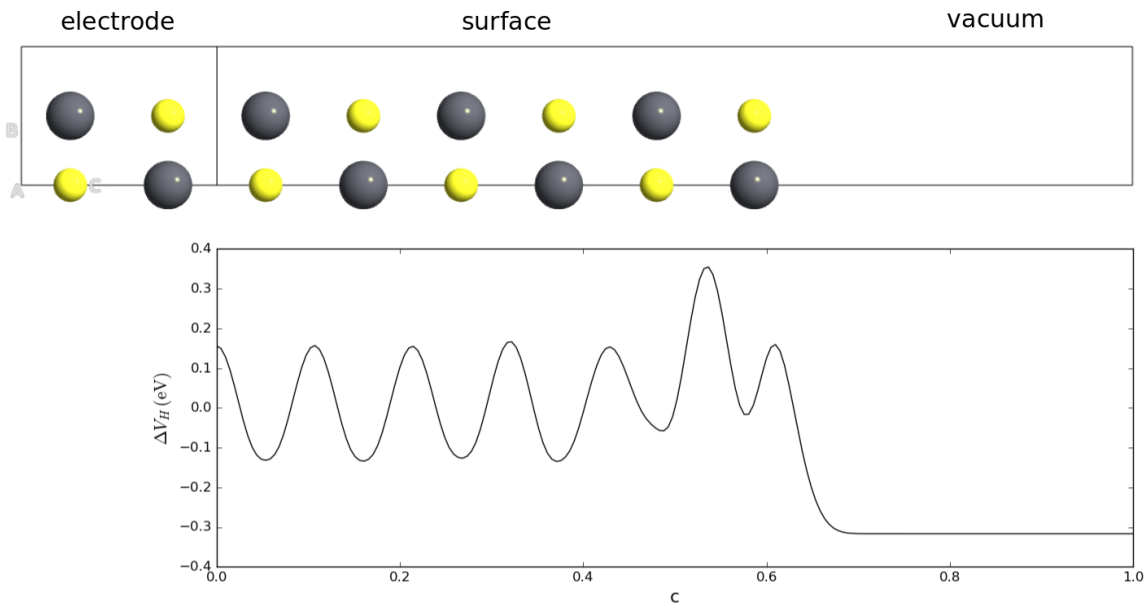


Fig. 3.108: The Greens function surface model has only a single surface–vacuum interface, and is geometrically and electronically connected to a periodic (bulk) electrode.

Advantages of NEGF

Greens function surface calculations have some significant advantages over traditional slab approaches:

- The slab model is a finite system and it therefore has a finite number of electrons. Consequently, when transferring charge from a molecule to the surface (or the other way around), the total number of electrons will change, and therefore the chemical potential of the electrons in the slab also changes. This is a spurious effect, and the Greens function surface model alleviates it completely by coupling the surface to an infinite reservoir of electrons at a fixed chemical potential (the electrode).
- Convergence of computed properties wrt. slab thickness is always an issue, and various corrections are ofte needed to avoid interactions with repeated images of the simulation cell. With the proper choice of boundary conditions, QuantumATK Greens function surface calculations converge very easily wrt. surface thickness, as we shall see later in this tutorial.

NEGF calculation with a single electrode

As already mentioned, the Greens function surface configuration consists of an electrode and a surface region (also called the central region). In the language of QuantumATK, it is therefore a device configuration with a single electrode. As in any QuantumATK device calculation (using NEGF), the calculation proceeds in two steps:

1. The electrode bulk electronic structure is calculated first. This is usually very fast.
2. The NEGF calculation then proceeds to calculating the ground state of the surface region only, given the constraint that the electronic structure at the electrode–surface interface must match.

As illustrated below, three different sorts of boundary conditions are used along the C-direction of the simulation cell. The electrode ensures full periodicity towards the left, while a Dirichlet boundary condition is used at the electrode–surface interface. The Neumann boundary condition is then the natural choice on the right-hand boundary.

Work function of Ag(100)

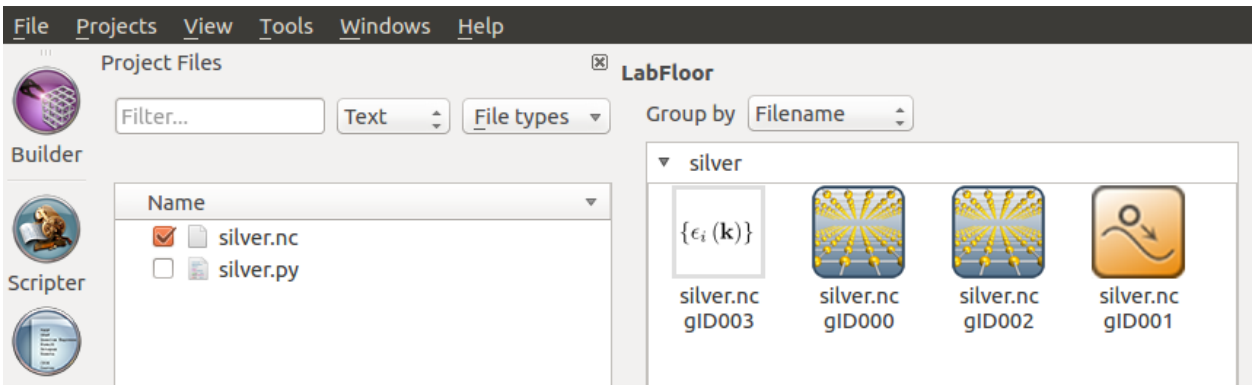
You will now calculate the work function of Ag(100) using DFT and Greens functions. The basic workflow is this:

1. Use the **Builder** to construct the surface configuration from a stress-optimized silver bulk, and specify ghost atoms.
2. Use the **Scripter** to create a script that calculates the ground state, and saves the Hartree difference potential (HDP) and chemical potential.
3. Use the **1D Projector** tool to visualize the HDP.
4. Calculate the work function from the HDP and chemical potential.

Build the surface configuration

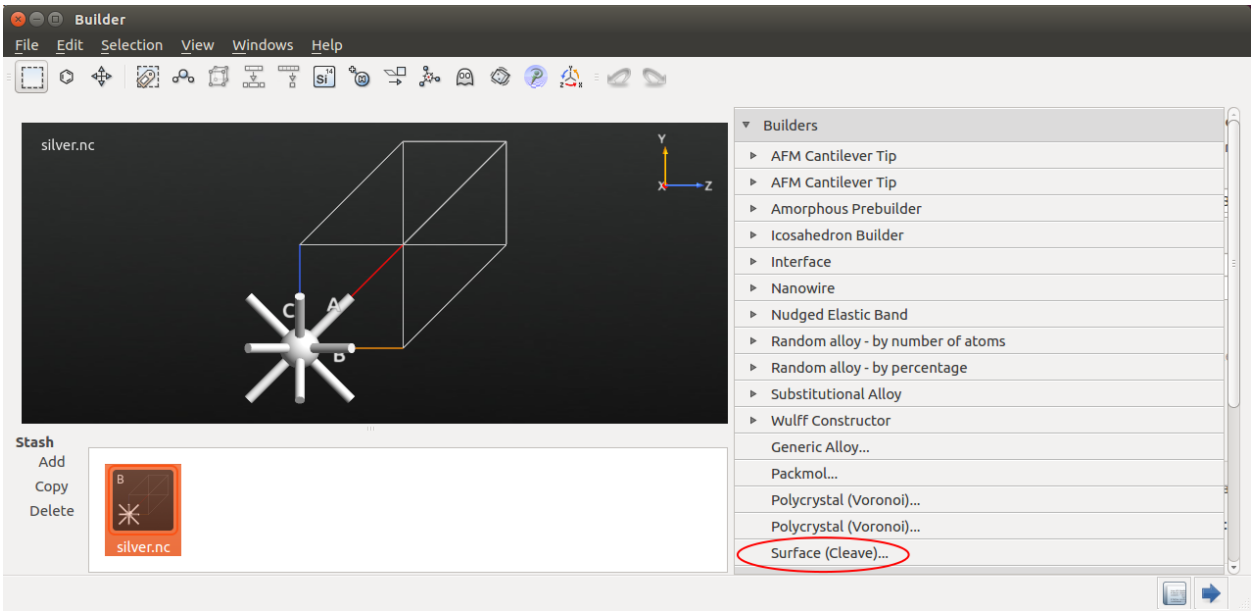
You will first need to cleave a silver bulk configuration along the (100) direction. To this end, you will use a silver bulk optimized at the LDA level using FHI pseudo-potentials and a SingleZetaPolarized basis set.

Create a new **QuantumATK Project**, and download a HDF5 file containing the optimized structure: `silver.hdf5`. Save it in the **Project Folder** such that its contents become visible on the **LabFloor**. You will notice that the silver silver band structure was also calculated – this will prove useful later on. You can also obtain the script used for the calculations: `silver.py`.

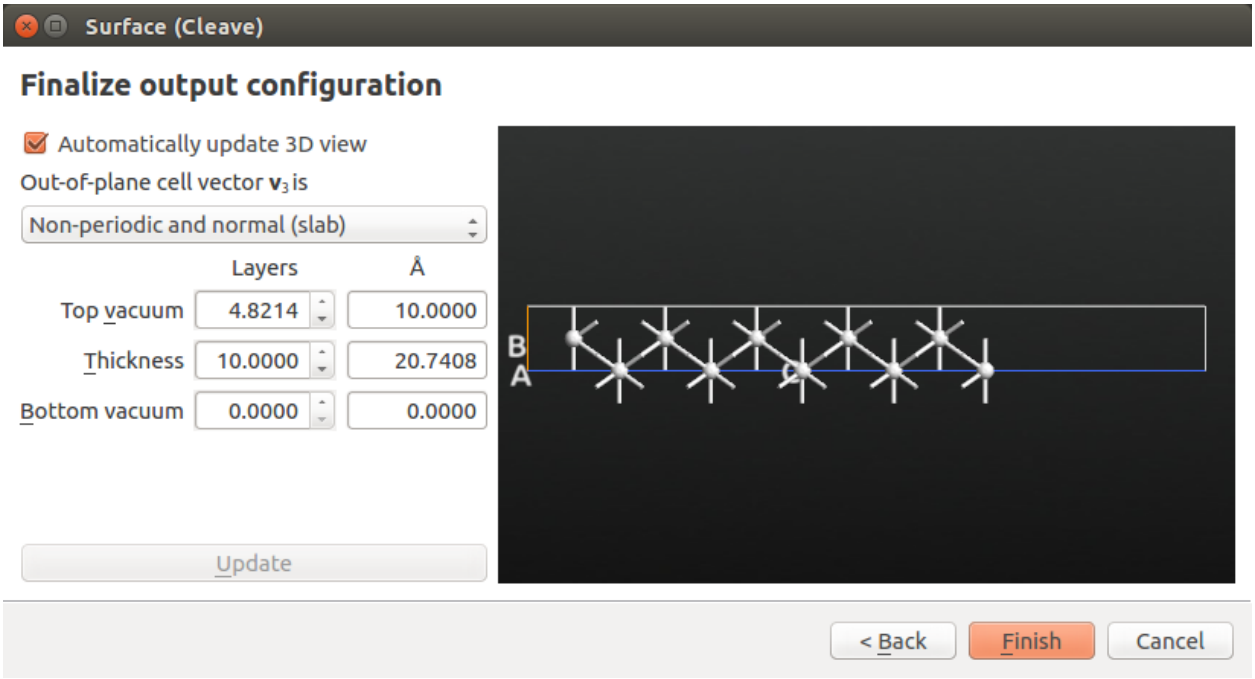


Tip: You may refer to the tutorials *Geometry optimization: CO/Pd(100)* and *Calculate the band structure of a crystal* for more information on how to perform a bulk geometry optimization and a band structure calculation.

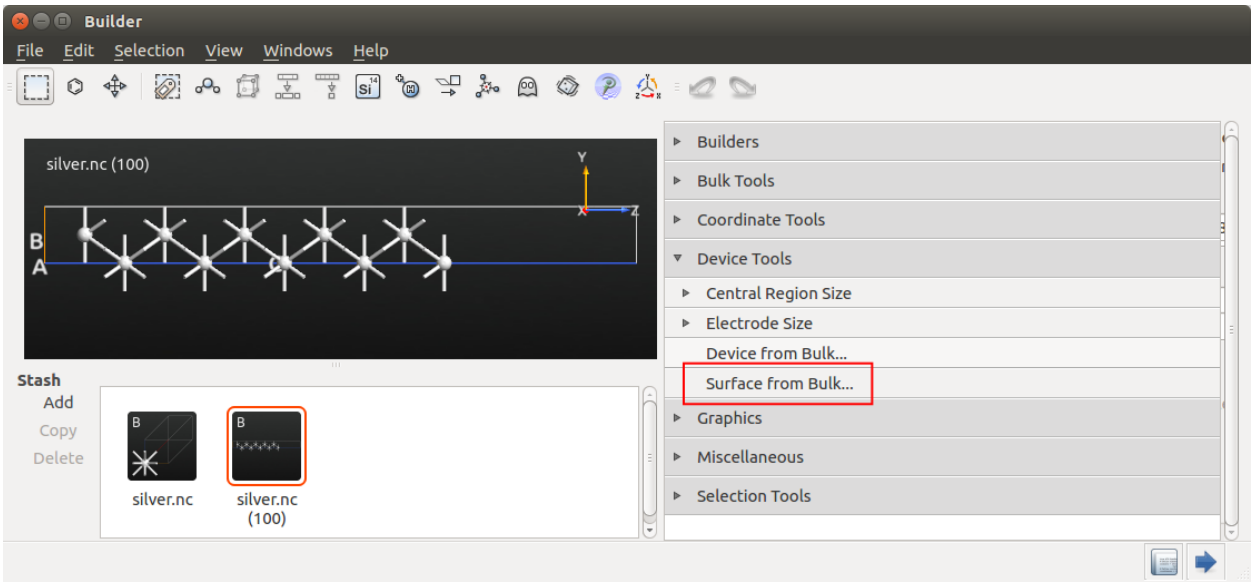
From the LabFloor, transfer the relaxed silver bulk configuration (object id `gID002`) to the Builder and confirm that the optimized LDA-SZP lattice constant is 4.148 Å. Then select the configuration and open the *Builders → Surface (Cleave)* tool.



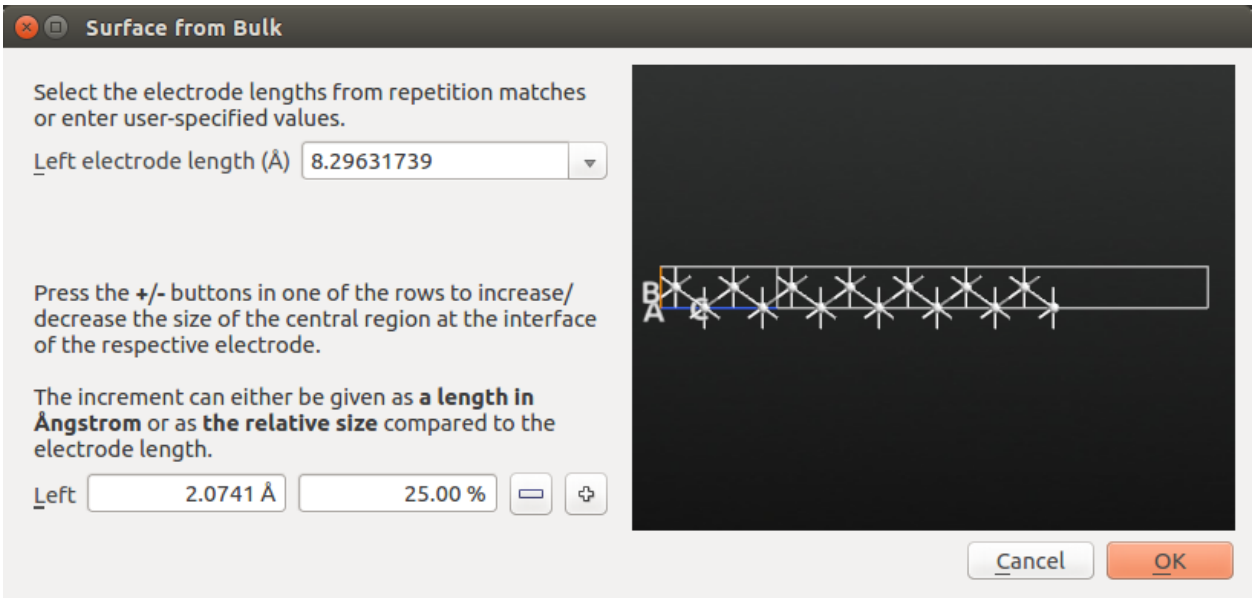
This tool allows you to cleave the silver crystal along the (100) direction: Click *Next* twice, and choose the Non-periodic and normal (slab) output configuration with 10 metal layers, as illustrated below.



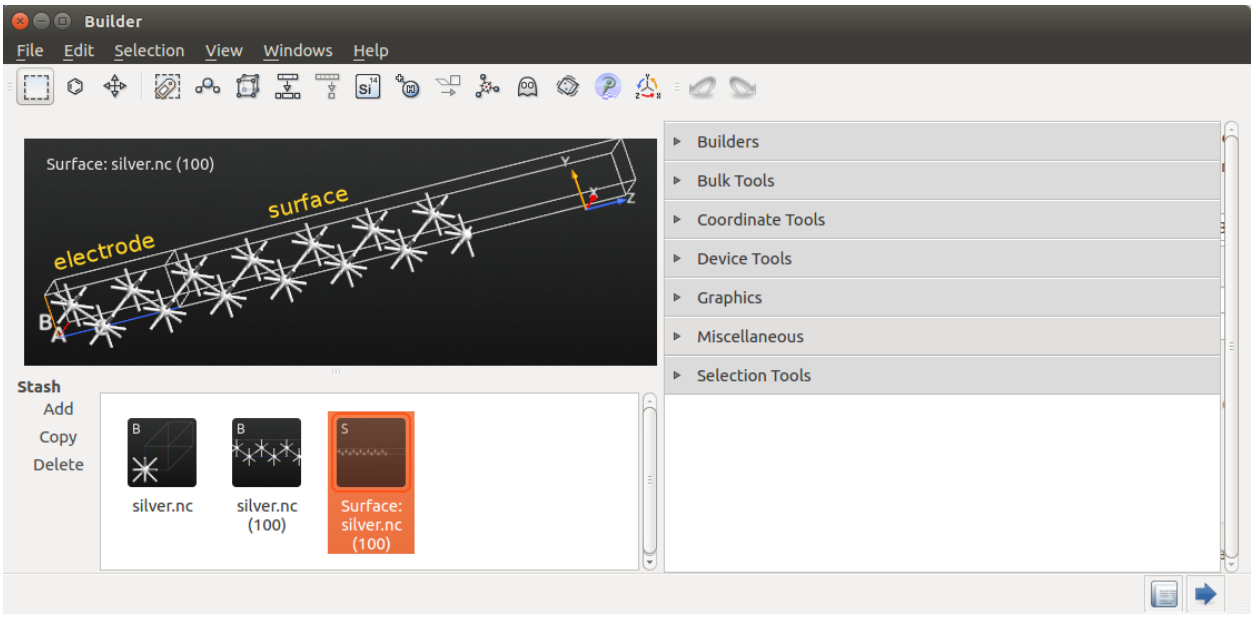
The item *silver.hdf5 (100)* is now in the **Stash**. Select it and use the *Device Tools* → *Surface From Bulk* to create the surface configuration needed for the Greens function surface calculations.



The **Surface from Bulk** widget allows you to select the size of both the electrode and the surface (here denoted the central region). The default electrode length of 8.3 Å will do just fine. Note that you can also change the size if the central region attached to the electrode.



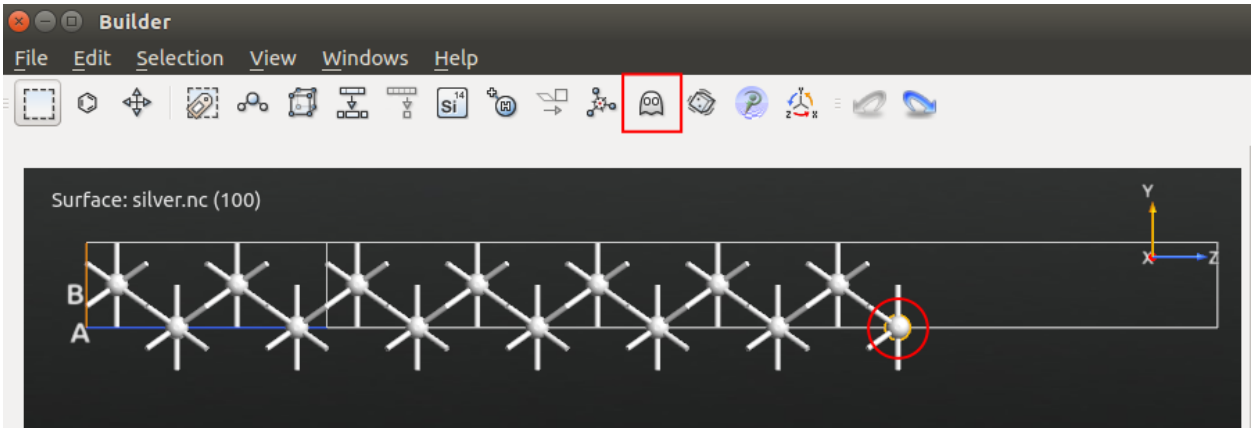
Click **OK** to add the configuration to the Stash. The name of the new item is automatically prefixed *Surface*.



Ghost atom

As discussed in the tutorial *Computing the work function of a metal surface using ghost atoms*, accurate work function calculations with a localized basis set require one or more ghost atoms immediately above the surface. This is most easily accomplished by turning the top-most atom into a ghost atom.

Select the top-most atom as shown in the image below, and click the icon in the top panel bar.



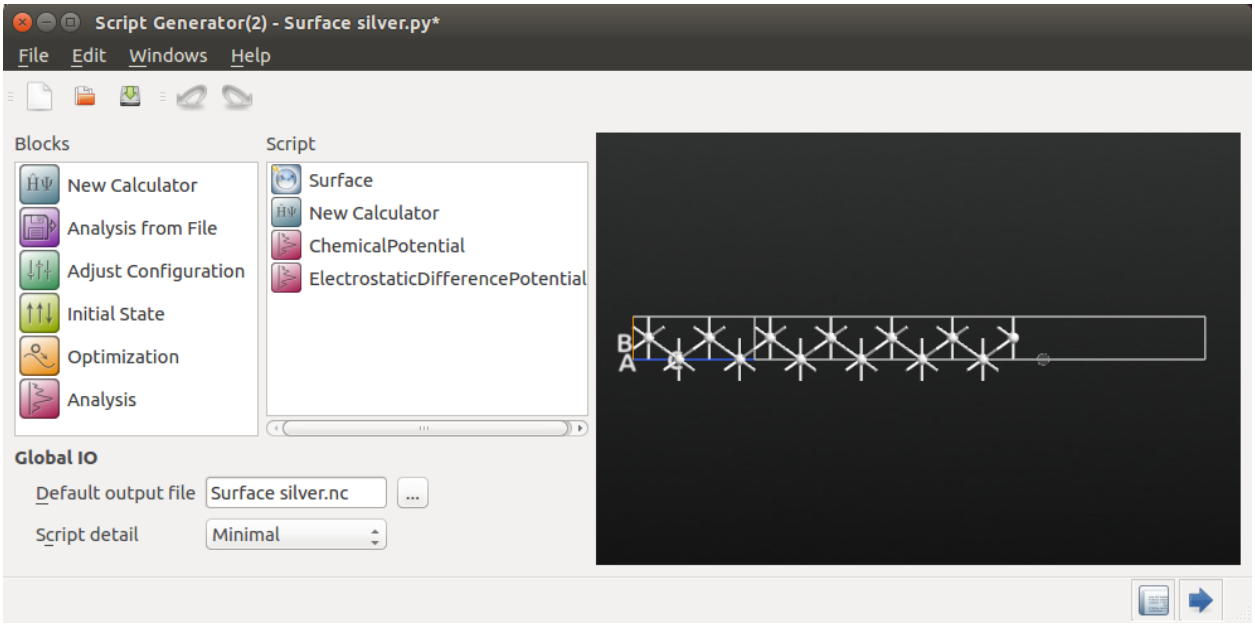
The atom is now designated a ghost atom, i.e. a basis set center with no nucleus.

Transfer the surface configuration to the Script Generator using the icon, and proceed to the next section to set up and run an QuantumATK script for the Greens function surface calculation.

Calculations

The **Script Generator** is a graphical interface for writing an QuantumATK Python script. Add the following blocks to the script by double-clicking them:

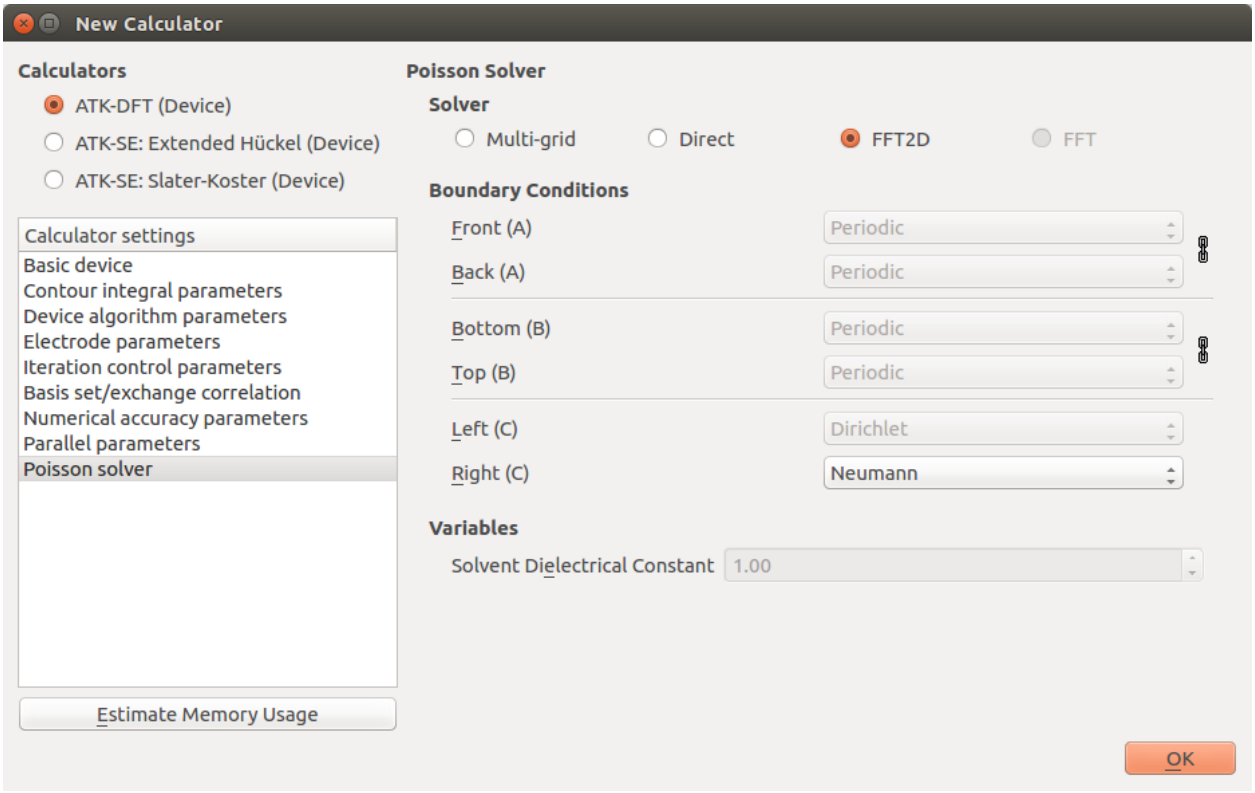
- New Calculator.
- *Analysis* → *ChemicalPotential*.
- *Analysis* → *HartreeDifferencePotential*.



Then double-click the added **New Calculator** block to open it. In the **Basic device** calculator settings, choose LDA exchange-correlation and a 9 x 9 x 100 k-point grid.

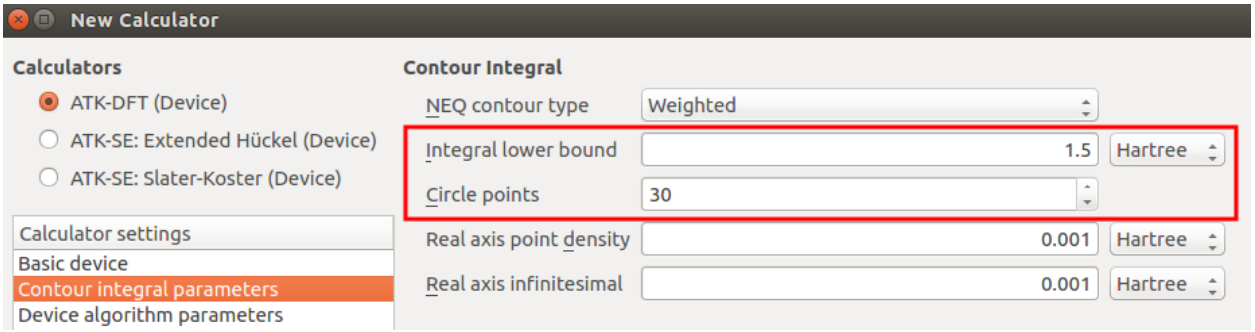
Note: As discussed in detail in the introductory tutorial *Transport calculations with QuantumATK*, it is vital to use a relatively large number of k-points along the C-direction in QuantumATK device calculations, e.g. 100. Importantly, these are only used in the electrode calculation, which is relatively cheap to calculate, and are not used in the main part of the QuantumATK device calculation: The central region is by definition not periodic along C, so it has only a single k-point along C.

Next, navigate to the **Basis set/exchange correlation** settings, and choose the `SingleZetaPolarized` basis set. Lastly, in the **Poisson solver** settings, change the boundary condition on the right-hand C-face to *Neumann*.

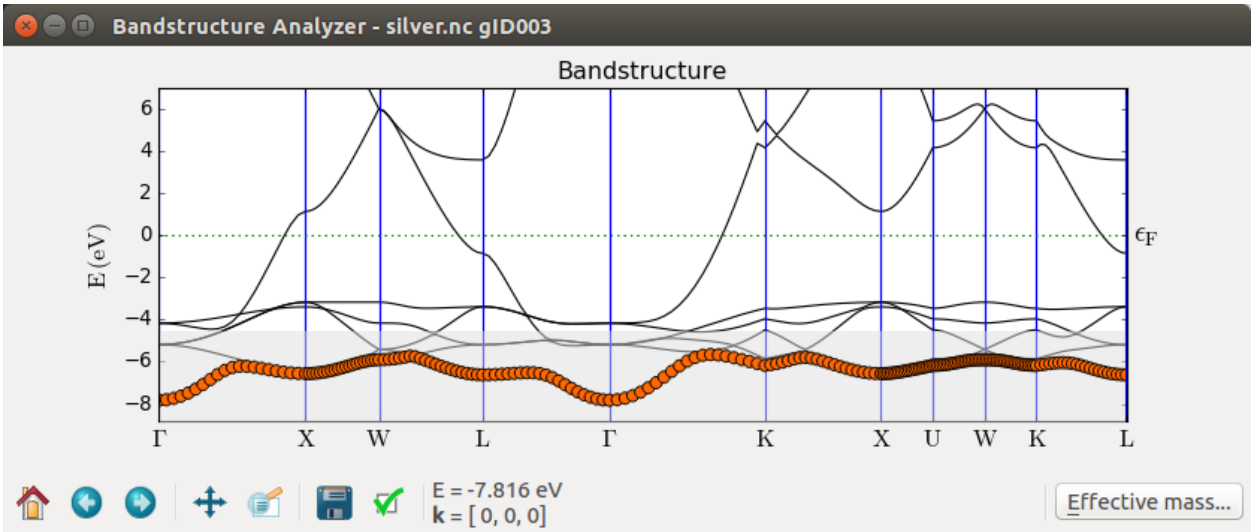


Contour integral

The NEGF method uses a contour integral to evaluate Greens functions. As explained in anothor tutorial (*Finite-bias calculations*), the default values for the settings *Integral lower bound* and *Circle points* may need to be increased if the applied pseudopotentials have deep-lying (semi-core) states.



It is therefore good practice to check the electrode band structure, in order to make sure that the lowest band does not fall outside the lower bound on the contour integral. In the present case, we can use the silver band structure stored in the file `silver.hdf5`, which is illustrated below.



The deepest energy eigenvalue is band is 7.8 eV (0.29 Hartree) below the Fermi level, which is well within the default contour integral energy window (1.5 Hartree below the Fermi level). No need to increase the lower bound, so you should just stick to the default value.

Save the script and run it

Back in the Script Generator main window, change the default output file name `silver_gfs.hdf5` and save the Python script as `silver_gfs.py`. If needed, you can also download the final script: `silver_gfs.py`.

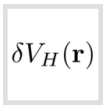
Then execute the script using the **Job Manager** or from a Terminal. The calculation should take about 4 minutes on a laptop.

Analysis

Once the calculation is finished, the HDF5 data file `silver_gfs.hdf5` arrives on the **LabFloor**. It contains the (one-probe) surface configuration with a converged calculator, and the ChemicalPotential and HartreeDifferencePotential (HDP) analysis objects:

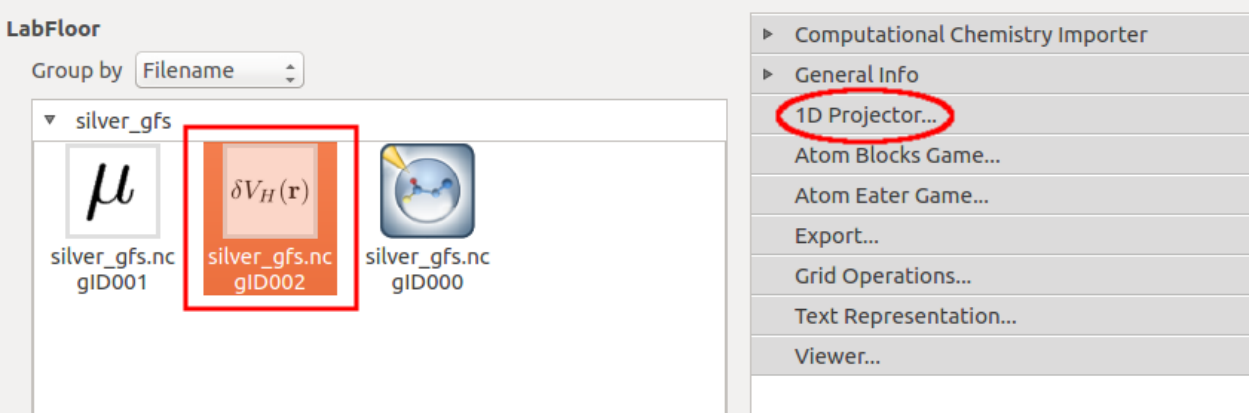
One-probe surface configuration

ChemicalPotential

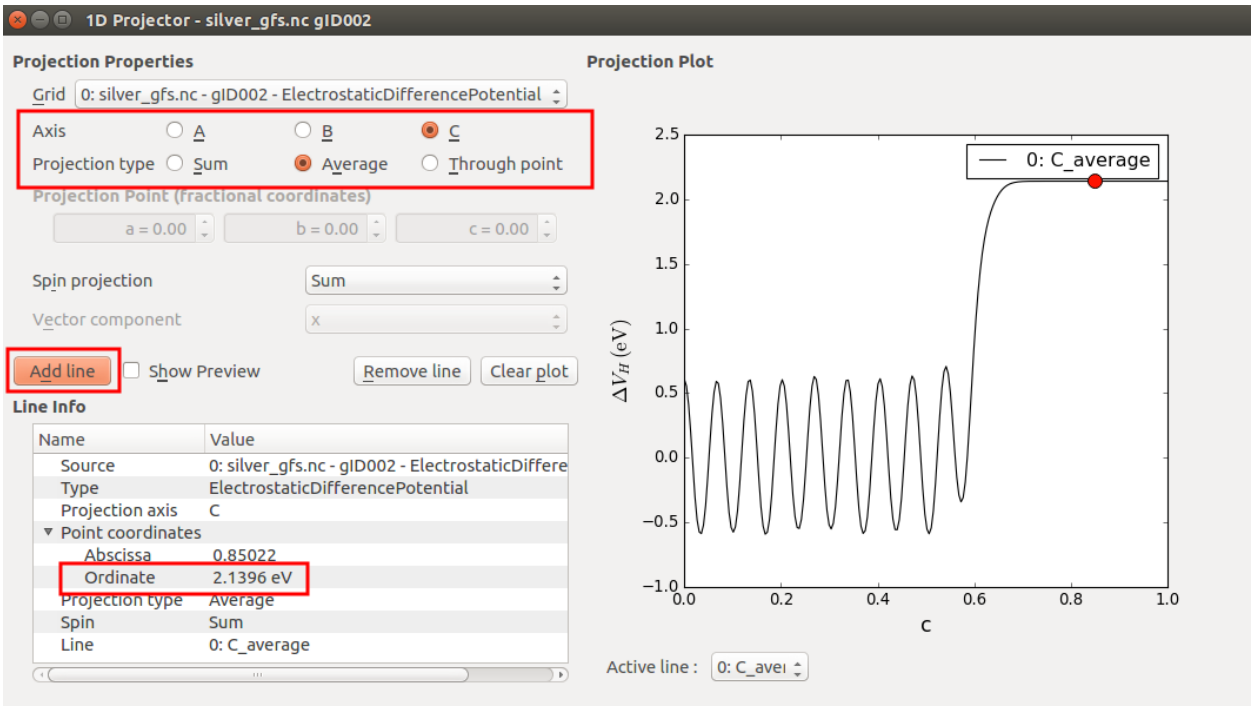


HartreeDifferencePotential

You should first take a look at the Hartree difference potential throughout the surface configuration. Use the mouse to select the HDP item, and then open the **1D Projector** tool from the right-hand plugins panel.



The HDP is represented on a 3D grid, but the 1D Projector allows you to project it onto any unit cell axis. Project the average HDP onto the C-axis in order to get the visualization shown below.

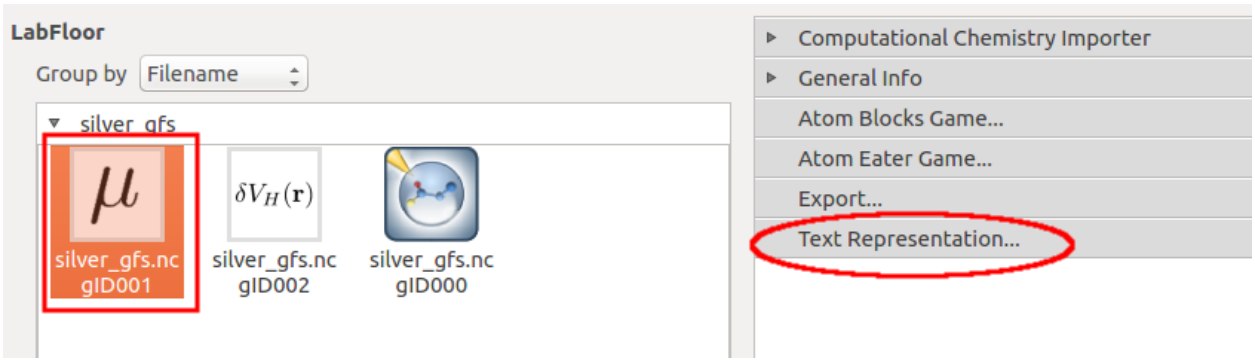


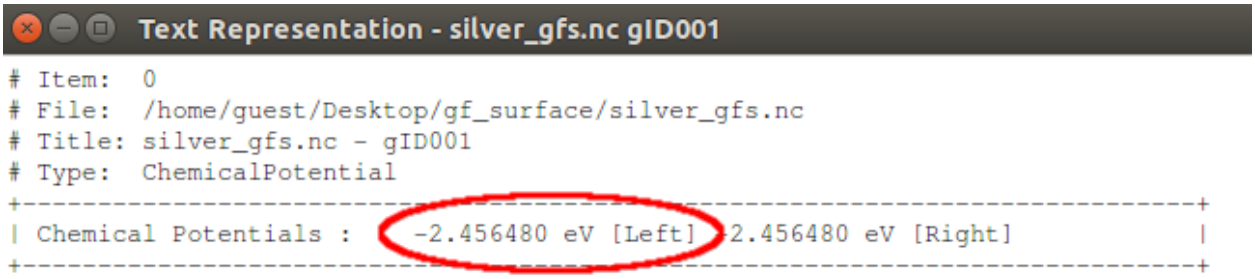
The Ag(100) work function is calculated from the difference between the (constant) Hartree difference potential in the vacuum region and the chemical potential (which in Greens function surface calculations is really the chemical potential of the electrode):

$$\mathrm{WF} = \delta V_{\mathrm{H}}^{\mathrm{c}=1} - \mu$$

By moving the red dot in the projection plot to the far-right end, where c=1, you can read off $\delta V_{\mathrm{H}}^{\mathrm{c}=1}$= +2.14 eV.

Next, use the **Text Representation** plugin to see the value of the chemical potential. You will find that μ= –2.46 eV.





The theoretically predicted work function is therefore 4.6 eV, which is in very good agreement with the experimental value (4.64 eV). Note, however, that the top surface layers were not relaxed. You can easily do this by inserting the
OptimizeGeometry block in the QuantumATK script.

Convergence wrt. metal layers

As a final illustration of the diference between Greens function surface calculations and traditional slab calculations, the figure below shows how the predicted Ag(100) work function converges wrt. the number of silver metal layers. The Greens function calculations are similar to the one done above, but one extra ghost atom is used.

It is quite clear that work functions computed using the Greens function surface model converge much faster than those calculated with the traditional slab model.

If you like, you can reproduce the calculations using the scripts `wf_slab.py` and `wf_surface.py`, and plot the results using `plot.py`.

3.11.3 Polarization

Spontaneous polarization of ferroelectric BaTiO3

Introduction

Ferroelectric (FE) materials have a spontaneous electric polarization that can be reversed by the application of an external electric field. FE materials find applications in capacitors, ferroelectric random access memory (RAM), and more recently in ferroelectric tunnel junction (FTJ) displaying giant electroresistance effects[1][2].

[1] X.Lou, Y. Zheng, and B. Wang, J. Appl. Phys., **111**, 074102, (2012)

[2] A.Chanthbouala, A. Crassous, V. Garcia, K. Bouzehouane, S. Fusil , X. Moya , J. Allibe , B. Dlubak, J. Grollier , S. Xavier , C. Deranlot , A. Moshar , R. Proksch , N. D. Mathur , M. Bibes,and A. Barthélémy, Nature Nanotechnology, **7**, 101, (2012)

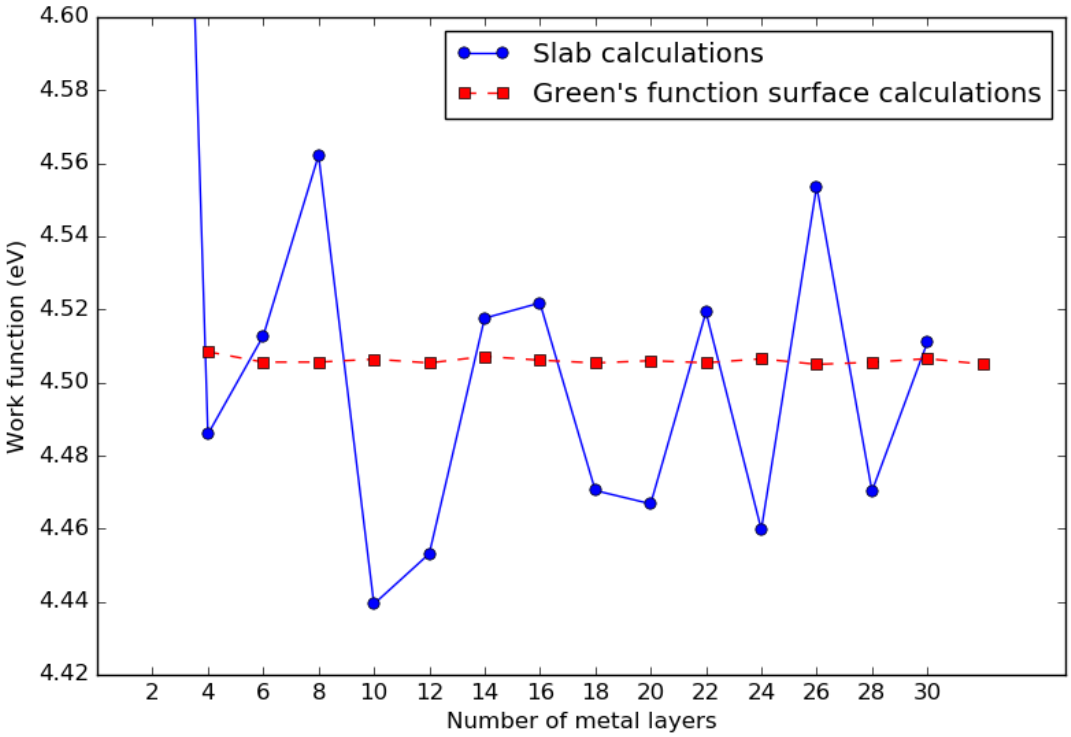


Fig. 3.109: Convergence of calculated Ag(100) work functions wrt. the number of metal layers. Work functions computed using the Greens function surface model converge much faster than those calculated with the traditional slab model. For surface configurations, the number of metal layers includes the two electrode atoms.

One of the most studied FE materials is barium titanate ($BaTiO_3$), which is the topic for this tutorial. Before continuing with the calculations, let us briefly summarize some central theoretical concepts first.

Modern theory of polarization

The theoretical understanding of FE materials is described by the so-called modern theory of polarization[3]. It is common to divide the polarization of a material into electronic and ionic parts. The latter is calculated using a simple classical electrostatic sum of point charges

$$\mathbf{P}_i = \frac{|e|}{\Omega} \sum_\nu Z^\nu_{\mathrm{ion}} \mathbf{r}^\nu,$$

where Z^ν_{ion} and $\mathbf{r}^\nu$ are the valence charge and position vector of atom ν, Ω is the unit cell volume, and the sum runs over all ions in the unit cell.

The electronic contribution to the polarization is obtained as[Page 1453, 3]

$$\mathbf{P}_e = -\frac{2|e|i}{(2\pi)^3} \int_A d\mathbf{k}_\perp \sum_{n=1}^{M} \int_0^{G_\parallel} \langle u_{\mathbf{k},n} | \frac{\partial}{\partial k_\parallel} | u_{\mathbf{k},n} \rangle dk_\parallel,$$

where the sum runs over occupied bands, and where $k_\parallel$ is parallel to the direction of polarization, and $G_\parallel$ is a reciprocal lattice vector in the same direction. The states $|u_{\mathbf{k},n}\rangle$ are the cell-periodic parts of the Bloch functions, $\psi_{\mathbf{k},n}(\mathbf{r}) = u_{n,\mathbf{k}}(\mathbf{r}) e^{i\mathbf{k}\cdot\mathbf{r}}$. The last integral is known as the *Berry phase*. The integral over the perpendicular directions can easily be converged with a few number of k-points. The number of k-points in the parallel direction should be larger, however.

The total polarization is simply the sum of the electronic and ionic contributions,

$$\mathbf{P}_t = \mathbf{P}_i + \mathbf{P}_e.$$

An important finding in Ref.[3] was that the polarization is a multivalued quantity, and in fact forms a lattice. The reason is that the electronic polarization $\mathbf{P}_e$ is determined by the Berry phase, which is only defined modulo 2π. Likewise, the ionic contribution $\mathbf{P}_i$ would attain a different value if all ionic positions were displaced by a lattice constant in either direction.

The polarization is thus a periodic function, and the period is called the polarization quantum, $\mathbf{P}^j_q = \frac{|e|\mathbf{R}^j}{\Omega}$, where $|e|$ is the electronic charge, $\mathbf{R}^j$ is the lattice vector j, and Ω is the unit cell volume.

Given the multivalued nature of the polarization, it is perhaps not surprising that only *differences* in polarization, $\Delta\mathbf{P}$, between two different structures is a well-defined property.

ATK computes and reports the electronic and ionic contributions separately, and also reports the polarization quantum.

Important: Note that the implementation does not work for metallic systems. Usage in 2D systems should be done with thorough testing of the used settings and results.

---

[3] R. King-Smith, and D. Vanderbilt, Phys. Rev. B, **47**, 1651, (1993)

Spontaneous polarization of ferroelectric $BaTiO_3$

The BaTiO3 crystal structure

Barium titanate ($BaTiO_3$) has a tetragonal crystal structure at room temperature, where the unit cell is slightly elongated in the c-direction. An internal stress further shifts the fractional coordinates in the c-direction away from their high symmetry positions. In this tutorial we use the experimental lattice constants and coordinates as obtained from the Inorganic Crystal Structure Database (ICSD) The structure is given in the QuantumATK format below[4]

```
# Set up lattice
lattice = SimpleTetragonal(3.9945*Angstrom, 4.0335*Angstrom)

# Define elements
elements = [Barium, Titanium, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.      ,  0.      ,  0.      ],
                          [ 0.5     ,  0.5     ,  0.51427 ],
                          [ 0.5     ,  0.5     ,  0.974477],
                          [ 0.5     ,  0.      ,  0.487618],
                          [ 0.      ,  0.5     ,  0.487618]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

Setting up the calculation

You will in this section set up a DFT calculation using the local density approximation (LDA) for the $BaTiO_3$ crystal and calculate the polarization. You will use **QuantumATK** for the calculation, and it is recommended that you go through the Basic QuantumATK Tutorial to be familiar with the basic work flow.

Start up **QuantumATK** and create a new project for this tutorial. Use a new, empty directory. Select the text for the $BaTiO_3$ structure in the python script above and drag it onto the **Script Generator** icon . The tool will interpret the script and open up with the imported geometry

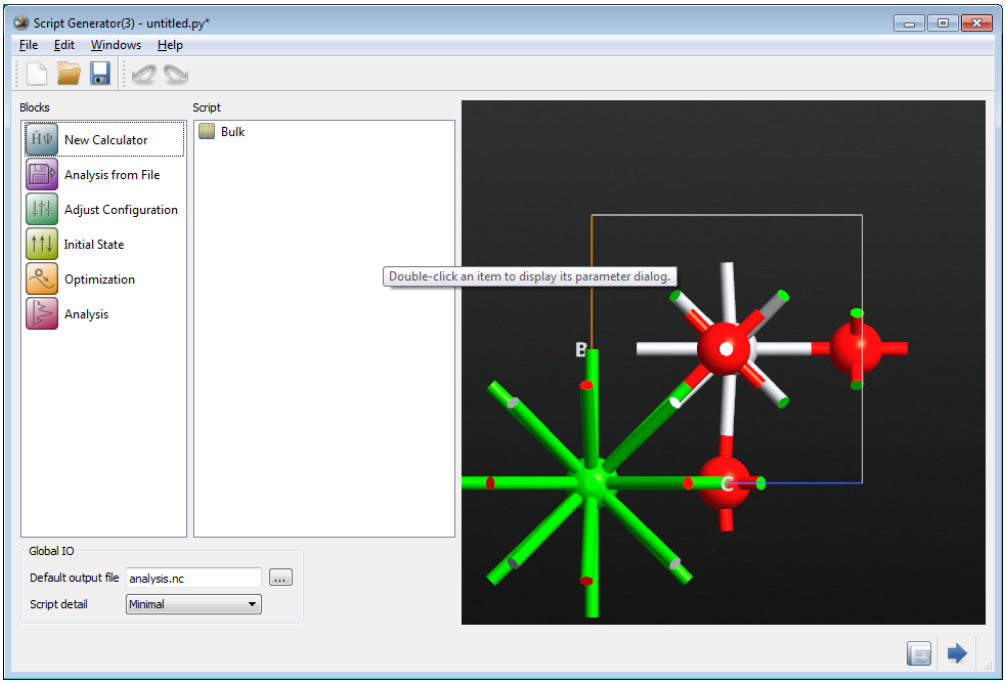
---

Tip: Alternatively you can save the script to a file in the project directory and drag and drop the file to the Script Generator from the **QuantumATK** main window.

Next do the following steps:

- Change the Default output file to `BaTiO3_lda.hdf5`.
- Add a **New Calculator**.

[4] H.Megaw, Acta Crystallographica, **15**, 972, (1962)



•Add *Analysis* → *Polarization*.

Now double-click the **New Calculator** block in the Script panel to open the calculator widget.

Set the k-point sampling to 5x5x5; the other default settings are fine.

The next step is to adjust the settings for the polarization analysis. Double-click the **Polarization** block.

Increase the number of k-points on the diagonal to 20. This is the number of k-points along the lines of integration and needs to be relatively high. You should always check for convergence by comparing calculations with different numbers of k-points. The other k-points with values of 5 correspond to the number of transverse k-points used for averaging over the Brillouin zone. The polarization values converge relatively fast with respect to the number of transverse k-points and we thus use the default value.

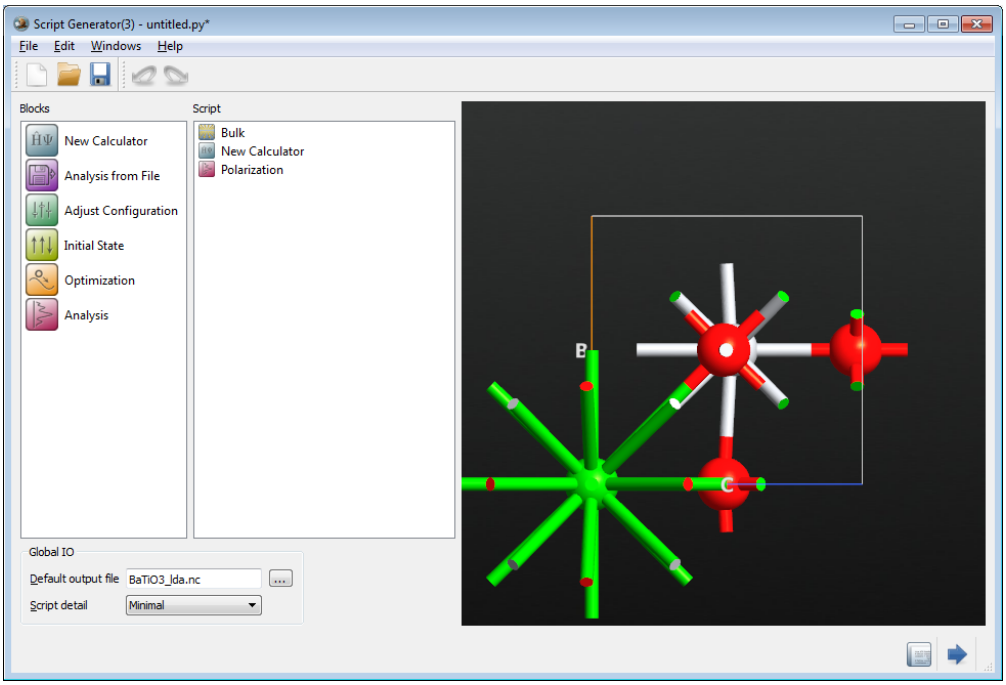
You have now finished the script setup. Save the script as BaTiO3_lda.py. Send the script to the **Job Manager** and run the job. After a few minutes the calculation has finished and you can inspect the results.

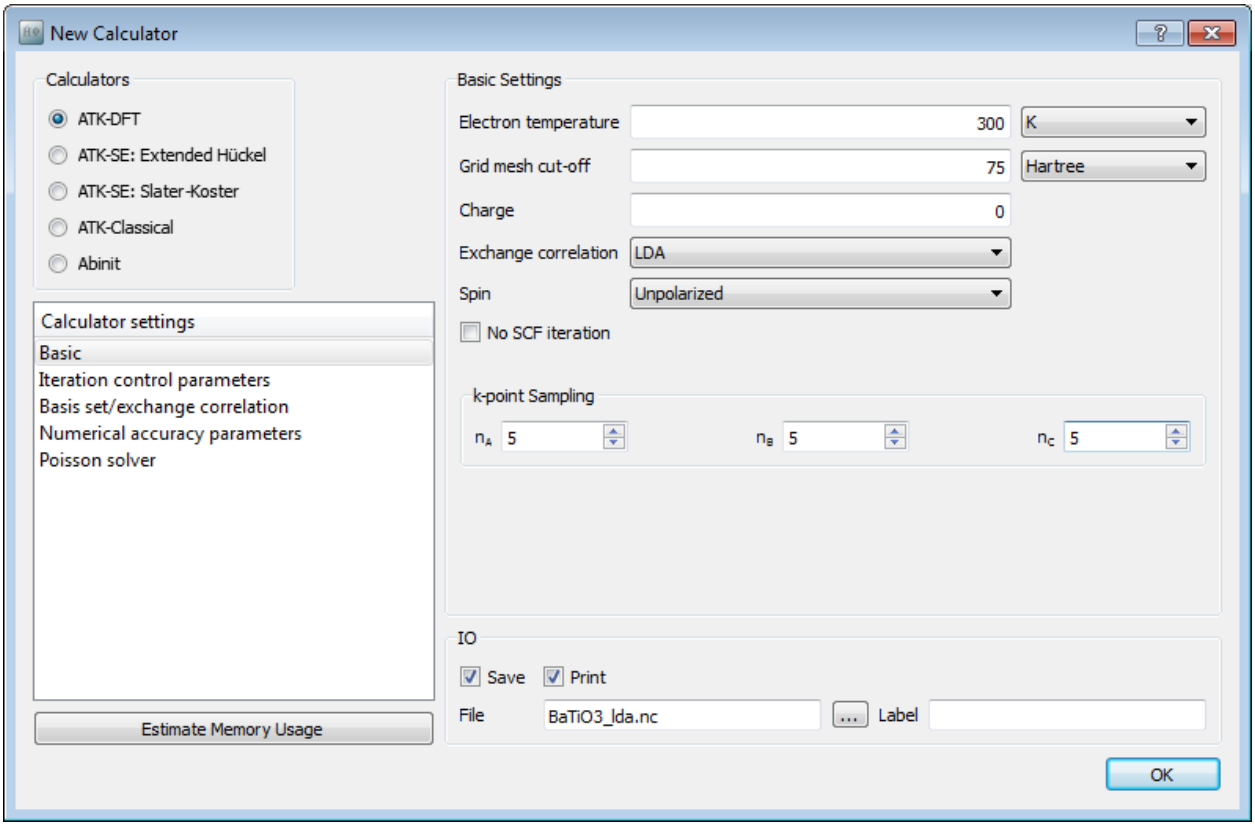
Analyzing the results

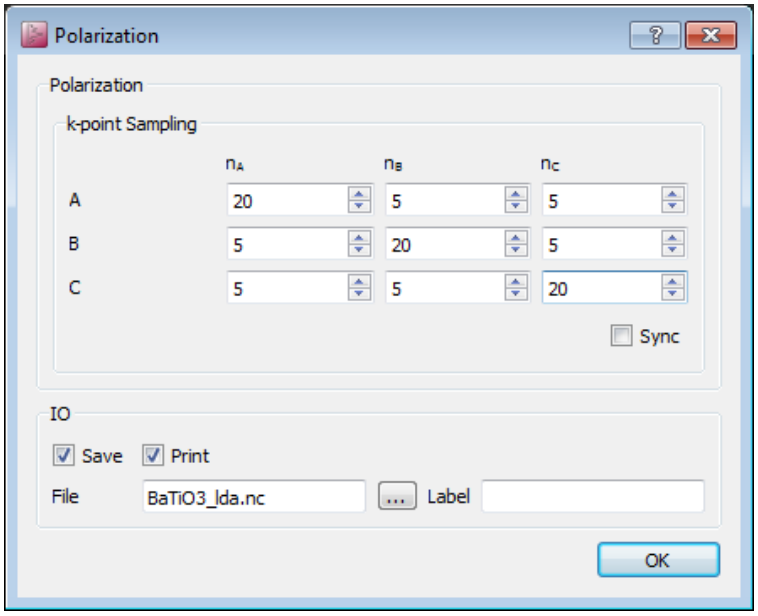
To inspect the calculated polarization reported in the log file, scroll down to the end of the log file and you will find a report as shown below.

```
+-------------------------------------------------------------------------------+
| Polarization                                                                  |
+-------------------------------------------------------------------------------+
| Electronic fractional polarization.                                           |
| Values wrapped to the interval [-0.5,0.5]                                     |
|       [ -1.25164671e-15 ]                                                     |
|  Pe=  [ -6.42868666e-16 ]                                                     |
|       [ -4.71901310e-01 ]                                                     |
+-------------------------------------------------------------------------------+
| Ionic fractional polarization.                                                |
```

(continues on next page)


Script Generator(3) - untitled.py*
File Edit Windows Help
Blocks
New Calculator
Analysis from File
Adjust Configuration
Initial State
Optimization
Analysis
Script
Bulk
New Calculator
Polarization
Global IO
Default output file
BaTiO3_lda.nc
Script detail
Minimal
B
C

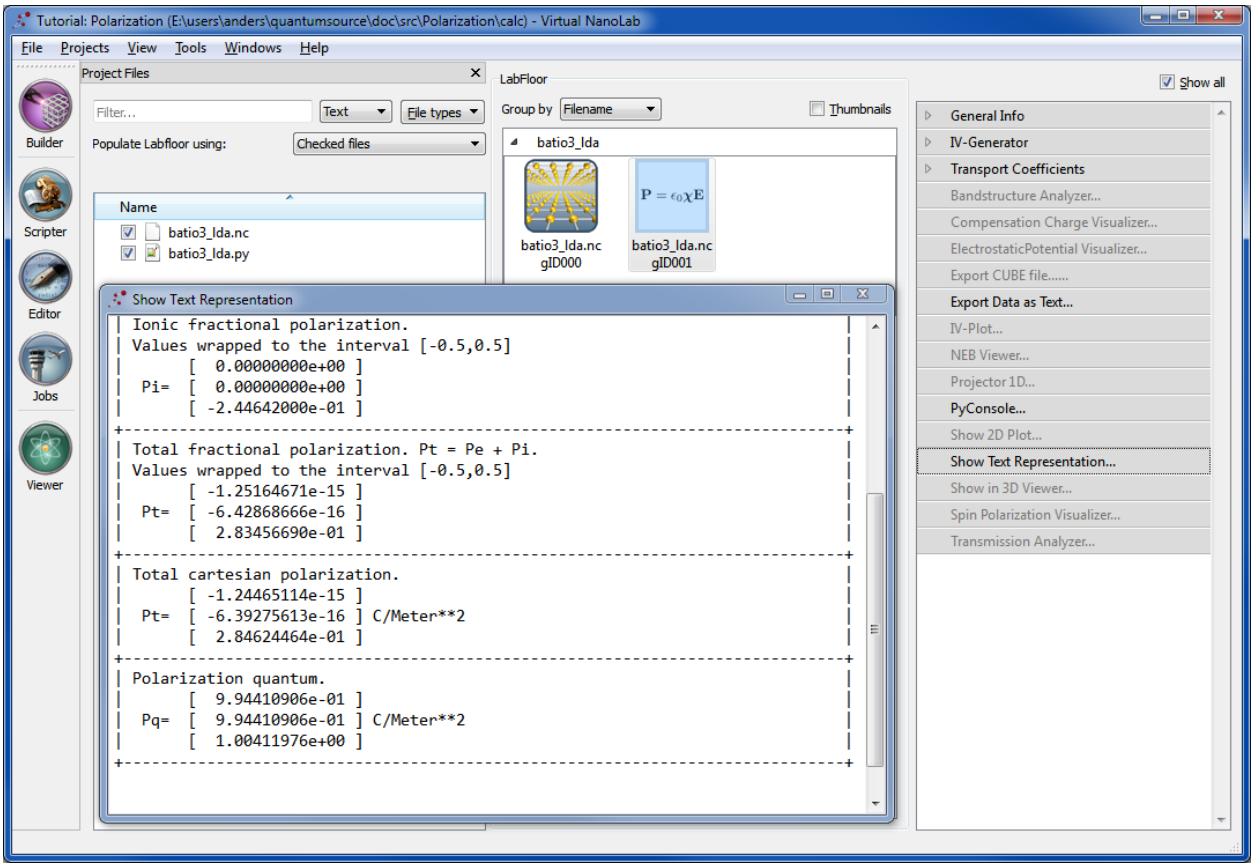

New Calculator
Calculators
ATK-DFT
ATK-SE: Extended Hückel
ATK-SE: Slater-Koster
ATK-Classical
Abinit
Calculator settings
Basic
Iteration control parameters
Basis set/exchange correlation
Numerical accuracy parameters
Poisson solver
Estimate Memory Usage
Basic Settings
Electron temperature
300
K
Grid mesh cut-off
75
Hartree
Charge
0
Exchange correlation
LDA
Spin
Unpolarized
No SCF iteration
k-point Sampling
n_A 5
n_B 5
n_C 5
IO
Save
Print
File
BaTiO3_lda.nc
Label
OK



(continued from previous page)

```
| Values wrapped to the interval [-0.5,0.5]                                    |
|       [  0.00000000e+00 ]                                                    |
|  Pi=  [  0.00000000e+00 ]                                                    |
|       [ -2.44642000e-01 ]                                                    |
+------------------------------------------------------------------------------+
| Total fractional polarization. Pt = Pe + Pi.                                 |
| Values wrapped to the interval [-0.5,0.5]                                    |
|       [ -1.25164671e-15 ]                                                    |
|  Pt=  [ -6.42868666e-16 ]                                                    |
|       [  2.83456690e-01 ]                                                    |
+------------------------------------------------------------------------------+
| Total cartesian polarization.                                                |
|       [ -1.24465114e-15 ]                                                    |
|  Pt=  [ -6.39275613e-16 ] C/Meter**2                                         |
|       [  2.84624464e-01 ]                                                    |
+------------------------------------------------------------------------------+
| Polarization quantum.                                                        |
|       [  9.94410906e-01 ]                                                    |
|  Pq=  [  9.94410906e-01 ] C/Meter**2                                         |
|       [  1.00411976e+00 ]                                                    |
+------------------------------------------------------------------------------+
```

Tip: The results can also be inspected by selecting the polarization object in the file BaTiO3_lda.hdf5 on the **LabFloor** and clicking **Show Text Representation**



Notes

The output contains five calculated quantities:

•First, the electronic fractional polarization $\mathbf{P}_e$ is calculated from the Berry phase obtained from the occupied bands, as described in *Modern theory of polarization*. The three values correspond to the x, y, and z directions.

•The second quantity $\mathbf{P}_i$ is the purely ionic fractional polarization $\mathbf{P}_i = \sum_j Z_j^{ion} \tau_j$, where Z_j^{ion} and τ_j are the valence charge and fractional coordinate of of atom j.

•The third quantity, $\mathbf{P}_t$, is the total fractional polarization, which is the sum of the electronic and ionic parts. As discussed in *Modern theory of polarization*, the polarization is a multivalued quantity, and therefore all fractional polarizations are wrapped to the interval [-0.5,0.5], which explains the sign change of the polarization in the z-direction. The sum $\mathbf{P}_e(z) + \mathbf{P}_i(z) = -0.717$ is outside the range and is thus wrapped by adding a fractional quantization quantum (equal to 1), i.e. $\mathbf{P}_t(z) = -0.717 + 1 = 0.283$.

•The fourth quantity is the total polarization $\mathbf{P}_t$ in Cartesian coordinates, expressed in units of C/m^2.

•The fifth quantity is the polarization quantum $\mathbf{P}_q$ introduced in *Modern theory of polarization*. It is relevant that $\mathbf{P}_t$ is small compared to $\mathbf{P}_q$.

According to the modern theory of polarization[3], only the difference in polarization between two configurations is a well-defined property. In order to calculate the spontaneous polarization of tetragonal $BaTiO_3$, it is thus necessary to also compute the polarization of the centrosymmetric, undistorted structure given below:

```
# Set up lattice
lattice = SimpleTetragonal(3.9945*Angstrom, 4.0335*Angstrom)
```

(continues on next page)

(continued from previous page)

```
# Define elements
elements = [Barium, Titanium, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.0    ,  0.      ,  0.  ],
                          [ 0.5    ,  0.5     ,  0.5 ],
                          [ 0.5    ,  0.5     ,  1.0 ],
                          [ 0.5    ,  0.      ,  0.5 ],
                          [ 0.0    ,  0.5     ,  0.5 ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

You may repeat the above steps to calculate the polarization for this structure. The results are that all polarization components are zero. The spontaneous polarization of tetragonal $BaTiO_3$, thus corresponds to the values reported above for the distorted structure.

The calculated value for the total Cartesian polarization in the z-direction $\mathbf{P}_t(z) = 0.284$ C/m^2 compares well with the experimental value 0.26 C/m^2[5].

References

3.11.4 Visualize the LUMO state of a water molecule

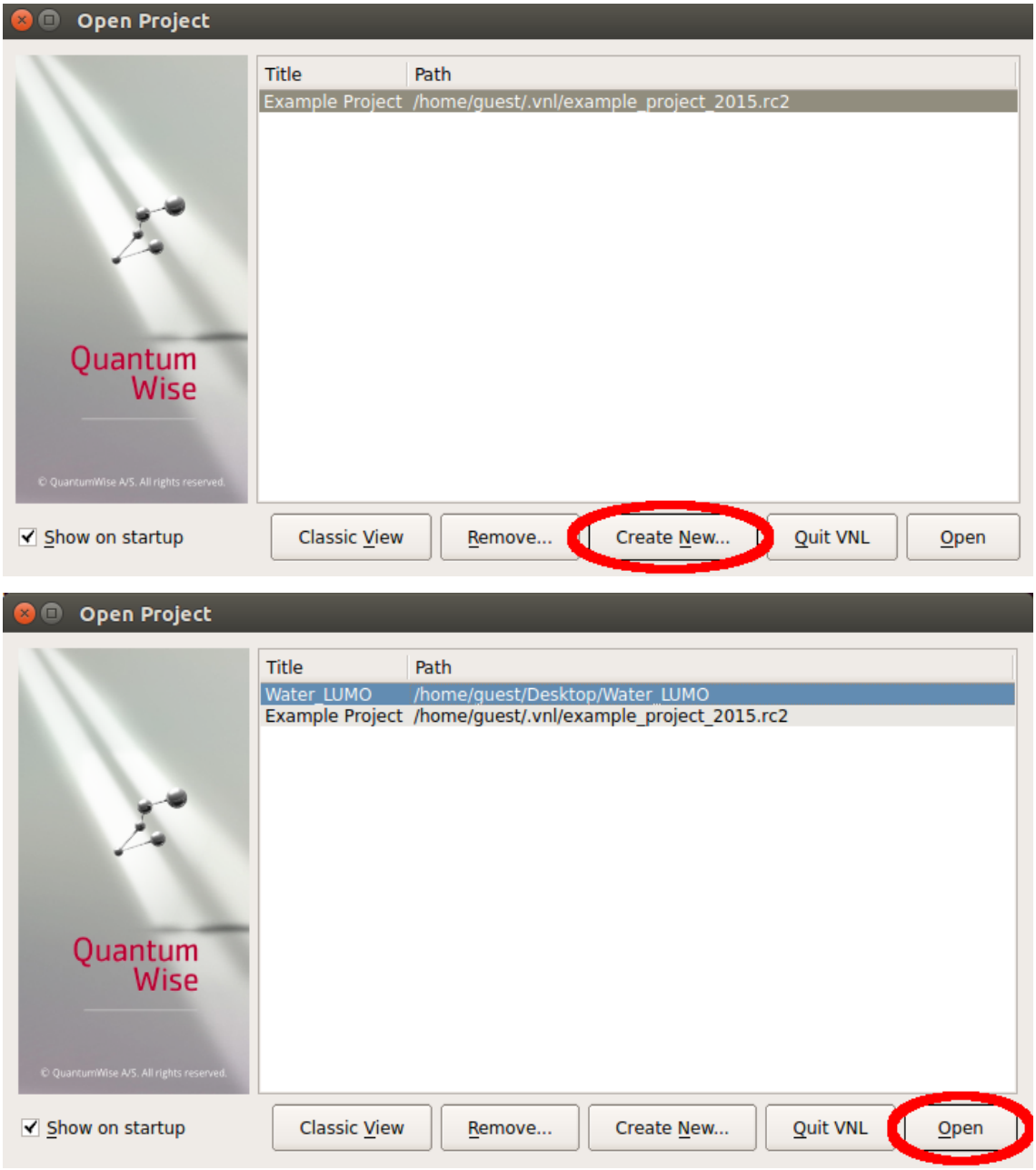
In this tutorial you will learn how to visualize a molecular orbital. As an example, you will visualize the LUMO state of water molecule. Specifically, you will:

- Build the molecule.
- Compute a single point calculation at the DFT level.
- Obtain the molecular eigenenergy spectrum and visualize the LUMO state.

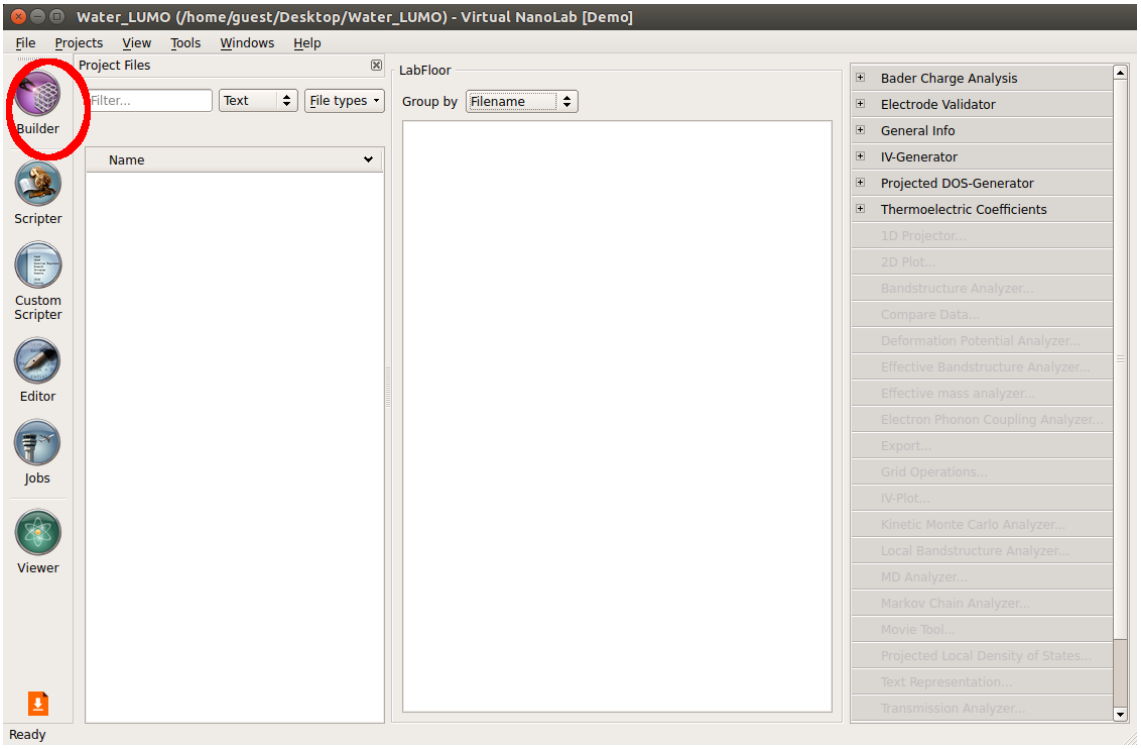
Building the molecule

1. Open **QuantumATK** and create a new project by clicking **Create New**. Give the project a *Title* (here: Water_LUMO), select the folder where data will be stored, then click **OK** and **Open** to start the project.

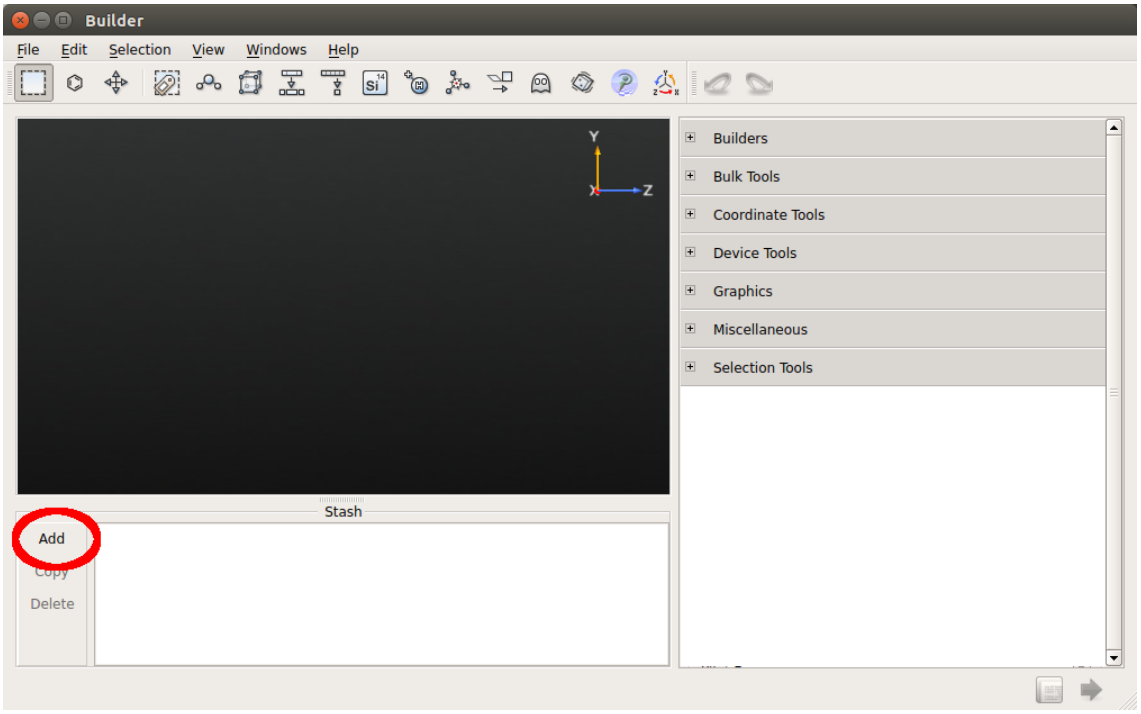
[5] H. Wieder, Phys. Rev. , **99**, 1161, (1955)



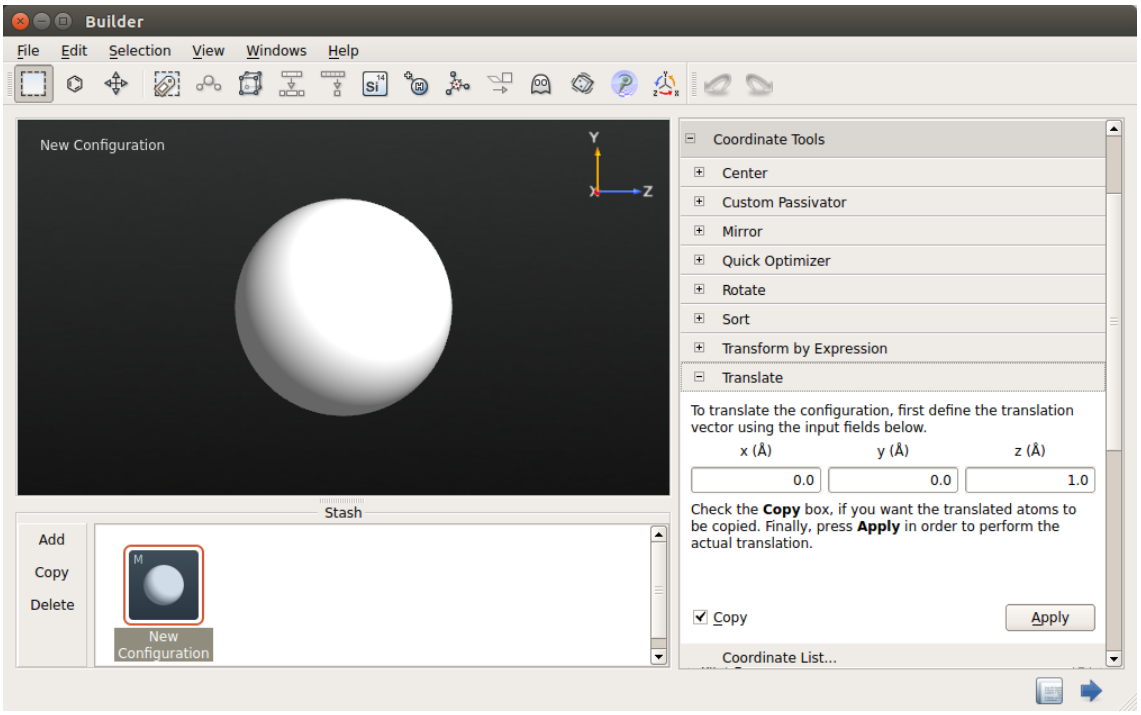
2.In the **QuantumATK** main window, click on the icon to open the **Builder**.



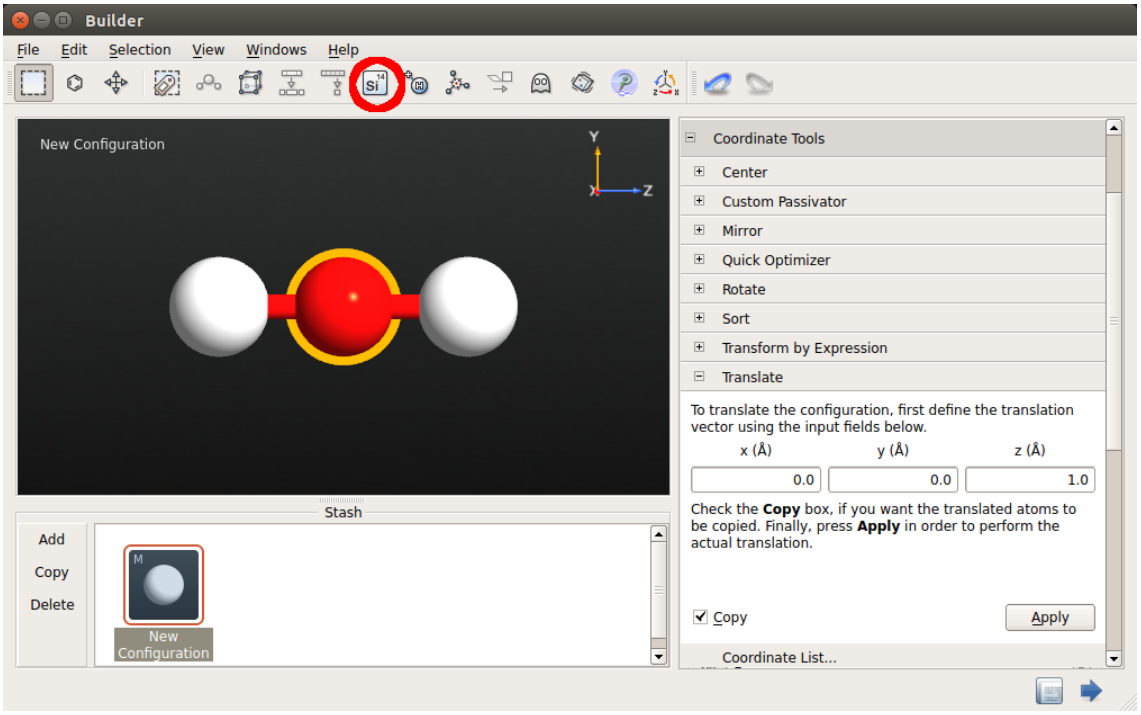
3.Go to the **Stash**, click *Add* → *New Configuration*. A hydrogen atom appears in the **Stash** and in the preview window of the **Builder**.



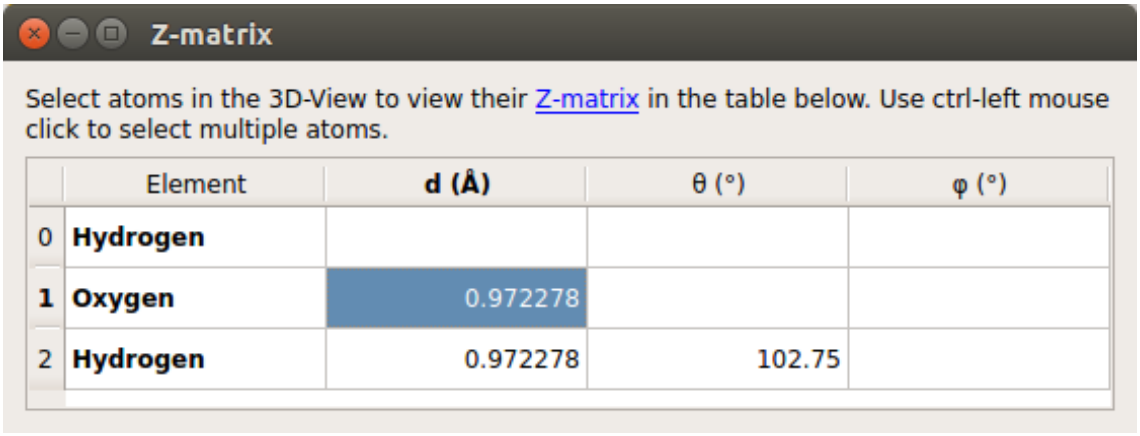
4.Go to *Coordinate Tools* → *Translate*. Tick the **Copy** option, increase Z to 1 Å, and click the **Apply** button twice to add two more hydrogen atoms.

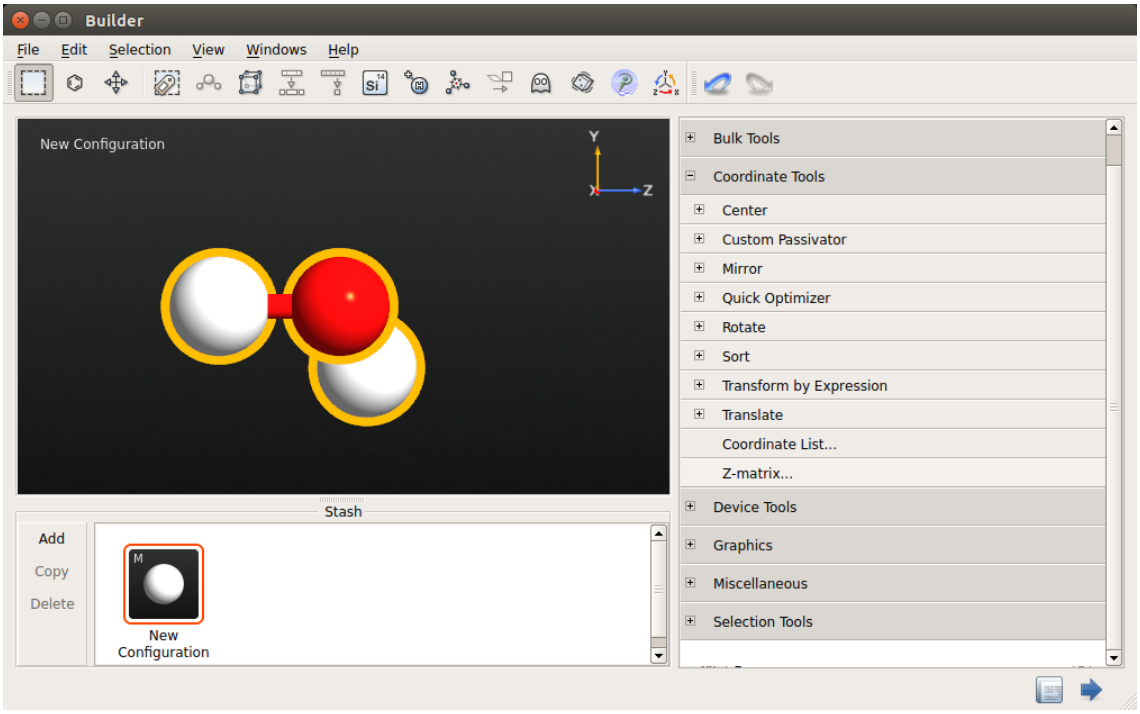


5.Go to the preview window. Press `Ctrl` + `R` to center the view and select the atom in the middle. Use the **Modify Element** tool in the toolbar to change it to oxygen.



6.Select all atoms using the left mouse button while holding down the `Ctrl` button of your keyboard. Go to *Coordinate Tools* → *Z-matrix* and set the distances and angles as shown in the picture below (H-O distance 0.972278 Å and H-O-H angle 102.75 degrees).





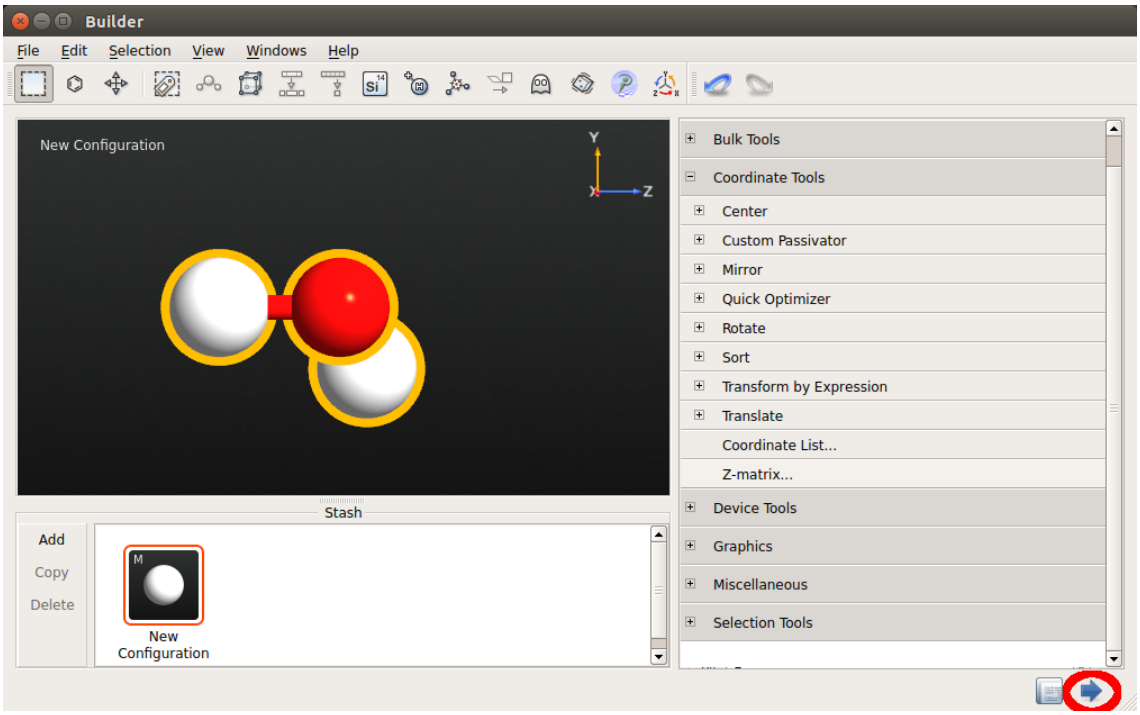
You have now built the water molecule completely from scratch!

Note: It is of course not necessary to always build structures from scratch. Instead, you can import configurations from the **Database**:

- Go to the **Stash** and select *Add* → *From Database*.
- From the menu click *Databases* → *Molecules* and search for *Water* in the search window.
- Select *Water* and add it to **Stash** by double-clicking it or by using the icon.

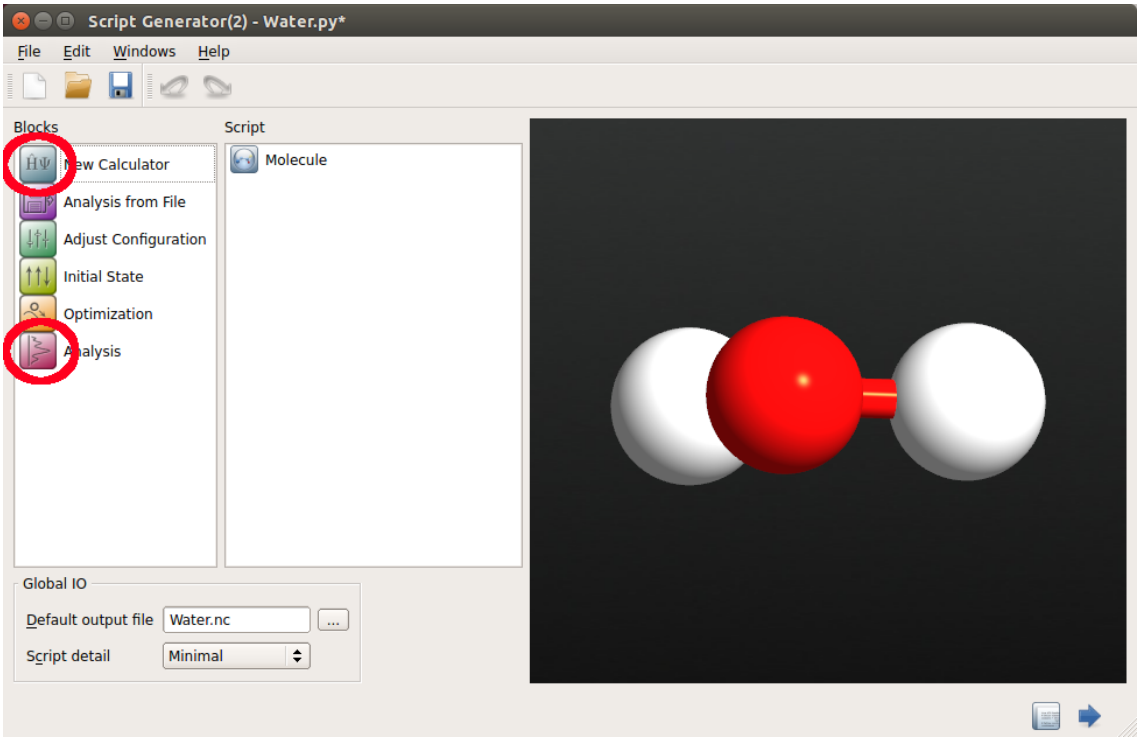
Calculating the LUMO state

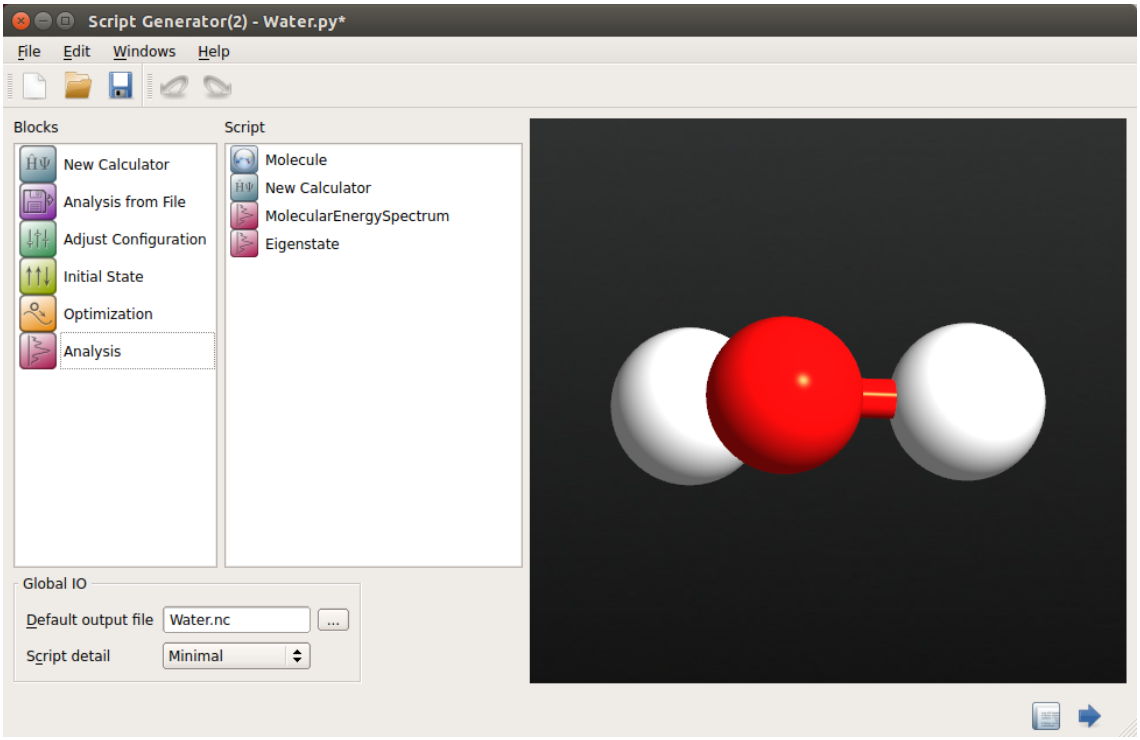
1.In the **Builder**, click in the lower right-hand corner, and select **Script Generator**.



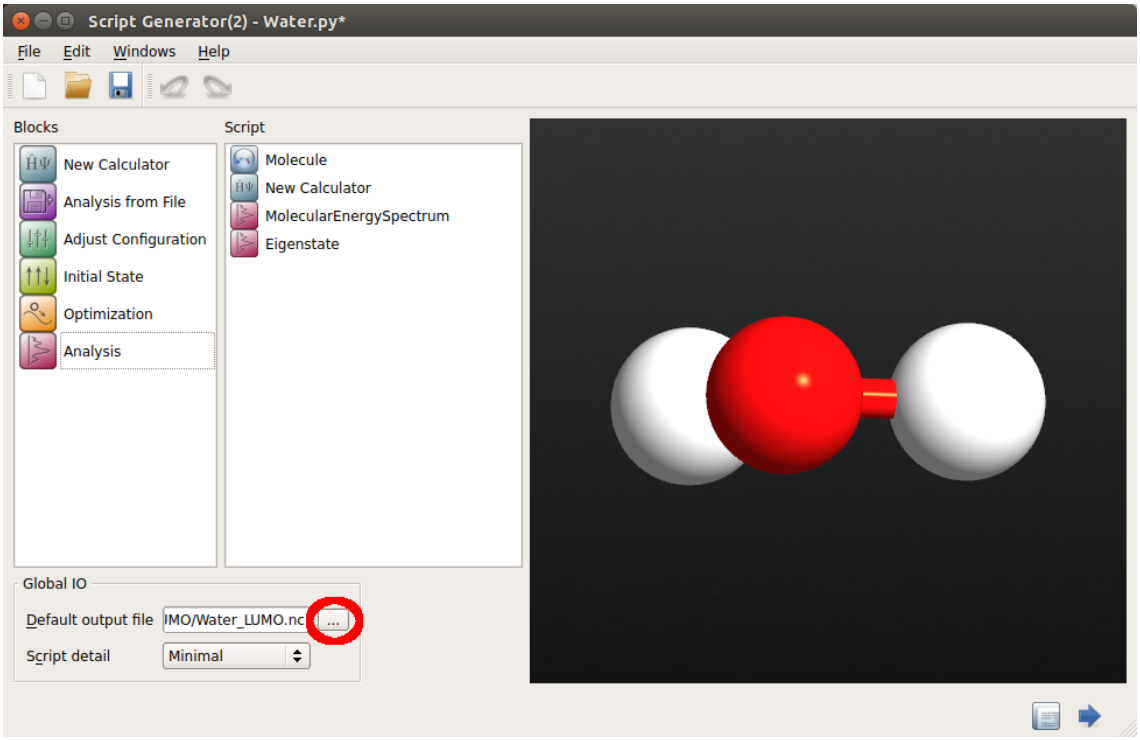
2.In the **Scripter**, add the following blocks by double-clicking the correspoding icons in the Blocks panel on the left-hand side of the screen:

- New Calculator.
- *Analysis → MolecularEnergySpectrum*.
- *Analysis → Eigenstate*.



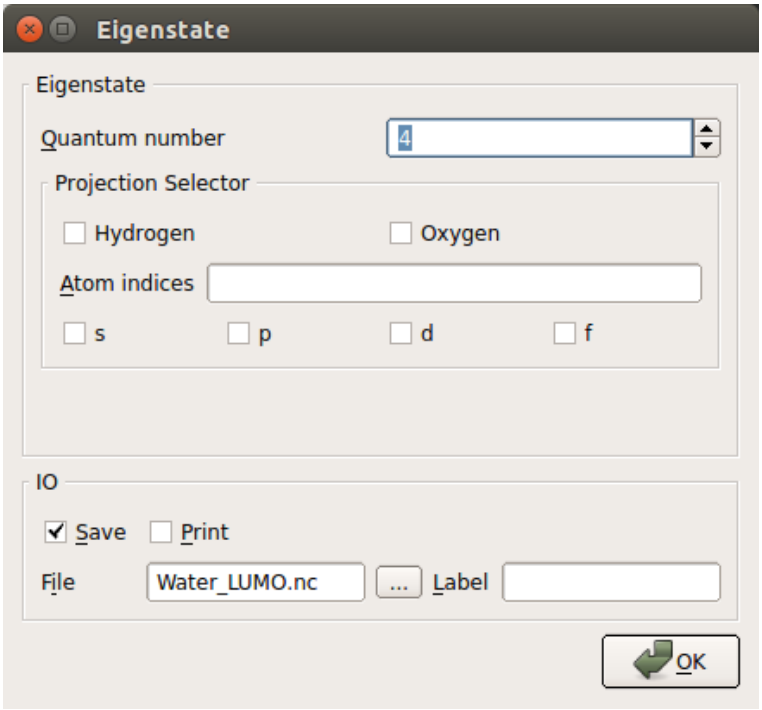


3. Go to **Global IO** and define a suitable location and name of the HDF5 file in which your results will be saved, using the ... button. In this tutorial, save this file as `Water_LUMO.hdf5`:



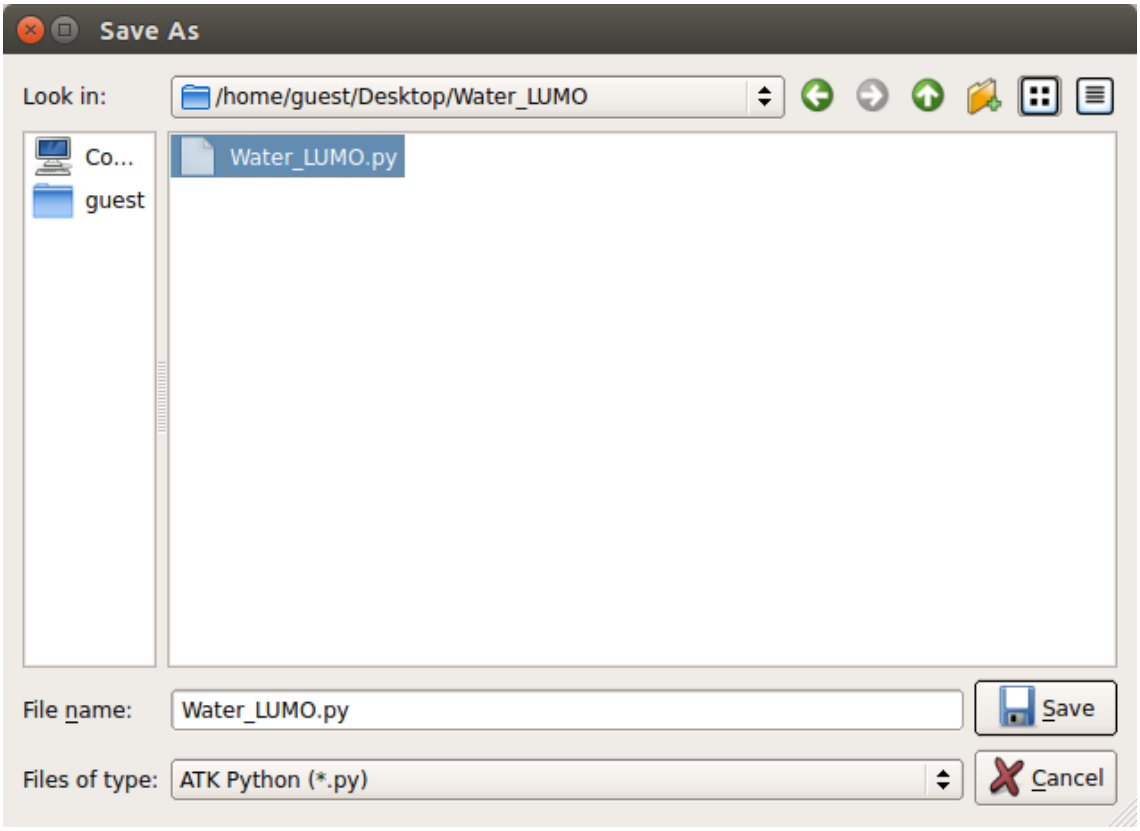
4.Now double-click the **Eigenstate** block and edit it:

•Since water has 8 valence electrons (one for each hydrogen and six for oxygen; the two 1s electrons are part of the pseudocore in QuantumATK), there will be 4 occupied states, each one doubly degenerate since this is a spin-independent calculation. Therefore set the quantum number to 4 for the LUMO state.

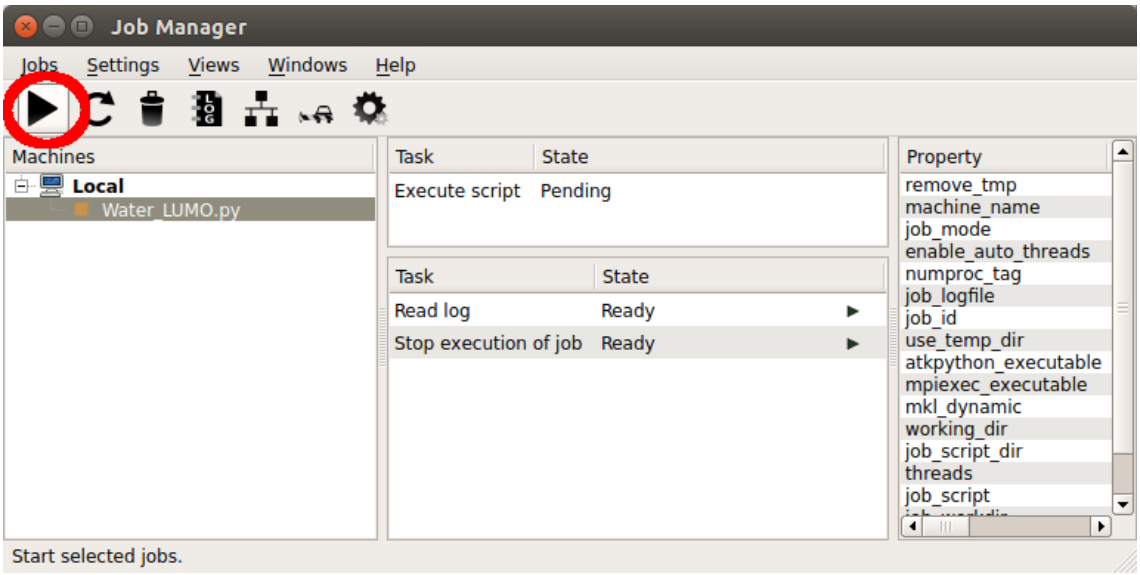


5.To run the calculation, use the **Send to** button to send the script to the **Job Manager**. Then, save the script

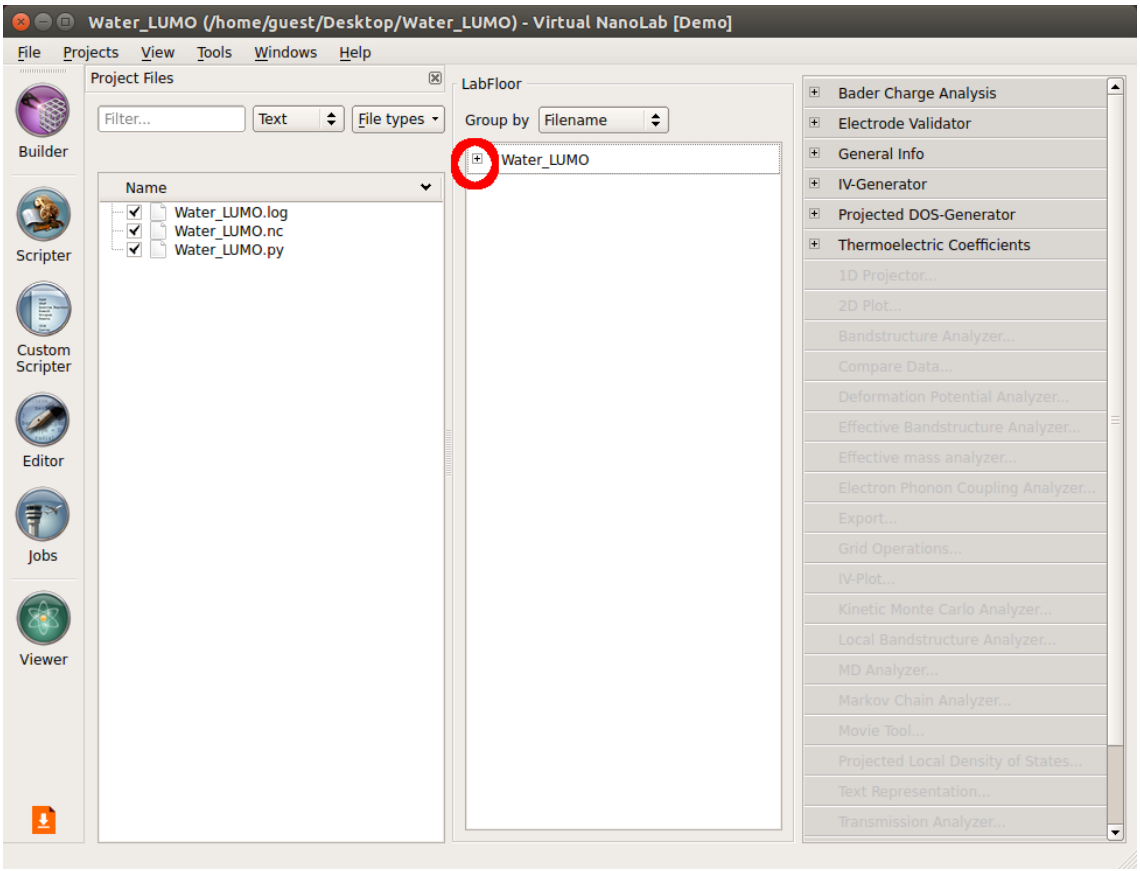
using the window that appears.



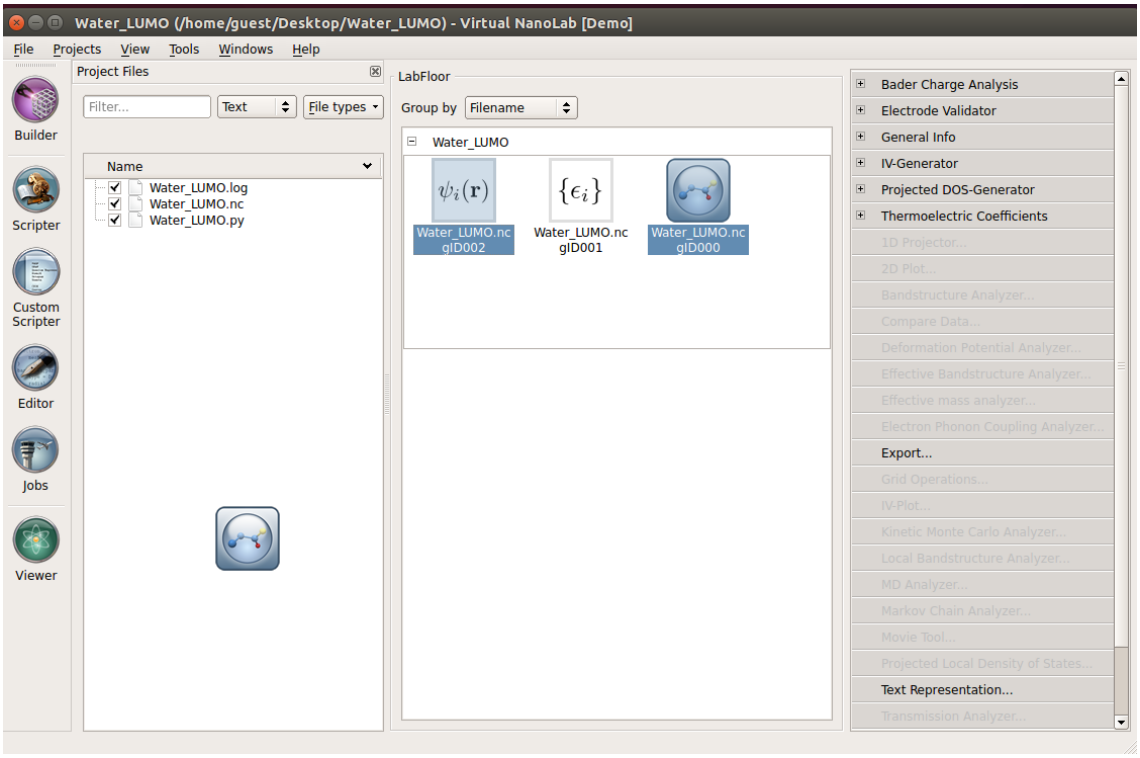
6. When the **Job Manager** opens up, click on the ▶ button to run the script.

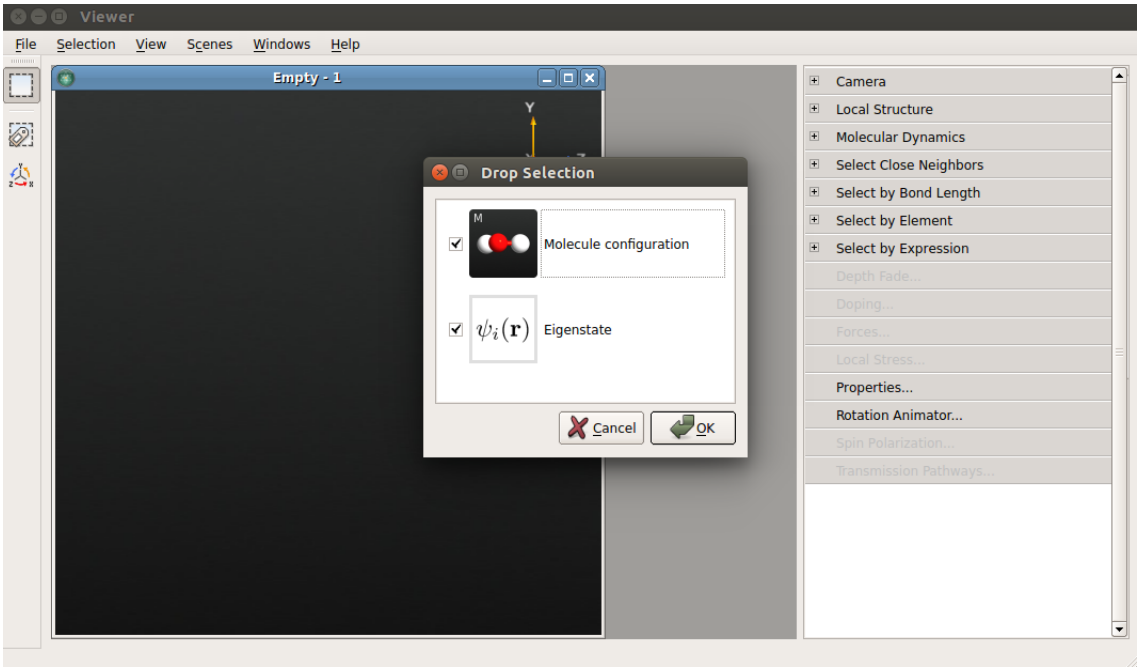


7. When the script has finished (it takes a minute or so), navigate to the **LabFloor** in the **QuantumATK** main window and click on the ⊞ tab of the HDF5 file in order to display the objects contained there.



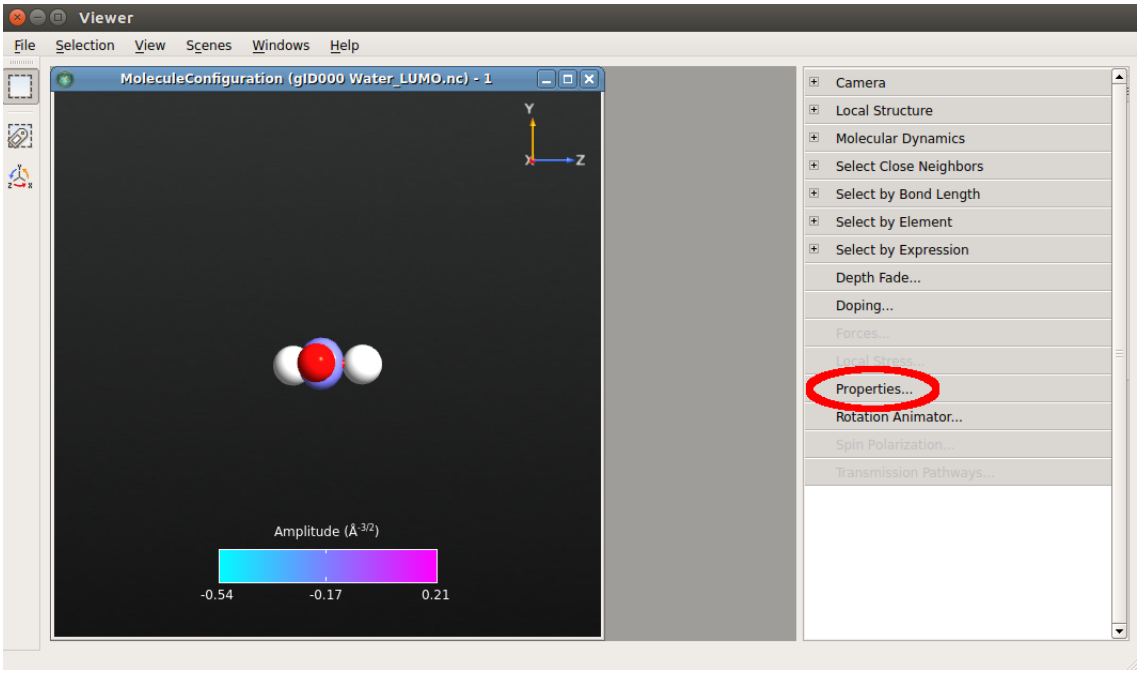
8. Select both the **Eigenstate** and **MolecularConfiguration** objects stored in the HDF5 file (use a left mouse click while holding down `Ctrl`), and drag and drop both objects onto the **Viewer**, which will then pop up.





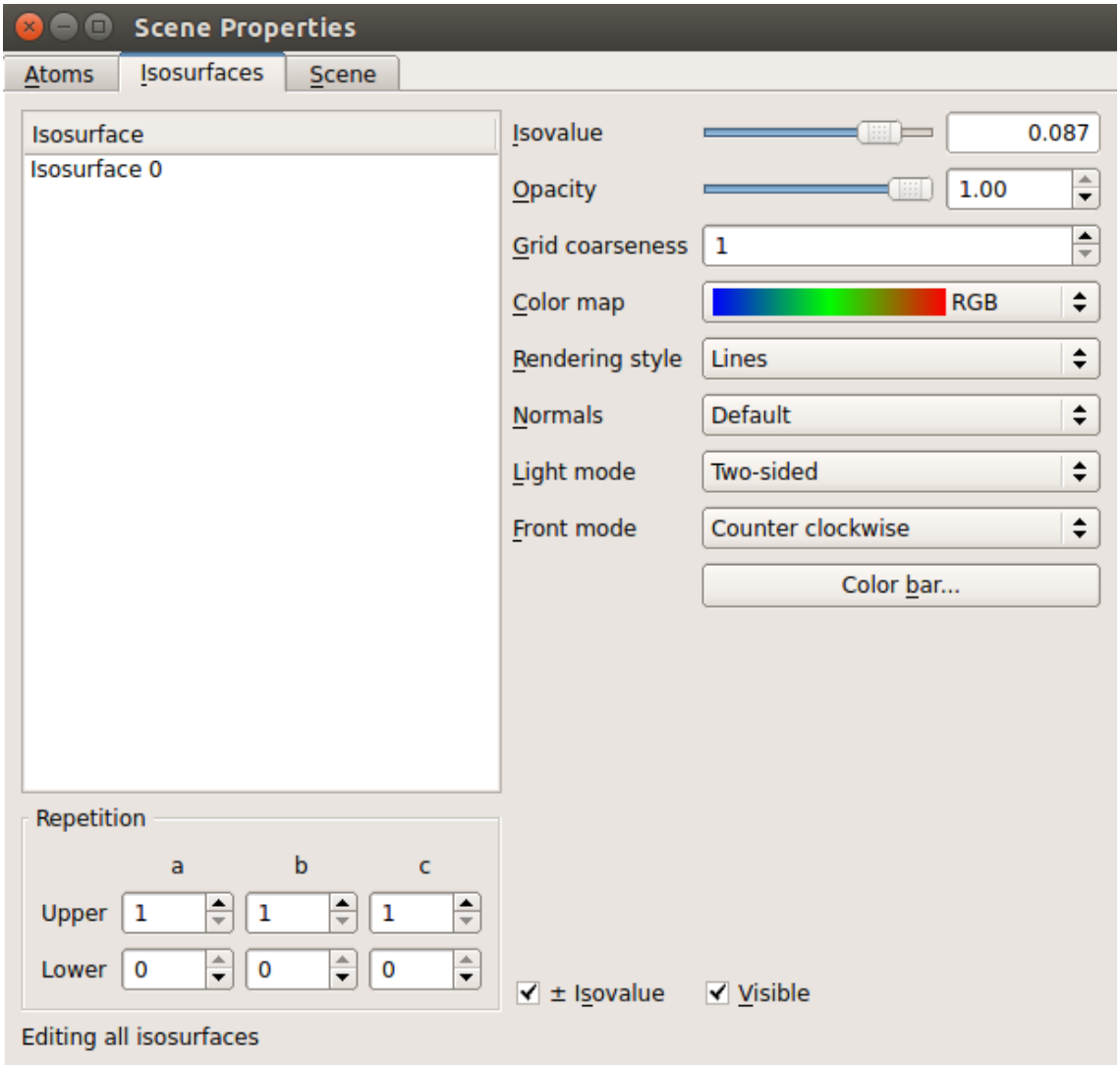
9.Click **OK**, and click **OK** again in the next window.

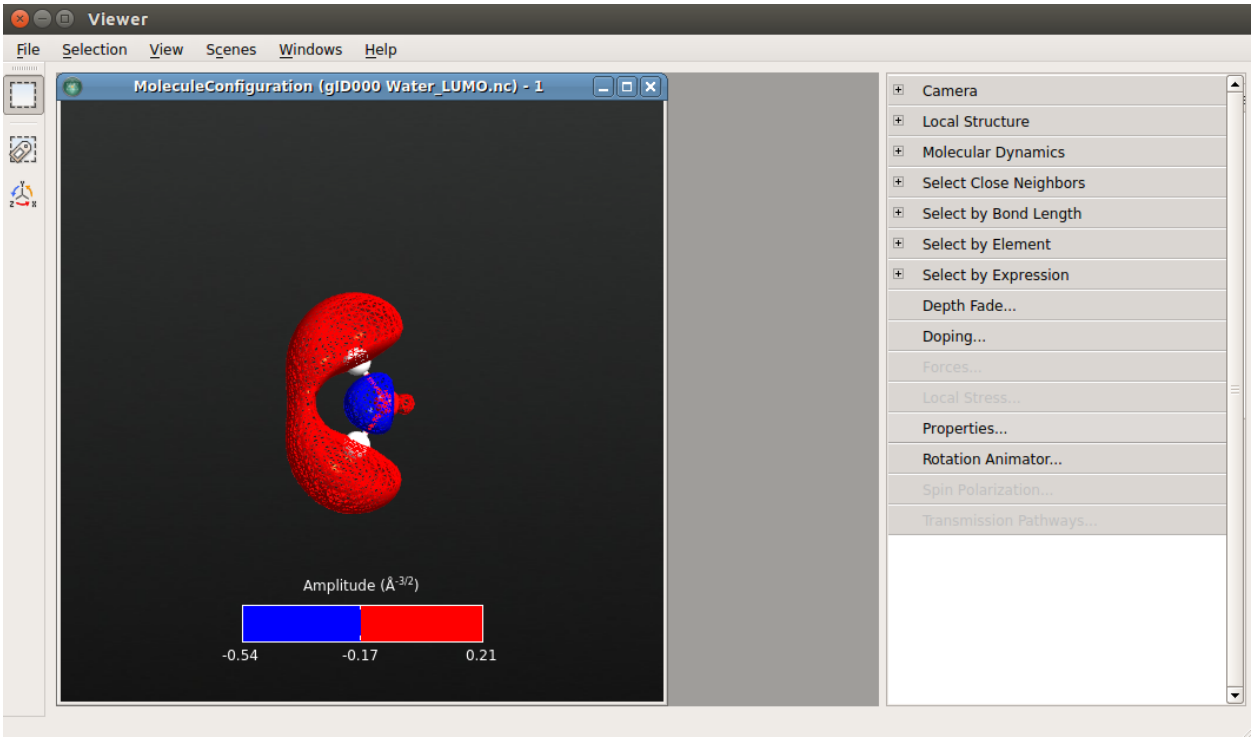
10.In the **Viewer**, click on **Properties**.



11. To obtain the same image as below, do as follows:

- Go to *Scene Properties* → *Scene* and uncheck *Compass enabled*.
- Go to *Scene Properties* → *Isosurface* and set the Isovalue to 0.087.
- Go to *Scene Properties* → *Isosurface* and set the Rendering style to Lines.
- Go to *Scene Properties* → *Atoms* and set the atoms and bonds radious to 0.25 and 0.05, respectively.


Scene Properties
Atoms
Isosurfaces
Scene
Isosurface
Isosurface 0
Isovalue
0.087
Opacity
1.00
Grid coarseness
1
Color map
RGB
Rendering style
Lines
Normals
Default
Light mode
Two-sided
Front mode
Counter clockwise
Color bar...
Repetition
a
b
c
Upper
1
1
1
Lower
0
0
0
± Isovalue
Visible
Editing all isosurfaces



Attention: Please note the use of the $\pm$*isovalue* option. When checked, it divides the full range of available isovalues into two equally large ranges, and plots the two ranges in complimentary colors.

3.11.5 How to calculate reaction barriers using the Nudged Elastic Band (NEB) method

Version: U-2022.12

Downloads & Links

`Pt_bulk_workflow.hdf5` `relax_initial_and_final_WF.hdf5` `NEB_jump_workflow.hdf5`
`NudgedElasticBand`

In this tutorial you will learn how to calculate reaction barriers for a reaction with well-defined initial and final states. As an example, we will use the diffusion of a single platinum ad-atom on a Pt(100) surface, but the methodology can be used in any situation where you can define an initial and final state of the system, connected by movement of one or more atoms.

Specifically, in this tutorial you will:

- create the Pt(100) surface with a Pt ad-atom;
- create a high-quality initial NEB path and run the NEB calculation;
- analyze the results and compare with the literature.

In order to have more details about the parameters used to construct and optimize the NEB object, you can check the *`NudgedElasticBand`* and `OptimizeNudgedElasticBand` pages in the manual.

Note: The NEB method employed in this tutorial requires the *a priori* knowledge of the initial and final configurations of the reaction, which is often a non-trivial task. However, other powerful methods, such as the Adaptive Kinetic Monte Carlo (AKMC) tool, allow study of reaction barriers in a system from just a user-defined initial state, as shown in *Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)*.

Create the initial and final states for the NEB

In this section you will set-up the structure of a Pt ad-atom on Pt(100), which will be used as the initial and final configurations in the NEB calculations. First, you will relax bulk platinum with the chosen calculator to ensure the Pt(100) slab does not have an inherent strain.

Relax bulk platinum

Open **QuantumATK** and create a new project (f.ex. *Pt_Pt100_NEB*). Relax bulk platinum with a **ForceFieldCalculator** using the *EAM_Pt_2004* parameter set and default optimization parameters. This simulation runs in a few seconds. If needed, you can download the workflow for this calculation here: `Pt_bulk_workflow.hdf5`.

Tip: Tutorial on geometry optimization: *Geometry optimization: CO/Pd(100)*

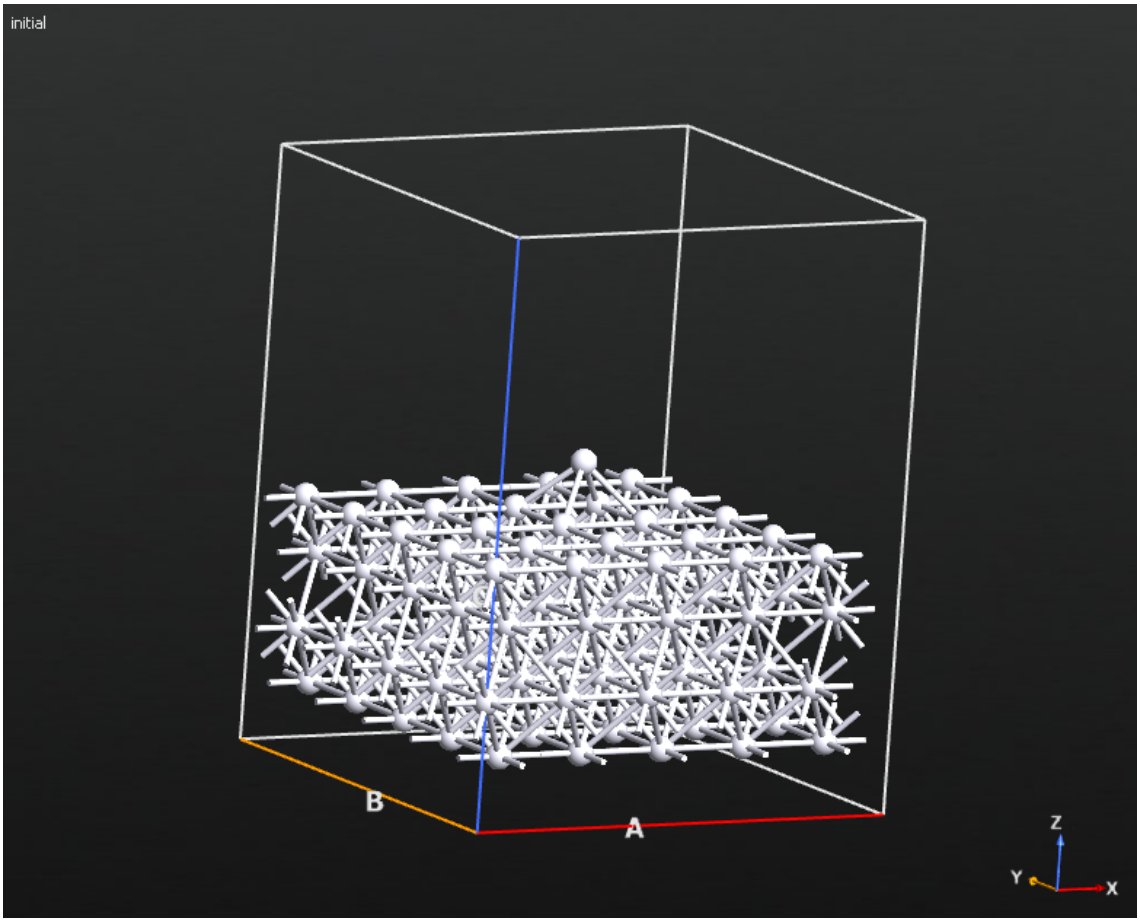
Create the initial state - a Pt(100) surface with a Pt ad-atom

1. Use the relaxed platinum bulk configuration to construct a 5×5 (100) slab with a thickness of 4 layers, using the **Surface (cleave)** tool in the **Builder**.

Tip: A guide to the **Surface (cleave)** tool: *Cleave tool guide*

1. In the **Stash**, right-click with the mouse on the newly created *Pt_bulk (100)* object and select *Duplicate*. Then, right-click on the newly created *Pt_bulk (100) (1)* object and select *Rename* to change its name to *initial*.
2. Click on the icon to open the **Camera** panel, and orient the system along the XY direction.
3. While holding down the `CTRL` button on your keyboard, use the left mouse button to select a square of four Pt atoms of the Pt(100) topmost layer. Then add an extra atom in between them by clicking on the icon.
4. Select the new Pt atom, and click on *Coordinate Tools* → *Translate* in the **Panel plugins** on the right-hand side of the screen. Define the translation vector as 1.6 in the z-direction to move the atom about 1.6 Å above the surface plane.

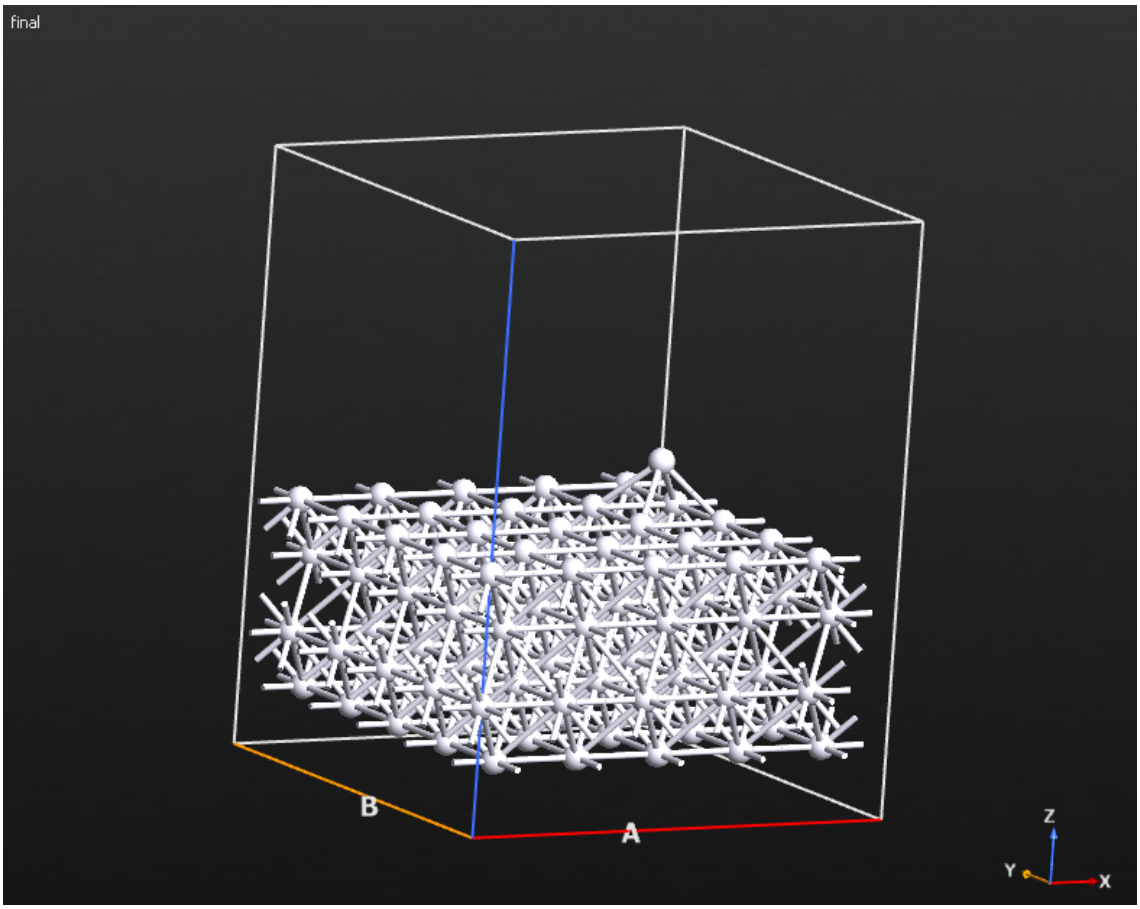
   The structure you have just obtained for a Pt ad-atom on Pt(100) will be used as the initial configuration for the NEB calculations performed in this tutorial.



Create the final configuration for the direct jump process

In this section, you will set up the final configuration for the most direct diffusion process possible for a Pt ad-atom on Pt(100), in which the ad-atom jumps directly between neighboring four-fold hollow sites of the Pt(100) surface.

1. In the **Stash**, duplicate the *initial* configuration and change its name to *final_jump*
2. Select the *final_jump* configuration in the **Stash**, select the Pt ad-atom and click *Coordinate Tools* → *Translate*.
3. In the **Translate Panel**, translate the Pt ad-atom 2.77195 Å in the x or y-direction.
4. In this way, you have obtained the final configuration for the NEB calculation of the direct jump diffusion process.



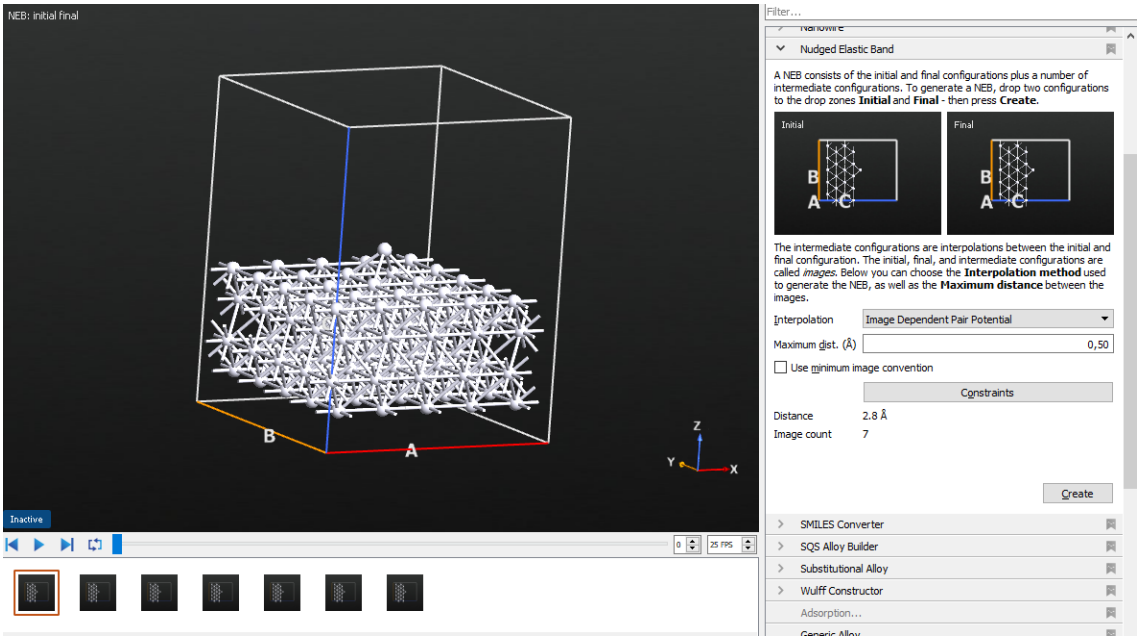
Relax the initial and final configurations

To ensure we get the best possible initial guess for the reaction path between the initial and final states, we need to relax them before constructing it. Use the same calculator as before to relax the two slabs with an ad-atom to a tolerance of *0.01 eV/Å*, and using *FIXED* atom constraints on the bottom two layers of the slab. You can use this workflow to do it: `relax_initial_and_final_WF.hdf5`

Set up and run the NEB calculations

Construct the NEB path

1. In the **Builder**, click on *Builders* → *Nudged Elastic Band* and drag and drop the relaxed *initial* and *final* configurations into the left and right panels.
2. In the **Nudged Elastic Band** panel, set the **Interpolation** to *Image dependent Pair Potential*. Click *Create* to create an object in the **Stash** named *NEB: initial final*, containing the initial and final configurations as well as the interpolated configurations along the reaction path.



Note: The *image dependent pair potential (IDPP)*[1] implemented in **QuantumATK** provides high quality initial guesses for the NEB reaction path compared to the more commonly employed linear interpolation method. For complex reaction paths, the improvement in terms of number of required iterations and speed can be substantial.

Run the NEB calculations

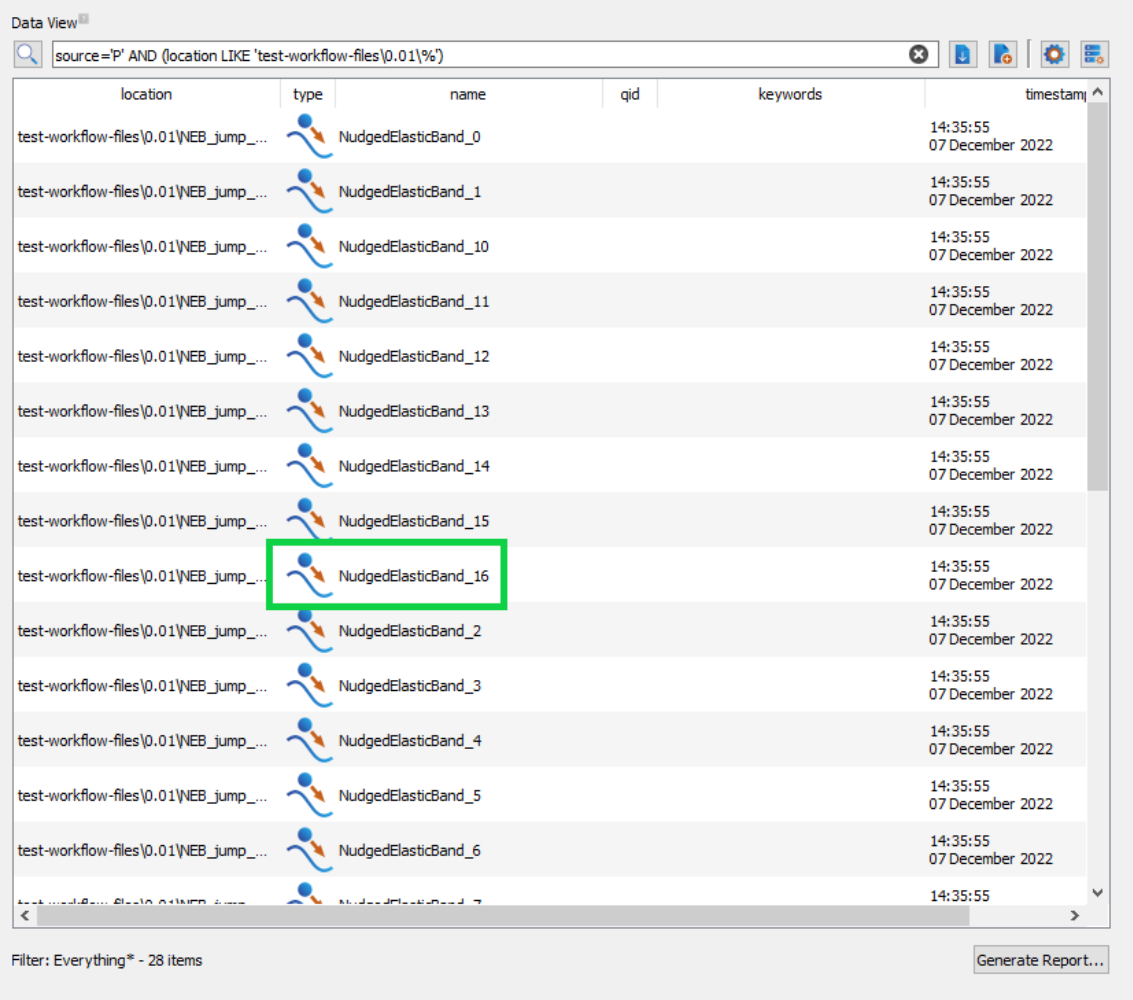
1. Click on the icon and send the *NEB: initial final* configuration to the **Workflow Builder**.
2. In the **Workflow Builder**, change the output **Filename** at the bottom of the middle panel to `NEB_Pt-jump.hdf5`.
3. Add the following blocks to the **Build** panel by double clicking on the corresponding icons in the **QuantumATK** tab in the right-hand panel:
   - Under **Calculators**: *Set ForceFieldCalculator*
   - Under **Optimization and Dynamics**: *OptimizeNudgedElasticBand*
4. Double-click on the *Set ForceFieldCalculator* block and select the *EAM_Pt_2004* parameter set again.
5. Double-click on the *OptimizeNudgedElasticBand* block, set the **Force tolerance** to *0.01 eV/* and set *FIXED* constraints on the bottom two layers.
6. Send the workflow to the **Job Manager** as a script and save the script as `NEB_Pt-jump.py`. This script runs in a few seconds, it can run locally.

   You should now have a workflow identical to this one: `NEB_jump_workflow.hdf5`

[1] S.ăSmidstrup, A.ăPedersen, K.ăStokbro, and H.ăJónsson. Improved initial guess for minimum energy path calculations. *J. Chem. Phys.*, 140:214106, 2014. doi:10.1063/1.4878664.

Analyze the results

Once you have run the NEB calculations for the direct jump diffusion process, many new objects should appear in the **Data View** section of **QuantumATK NanoLab**. There is a `NudgedElasticBand` object for each step in the optimization. They are stored in the file you defined before, here called *NEB_Pt-jump.hdf5*.



1. Select the last NEB trajectory, i.e. the one with the highest number, which contains the optimized NEB path for the direct jump process. Then, double-click on it to open it in the **Movie Tool**.

2. On the left, the **Movie Tool** shows the energy of each image, referenced to the energy of the first image. On the right is the corresponding atomic configuration of each image. You can now see the energy barrier for this diffusion process as the highest point in the plot on the left, and see the corresponding transition state configuration on the right by moving the cursor so the vertical gray line is at the transition state.

Note: As it can be seen in the tutorial at *Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)*, the AKMC method finds this path, among others, without any *a priori* knowledge of the potential energy surface of the system.

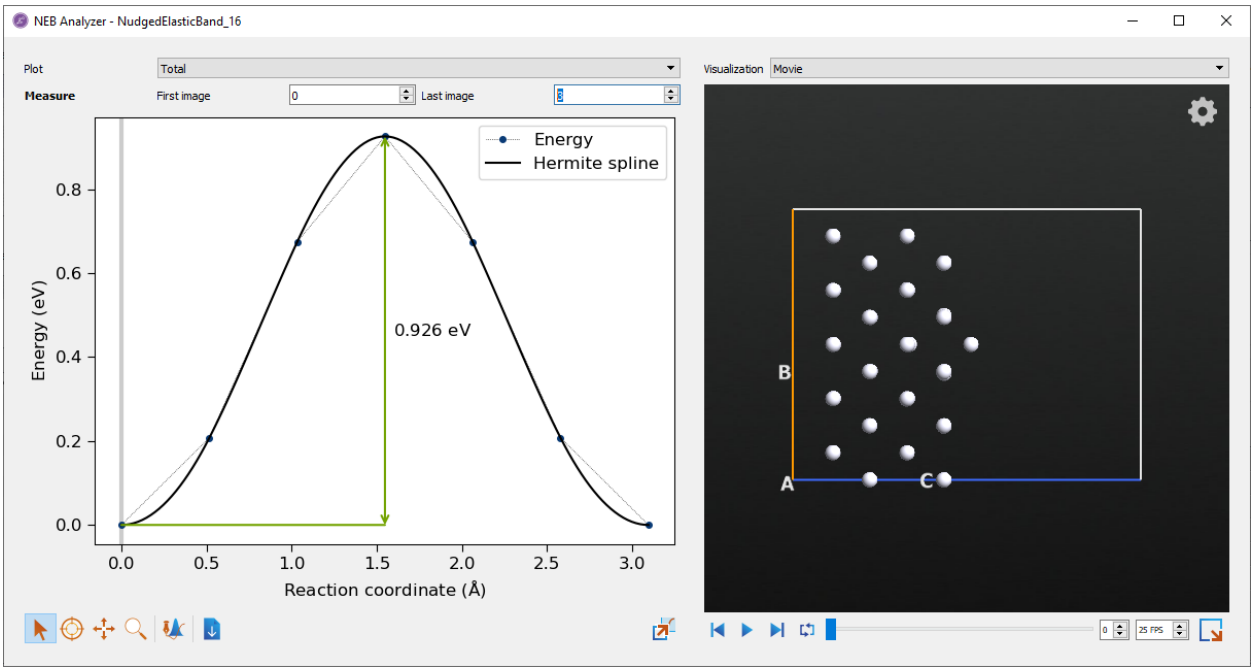


Fig. 3.110: Calculated reaction path for the direct jump diffusion process.

Summary

Throughout this tutorial, you have used a **ForceField** calculator and the **NanoLab** graphical user interface to set up, simulate and analyze the direct jump diffusion path for a Pt ad-atom on a Pt(100) surface. You can also repeat this tutorial with DFT by replacing the **Set ForceFieldCalculator** blocks with **Set LCAOCalculator** blocks, to compare directly with the DFT results reported in the literature[2].

References

3.11.6 Ammonia inversion reaction barrier using DFTB and NEB

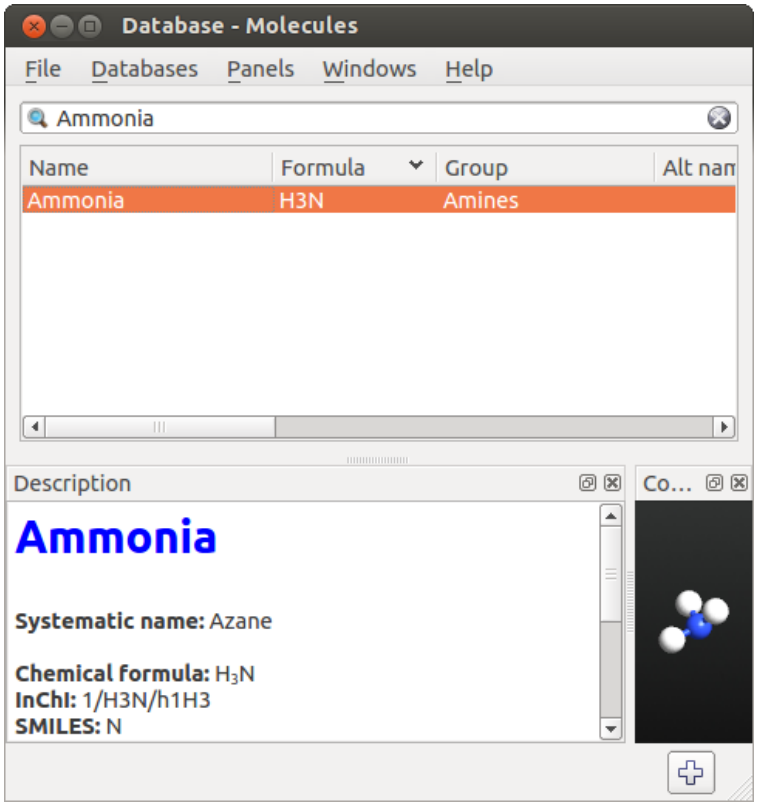
In this tutorial, you learn how to use a Nudged Elestic Band to calculate the reaction path of ammonia inversion. You will use the Density Functional based Tight Binding (DFTB) method (dftb.org) in order to make calculations fast.

Setting up the NEB object

•Open the **Builder** and click *Add* → *From Database*.

•The *Database* tool will open. From the menu, select *Databases* → *Molecules*. Type Ammonia in the search field, and you should see the following:

[2] P.ăJ. Feibelman. Surface diffusion mechanism versus electric field: Pt/Pt(100). *Phys. Rev. B*, 64:125403, 2001. doi:10.1103/PhysRevB.64.125403.

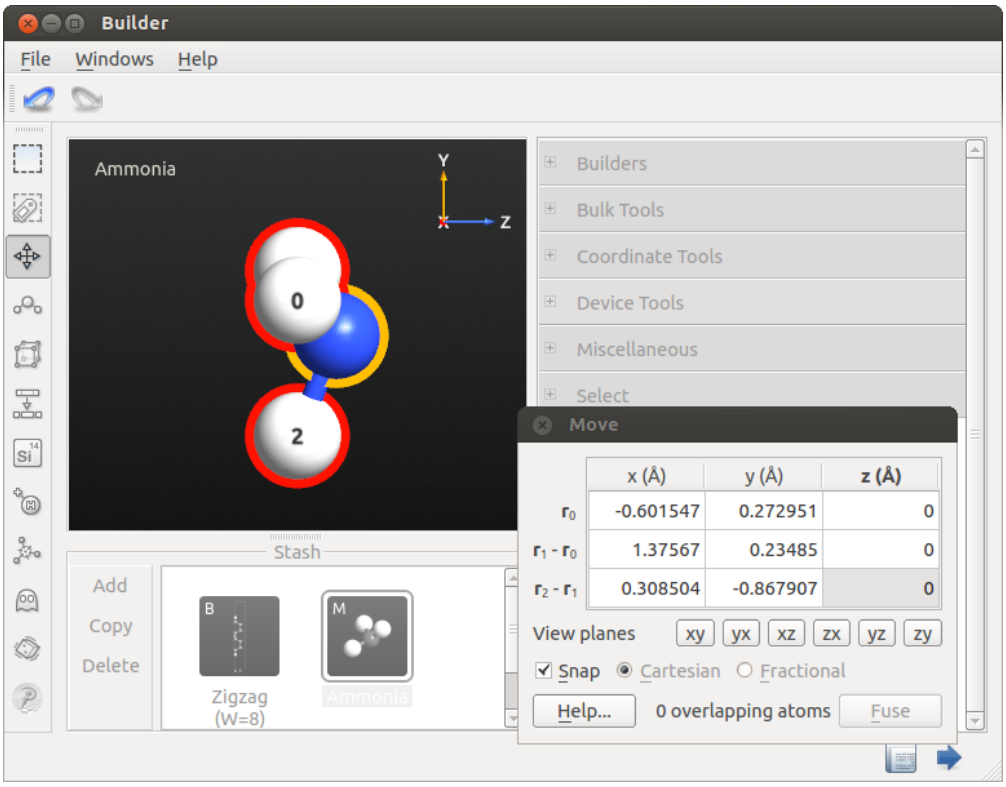


•Double-click the Ammonia entry (or click the icon), and the ammonia molecule will be added to the **Stash**.

Note: When running NEB calculations, it is important to restrict at least a few degrees freedom. What primarily happens in the ammonia reaction considered here, is that the nitrogen atom moves through the hydrogen plane. It is therefore a good idea to keep the hydrogen atom plane fixed when defining the initial reaction path. This can be acheived rather easily, see blow.

•In the **Builder**, activate the **Move Tool** . Select each of the hydrogen atoms as anchor atoms (marked by a red halo); this is the atom you will operate on. Since no atoms were selected when the **Move Tool** was opened, all atoms will simultaneously be selected (marked by a yellow halo); these atoms will be affected by any operation carried out in the **Move Tool**.

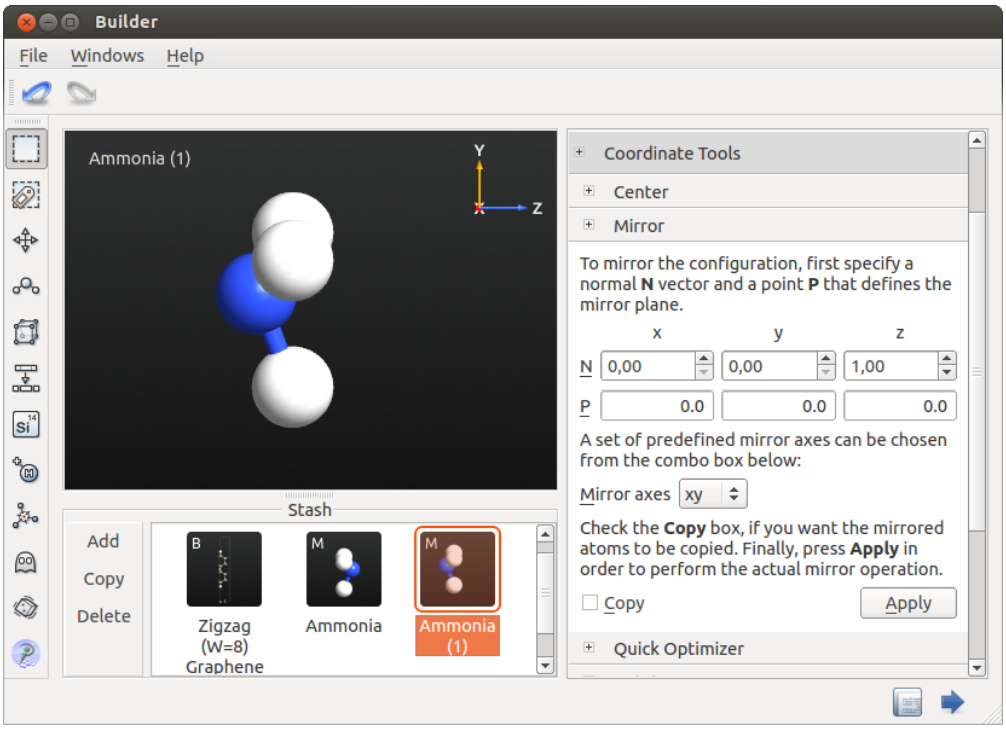
•In the **Move** widget, enter the value 0 for the z coordinate of the atoms, and press **Enter**.



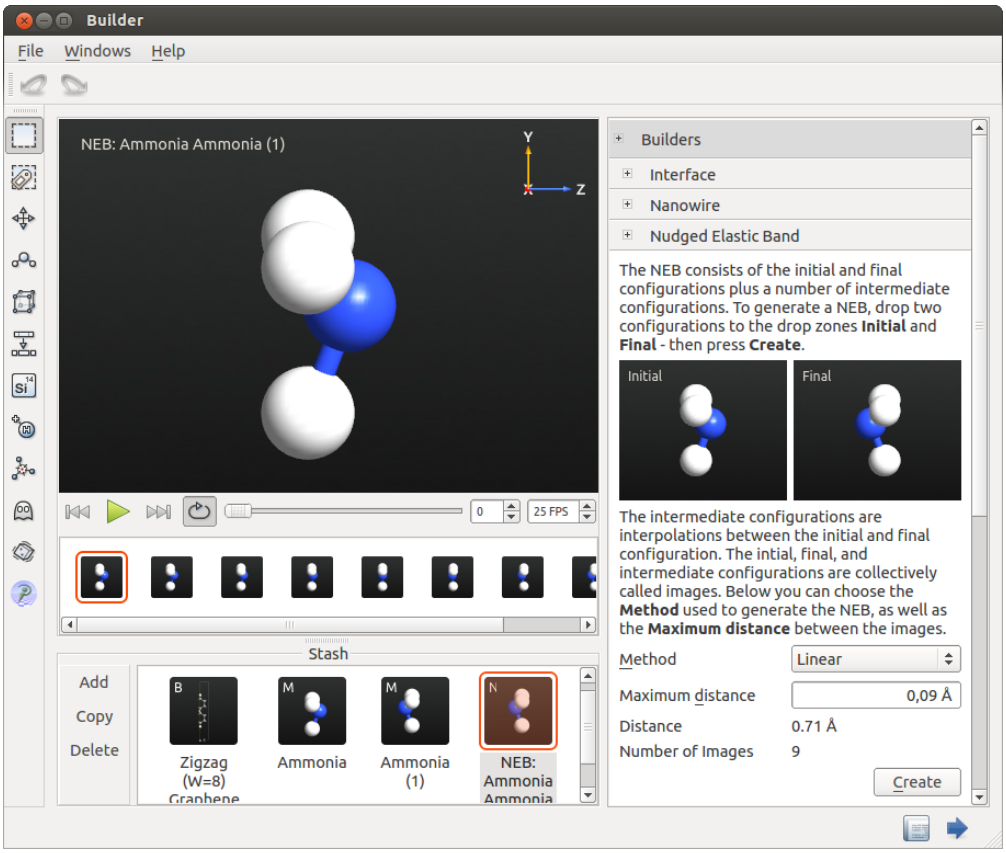
•Close the **Move** tool by closing the widget or clicking the **Select** tool.

•Now ensure that the ammonia molecule is selected on the stash by left-clicking it. Then click the **Copy** button . This adds an additional, identical ammonia molecule to the **Stash**.

•Open the **Mirror** tool from the **Coordinate Tools** in the right tool bar. Ensure that the mirror plane has a normal parallel to the Z axis and contains the origin (this is the default), and press **Apply** to invert the molecule.



- You have now defined the initial and final states of the reaction path. The next step is to set up the NEB configuration by inserting a number of images in-between these states.
- To this end, open the **Nudged Elastic Band** tool under **Builders** from the plugin panel, and drag the two configurations into the *Initial* and *Final* slots.
- Set the maximum distance between images to 0.09 Å in order to have an odd number of images. This is very important, since the reaction path is symmetric. Choose the **Linear** method and press **Create**. The NEB object will now be built and added to the **Stash**.
- Push the **Play** button to see the different configurations that will be used as starting configurations when optimizing the reaction pathway.



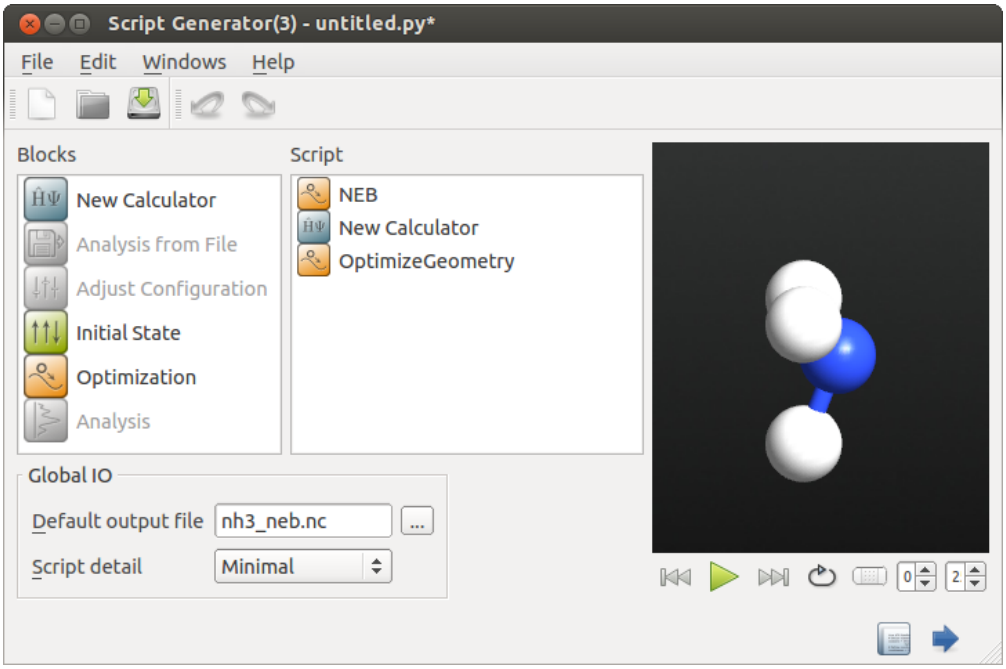
Performing the NEB simulation

To set up the actual calculations, send the NEB object to the **Script Generator** using the **Send To** button .

In the **Script Generator**, add the following blocks:

- **New Calculator**
- **OptimizeGeometry**

and change the default output name to `nh3_neb.nc`.

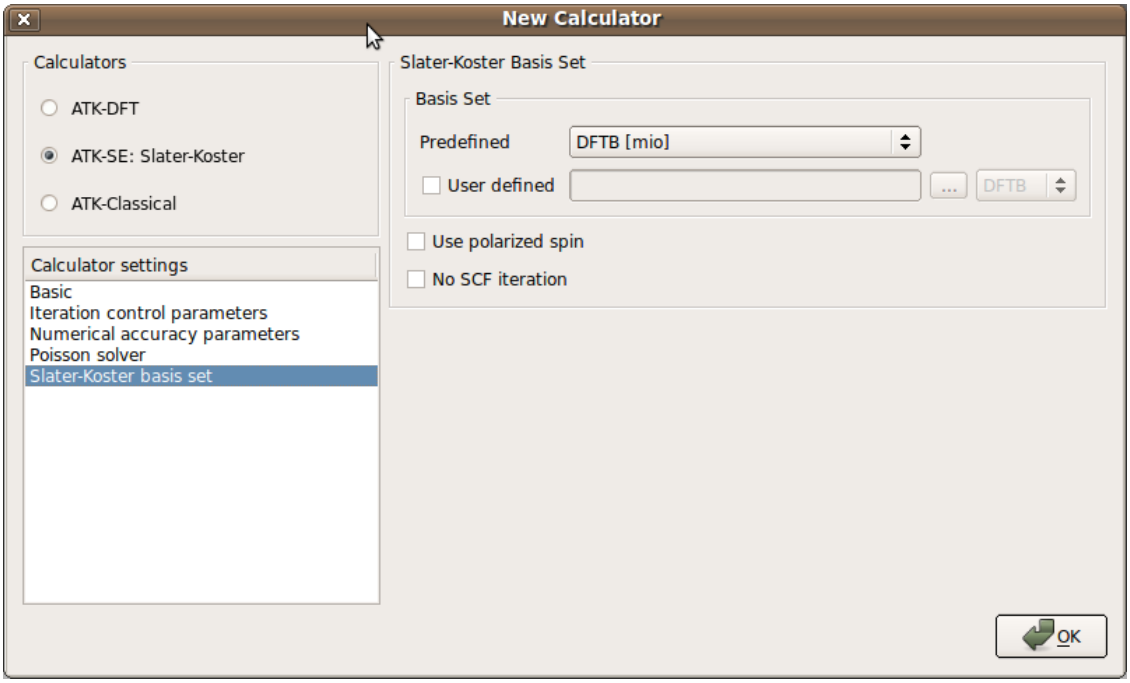


Open the **New Calculator** block and edit it:

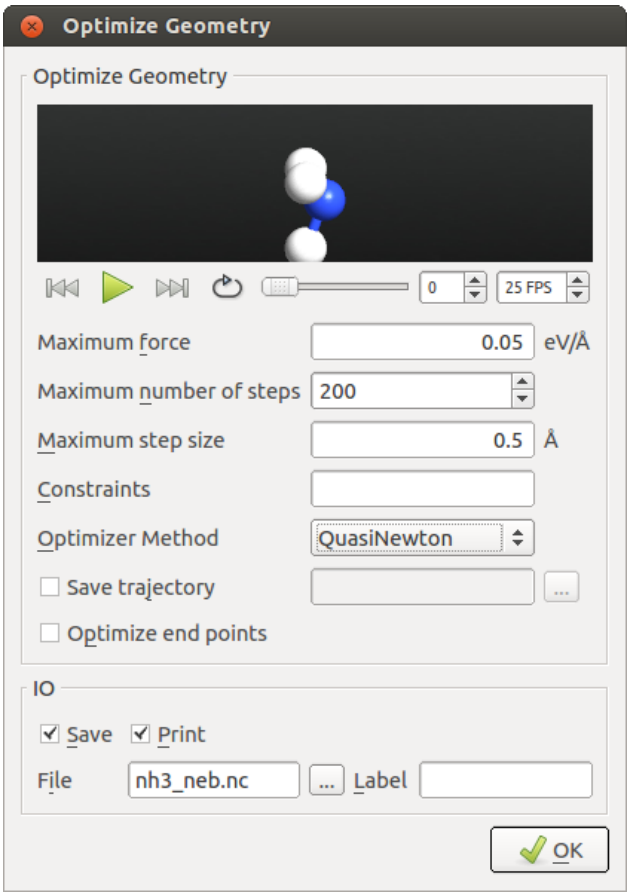
•Select the **ATK-SE: Slater-Koster** calculator.

•Go to the **Slater-Koster basis set** and check that the *DFTB [mio]* basis set is selected.

•Uncheck the **No SCF iteration** box.



Open the **OptimizeGeometry** block, and make sure it looks as shown below.



You are now ready to perform the NEB optimization. Send the script to the **Job Manager** and execute it.

The optimization will take 20-30 minutes depending on your computer.

Note: While most other implementations of the DFTB model use a pointwise electrostatic pair potential interaction model, ATK-SE uses a Poisson solver for calculating the electrostatic interactions. This approach is significantly slower for small systems, like the current ammonia molecule, but faster for large systems, where it scales as $O(N)$ instead of a typical $O(N^2)$ scaling.

The use of a Poisson solver ensures that the DFTB model is equivalent to the other calculators in QuantumATK, and thus can be used for modeling device geometries and allows for using implicit solvent models.

Analyzing the NEB simulation

- Return to the main **QuantumATK** window, and select the file nh3_neb.nc.
- Select the last NEB configuration in the file (the one with the highest ID number).
- Press the **Show configuration** and you will see the calculated energy barrier and the corresponding configurations in the reaction path.
- The reaction path can also be illustrated as an animation by pressing the **Play** button.

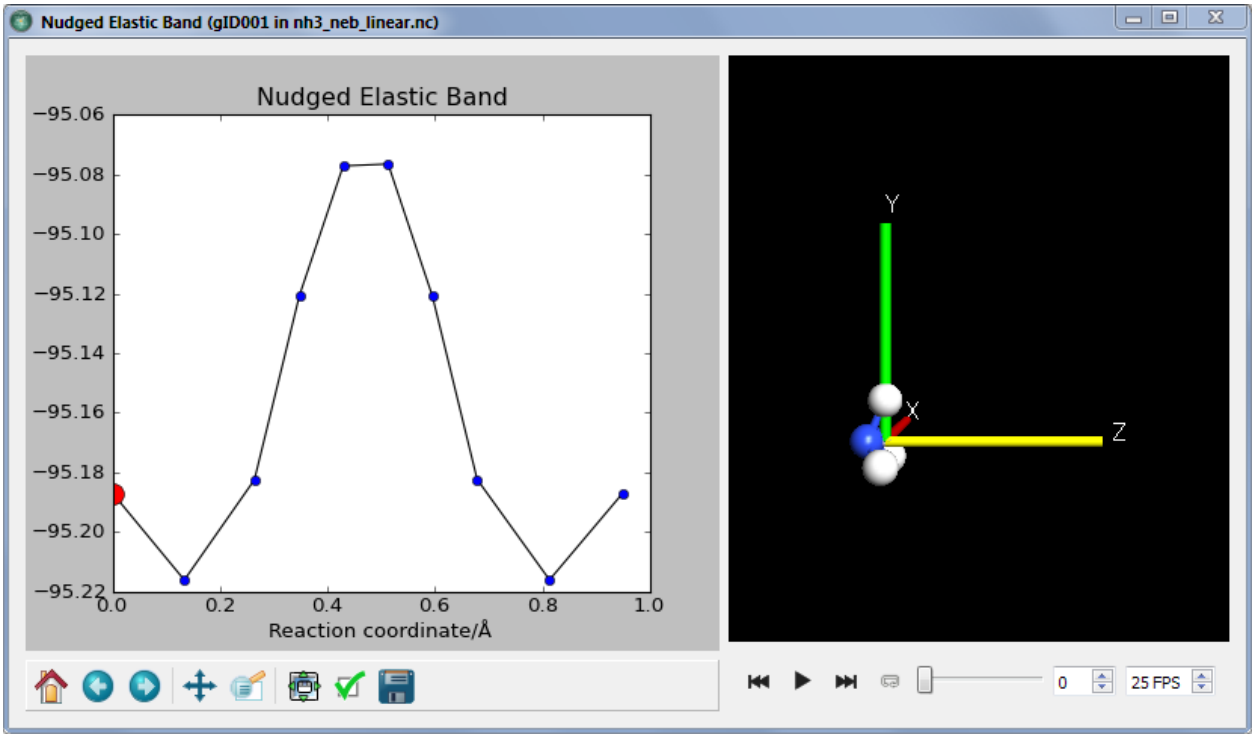


Fig. 3.111: Reaction path of ammonia inversion, computed with the DFTB [mio] method. The end-points were not optimized.

Attention: The obtained reaction barrier is not fully accurate, since the end points were not optimized. This is apparent from the fact that the second image has a lower energy than the end-points. For real studies with the NEB method, the end-points should always be optimized first.

Tip: You can closely inspect an individual optimized configuration in the reaction path by clicking the dots in the reaction paths plot (and stopping the movie).

A recipe for faster calculations

In the calculation above, you used a linear interpolation between the end-points to set up the initial guess for the NEB path. In many cases the linear path is, however, very far from the optimized one. Comparing the movie of the converged result above to the initial NEB path you set up, it can be seen that that the hydrogen atoms need to move out-wards to make room for the nitrogen atom as it passes through the center of the molecule, and then they move back in.

If this behavior could somehow be part of the initial path, one would save some steps in the ionic dynamics, and hence computational time. However, since the initial and final positions of the hydorgen atoms are identical, this cannot be captured by the linear interpolation.

There are, however, other methods for generating the initial path, and some are available in **ATK**.

•Go back to the **Builder**, and the **Nudged Elastic Band** plugin.

•Keep the image distance, but change the **Method** to **Image Dependent Pair Potential**, and create the NEB.

- If you now inspect the initial path, you can see that it already contains this behavior of the hydrogen atoms. This is one of the strengths of the IDPP method[1] for NEB calculations.
- Follow the same procedure as above to run the NEB calculation via the **Scripter**, and you will find that it runs about 30% faster this time, but that results are identical.

Note: The IDPP method is similar to the Halgren–Lipscomb method[2], which is also available in **ATK** (but not suitable for this particular system).

References

3.11.7 Reconstruction of the Si (100) surface - a geometry optimization study with QuantumATK

Introduction

Although LDA and GGA may not be able to produce a proper band gap in Si (and most other semiconductors), these simple DFT functionals are very capable when it comes to predicting geometrical properties. In this short tutorial we will demonstrate how to use QuantumATK to study the so-called asymmetric dimer reconstruction of a Si (100) surface.

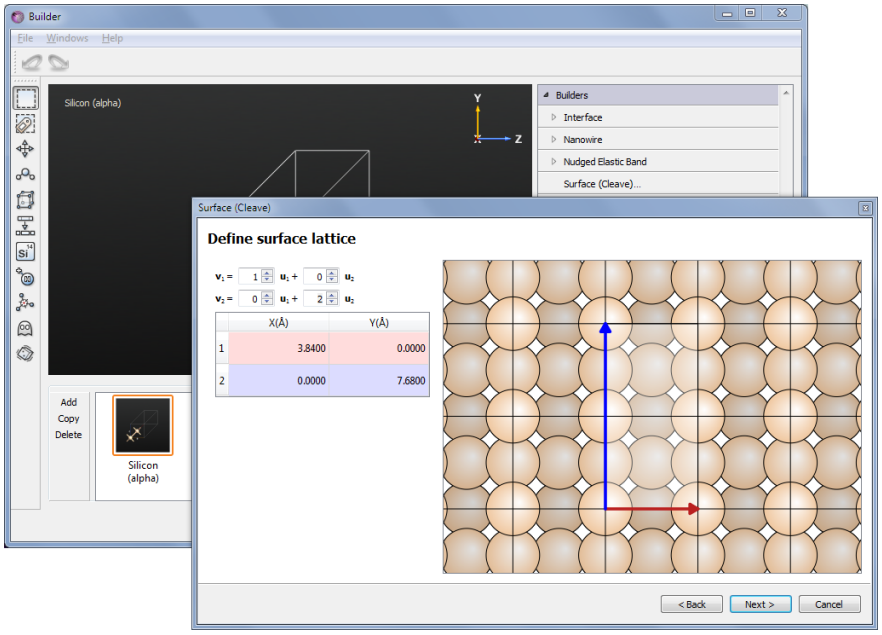
The focus will be on the physics and how to set up the model correctly, rather than how to operate QuantumATK. It will be assumed that you have experience building structures (for instance, cleaving surfaces) and setting up calculations and inspecting the results, and the steps will therefore only be indicated, not explained in any great detail.

Building the geometry

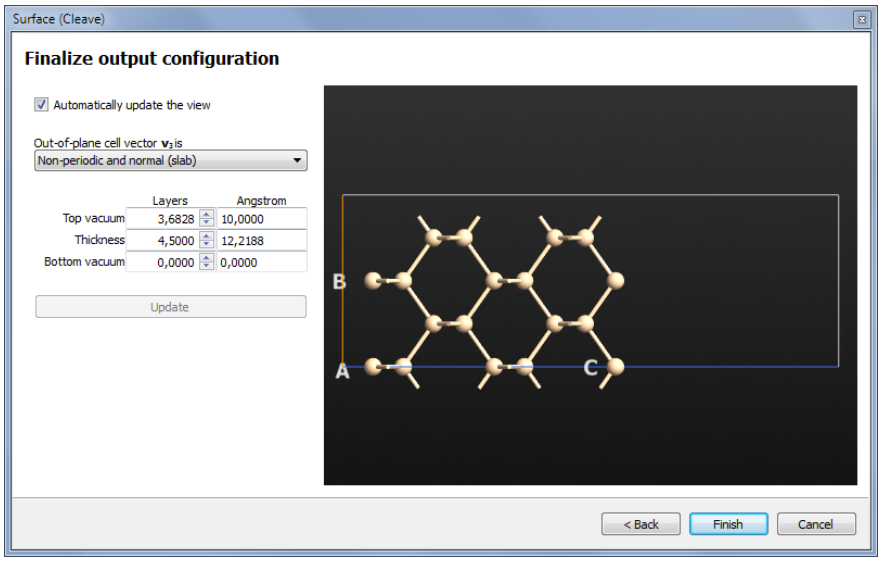
1. Start QuantumATK and open the **Builder**.
2. Add Silicon (alpha) from the Database (*Add* → *From Database*).
3. Open *Builders* → *Surface (Cleave)*.
   - Cleave the structure along 100 by entering the corresponding Miller indices.
   - Since the dimer reconstruction can only occur in a 2x1 supercell, you need to make a larger surface cell than the smallest one. Therefore, on the second page set $\mathbf{v_2} = 2\mathbf{u_2}$.

[1] S. Smidstrup *et al.*, *Improved initial guess for minimum energy path calculations*, J. Chem. Phys. **140**, 214106 (2014) link

[2] T. A. Halgren and W. N. Lipscomb, Chem. Phys. Lett. **49**, 225 (1977)



•On the final page of the Surface tool, choose a Slab configuration with thickness 4.5 layers (the default size of the vacuum is fine).

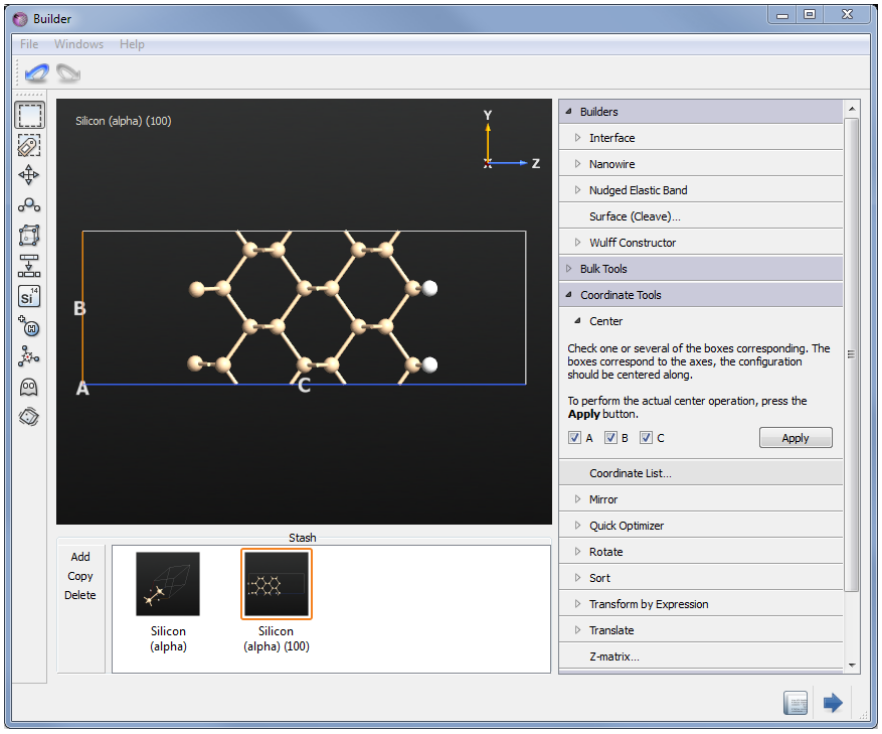


4.Center the structure in all directions (*Coordinate Tools → Center*).

5.Select the two Si atoms with the *smallest* Z-coordinate and add a small random perturbation to their positions by clicking the `Rattle` tool a few times. This is because the completely symmetric state is a meta-stable configuration, and its best to not start the optimization in such a state.

6.Select the two Si atoms with the *highest* Z-coordinate and passivate them to saturate all Si bonds in the structure.

The resulting structure is shown in the picture below.

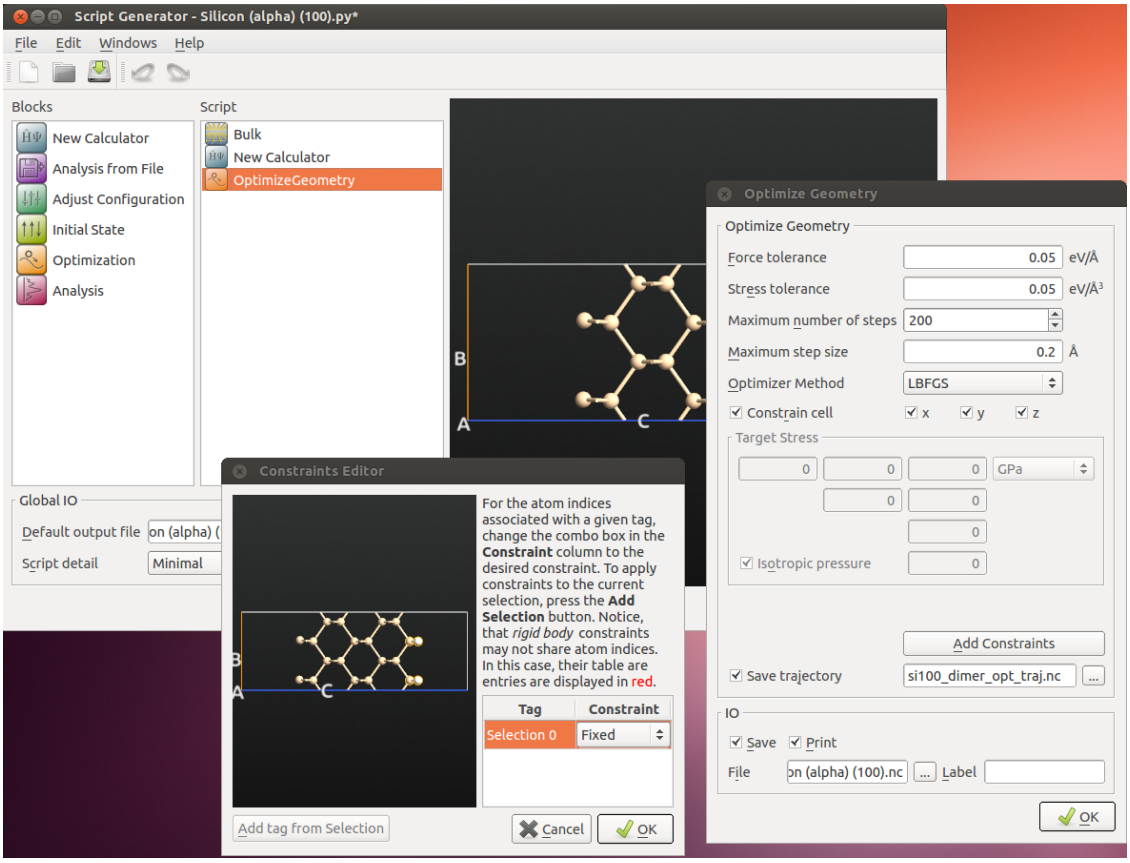


Setting up the calculation

Send the structure to the **ScriptGenerator**.

1. Insert a **New Calculator** block, then an **OptimizeGeometry** block.
2. For the calculator you can keep all parameters at their default except the **k-point sampling**. Set it to 9x9 (of course 1 point is enough in the C direction, since its a slab) - accuracy is important since the difference in energy between the symmetric and asymmetric dimer states is very small[1].
3. For the optimization, constrain the two bottom Si atoms and the 4 hydrogens. It is always important to limit the degrees of freedom in a relaxation if possible, to project out pure translations and rotations.
4. Also, make sure to tick `Save trajectory` and set a name for the trajectory file.

[1] A. Ramstad, G. Brocks, and P.J. Kelly: Theoretical study of the Si(100) surface reconstruction. Phys. Rev. B 51, 14504 (1995)



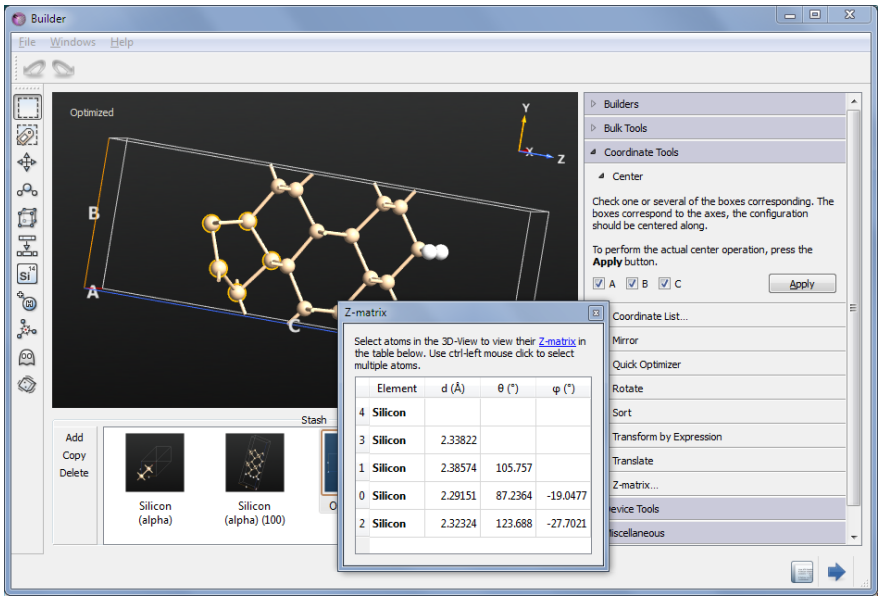
5. Set the name of the output file, and the calculation is ready to be run. Save the script!

   Send the script to the Job Manager, or transfer it to a cluster and run it there. This calculation parallelizes very well, and takes between one and several hours depending on how many MPI nodes it is run on.

Results

When the calculation is done, select the HDF5 file in QuantumATK and inspect the second Bulk configuration, with Id gID001 - this is the optimized geometry.

Drag-and-Drop it on the **Builder** - it is immediately visible that indeed an asymmetric dimer state has been obtained. To display the bond lengths just hoover the mouse over the respective bond - the bond distance will be displayed in the tool tip. To perform measurements of the bond lengths and angles, open *Coordinate Tools* → *Z-Matrix*, and select various atoms in the surface - both for the original and optimized geometry - and compare.



It is also interesting to analyze the relaxation trajectory. Select the LabFloor object in which you have the saved the trajectory, and open it in the **Viewer** to watch a movie of the optimization.

If we plot the total energy and forces for each relaxation step, we see that the total energy is more or less constantly decreasing, as expected from the fact that we use the *LBFGS* algorithm, which indeed is designed to minimize the total energy rather than just following the forces. The following simple script reads the trajectory data and plots the total energy and maximum force for each step:

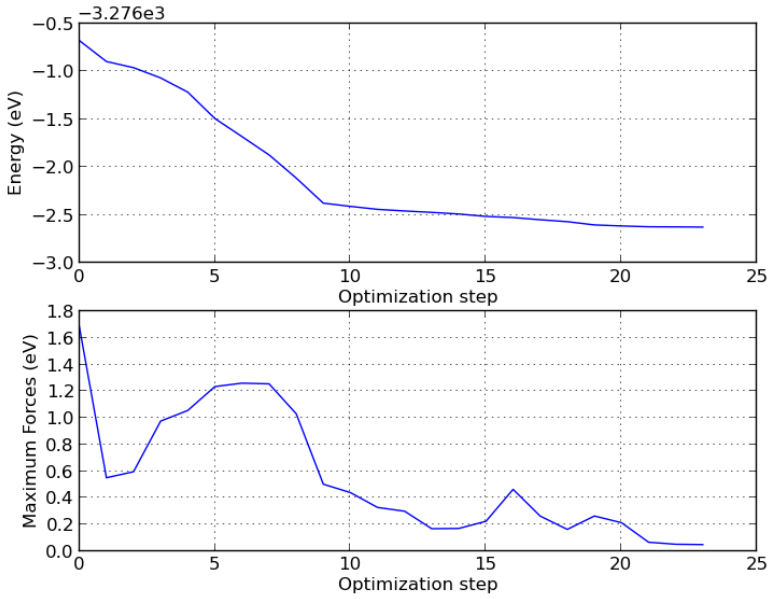
```
traj = nlread('si_100_traj.hdf5', Trajectory)[0]
energies = []
max_forces = []
for i in range(len(traj)):
    energies.append(traj.imageEnergy(i).inUnitsOf(eV))
    forces = traj.imageForces(i).inUnitsOf(eV/Angstrom)
    force_magnitude = (forces**2).sum(axis=1)**0.5
    max_forces.append(numpy.max(forces))

# Plot the energy and max. force curve using pylab.
import pylab

pylab.figure()
pylab.subplot(2, 1, 1)
pylab.plot(range(len(traj)), energies)
pylab.xlabel('Optimization step')
pylab.ylabel('Energy (eV)')
pylab.grid(True)

pylab.subplot(2, 1, 2)
pylab.plot(range(len(traj)), max_forces)
pylab.xlabel('Optimization step')
pylab.ylabel('Maximum Forces (eV/Ang)')
pylab.grid(True)
```

(continues on next page)

(continued from previous page)

```
pylab.show()
```

The result may look like the figure below, although the exact curve depends on the exact initial displacement of the top silicon atoms.



From this plot, combined with the trajectory movie, you can learn many things. First of all, note that unless you use a criterion for the forces which is below 0.2 eV/Å, you would be fooled into thinking the symmetric dimer is the minimum, since this state is passed on the way to the asymmetric one! This is also possible to see in the trajectory movie. In fact, although it has not been shown here, it is well known that unless care is taken to have an accurate method (i.e. enough k-points) and even enough vacuum around the slab, the symmetric dimer will be predicted to be the most energetically favorable.

Specifically, the symmetric dimer local minimum is reached around step 13 (see the trajectory picture above), but as the algorithm tries to reduce the total energy further, the forces rise again and the asymmetry starts to appear. After step 15 an asymmetry has been established for real, and the asymmetric dimer is finally formed at around step 20.

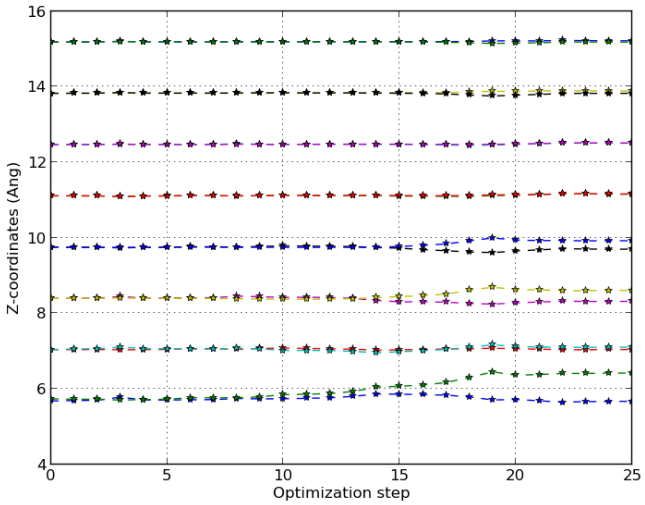
You can use the following script to visualize the vertical relaxation of the atomic layers:

```
traj = nlread('si_100_traj.hdf5', Trajectory)[0]
number_of_steps = len(traj)
z_coordinates = []

for i in range(number_of_steps):
    coordinates = traj.image(i).cartesianCoordinates().inUnitsOf(Angstrom)
    # Append all z-coordinates, apart from the fixed bottom layer.
    z_coordinates.append(coordinates[:16, 2].tolist())

# Convert to numpy array.
z_coordinates = numpy.array(z_coordinates)
# Plot the z-coordinates using pylab.
import pylab
```

(continues on next page)

(continued from previous page)

```
pylab.figure()
# Loop over all unconstrained atoms.
for i in range(16):
    pylab.plot(range(number_of_steps), z_coordinates[:, i], '--*')
pylab.xlabel('Optimization step')
pylab.ylabel('Z-coordinates (Ang)')
pylab.grid(True)

pylab.show()
```



It is important to note that the asymmetry appears as a result of a long-range interaction, which also is why short-range methods like tight-binding or classical potentials are only able to relax the configuration into the symmetric dimer.

Summary

This study has shown how QuantumATK can succesfully predict even a rather complicated geometric reconstruction by using geometry optimization in DFT. The Si 100 surface first forms a symmetric dimer which in turns induces an asymmetry in the electron landscape, under which we again optimize the structure to find the ground state asymmetric structure. The asymmetry is a long-range effect which originates a few layers below the surface where the Si lattice is compressed downwards under the dimer.

References

3.11.8 Computing the work function of a metal surface using ghost atoms

Version: 2017

In this tutorial you will learn how to compute the work function of a metallic surface using **ATK**. You will create a slab model of Ag (100) surface, use ghost atoms to improve the description of its electronic properties, and use the appropriate boundary conditions to describe correctly the presence of vacuum above it. It is assumed that you have some experience with **ATK**.

Why use ghost atoms?

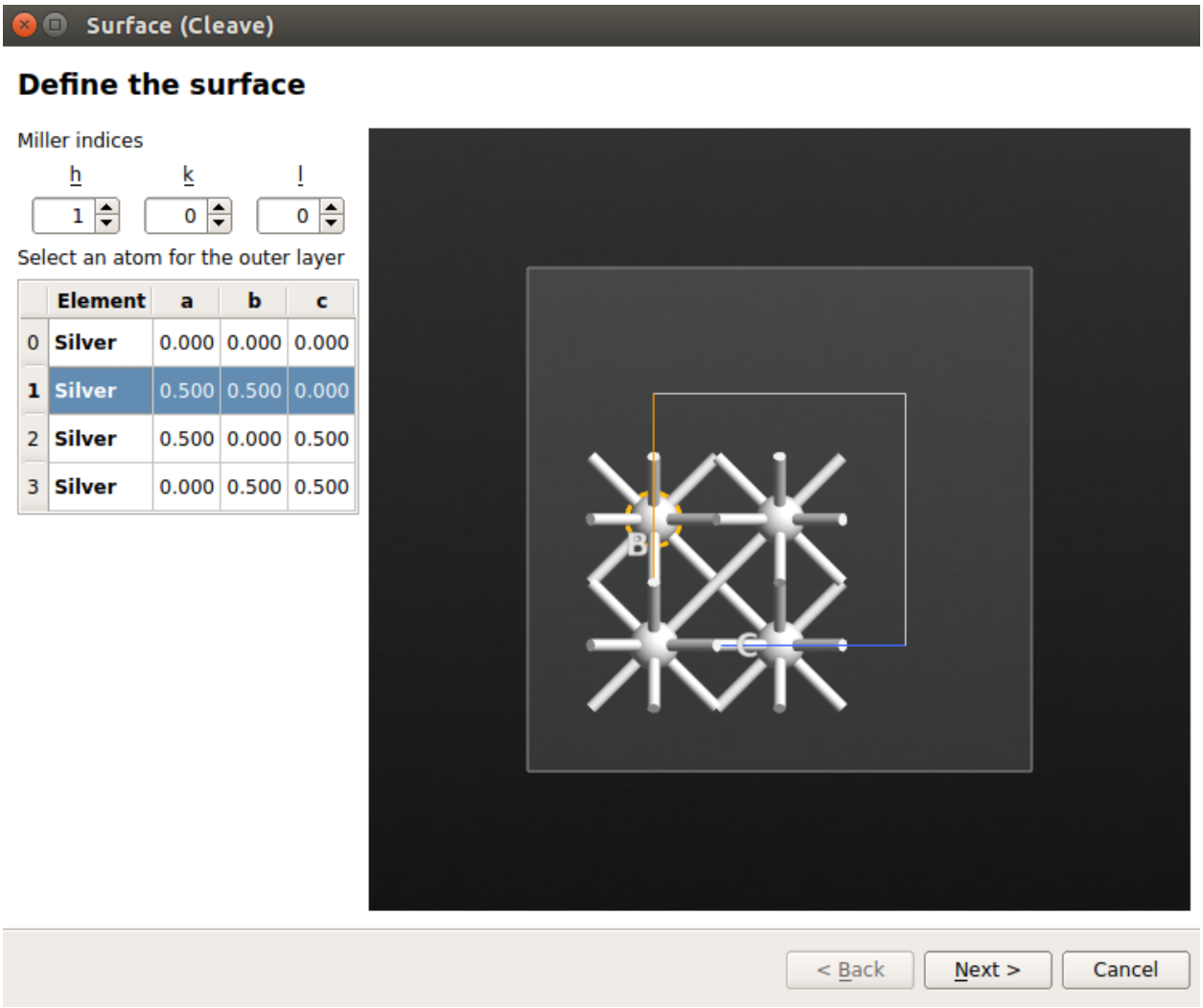
Work function calculations depend critically on accurately simulating the decay of the surface charge density into vacuum, but short-ranged localized basis sets are sometimes not adequate for this. In this case, extra LCAO basis functions are positioned above the surface – these are called ghost atoms.

Setting up the geometry

In this section, you will set up a slab model of Ag(100) surface with ghost atoms, and calculate the work function of the surface.

1. Open **Builder** (icon). Click *Add* → *Add From Database*, find *silver*, and add it to **Stash**.

2. Open *Builders* → *Surface (Cleave)*. Use **Surface (Cleave)** widget to define the surface by cleaving the bulk silver normal to the [100] direction.

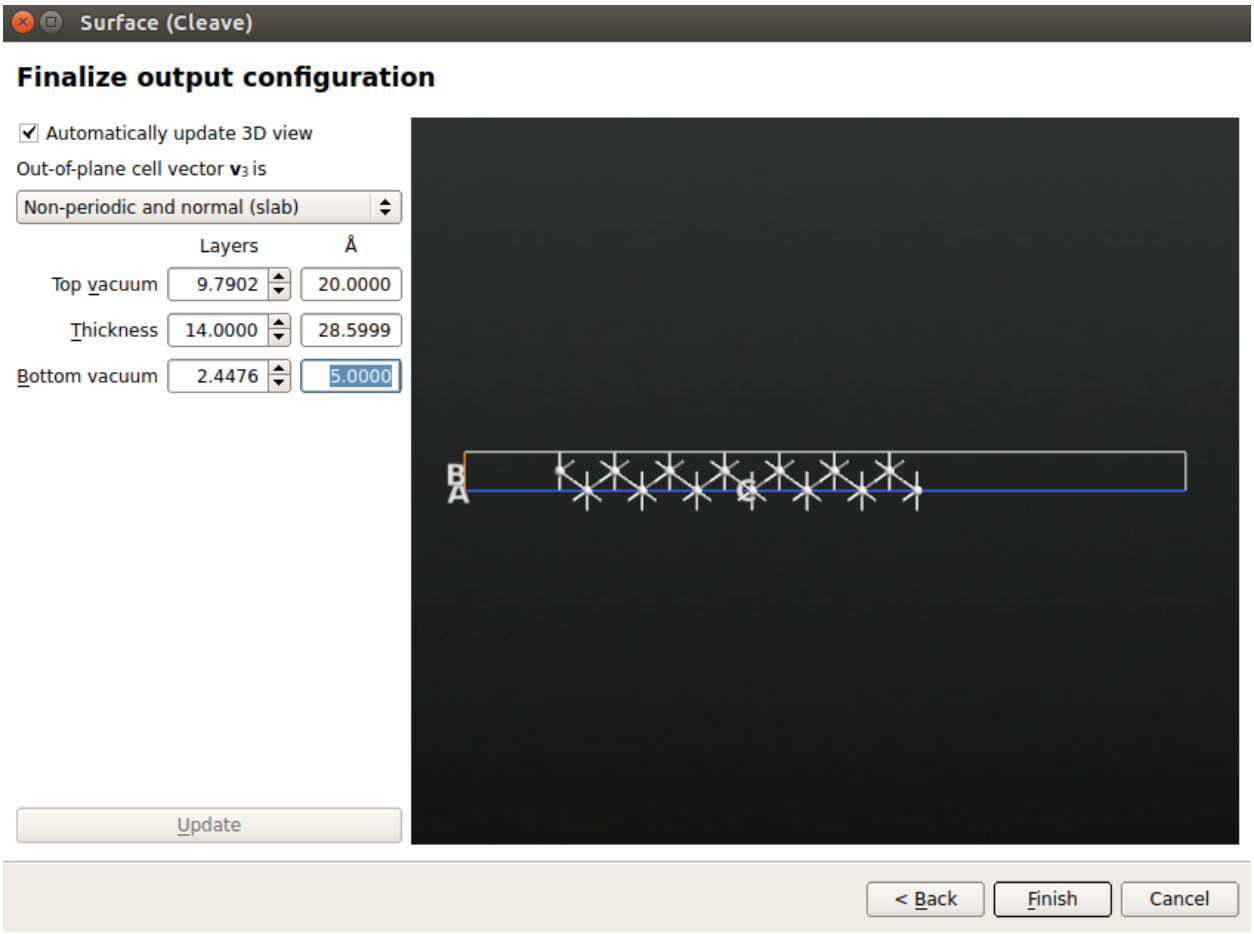


3. Click Next twice. On the final page of the **Surface (Cleave)** widget, set the out-of-plane cell vector $\mathbf{v}_3$ to be *Non-periodic and normal*. Set the rest of the parameters as follows:

- Top vacuum = 20 Å. This large vacuum above the surface ensures that the effective potential has enough space to decay to the vacuum level.

•Thickness = 14 layers.

•Bottom vacuum = 5 Å. This vacuum ensures that no basis functions extend to the bottom edge of the cell (the radius of the basis functions for default silver is 3.7 Å).



Attention: This parameter setting ensures the convergence of work function.

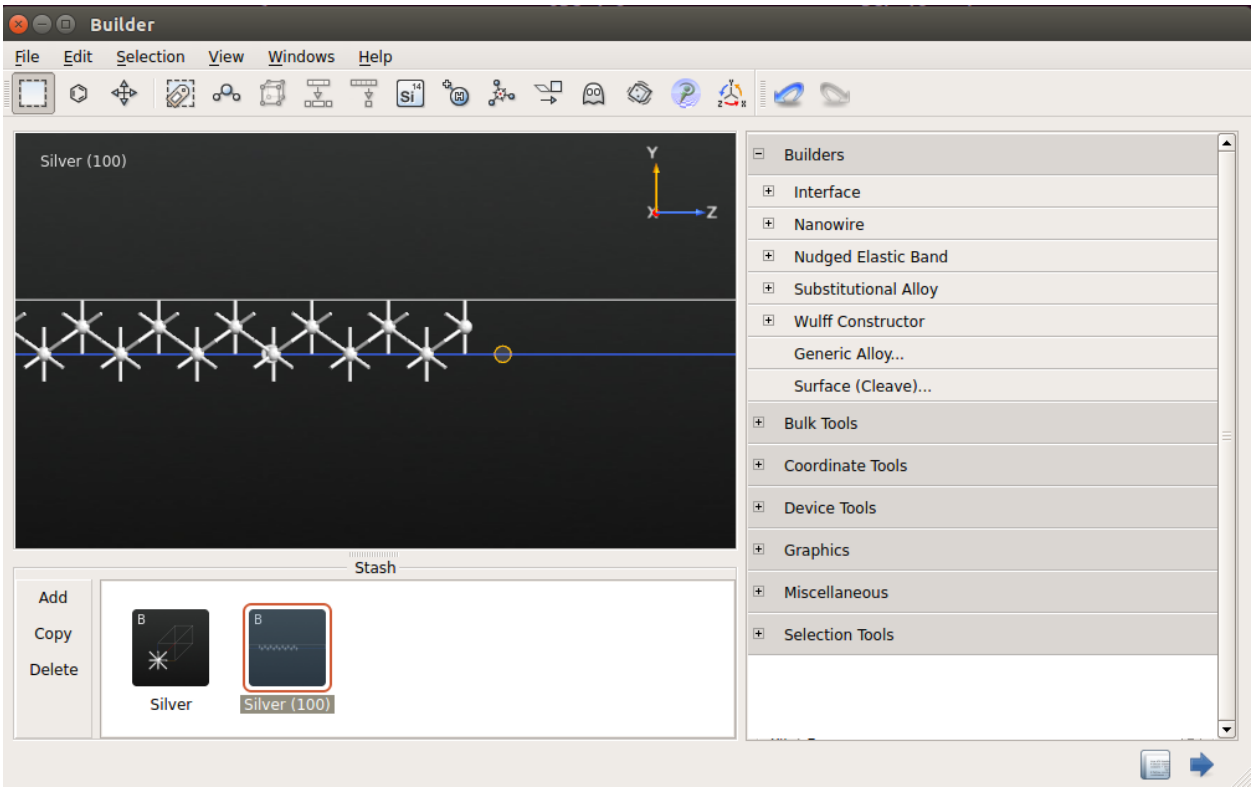
Warning: In the direction normal to the surface, the non-periodicity of the system requires appropriate boundary conditions.

4.Click *Finish*. The surface structure will now appear in **Stash**.

5.To finalize the geometry setup, select the atom with the largest Z coordinate in **Builder** and turn it into a ghost atom by clicking the icon .

6.As a result, the slab consists of 13 layers. The role of the ghost atom on the surface is to describe the smooth decay of electron density in the vacuum.

7.Send the structure to **Script Generator** (icon).



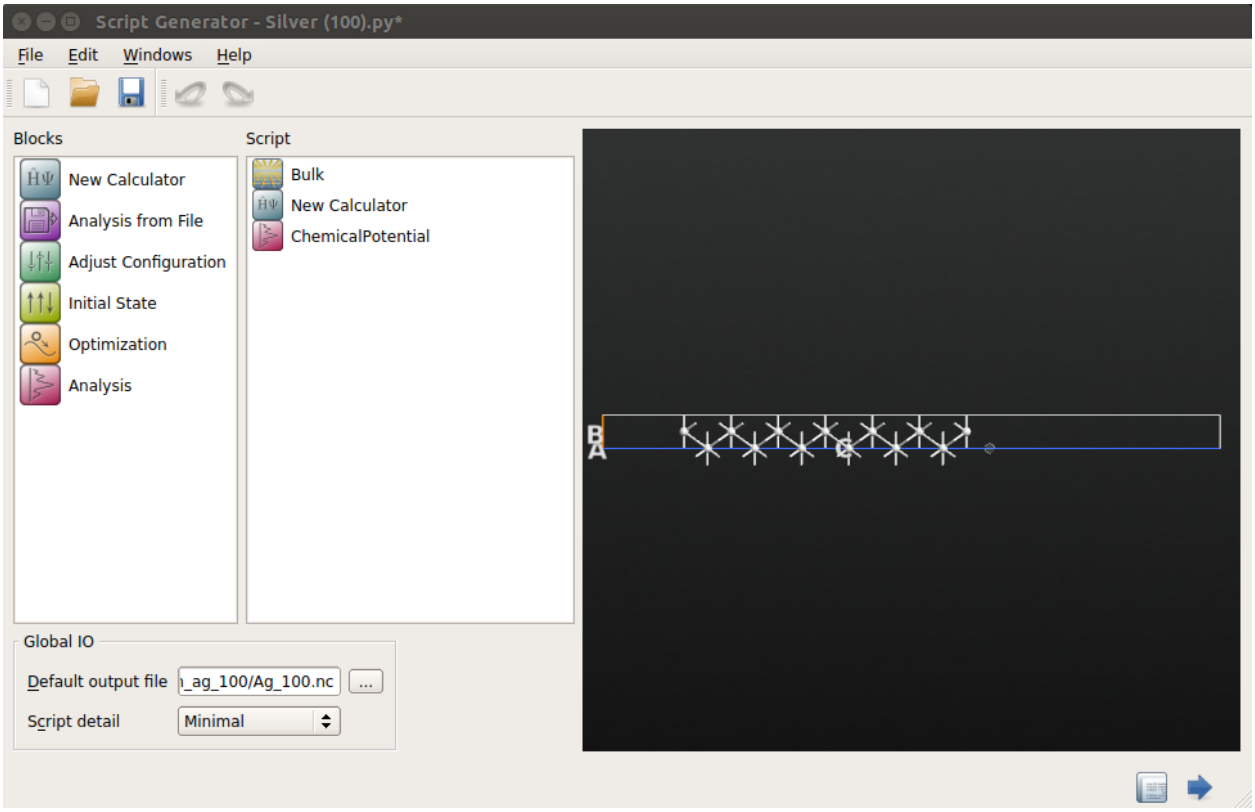
Note: Changing an atom into a ghost atom means setting its potential, e.g., pseudopotential or nuclear charge there, to be zero. The basis functions are, however, still deployed there to describe the electron density. An alternative is to extend the range of the basis functions at every atomic position, although it makes the calculation expensive if the system is not as simple as in this example.

Defining the parameters of the calculation

1. In **Script Generator**, insert a **New Calculator** by double-clicking it in **Blocks** panel. Also, click *Analysis* → *ChemicalPotential* to compute the chemical potential of the slab.

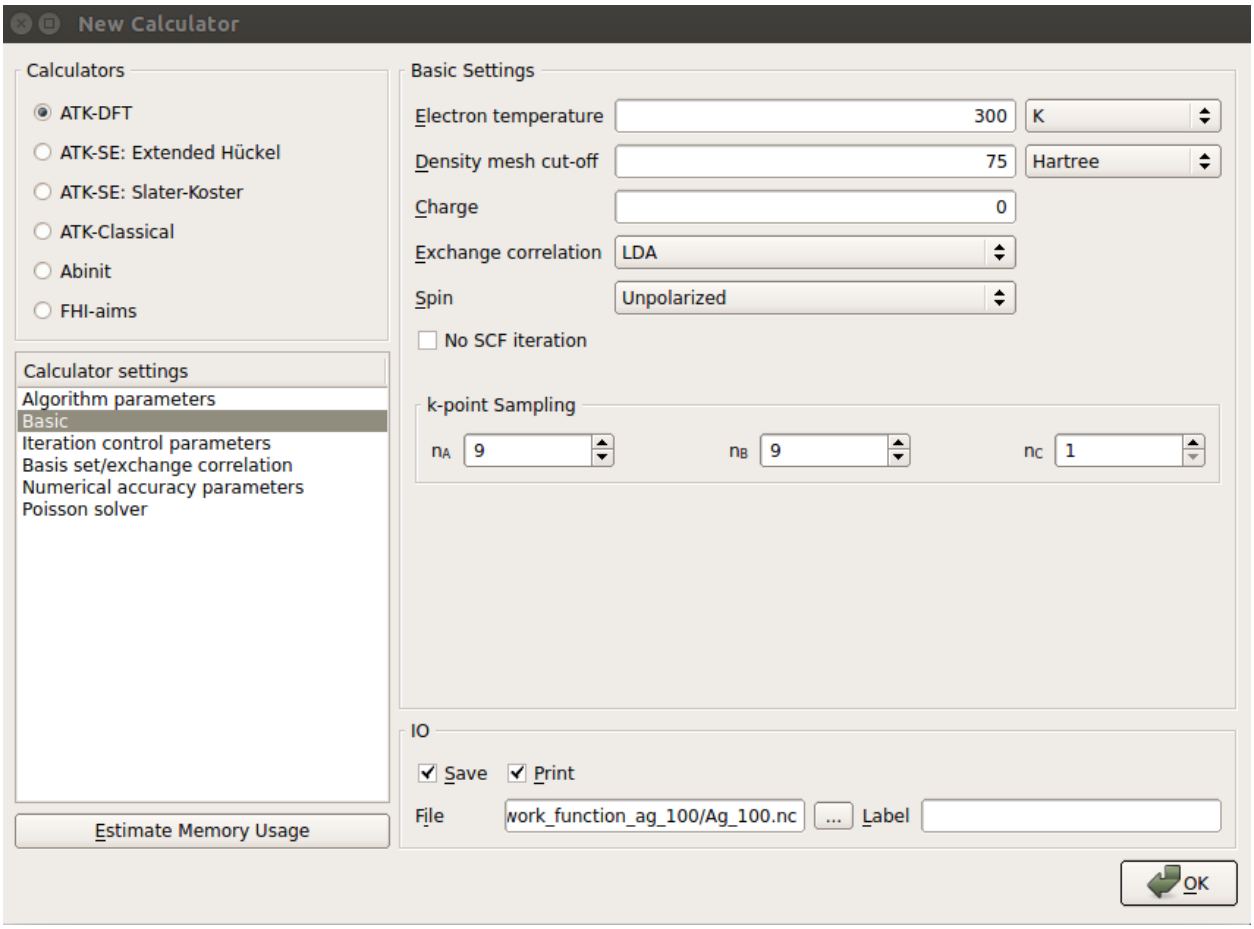
Note: Imposing appropriate boundary conditions at the top and bottom of the slab as shown below makes the chemical potential equal to the work function.

2. Set a suitable file name for the HDF5 file, e.g., *Ag_100.hdf5*.



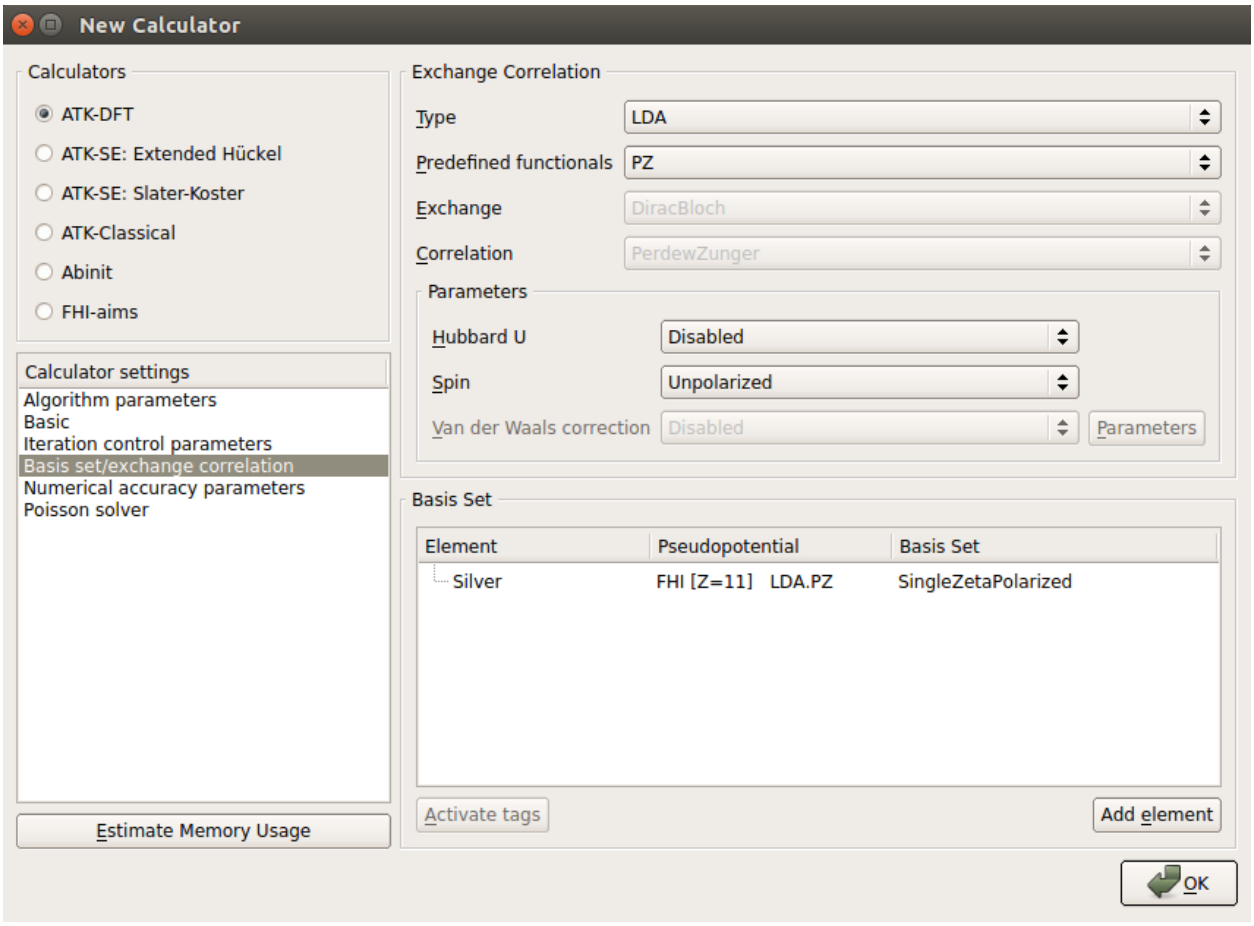
3.Double-click the **New Calculator** in **Script** panel and set the following parameters:

•For accurate description of the electronic structure, set the k-point sampling to be 9x9x1 in *Basic* section of **Calculator settings**



Note: Remember in general to check the convergence of the properties of interest with respect to the fineness of the k-point grid.

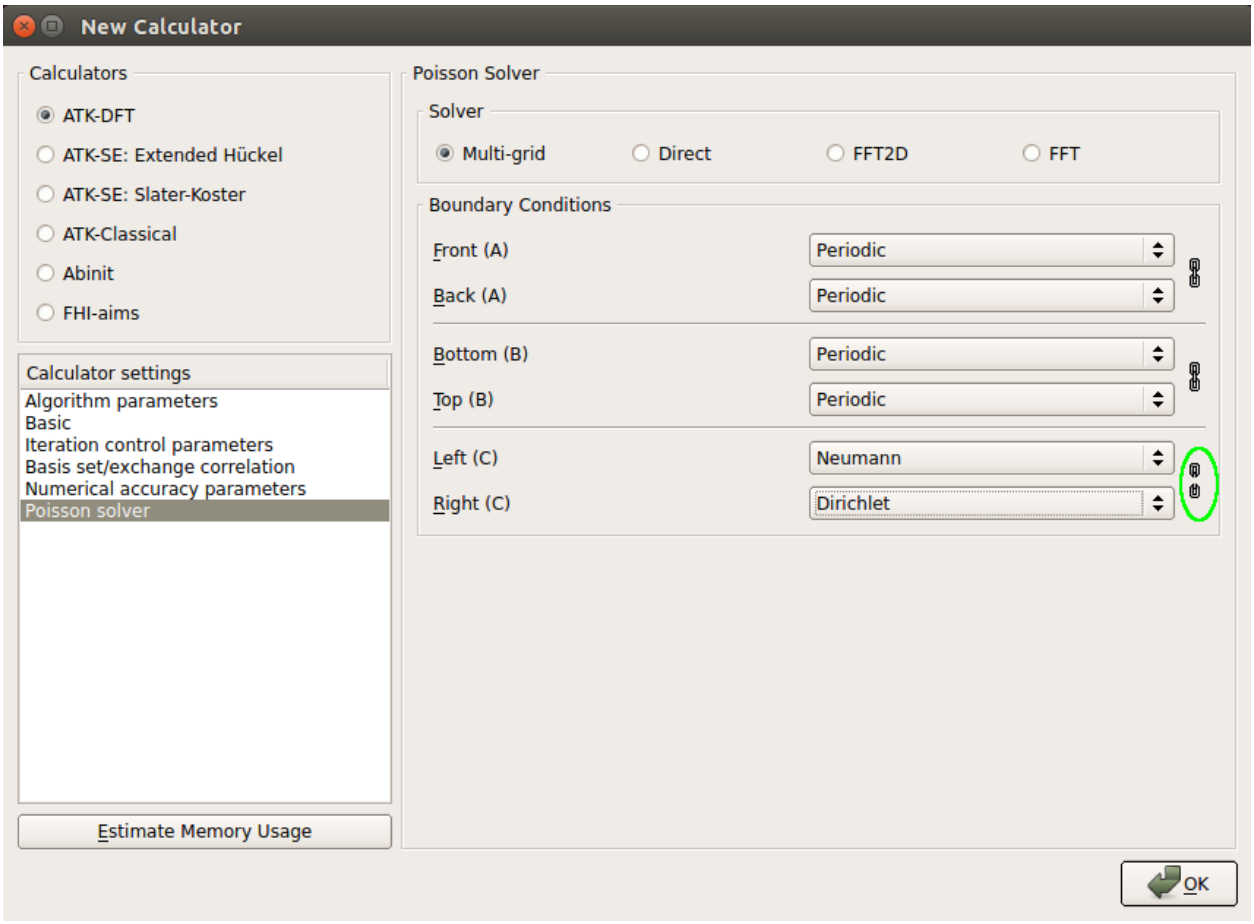
•In *Basis sets/Exchange correlation* section, select *SingleZetaPolarized* for the basis set.



Note: Employing this basis set achieves good accuracy for this slab model with moderate computational effort. For other systems, however, it is not ensured that the use of the same basis set always provides enough accuracy.

Warning: Generally it is not advisable to use the TB09-MGGA to compute the work function, because the TB09-MGGA potential diverges in the vacuum. For the use of TB09-MGGA, the c-parameter should be tuned to obtain a reasonable value of the work function, and correct electronic structure of the material. In the present case, setting c = 0.94 gives WF = 4.44 eV, in excellent agreement with the LDA result (see below). Note that the value of c differs from that calculated self-consistently for bulk Ag, c = 1.31.

•Finally, the most important setting: in *Poisson solver* section, select the Multi-grid Poisson solver. Then, click chain icon (marked in green in the picture below) to unlink the settings of the Left and Right cell boudaries, and set the Neumann boundary condition on the Left boundary, and the Dirichlet boundary condition on the Right cell boundary, respectively.

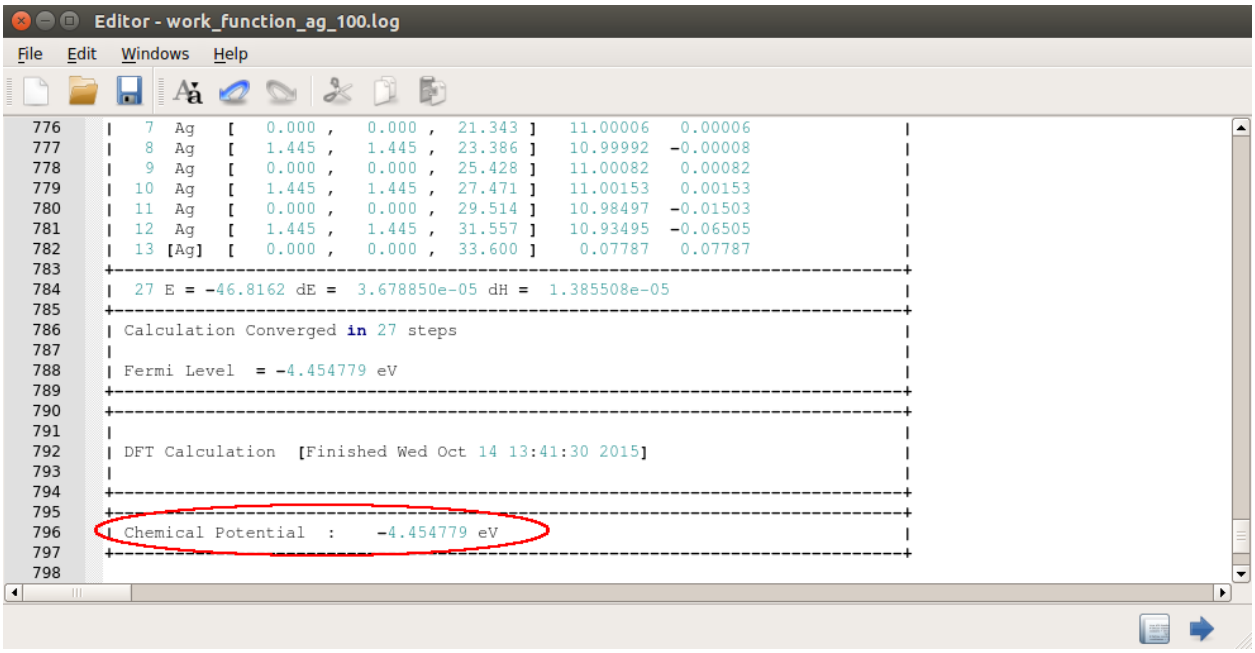


Note: Dirichlet boundary condition determines the Hartree potential to be zero on the right cell boundary, to mimic the presence of vacuum far away from the surface. Because of non-periodicity in the C direction, Neumann condition is used to demand the gradient of the Hartree potential to vanish on the left of the system.

Calculation and analysis

1. Send the structure to **Job Manager** and carry out the calculation. After the calculation, inspect the log file by clicking the button.

2. Around the end of the log file, the chemical potential, or the work function is printed. The value obtained for Ag 100 is 4.45 eV, which lies in the range from 4.22 to 4.81 eV, reported by experiments.



3.Just to demonstrate that with **ATK** it is also possible to get accurate results for other metals, see the comparisons with experiments in the table below:

Table 3.4: Comparison between measured and calculated Work Functions (WFs) for different metal surfaces.

Surface	Measured WF (eV)	WF calculated using the LDA (eV)[Page 1499, 1]
Ag (100)	4.64	4.45
Ag (110)	4.52	4.17[2]
Ag (111)	4.74	4.71
Al (100)	4.41	4.42
Au (100)	5.28	5.38[3]
Ca (100)	2.87	2.85
Cu (100)	4.53-5.10	4.92
Li (100)	2.93	3.14
Nb (100)	3.95-4.87	3.95
Pb (100)	4.25	3.95
Pt (100)	5.12-5.93	5.97
Tl (100)	3.84	3.74
W (100)	4.32–5.22	4.01

[1] All calculations were performed using a DoubleZetaPolarized basis set and a grid of 9x9x1 k-points. The rest of the parameters were set as in the example above. Note that this means that all values are not necessarily converged in these parameters. The surfaces are also not relaxed.

[2] The Ag(110) surface is known to undergo reconstruction; the computed value shown here does not take that into account.

[3] Fortuitously, the calculated work function for gold with SingleZetaPolarized is 5.28 eV.

Comments

•Above we have commented in a few places on the importance of carefully checking convergence in certain numerical as well as geometrical parameters. The work function is quite sensitive to all of them, and so, for convenience, let us summarize the most important ones:

–**Number of slab layers**: for silver, tests show that 13 layers, as we used above, is sufficient, but any smaller number gives different results;

–**Extension of vacuum on both sides**: 20 Å on the right cell boundary, as used in this example, is on the safe side; at least for Ag(100), even just 10 Å is enough;

–**K-point sampling**: 9x9x1 gives reasonable results, but as always with k-point sampling the convergence in this parameter is not a simple monotonous function;

–**Basis set**: for transition metals, polarization orbitals are needed to get correctly shaped atomic functions, and hence we choose *SingleZetaPolarized* for silver. However, for other categories of elements other rules may apply. Independently on the element, the *DoubleZeta* basis set is a good work horse. *DoubleZetaPolarized* is always a safer choice, although it demands a bit more computation time;

–**Exchange-correlation functional**: simple metals are generally well described by LDA, but this is certainly not true for all materials.

•The script can be run in parallel using MPI for a considerable performance benefit, due to the large number of k-points, which is one of the parameters QuantumATK parallelizes well over.

•By comparing the values reported in the table above for the (100), (110) and (111) crystalline faces of Ag, it appears clear that the work function depends on the crystalline face considered.

List of references:

•A. W. Dweydari and C. H. B. Mee, physica status solidi (a) 27, 223 (1975)

•M. Chelvayohan and C. H. B. Mee, J. Phys. C. Solid State Phys. 15, 2305 (1982)

•CRC Handbook on Chemistry and Physics, pages 12-114 (2008)

•J. K. Grepstad, P. O. Gartland and B. J. Slagsvold, Surface Science 57, 348-362 (1976)

•J. C. Rivière, Appl. Phys. Lett. 8, 172 (1966)

3.11.9 Tuning the work function of silver by deposition of ultrathin oxide layers

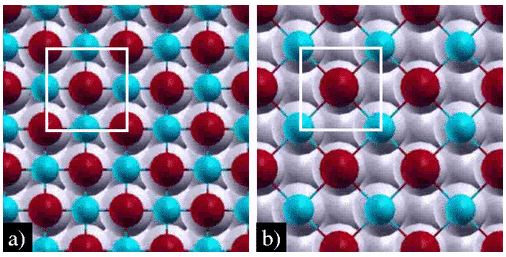
Version: 2015.1

Ultra-thin films of insulating materials deposited on a metal substrate constitute a peculiar class of materials with tunable properties and growing potential in different fields[1][2][3]. In this tutorial, you will learn how to change the work function of a substrate by depositing an overlayer of another material.

[1] Livia Giordano and Gianfranco Pacchioni. Oxide Films at the Nanoscale: New Structures, New Functions, and New Materials. *Accounts of Chemical Research*, 44(11):1244–1252, 2011. doi:10.1021/ar200139y.

[2] Gianfranco Pacchioni. Two-Dimensional Oxides: Multifunctional Materials for Advanced Technologies. *Chemistry - A European Journal*, 18(33):10144–10158, 2012. doi:10.1002/chem.201201117.

[3] Gianfranco Pacchioni and Freund Hajo. Electron Transfer at Oxide Surfaces. The MgO Paradigm: from Defects to Ultrathin Films. *Chemical Reviews*, 113(6):4035–4072, 2013. doi:10.1021/cr3002017.



Note: **Work function tuning** An important consequence of the deposition of a thin insulating film on a metal substrate is the induced change in work function of the metal support, which can be lowered or increased depending on the nature of the interface. Examples of such changes in work function have been reported in the literature. Kelvin probe force microscopy or scanning tunneling microscopy studies of alkali chloride thin films on Au(111) and Ag(100) have shown work function reductions of 0.5–1.2 eV[4][5]. Another work based on field-emission resonance has found for NaCl islands of up to 3 ML (monolayers) a work function reduction of 1.3 eV[6]. Theoretical calculations have predicted a reduction in work function for NaCl, MgO, and other oxides on various metals[7][8][9].

You will here calculate the work function change of a metallic Ag(100) surface as a consequence of depositing 1 to 3 layers of insulating MgO. The procedure to calculate the work function follows the prescription given in the tutorial *Computing the work function of a metal surface using ghost atoms*.

In particular, you will:

1. create and optimize the Ag and MgO bulk structures;
2. construct the Ag(100) and MgO(100) surfaces;
3. create the MgO(100)/Ag(100) interface;
4. set up work function calculations and run them;
5. analyze the results and compare with literature.

Warning: **Computational settings** Throughout this tutorial you will use a particular set of computational settings (mesh cut-off, k-point sampling, basis-set, exchange-correlation, number of metal layers, etc.) which are chosen such that the QuantumATK results can be compared to literature. Always keep in mind, however, that you should check that your results are properly converged with respect to such settings.

[4] Christian Loppacher, Ulrich Zerweck, and M.ăLukas Eng. Kelvin probe force microscopy of alkali chloride thin films on Au(111). *Nanotechnology*, 15(2):S9, 2004. URL: http://iopscience.iop.org/article/10.1088/0957-4484/15/2/003/meta.

[5] Pivetta Marina, Patthey François, Stengel Massimiliano, Baldereschi Alfonso, and Schneider Wolf-Dieter. Local work function Moire pattern on ultrathin ionic films: NaCl on Ag(100). *Phys. Rev. B*, 72:115404, Sep 2005. doi:10.1103/PhysRevB.72.115404.

[6] Ploigt Hans-Christoph, Brun Christophe, Pivetta Marina, Patthey François, and Schneider Wolf-Dieter. Local work function changes determined by field emission resonances: NaClAg(100). *Phys. Rev. B*, 76:195404, Nov 2007. doi:10.1103/PhysRevB.76.195404.

[7] G.ăButti, M.ăI. Trioni, and H.ăIshida. Electronic properties calculation of MgO thin films adsorbed on semi-infinite Ag(001). *Phys. Rev. B*, 70:195425, Nov 2004. doi:10.1103/PhysRevB.70.195425.

[8] Livia Giordano, Fabrizio Cinquini, and Gianfranco Pacchioni. Tuning the surface metal work function by deposition of ultrathin oxide films: Density functional calculations. *Phys. Rev. B*, 73:045414, Jan 2006. doi:10.1103/PhysRevB.73.045414.

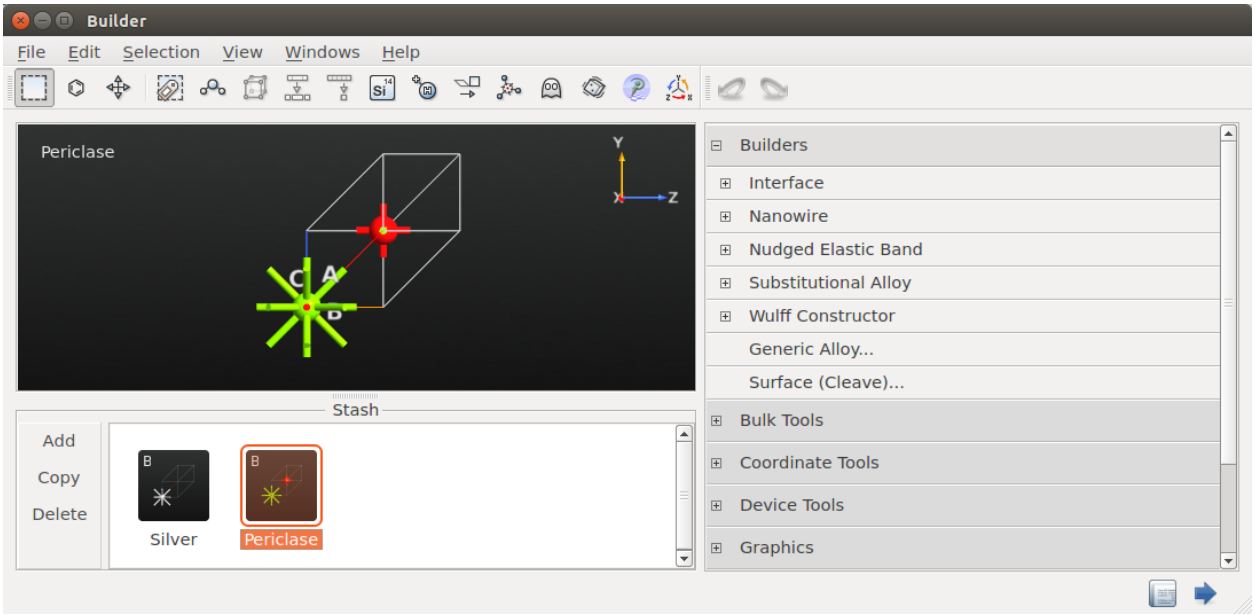
[9] Stefano Prada, Umberto Martinez, and Gianfranco Pacchioni. Work function changes induced by deposition of ultrathin dielectric films on metals: A theoretical analysis. *Phys. Rev. B*, 78:235423, Dec 2008. doi:10.1103/PhysRevB.78.235423.

Ag(100) and MgO(100) surfaces

Do the following to build the silver and magnesium oxide bulks:

•Open the **Builder** and click *Add* → *Add From Database*, locate Silver in the database, and add it to the Stash.

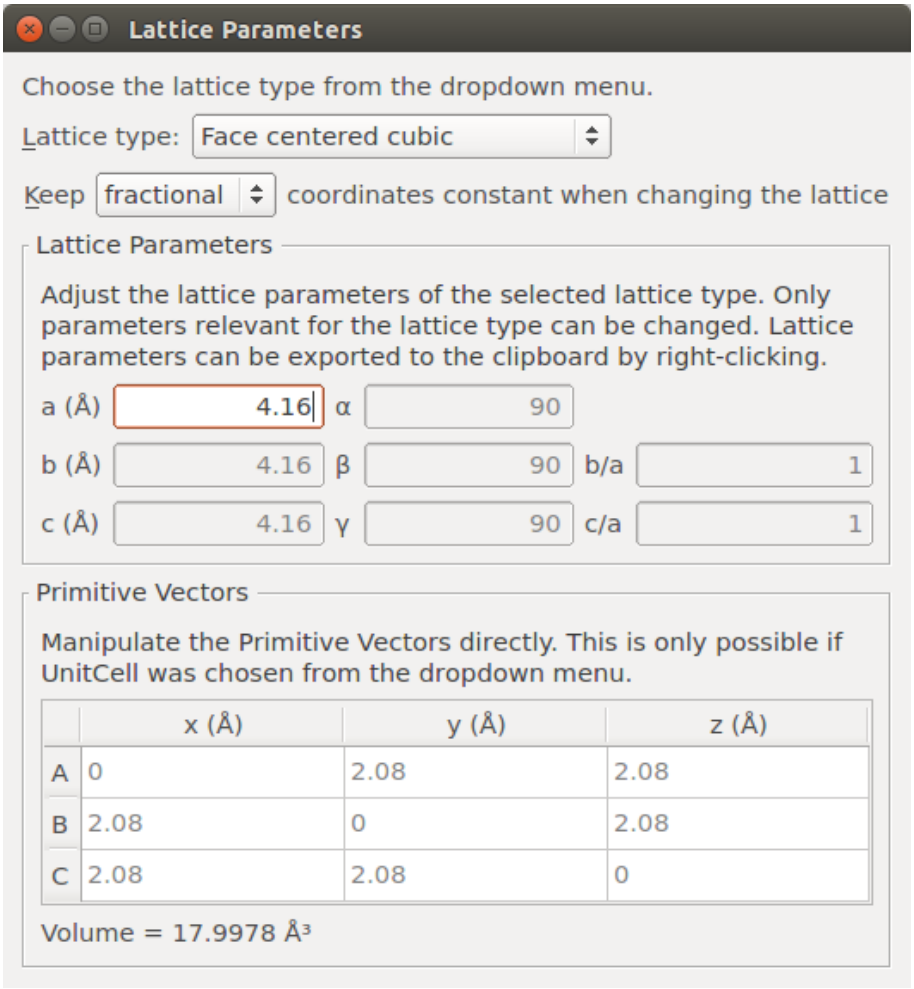
•Also locate MgO and add it to the Stash. You should now have to bulk configurations in the Stash:



You will in this tutorial use the PW91 exchange-correlation functional for DFT simulations, in order to compare to literature results from Prada *et al.*[Page 1501, 9]. The Ag lattice constant is 4.16 Å within PW91, so use this value for the silver bulk configuration:

•Select the Silver Stash item, and open the *Bulk Tools* → *Lattice Parameters* plugin.

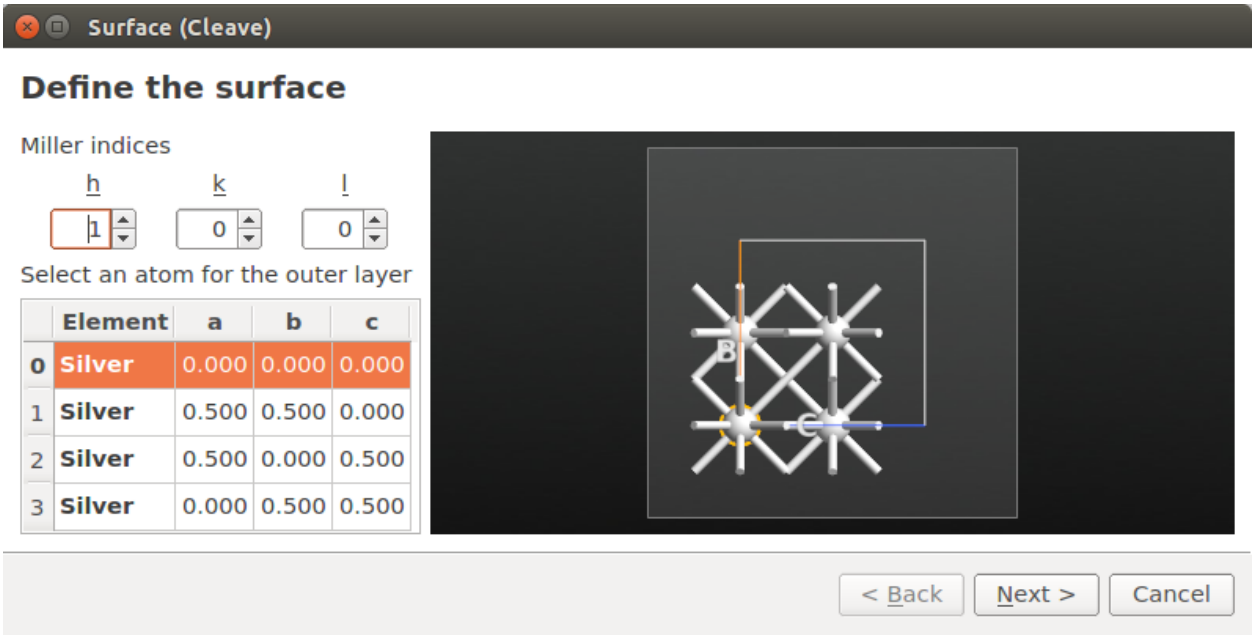
•Make sure fractional coordinates of the atoms are conserved when the lattice is changed, and enter 4.16 Å for the lattice parameter. Close the window.



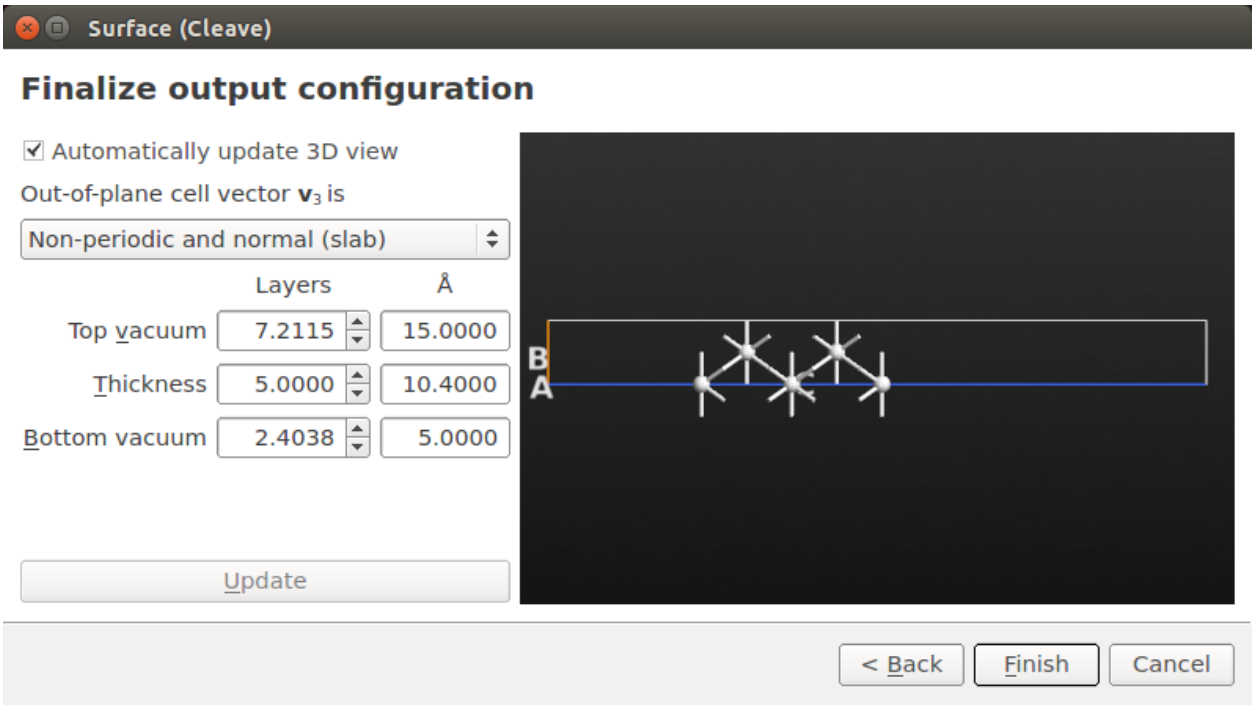
Hint: You could also simply perform a DFT geometry optimization of the silver bulk configuration using PW91 and 11x11x11 k-points, and thereby relax the unit cell. The result of this more general approach would be approximately 4.16 Å for the lattice constant.

Next, create the Ag(100) surface:

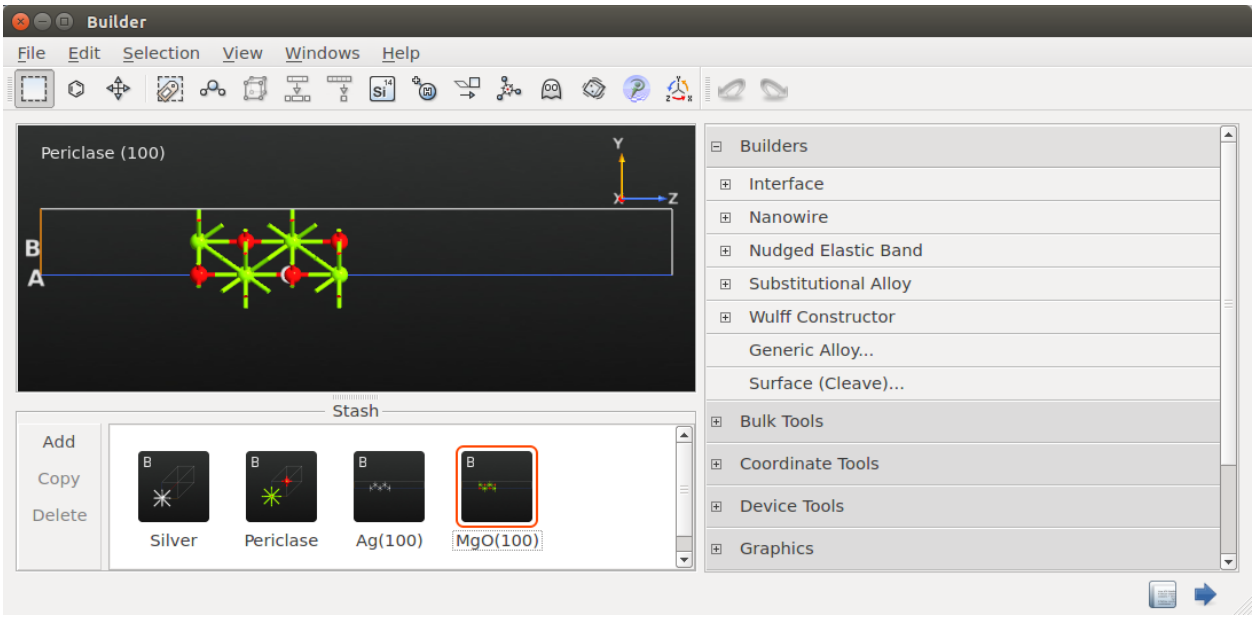
•Use the *Builders* → *Surface (Cleave)* tool to cleave the bulk silver along the [100] direction. Click *Next* twice.



•Choose *Non-periodic and normal (slab)* for the out-of-plane cell vector, and set the slab thickness to 5 layers with 15 Å top vacuum and 5 Å bottom vacuum. The large vacuum ensures that the effective potential can decay smoothly to zero above the slab.



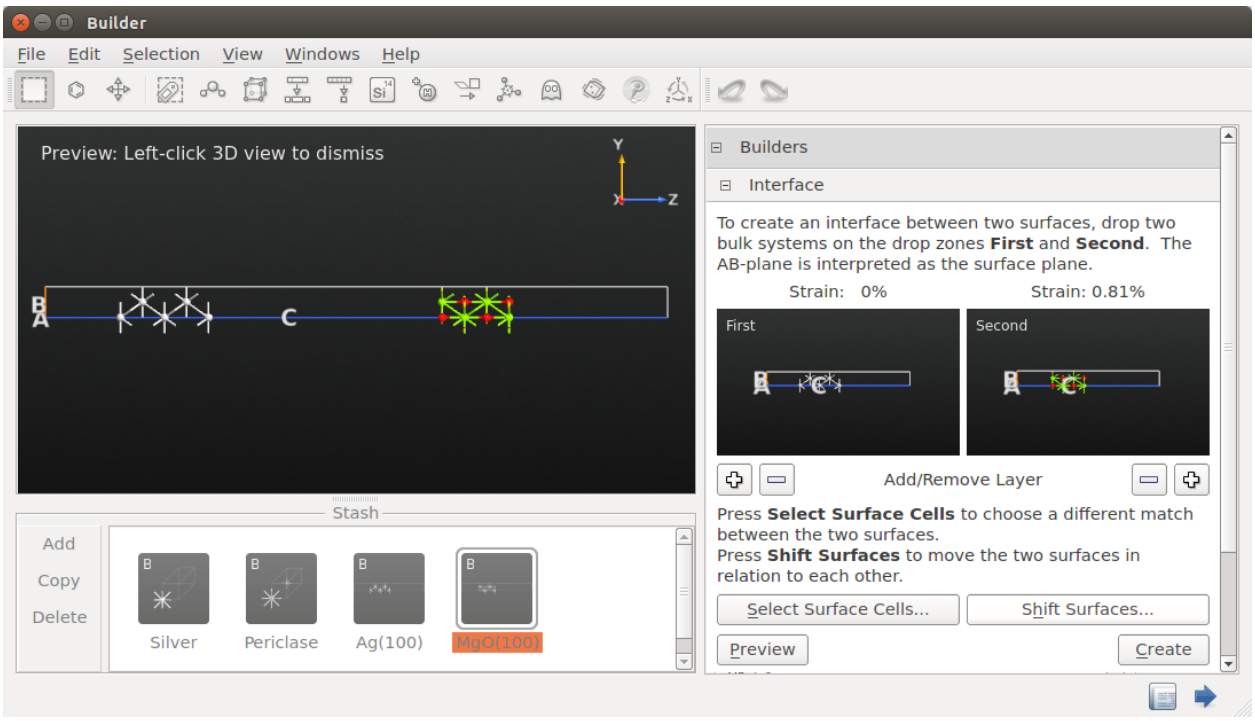
Follow the same steps to create a 4-layer MgO(100) surface, and then rename the two new Stash items to Ag(100) and MgO(100), respectively.



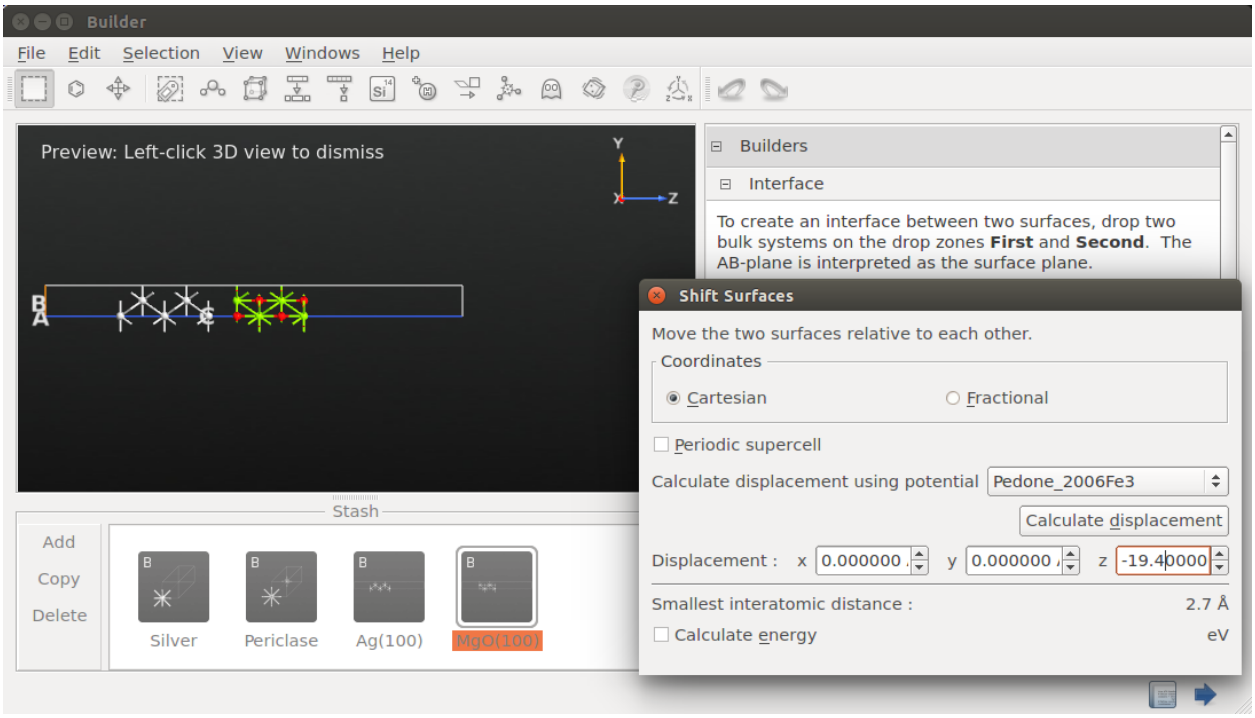
Ag/MgO interface

You should now use the *Builders* → *Interface* plugin to create the Ag(100)/MgO (100) interface:

•Open the Interface plugin, and drag and drop the Ag(100) and MgO(100) configurations into the two drop zones. Notice that the *Second* surface (MgO) is automatically strained slightly, 0.81%, such that it matches the *First* surface (Ag):



•The Ag–MgO separation is way too large. Click the **Shift Surfaces** botton and enter a displacement along z of -19.4 Å in order to move the MgO(100) slab closer to the Ag(100). The interface separation should now be 2.7 Å. Note that an O atom is in an on-top position relative to an Ag atom, so you do not have to shift the MgO slab in the xy plane:



•Close the Shift Surfaces window and click *Create* to add the interface to the Stash.

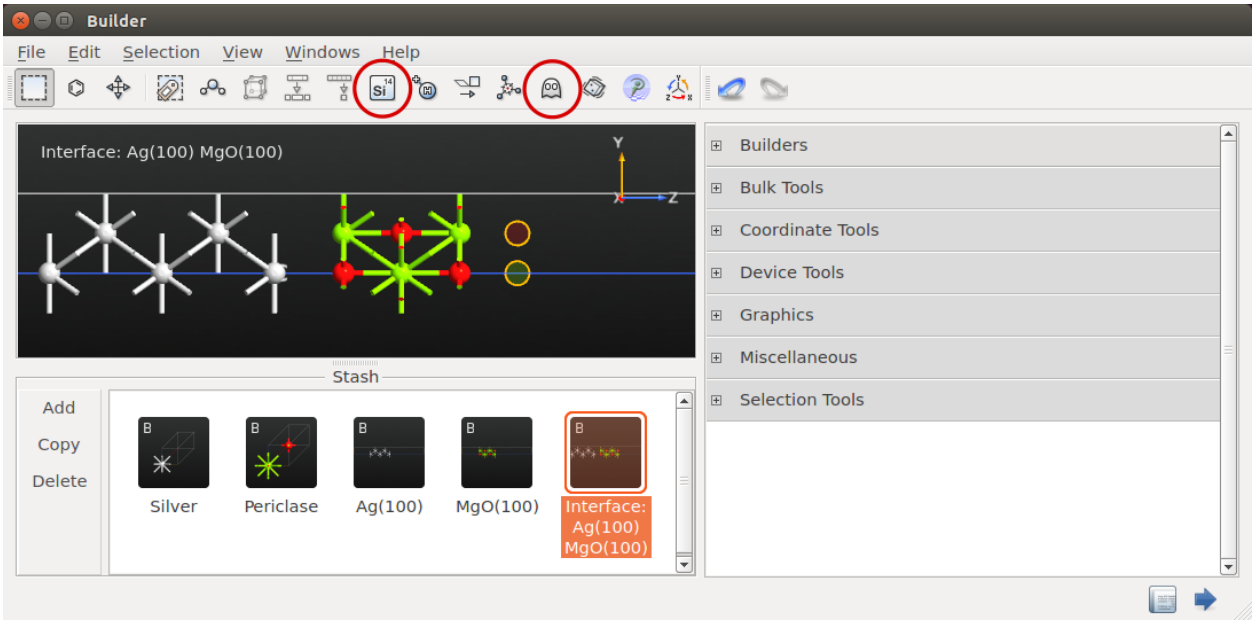
Tip: You can learn more about the Interface Builder in the Technical Notes on *Interface Builder*.

DFT calculations

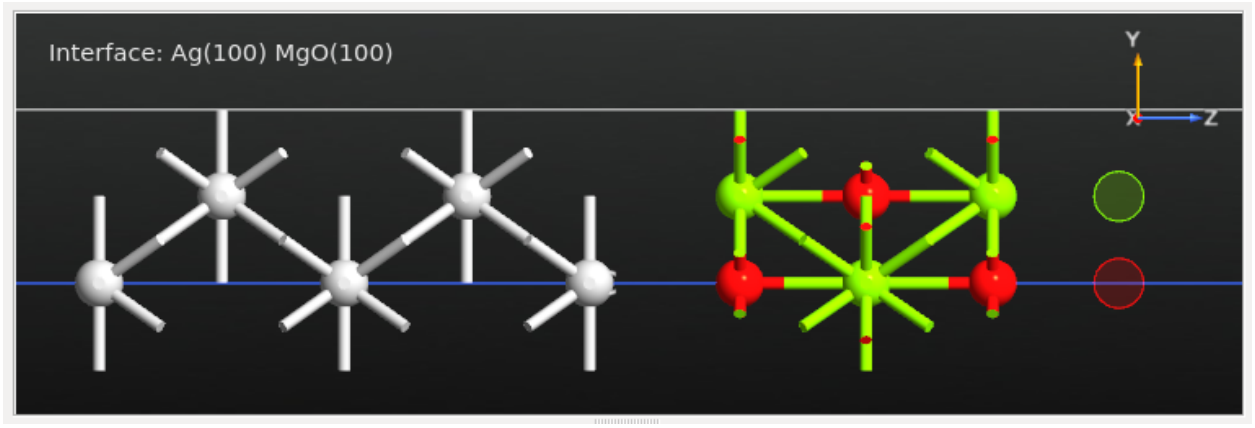
As explained in the tutorial *Computing the work function of a metal surface using ghost atoms*, you will need to add ghost atoms above the surface or interface. Work function calculations require a very good description of the charge density extending into the vacuum, and ghost atoms offer exactly this.

Adding ghost atoms

•Select the outermost O and Mg surface atoms (two atoms in total) and click the icon in the tool bar at the top of the Builder window:



•Next, swap the identity of the two ghost atoms, such that the O ghost atom is above the surface O atom and likewise for Mg. You can simply select a (ghost) atom and use the Si icon encircled in the image above to select from the periodic table.

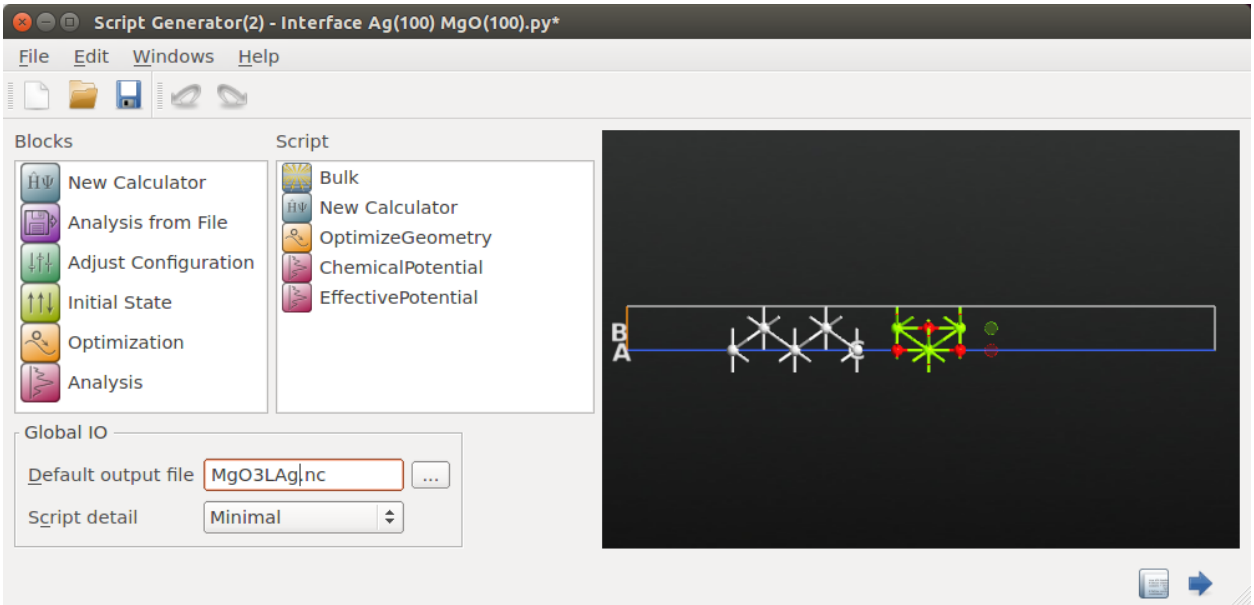


•Finally, send the configuration to the **Script Generator** .

ATK-DFT calculations

The work function is identified as the chemical potential when the effective potential is zero on the cell boundary above the surface. A Dirichlet boundary condition (BC) is used to enforce this. You will also compute the effective potential in order to inspect it visually. Set up the required DFT calculations in the following manner:

•Change the default output name to `MgO3LAg.nc` and add four blocks to the script: New Calculator, Optimize-Geometry, ChemicalPotential, and EffectivePotential.



•Open the New Calculator block and set the following parameters:

–*k-point Sampling*: 11x11x1;

–*Iteration Control*: Tolerance = 10^{-5} Hartree;

–*Exchange Correlation*: GGA.PW91;

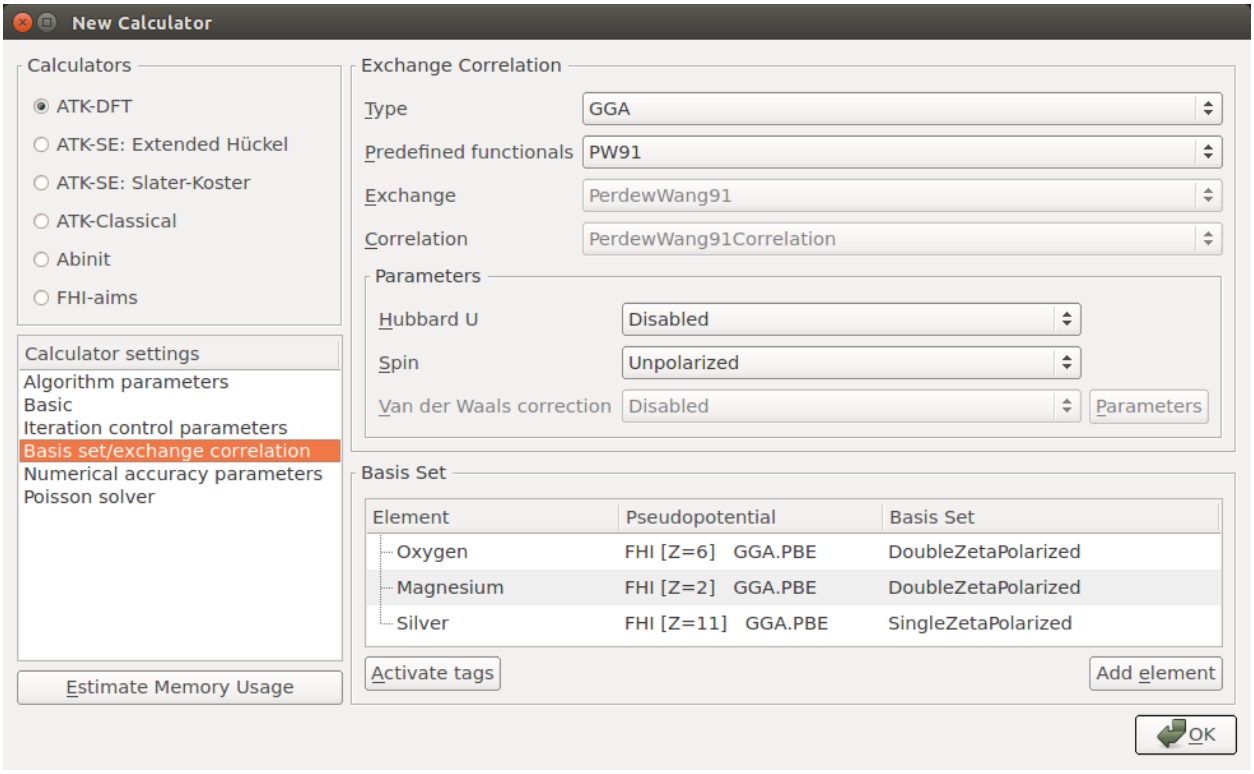
–*Basis set*: DoubleZetaPolarized for O and Mg, SingleZetaPolarized for Ag;

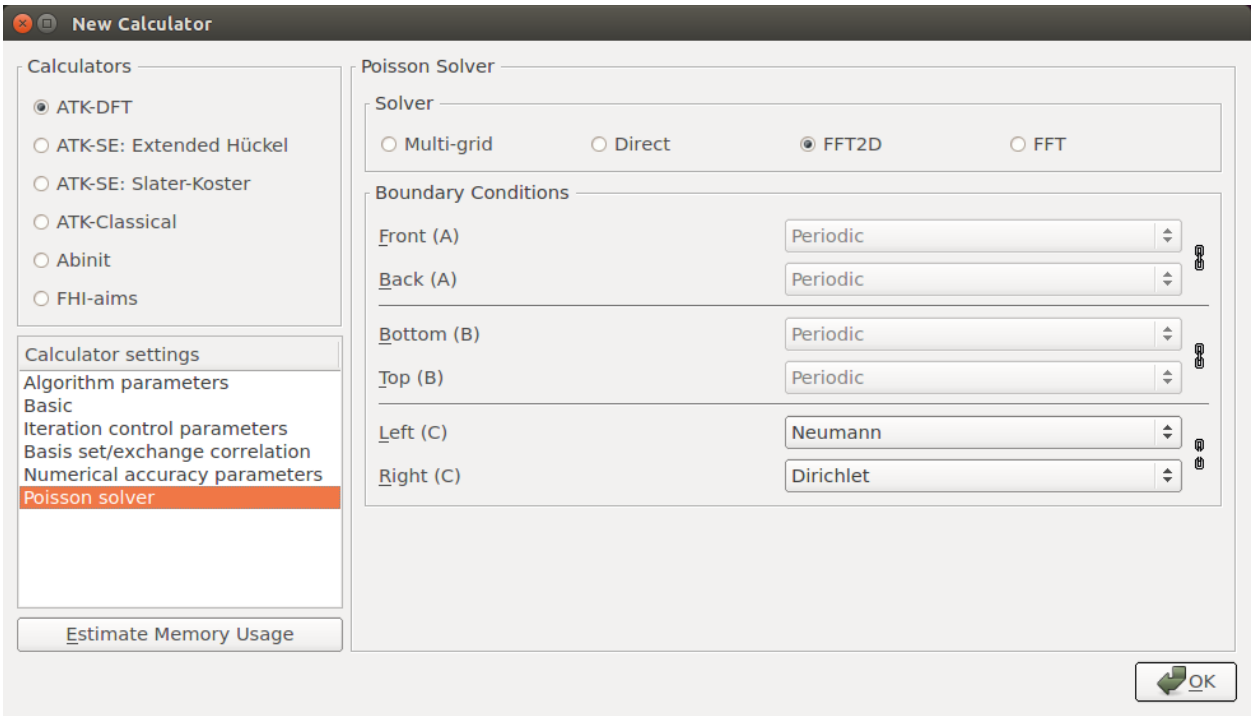
–*Poisson solver*:

∗choose the FFT2D solver,

∗select a Neumann BC on the left C face,

∗select a Dirichlet BC on the right C face.



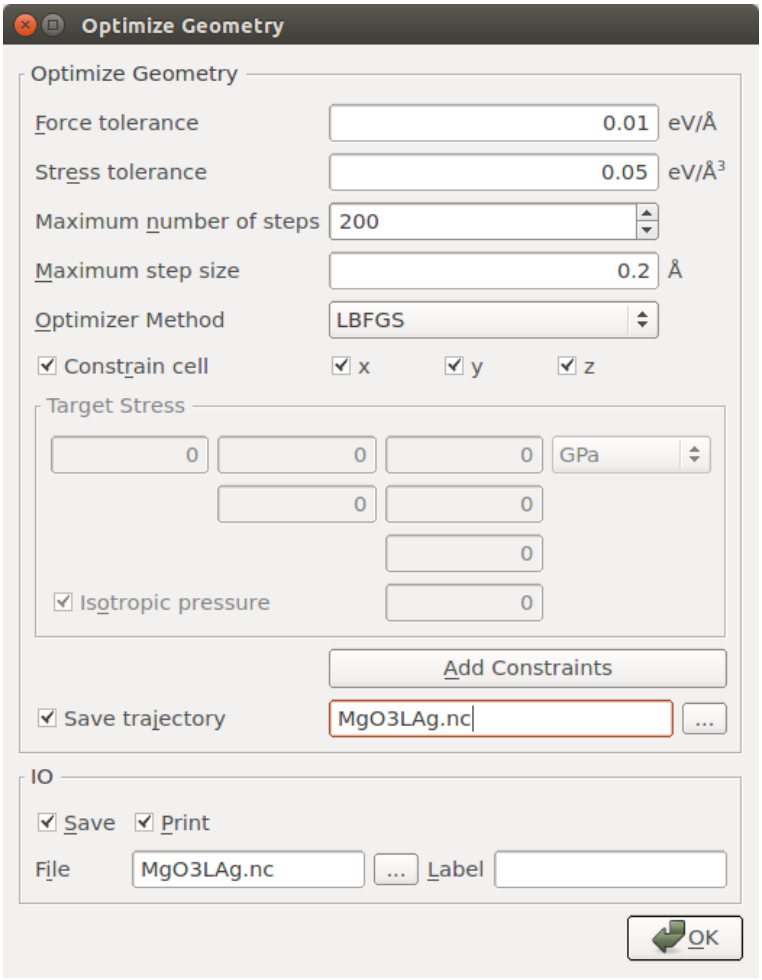


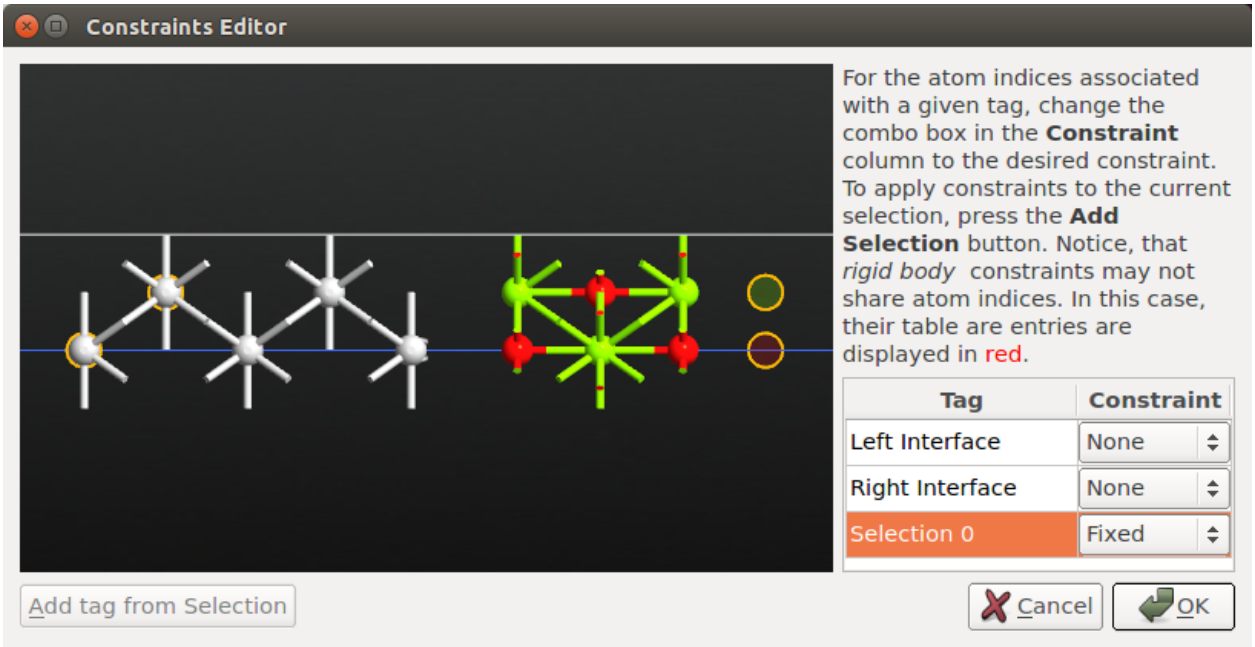
•Next, open the OptimizeGeometry block and edit it. In particular, make sure to constrain the bottom of the Ag(100) slab and the ghost atoms during the optimization:

–Decrease the force tolerance to 0.01 eV/Å.

–Tick *Save trajectory*, and enter `Mg03LAg.nc` as the file name.

–Click **Add Constraints** and select the bottom two Ag atoms and the two ghost atoms. Then click *Add tag from selection* and choose a *Fixed* constraint for that group (Selection 0).

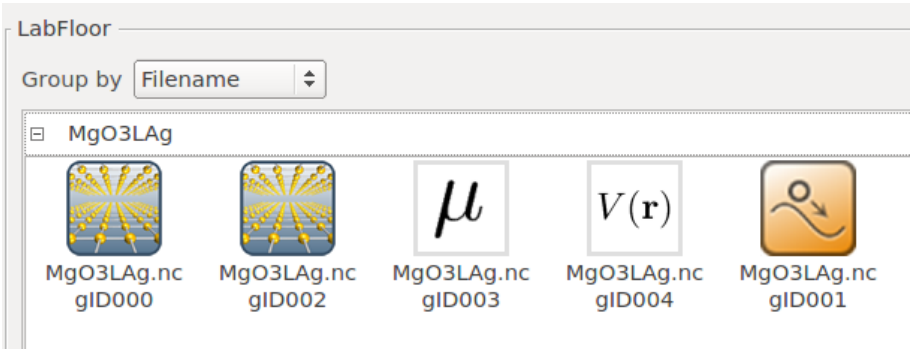




•Save the script as `MgO3LAg.py` and execute it using the **Job Manager** . If needed, you can also download the final script here: `MgO3LAg.py`. It should only take about five minutes to finish if executed in serial on a modern laptop. This can be reduced to 2.5 minutes if executed in parallel with 4 MPI processes.

Analyzing the results

The OptimizeGeometry, ChemicalPotential and EffectivePotential objects should now have appeared on the QuantumATK **LabFloor**:



OptimizeGeometry **ChemicalPotential** **EffectivePotential**

Try to select the OptimizeGeometry object and visualize the relaxation trajectory by clicking the **Viewer** plugin on the right-hand side of the LabFloor. Click to start the video. Confirm that the constrained Ag and ghost atoms are indeed fixed during relaxation.

Work function

Select the ChemicalPotential object and click the **Show Text Representation** plugin to read off the calculated chemical potential of -2.99 eV.

The work function for this 3 layer MgO on Ag(100) is therefore 2.99 eV, which is in a very good agreement with the calculated value of 2.96 eV reported in the literature by Prada *et al.*[Page 1501, 9].

You can also create the 2L-MgO/Ag(100), 1L-MgO/Ag(100) and the Ag(100) systems following the steps described above. Another approach is to use the relaxed 3L-MgO/Ag(100) configuration as a starting point for the other systems. This will reduce the number of BFGS steps needed for relaxing those systems, and thereby save computational time.

For example, to create the 2L-MgO/Ag system and calculate the work function:

•Locate the relaxed 3L-MgO/Ag(100) configuration on the LabFloor. It has object ID *glD002*.

•Transfer it to the Builder and rename it `MgO2LAg`.

•Delete the two O and Mg ghost atoms, and convert the new surface O and Mg atoms to ghost atoms.

•Swap the Mg and O ghost atoms as described above.

•Send the configuration to the **Scripter** and set up the calculation as descibed above.

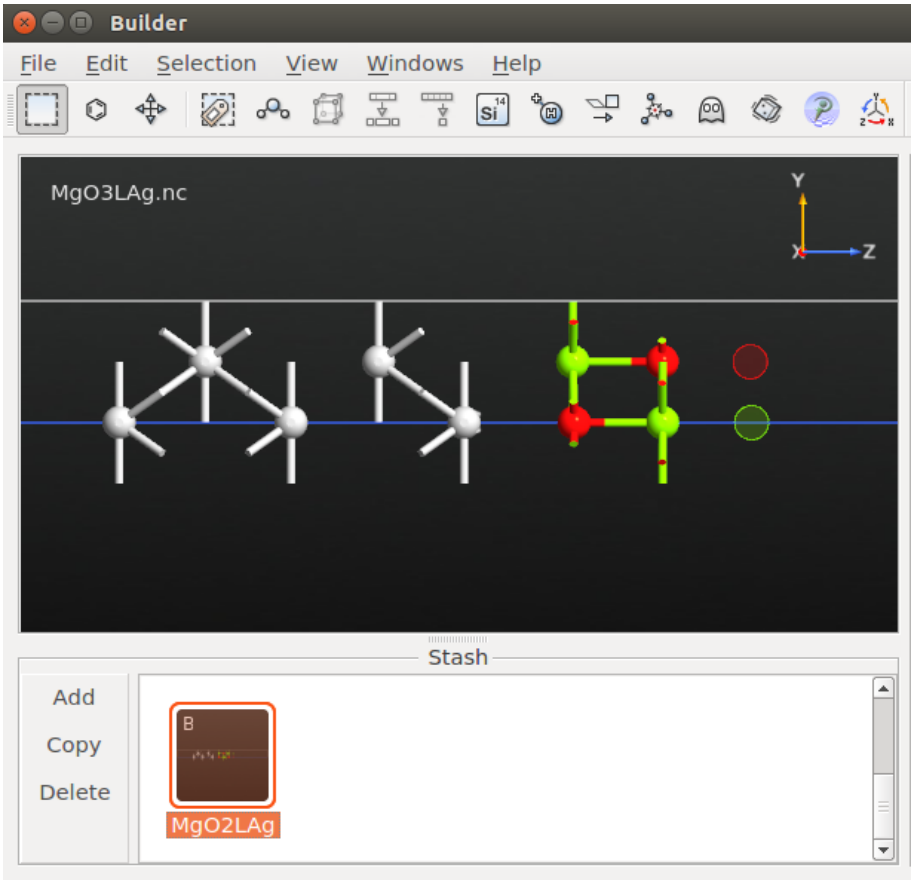


Table 3.5: Table: Calculated PW91 work functions (eV). The corresponding work function difference wrt. to clean Ag(100) is reported in parenthesis.

	QuantumATK	Pada *et al.*[Page 1501, 9]
Ag(100)	4.22	4.23
1L-MgO/Ag	3.31 *(-0.91)*	3.29 *(-0.94)*
2L-MgO/Ag	2.97 *(-1.25)*	2.95 *(-1.28)*
3L-MgO/Ag	2.99 *(-1.23)*	2.96 *(-1.27)*

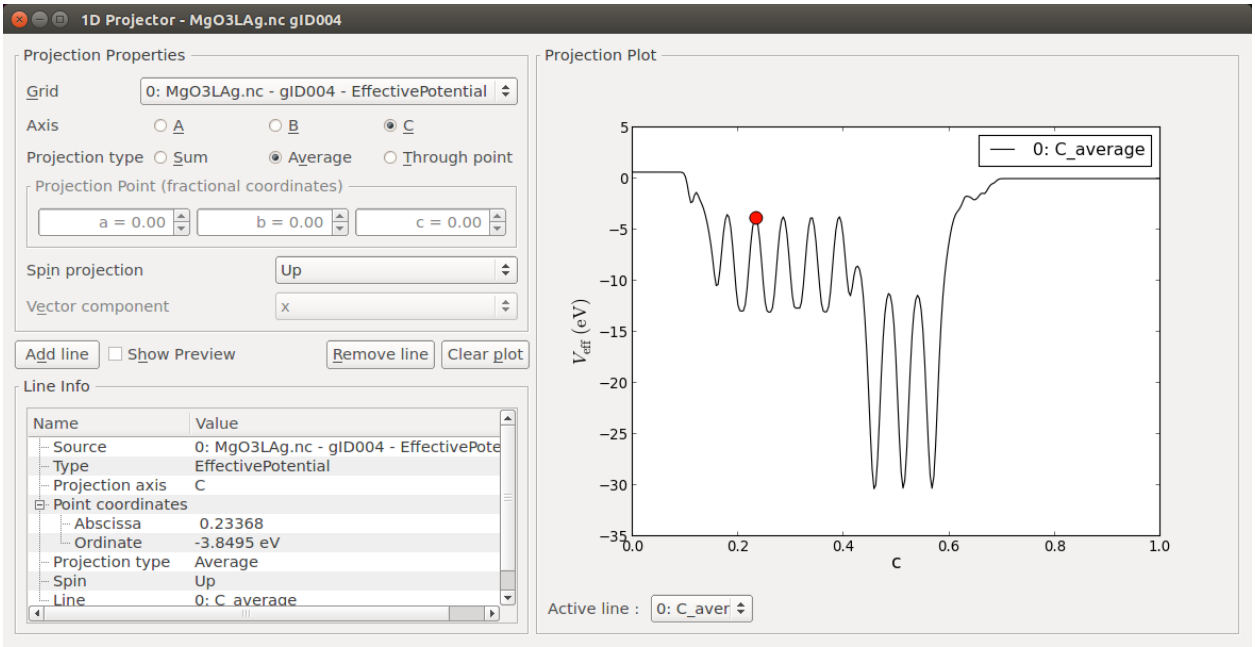
The scripts needed to calculate all the work functions in the table column labelled QuantumATK may be downloaded here: `Ag100.py`, `MgO1LAg.py`, `MgO2LAg.py`, `MgO3LAg.py`.

Tip: The table above shows excellent agreement between QuantumATK and literature (VASP) work function calculations. However, results may change if the computational settings change. For example, a SZP basis set was used for the Ag atoms – a DZP basis set may give slightly different results. The type of pseudopotentials used may also affect results, and more ghost atoms may be needed in some cases.

Effective potential

You have used a particular set of boundary conditions for the work function calculations – Neumann on the left C face, and Dirichlet on the right C face. You can now use the **1D Projector** plugin to visualize the average effective potential in the calculations:

- Select the EffectivePotential object on the LabFloor, and click the 1D Projector plugin.
- Select to project along the C axis using the *Average* projection type, and click **Add line** to plot the projection:



The effective potential is clearly different in the Ag(100) and MgO(100) regions. Moreover, the effect of the two different BCs are quite clear from the value and slope of the potential in both ends of the supercell:

- The Neumann BC on the left C face has imposed a zero slope of the effective potential on the boundary, but does not constrain the actual value of the potential on the boundary.
- On the contrary, the Dirichlet BC on the right C face has forced the effective potential to zero on the boundary, and the slope just happens to be approximately zero in the vacuum region.

1D Projector plugin

Common QuantumATK grid objects

BlochState, EffectivePotential, Eigenstate, ElectronDensity, ElectronDifferenceDensity, ElectrostaticDifferencePotential, ExchangeCorrelationPotential, ExternalPotential, LocalDeviceDensityOfStates, TransmissionEigenstate, ElectronLocalizationFunction.

The 1D Projector can be used to project all sorts of 3D grid data onto a 1D representation. This is very useful for visualization purposes, and works for a wise range of QuantumATK grid objects (see the box).

Several options are available in the plugin widget:

Grid
You can open the projector tool with multiple objects selected on the LabFloor, to plot them next to each other. Here

you choose which one to plot.

Axis
Choose along which direction you want to project your 3D data grid.

Projection type
Sum or average all the data in the plane perpendicular to the selected direction. You can also simply plot the single values along a line passing through a particular projection point.

Projection point
Specify, in fractional coordinates, the projection point to be used in case the projection type *Through point* is selected.

Spin projection
In case of spin polarized calculations, collinear or non-collinear, you can select the specific spin projection.

Add line
Once the options above are specified, click this button to plot your projection in the window on the right-hand side. You can add more projections to the same plot.

Remove line
Select one line in the Projection Plot window and click to remove the line from the plot.

Clear plot
Remove all lines from the plot.

Line Info
Some useful information about the currently selected plot line/point are shown. Note that the plot is interactive. Click on any point of the plot to print the corresponding information.

Projection Plot
Right-click to zoom, customize or export data to file.

References

3.11.10 Calculating Reaction Rates using Harmonic Transition State Theory

Introduction

In this tutorial, you will learn to use **QuantumATK** for calculating reaction rates using the nudged elastic band (NEB) method and harmonic transition state theory (HTST). These two techniques are a powerful combination that allow researchers to model solid-state reactions (or more generally any rare event dynamics). Typically, reactions in the solid state are too slow to efficiently study with molecular dynamics (MD) simulations. HTST, on the other hand, takes a statistical mechanics approach to computing the reaction rate. Therefore, in constrast to MD simulations, the amount of computational work required to calculate a reaction rate is independent of the timescale of the reaction.

Theory

HTST[1] is derived from a more general formalism known as transition state theory (TST)[2][3]. TST works by defining a transition state (a hyperplane) that separates the reactant from the products. Then the reaction rate is calculated as the probability of finding the system at the transition state relative to the entire reactant state, multiplied with the rate of crossing from the reactant region to the product region at the transition state. In general, TST is difficult to use, because it requires a definition of the transition state hyperplane. Additionally, calculating the probability of being at the

[1] GeorgeăH. Vineyard. Frequency factors and isotope effects in solid state rate processes. *Journal of Physics and Chemistry of Solids*, 3(1):121–127, 1957. doi:10.1016/0022-3697(57)90059-8.

[2] Henry Eyring. The Activated Complex in Chemical Reactions. *The Journal of Chemical Physics*, 3(2):107–115, 1935. doi:10.1063/1.1749604.

[3] ArthurăF. Voter and JimmieăD. Doll. Transition state theory description of surface selfdiffusion: Comparison with classical trajectory results. *The Journal of Chemical Physics*, 80(11):5832–5838, 1984. doi:10.1063/1.446610.

transition state requires evaluating the full partition function of the transition state region and the reactant region, which is computationally prohibitive when using density functional theory, and challenging even for fast classical potentials.

TST can be made a more practically useful theory by making two simplifying assumptions. The first is that saddle points on the potential energy surface represent bottle neck regions that separate reactant from product configurations. This solves the issue of determining the hyperplane in TST by defining it as the hyperplane normal to the unstable direction at the saddle point. The second is that the potential energy surfaces around the reactant configuration and the saddle point are locally harmonic (quadratic). This allows for a second order Taylor series expansion of the potential energy surface at those points and the resulting configuration integrals needed in TST are now closed-form expressions. The combination of these two assumptions is called HTST.

HTST is a good approximation for many solid-state reactions, because the energy barriers are often large compared to the average kinetic energy in the system ($k_{\mathrm{B}}T$). This is important because it ensures that the system reaches a local equilibrium between reactive events. If the barriers are relatively small compared to the thermal energy, then there are dynamical correlations in the trajectory. This means that the history of the trajectory is an important factor in determining the reaction rate and would require the use of a more sophisticated statistical model. Another reason that solid-state reactions are described well by HTST is that the harmonic approximation is often valid. Near the saddle points the potential energy rises rapidly and creates a small bottleneck region, however, in some soft systems (e.g. conformation changes in a protein) the potential energy is relatively flat in the region around the saddle point and is not well characterized by harmonic expansion of the saddle point.

The first step in using HTST is to find the saddle point associated with the reaction in question. There are a number of techniques that can be used, but here we will use the NEB. NEB optimization can be performed in such a way that one of the images converges to a saddle point (as opposed to all the images being evenly spaced along the reaction coordinate). This is called climbing image NEB (CI-NEB) and an optimized CI-NEB should have a saddle point as its highest energy image.

Once the saddle point has been obtained the following formula is used to calculate the reaction rates:

$$k_{\mathrm{HTST}} = \frac{\prod_i^{3N} \nu_i^{\mathrm{R}}}{\prod_i^{3N-1} \nu_i^{\mathrm{S}}} \exp\left[-\left(E^{\mathrm{S}} - E^{\mathrm{R}}\right)/k_{\mathrm{B}}T\right]$$

where N is the number of atoms; ν_i^{R} and ν_i^{S} are the stable (real valued) normal mode frequencies at the reactant and saddle point (the $3N-1$ at the saddle is due to there being exactly one imaginary frequency at the saddle); E^{R} and E^{S} are the energies at the minimum and saddle point; k_{B} is Boltzmanns constant; and T is the temperature.

The functional form of HTST is analogous to the empirically derived Arrhenius rate law:

$$k_{\mathrm{Arrhenius}} = A \exp\left[-E_{\mathrm{a}}/k_{\mathrm{B}}T\right].$$

The pre-exponential factor, A, in the Arrhenius rate law corresponds to the ratio of the product of the vibrational modes at the minimum over the product of the stable vibrational modes at the saddle in HTST. This can be interpreted as an attempt frequency, i.e. the number of times per second that the system vibrates along the reaction direction.

The activation energy, E_{a}, in the Arrhenius rate law is defined as the difference in energy between the saddle and minimum in HTST. The exponential term represents the Boltzmann probability that a vibrational along the reaction direction will overcome the barrier and continue into the product state.

Thus, the HTST rate equation can be interpreted as:

$$k_{\mathrm{HTST}} = [\text{Attempt Frequency}] \cdot [\text{Probability of Success Per Attempt}]$$

Note: In most systems, the prefactor does not change significantly (ca. a factor of 10) from reaction to reaction. However, the exponential term (the Boltzmann probability of the system being at the transition state) can easily span 10 orders of magnitude. This is because most solid-state reactions are thermally activiated processes, whose rate primary depends upon the energy barrier.

Despite the small contribution of the prefactor to the difference in rate between different mechanisms it can be more computationally expensive to compute than the energy barrier.

It is thus recommended to experiment with the system to see if it is neccessary to compute the prefactor every time. In reactions involving light atoms (first two rows of the period table) the prefactor is often in the order of 10^{14} Hz, while for heavier metal atoms it may be as small as 10^{11} Hz. These frequencies are related to a typical vibrational period in these systems, which primarily depends on the atomic mass.

Modeling Pt Adatom Diffusion on Pt(100)

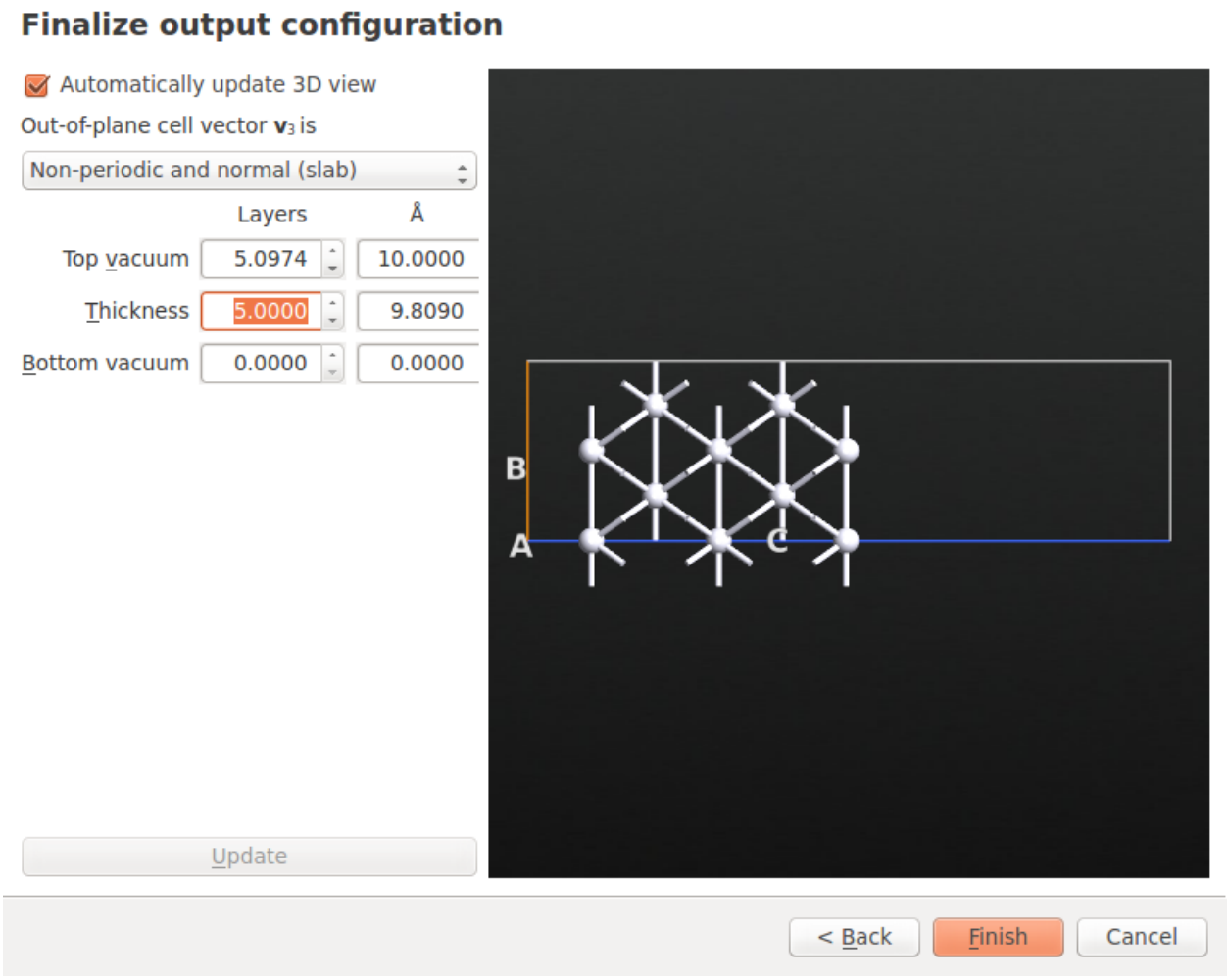
As an model system we will examine Pt adatom diffusion on a Pt(100) surface. There are at two well known mechanisms for adatom diffusion on this surface. The first is the hop mechanism where the adatom directly moves to an adjacent hollow site. The second is the exchange mechanism where the adatom exchanges places with a surface atom and displaces it into an adjacent hollow site.

There are several interesting things we will learn from this system. The first is that although DFT and classical potentials often agree about equilibrium properties, where the classical potentials have been fit, the agreement is often quite bad near saddle points. Often saddle points describe bond breaking and bond formation steps that can be particularly challenging to describe in many classical potentials, especially those potentials that make use of explicit angle terms as is often used to describe Si and C bonding. The second is that what may initially look like a simple one step reaction sometimes actually involves multiple steps. It is not uncommon to encounter a relatively unstable reaction intermediate between the chosen initial and final geometries. This will be an important point to understand since, formally, HTST only describes elementary single step reactions.

Constructing the Pt(100) surface

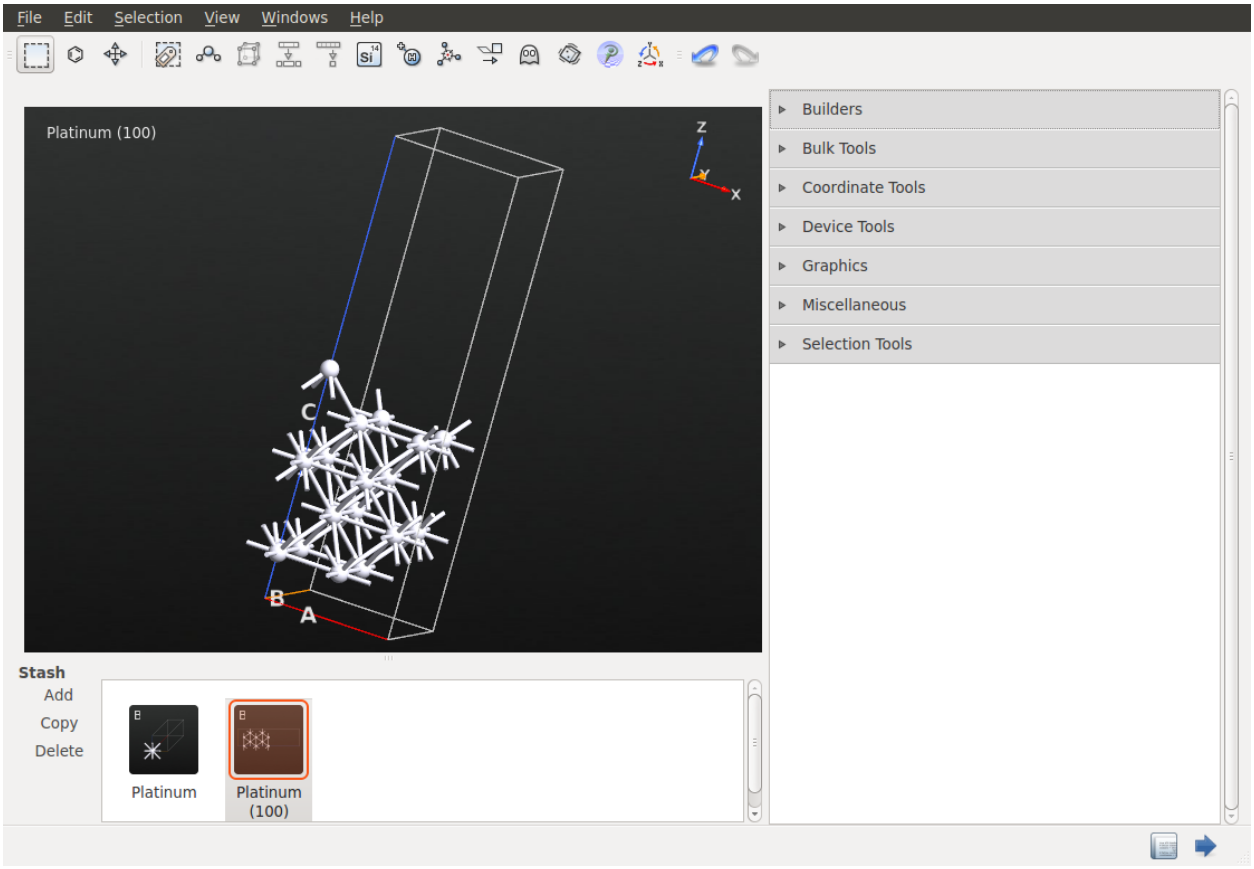
In the **Builder** click *Add* → *From Database* and then search for platinum. A list of structures containing platinum will appear, scroll through them to find Platinum and double click on it.

Then click *Builders* → *Surface (Cleave)* and set the Miller indices to (100) and click *Next*. Create a 2x2 surface lattice and click *Next*. Then choose the Out-of-plane cell vector to be *Non-periodic and normal (slab)*, set the thickness to 5 layers, and click *Finish*.



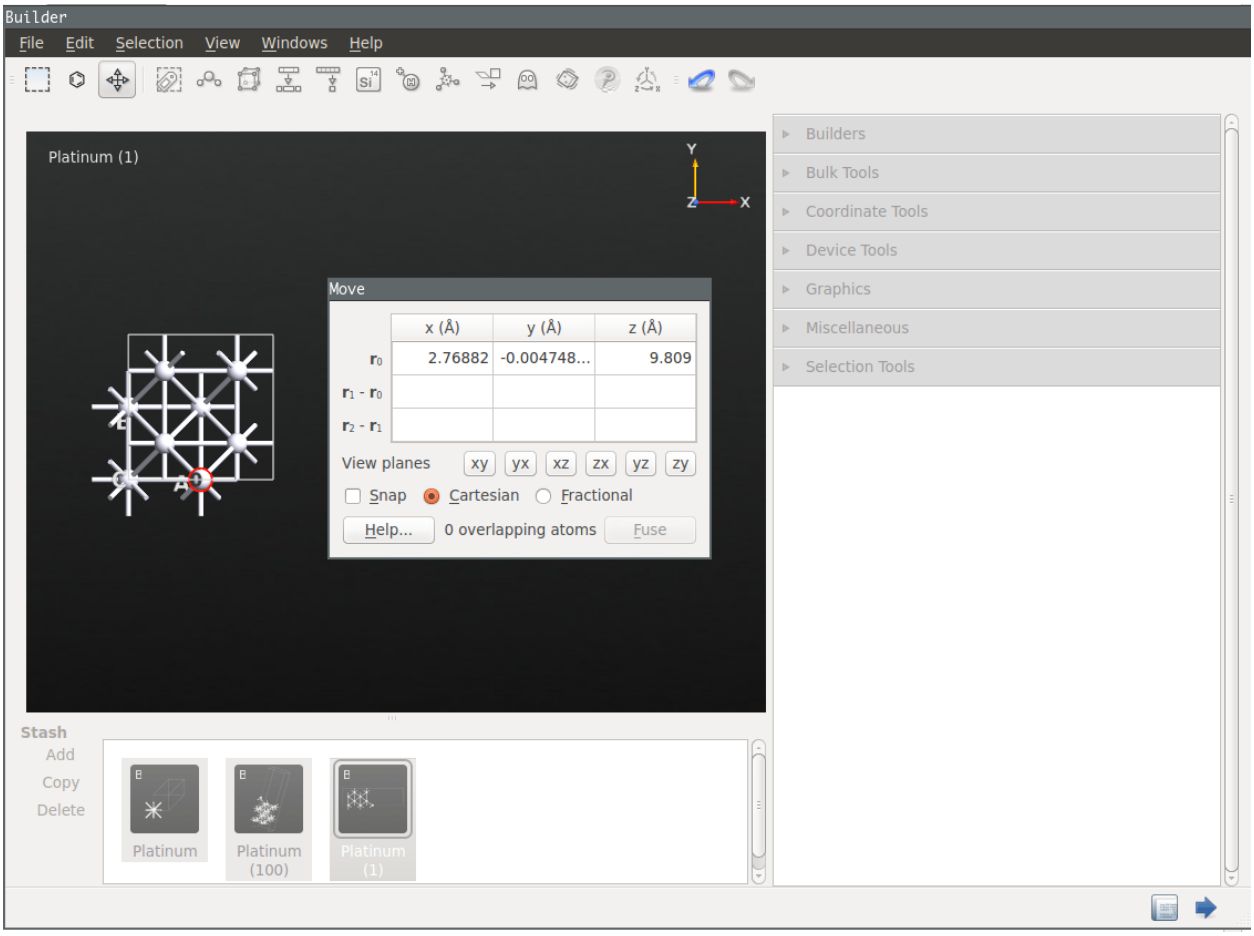
For later visualization it is nice to center the slab model in the x-y plane. To do this click *Coordinate Tools* → *Center*, uncheck the *C* direction and click *Apply*.

Delete all but one of the atoms in the 5th layer. The remaining atom will be the diffusing adatom in the NEB calculation.



Constructing the NEB Configuration

Make a copy of the configuration by clicking the *Copy* button in the **Stash**. Double click on the new configuration. Now select the adatom and click on the move tool in the toolbar and drag the adatom to and adjacent hollow site. This is easiest to do if you first camera view plane to the XY plane.



Click *Builders* → *Nudged Elastic Band* and drag both configurations from the stash to the *initial* and *final* drop zones. Click on *Constraints* and select the bottom most layer. Click *Add tag from Selection*, then change the constraint type from *None* to *Fixed* and click *OK*. Change the NEB method from *Linear* to *Image Dependent Pair Potential*. In this case, as few as 5 images should work well, since it is a relatively simple reaction coordinate. Set the *Maximum distance* to 1 Å and click *Create*.

Right click on the NEB configuration in the *Stash* and select *SendTo* → *Script Generator*.

Optimizing the NEB and Calculating the HTST Rate

Double click on *New Calculator* in the *Blocks* list to add a calculator to the *Script*. Now double click on the just added *New Calculator* in the *Script*. Change the k-point sampling to be 3x3x1 and the *Exchange correlation* function to *GGA*. Then click *OK*.

Double click on *Optimization* → *OptimizeGeometry* in the *Blocks* list and the double click on *OptimizeGeometry* in the *Script* list. Change the *Force Tolerance* to 0.01 eV/Å. Then click *Add Constraints*, select the bottom most layer of atoms, click *Add tag from Selection*, and change the constraint type from *None* to *Fixed*. Check the *Climbing Image Method* check box as well as *Optimize end points*.

It is crucial that a CI-NEB calculation is performed. If a regular NEB optimization is used then the images will be equally spaced along the reaction coordinate. This means that there will most likely not be an image at the saddle point. While, a converged CI-NEB calculation should have a saddle point as the highest energy image.

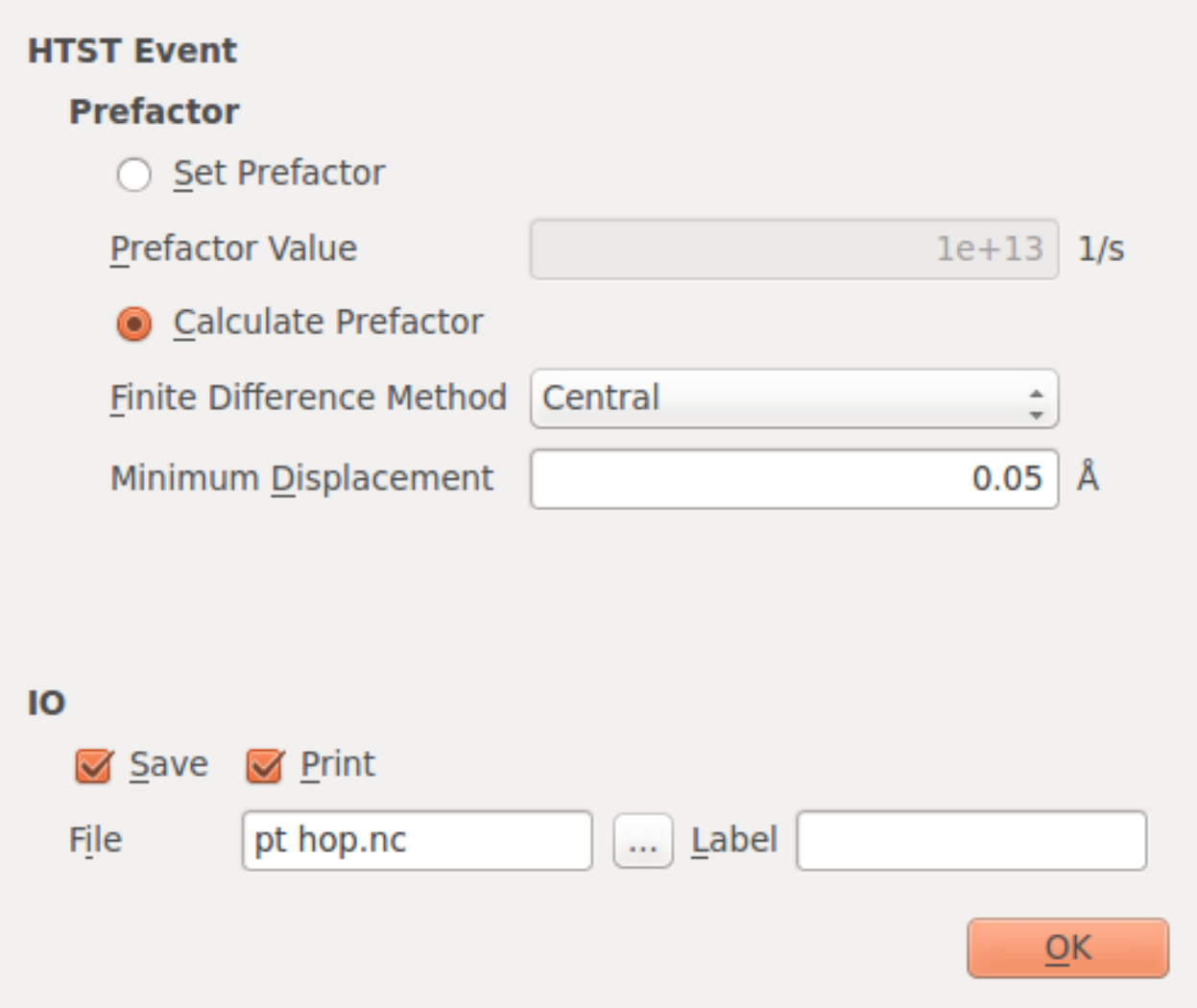
Note: In this tutorial, we are optimizing the NEB end points and the intermediate images in one step. This works fine

for this simple case where the endpoints are close to their optimized geometries, however, it is typically a good idea to break this process up into three steps:

1. Optimize the initial end point.
2. Optimize the final end point.
3. Create the NEB configuration from the optimized endpoints and optimize it.

This approach allows you to analyze and verify the optimized end points to ensure that the geometry optmization did not significantly alter them.

Double click on *Analysis* → *HTSTEvent* in the *Blocks* list then double click on the *HTSTEvent* in the *Script* list. There are two options for determining the prefactor in the HTST rate equation (see the first equation in the introduction).



The first way to calculate the prefactor is to calculate the dynamical matrix for a subset of the atoms in the system. The atoms included in the calculation are those that move by by more than the *Minimum Displacement* parameter along the reaction coordinate along with their nearest neighbors (as determined using covalent radii). The second method is to simply set the prefactor to a known value.

Change the default output file `pt hop.nc`. Then either save the script and run it with *atkpython* or send it to the **Job Manager** to run it.

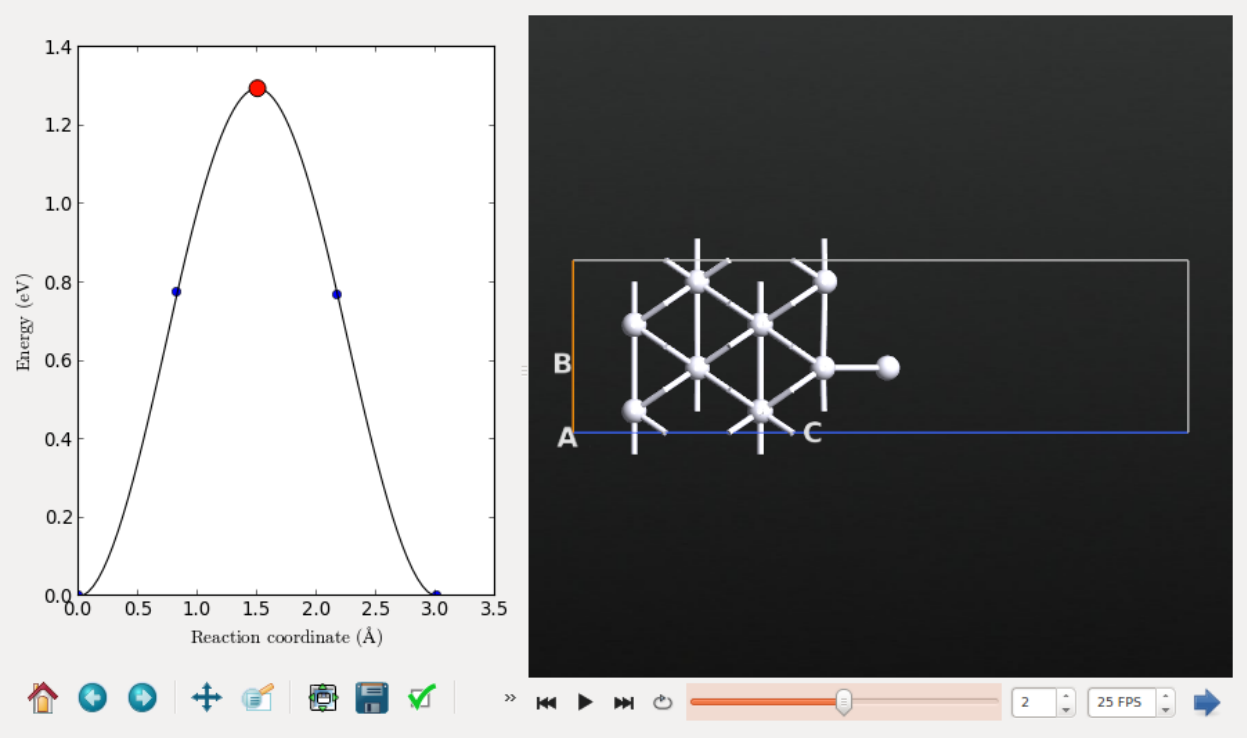
The calculation may take a couple of hours depending upon your hardware and the number of MPI processes used.

Note: It is possible that the highest energy image of the converged CI-NEB calculation will not be a first order saddle point. This will be detected when the frequencies at the saddle point are calculated. An error message will be printed to the log and the calculated frequencies will be printed. There are a number of reasons why this may occur:

1. The NEB optimization did not converge. Consider increasing the maximum number of steps in the optimization and double checking the input geometries to be sure that the initial guess is reasonable.
2. The NEB optimization may have converged, but the force tolerance is too large. Try making the force tolerance smaller. A good value for many systems is 10^{-2} eV/Å, but it may sometimes be required to use 10^{-3} eV/Å.
3. If negative frequencies at the saddle still persist, then it is possible the NEB has converged to a second-order saddle point (i.e. a saddle point with two negative frequencies). In this case you should try to build a different initial guess for the NEB configuration to see if it will converge to the correct first-order saddle point.

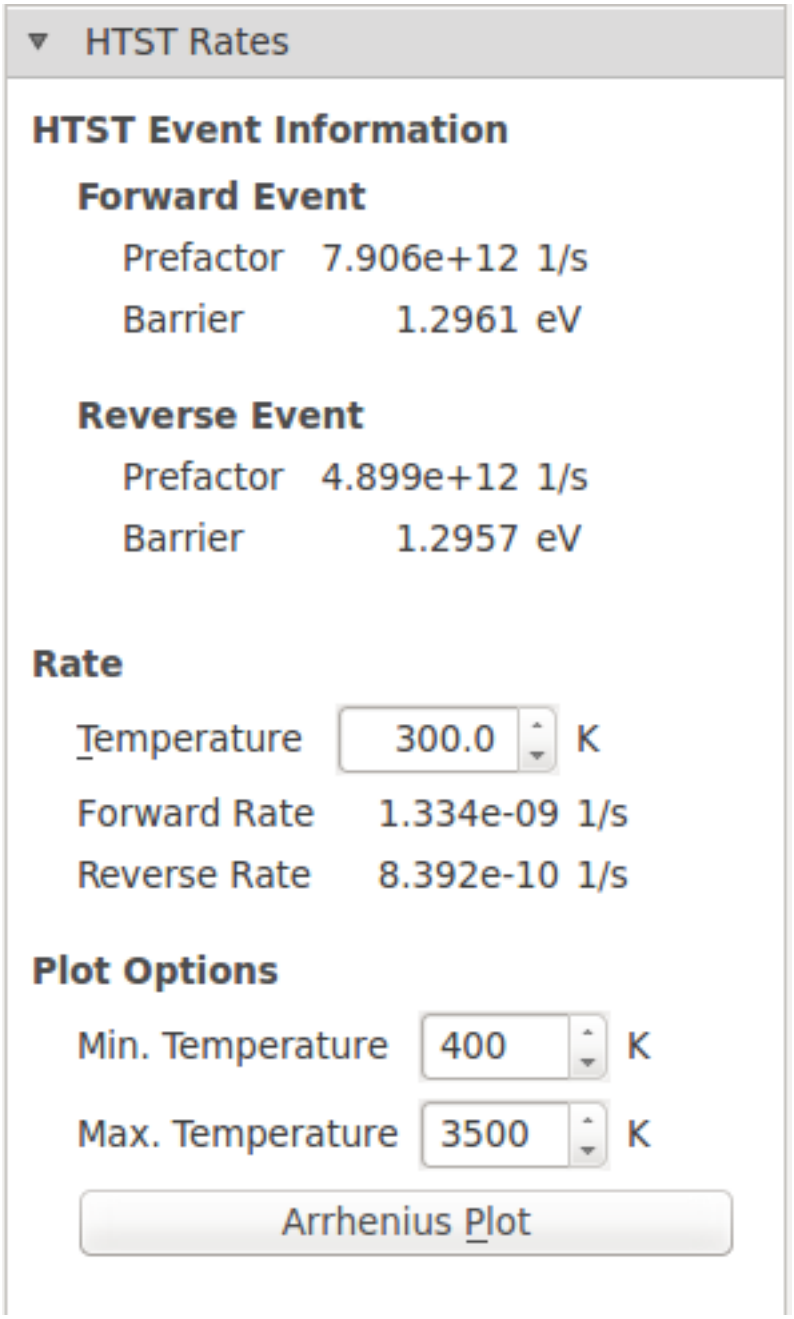
Analyzing the Results

After the calculation finishes the file `pt hop.nc` should contain the initial and converged NEB configurations, as well as the HTSTEvent. Click on the second NEB configuration and click on the *Movie Tool* in the panel on the right.



The potential energy curve shows that the barrier is about 1.3 eV. The movie of the reaction visualizes the expected hopping process. There is a single barrier between the initial and final state. Close the *Movie Tool* window. Select the HTSTEvent object on the LabFloor and click the *HTST Rates* plugin in the panel on the right.

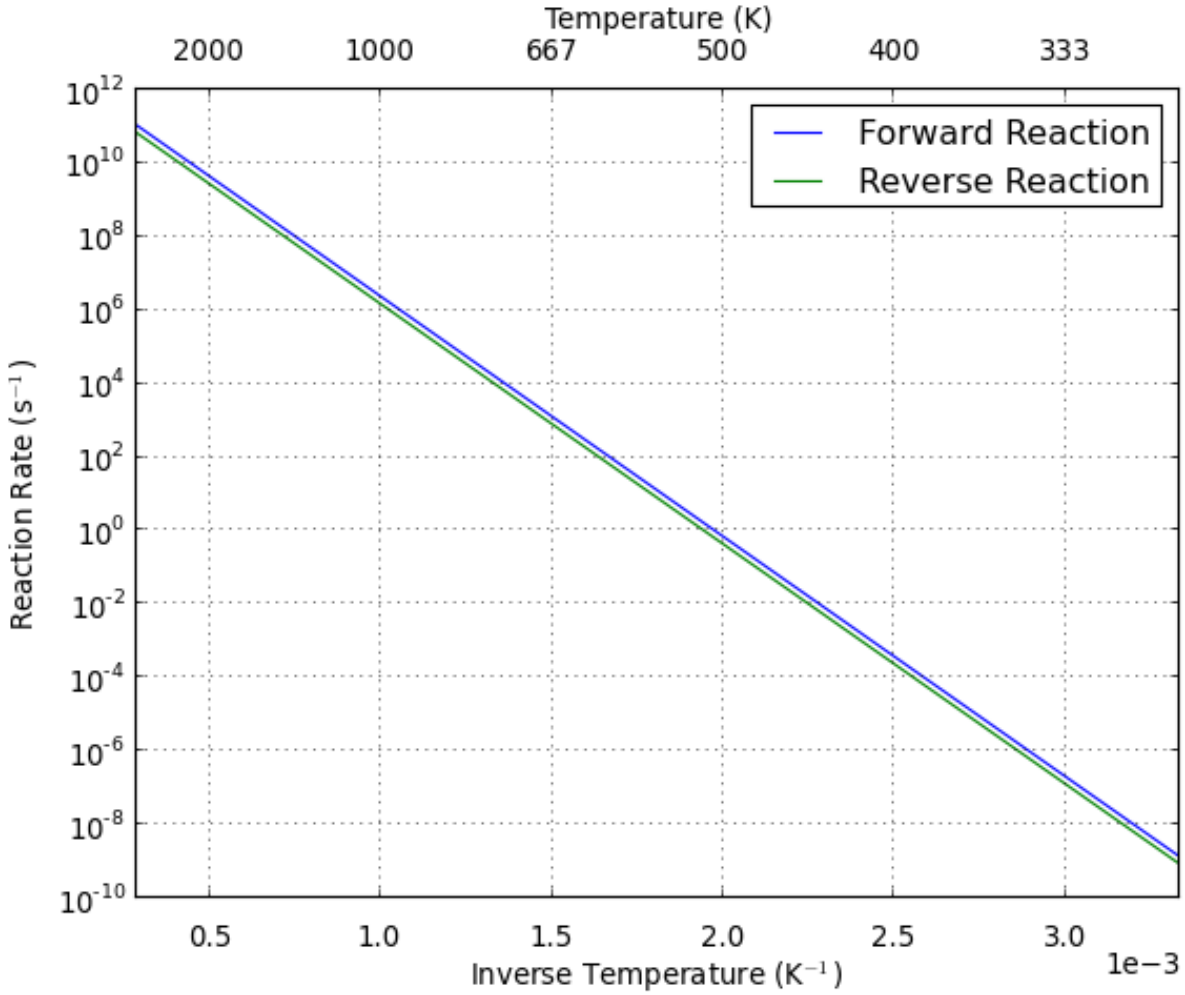
Information about about the reaction such as the barriers and prefactors are shown. The rate can be calculated at any temperature by changing the temperature value, which is 300 K by default.



The forward and reverse barrier and prefactor should be identically theoretically, however, in practice they are not. This is due to small differences in the NEB endpoint geometries since each was minimized from a slightly different starting structure. Also, since this is an ATK-DFT calculation, translational symmetry is not perfectly preserved due to the eggbox effect. This effect describes the relationship between the real-space grids used to represent the charge density and potentials and the positions of the atoms. These grids will be aligned differently with different atoms thus breaking the translational symmetry. The effect is very small for the total energy in this case (< 1 meV), but is more pronounced for the prefactor, where it differs by a factor of 1.6. These errors can be minimized by increasing the *Density mesh cut-off* in the ATK-DFT calculator settings, but it increases the computational cost of the simulation and may not be necessary to get meaningful results.

It is also possible to make an Arrhenius plot to show the temperature dependence of the HTST rate. An Arrhenius plot is the $\log(\text{rate})$ vs $1/T$. When the temperature dependence is plotted this way, the plots will always be straight lines. The slope of the line is $-\Delta E/k_{\text{B}}$ and the y-intercept is the prefactor. From the plot it is easy to interpret that the

prefactor is the maximum possible rate predicted by HTST.

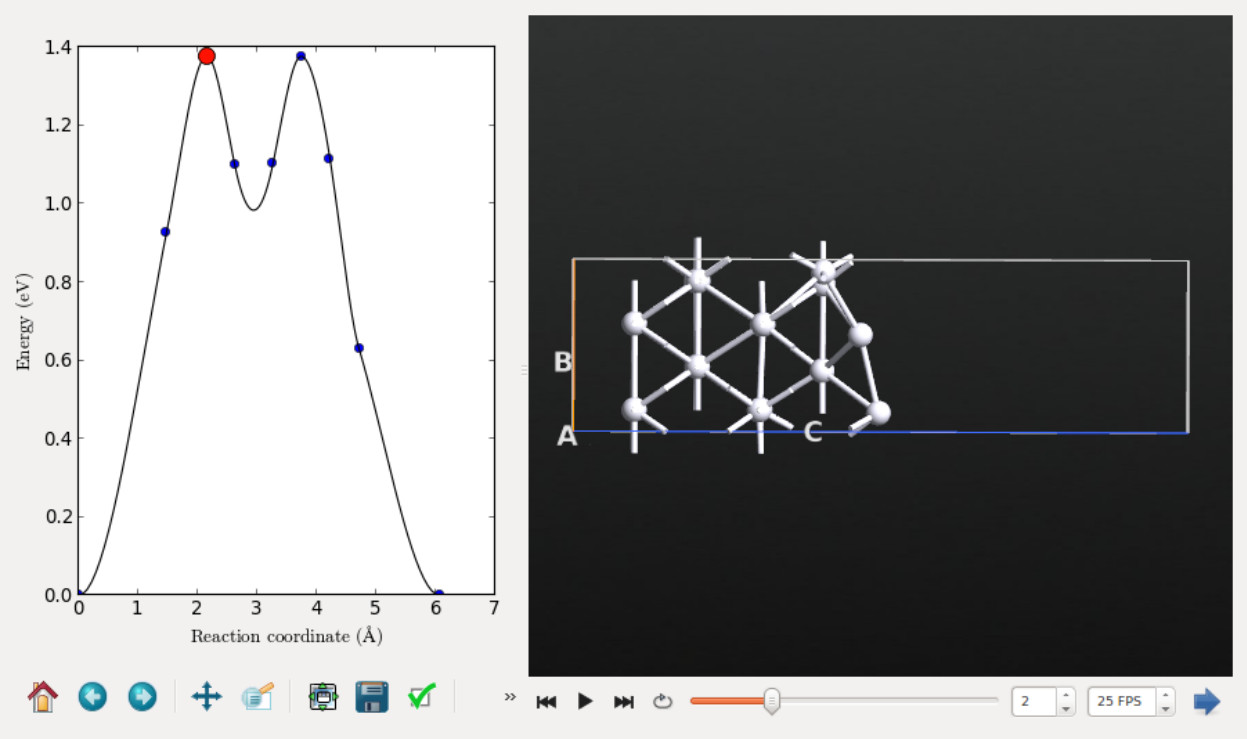


The reaction rate for Pt diffusion via the hopping mechanism is extremely slow at room temperature. The average time between reactions (the inverse of the rate) is on the order of centuries. However, at 600 K we can expect the reaction to occur tens of times per second.

Calculating the Rate for Multiple Elementary Reaction Steps

The exchange mechanism for Pt adatom diffusion on Pt(100) is an interesting case study, because it does not proceed via a single elementary reaction like the hop mechanism does. Details for setting up the calculation are given in another *tutorial*, but the results of the calculation and how to understand the reaction using HTST will be explained here.

The converged CI-NEB calculation shows the following reaction coordinate diagram:



Here you can see that there are two saddle points and thus two reaction steps. The first barrier is about 1.4 eV and the second barrier is about 0.35 eV. Calculating the HTST rate for this reaction would involve splitting the calculation up into two separate NEB configurations to determine the rate constants for each step of the reaction.

Depending on what you are trying to model, it can often be a reasonable approximation to ignore the intermediate state. For example, at 600 K the initial reaction step has a rate of about 1.7 Hz, while the second step has a reaction rate of about 1 GHz ($10^{12}\exp[-0.35eV/600Kk_{\mathrm{B}}]$). Therefore, it may be reasonable to ignore the reaction intermediate.

A more careful study would involve running a kinetic Monte Carlo simulation to determine the kinetics of the system. As soon as more than one reaction step is involved, the reaction kinetics will no longer strictly obey a first order rate law, even if it is often a valid approximation.

References

3.11.11 Calculation of Formation Energies

In this tutorial you will learn how to calculate cohesive and defect formation energies of different systems by total energy calculations. In particular, three examples are shown:

1. GaAs bulk;
2. Ga vacancy in GaAs;
3. Oxygen vacancy on a MgO surface.

Formation energy calculations (or cohesive energy)

In **ATK** you can easily calculate the formation energy (or cohesive energy) of your system from conventional total energy calculations. In particular, you can calculate the formation energy from the total energy of your system and the total energy of its constituent parts:

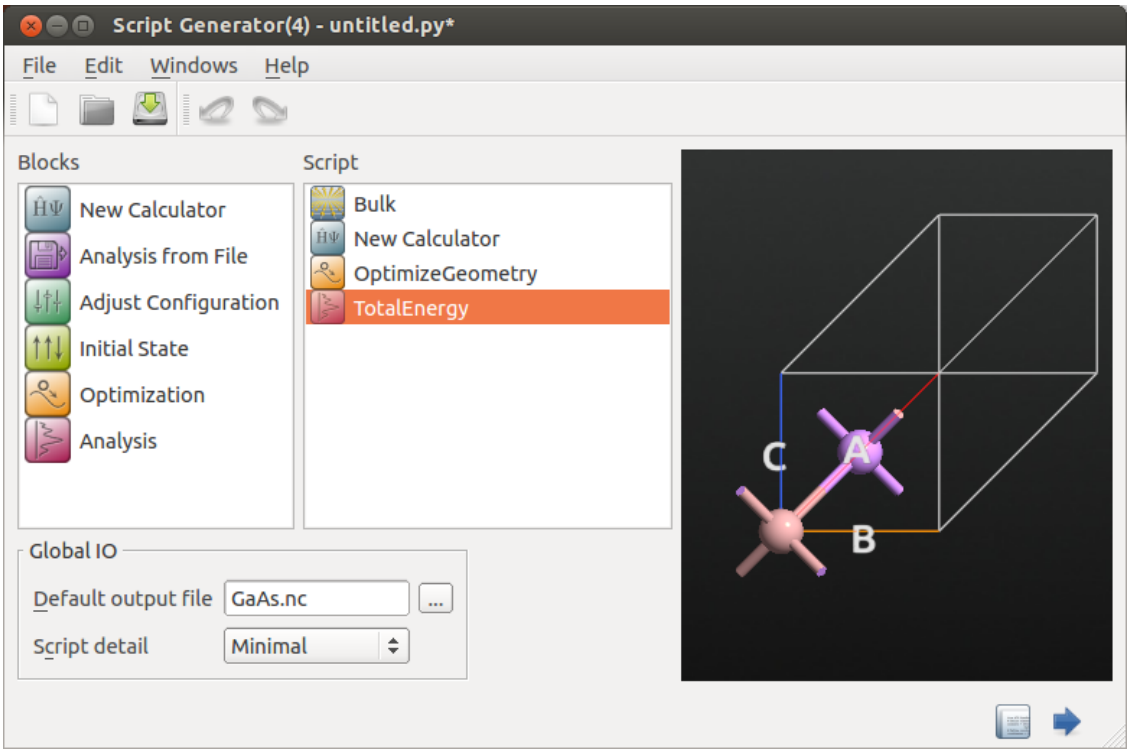
$$E_{form} = E_{tot} \Sigma_x E_{tot}(x). \tag{3.14}$$

E_{form} represents the energy required to dissociate the material into its individual components, x. You should pay particular attention when you choose the reference systems for your individual components, which could be bulk or gas phases.

Note:

- E_{form} is calculated from electronic structure calculations and does not take into account thermal and zero point energy terms (vibrational contributions).
- You may need well converged calculations (mesh cut-off and k-points).
- Remember to normalize the total energy you get by the number of atoms which compose your system.
- Pay attention to spin polarization. In particular if your reference system is a molecule with particular spin state, e.g. O_2.

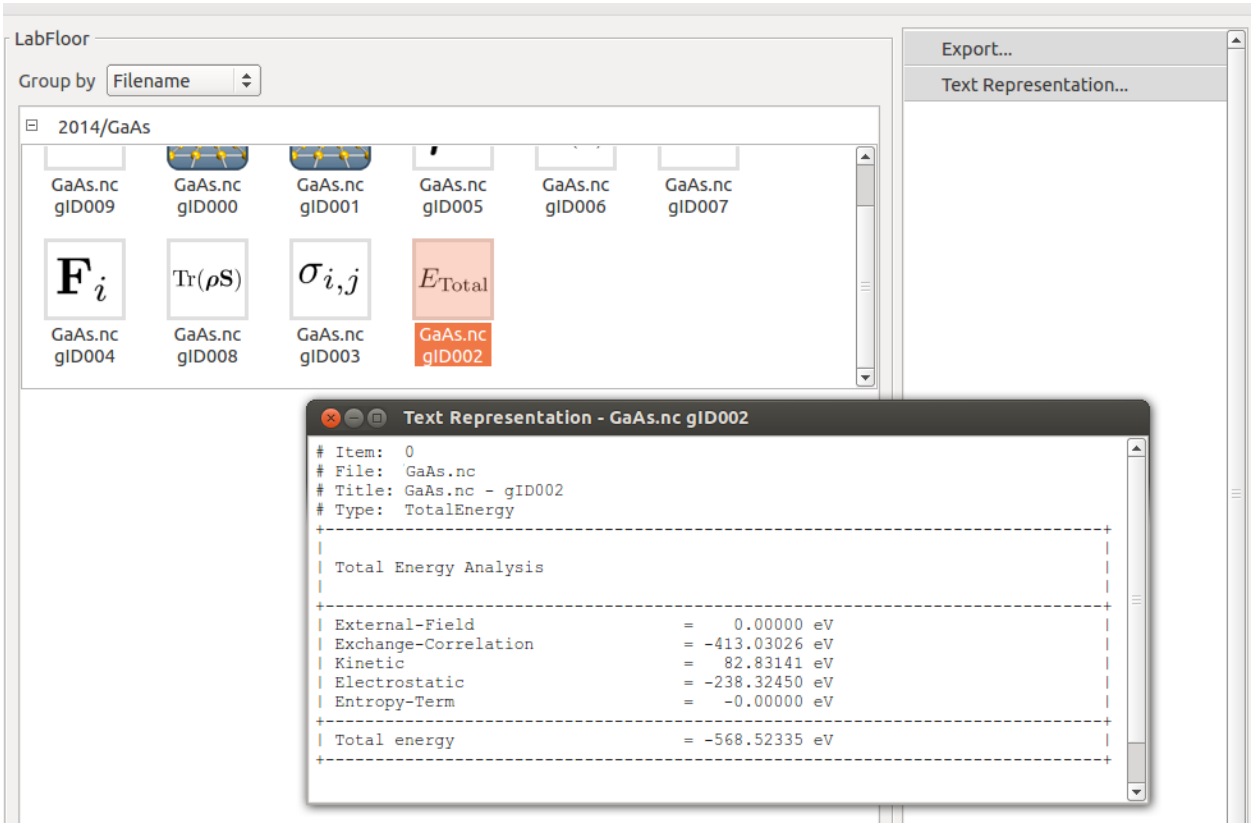
In order to calculate the total energy of your system, from the **Scripter** add a **GeometryOptimization** block and a **TotalEnergy** analysis:



Note: For a detailed explanation of each energy terms contained in the **TotalEnergy** analysis in **ATK-DFT** and

ATK-Huckel for bulk and device configurations, see the notes in *TotalEnergy*.

You can read the total energy of your system, in eV, in your log file and by analyzing the **TotalEnergy** object on the **LabFloor** using the **Text Representation** tool:



Cohesive energy of a bulk system

Let us take the simple example of a bulk system, GaAs, where you can take the bulk phases as reference systems for its individual components, Ga and As.

In this case you can calculate the formation energy of bulk GaAs from Ga and As bulk:

$$E_{form}^{\mathrm{GaAs}} = E_{tot}^{\mathrm{GaAs}} - E_{tot}^{\mathrm{Ga}}/n_{\mathrm{Ga}} - E_{tot}^{\mathrm{As}}/n_{\mathrm{As}},$$

where n_{Ga} and n_{As} are the number of atoms in the Ga and As bulk unit cells, respectively.

Results

With LDA and the FHI DoubleZetaPolarized basis set you will get E_{form}^{GaAs} = -0.6645 eV (exp: -0.73 eV[1]). The negative sign means that 0.6645 eV are gained when forming GaAs from Ga and As bulk.

[1] El-Mellouhi F. and Mousseau N. Self-vacancies in gallium arsenide: An ab initio calculation Physical Review B 7, 125207 (2005)

Attention: The choice of the pseudopotentials and basis set can be crucial for formation energy calculations, and it very much depends on the elements included in your system.

In this particular case, the Ga element and its components are more sensitive to the pseudopotential and basis set definition.

In QuantumATK the HGH and OMX pseudopotentials are also available.

By using the HGH pseudopotentials (Tier 4 basis set) for the calculation of the formation energy of GaAs, you will get E_{form}^{GaAs} = -0.76 eV, in very good agreement with the reported experimental value[Page 1527, 1].

Note: You can apply the same methods for other systems, such as molecular crystals. In this case, the terms $\Sigma_x E_{tot}(x)$ in the equation above corresponds to the total energies of the isolated molecules forming the crystal.

Attention: Be aware that you may need to apply the counterpoise correction to account to the basis set superposition error depending on the basis set. For more details please refer to the *Including the D2 dispersion correction* tutorial.

Note: You can also calculate the molecule formation energy from the total energy of your molecule and the total energy of the isolated atoms forming the molecule. Remember that especially in this case you should run spin-polarized calculations. Be also aware of DFT limitations in describing isolated species, e.g. O atom for the calculation of O_2 formation energy[2].

Defect formation energy calculations

By using the same approach you can calculate defect formation energies from total energy calculations.

Ga vacancy in GaAs

Consider a GaAs crystal with a Ga vacancy. The defect formation energy can be calculated from:

$$E_{form} = E_{tot}^{\mathrm{Ga}_{1-x}\mathrm{As}} E_{tot}^{\mathrm{GaAs}} + x \cdot E_{tot}^{\mathrm{Ga}}, \tag{3.15}$$

where $E_{tot}^{\mathrm{Ga}_{1-x}\mathrm{As}}$ is the total energy of GaAs containing x gallium vacancies, E_{tot}^{GaAs} is the total energy of the perfect GaAs crystal, and E_{tot}^{Ga} is the total energy of bulk Ga.

Note: Besides the usual parameters to check for convergence you should also pay attention to the cell size which is directly related to the defect concentration.

[2] Wang L. et al. Oxidation energies of transition metal oxides within the GGA+ U framework Physical Review B 73, 195107 (2006)

Results

For LDA and FHI DoubleZetaPolarized basis set the **ATK-DFT** formation energy of a gallium vacancy in a 216 atoms GaAs unit cell is 3.15 eV (compared to 2.9 eV in[Page 1527, 1]).

Surface O vacancy on MgO(100)

Finally, let us take the example of an oxygen vacancy on a MgO(100) surface. In this case your reference system is either O_2 in the gas phase or a single oxygen atom. The corresponding formation energies are reported in the table below for a 3x3 MgO(100) surface.

	Formation energy (eV)	
	1/2 O_2	O_2
ATK-PBE	6.82	10.36
ATK-PBE - ghost atom	6.61	10.16
VASP PW91[3]	6.32	9.48

Depending on the extension of your basis set you may need to include the BSSE correction as explained in the *Including the D2 dispersion correction* tutorial. In this case, the first term in Equation (3.15) is the total energy of the vacancy configuration with the atom that form the vacancy as ghost atom (ATK-PBE - ghost atom in the table), instead of actually removing them from the configuration.

References

3.11.12 Elastic constants

QuantumATK provide a very simple way to set up and execute calculations of the **elastic constants for arbitrary bulk configurations**. The method is generalized and can easily be used with density functional theory (ATK-DFT), semi-empirical methods (ATK-SE), or classical potentials (ATK-ForceField). In this tutorial, you will learn how to calculate elastic constants with QuantumATK.

Note: In a bulk solid, the elastic constants relate the linear response of the stress tensor σ to an external strain ε on the system. The elastic constants thereby describe the directional stiffness of the material under specific types of deformations. More general moduli, such as bulk, shear, or Youngs modulus, can also be calculated.

All of these quantities characterize the mechanical properties of a solid. In addition, elastic constants and moduli are often employed as fitting observables for parameterization of classical potentials.

---

[3] Giordano L. et al. F and F+ Centers on MgO/Ag(100) or MgO/Mo(100) Ultrathin Films: Are They Stable? J. Phys. Chem. C 112, 3857 (2008)

Methodology

The stress and strain tensors are always symmetric 3x3 matrices, and they can therefore be expressed more compactly as 6-vectors, using the so-called **Voigt notation**:

$$\boldsymbol{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}),$$

and

$$\boldsymbol{\varepsilon} = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, 2\varepsilon_{yz}, 2\varepsilon_{xz}, 2\varepsilon_{xy}).$$

The linear response of the stress vector to a given strain vector can then be written as

$$\boldsymbol{\sigma} = \boldsymbol{C} \cdot \boldsymbol{\varepsilon},$$

where the symmetric 6x6 matrix $\boldsymbol{C}$ contains the **elastic constants**.

Depending on the crystal symmetry, the number of independent entries in $\boldsymbol{C}$ can be reduced further. For instance, in a cubic crystal only three entries, C_{11}, C_{12}, and C_{44}, are independent:

$$\begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{11} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{12} & C_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & C_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & C_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & C_{44} \end{pmatrix} \cdot \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \\ \varepsilon_5 \\ \varepsilon_6 \end{pmatrix}$$

To obtain the elastic constants, one should apply small deformations, η, to the simulation cell along selected strain vectors, and calculate the resulting stress vectors. The linear stress contribution is obtained by fitting the $\sigma_i(\eta)$ curves of each Voigt stress component and for every strain vector. The independent elastic constants are then calculated as the least-squares solution to a linear system of equations, taking the crystal symmetry into account.

Note: For the calculation of elastic constants, QuantumATK employs the Lagrangian strain and stress tensors instead of the corresponding physical tensors.

To minimize the number of stress calculations, QuantumATK uses the Universal Linearly-Independent Coupling Strain (ULICS) vectors (Ref.[1]). For each strain vector, typically three deformations ($-\eta$, 0, $+\eta$), centered at the reference configuration ($\eta = 0$), are applied.

Important: For configurations with more than one atom in the cell, the atomic positions of each strained cell must be **optimized** before the stress is calculated; QuantumATK automatically takes care of this. You should however ensure that the cell vectors (and atomic positions) of the reference configuration are well optimized **before** starting the calculation of the elastic constants.

---

[1] R.ăYu, J.ăZhu, and H.Q. Ye. Calculations of single-crystal elastic constants made simple. *Computer Physics Communications*, 181(3):671–675, 2010. doi:10.1016/j.cpc.2009.11.017.

Calculating elastic constants using classical potentials

You will now perform an elastic constant calculation based on classical potentials, using **ATK-ForceField**. The advantage of of classical potentials, apart from the calculation speed, is that that most properties, such as energies, forces and stress, are smooth functions of the coordinates. This makes calculation of elastic constants particularly robust with respect to the setting sused for calculating the strain.

Two examples will be considered:

- A bulk silicon crystal, using the Stillinger–Weber potential[2].
- SiO_2 α-quartz, using the Pedone2006 Fe2 potential[3].

Bulk silicon

Open the QuantumATK **Builder** and use *Add* → *From Database* to add Silicon (alpha) to the Stash. Then send the configuration to the **Script Generator** and do the following:

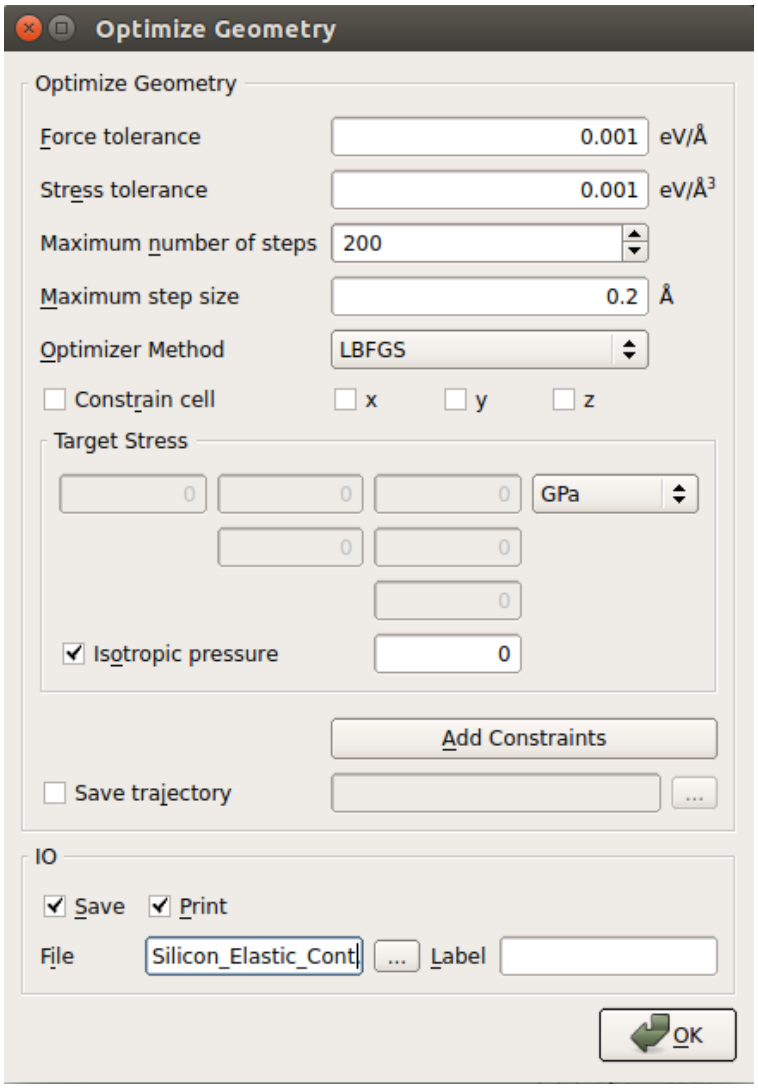
- Add a New Calculator block.
- Select **ATK-ForceField** and the Stillinger_Weber_Si_1985 potential.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

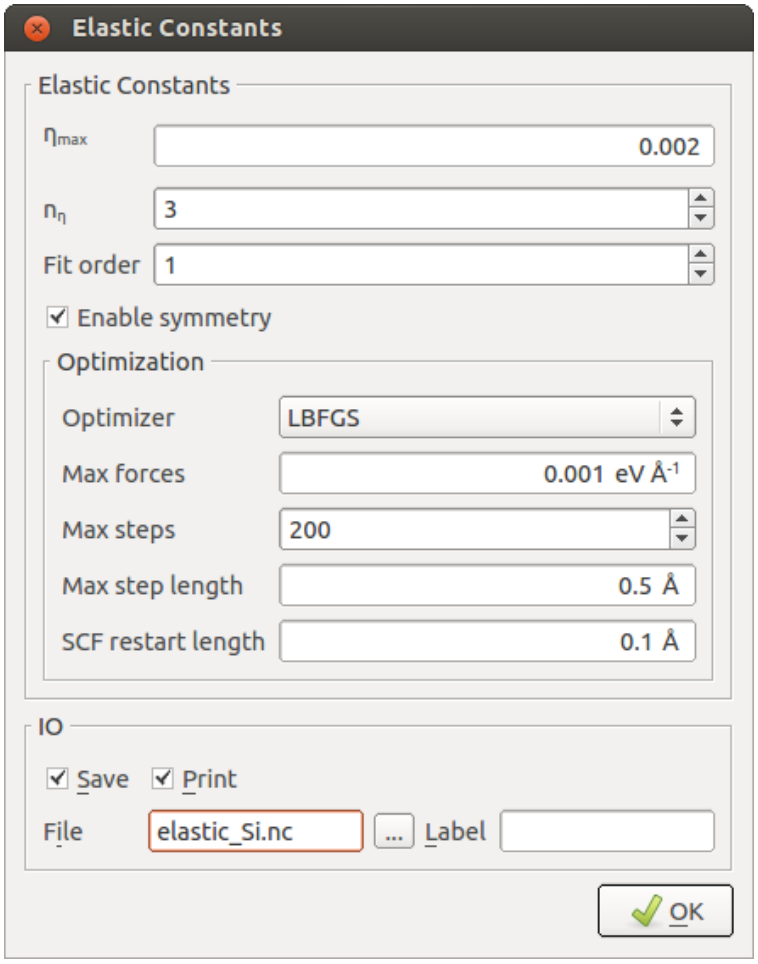
- Add a *Optimize* → *OptimizeGeometry* object to the script and edit it:
  - select sufficiently accurate settings for maximum force and maximum stress, e.g. 0.001 eV/Å for the force and 0.001 eV/Åsup:*3* for the stress;
  - uncheck the Constrain cell check box to enable optimization of the cell vectors;
  - select **LBFGS** as the optimizer method.

[2] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

[3] Pedone Alfonso, Malavasi Gianluca, Menziani M.ăCristina, N.ăCormack Alastair, and Segre Ulderico. A New Self-Consistent Empirical Interatomic Potential Model for Oxides, Silicates, and Silica-Based Glasses. *The Journal of Physical Chemistry B*, 110(24):11780–11795, 2006. doi:10.1021/jp0611018.



•Back in the Scripter, add an *Analysis* → *ElasticConstants* block Open the block to set up the analysis.



The first set of parameters and options refer to the stress/strain calculation:

Parameter: η_{max}
Specifies the maximum deformation magnitude applied to the cell. A sufficiently large value should be chosen to achieve a significant variation in stress, but small enough to remain essentially within the linear regime. The default value of 0.002 is in most cases a good choice. Elastic constants calculated with classical potentials are in general not too sensitive to this parameter.

Parameter: n_η
Specifies the number of intermediate deformations between $-\eta_{max}$ and $+\eta_{max}$ for every strain vector. A higher value, together with a higher-order fit (as specified by the next parameter), may help to filter out possible non-linear contributions, but also increases the number of calculations that need to be performed. The default value is 3 and usually works well.

Parameter: ***Fit order***
Specifies the highest polynomial order in the stress vs. η fitting procedure, which should be smaller than n_η. For evaluation of the elastic constants, only the linear contribution is used.

Option: ***Enable symmetry***
Allows QuantumATK to detect the crystal symmetry and calculate only the independent constants for this lattice symmetry. Otherwise, all 21 constants of the upper triangle of the $\boldsymbol{C}$ matrix are treated as independent.

The remaining settings are related to the force optimization (of the atomic coordinates) that can be invoked before the stress of each strained system is calculated:

Option: ***Optimizer***

Sets the optimizer method. If you select None, no force optimization is carried out before the stress is calculated, which is the fastest, but least accurate option. If there is only one atom in the unit cell, there is no need for an optimization of the internal coordinates and the optimizer is **automatically disabled**.

Parameters for force and stress minimization
The remaining parameters control the optimizer setttings. These options are in fact the same as in the Optimize-Geometry object. While most of these parameters are only relevant in special cases, the Max forces parameter should be chosen sufficiently accurate. The default value of 0.005 eV/Å (which is 10x lower than the usual default in QuantumATK) represents a good balance between accuracy and efficiency. Since the example at hand uses fast ATK-ForceField calculations, you can easily afford to reduce this value even further to 0.001 eV/Å to achieve a higher precision.

Execute the calculation by sending the script to the **Job Manager**, again using the button. You may be asked to save the script again. Click the button to start the job. It takes around five seconds to finish on a laptop.

Analysis of the results

You can find information about the calculation and results in the *log file**. As you see, QuantumATK has correctly recognized the cubic symmetry of the FCC silicon crystal and calculates only the three independent elastic constants:

```
+-------------------------------------------------------------------------------+
| Calculating elastic constants for the given bulk configuration                |
+-------------------------------------------------------------------------------+
| Detected space group number:    227                                           |
| Detected lattice symmetry:      Cubic                                         |
| This lattice symmetry has 3 independent elastic constants:                    |
| C11  C12  C44                                                                 |
+-------------------------------------------------------------------------------+
```

The elastic constants matrix is printed at the bottom of the log file, and is in excellent agreement with literature results using the Stillinger–Weber potential[4]:

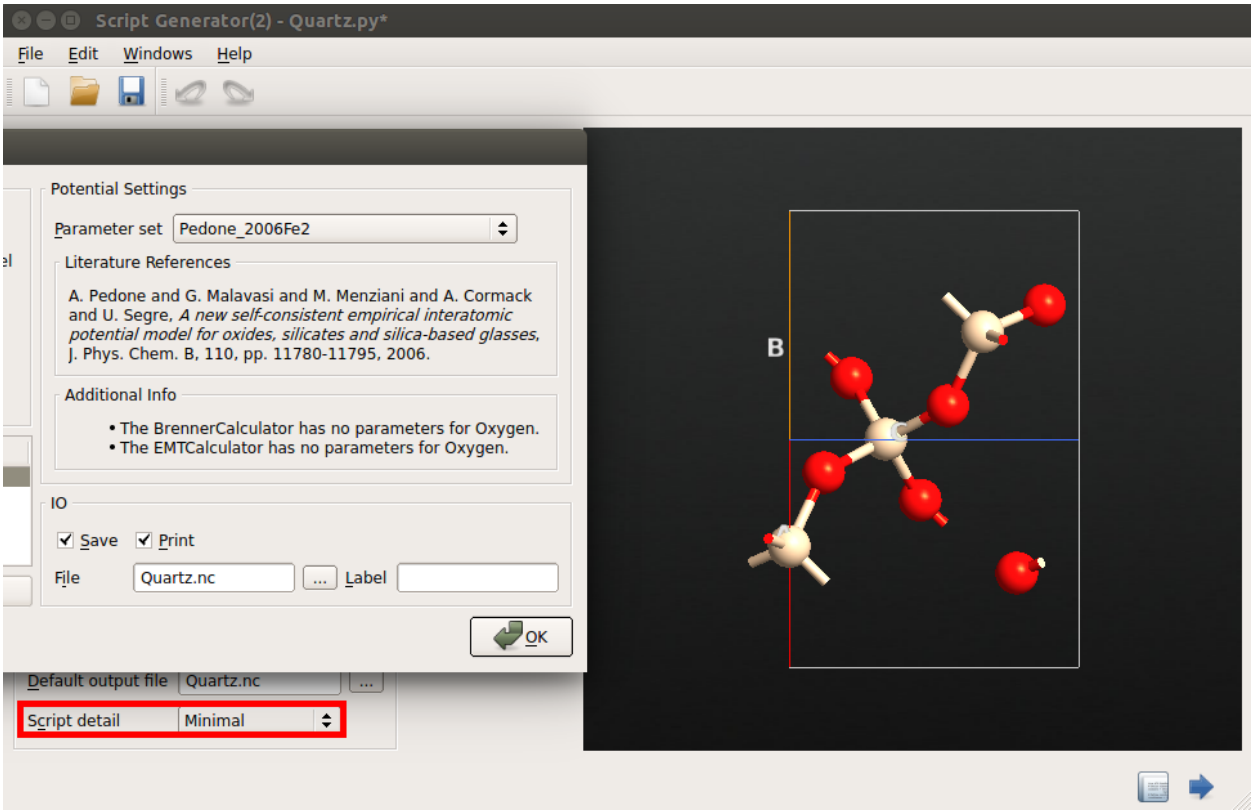
$$C_{11} = 151.4\,\mathrm{GPa}, \quad (3.16)$$
$$C_{12} = 76.4\,\mathrm{GPa},$$
$$C_{44} = 56.4\,\mathrm{GPa}.$$

Tip: The log file also records more detailed analysis of the elastic constant matrix. It contains the elastic compliance matrix, which is the inverse of the elastic constants matrix. More general moduli, such as bulk, shear or Youngs modulus, are also calculated from the elastic constants. Be aware tha there are three different definitions of bulk and shear modulus, according to the slightly different formulae from Voigt and Reuss. All of them are listed in the log output for comparison.

[4] E.ăRoger Cowley. Lattice Dynamics of Silicon with Empirical Many-Body Potentials. *Phys. Rev. Lett.*, 60:2379–2381, Jun 1988. doi:10.1103/PhysRevLett.60.2379.

SiO_2 quartz

You can use the same method to compute the elastic properties of more complex crystals, such as SiO_2 α-quartz. Follow the same steps above and select the Pedone2006_Fe2 potential , which is from Ref.[Page 1531, 3].



Accurate long-range electrostatics

After setting up the script in the Script Generator, change the **Script Detail** setting in the Global IO field from Minimal to Show defaults, and transfer the script to the Editor in order to make one small modification. In the calculator block, you will find that all the pair potentials are now individually listed and added to the potential set (this is a result of the Show defaults setting).

The Pedone potential includes long-range electrostatic interations. You can manually increase the cutoff distance for these interactions (`r_cut`). To achieve a high accuracy of the stress calculation, you should change this parameter from the default of 9.0 Å to a larger value of 15 Å, to account for the long-range nature of the Coulomb potential:

```
potentialSet.setCoulombSolver(CoulombDSF(r_cut=15.0*Angstrom, alpha = 0.2))
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")
```

This will affect the speed of the calculations, but since the elastic constants calculation does not involve extensively long simulations, you will hardly notice it.

Results

Performing the calculation will give the following results:

```
+-------------------------------------------------------------------------------+
| Elastic Constants in GPa                                                      |
+-------------------------------------------------------------------------------+
|    86.55       8.71      11.16     -18.36       0.00       0.00              |
|               86.55      11.16      18.36       0.00       0.00              |
|                         106.67       0.00       0.00       0.00              |
|                                      49.41       0.00       0.00              |
|                                                  49.41     -18.36              |
|                                                             38.92              |
+-------------------------------------------------------------------------------+
```

Due to the rhombohedral symmetry, there are six independent constants, $C_{11}, C_{12}, C_{13}, C_{14}, C_{33}$ and C_{44}. All values reproduce very well the results of the original publication, Ref.[Page 1531, 3].

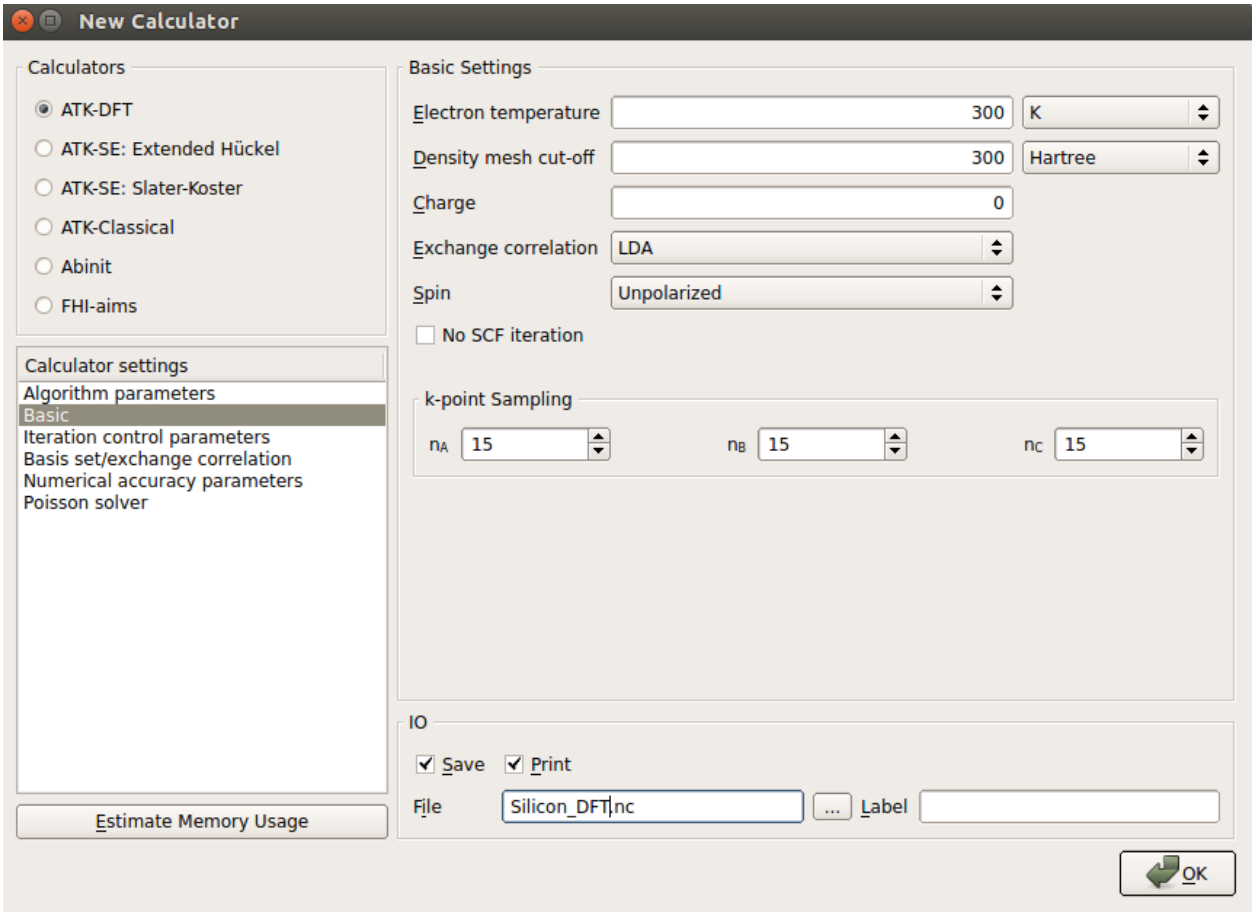
Calculate elastic constants using DFT

Finally, you will perform a calculation of the elastic constants of silicon using density functional theory. The framework outlined above can be used again – you just need to replace the calculator such that is uses **ATK-DFT with GGA exchange correlation**.

Use again the Builder to create the silicon structure and send it to the Script Generator. Add the **New Calculator** , **OptimizeGeometry** and **ElasticConstants** blocks.

To obtain numerically precise stress differences between the strained conformations, we generally have to employ some slightly tighter calculator settings than the defaults:

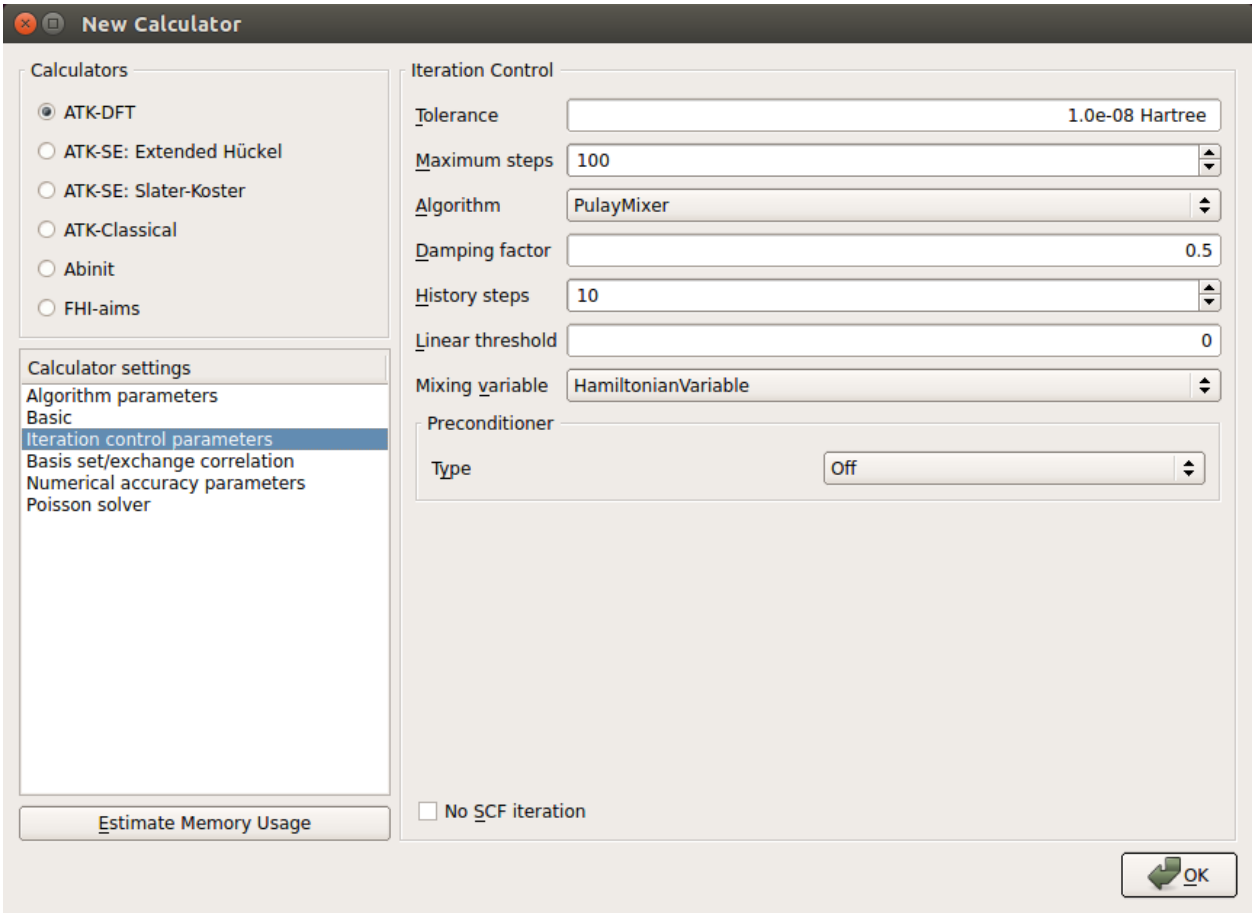
- Choose **ATK-DFT** as the calculator, and set the following parameters in **Basic**:
  - set the grid mesh cut-off to 300 Hartree;
  - set k-point sampling to 15x15x15.



•In the **Iteration control parameters** tab:

–set the tolerance parameter to 1.0e-08;

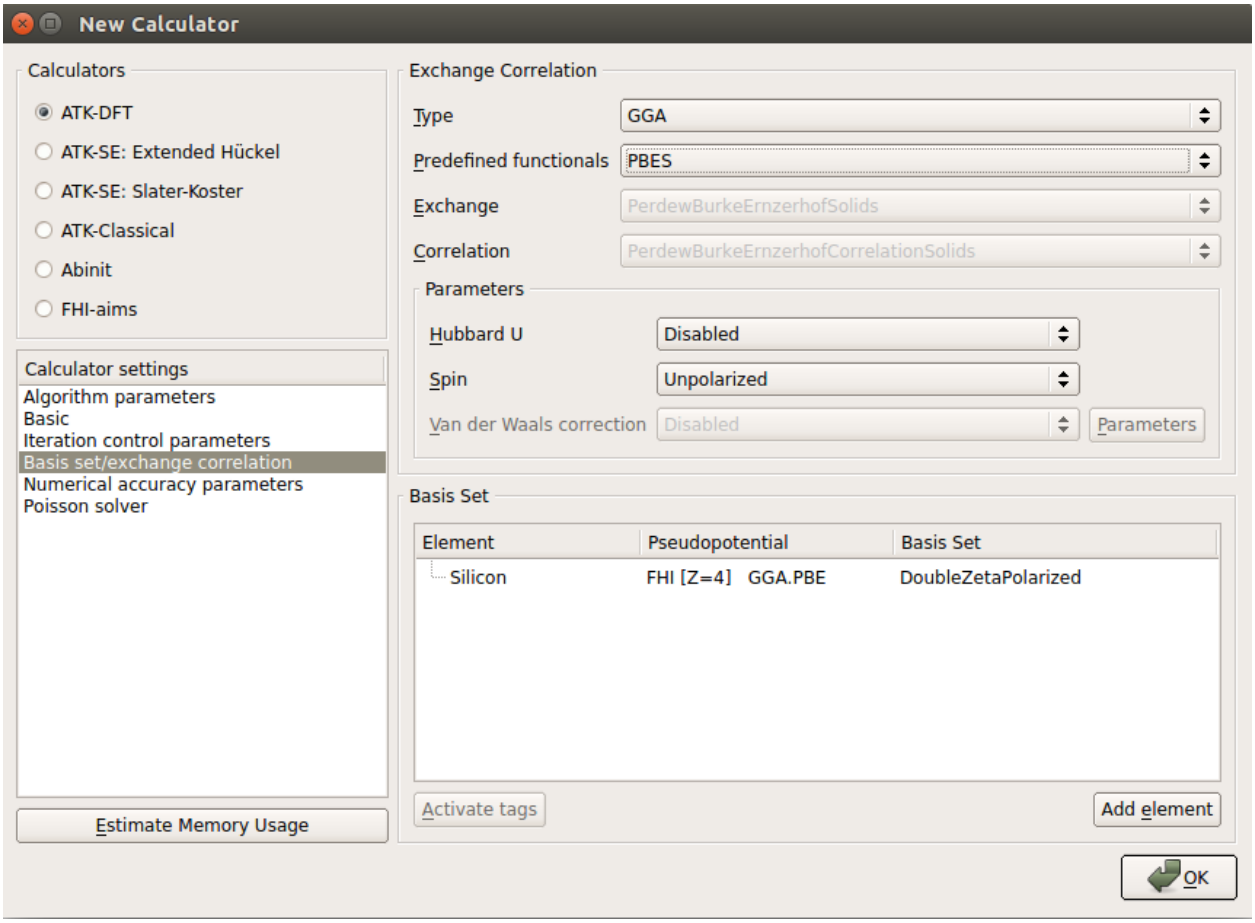
–set the damping factor to 0.5 and the number of history steps to 10.



- In the **Basis set/exchange correlation** tab use the PBEsol functional, which offers very reliable DFT predictions of bulk moduli and related quantities:

  – choose **GGA** for the exchange-correlation type;

  – select in **PBES** in the list of predefined functionals;



The settings in the **OptimizeGeometry** and **ElasticConstants** blocks should be almost identical to the ones chosen for the classical potentials, except that the η_{max} value in the **ElasticConstants** settings should be 0.003 (which means that a larger strain will be employed, resulting in a more pronounced change in the stress).

This calculation takes only about 10 minutes to finish. The calculated elastic constants are

```
+-------------------------------------------------------------------------------+
| Elastic Constants in GPa                                                      |
+-------------------------------------------------------------------------------+
|   161.02      65.22      65.22       0.00       0.00       0.00              |
|              161.02      65.22       0.00       0.00       0.00              |
|                        161.02       0.00       0.00       0.00              |
|                                    75.68       0.00       0.00              |
|                                               75.68       0.00              |
|                                                          75.68              |
+-------------------------------------------------------------------------------+
```

The results are in reasonably good agreement with the experimental values, Ref.[5], which are listed below:

$$C_{11} = 167.4\,\text{GPa} \tag{3.19}$$
$$C_{12} = 65.2\,\text{GPa}$$
$$C_{44} = 79.6\,\text{GPa}$$

[5] H.ăJ. McSkimin, W.ăL. Bond, E.ăBuehler, and G.ăK. Teal. Measurement of the Elastic Constants of Silicon Single Crystals and Their Thermal Coefficients. *Phys. Rev.*, 83:1080–1080, Sep 1951. doi:10.1103/PhysRev.83.1080.

References

3.11.13 Geometry optimization: CO/Pd(100)

Version: 2015.1

In this tutorial you will learn how to do a geometry optimization using *Fixed* and *Rigid* constrains on the atoms. As an example, you will study adsorption of the CO molecule on a Pd(100) surface. You will use the ATK-DFT calculator engine and a standard GGA functional.

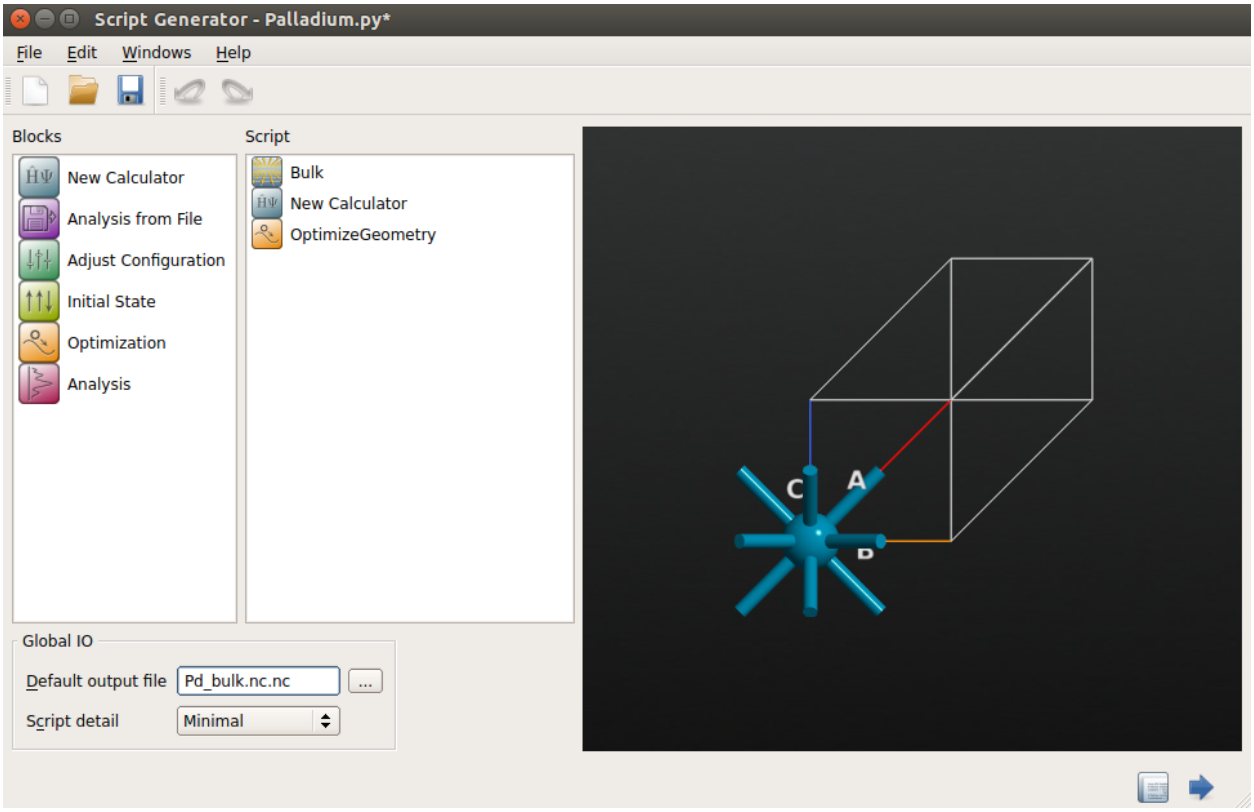
Specifically, you will:

1. optimize the bulk palladium crystal;
2. cleave the bulk to create the Pd(100) surface and relax it;
3. adsorb the CO molecule in the surface, and relax the system in two steps;
   1. first apply *Rigid* constraints on atoms during an initial relaxation;
   2. then apply *Fixed* constraints on the bottom Pd atoms in a final optimization.
4. lastly, relax the CO molecule and calculate the adsorption energy.

Bulk palladium

Create a new project called Pd100_CO and open the **Builder** .

- Use *Add* → *From Database* to import the Pd bulk to the Stash and send the configuration to the **Script Generator** using the send to button .
- In the Script Generator, add the **New Calculator** and **OptimizeGeometry** blocks to the script by double-clicking them, and change the default output file name to `Pd_bulk.hdf5`.



•Double-click the **New Calculator** block and set the following parameters in the window that opens up:

–*k-point sampling*: 9x9x9;

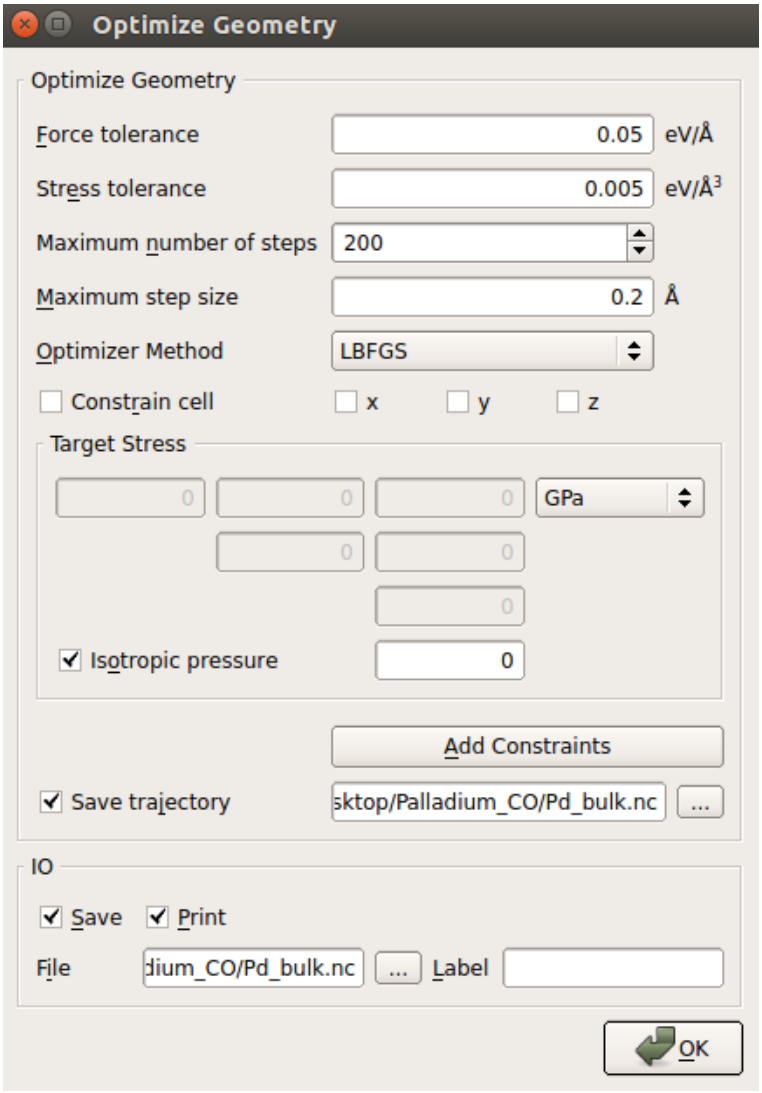
–*exchange-correlation*: GGA-PBE functional.

•Open the **OptimizeGeometry** block, and set the parameters as shown in the image below. In particular:

–enable stress relaxation by unticking *Constrain cell*;

–decrease the stress tolerance to 0.005 eV/Å^3;

–save the relaxation trajectory by ticking *Save trajectory* and enter `Pd_bulk.hdf5` for the trajectory file name.

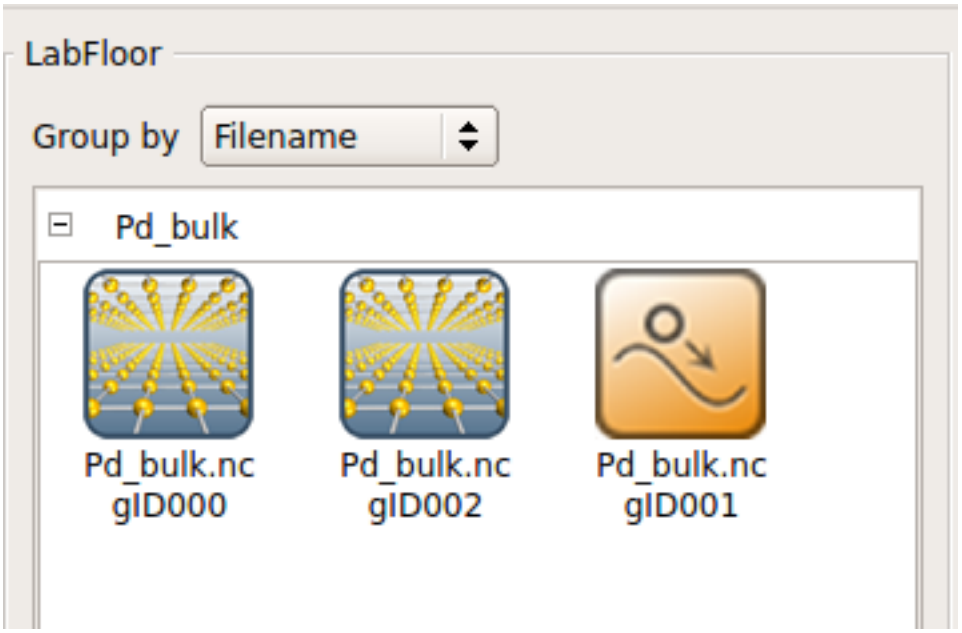


Tip: It is usually a good idea to save the trajectory file. In case your geometry optimization is interrupted you can simply restart it from one of the latest images in the trajectory. If you would like to know more about how to restart a stopped calculation, check out the tutorial *Restarting a stopped calculation*.

•Click *OK* and the OptimizeGeometry window will close.

•The script is now finished. Send it to the **Job Manager** using the button. Save the script in the window that appears and run the calculation. It will take a few seconds. If needed, you can download the script here: `Pd_bulk.py`.

Once the bulk relaxation is finished, the calculation output will appear on the LabFloor. The bulk configurations with IDs *glD000* and *glD002* are the initial and final structures, respectively, while the relaxation trajectory has ID *glD001*.



You can drag and drop these objects to the **Viewer** to visualize them. In the Viewer, you can check the lattice constant by hovering the mouse on the cell boundaries. You could also check the lattice constant in the newly formed log file.

In the following, you should of course use the relaxed bulk structure, *glD002*.

Build the Pd(100) surface and relax it

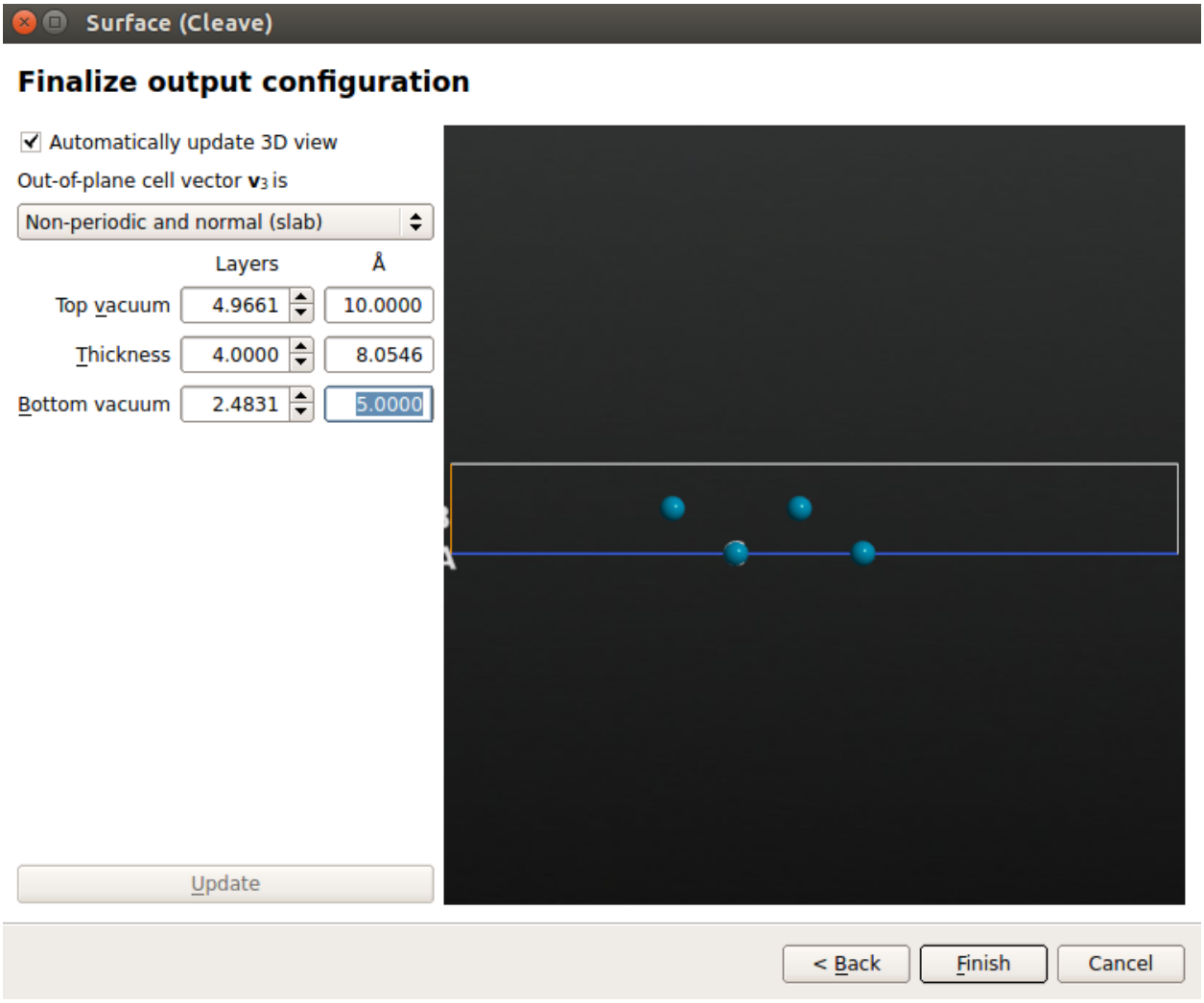
•Drag and drop the relaxed bulk configuration to the **Builder** .

•Open the *Builders* -→ *Surface (Cleave)* plug-in.

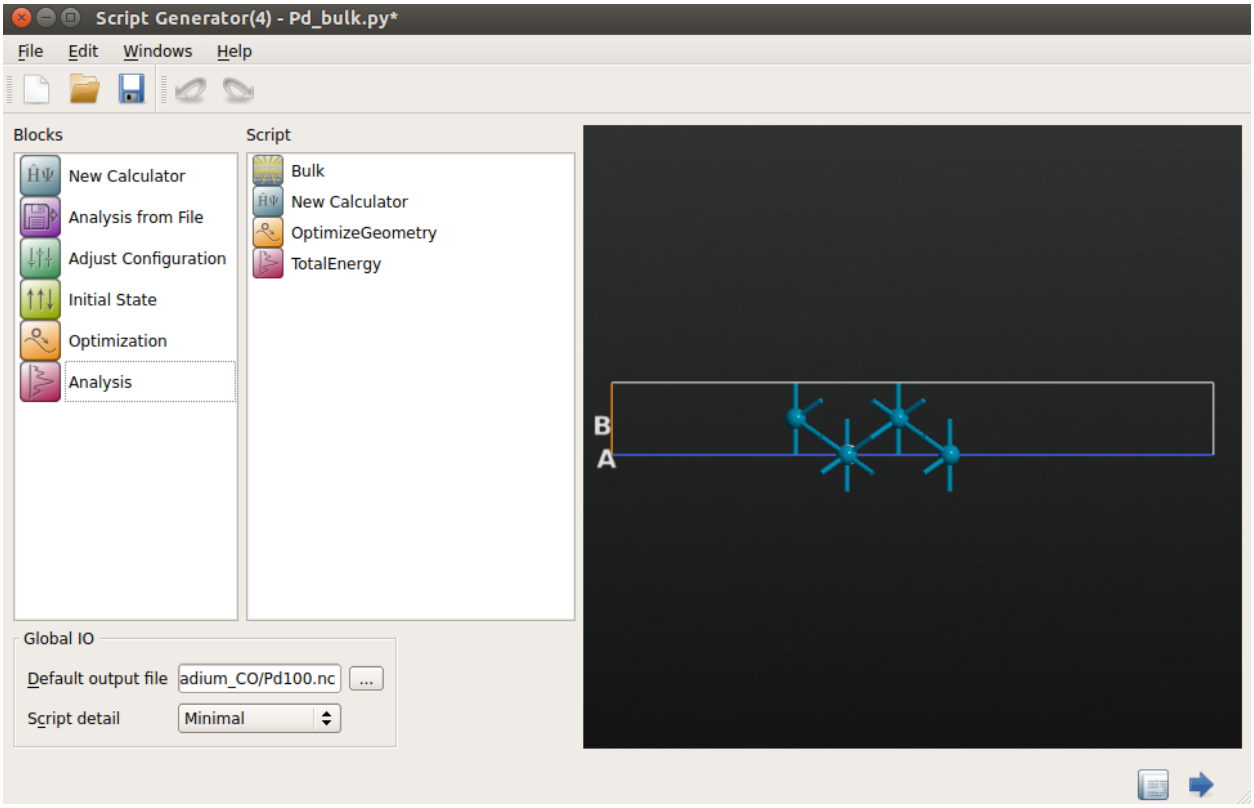
•Keep the default Miller indices and click *Next*.

•Keep the default 1x1 surface lattice and click *Next*.

•Choose a *Non-periodic* and *normal (slab) out-of-plane cell vector*, 4 slab metal layers, and 10 Å and 5 Å for the top and bottom vacuums, respectively. Click *Finish* to build the surface.



•Send the slab structure to the **Script Generator** and add the **New Calculator**, **OptimizeGeometry** and **TotalEnergy** blocks to the script, and set the default output file name to `Pd100.hdf5`.

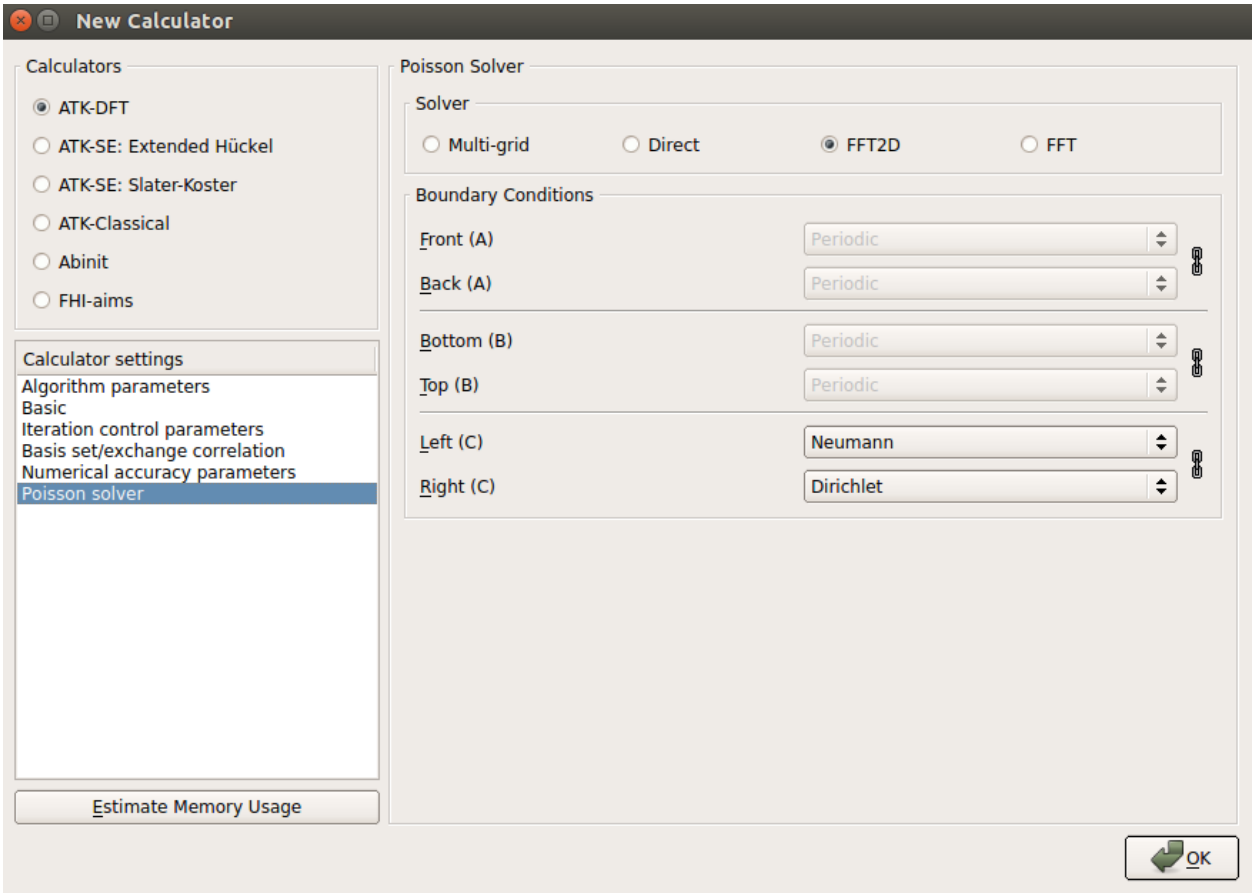


•Choose the following set of calculator parameters:

–*k-point sampling*: 9x9x1 k-points;

–*exchange-correlation*: GGA-PBE;

–*FFT2D Poisson solver* with *Neumann* boundary conditions under the slab and *Dirichlet* boundary conditions above the slab:



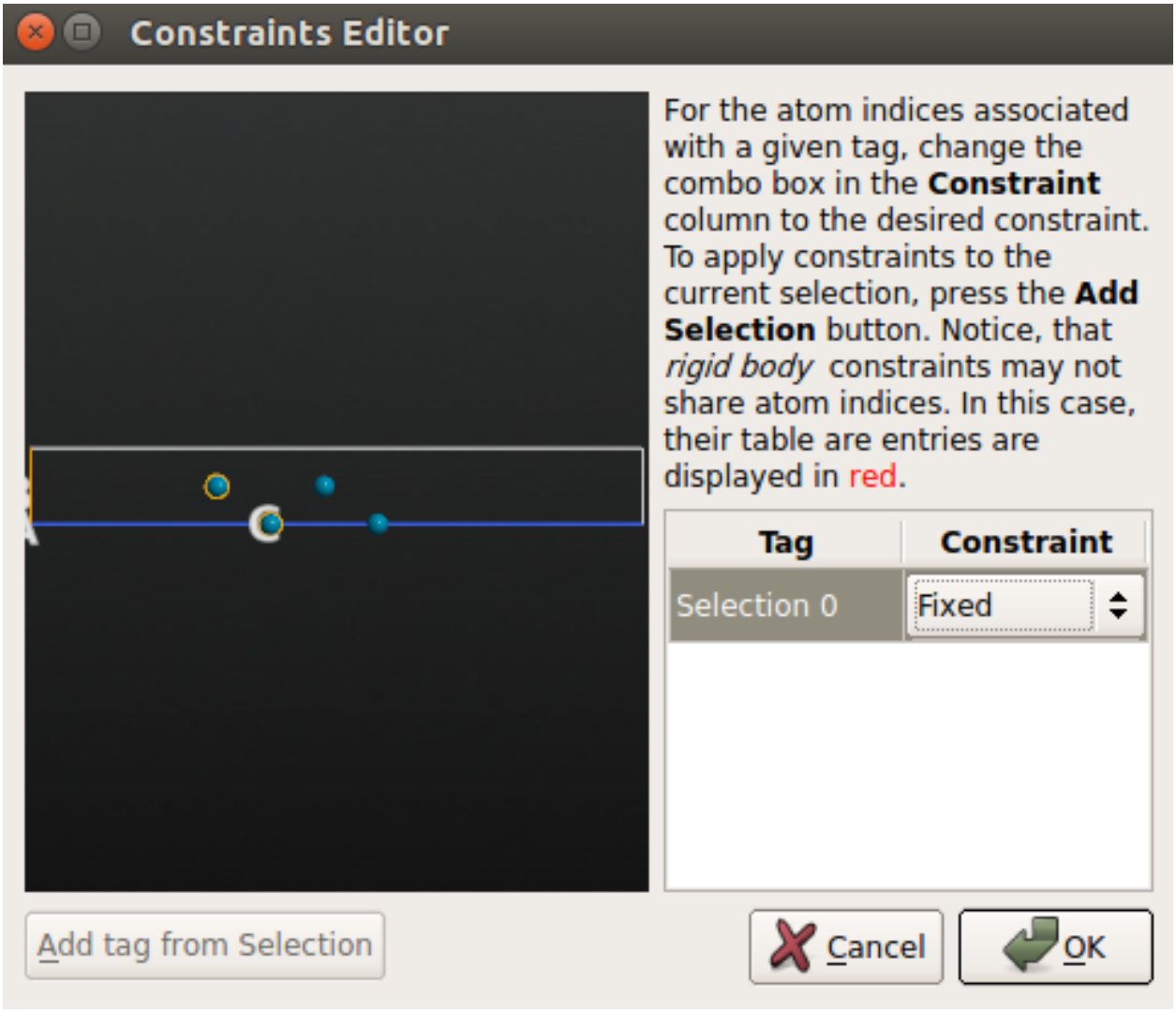
Note: This particular set of boundary conditions along the C direction of the simulation cell is appropriate for slab calculations, where the vacuum above the slab could in principle extend to infinity.

•Open the **OptimizeGeometry** block and set it up for force relaxation:

–save the trajectory to `Pd100_traj.hdf5`;

–click **Add Constraints** to open the Constraints Editor;

–use the mouse to select the bottom two layers of the Pd(100) surface and click *Add tag from selection*. Those two atoms are now tagged Selection 0. Then choose the *Fixed* constraint for the Selection 0 group.



•There is nothing to edit in the **TotalEnergy** block, so the script is now finished. Save it as `Pd100.py` and execute it using the **Job Manager** . The calculation will take roughly one minute. If needed, you can download the script here: `Pd100.py`.

Once the Pd(100) surface has been geometry optimized, the results will appear on the **LabFloor**. You can again visualize the relaxation trajectory using the **Viewer** .

Tip: You can check the total energy of the relaxed Pd(100) surface by selecting the TotalEnergy object and clicking on the **Text Representation** plugin. The energy is also printed in the log file.

Relax the CO/Pd(100) system

The next step is to add the CO molecule to the surface and relax the system. You should use the CO/Pd(100) structure available here: `CO_Pd100.py`. If needed, you can learn how to add a molecule to a surface from the tutorial *Building molecule–surface systems: Benzene on Au(111)*.

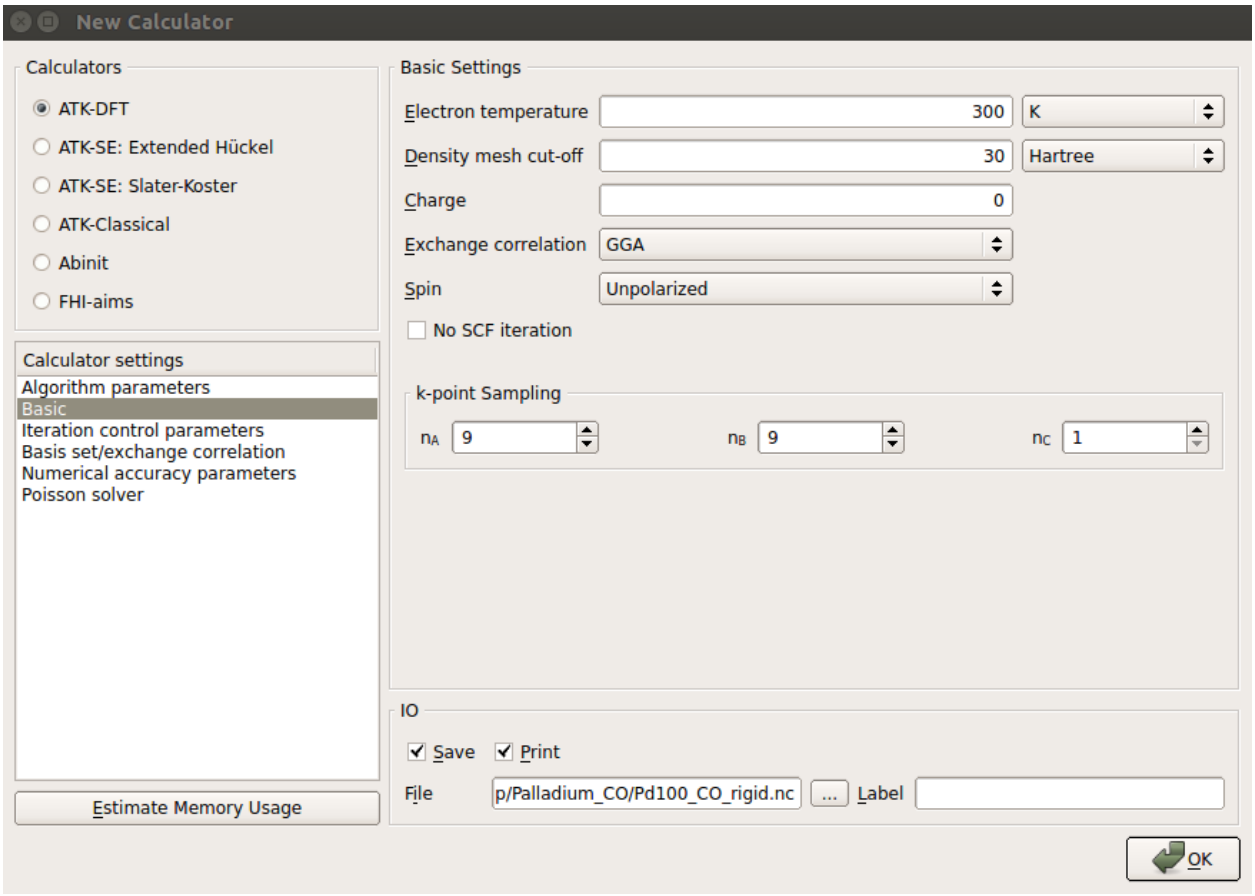
As mentioned in the introduction, you will relax the CO/Pd(100) configuration in two steps:

1. use a *Rigid* constraint for CO and *Fixed* constrain for the surface, together with a relatively low electron density mesh cut-off, in order to get a quick idea of where the CO will be adsorbed;
2. the final relaxation step does not constrain the CO molecule and just fixes the bottom of the Pd(100) slab.

Rigid relaxation

Transfer the provided CO/Pd(100) configuration (`CO_Pd100.py`) to the **Script Generator** and add the **New Calculator** and **OptimizeGeometry** blocks. Enter `Pd100_CO_rigid.hdf5` for the default output file name, and then edit the script:

- In the **New Calculator** block, use the same settings as for the Pd(100) relaxation in the previous section, but decrease the density mesh-cut off to 30 Hartree.

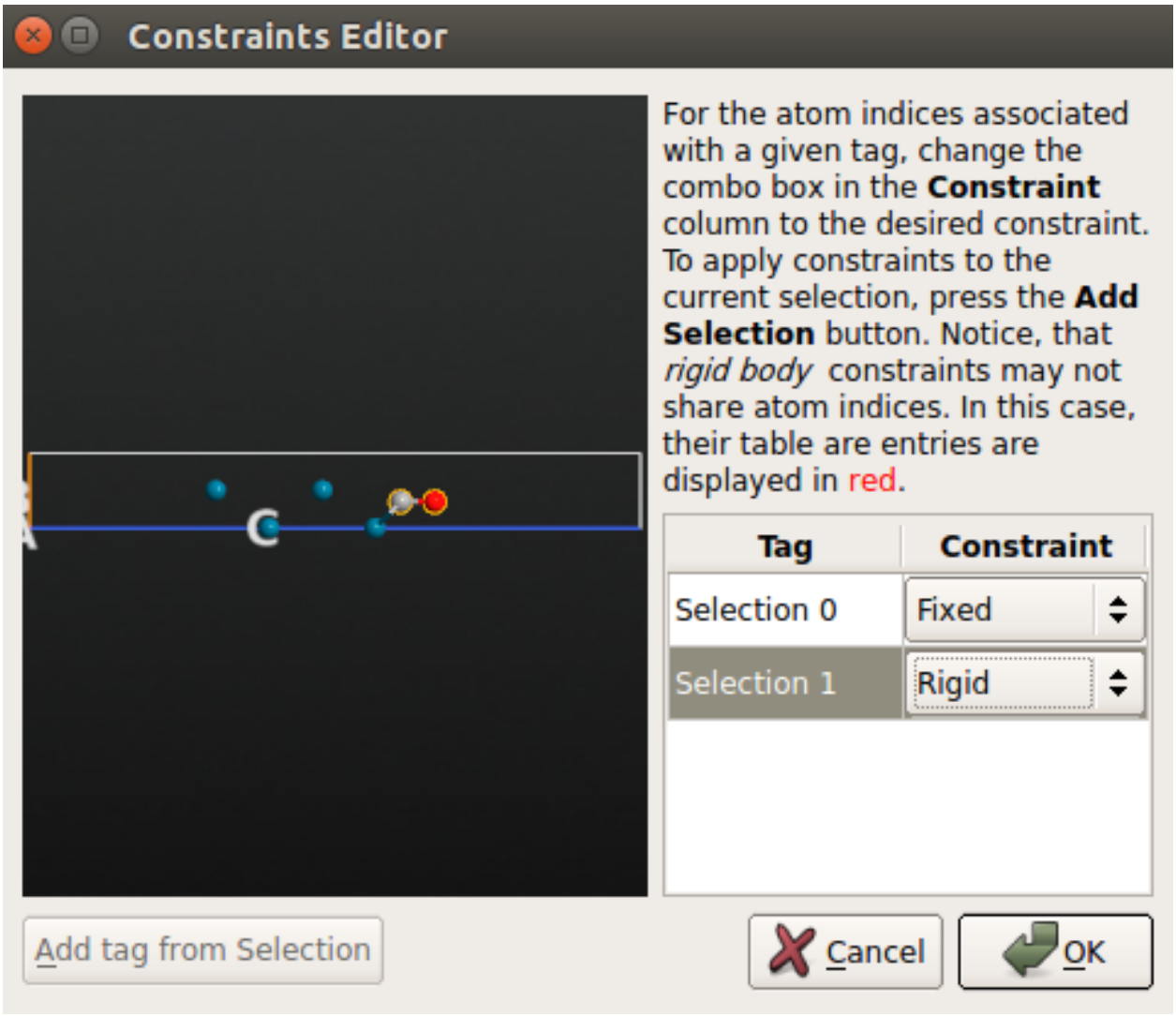


- Set up the **OptimizeGeometry** block similarly to the one used for Pd(100), but choose a slightly different set of

constraints:

–select the *Fixed* constraint for all atoms in the metal slab;

–select the *Rigid* constraint for the CO molecule (add the C and O atoms to the group Selection 1).



Note: The Rigid constraint on a group of atoms causes those atoms to move like a rigid body during force optimization.

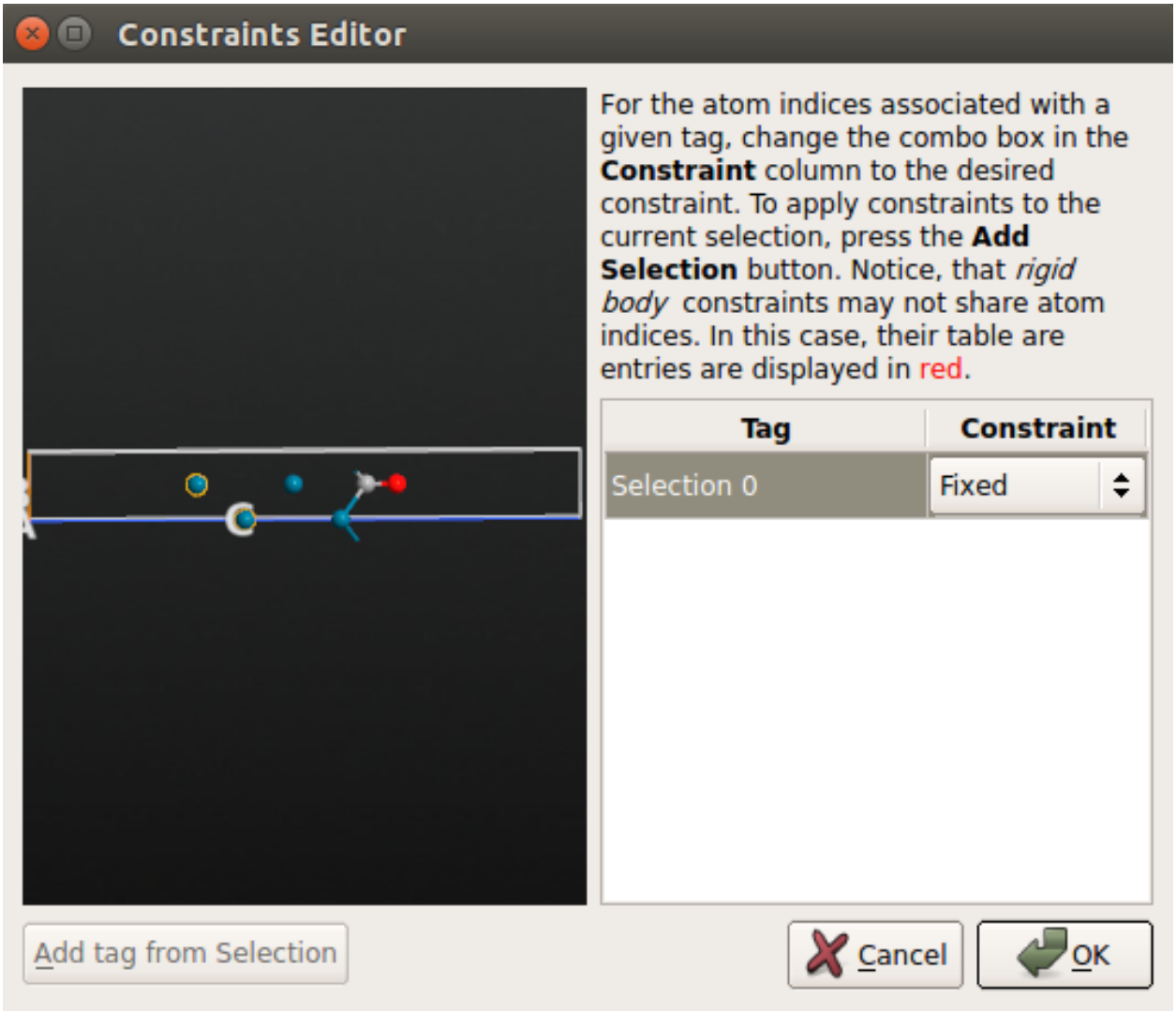
•Save the script as `Pd100_CO_rigid.py` and run it using the **Job Manager**. The simulation should take roughly 5 minutes if executed in serial. You can download the script here: `Pd100_CO_rigid.py`.

Important: If you take a look at the trajectory, you will see that the CO molecule has moved slightly away from its initial position, but the C–O bond length has not changed, and the molecule has not rotated. This is how the rigid constraint works.

Final relaxation

You will now perform the final geometry optimization of the CO/Pd(100) structure, starting from the result of the previous step.

- Send the configuration with ID *glD002* in `Pd_CO_rigid.hdf5` to the **Scripter**, and enter `Pd_CO.hdf5` for the default output file name.
- Once again, add the **New Calculation**, **OptimizeGeometry** and **TotalEnergy** blocks, and edit them similarly to what you did in the previous section. This time, however, use a 75 Hartree mesh-cutoff for the ATK-DFT calculator, and use only a *Fixed* constraint for the bottom two Pd atoms:



- Save the script as `Pd_CO.py` and execute it using the Job Manager or in a Terminal. The calculations should take about 20 min in total, but can of course be quicker if run in parallel on more CPUs. The script is also available here: `Pd100_CO.py`.

Tip: You can actually use the **Viewer** tool to monitor the progress of the geometry optimization while the

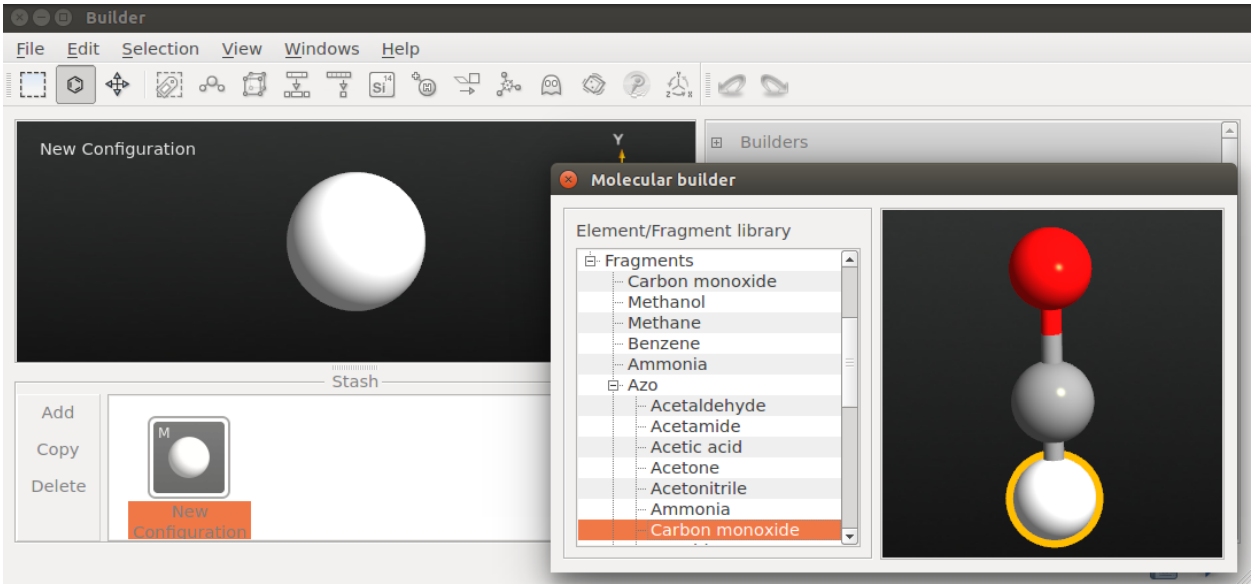
calculation is running. The trajectory object is continuously updated as new relaxations steps are taken.

When the calculation has finished you should have a look at the full relaxation trajectory, and see how the CO molecule has moved from a position around the hollow adsorption site on the Pd(100) surface to a bridge site between two Pd surface atoms:

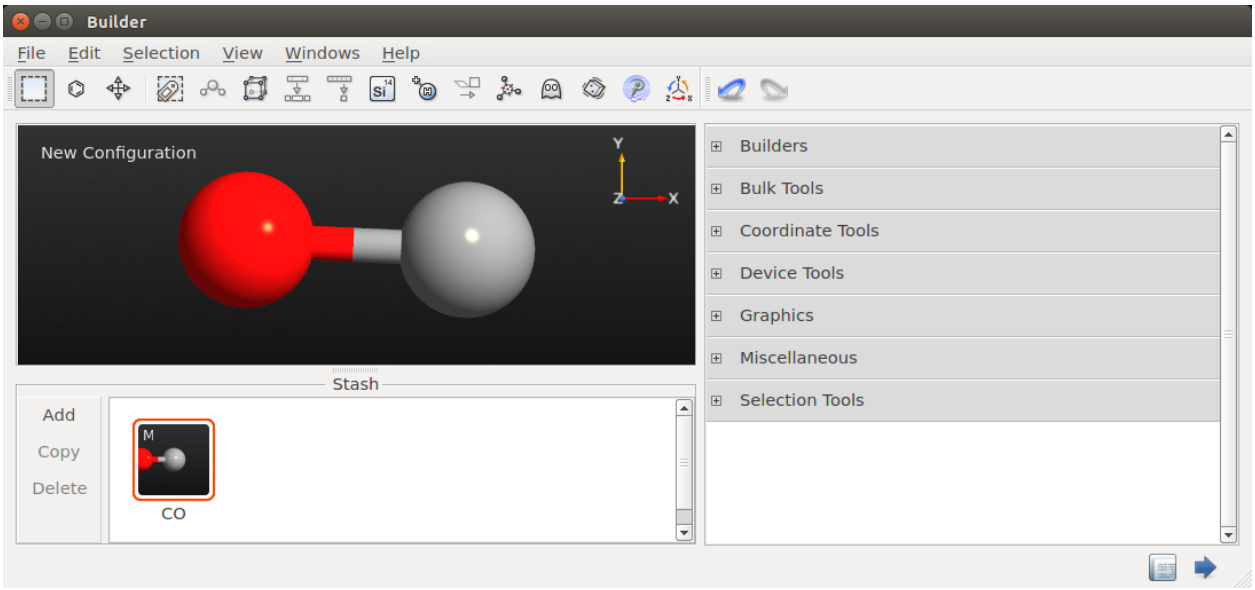
Relax the CO molecule

In order to calculate the CO adsorption energy on Pd(100) you also need to relax the isolated CO molecule. We can obtain it following the next steps:

- go to the **Builder** and add a new configuration using *Add* → *New Configuration*;
- click the **Molecular Builder** tool and click *Fragments* → *Carbon monoxide* in the panel that opens up;



- next, click the hydrogen atom in the `New Configuration` Stash item to substitute it for the CO fragment, close the Molecular Builder window, and rename the Stash item to `CO`.



Tip: For more details about how to use the **Molecular Builder**, you can check out this tutorial: *Molecular builder*.

Now you just need to relax the CO atomic coordinates and compute the total energy. Send the structure to the **Script Generator** and add the **New Calculator**, **OptimizeGeometry** and **TotalEnergy** blocks. Enter `CO.hdf5` for the default output file name, and then edit the blocks a bit:

- **New Calculator**: Choose the GGA-PBE functional;
- **OptimizeGeometry**: You may want to save the trajectory or decrease the force tolerance, but its not mandatory;

Save the script as `CO.py` and run the calculation – it should be very fast. The script is also available for download: `CO.py`.

You will find that the C–O bond length relaxes to 1.143 Å, which matches experiments very well.

Adsorption energy

You are now ready to calculate the CO/Pd(100) adsorption energy at a full monolayer coverage, ΔE, which is of course given by total energy differences:

$$\Delta E = E_{\text{products}} - E_{\text{reactants}} = E_{\text{Pd(100)/CO}} - E_{\text{Pd(100)}} - E_{\text{CO}}$$

You can use the **Text Representation** plugin to see all three total energies and then calculate ΔE, or you can use a script to do it: `adsorption_energy.py`.

The adsorption energy, calculated using PBE and the FHI pseudopotentials with a DZP basis set, is ΔE = –1.97 eV. However, this number is affected by the so-called basis set superposition error, so you should proceed to the next section to correct for this.

Counterpoise correction

The basis set superposition error (BSSE) is due to the incompleteness of the LCAO basis set, and can have a significant impact on energy differences between different sub-systems. As explained in a tutorial (*DFT-D and basis-set superposition error*), the superposition of Pd and CO basis sets in the CO/Pd(100) system can give rise to an artificial lowering of the total energy because the Pd(100) and CO sub-systems can borrow basis functions from each other. The result is a too large adsorption energy.

The standard approach to neutralize the BSSE is to apply a so-called counterpoise correction. In QuantumATK, this is done using the `counterpoiseCorrected`.

You should therefore calculate the counterpoise (CP) corrected total energy for the CO/Pd(100) system in order to compute the CP corrected adsorption energy:

$$\Delta E^{\mathrm{CP}} = E^{\mathrm{CP}}_{\mathrm{Pd(100)/CO}} - E_{\mathrm{Pd(100)}} - E_{\mathrm{CO}}$$

Use the script shown below (can be downloadad here: `Pd_CO_cp.py`). The script reads in the previously relaxed CO/Pd(100) configuration and the calculator that was used, and creates a new calculator that applies the CP correction. The configuration is then relaxed and the total energy is calculated.

Run the script using the Job Manager or in a Terminal. It should perform 4 very small relaxation steps, and finish within 10 minutes. The CP correction will increase the total energy of the CO/Pd(100) system, resulting in an adsorption energy of ΔE^{CP} = –1.68 eV.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

bulk_configuration = nlread('Pd100_CO.nc', BulkConfiguration)[-1]

# Add tags
bulk_configuration.addTags('CO',          [4, 5])
bulk_configuration.addTags('Pd',          [0, 1, 2, 3])
bulk_configuration.addTags('Selection 0', [0, 1])

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Get the calculator settings from non-BSSE calculation.
calculator = bulk_configuration.calculator()
basis_set = calculator.basisSet()
exchange_correlation = calculator.exchangeCorrelation()
numerical_accuracy_parameters = calculator.numericalAccuracyParameters()
poisson_solver = calculator.poissonSolver()

# Create BSSE calculator.
bsse_calculator = counterpoiseCorrected(LCAOCalculator, ["CO", "Pd"])
calculator = bsse_calculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=poisson_solver,
    )

```

(continues on next page)

(continued from previous page)

```
bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('Pd100_CO_cp.nc', bulk_configuration)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
constraints = [0, 1]

bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.05*eV/Ang,
        max_steps=200,
        max_step_length=0.2*Ang,
        constraints=constraints,
        trajectory_filename='Pd100_CO_cp.nc',
        disable_stress=True,
        optimizer_method=LBFGS(),
        )
nlsave('Pd100_CO_cp.nc', bulk_configuration)
nlprint(bulk_configuration)

# ------------------------------------------------------------
# Total Energy
# ------------------------------------------------------------
total_energy = TotalEnergy(bulk_configuration)
nlsave('Pd100_CO_cp.nc', total_energy)
nlprint(total_energy)
```

3.11.14 Modeling Vacancy Diffusion in $Si_{0.5}$ $Ge_{0.5}$ with AKMC

Version: 2017

Downloads & Links

initial-structure.py adding-akmc.py

In this tutorial, we will model vacancy diffusion in a $Si_{0.5}$ $Ge_{0.5}$ alloy using the Adaptive Kinetic Monte Carlo method (AKMC). AKMC is an algorithm for performing Kinetic Monte Carlo (KMC) simulations on-the-fly without having to have a predefined set of states and reaction mechanisms. The reaction mechanisms are determined by locating saddle points on the potential energy surface and then using Harmonic Transition State Theory (HTST) to calculate the reaction rate. You can learn more about the AKMC method in this tutorial: *Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)*.

Note: For typical MD simulations, it is impractical to model more than a couple of hundred nanoseconds. However, by using AKMC, we will be able to model the dynamics of the system on a longer time scale. This is because the KMC simulation describes the dynamics of the system as transitions between states that – once found – can be computed very quickly.

Obtaining an Initial Structure

In order to generate the initial $Si_{0.5}$ $Ge_{0.5}$ alloy structure, we will use the *Builders* → *Substitutional Alloy* in the **Builder**, implemented in the **ATK 2015** version. If you have an older version, you can use this script instead: `initial-structure.py`.

1. Open the **Builder**.

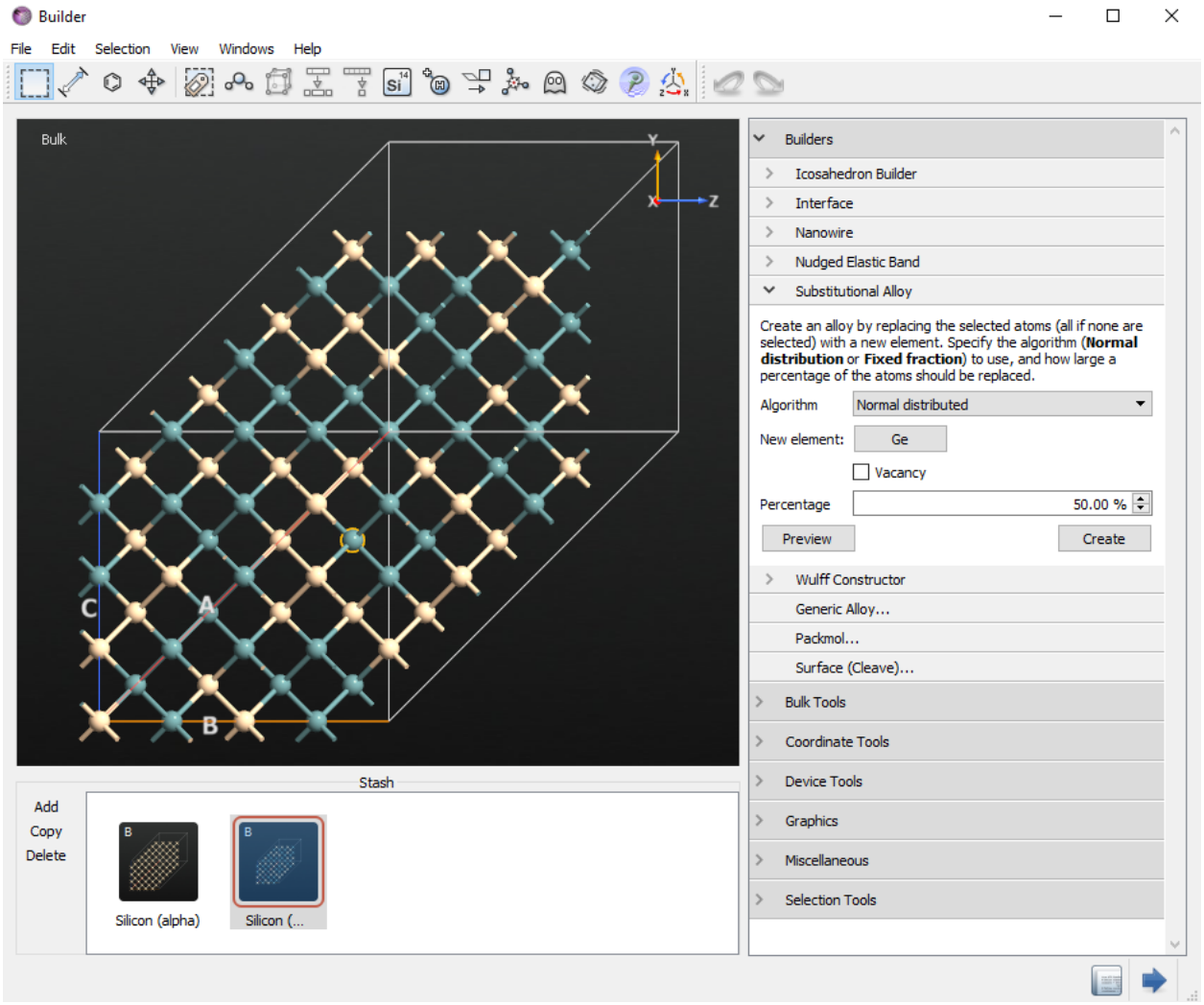
2. Using *Add* → *From Database*, add the Silicon (alpha).

3. Click the *Bulk Tools* → *Repeat*, set A = 4, B = 4, C = 4, and press the *Apply* button.

4. Select the *Builders* → *Substitutional Alloy*.

5. Use the default to substitute 50 percent of Si with Ge.

6. Click the *Create* button.



You now see the initial alloy in the main window of the **Builder** and in the **Stash**. It might be different from the one shown here, as the alloy is randomly created. We make one Si vacancy in the initial structure. In this case, we delete one yellowish Si behind blue-colored Ge at the front site. You can make one Si vacancy randomly.

Now we will optimize the initial structure.

1. Send the initial configuration to the **Script Generator** by clicking the button.

2. Add **New Calculator** and **OptimizeGeometry** blocks.

3. Change the output name to `alloy.hdf5`.

4. Double-click the **New Calculator** block and select the *ATK-Classical* calculator and choose the *StillingerWeber_SiGe_1995* Parameter set.

5. In the **OptimizeGeometry** block, set the Force tolerance to 0.01 eV/Å.

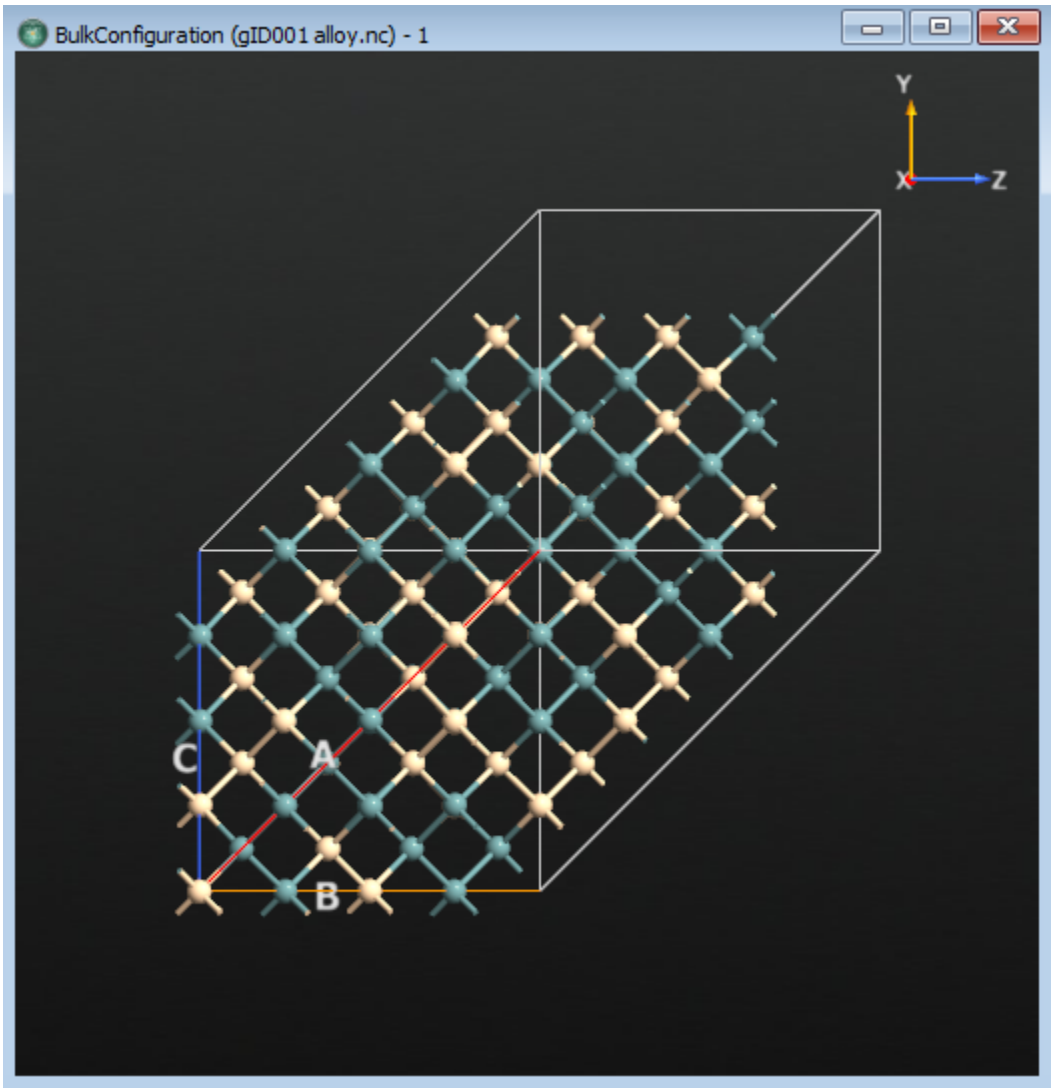
6. Send the script to the **Job manager** by clicking the button.

You can also download and use the script `initial-structure.py`. This script generates a silicon supercell and then

randomly replaces Si atoms with Ge atoms in order to achieve a 50:50 ratio and then creates a Si vacancy as shown below.

Run the script either by dragging it from the main **QuantumATK** window and dropping it on the **Job Manager**, or directly through the terminal.

Now you have generated a 4x4x4 supercell of SiGe with one vacancy, calculated the optimized geometry and saved the structure to `alloy.hdf5`. Note that this is a very small system, useful only for illustrative purposes. You will need a larger system for scientifically valid modelling. You can visualize the structure in **QuantumATK** by selecting the *gID0001* object from the alloy in the **LabFloor** and drag and drop it on the **Viewer**. The result is shown below.



There is a vacancy in the structure, but it is difficult to see. You will perform a *LocalStructure* calculation to see which atoms do not have the typical diamond-like packing.

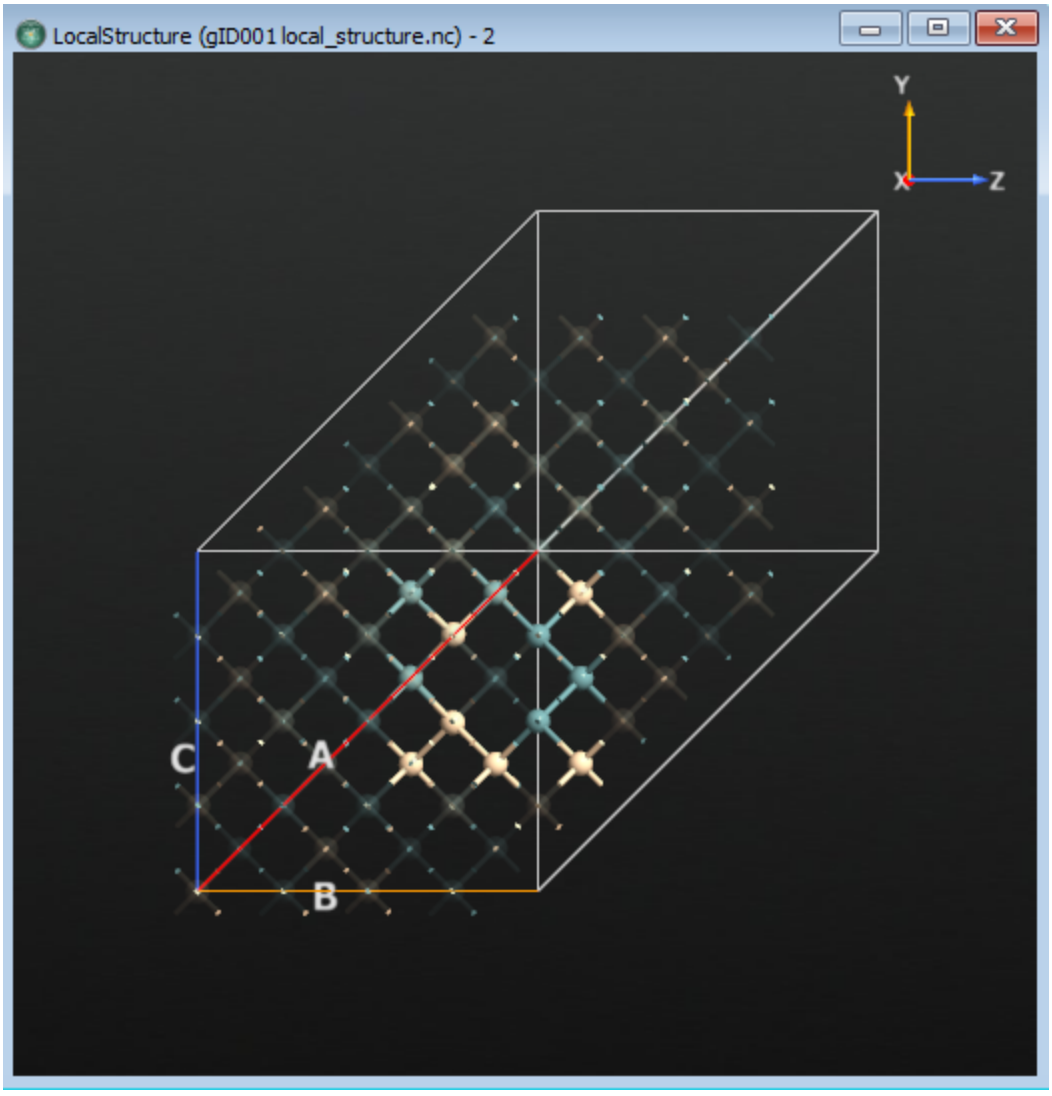
1. Drag the *gID0001* object from the **LabFloor** to the **Scripter**.

2. In the **Scripter**, double click on *Analysis* → *LocalStructure*.

3. Choose the default output file to be `local_structure.hdf5`.

4. Send the script to the **Job Manager** using the button and run the calculation. It should take no more than a minute.

Now you can get a better look at the structure around the vacancies. From the main window, open the **LocalStructure** item in the **Viewer** and expand the **Local Structure** panel in the sidebar. Select *Diamond-like*. Now all of the atoms that are not near the vacancies, and thus have the ideal crystal structure, have been selected. Open the **Properties** dialog and in the **Atoms** tab and change the opacity to 0.1 for the selected atoms. This will highlight the atoms near the vacancies, by making the others almost transparent. The result is shown in the figure below. Note the obvious vacancy in middle of the cluster.



Running the AKMC Simulation

Now that you have an optimized initial structure, you can run the AKMC simulation. Drag `alloy.hdf5` from the main **QuantumATK** window to the **Script Generator**. The **New calculator** will be automatically loaded in the Scripter. Starting from **ATK 2017**, AdaptiveKineticMonteCarlo is implemented in the **Scripter**. If you have an older version, you can download the script `adding-akmc.py` and append the contents in the **Editor** instead of following the steps below.

Add the *Optimization* → *AdaptiveKineticMonteCarlo* block.

1. Double click **AdaptiveKineticMonteCarlo**.
2. In the *Saddle Search* section, set the *Number of searches* to 100.
3. Go to the *Optimization* section, and press the *Atomic Constraint Editor* button.
   - Select one atom as far away as possible from the vacancy
   - Press the *Add tag from Selection* button
   - Change from None to *Fixed* for Constraint.
   - Press the *OK* button.
4. In the *Molecular Dynamics* part, set the *temperature* to 1200 K.

Note: If you look at the **Help** button on the AdaptiveKineticMonteCarlo widget, settings and definitions are introduced in detail. Higher number of searches and higher confidence give us more accuracy, but is also computationally more expensive.

---

5. Save the output file as `akmc.hdf5`.

Tip: When you see the AKMC script (either from the **Scripter** or in `adding-akmc.py`), it enables a restart of the simulation, by reusing the existing files if they exist. For a detailed description of the parameters for the AKMC object, please see the tutorial: *Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)*.

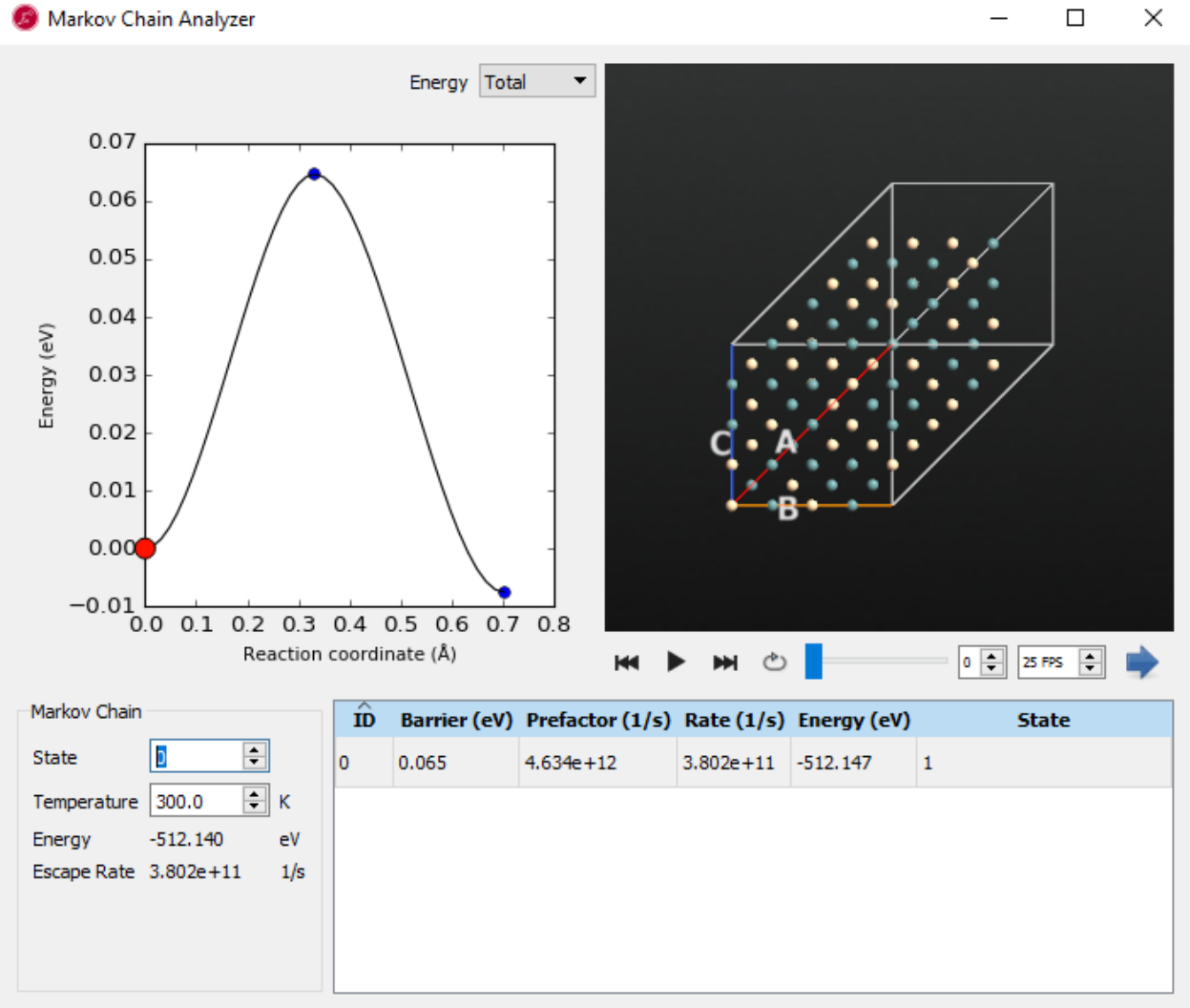
Now run the script. Running the script should take no more than half an hour on a desktop machine. At the end, you will have a *MarkovChain* object stored inside the `akmc_markov_chain.hdf5` file as well as a *KineticMonteCarlo* object stored inside the `akmc_kmc.hdf5` file. The *MarkovChain* object stores all of the states (configurations) that were found during the simulation as well as the connectivity between them. It can be used to calculate the full rate matrix for the whole system.

Note: Due to the random elements in these methods, you might not have exactly the same results as those analyzed here. If you want to follow the analysis exactly, you can download the files used in creating the tutorial here: `akmc_kmc.hdf5`, `akmc_markov_chain.hdf5` and `akmc_log.hdf5`

Select the *gID000* object in the `akmc_markov_chain.hdf5` file on the **Labfloor**, and click the **Markov Chain Analyzer** in the right panel.

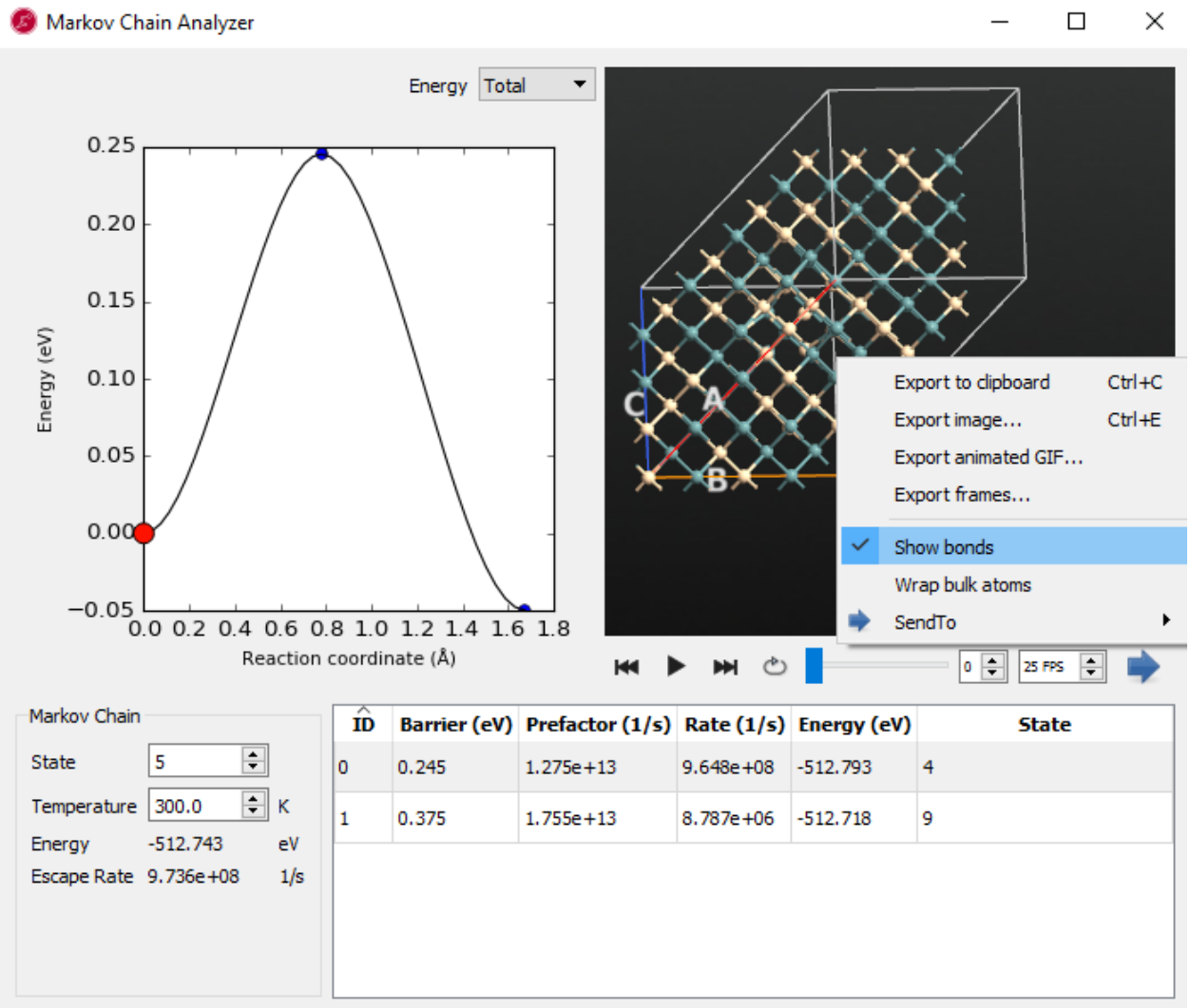
Examine the each state (here, state 0 and state 5 are shown in the below images) and use the play button ▶ in the movie tool to see the diffusion and movement of atoms. From state 0, the system has a very low energy barrier of 0.065 eV towards state 1. When you look at the movie, it seems to be a kind of relaxation and not real diffusion.

•**State 0**

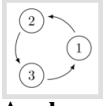


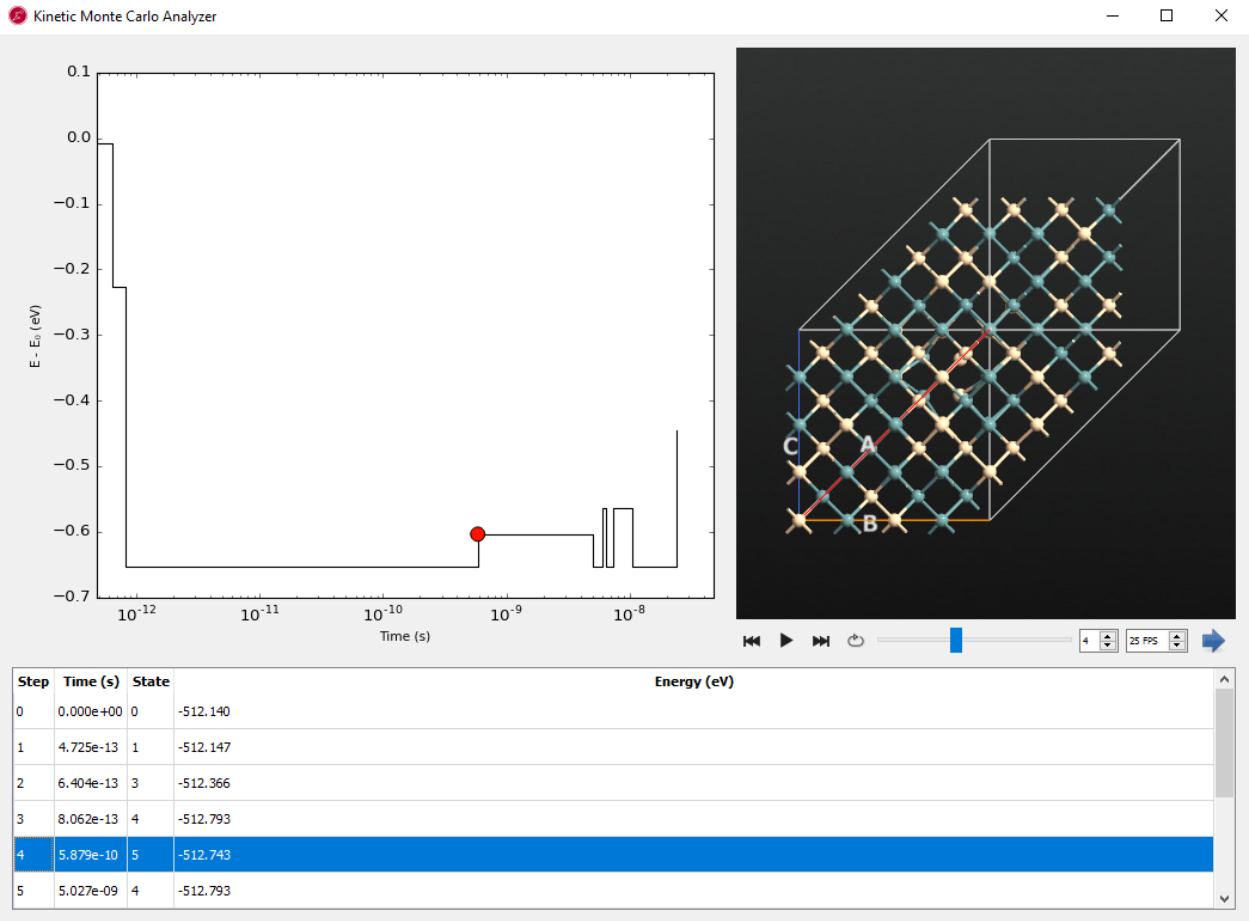
From state 5 we have found saddle points that are connected to state 4 and state 9. They have relatively higher barriers than the one leading from state 0. When you see the movie, silicon and germanium are diffusing and exchanged in the vacancy. If you click the right mouse button on the structure window, you can show the bonds. It can be helpful to show the movement of the atoms when going along the energy diagram.

•**State 5**



Note: Remember that you will probably have other results from a randomly generated alloy.

The *Kinetic Monte Carlo* object stores the time and state-to-state trajectory and is also able to take steps along the Markov chain (i.e. KMC steps). Select both objects, $\sum_i k_i$ in the `akmc_kmc.hdf5` and  in the `akmc_markov_chain.hdf5`, pressing the `ctrl` key, then see the activated **Kinetic Monte Carlo Analyzer** in the right panel. The output should look like, but probably not be identical to, the one seen below:



Step	Time (s)	State	Energy (eV)
0	0.000e+00	0	-512.140
1	4.725e-13	1	-512.147
2	6.404e-13	3	-512.366
3	8.062e-13	4	-512.793
4	5.879e-10	5	-512.743
5	5.027e-09	4	-512.793

This analyzer shows the state-to-state trajectory along time from the *Kinetic Monte Carlo* simulation. The KMC step, elapsed time, visited state id and energy are shown in the table. Click the ▶ movie button. You see the state to state dynamics with the energy vs. time plot. Since the plot is logarithmic on the time axis, the 0 and 1 steps near zero seconds cannot be seen clearly. You can find the different saddle states and staying time or escape time from each state. Starting from the initial state (0 eV), this plot shows the energy difference for other searched saddle points along the time.

In the **LabFloor**, click on the $\sum_i k_i$ object in the `akmc_kmc.hdf5` and select **Text Representation** from the panel on the right.

```
# Item:  0
# File:  C:\tutorial\akmc\akmc_kmc.nc
# Title: akmc_kmc.nc - gID000
# Type:  KineticMonteCarlo
    Step #   Time (seconds)  State
       0   0.00000000e+00      0
       1   4.72548280e-13      1
       2   6.40378862e-13      3
       3   8.06232998e-13      4
       4   5.87851546e-10      5
       5   5.02734297e-09      4
```

(continues on next page)

(continued from previous page)

```
                6   5.98297032e-09      7
                7   6.50322348e-09      4
                8   7.34741500e-09      7
                9   1.04551543e-08      4
               10   2.38590351e-08      8
               .
               .
               .
                    1501   9.97728735e-07      3
                    1502   9.99898062e-07      4
                    1503   1.00003149e-06      3
                    1504   1.00593170e-06      4
                    1505   1.00605679e-06      3
```

It includes the same information which we have seen in the **Kinetic Monte Carlo Analyzer** table. Here we show the results from a much longer simulation, with 500 saddle searches. The first column is the step number for the KMC algorithm, the second column represents the total elapsed time in the KMC simulation and the third column shows the current state number, which corresponds to a particular minimum energy configuration in the *MarkovChain* object. Note the break and how the time crosses 1 microsecond in the bottom text part (*step # 1501~1505*), which is actually from the middle of the file: `akmc_kmc_500.hdf5`

Note: Do not confuse the KMC step and saddle point number. The saddle points are discovered as they represent the most likely events to be chosen in the KMC simulation. To ensure the accuracy of the simulation, each state in the Markov chain has an associated confidence. This confidence is an estimate of the probability that for each KMC step taken from that state, the correct event was chosen. Saddle searches are executed in each state until a user defined confidence level is reached. Once the confidence level is reached, the KMC simulation will proceed until a state is reached where the confidence is less than the target level.

The output above includes more KMC steps than the 500 saddle searches that was specified in the script. This shows that the KMC algorithm, to some degree, has been jumping back and forth between the same states. However, as AKMC is a stochastic method it is also entirely possible to see the opposite, with the total number of KMC steps being much smaller than the specified number of saddle searches. This behavior is also very system-dependent.

Note: Note how the final time is greater than a microsecond. This is a much longer timescale than we could probe using MD alone.

On the **LabFloor** you will also find an *akmc_log* object. Open it by clicking on **Text Representation**. You

should see a message equivalent to what is shown below. This shows the results of the 100 saddle searches you specified in the script.

```
# Item:  0
# File:  C:\tutorial\akmc\akmc_log.nc
# Title: akmc_log.nc - gID000
# Type:  AKMCLog
state id search number confidence  message
       0             0   0.000000  Found new state
       0             1   0.864665  Found saddle connecting to state 1 again
```

(continues on next page)

(continued from previous page)

```
      0             2   0.950213  Found saddle connecting to state 1 again
      0             3   0.981684  Found saddle connecting to state 1 again
      0             4   0.993262  Found saddle connecting to state 1 again
      1             5   0.000000  Found new state
      1             6   0.000000  Found new state
      1             7   0.000000  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      1             8   0.375441  Found saddle connecting to state 3 again
      1             9   0.375441  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      1            10   0.375441  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      1            11   0.840168  Found saddle connecting to state 2 again
      1            12   0.840168  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      1            13   0.877314  Found saddle connecting to state 3 again
      1            14   0.923293  Found saddle connecting to state 2 again
      1            15   0.923293  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      1            16   0.940208  Found saddle connecting to state 2 again
      1            17   0.940208  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
```

The first column is the *id* of the initial state used for the saddle search, the second column is the *search number* (out of the 100 saddle searches), the third column is the calculated *confidence* that the processes starting in the current basin are adequately sampled and the last column is the result of the search. In this case, the algorithm finds 1 new state and meets the confidence of 99% after 5 saddle searches starting from state 0, then go to state 1 (*id* 1). It finds a total number of 2 new states (states 2 and 3), 17 searches find a duplicate state and 6 searches find no new states for various reasons. The initial state from *id* 0 to *id* 1 has changed, showing that the KMC algorithm has taken a step to a new state, and the saddle searches then continue from there.

Conclusion

In this tutorial we examined how to perform an AKMC simulation for the diffusion in an alloy with a vacancy at 300K. The AKMC method is useful to search for saddle points near the initial state which are relevant on a microseconds time scale. When you run a longer simulation or search more saddle points, you can model timescales which are comparable to experiments aiming to understand the state-to-state dynamics.

In a real study it is always important to verify the results of a classical potential and re-optimizing the NEBs describing the barriers using DFT, would be a good idea to determine pathways in the system more exactly.

3.11.15 Computing the piezoelectric tensor for AlN

Version: 2016.0

In this tutorial you will learn how to obtain the piezoelectric tensor and its coefficients with QuantumATK. Specifically, you will:

1. compute the piezoelectric tensor;
2. calculate the e_{33} coefficient by following these steps:

- calculate the changes in the relevant structural parameters due to strain;

•compute the piezoelectric tensor in the clamped-ion approximation and extract the coefficient $e_{33}(0)$;

•compute the Born effective charge.

Introduction

Piezoelectric materials exhibit an induced electric polarization upon the application of external macroscopic strain. The polarization can be reversed by applying an external electric field. These materials find application in a variety of Microelectromechanical Systems (MEMS).

In this tutorial we are going to study the piezoelectric constant of AlN. This synthetic ceramic material is used in a variety of MEMS such as Surface Acoustic Wave sensors (SAWs), and Film Bulk Acoustic Resonators (FBAR).

The piezoelectric tensor

In the absence of external fields, the total macroscopic polarization P of a solid is the sum of the spontaneous polarization P_{eq} (strain independent) of the equilibrium structure, and of the piezoelectric polarization induced by strain P_p (strain dependent).

$$P = P_{eq} + P_p$$

The piezoelectric tensor can be expressed as:

$$\gamma_{\delta\alpha} = \frac{\Delta P_\delta}{\Delta \epsilon_\alpha}$$

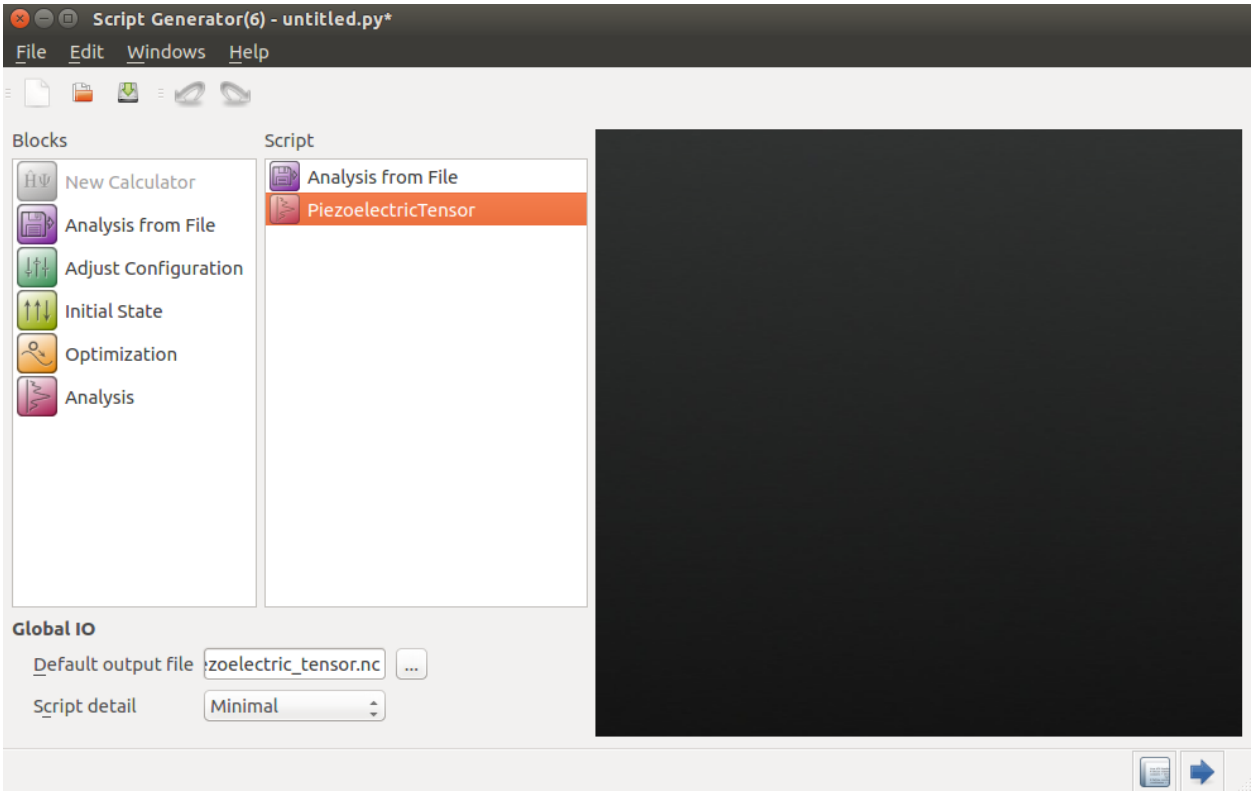
In QuantumATK $\gamma_{\delta\alpha}$ is calculated using a finite-difference approach and P is calculated using a Berry-phase approach. You can refer to the *Polarization* tutorial for more information on how P is evaluated using modern theory of polarization.

Computing the piezoelectric tensor

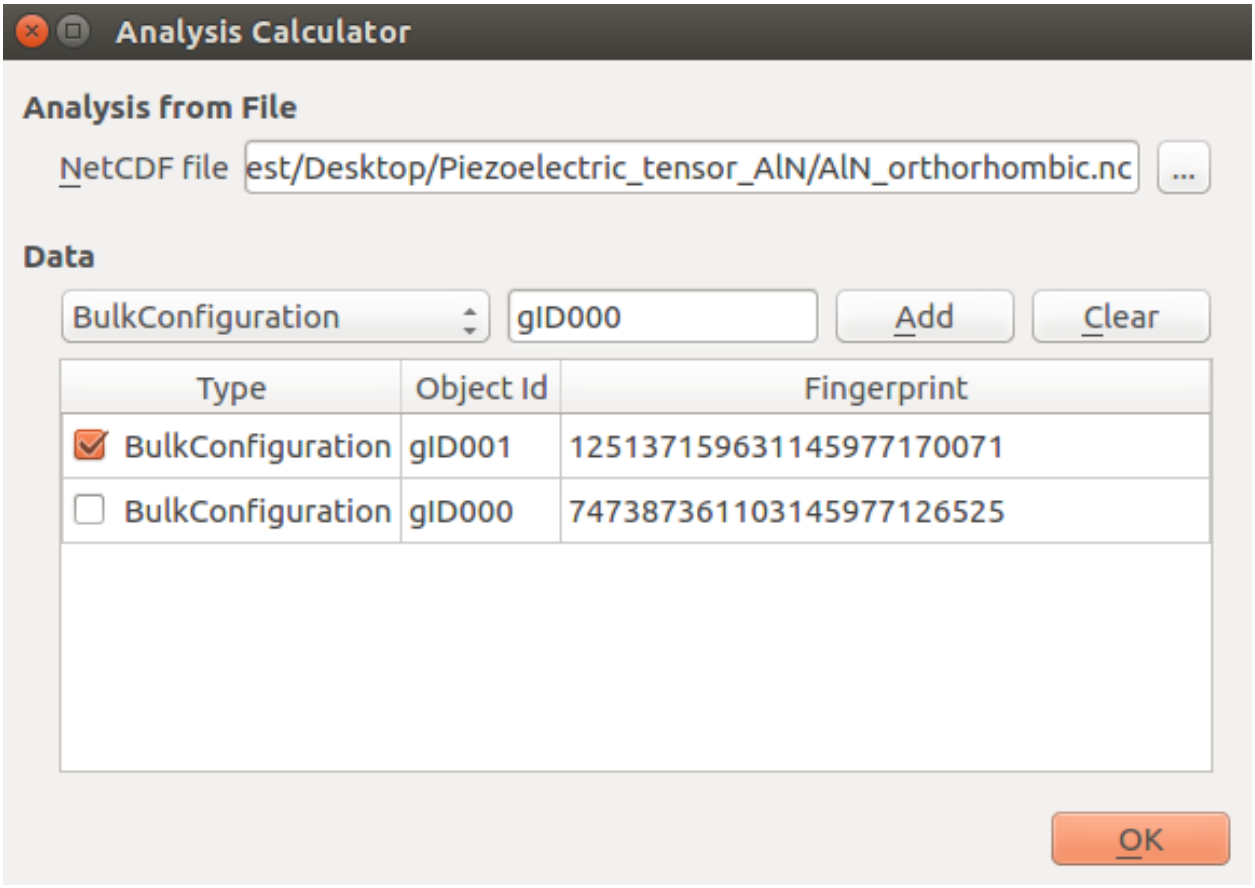
The first thing you should do is to import the AlN (Hexagonal) structure in the **Stash**, optimize its bulk structure and change the cell geometry to Orthorhombic. In this tutorial, you will use a AlN (Orthorhombic) bulk optimized at PBE using FHI pseudopotentials and DZP basis set.

Create a new **QuantumATK project**, and download the HDF5 file containing the optimized structure: (`AlN_orthorhombic.hdf5`). Save it in the **Project Folder** and its content will become visible in the **LabFloor**.

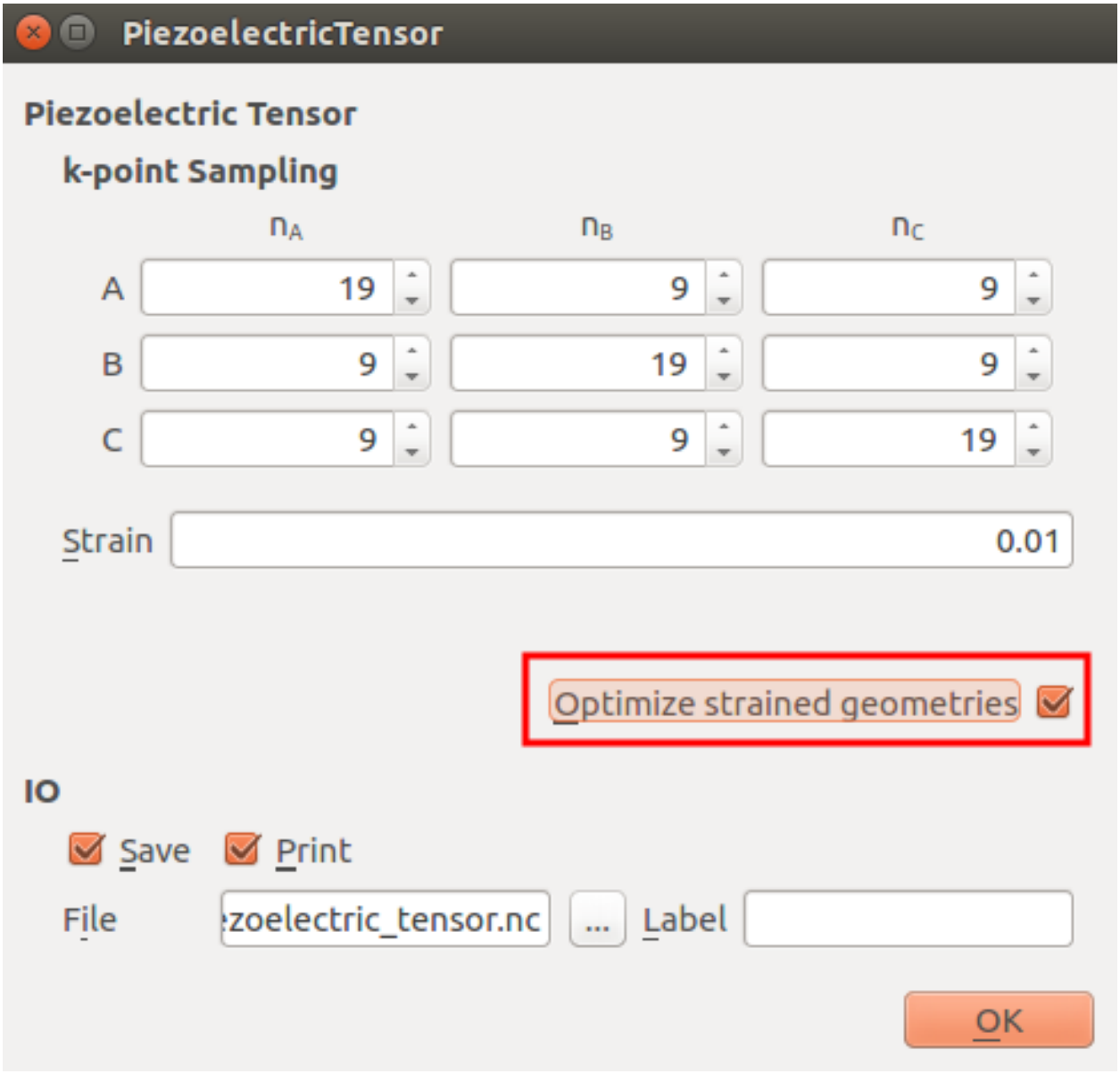
1. Go to **Scripter** and double click the *Analysis from File* and *PiezoelectricTensor* (click *Analysis* → *PiezoelectricTensor*) blocks to add them to the script.



2.Double-click the *Analysis from file* block and select the object containing the optimized bulk structure (object id: *gID001*), as shown below.



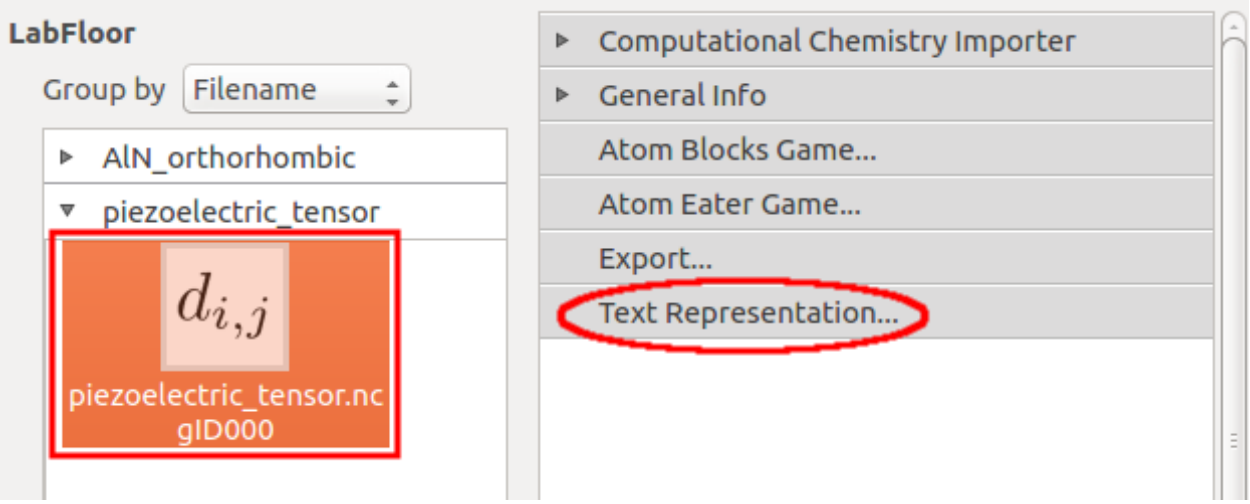
3.In the *PiezoelectricTensor* block, set the k-point sampling as in the next figure and tick Optimize strained geometries.



4.Save the HDF5 file and send the script to the **Job Manager** to run it.

Analysis of the e_{33} coefficient

In the **LabFloor**, select the **PiezoelectricTensor** object and click the Text Representation plugin in the left hand panel to inspect the piezoelectric tensor.



Text Representation - piezoelectric_tensor.nc gID000

```
# Item:  0
# File:  /home/guest/Desktop/Piezoelectric_tensor_AlN/piezoelectric_tensor.nc
# Title: piezoelectric_tensor.nc - gID000
# Type:  PiezoelectricTensor
+-------------------------------------------------------------------------------+
| Piezoelectric Tensor Report                                                   |
+-------------------------------------------------------------------------------+
|                                                                               |
| Tensor in units of [C/m**2]:                                                  |
|                                                                               |
|           x              y              z                                    |
| xx    1.29336e-11   -3.09261e-03   -8.80153e-01                               |
| yy   -5.80258e-12    1.12023e-03   -8.44547e-01                               |
| zz    3.01710e-11    8.57853e-04    1.48498e+00                               |
| yz   -9.05448e-14    5.50229e-01   -8.64941e-05                               |
| xz    5.00226e-01    5.79785e-13   -7.69274e-11                               |
| xy   -9.41930e-04    2.27296e-05    2.41124e-07                               |
|                                                                               |
+-------------------------------------------------------------------------------+
```

The predicted value for e_{33} is 1.4849 C/m^2. This value is in good agreement with the value found in the literature (1.46 C/m^2).[1]

In the next section we are going to see how you can calculate e_{33} in an alternative way.

[1] Fabio Bernardini, Vincenzo Fiorentini, and David Vanderbilt. Spontaneous polarization and piezoelectric constants of III-V nitrides. *Phys. Rev. B*, 56:R10024–R10027, Oct 1997. doi:10.1103/PhysRevB.56.R10024.

Alternative way of calculating the piezoelectric coefficient e_{33}

The piezoelectric coefficient e_{33} can be also calculated as the sum of two terms[Page 1569, 12]:

1. the clamped-ion term $e_{33}(0)$, that expresses the electronic response to strain.
2. a second term describing the effect of the internal strain on the piezoelectric polarization.

The complete expression for e_{33} thus reads:

$$e_{33} = e_{33}(0) + \frac{4eZ^*}{\sqrt{3}a^2}\frac{du}{d\epsilon_3}$$

where e is the electronic charge, ϵ_3 is the macroscopic applied strain and Z^* is the Born effective charge, which depends on the change in polarization upon the dispacement of an ion (or rather, a periodic sublattice of equivalent ions):

$$Z^*_{\nu,i,j} = \frac{\Omega}{|e|}\frac{\partial P^i_t}{\partial r^\nu_j}$$

where Ω is the unit cell volume, P^i_t is the total polarization along the Cartesian direction i, and r^ν_j is the coordinate of ion ν in direction j. The Born effective charge is also referred to as effective charge or dynamical charge.

Note: The Born effective charge is a tensor. În fact, when an ion is displaced in direction j, this will clearly affect the polarization in same direction j. However, it may also lead to a change in polarization in another direction i perpendicular to j.

In the calculations below the derivative will be approximated using finite differences:

$$\frac{\partial P^z_t}{\partial r^\nu_z} \approx \frac{P^z_t(+\delta\hat{z},\nu) - P^z_t(-\delta\hat{z},\nu)}{2\delta},$$

where $P^z_t(\pm\delta\hat{z},\nu)$ is the polarization along the z-direction when atom ν has been displaced by the amount δ in the positive/negative z-direction.

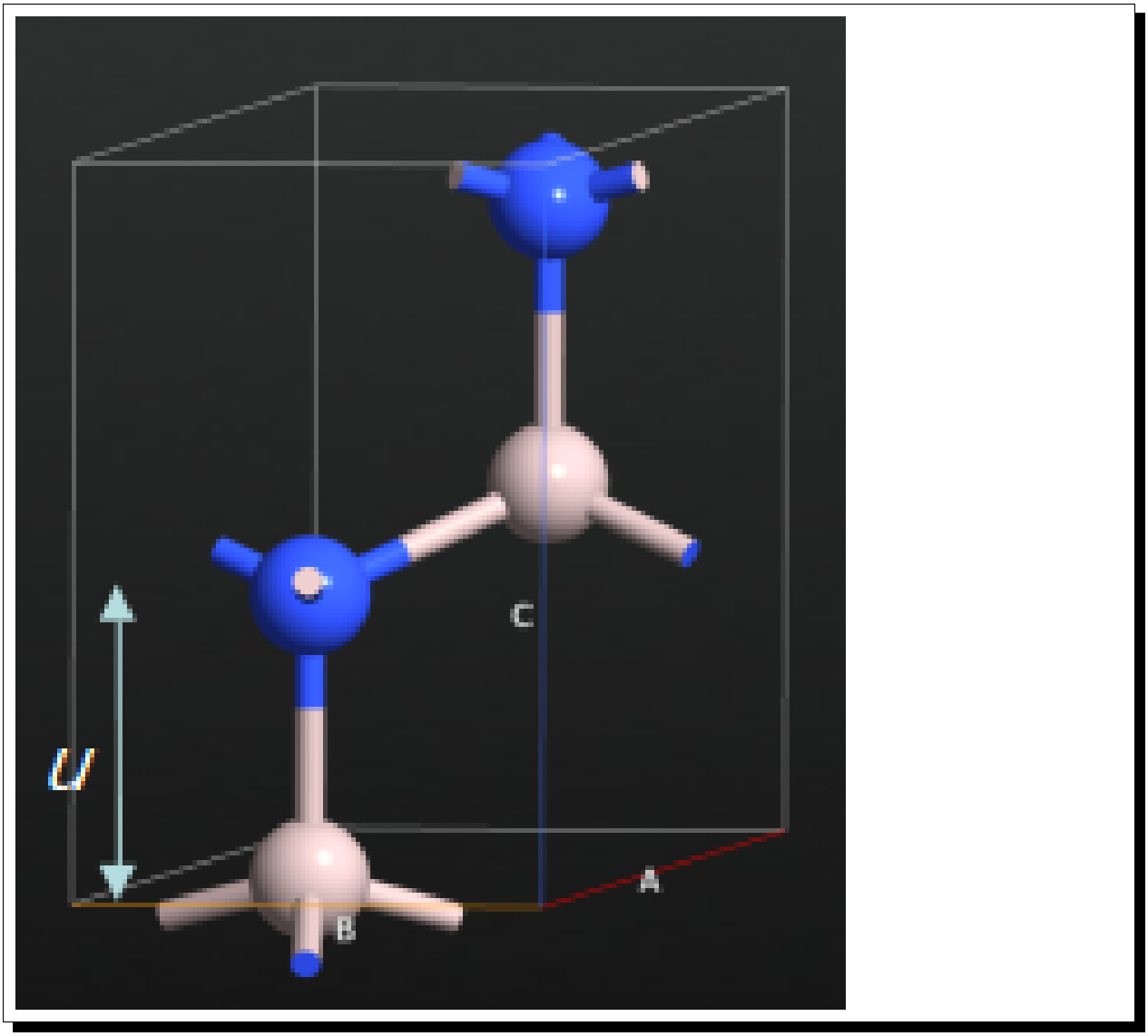
In the following, we will obtain e_{33} by calculating each term explicitly. Specifically, we will:

1. calculate the variation of the interatomic distance due to strain ($\frac{du}{d\epsilon_\alpha}$);
2. compute the piezoelectric constant in the clamped-ion approximation $e_{33}(0)$;
3. compute the Born effective charge Z^*.

The variation of the interatomic distance due to strain

The AlN hexagonal cell

[2] T.ăYokoyama, Y.ăIwazaki, T.ăNishihara, and M.ăUeda. Analysis on electromechanical coupling coefficients in AlN-based bulk acoustic wave resonators based on first-principle calculations. In *Ultrasonics Symposium (IUS), 2012 IEEE International*, 551–554. Oct 2012. doi:10.1109/ULTSYM.2012.0137.



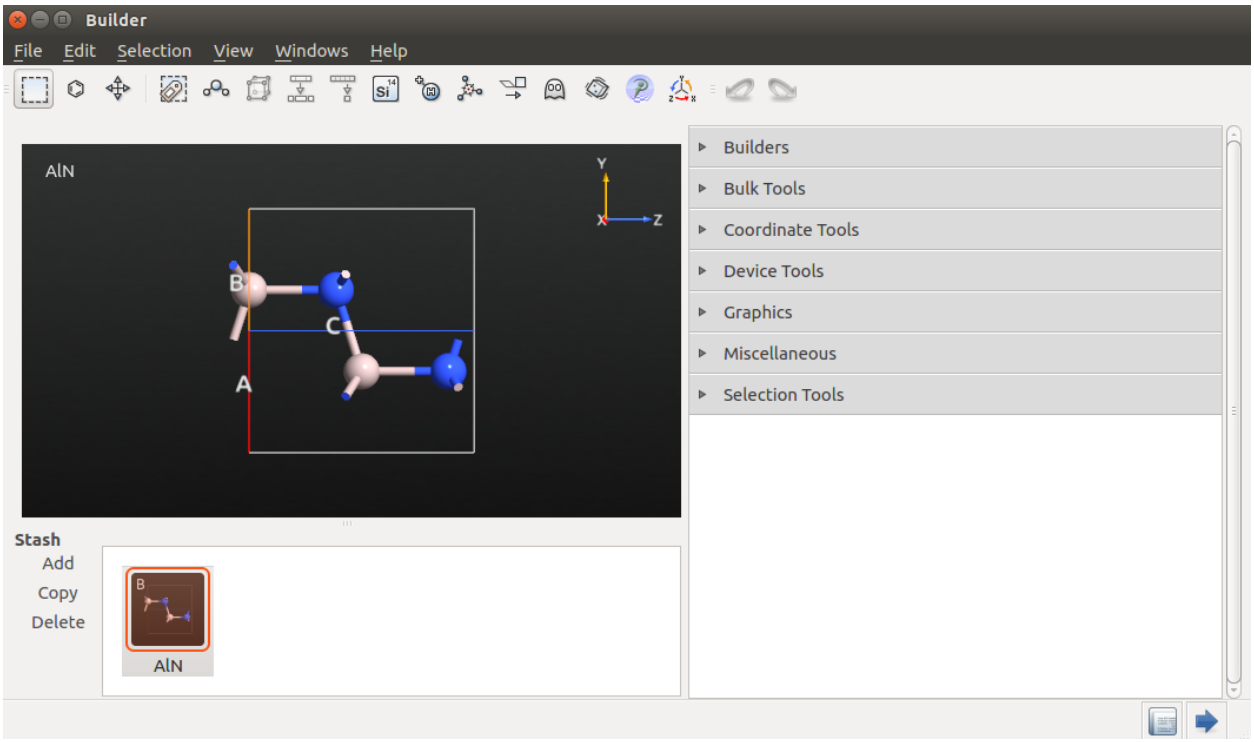
This variation is expressed as $\frac{du}{d\epsilon_\alpha}$ where du represents the difference in the interatomic distance u ($du = u' - u$) and $d\epsilon_3$ represents the applied strain ($d\epsilon_3 = c' - c$) in the direction parallel to the bond.

In order to calculate du we will:

1. relax the bulk structure and calculate u;
2. apply a 1% strain to the unit cell in c direction, relax the internal parameters and get the interatomic distance under strain, u'.

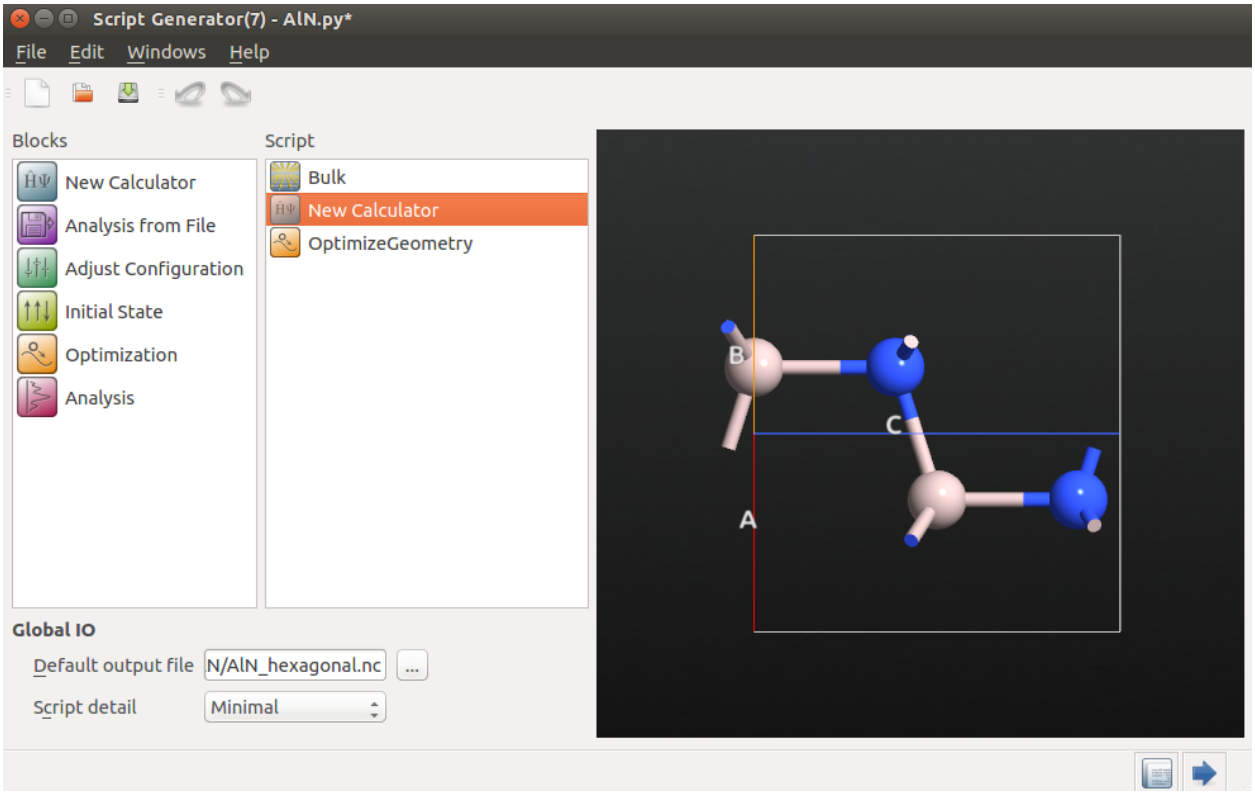
Calculating the interatomic distance u for the relaxed system

1.Go to **Builder** and click *Add* → *Form Database*, locate AlN (Hexagonal) in the database, and add it to the **Stash**.



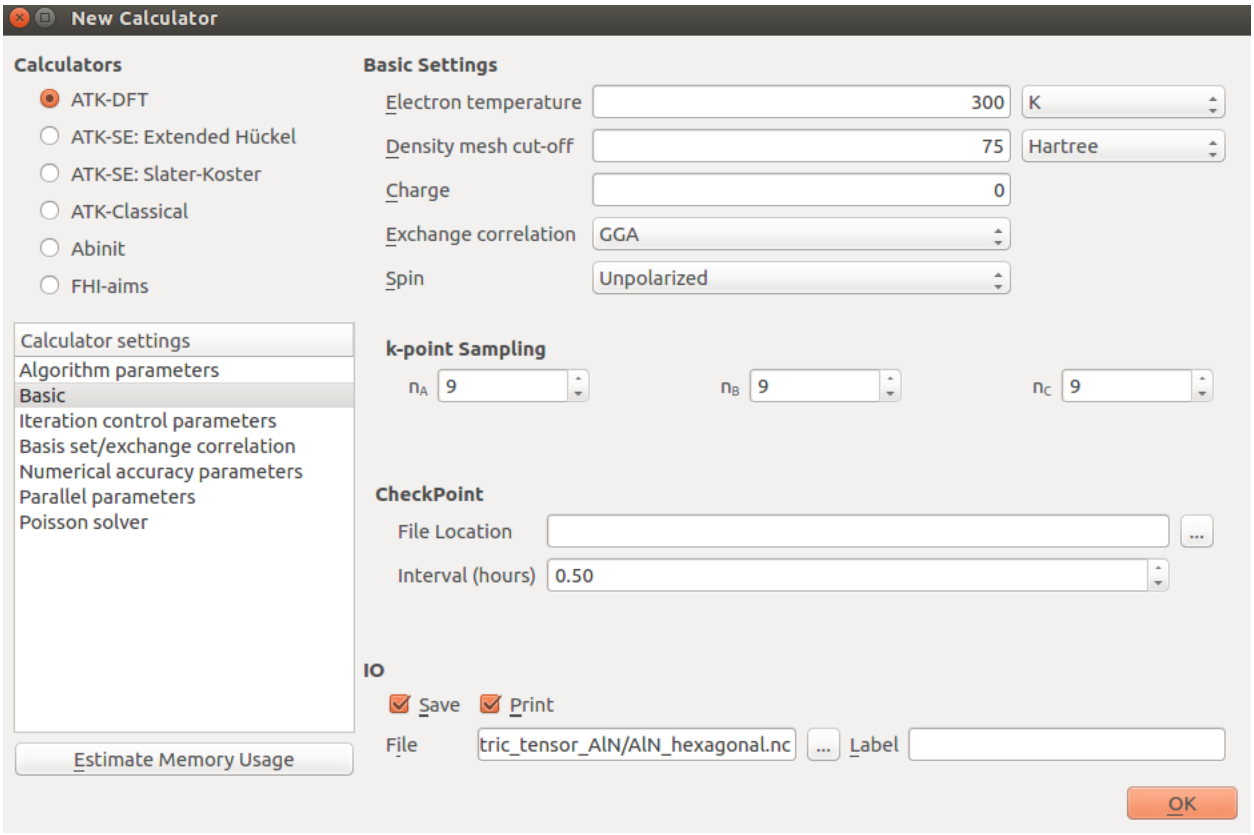
2.Send the bulk configuration to the **Script Generator**.

3.In the Script Generator, double-click the *New Calculator* and *OptimizeGeometry* (click *Optimization* → *Optimize Geometry*) blocks to add them to the script.

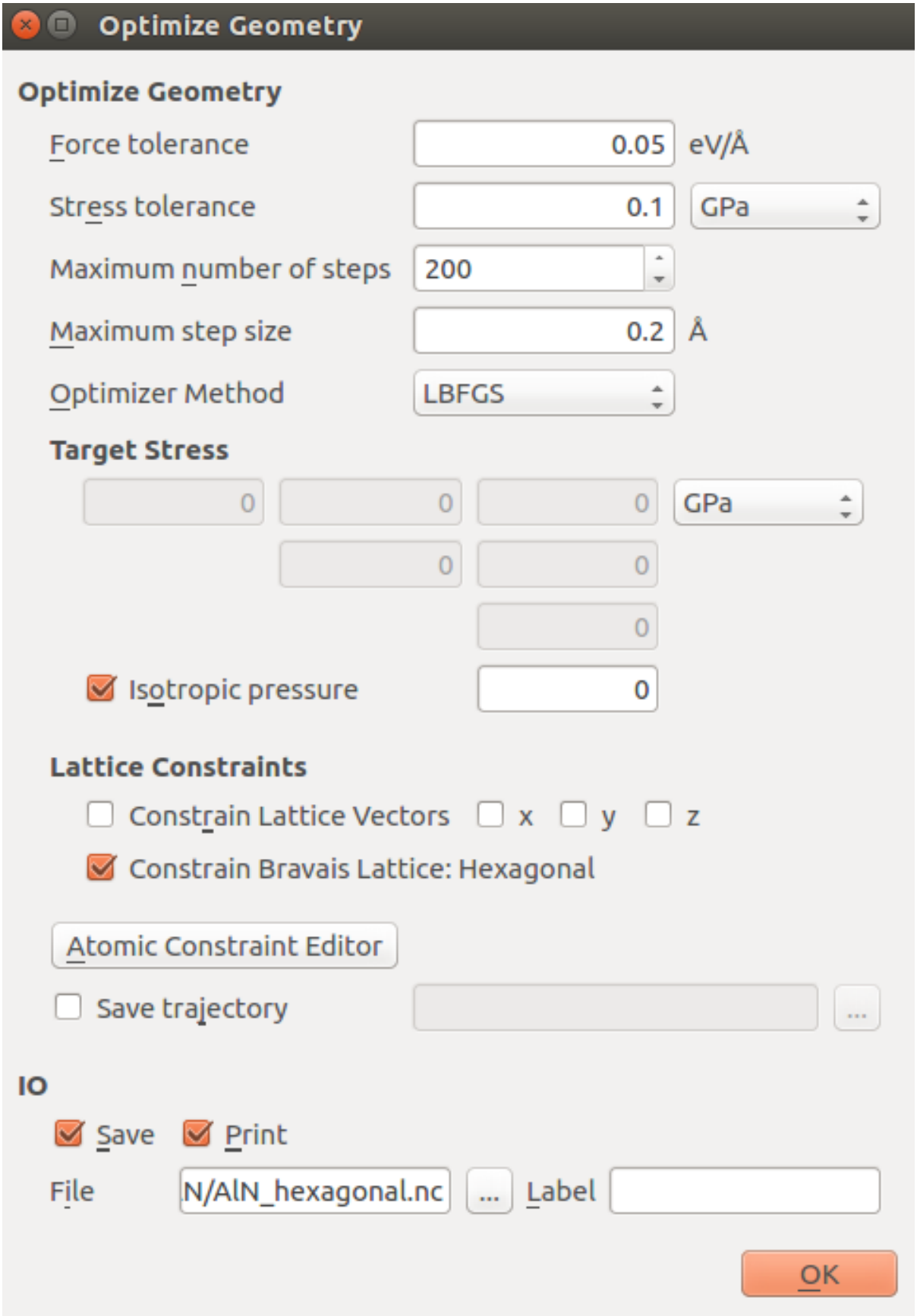


4.In the *New Calculator* block set the following parameters:

- *exchange and correlation*: GGA.PBE
- *Basis set*: DZP
- *k-point sampling*: (9,9,9)

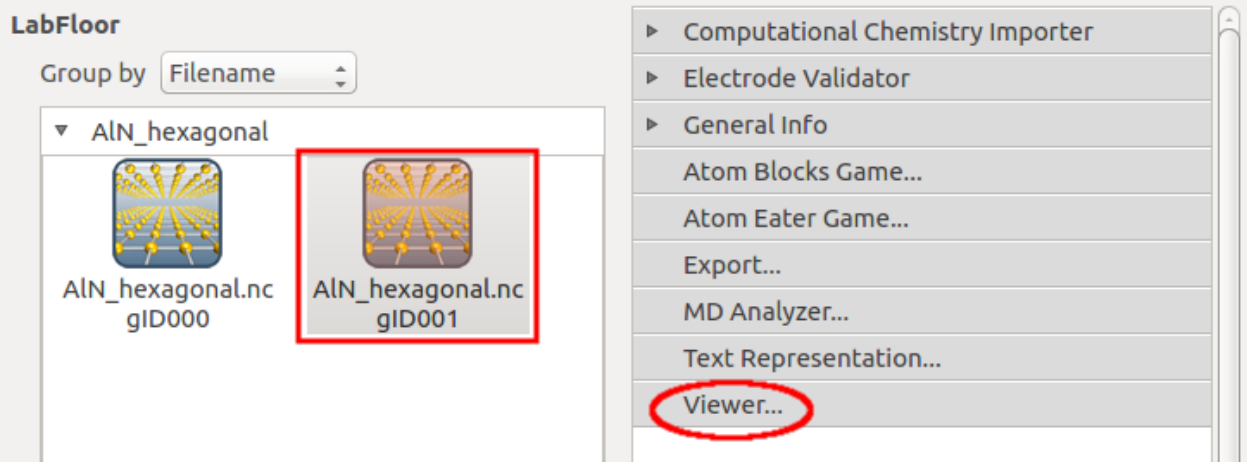


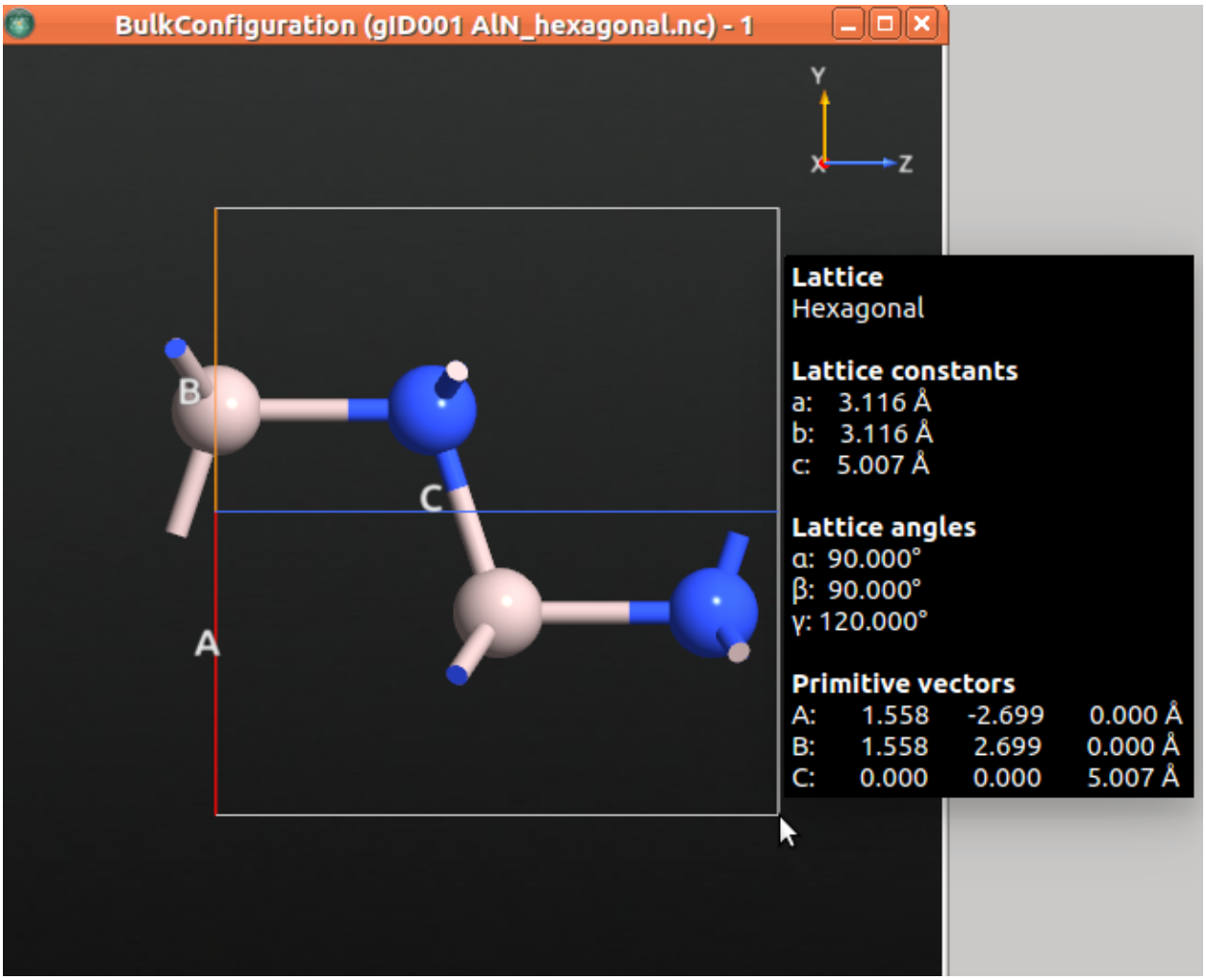
5.In the *OptimizeGeometry* block untick Constrain Lattice Vectors.

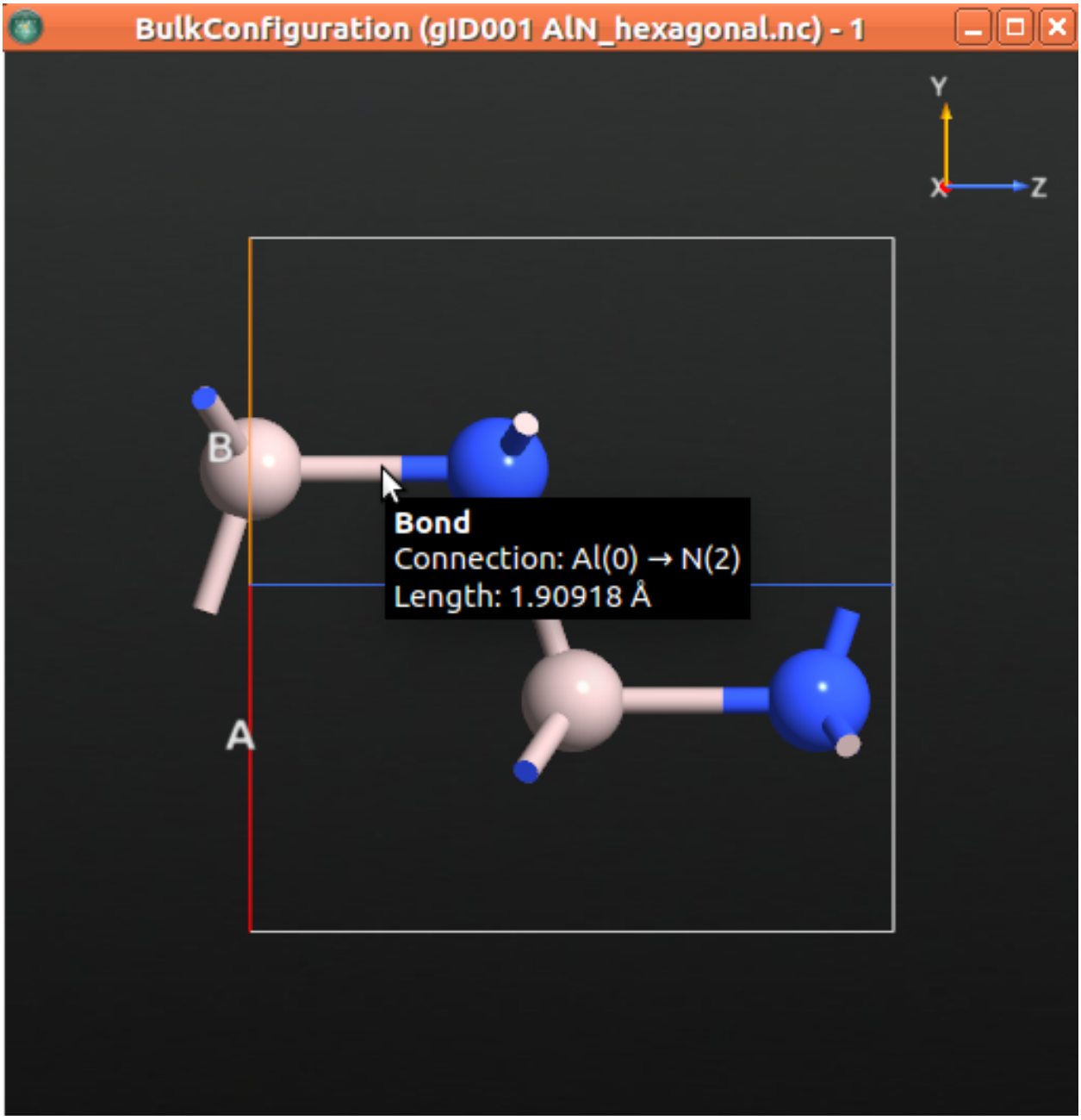

Optimize Geometry
Optimize Geometry
Force tolerance
0.05
eV/Å
Stress tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer Method
LBFGS
Target Stress
0
0
0
GPa
0
0
0
Isotropic pressure
0
Lattice Constraints
Constrain Lattice Vectors
x
y
z
Constrain Bravais Lattice: Hexagonal
Atomic Constraint Editor
Save trajectory
...
IO
Save
Print
File
N/AlN_hexagonal.nc
...
Label
OK

6. Save the HDF5 file as `AlN_hexagonal.hdf5` and send the script to the **Job Manager** to run it.

   Once the job is finshed, the optimized bulk structure should appear in the **LabFloor**. Select it and click the **Viewer** plugin to visualize it.







The table below shows the relaxed structural parameters.

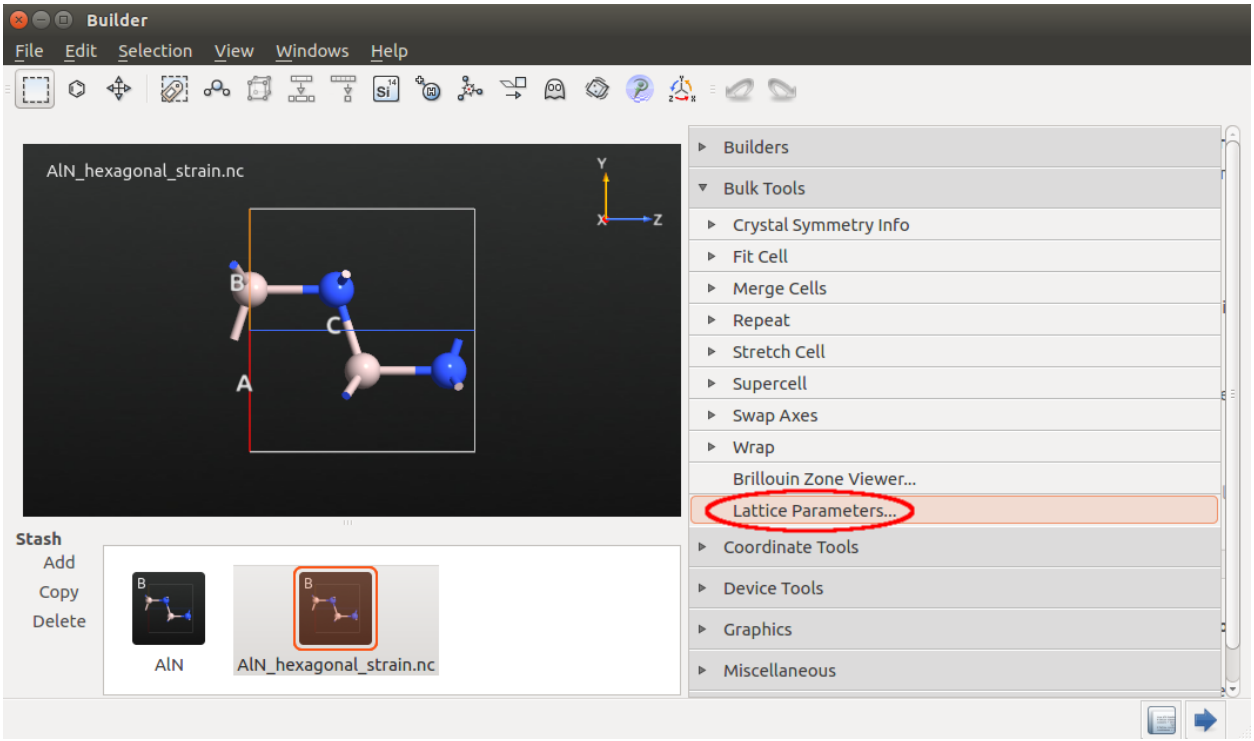
Table 3.6: Table: Relaxed cell parameters in

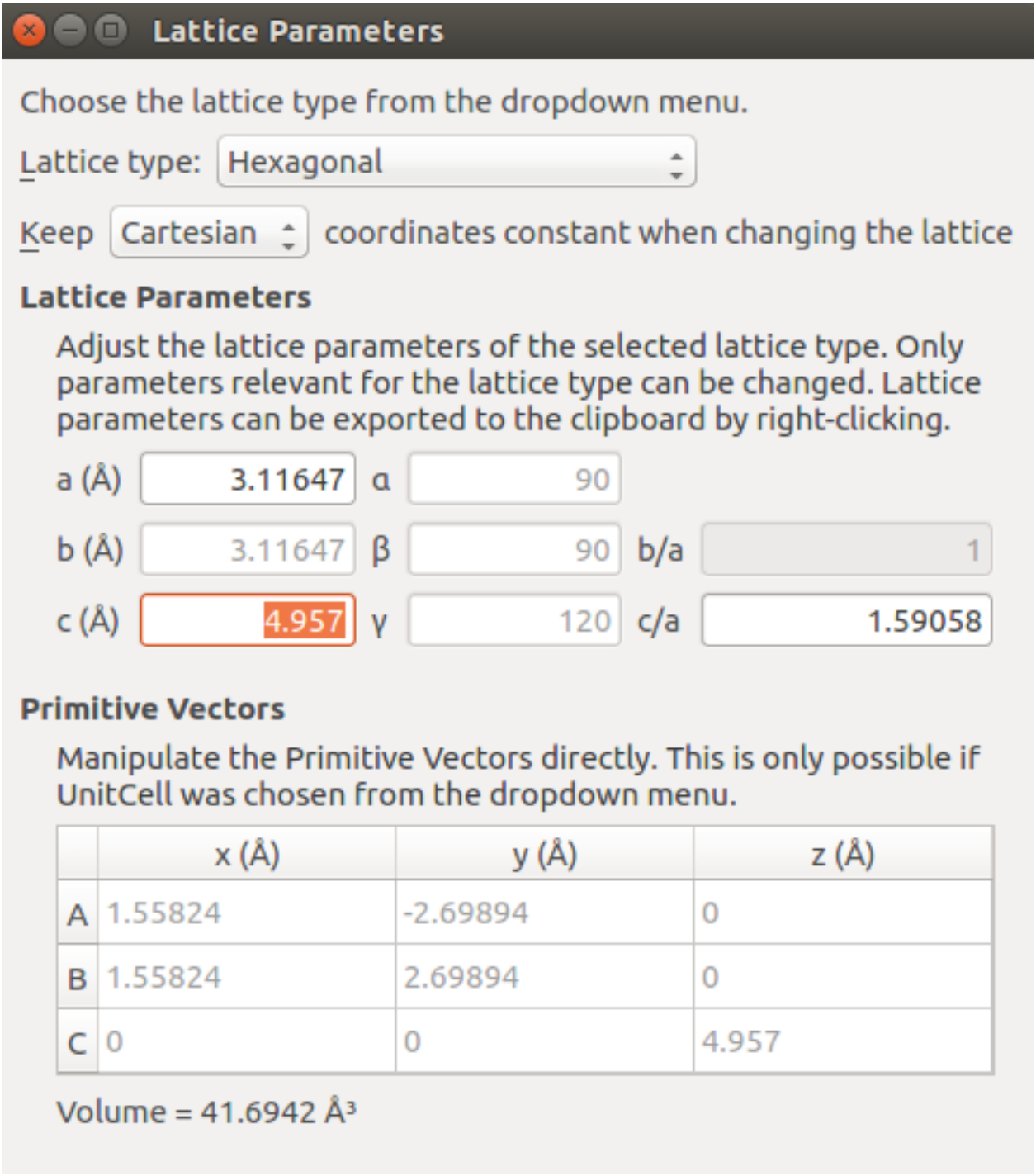
	QuantumATK	Bernardini *et al.*[Page 1569, 1]
a	3.116	3.0766
c	5.007	4.9810
c/a	1.607	1.6190
u/c	0.381	0.380

The value of the relaxed c lattice constant is 5.007 Å. In the next step, we will apply a 1% compressive strain in the c direction by diminishing the lattice constant in c direction in 0.05 Å.

Internal relaxation under strain in z direction

1. Send the relaxed structure to the **Builder** and rename it as `AlN_hexagonal_strain.hdf5`.
2. Go to *Bulk Tools* → *Lattice Parameters* to decrease the lattice parameter in 1% in c direction.





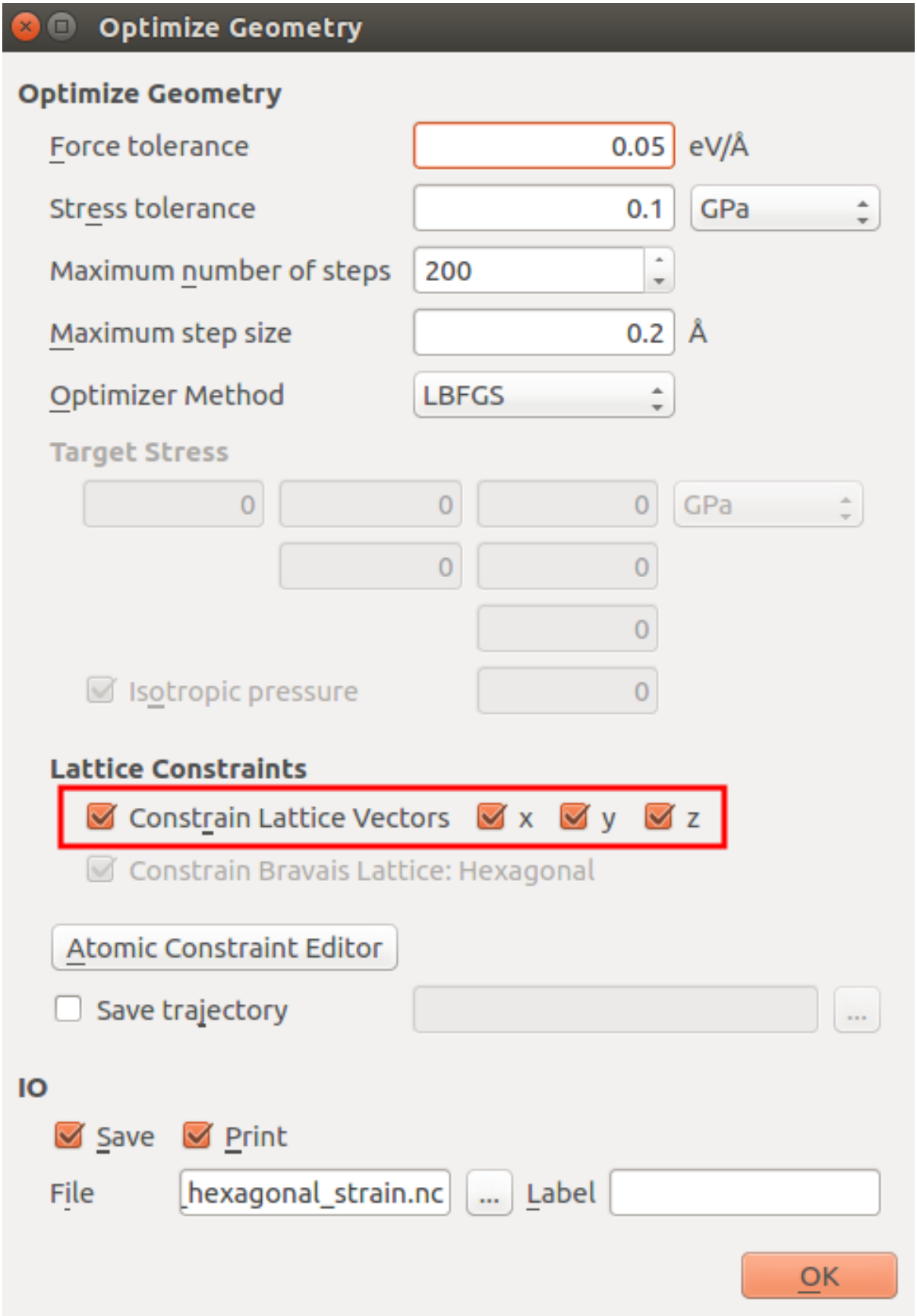
3.Send the structure to the **Script Generator** to create a script.

4.In the **Script Generator**, double-click the *New Calculator* and *OptimizeGeometry* blocks.

5.In the *New Calculator*, set the following parameters:

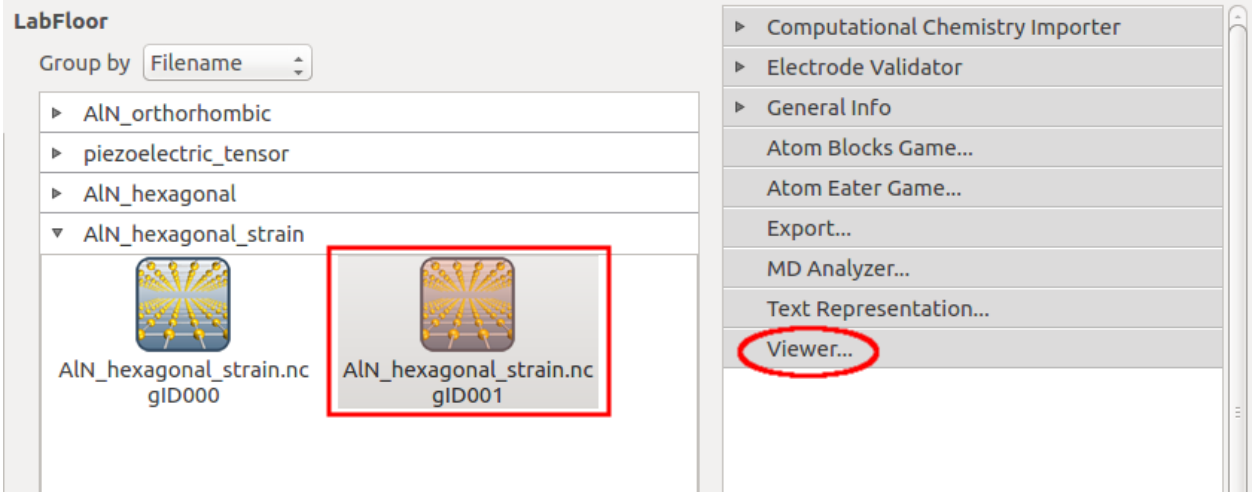
- *exchange and correlation*: GGA.PBE
- *k-point sampling*: (9,9,9)

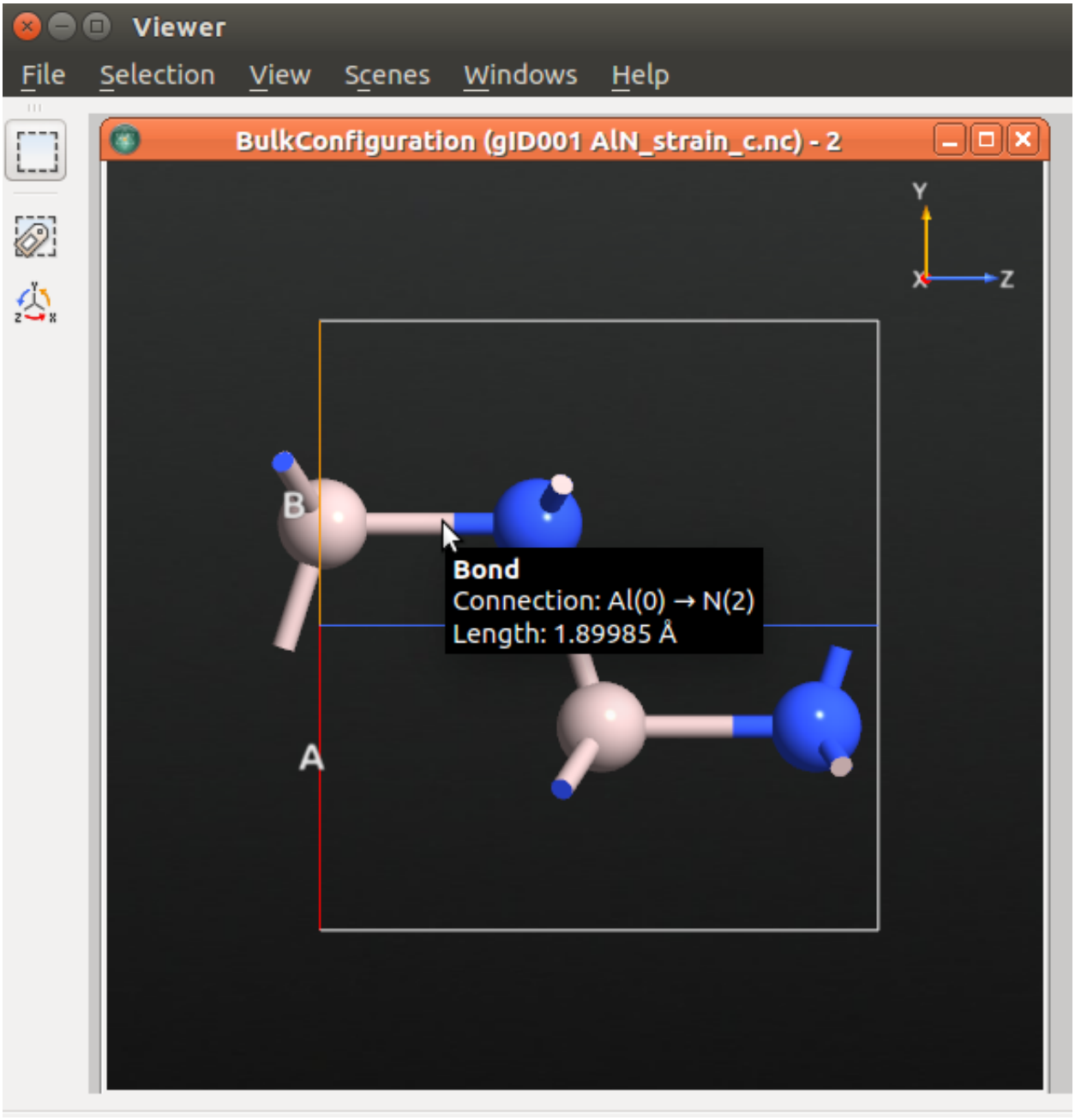
6.In the *OptimizeGeometry* block, constrain the lattice vectors as shown in the next figure.


Optimize Geometry
Optimize Geometry
Force tolerance
0.05
eV/Å
Stress tolerance
0.1
GPa
Maximum number of steps
200
Maximum step size
0.2
Å
Optimizer Method
LBFGS
Target Stress
0
0
0
GPa
0
0
0
Isotropic pressure
0
Lattice Constraints
Constrain Lattice Vectors
x
y
z
Constrain Bravais Lattice: Hexagonal
Atomic Constraint Editor
Save trajectory
...
IO
Save
Print
File
hexagonal_strain.nc
...
Label
OK

7. Send the script to the **Job Manager** and run the script.

   Once the calculation is finished, select the optimized bulk structure in the **LabFloor** and click the **Viewer** plugin to visualize the parameters.



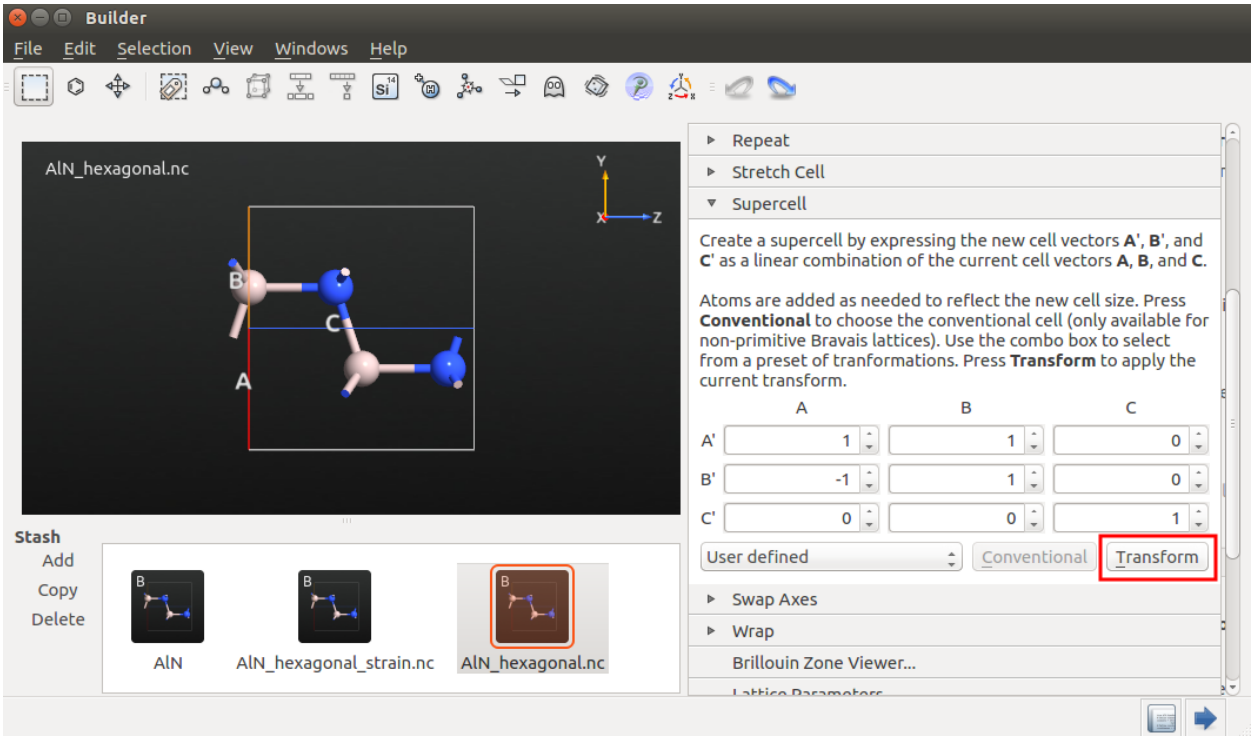


The predicted u' value is 1.89985 Å, and therefore the predicted $\frac{du}{d\epsilon_3}$ value is -0.1866. This value is in agreement with the value found in the literature (-0.18)[Page 1569, 1].

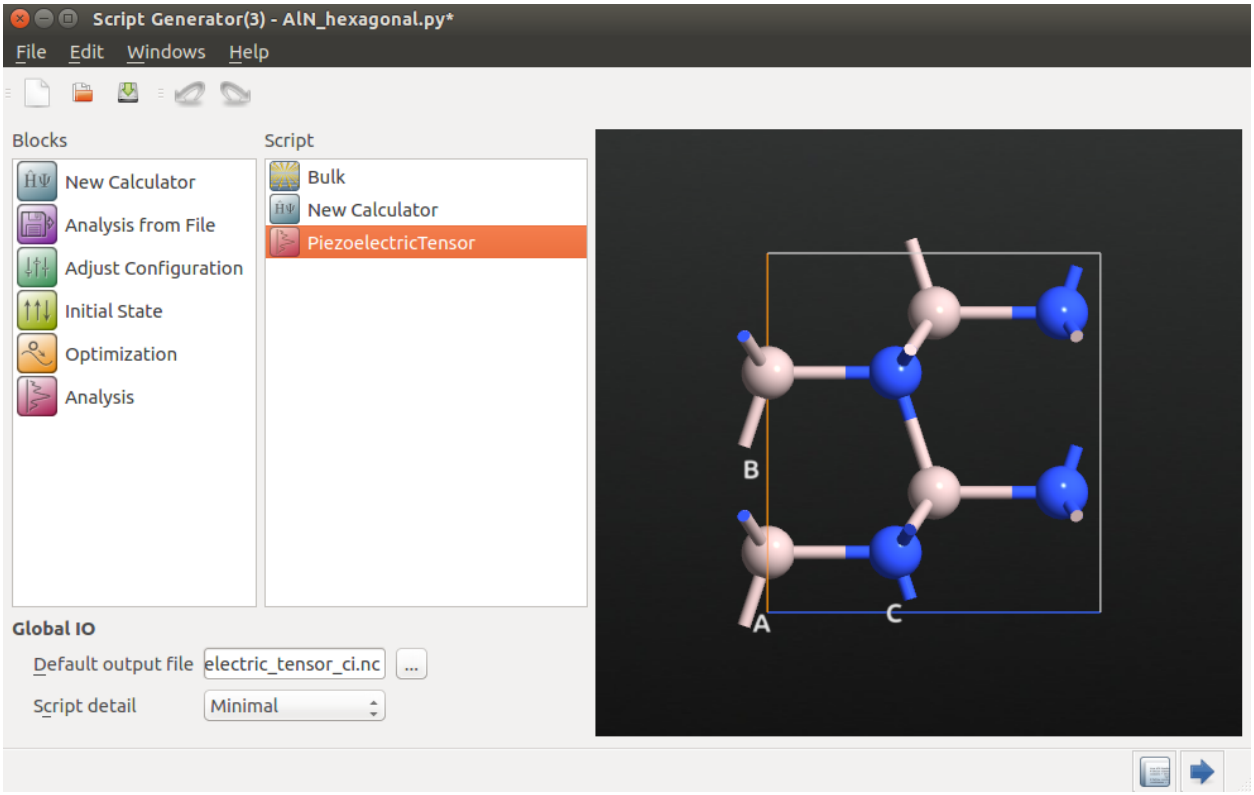
Computing the piezoelectric tensor in the clamped-ion approximation

1.Send the relaxed structure to the **Builder**.

2.In the **Builder**, go to *Bulk Tools* → *Supercell* to transform the cell according to the figure below.



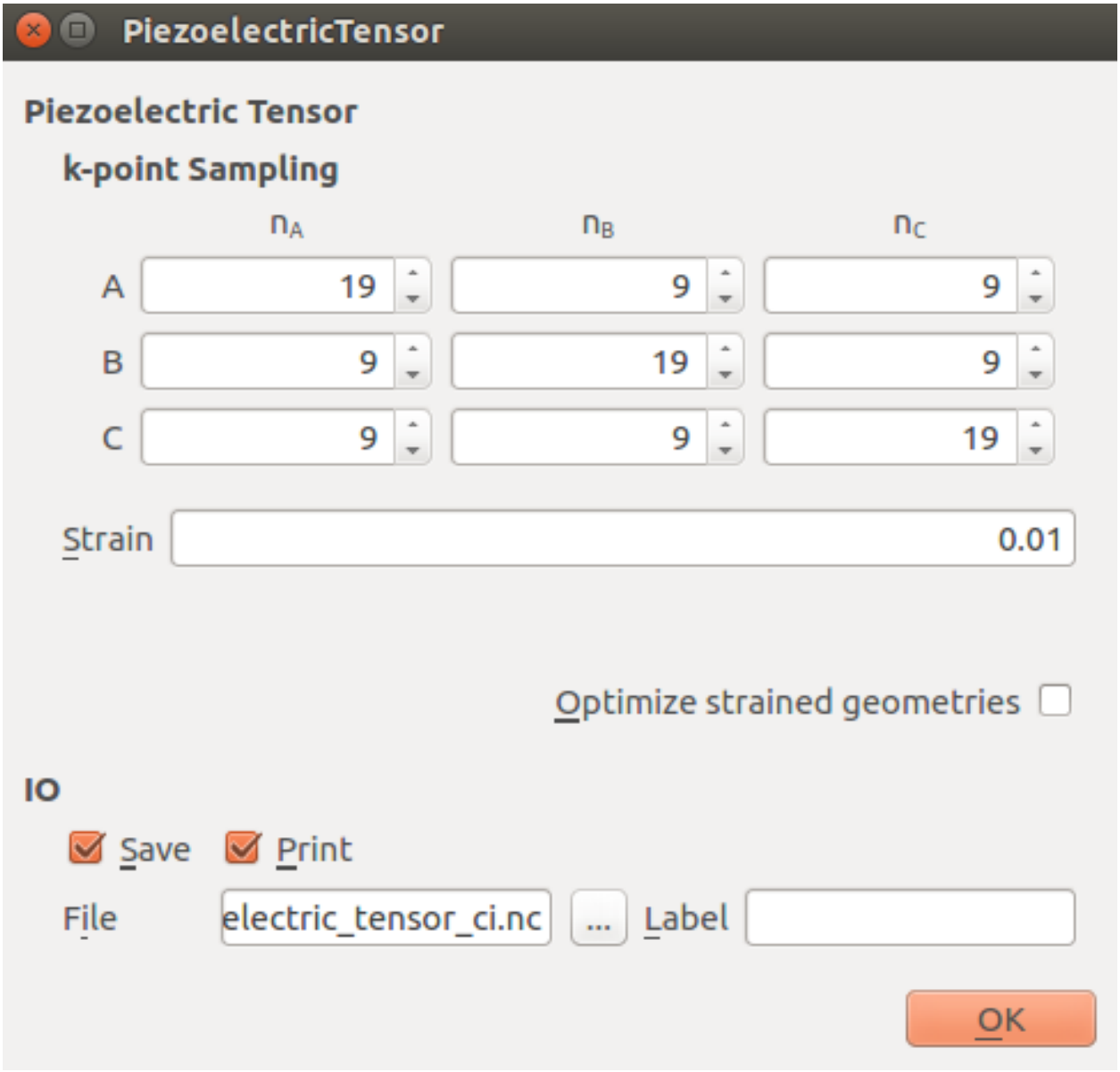
3.Send the bulk configuration to the **Script Generator** and add the *New Calculator* and the *Analysis* → *PiezoelectricTensor* blocks to the script.



4.In the *New Calculator* block set the following parameters:

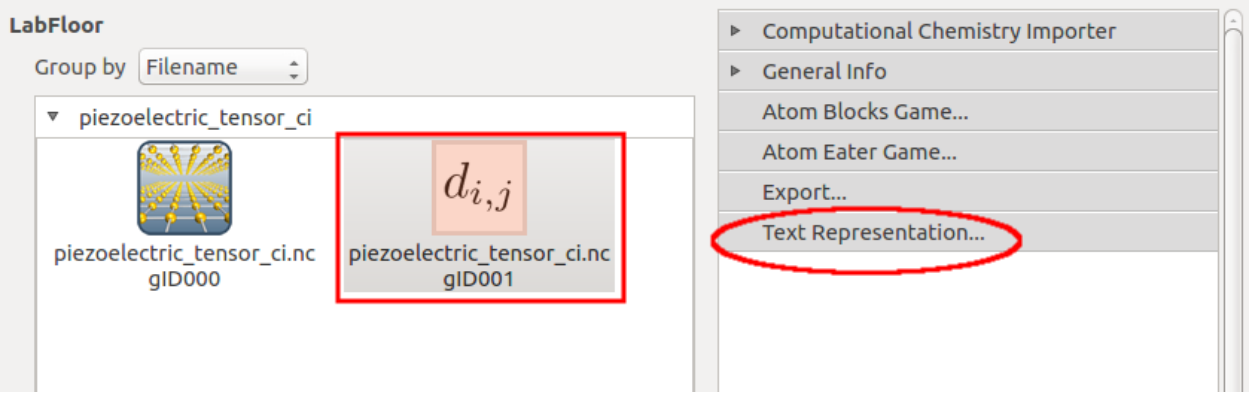
- *exchange and correlation*: GGA.PBE
- *k-point sampling*: (9,9,9)

5.In the *PiezoelectricTensor* block set a [(19,9,9), (9,19,9), (9,9,19)] k-point sampling as shown below.



6. Save the HDF5 file as `piezelectric_tensor_ci.hdf5` and send the script to the **Job Manager** to run it.

   Once the calculation is finished, the *PiezoelectricTensor* object will appear in the **LabFloor**. Use the mouse to select the object and click the Text Representation plugin to visualize the tensor.



Text Representation - piezoelectric_tensor_ci.nc gID001

```
# Item:  0
# File:  /home/guest/Desktop/Piezoelectric_tensor_AlN/piezoelectric_tensor_ci.nc
# Title: piezoelectric_tensor_ci.nc - gID001
# Type:  PiezoelectricTensor
+-------------------------------------------------------------------------------+
| Piezoelectric Tensor Report                                                   |
+-------------------------------------------------------------------------------+
|                                                                               |
| Tensor in units of [C/m**2]:                                                  |
|                                                                               |
|           x              y              z                                    |
| xx    1.78324e-13   -1.58097e-05   -1.20459e-01                               |
| yy    6.41823e-12   -6.95460e-05   -1.20540e-01                               |
| zz    6.87499e-12    7.95944e-05   -4.42412e-01                               |
| yz   -2.57723e-13    5.49772e-01   -7.32322e-05                               |
| xz    4.99898e-01   -4.20594e-13   -6.18672e-12                               |
| xy    1.12906e-04   -6.35505e-12    9.15934e-13                               |
|                                                                               |
+-------------------------------------------------------------------------------+
```

The predicted $e_{33}(0)$ value is -0.4424 C/m^2 , which is in good agreement with the value found in the literature[Page 1569, 1].

Computing the Born effective charge

In order to compute the Born effective charge in QuantumATK, you need to use the script provided here (`born_charges.py`).

```
#read saved configuration
configuration0 = nlread('piezoelectric_tensor_ci.hdf5', object_id='gID000')[0]

# Get the fractional coordinates of read configuration
R0 = configuration0.fractionalCoordinates()

# Get the elements
elements = configuration0.elements()

# Get the lattice
lattice = configuration0.bravaisLattice()
```

(continues on next page)

(continued from previous page)

```

# From the lattice extract unit cell volume and length of lattice vector in z-direction
volume = lattice.unitCellVolume()

vectors = lattice.primitiveVectors()
c = vectors[2][2]

# Get the calculator
calculator = configuration0.calculator()

# Number of atoms
numberOfAtoms = len(R0)

# Fractional displacement in the +/- z direction
delta_z = 0.01

# Array for storing the calculated Born Charges
bornCharges = numpy.zeros(numberOfAtoms)


# Loop over atoms in unit cell
for nAtom in range(numberOfAtoms):
    # Loop over displacement in positive/negative z-direction

    # List with polarization values
    Pz = []
    for s in [1,-1]:
        # Make a copy of the initial coordinates
        R = R0.copy()

        # Displace z-coordinate if atom 'nAtom'
        R[nAtom,2] += s*delta_z

        # Make a new configuration with the displaced atom
        configuration = BulkConfiguration(bravais_lattice=lattice,
                                          elements=elements,
                                          fractional_coordinates=R)

        # Set the calculator using the saved configuration as initial state
        configuration.setCalculator(calculator,initial_state=configuration0)

        # Update the configuration (DFT calculation)
        configuration.update()

        # Calculate polarization. Only use fine k-sampling in the z-direction
        polarization = Polarization(configuration=configuration,
                                    kpoints_a=MonkhorstPackGrid(2,2,2),
                                    kpoints_b=MonkhorstPackGrid(2,2,2),
                                    kpoints_c=MonkhorstPackGrid(5,5,20),
                                    )

        # Print polarization
```

(continues on next page)

(continued from previous page)

```
        nlprint(polarization)

        # Get the total cartesian polarization
        Pt = polarization.totalCartesianPolarization()

        # Append the z-component to the Pz list
        Pz.append(Pt[2])

    # Make a finite difference approximation for the derivative
    dP = (Pz[0]-Pz[1])/(2*delta_z*c)

    # Calculate Born charge
    born_charge = volume*dP

    # Add the value (in units of electron charge) to the list 'bornCharges'
    bornCharges[nAtom] = born_charge.inUnitsOf(elementary_charge)



# Finally print out the results
print('')
print('+-------------------------------+')
print('| Born effective charges (e)    |')
print('+-------------------------------+')

for nAtom in range(numberOfAtoms):
    print('  %2s' %elements[nAtom].symbol() + '  :      %4.4f       '
↪%bornCharges[nAtom]    )

print('+-------------------------------+')
print('  Sum :      %4.4f       ' %numpy.sum(bornCharges))
print('+-------------------------------+')
```

- The script starts reading the results from the previous calculation. This will serve as a good starting guess for the DFT calculations to be performed later in the script (where the atoms are displaced).
- Extract the fractional coordinates, the list of elements, the lattice, and the calculator from the configuration, and define a few convenient variables such as the unit cell volume and the length of the C lattice vector.
- The parameter **delta_z** is the fractional displacement when calculating the derivatives.
- There are two for loops in the script. The outer loop runs over the atoms in the unit cell, and the inner one (over the variable s) performs the displacements in the positive and negative z-direction.
- At the end of the script, the results are printed out.

Running the script

- Download the above `born_charges.py` script and save it in the **Project Folder**.
- Drag the script to the **Job Manager** to run it.
- When the job is finished, the calculated Born effective charges are written in the `born_charges.log` file.
- As indicated by the calculated sum of the Born effective charges, the calculated values fulfill the acoustic sum rule $\sum_s Z^*_{s,ij} = 0$ with only a small error.

You have already got all the components you needed to compute to calculate e_{33}. By substituting the computed values in the formula for e_{33}, you will obtain $e_{33} = 1.464$ C/m^2, which is in good agreement with the value 1.4849 C/m^2 calculated above and with the value found in the literature (1.464 C/m^2)[Page 1569, 1].

References

3.11.16 Formation energies of charged defects - manual workflow

Version: 2016.4

Downloads & Links

`bulk-As.py` `bulk-GaAs.py` `big-bulk-GaAs.py` `vacancy-calc.py` `formation-energies.py`

In this tutorial you will learn how to calculate formation energies of neutral and charged defects. Specifically, we will work with GaAs, which supports various charge states in both As and Ga vacancies.

You might also want to consult the introductory tutorial on formation energies, which focuses on bulk systems and neutral defects: *Calculation of Formation Energies*.

Note: In the O-2018.6 release, we introduced a so-called *Study Object* to automate the workflows described in this tutorial. That is now the recommended way of studying charged point defects in QuantumATK, and if you have access to O-2018.06 or newer, you should go to the corresponding tutorial *Formation energies and transition levels of charged defects*. Users of previous versions can still do the calculations and analysis using the more complex workflow, that is described here.

Procedure for calculating the formation energy

The formation energy E^f of a defect X is defined as the energy difference between the investigated system and the components in their reference states. This is analogous to the formation energy of a bulk material, but for charged defects it is less clear how to choose the reference states. Here we follow the procedure outlined in the review by C. Freysoldt *et al.*[1]:

$$E^f[X^q] = E_{tot}[X^q] - E_{tot}[bulk] - \sum_i n_i\mu_i + q\left(E_{VBM} + \mu_e\right) + E_{corr}.$$

[1] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and ChrisăG. Van De Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86(1):253–305, 2014. doi:10.1103/RevModPhys.86.253.

$E_{tot}[X^q]$ is the total energy of the defect state with charge q, which is given in units of the elementary charge ($e > 0$), $E_{tot}[bulk]$ is the total energy of a bulk unit cell of the same size as the defected one, and $n_i\mu_i$ is the reference energy of n_i added atoms of element i at chemical potential μ_i. The term in parenthesis accounts for the chemical potential of the electron(s) involved in charging the defect. E_{VBM} is the valence band maximum as given by the ATK band structure calculation for the bulk material of study (which is bulk undefected GaAs in this tutorial), and μ_e is the electron chemical potential defined here with respect to the top of the corresponding valence band. The μ_e parameter can then be treated as a free parameter, allowing to account for a shift of the Fermi level, e.g., due to doping. Note that $\mu_e = E_{gap}/2$ corresponds to the undoped semiconductor case, where E_{gap} is the intrinsic semiconductor band gap. Finally, E_{corr} is a sum of relevant correctional terms. In our case, it corrects for electrostatic interactions between periodic images of the charged defects. Due to the long-ranged nature of the Coulomb interaction, it is very difficult to work with systems that are large enough not to need this correction. More details of the correction scheme can be found in the *Appendix*.

In order to calculate the formation energy, we need to compute each of the terms outlined above. The workflow is listed below:

•Select the compound and defect to be investigated, and the computational model to be used (exchange-correlation, pseudopotentials, etc.).

•Relax the pristine bulk system using the chosen computational model.

•Determine the chemical potential of the elements in the defect. In this case we use the total energy of the bulk for the element removed to create the vacancy.

•Setup the undefected supercell to calculate $E_{tot}[bulk]$ and E_{VBM}.

•Introduce the (charged) defect and relax the system to obtain $E_{tot}[X^q]$.

•Calculate the correction terms in E_{corr} and finally the formation energy.

Neutral As vacancy in GaAs

As an example of a defect, we will calculate the formation energy of the As vacancy in GaAs, a vacancy which is capable of supporting many different charge states. However, we will first focus on the neutral vacancy defect, V^0, before adding charge. In this case, the formation energy equation reduces to

$$E^f[V^0] = E_{tot}[V^0] - E_{tot}[As] - (-1)\mu_{As}.$$

The first two terms are straightforward to calculate, though possibly time-consuming. Conversely, the third term is the reference chemical potential for As, and our choice here has a direct impact on the absolute value of the computed formation energy for the defect. The most obvious choice here is bulk As, which is what we will use, but other choices may be sensible representations of specific situations. We will not go into details of the matter.

Total energies will be calculated using DFT with the PBESol exchange-correlation functional, a GGA-level approximation designed to work well for solids[2].

[2] J.ăP. Perdew, A.ăRuzsinszky, G.ăI. Csonka, O.ăA. Vydrov, G.ăE. Scuseria, L.ăA. Constantin, X.ăZhou, and K.ăBurke. Restoring the Density-Gradient Expansion for Exchange in Solids and Surfaces. *Phys. Rev. Lett.*, 100(13):136406, 2008. doi:10.1103/PhysRevLett.100.136406.

Optimized bulk geometry

We first need to relax the minimal unit cell and internal coordinates of the GaAs and As bulks. For the latter, we also need the total energy per As atom. Use a DFT calculator with PBESol and the **OptimizeGeometry** object to relax the stress and forces on both bulks. This is described in detail elsewhere, e.g. in the tutorial *Calculation of Formation Energies*. For your convenience, we have also provided the scripts here:

`bulk-GaAs.py`

`bulk-As.py`

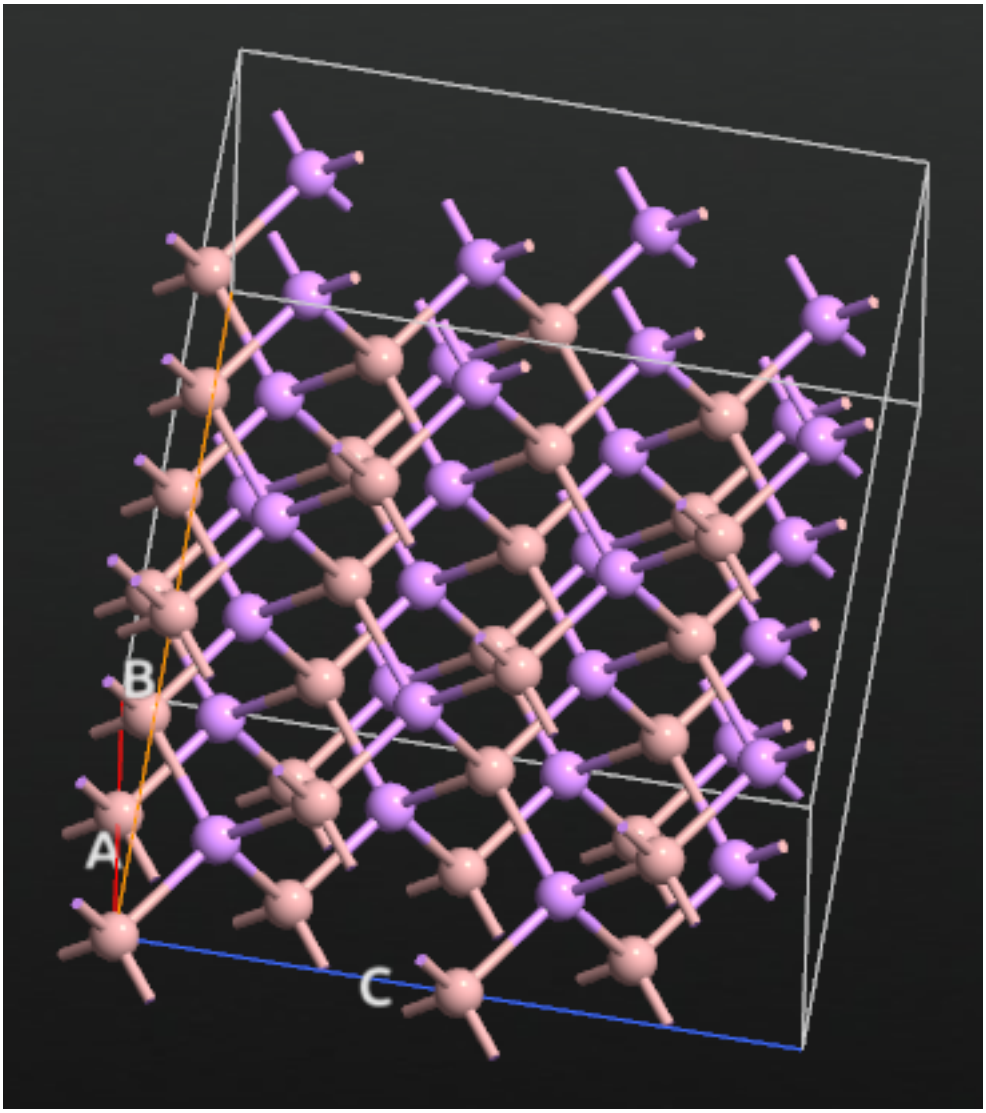
Larger supercell for bulk and defect calculations

Now find the optimized *BulkConfiguration* on the *LabFloor*, it should have an ID of gID001, and drop it on the **Builder** to set up a new *BulkConfiguration* with the optimized lattice constant. You may also find the optimized lattice parameter in the text output. However, we still need to make a bigger cell, and we would also like to have an orthogonal cell, as it will make the electrostatic corrections simpler.

- To do this, select the GaAs system in the *Stash* and go to *Bulk Tools* → *Supercell* on the right, choose *Conventional* and click *Transform*. The GaAs is now represented by a, much larger, cubic cell.
- We still need to make it bigger in order to get somewhat reasonable results for the defect, so go to *Bulk Tools* → *Repeat* and increase all three numbers to 2 and click *Apply*.
- Now send this configuration to the **Script Generator** and add a **New calculator**, a **Density of States** object and a **TotalEnergy** analysis object.
- Change exchange-correlation functional to PBES (for PBESol) and the k-point sampling to 4x4x4, which is adequate due to the large computational cell. Change k-points sampling in **Density of States** object to 8x8x8 k-points.
- Now do the calculation - It should take about 10 minutes.

`big-bulk-GaAs.py`

> **Warning:** Please note that the 2x2x2 supercell used here is too small to get reliable (publication quality) results: The defect concentration is very high, and there will be (artificial) elastic interaction between the defects. At the end of this tutorial we will investigate how results change for a 3x3x3 cell.

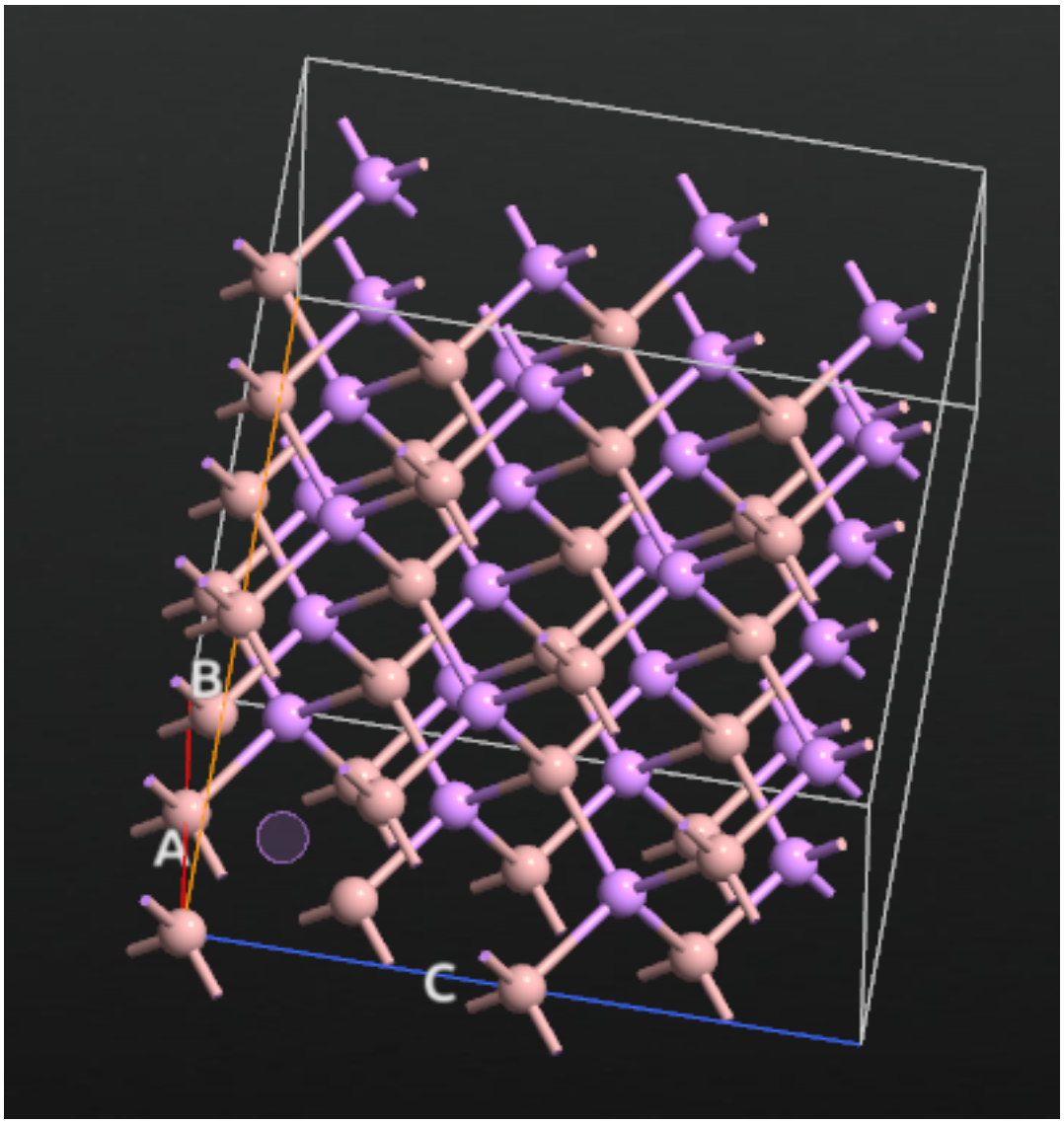


Introducing a vacancy

To set up the neutral defect calculation you can either use the script `vacancy-calc.py` or construct a script on your own. In the latter case, return to the Builder and select the 2x2x2 orthogonal unit cell in the Stash. Then do the following:

- Select an As atom, preferably the one closest to the origin, and convert it to a ghost atom by clicking the icon in the top bar.
- Send this configuration to the **Script generator** and add a **New Calculator**, an **OptimizeGeometry** object and a **TotalEnergy** object.
- Change the calculator settings to match those used for the pristine 2x2x2 GaAs. Default settings are fine for the other two objects. Run the calculation – it may take a couple of hours due to the geometry optimization.

Attention: Be aware that, in general, we need to account for the basis set superposition error, by leaving the basis functions associated with an atom when removing it to create the vacancy. In practice, this is achieved by making the atom into a ghost atom instead of removing it. This improves the result, perhaps only slightly, for most local basis sets, and carries an insignificant increase in computational cost.



Formation energy

We get total energies $E_{tot}[GaAs]$ = -7460.13 eV for the defect-free bulk configuration and $E_{tot}[V^0]$ = -7285.59 eV for the configuration with the neutral As vacancy, and combined with the result for the As bulk above, we find a formation energy for the neutral As vacancy of 3.22 eV. You can also use the script `formation-energies.py` to compute $E^f[V^0]$.

Charged As vacancies in GaAs

Now that we have calculated the formation energy of the neutral defect, we turn our attention to incorporation of charges in our modeling. If we again focus on the As vacancy, the formation energy is computed according to:

$$E^f[V^q] = E_{tot}[V^q] - E_{tot}[GaAs] + \mu_{As} + q\,(E_{VBM} + \mu_e) + E_{corr}.$$

The first term is similar to what we have just calculated for the neutral defect, while the second and third terms are exactly the same as previously and do not need to be re-calculated. The valence band maximum is determined through a density of states (DOS) calculation for the undefected bulk, while μ_e will be treated as a free parameter. The last term, E_{corr} is calculated according to the scheme of Freysoldt et al.[3] , which also includes a small correction to the VBM, to account for the shift introduced by the defect, as discussed in the *Appendix*. Note that the correction term depends on the screening properties of the bulk material surrounding the charged defect. That requires knowing the dielectric constant value for the undefected bulk GaAs that can be either taken from experiment or computed as further explained in the *Appendix*.

Charged defects

The As vacancy can sustain a total of 5 charge states, including the neutral one considered above. We will now calculate the formation energies of these defects. For very large systems it may be worthwhile to use the relaxed state of the neutral defect as the initial guess for the charged ones, but this is not important for the system size used here.

The charges can either be added in the calculator when following the procedure described above, or by directly making copies of the script and adding `charge=q,` in the calculator object. You can also use the script `vacancy-calc.py`, which will take the `.nc` file of a large bulk supercell as input and add a defect before optimizing the geometry and computing the total energy.

Note: Remember to modified the entries `defect_charge` and `saved_filename` in the input section of script `vacancy-calc.py` to select the correct charge state and the name for the corresponding output file.

Formation energies

Now that we have all the pieces we need, we can calculate the full formation energies by extracting the total energies from the saved data files and calculating the correctional terms. For this, you can use the script `formation-energies.py`. It takes as input the `.nc` files with the relaxed defect states, as well as an undefected supercell of the same size including a valid DOS calculation. You will also need the dielectric constant, calculated using your chosen computational model, as detailed in this tutorial: *Optical Properties of Silicon*. We also show how to do it for GaAs in the *Appendix*.

[3] Christoph Freysoldt, Jörg Neugebauer, and ChrisăG. Van De Walle. Fully Ab Initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, 2009. doi:10.1103/PhysRevLett.102.016402.

Table 3.7: Formation energies (with periodic and band offset corrections included) for an As vacancy in a 2x2x2 GaAs supercell.

Charge state	Formation energy (with E_{corr} included)	Periodic correction	Band offset correction
+1	3.04 eV	0.13 eV	-0.03 eV
0	3.22 eV	0.00 eV	0.00 eV
-1	3.56 eV	0.13 eV	0.00 eV
-2	4.23 eV	0.54 eV	-0.03 eV
-3	5.18 eV	1.20 eV	-0.10 eV

The formation energies given above were calculated for an electron chemical potential at the VBM, that is, for $\mu_e = 0$ eV. Considering μ_e as a free parameter that shifts the Fermi level, we can also plot the formation energy for each defect as a function of the electron chemical potential. This would correspond to varying the doping level of the material. The effect is illustrated in the figure below, where the most stable charge state at a given μ_e is identified with the lowest line. This shows that in p-type doped GaAs, corresponding to $\mu_e = 0$ eV, the As vacancy is most stable with a positive charge. By increasing the electron chemical potential, the most stable charge state changes to neutral and from about $\mu_e > 0.35$ eV, the most stable state is negatively charged. In particular, the As vacancy with the charge $q = -2$ is the most stable defect in the intrinsic (undoped) GaAs for which $\mu_e = E_{gap}/2 \sim 0.75$ eV.

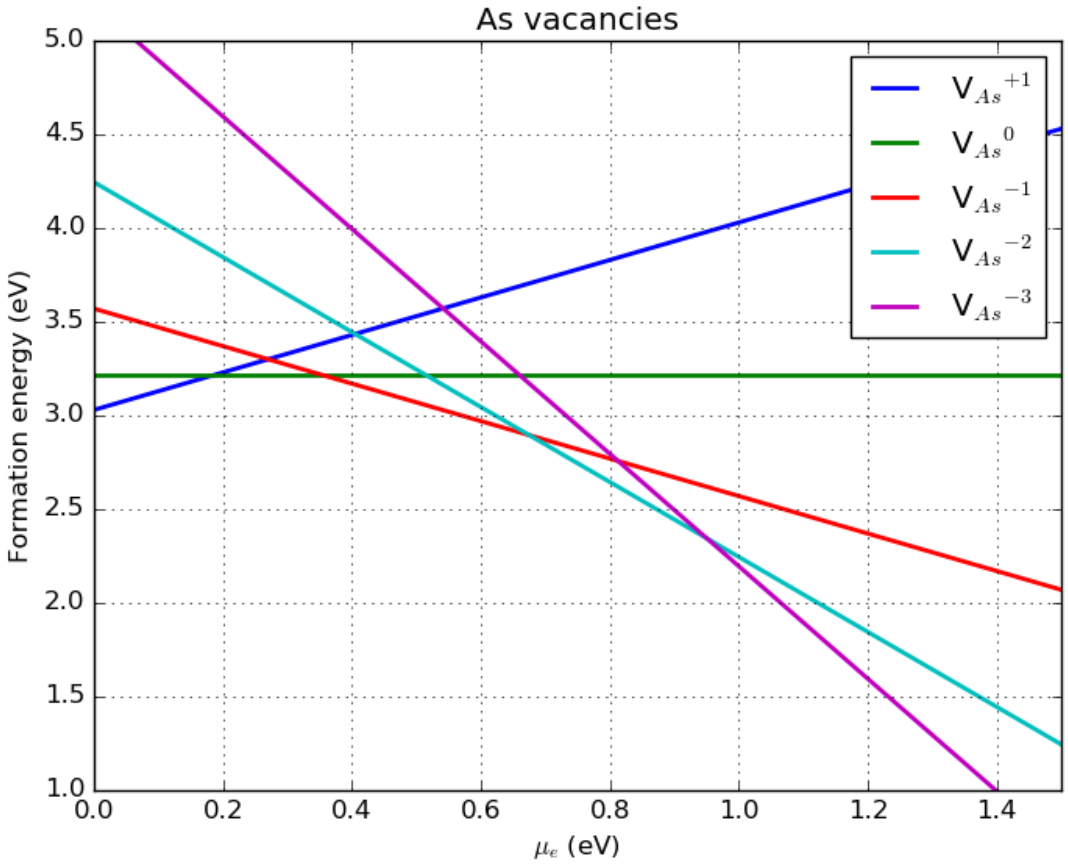


Fig. 3.112: Formation energy for various charge states of the As vacancy in GaAs as a function of μ_e (defined with respect to the VBM), with the VBM at zero, computed with a 2x2x2 orthogonal supercell.

Influence of system size

If we increase the supercell size to 3x3x3, and change the k-point grid to 2x2x2, we find the formation energies listed in the table below, together with literature values from[4], which uses the same size of supercell.

Table 3.8: Formation energies (with periodic and band offset corrections included) compared to literature.

Charge state	2x2x2	3x3x3	Reference (3x3x3)
+1	3.04 eV	2.95 eV	2.79 eV
0	3.22 eV	3.26 eV	3.25 eV
-1	3.56 eV	3.67 eV	3.33 eV
-2	4.23 eV	4.35 eV	4.52 eV
-3	5.18 eV	5.25 eV	5.86 eV

We see how the formation energies are slightly different, with the shift being about 0.04 eV for the neutral defect. The difference is slightly larger for the charged defects, which is due to a slight shift of the VBM, caused by a slightly different k-point density. In this case there is not perfect agreement with the literature reference, but it should be noted that the reference does not use a ghost atom, and uses an older electrostatic correction scheme and LDA for exchange-correlation.

The changes also influence the plot of formation energy versus μ_e; although the slope of the lines are the same, their intercepts are slightly different, leading to a significant change in the overall picture of defect stability. Specifically, the neutral state is now only stable in a very narrow window, and the negatively charged states are most stable from $\mu_e = 0.4$ eV.

Appendix

The periodic correction scheme

As already mentioned, we use the correction scheme proposed by Freysoldt *et al.*[Page 1595, 3] to correct for the electrostatic interaction between the periodic images of the charges and the band shift caused by introduction of the defect. This correction contains two terms:

$$E_{corr} = E_{lat} - q\Delta V_{q/b},$$

where E_{lat} corrects for the interaction between the periodic images of the charge and $q\Delta V_{q/b}$ corrects for the band shift. E_{lat} is calculated by considering the electrostatics of a model charge distribution and comparing the potential in the periodic cell with the potential where the boundary conditions are not periodic:

$$E_{lat} = \int_{\Omega} \left[\frac{1}{2} q(\tilde{V}_q^{lr} - V_q^{lr}) \right] d^3r.$$

V_q^{lr} is the long-range potential of the model charge distribution and $\tilde{V}_q^{lr}$ is the corresponding quantity in the periodic cell. $\Delta V_{q/b}$ is calculated from the shift in the electrostatic potential due to introduction of the defect:

$$\Delta V_{q/b} = \tilde{V}_{q/b} - \tilde{V}_q^{lr},$$

where $\tilde{V}_{q/b}$ is the difference between the electrostatic potential in the pristine bulk configuration and with the defect added. $\Delta V_{q/b}$ should be evaluated far from the defect, where it is constant, in order to get the correct offset in the zero-point reference energy . In our implementation, it is averaged over the five grid points furthest from the defect. We plot these quantities in the figure below, where $\Delta V_{q/b}$ is the black line:

[4] Fedwa El-Mellouhi and Normand Mousseau. Self-vacancies in gallium arsenide: An ab initio calculation. *Phys. Rev. B*, 71(12):125207, 2005. doi:10.1103/PhysRevB.71.125207.

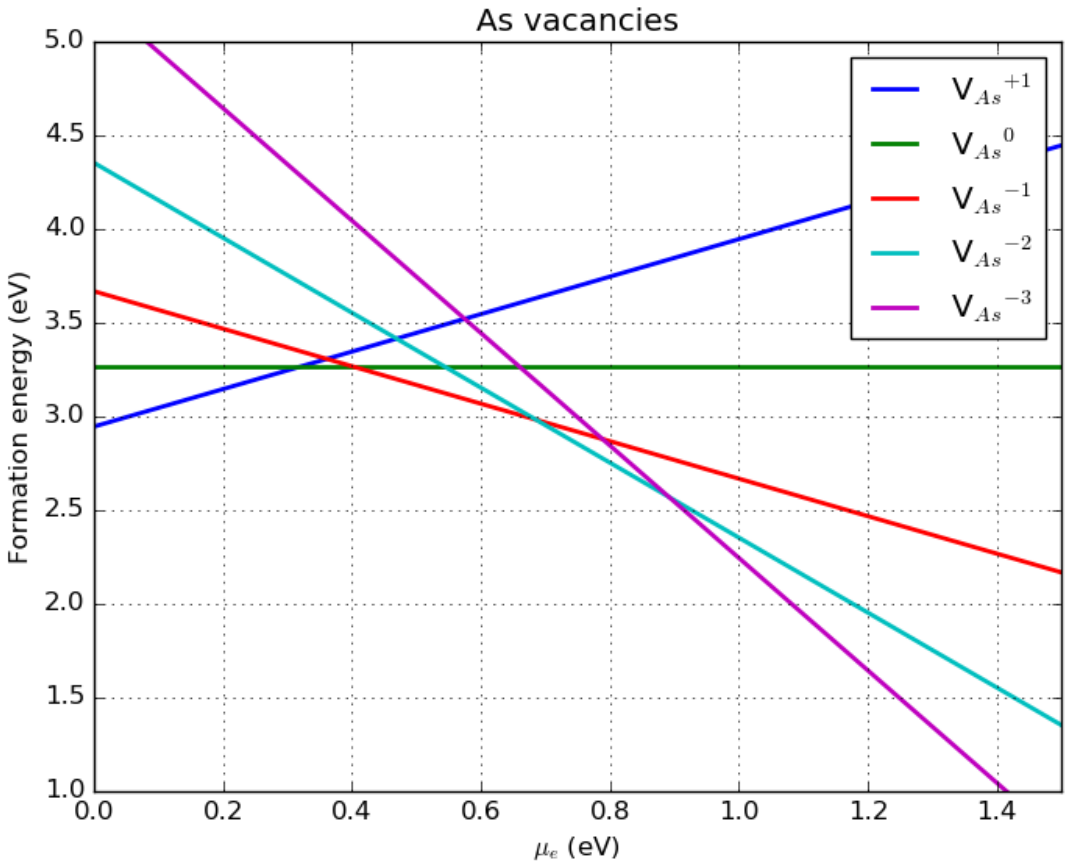


Fig. 3.113: Formation energy for various charge states of the As vacancy in GaAs as a function of μ_e (defined with respect to the VBM), with the VBM at zero, for a 3x3x3 orthogonal supercell.

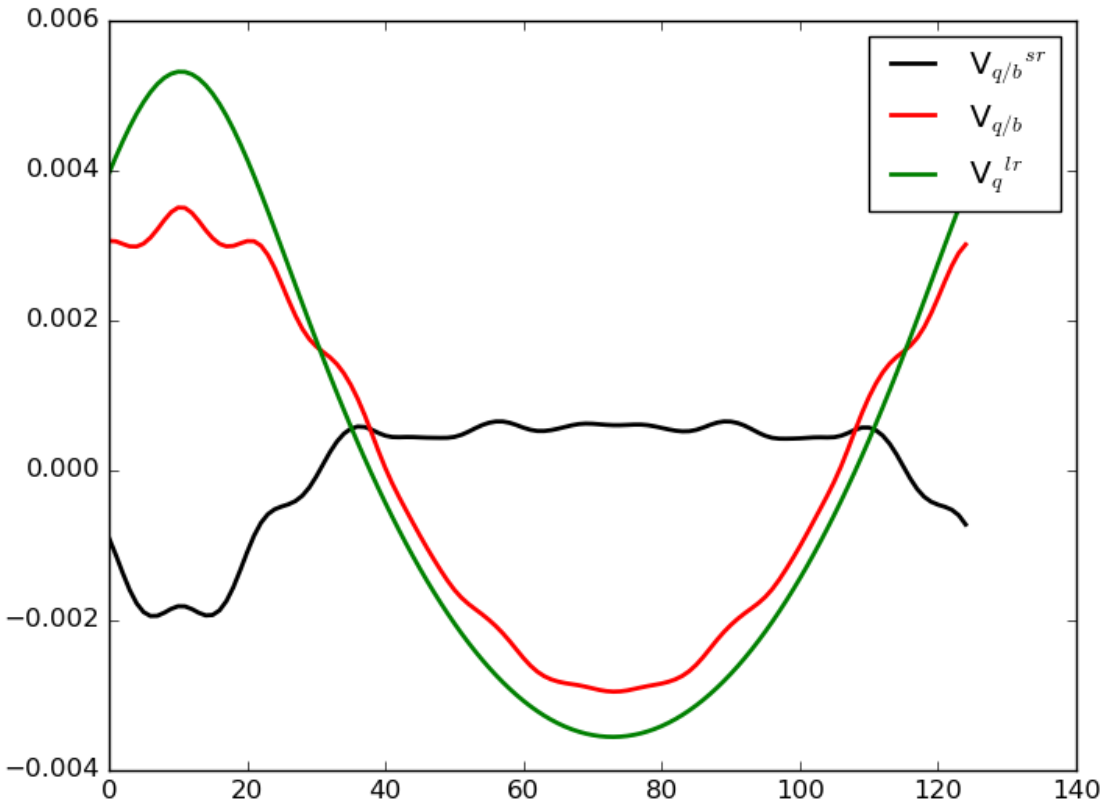


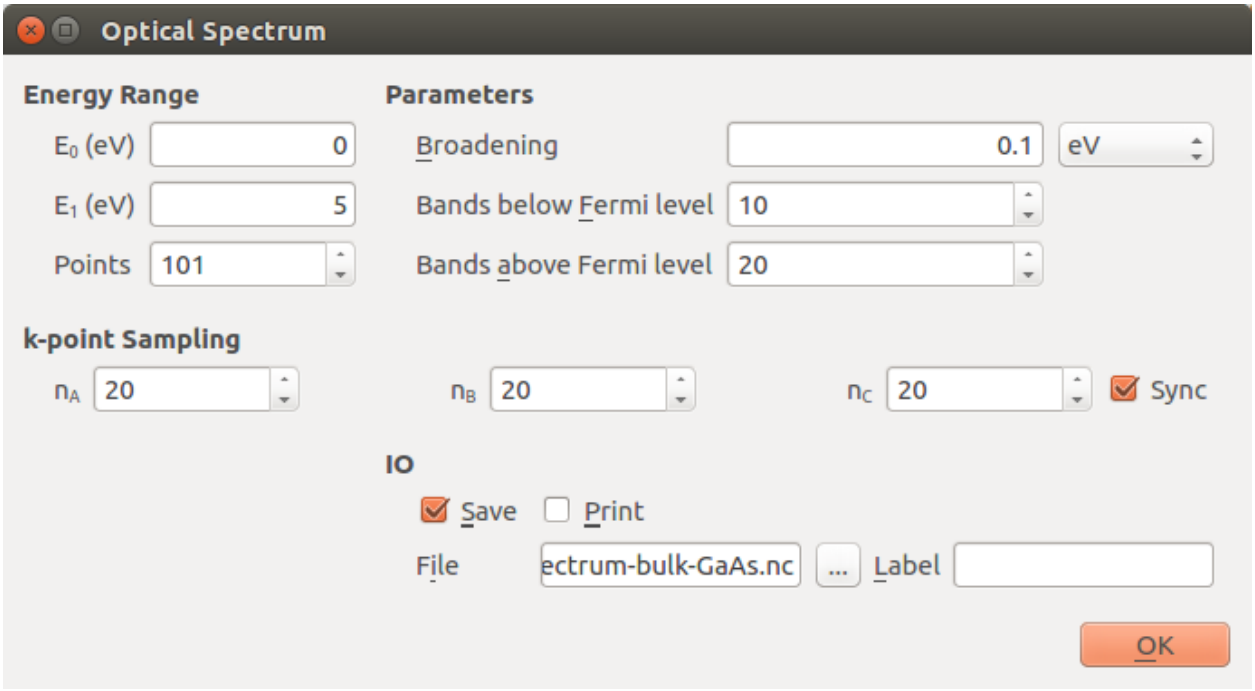
Fig. 3.114: Plot of the terms in the band offset correction. Green is the model potential from the defect, red is the change due to the introduction of the defect, and black is then the offset in the zero-point of the absolute energy.

Note that the obtained difference is defined between two *screened* potentials, see the reference[Page 1595, 3] and the script `formation-energies.py`, where the screening is due to the semiconductor material surrounding the charged defect. The dielectric screening is described by the static dielectric constant computed in the following section for bulk GaAs.

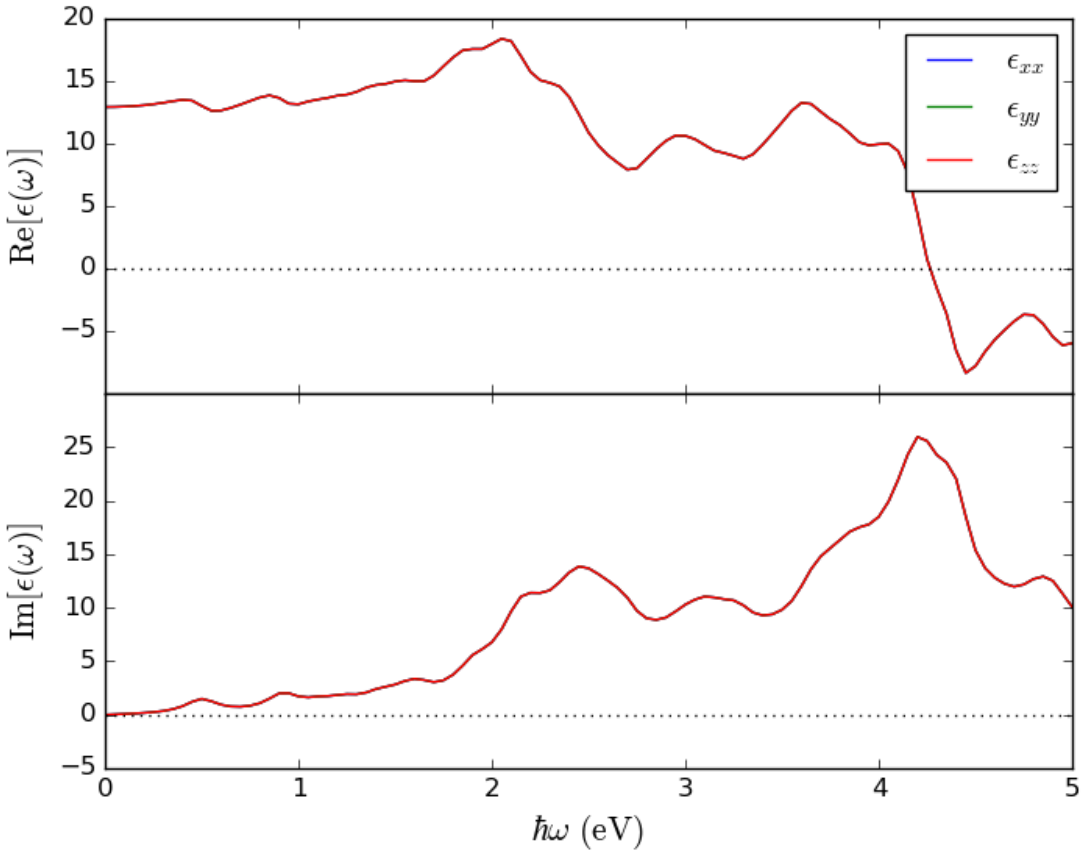
Dielectric constant of GaAs

To account for the screening effect of the semiconductor material (surrounding the charged defect) on the band offset correction, one should compute the dielectric constant to be used in the script `formation-energies.py`. The dielectric constant can be calculated through an *OpticalSpectrum* analysis on a primitive GaAs bulk cell. It is important that the calculation is well-converged with respect to k-points, and also that the number of bands in the OpticalSpectrum is adequate for sampling all relevant transitions. In this case, a 20x20x20 k-point grid, 10 bands below the Fermi level and 20 bands above is sufficient to ensure this.

Drag and drop your optimized GaAs bulk structure to the **Builder** and send it to the **Script Generator**. Add a *New Calculator* and set it to the same parameters as the previous calculation, except for the k-points, which must be 20x20x20. Add an *OpticalSpectrum* analysis object, and set it to 20x20x20 k-points, 10 bands below the Fermi level and 20 above. The box should now look as the one in the figure below.



Run the calculation, which should take no more than a few minutes. You should now see the *OpticalSpectrum* on the **LabFloor**, which you can open with the *Optical Spectrum Analyzer* on the right. This will open a window with a plot which should look like this:



In this case, we are only interested in the intercept of the upper plot, which looks to be about 13. To get the exact calculated value, open the *OpticalSpectrum* with the *Text Representation* and find the part where it says Real part of the dielectric tensor. Here you can find the calculated value for the dielectric constant at zero frequency to be 12.9, as seen in the figure below:

Text Representation - bulk-GaAs.nc gID010

```
 4.80000e+00   1.26758e+01   1.26758e+01   1.26758e+01
 4.85000e+00   1.29136e+01   1.29136e+01   1.29136e+01
 4.90000e+00   1.25123e+01   1.25123e+01   1.25123e+01
 4.95000e+00   1.13144e+01   1.13144e+01   1.13144e+01
 5.00000e+00   1.00541e+01   1.00541e+01   1.00541e+01


   Real part of the dielectric tensor.
Spin.Sum:
     energy          xx            yy            zz
 0.00000e+00   1.29316e+01   1.29316e+01   1.29316e+01
 5.00000e-02   1.29398e+01   1.29398e+01   1.29398e+01
 1.00000e-01   1.29650e+01   1.29650e+01   1.29650e+01
 1.50000e-01   1.30082e+01   1.30082e+01   1.30082e+01
 2.00000e-01   1.30714e+01   1.30714e+01   1.30714e+01
 2.50000e-01   1.31571e+01   1.31571e+01   1.31571e+01
```

References

3.11.17 Boron diffusion in bulk silicon

Version: 2016.0

In this tutorial we will use an Adaptive Kinetic Monte Carlo (AKMC) algorithm with ATK-DFT to investigate the diffusion of a single B atom in a bulk Si lattice. You can read more about the AKMC method in the tutorial *Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)*.

Boron is known to diffuse though silicon during ion beam implantation due to the presence of a large number of defects. Furthermore, boron is most stable at substitutional lattice sites. In this tutorial we will investigate if the B atom is mobile in a defect-free Si lattice at 300 K, with B in a substitutional lattice site as the initial state.

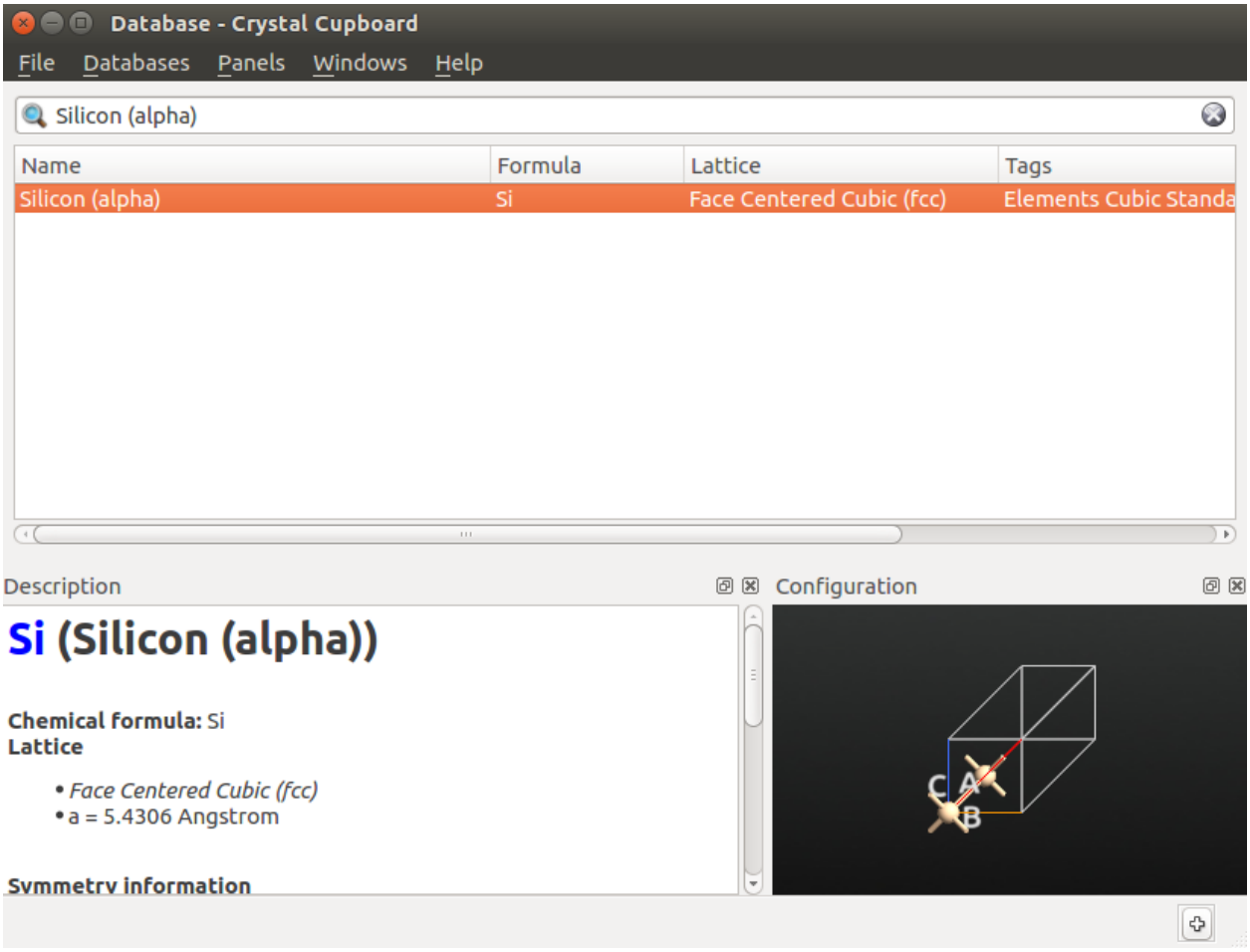
Attention: Note that this tutorial requires access to a computing cluster, as the calculation time exceeds several CPU-weeks. AKMC requires a minimum of 10^4–10^5 force and energy calculations to give meaningful results, making the length of the individual calculation very important for the overall duration.

Creating the B-doped Si crystal

We will need to create a bulk Si crystal with a cubic supercell with 64 atoms and then substitute one of them with B. However, first we need to find the correct lattice constant for Si with the computational model we will be using in this tutorial.

Optimize the Si bulk lattice

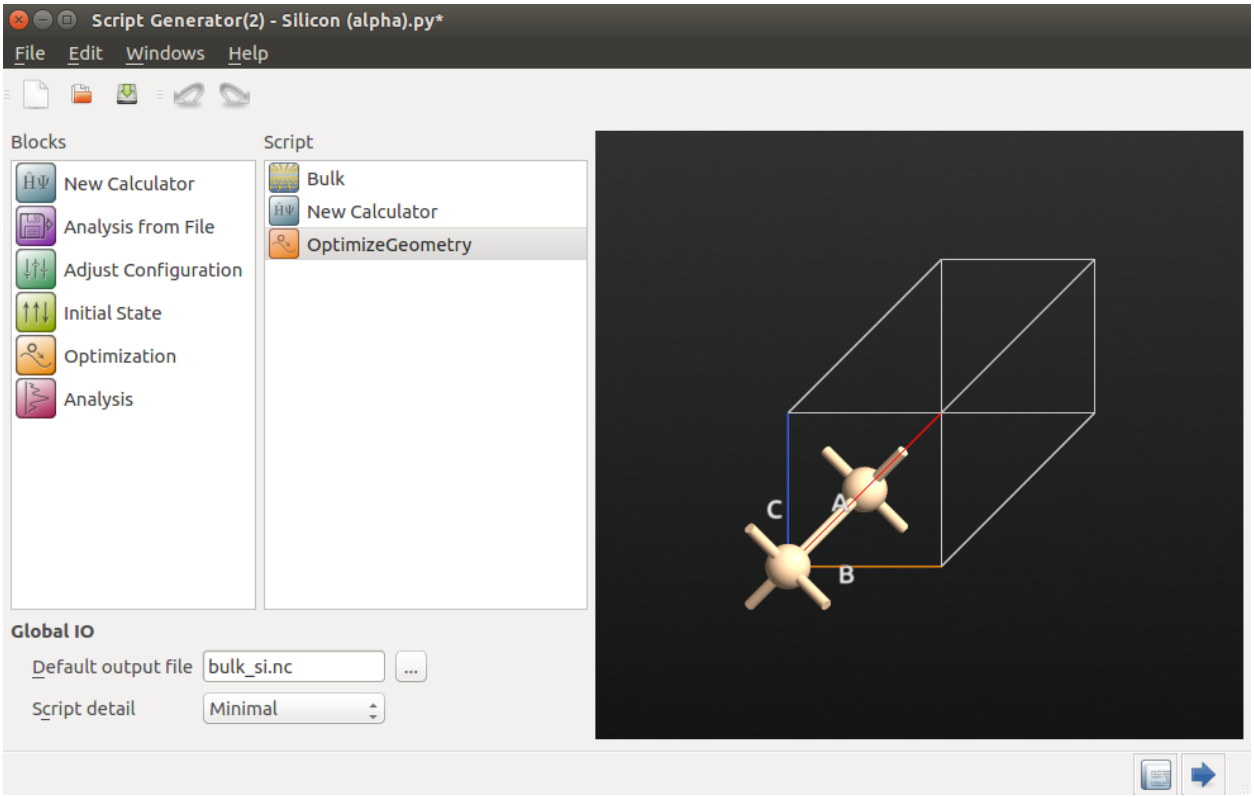
- Open the **Builder** and select *Add* → *From Database* to access QuantumATKs database of compounds.
- Search for Silicon (alpha) and add it to the stash by clicking the icon in the lower right corner.



•Send the configuration to the **Script Generator** (use the icon for this), and add the following blocks to the script:

- New Calculator
- OptimizeGeometry

Then change the default output file to `bulk_si.hdf5`.



Next, we will choose calculator settings that are optimized for speed rather than accuracy, in order to show the functionality of the AKMC algorithm with DFT. Always check the accuracy of your settings when you want publication-quality data.

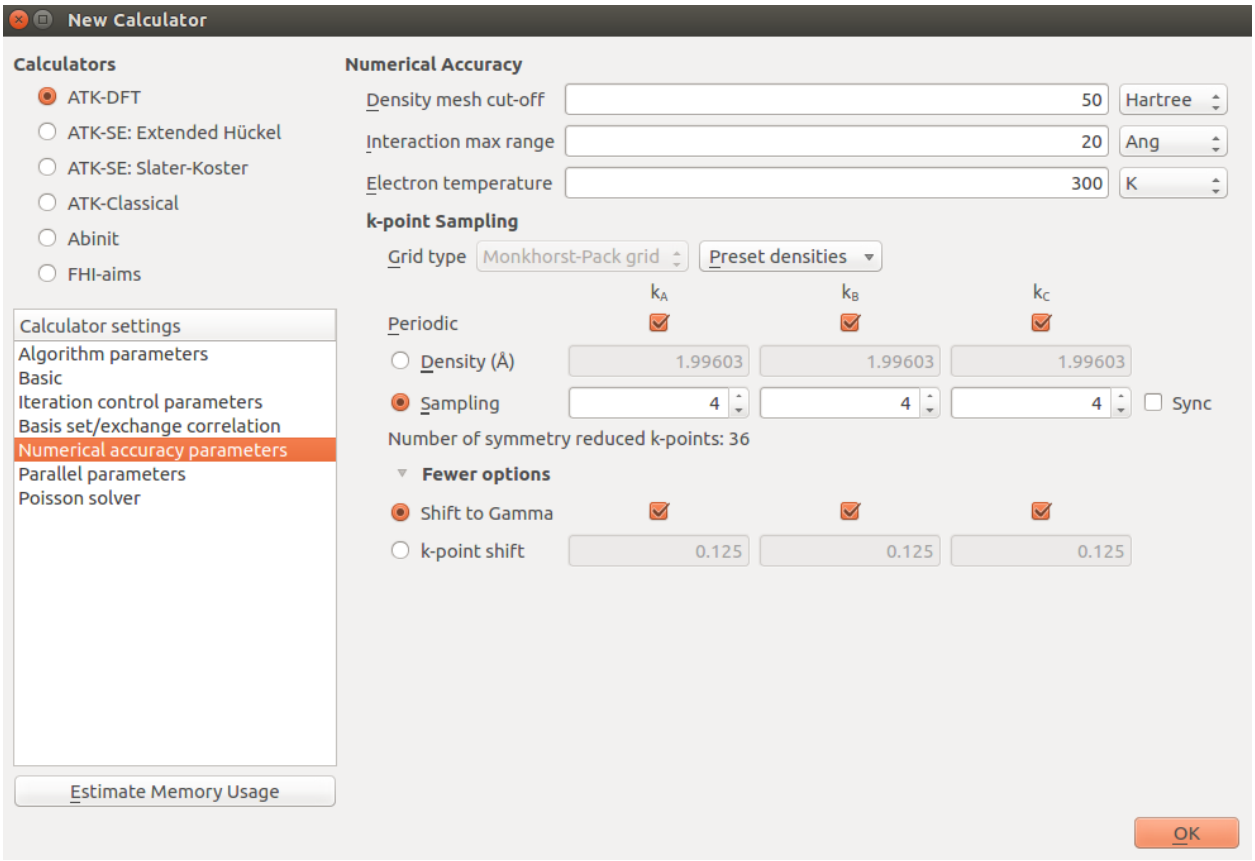
- **New Calculator**

  - Basis set: *SingleZetaPolarized*

  - Exchange-Correlation: *GGA.PBES* to use the PBEsol functional.

  - k-point sampling: *4x4x4* and make sure it is shifted to gamma by ticking the three boxes found by ticking **More options**. This is a very low k-point density, but equivalent to the sampling we will use during the AKMC, where we will use only the gamma point to speed things up.

  - Set the mesh cutoff to 50 Hartree.



•OptimizeGeometry

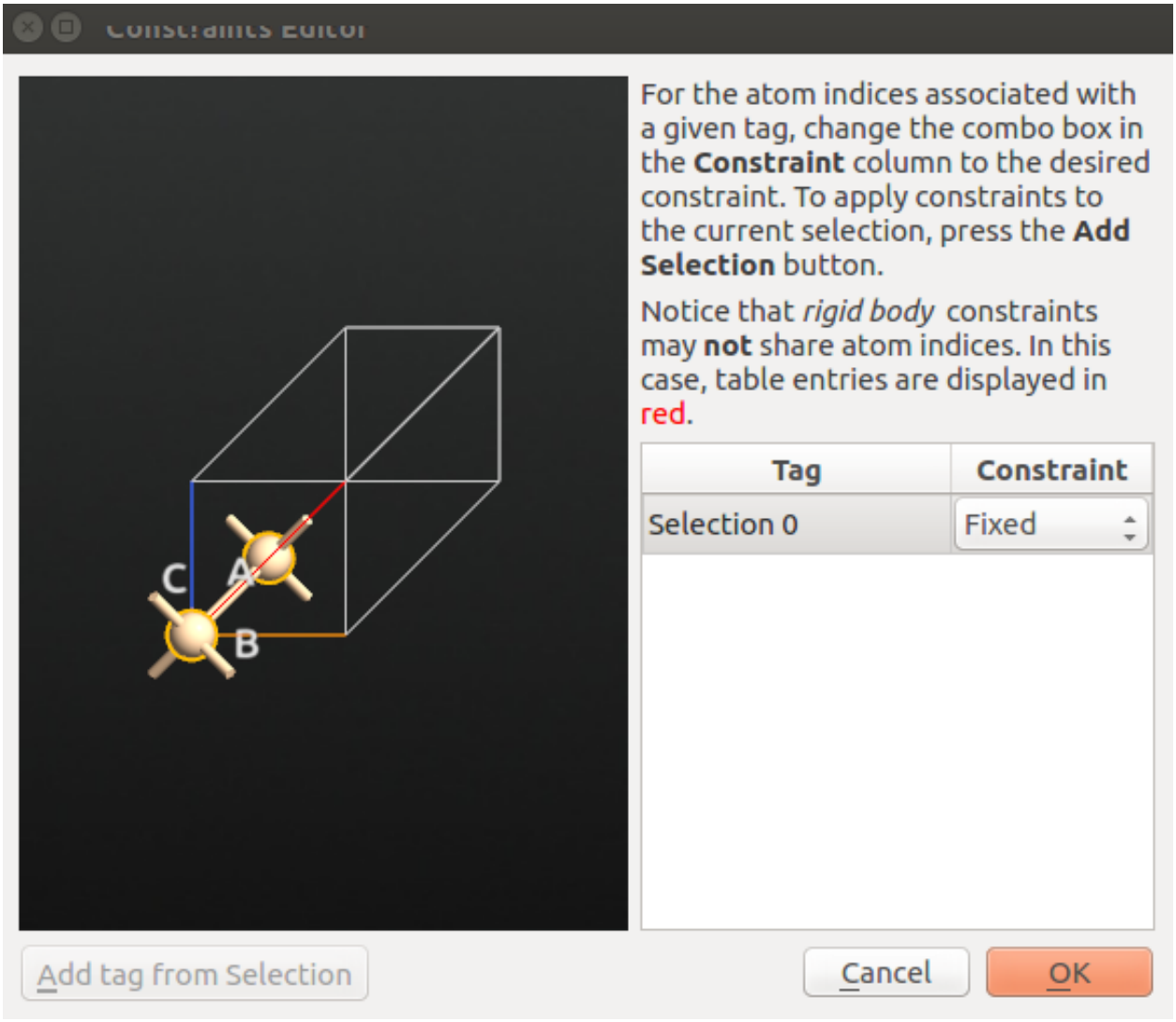
–Untick *Constrain Lattice Vectors*.

–Keep the fractional coordinates of Si atoms fixed:

∗open **Atomic Constraint Editor** and select one Si atom;

∗hold down `Ctrl` while selecting the other atom and click **Add tag from Selection**.

∗choose *Fixed* under **Constraint** and click OK.



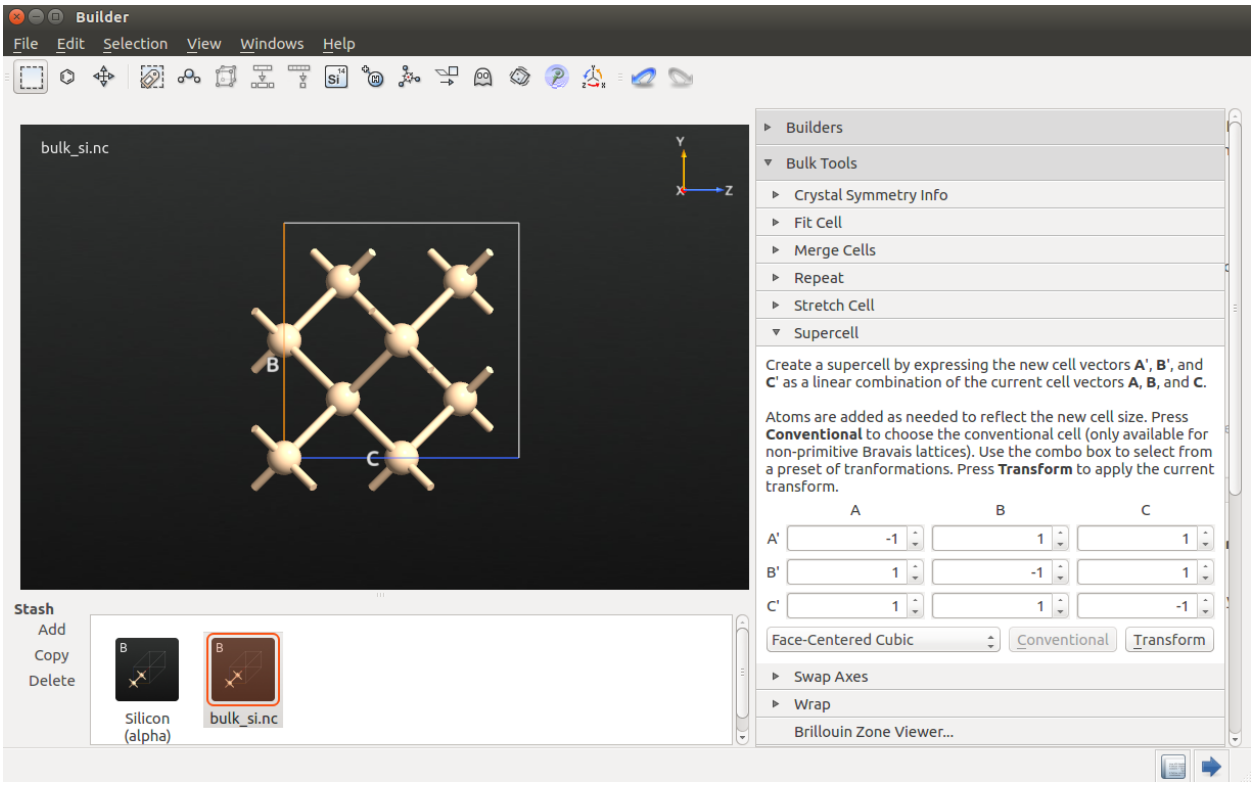
Leave everything else at default in the OptimizeGeometry block and send the script to the **Job Manager** to run the script. The calculation will take a few minutes.

Build the B-doped Si crystal structure

Once the calculation is finished, you can find the optimized Si bulk structure on the **LabFloor**.

•Select the optimized geometry and drop it on the **Builder**.

•Make the cell cubic by clicking on *Bulk Tools* → *Supercell*, then click on *Conventional* and then *Transform*. Note how the number of atoms has increased to 8.



•To make the 64 atom cell, click *Bulk Tools* → *Repeat* and increase A, B and C to 2 before clicking *Apply*.

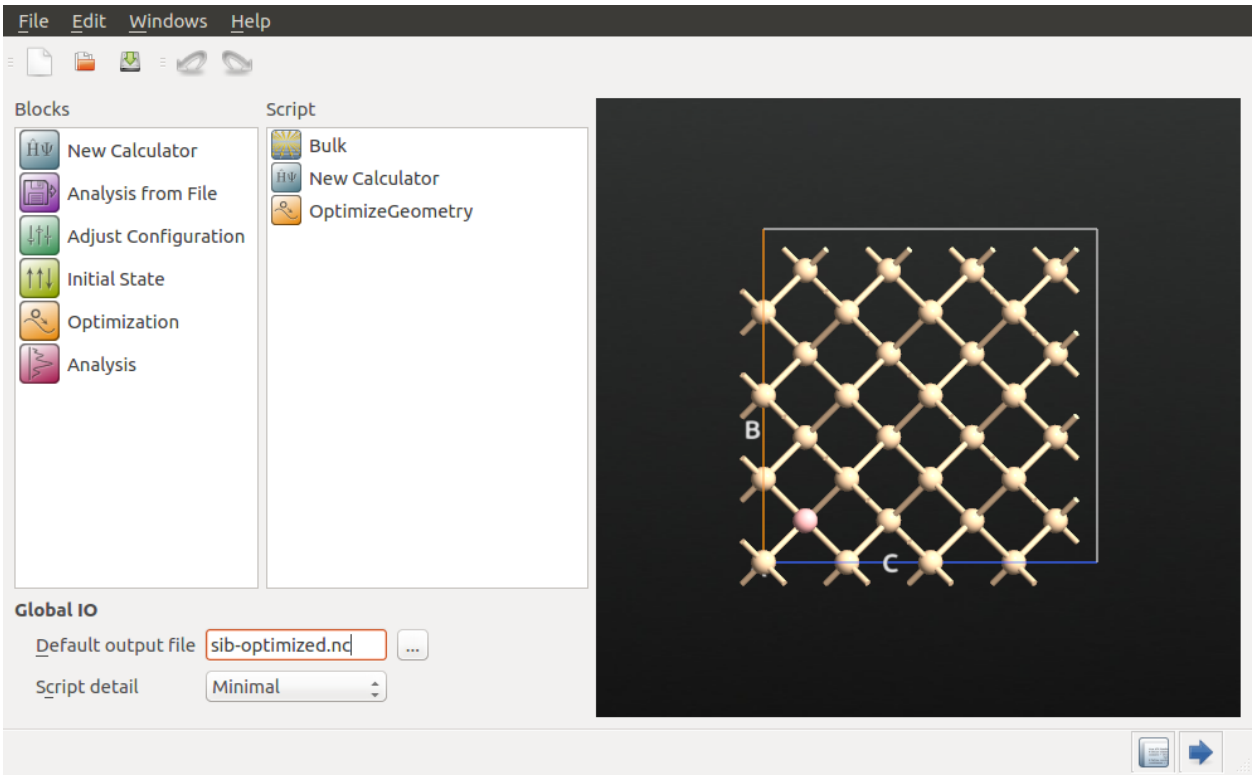
•Select the Si atom you would like to replace with B, e.g. atom number 8, close to the origin at [0.125, 0.125, 0.125] in fractional coordinates.

•Click the Periodic Table plugin and select B in the periodic table.

Setup the script

Send the structure to the **Script Generator**, change the default output file to `sib-optimized.hdf5`, and add the following blocks to the script:

- New Calculator
- OptimizeGeometry



Next, choose the following settings for the respective blocks.

•**New Calculator**:

•k-points sampling: *1x1x1*, sampling only the gamma point;

•basis set: *SingleZetaPolarized* for Silicon and *DoubleZetaPolarized* for Boron;

•apply the same settings as before for everything else.

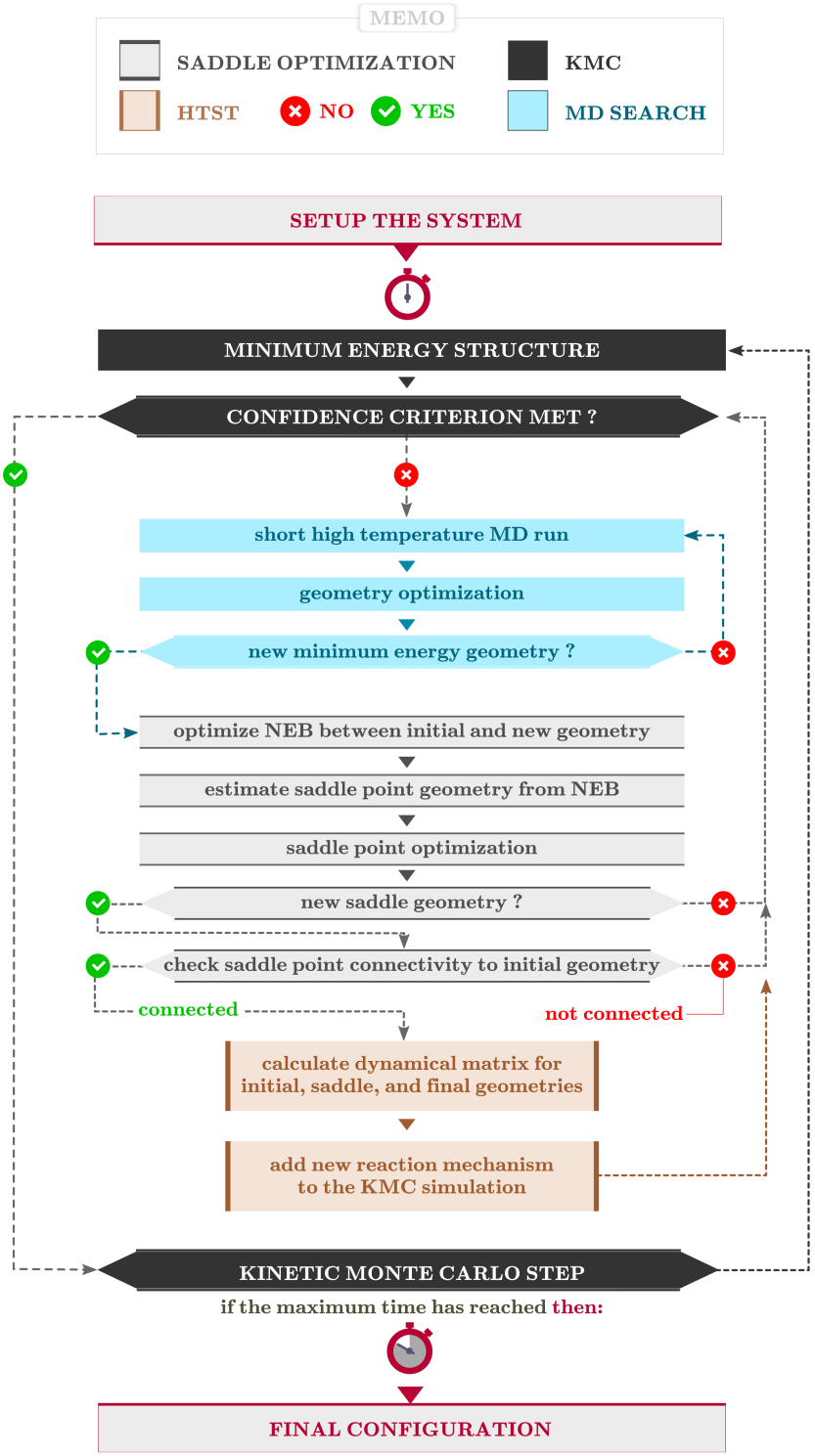
•**OptimizeGeometry**: change the convergence criterion to 0.01 eV/Å, and keep the default values for everything else.

Send the script to the **Job Manager** and run the script.

When the job has finished after about 10 minutes, depending on hardware, we are ready to do the actual AKMC simulation.

Running the AKMC simulation

Setting up the AKMC calculation requires using only a few lines of Python scripting, and has been prepared in the script `sib-with-akmc.py`. The different parts of the script will be explained below. Download the script, and make sure the file name in the beginning points at the file with your optimized *BulkConfiguration*.



Note: Support for setting up an AKMC calculation using the Script Generator is planned for QuantumATK 2017.

Details of the script

First, we change the logging behavior of QuantumATK to the verbosity mode *MinimalLog*, which will greatly reduce the information in the log file for each SCF cycle. This will improve readability of the log files from the AKMC simulation, while leaving essential SCF information available. The optimized configuration is then loaded from the saved `.hdf5`-file.

```
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Read in configuration
bulk_configuration = nlread('../sib-optimized.nc', BulkConfiguration)[-1]
```

See also:

For more information on how to tune the verbosity of QuantumATK logging output, see the Reference Manual entry on *setVerbosity()*. In particular, the options CALCULATOR_UPDATE=False and PROGRESS_BARS=False could be relevant here; they remove all the SCF information and progress bars from the output, respectively.

We then proceed to set up a calculator that is identical to the one used for structural relaxation, except for the addition of the *ParallelParameters* functionality, which is used to ensure that just one process will be used for each AKMC saddle search. For a bigger system, it might make sense to use more than one process per saddle search, but the optimal distribution depends on the setup and queuing rules of your supercomputer.

```
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#----------------------------------------
# Basis Set
#----------------------------------------
basis_set = [
    GGABasis.Boron_DoubleZetaPolarized,
    GGABasis.Silicon_SingleZetaPolarized,
    ]

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = GGA.PBES

k_point_sampling = MonkhorstPackGrid(
    na=1,
    nb=1,
    nc=1,
    shift_to_gamma=[True, True, True],
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=50.0*Hartree
    )
```

(continues on next page)

(continued from previous page)

```
parallel_parameters = ParallelParameters(
    processes_per_saddle_search=1,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    parallel_parameters=parallel_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
```

The final part of the script is the actual AKMC simulation. First, there are two blocks which check if a previous AKMC simulation has already been run, and reuse that information if possible, or initialize new objects if no previous results are present.

```
# ------------------------------------------------------------------
# AKMC
# ------------------------------------------------------------------

# Reusing existing MarkovChain object if it already exists, otherwise a new one is␣
↪created
if os.path.isfile('akmc_markov_chain.nc'):
    markov_chain = nlread('akmc_markov_chain.nc')[0]
else:
    markov_chain = MarkovChain(bulk_configuration, TotalEnergy(bulk_configuration).
↪evaluate())

# Reusing existing KMC object if it already exists, otherwise a new one is created
if os.path.isfile('akmc_kmc.nc'):
    kmc = nlread('akmc_kmc.nc')[0]
else:
    kmc = None
```

Next comes *SaddleSearchParameters*, where we restrict NEB calculations to no more than 5 images. For this system, we expect fairly simple reactions, which will be well-described with 5 images or less. For more complicated systems, more than 5 images per NEB might be needed. This is not a very expensive part of the overall calculation, but we need to optimize for speed when using DFT.

```
# Modify the default maximum number of NEB images to 5.
saddle_search_parameters = SaddleSearchParameters(max_neb_images=5)
```

Finally the AKMC calculation is set up, including the *SaddleSearchParameters* object, and then started in the final line. For solid state systems it is often a good approximation to assume the prefactor, which is very expensive to calculate. In this case we use a value of 10^{13} s^{-1}, which should be appropriate for this system. As AKMC is a stochastic method, it is important to do enough saddle searches to ensure the reaction space is adequately sampled. This is also why this script is designed to make additional runs straightforward to do.

```
# Setup the AKMC simulation.
akmc = AdaptiveKineticMonteCarlo(markov_chain,
```

(continues on next page)

(continued from previous page)

```
                                    calculator=calculator,
                                    kmc_temperature=300*Kelvin,
                                    md_temperature=3000*Kelvin,
                                    saddle_search_parameters=saddle_search_parameters,
                                    constraints=[15],
                                    kmc=kmc,
                                    confidence=0.99,
                                    assumed_prefactor=1e13/Second,
                                    write_searches=False,
                                    write_kmc=True,
                                    write_markov_chain=True,
                                    write_log=True)

# Run 200 saddle searches.
akmc.run(200)
```

Warning: In AKMC calculations, you must constrain at least one atom to avoid drift of the system during the high-temperature MD. However, this will introduce a slight error in the potential energy surface (PES) close to the constrained atom(s), so the constraint(s) should in general be applied as far as possible from where you expect transitions to happen. In this case the constraint is chosen to be an atom exactly half a supercell away from the B atom in all directions, to minimize the impact on any reactions involving B.

Note: For further information on the parameters in the AKMC calculation, see for example the tutorial: *Modeling Vacancy Diffusion in Si0.5 Ge0.5 with AKMC* . In the present tutorial we focus on those features which need special attention when a DFT calculator is used.

Running the AKMC job

The script needs to run on a supercomputer, as it takes up to two full days per saddle search. In this case we run 2x50 hours on 3 nodes, for a total of 48 cores.

Attention: If you modify the parallelization options to better fit your system and the architecture of your favorite supercomputer, remember to leave one process for the AKMC algorithm itself. If you have N available CPU cores, only N-1 cores are available for saddle searches.

Analyzing the results

The script generates the following output files:

•One log-file for each saddle search:

–`saddle_search_*_*.log`

•The central log-file for the AKMC algorithm:

–`akmc_log.hdf5`

•A file containing the Markov Chain object:

–akmc_markov_chain.hdf5

•If any KMC steps have been taken by the algorithm: A file containing the Kinetic Monte Carlo object:

–akmc_kmc.hdf5

•A file containing everything sent to std out:

–sib-with-akmc.log

```
4.7M Sep  4 12:54 saddle_search_0_141.log
4.8M Sep  4 13:06 saddle_search_0_136.log
5.3M Sep  4 14:17 saddle_search_0_144.log
3.9M Sep  4 14:23 saddle_search_0_150.log
620K Sep  4 14:23 akmc_markov_chain.nc
136K Sep  4 14:23 akmc_log.nc
 14M Sep  4 14:44 saddle_search_0_118.log
2.8M Sep  4 14:48 saddle_search_0_142.log
6.1M Sep  4 14:48 saddle_search_0_149.log
4.9M Sep  4 14:48 saddle_search_0_153.log
3.6M Sep  4 14:49 saddle_search_0_148.log
3.2M Sep  4 14:49 saddle_search_0_140.log
4.3M Sep  4 14:50 saddle_search_0_147.log
3.1M Sep  4 14:50 saddle_search_0_139.log
7.5M Sep  4 14:51 saddle_search_0_151.log
3.2M Sep  4 14:51 saddle_search_0_146.log
3.2M Sep  4 14:53 saddle_search_0_145.log
5.2M Sep  4 14:53 saddle_search_0_155.log
124M Sep  4 14:53 sib-with-akmc.log
9.0M Sep  4 14:53 saddle_search_0_98.log
 20M Sep  4 14:53 saddle_search_0_87.log
 19M Sep  4 14:53 saddle_search_0_82.log
 21M Sep  4 14:53 saddle_search_0_66.log
 20M Sep  4 14:53 saddle_search_0_58.log
 20M Sep  4 14:53 saddle_search_0_50.log
128K Sep  4 14:53 saddle_search_0_176.log
146K Sep  4 14:53 saddle_search_0_175.log
424K Sep  4 14:53 saddle_search_0_174.log
494K Sep  4 14:53 saddle_search_0_173.log
```

In this case, no KMC steps have been taken, so there is no file named `akmc_kmc.hdf5`. This is also why all the saddle search files have a 0 as the first number; they have all been started from the original initial state. Many of the saddle

search files have the same time-stamp, as they have been killed by the queuing system on the cluster.

Inspecting the AKMC log-file

The AKMC log file can be most easily inspected using *Text Representation* in the QuantumATK Panel bar. This will bring up the following window:

```
# Title: akmc_log.nc - gID000
# Type:  AKMCLog
state id search number confidence  message
       0             0   0.000000  Found new state
       0             1   0.000000  The saddle point optimization did not converge.
       0             2   0.000000  Found new state
       0             3   0.000000  Found new state
       0             4   0.000000  The saddle point does not connect back to the initial minimum.
       0             5   0.000000  Found new state
       0             6   0.000000  Found new state
       0             7   0.000000  Found new state
       0             8   0.000000  Found new state
       0             9   0.000000  Found new state
       0            10   0.000000  Found new state
       0            11   0.000000  Found new state
       0            12   0.000000  Found new state
       0            13   0.000000  Found new state
       0            14   0.000000  Found new state
       0            15   0.000000  The saddle point does not connect back to the initial minimum.
       0            16   0.000000  Found new state
       0            17   0.000000  The saddle point optimization did not converge.
       0            18   0.000000  Found new state
       0            19   0.000000  The saddle point does not connect back to the initial minimum.
       0            20   0.000000  The saddle point does not connect back to the initial minimum.
       0            21   0.000000  The saddle point does not connect back to the initial minimum.
       0            22   0.000000  The saddle point does not connect back to the initial minimum.
       0            23   0.000000  The saddle point does not connect back to the initial minimum.
       0            24   0.000000  The saddle point does not connect back to the initial minimum.
       0            25   0.000000  The saddle point does not connect back to the initial minimum.
       0            26   0.000000  The saddle point does not connect back to the initial minimum.
       0            27   0.000000  The saddle point does not connect back to the initial minimum.
       0            28   0.000000  The saddle point does not connect back to the initial minimum.
       0            29   0.000000  The saddle point does not connect back to the initial minimum.
       0            30   0.000000  The saddle point does not connect back to the initial minimum.
       0            31   0.000000  Found new state
       0            32   0.000000  Found new state
       0            33   0.000000  The saddle point does not connect back to the initial minimum.
       0            34   0.000000  The saddle point does not connect back to the initial minimum.
       0            35   0.000000  The saddle point does not connect back to the initial minimum.
       0            36   0.000000  The saddle point does not connect back to the initial minimum.
       0            37   0.000000  Found new state
       0            38   0.000000  Found new state
       0            39   0.000000  Found new state
       0            40   0.000000  The saddle point optimization did not converge.
       0            41   0.000000  The saddle point does not connect back to the initial minimum.
       0            42   0.000000  Found new state
       0            43   0.000000  The saddle point does not connect back to the initial minimum.
       0            44   0.000000  The saddle point does not connect back to the initial minimum.
       0            45   0.000000  Found new state
       0            46   0.000000  The saddle point does not connect back to the initial minimum.
       0            47   0.000000  Found new state
       0            48   0.000000  The saddle point optimization did not converge.
       0            49   0.000000  Found new state
       0            50   0.000000  The saddle point optimization did not converge.
       0            51   0.000000  Found new state
       0            52   0.000000  The saddle point optimization did not converge.
       0            53   0.022956  Found saddle connecting to state 3 again
       0            54   0.018549  Found new state
       0            55   0.018491  Found new state
       0            56   0.018470  Found new state
```

This shows information about each completed saddle search.

•state id is the id number of the state, which the saddle searches are originating from – in this case state 0, which is the

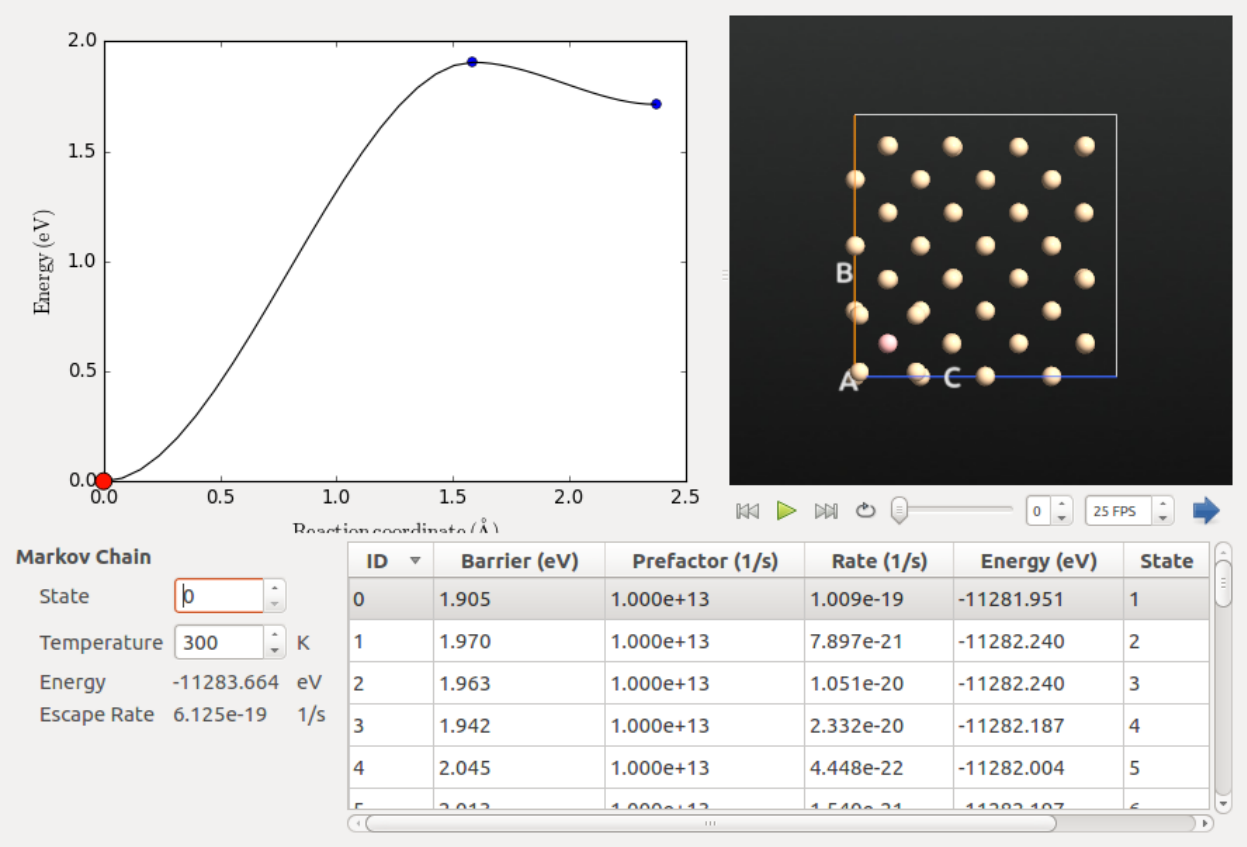
original initial state.

- search number is the id number of the saddle search, corresponding to the second number in the name of the log files.
- confidence is the confidence, on a scale from 0 to 1, that the current initial state has been adequately sampled.
- message is a short message describing the result of the saddle search.

We see that many saddle searches find new states and that about as many find saddle points which are not connected to the initial state. The former indicates that there are many saddle points with similar barriers, while the latter indicates that the potential energy surface (PES) is somewhat complicated. The confidence remains low because many searches discover new states, while only one search finds an already known state. This indicates that there are many relevant states, and the algorithm thus requires much more data before the PES has been adequately sampled around our original initial state and the first KMC step can be performed.

Inspecting the Markov Chain

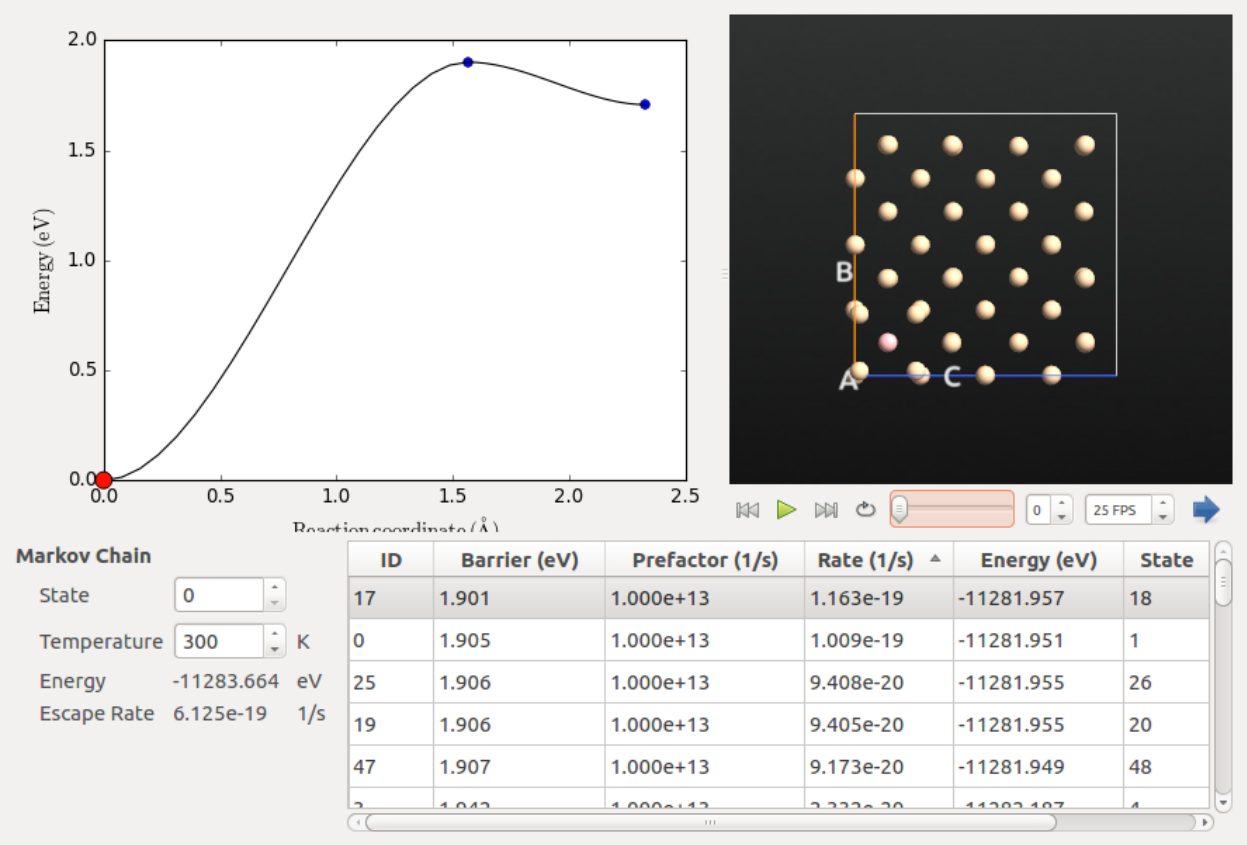
The Markov Chain is a list of all the discovered states and the connections between them. It can be visualized using the **Markov Chain analyzer** plugin in the QuantumATK panel bar. In this case, all the connections originate in the original initial state (state 0). You can view all the connections for a different state by changing *State* to a value other than 0. You can also change the *Temperature* to see how the rates change. In this case the rates are all extremely small at 300 K, as the barriers are quite high. However, due to the exponential dependence of the rate on 1/T, doubling the temperature to 600 K increases the rates by about 16 orders of magnitude.



ID	Barrier (eV)	Prefactor (1/s)	Rate (1/s)	Energy (eV)	State
0	1.905	1.000e+13	1.009e-19	-11281.951	1
1	1.970	1.000e+13	7.897e-21	-11282.240	2
2	1.963	1.000e+13	1.051e-20	-11282.240	3
3	1.942	1.000e+13	2.332e-20	-11282.187	4
4	2.045	1.000e+13	4.448e-22	-11282.004	5

Validating the assumed prefactor

We will now verify that the assumed prefactor used in the AKMC calculations is reasonable. We select the transition between states 0 and 17, as it has the lowest barrier and involves atoms close to the B atom.



ID	Barrier (eV)	Prefactor (1/s)	Rate (1/s)	Energy (eV)	State
17	1.901	1.000e+13	1.163e-19	-11281.957	18
0	1.905	1.000e+13	1.009e-19	-11281.951	1
25	1.906	1.000e+13	9.408e-20	-11281.955	26
19	1.906	1.000e+13	9.405e-20	-11281.955	20
47	1.907	1.000e+13	9.173e-20	-11281.949	48

Tip: You can sort the states by each of the columns. In this case, we sorted by barrier to identify the states with the lowest barrier. They also have the highest rates as the prefactor is identical for all transitions.

The procedure for this calculation is as follows:

- Extract the initial, saddle and final states from the Markov Chain, and optimize their geometry.
- Create a NEB configuration with more images and optimize it, first without and then with climbing image.
- Do the actual prefactor calculation using *HTSTEvent*.

The first point is covered in the script: `extract-neb-from-markov-chain.py`. The Markov Chain object stores the final states and saddle point states for all the discovered transitions between the known states. These configurations can then be extracted as shown in the first lines of code below, for the initial and final states. Afterwards, we define a calculator identical to the one used in the AKMC simulation and optimize the geometry with a tolerance of 0.01 eV/Å. This is a stricter criterion than the default in **OptimizeGeometry**, but identical to the default in the AKMC simulation, which is selected to ensure convergence of the prefactor calculation. For some systems it might be needed to converge even more tightly.

```
markov_chain = nlread('akmc_markov_chain.nc')[0]

initial_state_id = 0
```

(continues on next page)

(continued from previous page)

```
final_state_id = 17

initial_conf = markov_chain.getStateConfiguration(initial_state_id)
final_conf = markov_chain.getStateConfiguration(final_state_id)
```

extract-neb-from-markov-chain.py

In the next script, `optimize-neb-from-markov-chain.py`, we create a NEB configuration based on the minimized initial and final configurations, plus the saddle configuration as extracted from the Markov Chain.

```

markov_chain = nlread('akmc_markov_chain.nc')[0]

initial_conf = nlread('initial_conf.nc',BulkConfiguration)[-1]
final_conf = nlread('final_conf.nc',BulkConfiguration)[-1]

initial_state_id = 0
final_state_id = 17

# Make the NEB configuration

saddle_conf = markov_chain.getSaddleConfiguration(initial_state_id, final_state_id)

configuration_list = [initial_conf, saddle_conf, final_conf]

neb_configuration = NudgedElasticBand(configuration_list, image_distance=1.0*Angstrom,␣
↪generate_images=True, wrap_pbc=True)

nlsave('neb-configuration.nc',neb_configuration)
```

Then we define a calculator identical to the one used in the AKMC simulation (not shown here), and proceed to optimize the NEB; this is done with a regular optimization first, and then an additional one with climbing image turned on. This optimization can take several hours, even on a multi-core machine.

```

neb_configuration.setCalculator(calculator)

optimized_neb_configuration = OptimizeNudgedElasticBand(
        neb_configuration,
        max_forces=0.01*eV/Ang)

nlsave('optimized-neb-configuration.nc',optimized_neb_configuration)

ci_optimized_neb_configuration = OptimizeNudgedElasticBand(
        optimized_neb_configuration,
        climbing_image=True,
        max_forces=0.01*eV/Ang)

nlsave('ci-optimized-neb-configuration.nc',ci_optimized_neb_configuration)
```

optimize-neb-from-markov-chain.py

Note: You can also do the NEB optimization with climbing image turned on from the beginning, but this is a less

stable approach and can lead to convergence problems. Doing it in two steps, as we do here, is slightly slower, but will be much more likely to converge without issues.

Finally, we calculate the prefactor with a simple script using HTSTEvent, reading in the fully optimized NEB configuration.

```
neb_configuration = nlread('ci-optimized-neb-configuration.nc',NudgedElasticBand)[-1]

# ------------------------------------------------------------
# HTST Event
# ------------------------------------------------------------
htst_event = HTSTEvent(
    neb_configuration,
    finite_difference_method=Central,
    minimum_displacement=0.05*Angstrom,
    )
nlsave('htst-event.nc', htst_event)
nlprint(htst_event)
```

calculate-prefactor.py

This script requires approximately 600 energy and force calculations and ran for about 2.5 hours on an 8-core node. When it has finished, the results are found at the end of the log-file containing the std output:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| HTST Event Information                                                        |
|                                                                               |
+-------------------------------------------------------------------------------+
|    Reactant Energy: -11283.6643027 eV                                         |
|    Saddle   Energy: -11281.7203394 eV                                         |
|    Product  Energy: -11282.1925809 eV                                         |
|   Forward  Barrier:      1.9439633 eV                                         |
|   Backward Barrier:      0.4722415 eV                                         |
| Forward  Prefactor:      1.792e+14 1/s                                        |
| Backward Prefactor:      6.939e+13 1/s                                        |
+-------------------------------------------------------------------------------+
```

We see that the forward prefactor is a little more than an order of magnitude larger than the estimated value of 10^{13} s^{-1}, while the backward prefactor is a little less than an order of magnitude off. This validates our assumed prefactor, as this deviation from the assumption is much smaller than the many orders of magnitude the rate can vary due to the exponential dependence on the barrier.

Conclusions

The AKMC simulations indicate that atomic boron in bulk silicon is very stable in a substitutional lattice site, and effectively immobile at temperatures close to room temperature. A reaction rate prefactor of 10^{13} s^{-1} per second was assumed and later verified.

3.11.18 Adaptive Kinetic Monte Carlo Simulation of Pt Island Ripening

Version: 2017

Downloads & Links

`initial.py` `morse_potential.py` `adding_akmc.py` `akmc.py`

This tutorial is on the use of the Adaptive Kinetic Monte Carlo (AKMC) method. It will cover setting up a simulation to model Pt island ripening on a Pt(111) surface, how to implement the Morse potential, and how to analyze the results. For more information on the AKMC method, you may look at the following tutorials: *AKMC Simulation of Pt on Pt(100)* and *Modeling Vacancy Diffusion in SiGe alloy*.

Introduction

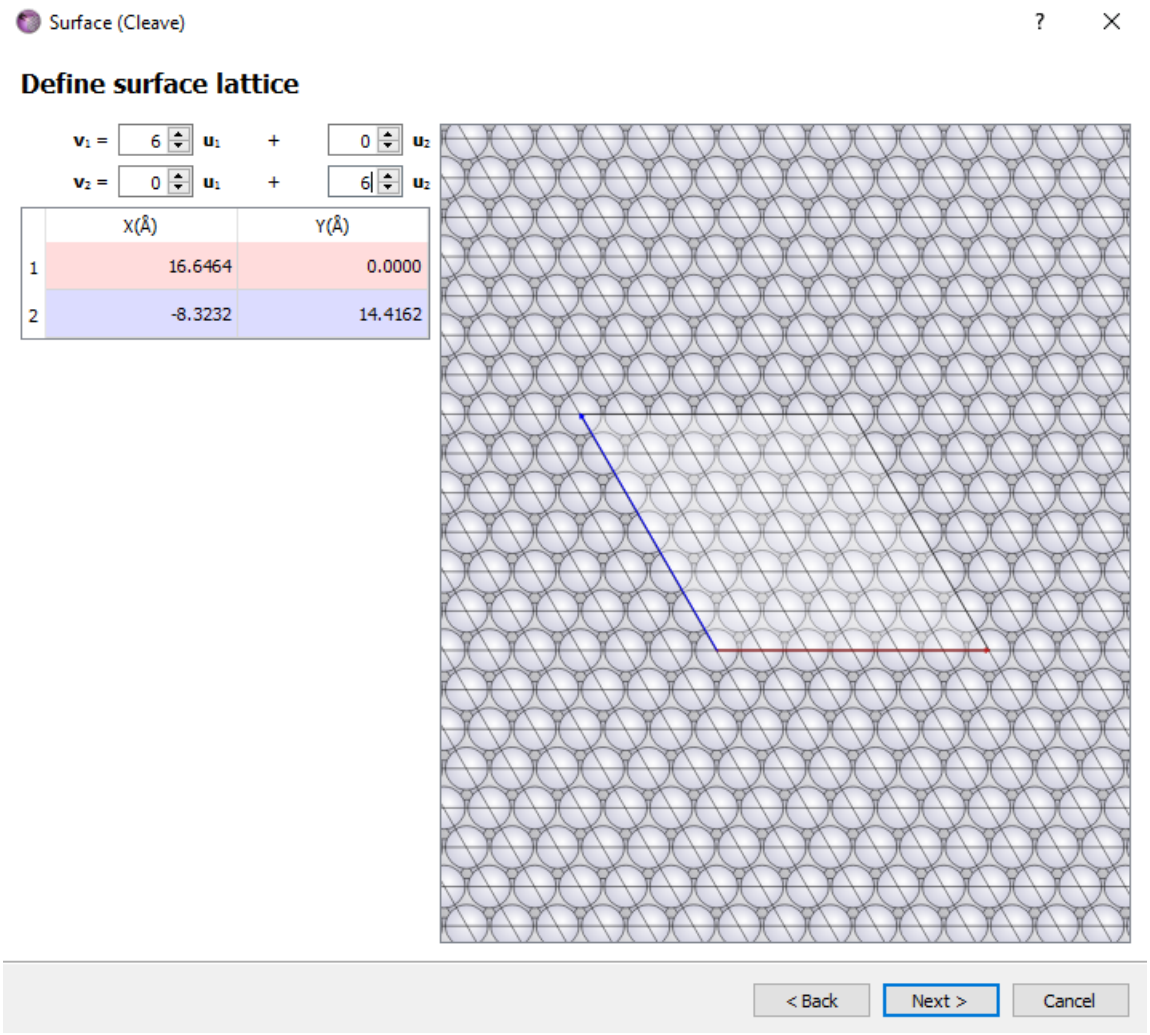
In this tutorial we will model island ripening of Pt adatoms on a Pt(111) surface at 300 K using the Adaptive Kinetic Monte Carlo (AKMC) method. The initial surface will have seven adatoms and the most stable structure is a compact heptamer island. The atomic interactions will be modeled using a simple Morse potential to allow us to compare our results to a published benchmark study[1]. The use of the Morse potential also allow us to showcase the use of AKMC without resorting to computationally expensive DFT calculations, but it is not expected to directly reproduce a particular experimental result.

Creating the initial configuration

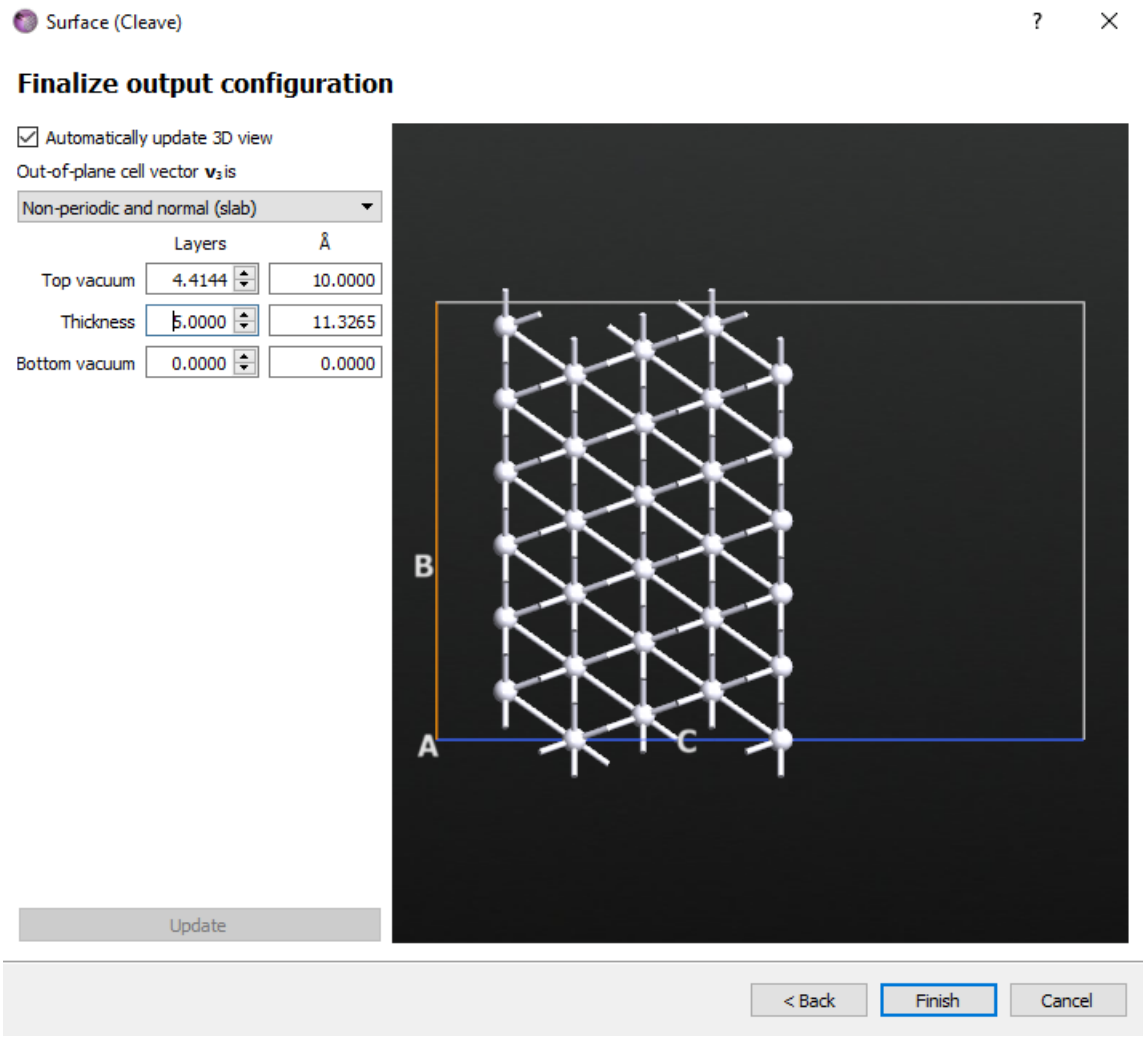
The initial configuration will be a 4 layer *p* (6x6)-Pt(111) slab with seven adatoms occupying three-fold hollow sites.

1. Open the **Builder**.
2. Import the Pt *fcc* bulk configuration by clicking on *Add* → *From Database* and search for Platinum.
3. The Pt primitive cell will appear in the **Stash**.
4. Click on *Builders* → *Surface (Cleave)*.
5. In the new window, choose the (111) Miller indices.
6. Define the *p* (6x6) surface unit cell.

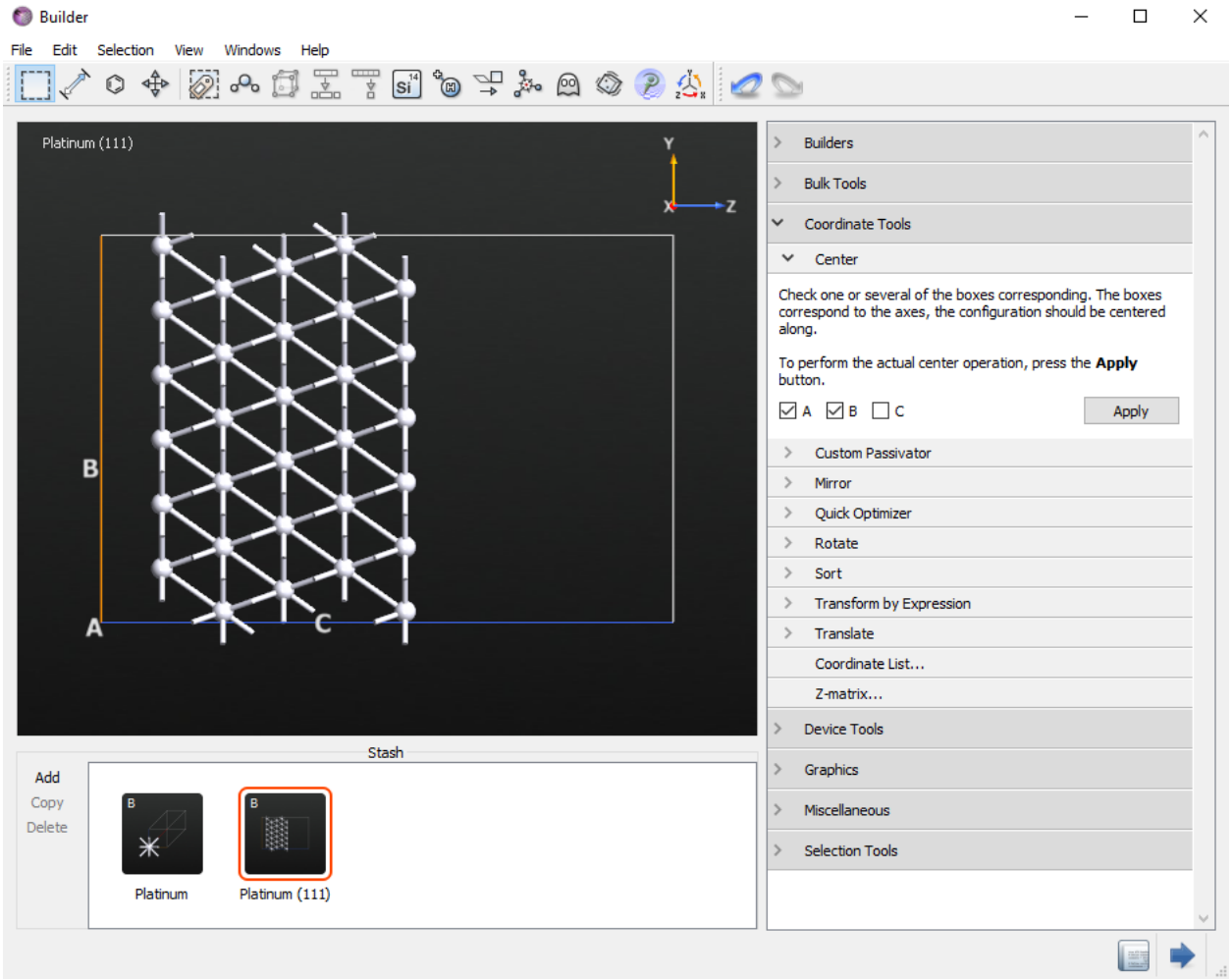
[1] Rye Terrell, Matthew Welborn, SamuelăT. Chill, and Graeme Henkelman. Database of atomistic reaction mechanisms with application to kinetic Monte Carlo. *J. Chem. Phys.*, 137:014105, 2012. doi:10.1063/1.4730746.



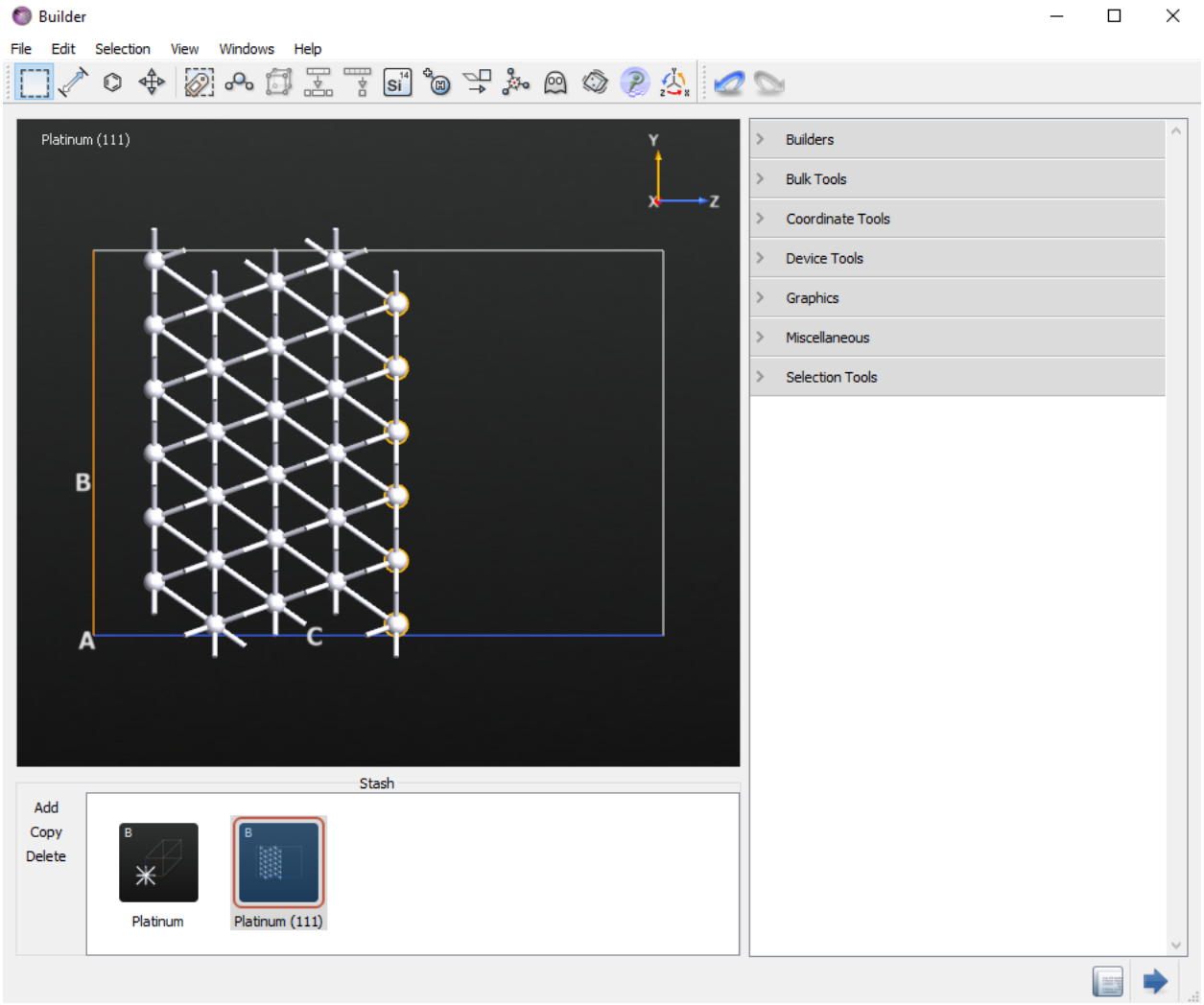
7.Choose the *Non-periodic and normal (slab)* option and a thickness of 5 layers. Click *Finish* button.



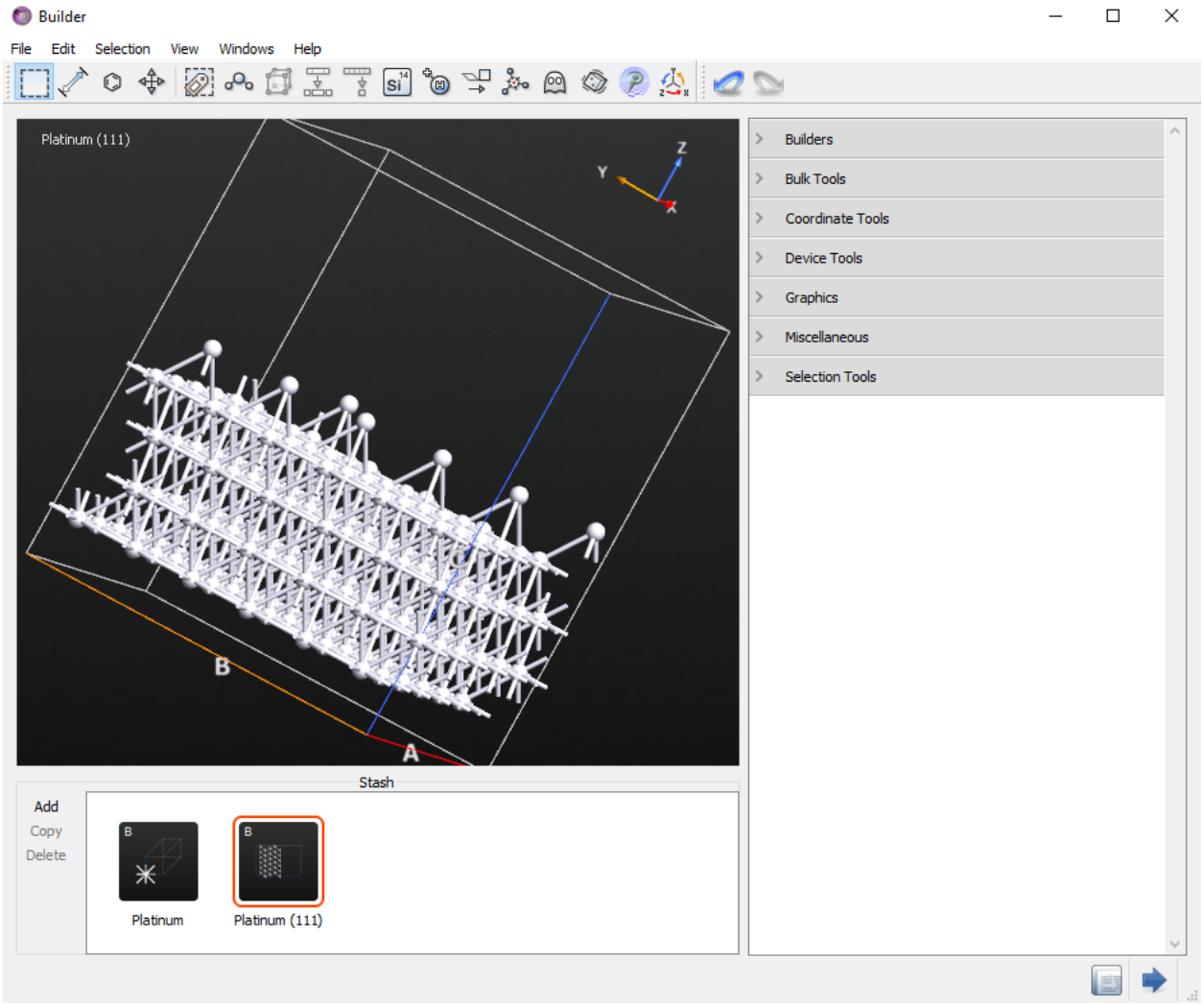
8.Click on *Coordinate Tools* → *Center* and select the *A* and *B* axes and click the *Apply* button.



9.Select the whole top layer.



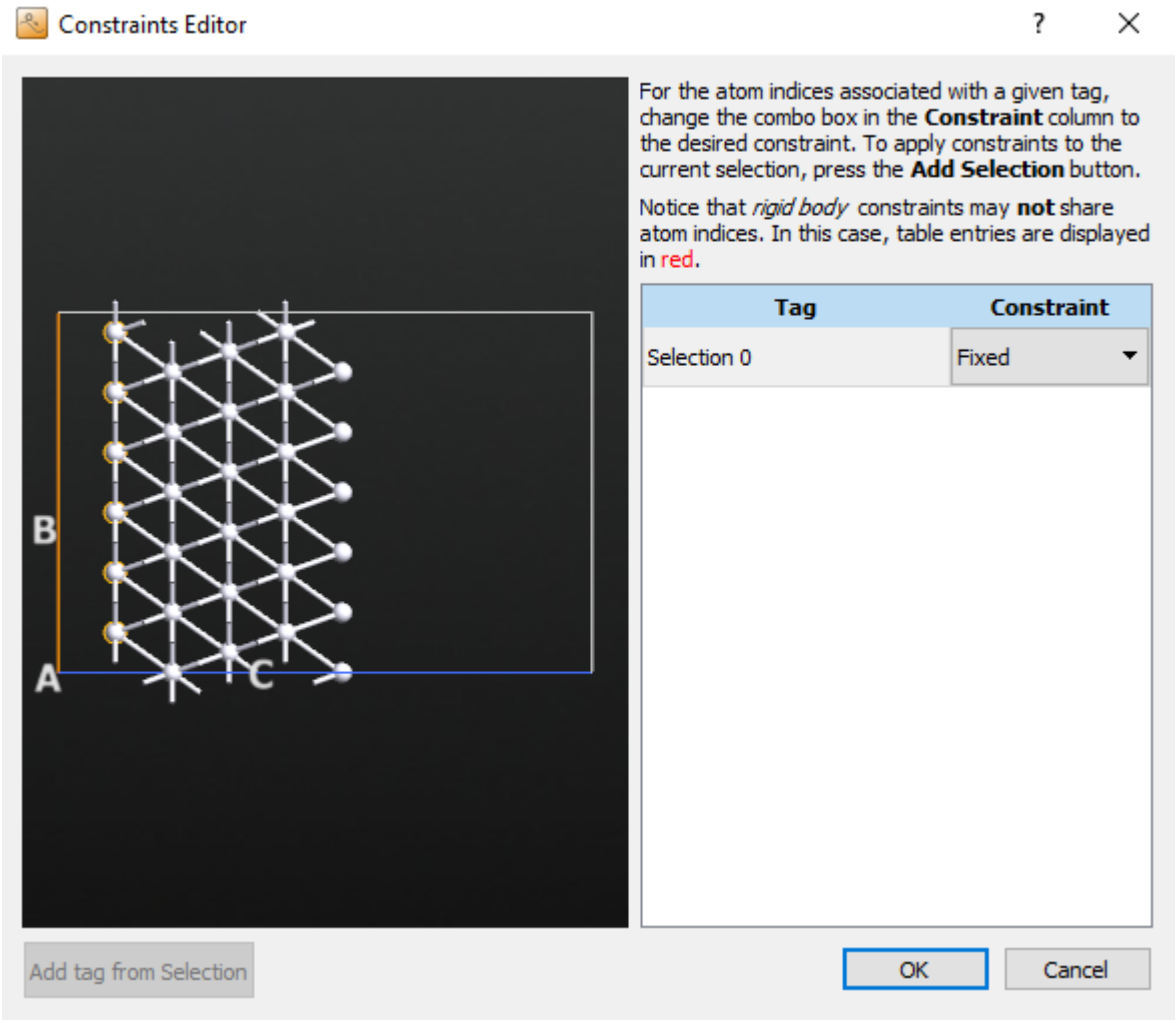
10. Un-select seven non-contiguous atoms holding the `shift` key on your keyboard and then press the `del` key to remove the selected atoms. There should now be seven adatoms. It should look similar to the image below.



11. Send the configuration to the **Script Generator** by clicking the or dragging the structure onto the **Scripter**.

For now, do not add a calculator block. Since we are using a *Morse potential*, we will specify the calculator later. The parameters for the Morse potential were originally fitted to model bulk Pt and are the same as used inPage 1619, 1.

1. Add an *Optimization* → *OptimizeGeometry* block and double-click it. Set the Force tolerance to 0.01 eV/Å. Then click *Add Constraints* and select the bottom row of atoms, click *Add tag from Selection*, and then set the Constraint to *Fixed*.



2.Change the Default output file to `initial.hdf5`.

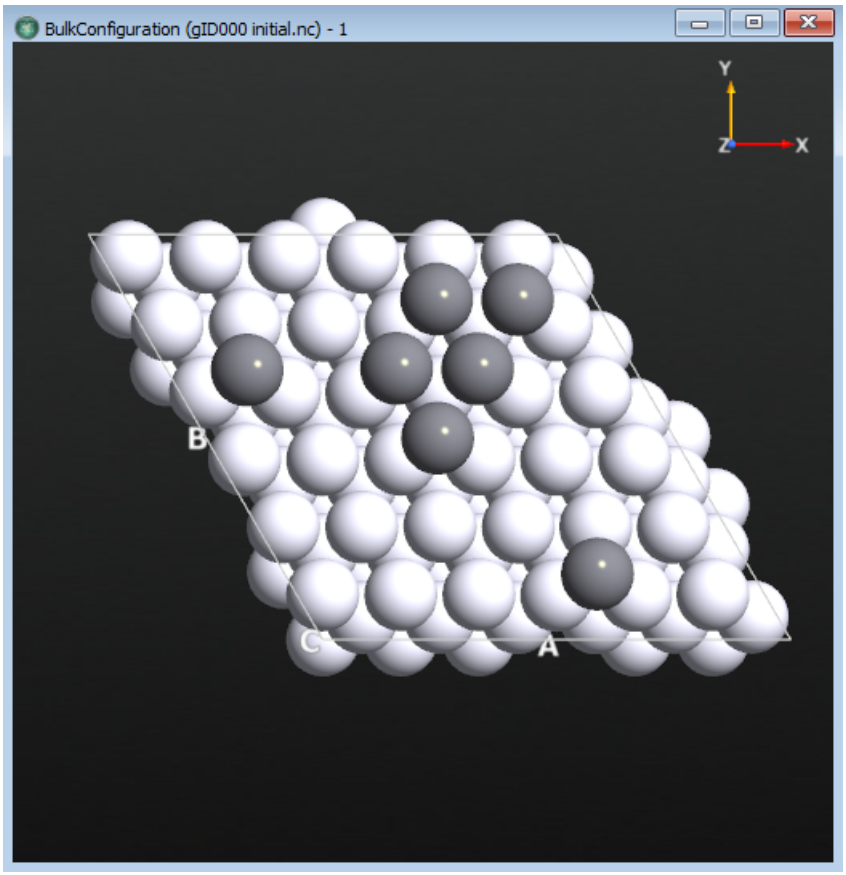
3.Send the script to the **Editor**.

4.We now need to define the Morse calculator. Insert the following code above the call to *OptimizeGeometry*: `morse_potential.py`

```
potentialSet = TremoloXPotentialSet(name='Pt-Morse')
potentialSet.addParticleType(ParticleType.fromElement(Platinum))
potentialSet.addPotential(MorsePotential(
    'Pt',
    'Pt',
    r_0=2.897*Angstrom,
    k=1.6047*Angstrom**-1,
    E_0=0.7102*eV,
    r_i=6.5*Angstrom,
    r_cut=9.5*Angstrom)
)
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)
```

5.Now **save** the script as `initial.py` or download the `initial.py` and send it to the **Job Manager** and run it.

The optimized geometry can now be selected on the **LabFloor** and visualized in the **Viewer**. It may be useful to click on **Properties** in the right-hand panel and tweak the way the atoms are rendered to more easily distinguish the adatoms from the bulk atoms. In the image below, the atoms are rendered as covalent radius spheres and the adatoms are in a darker gray than the bulk Pt.



Tip: If you wish to re-create the above figure, you can do the following: Select the file `initial.hdf5` file on the **LabFloor**. Click the **Viewer** on the right-hand panel. Open the **Properties** window. Select the *Covalent* option under **Atoms** and check *Wrap atoms* under **Bulk**. The atoms are now shown as balls with their covalent radius. Choose the top 7 Pt atoms and then change to another color (here we chose gray). Now you can distinguish the 7 adatoms on the Pt surface.

Setting up the AKMC Simulation

We will now set up and run the AKMC simulation. The goal is to determine the time required to form a compact heptamer island at 300 K.

Now that you have an optimized initial structure, you can run the AKMC simulation. Drag `initial.hdf5` from the main QuantumATK window to the **Scripter**. Starting from **ATK 2017**, **AdaptiveKineticMonteCarlo** is implemented in the Scripter.

Note: If you have an older version, you can download the script `adding_akmc.py` and append the contents in the

Editor instead of following the steps below. Instead, go to the next section: *Adding the calculator*

We use the Morse potential which is the same as used in[Page 1619, 1]. After setting up the AKMC simulation, we will add the Morse Potential calculator in the editor in the next section.

In the **Scripter**, add an *Optimization* → *AdaptiveKineticMonteCarlo* block.

1. Double click the **AdaptiveKineticMonteCarlo** block.
2. Go to the **Optimization** tab, and click on the *Atomic Constraint Editor* to add a constraint.
3. Select the bottom layer to be constrained, as in the initial optimization, and press the *Add tag from Selection* button. Change to the *Fixed* constraint. This will remove the red-colored warning.
4. In the **Kinetic Monte Carlo** tab, set the value to 20 in the *Superbasin creation threshold*.
5. For **HTST Prefactor**, we use a fixed Prefactor value of 10^{13} s^{-1} because it does not change much from one reaction to another.

Note: For most solid-state systems, the prefactor is not important to calculate directly (and it is expensive to do so). The primary difference between different reaction mechanisms is the size of the energy barrier. In the referenced simulation we are comparing against, this step was skipped.

6. In the *Saddle Search* settings, set the number of searches to 200.
7. Change the Default output file name to `akmc.hdf5` and go to the **Editor** by clicking the icon.

Adding the calculator

Now we need to add the *Morse potential* calculator above the AKMC calculation in the **Editor** using the script `morse_potential.py`.

```
# Morse-potential
potentialSet = TremoloXPotentialSet(name='Pt-Morse')
potentialSet.addParticleType(ParticleType.fromElement(Platinum))
potentialSet.addPotential(MorsePotential(
    'Pt',
    'Pt',
    r_0=2.897*Angstrom,
    k=1.6047*Angstrom**-1,
    E_0=0.7102*eV,
    r_i=6.5*Angstrom,
    r_cut=9.5*Angstrom)
)
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Adaptive Kinetic Monte Carlo
# ------------------------------------------------------------
htst_parameters = HTSTParameters(
```

(continues on next page)

(continued from previous page)

```
    assumed_prefactor=1e+13*1/Second,
    )

if os.path.isfile('akmc_markov_chain.hdf5'):
    markov_chain = nlread('akmc_markov_chain.hdf5')[-1]
else:
    markov_chain = MarkovChain(
        configuration=bulk_configuration,
        configuration_energy=TotalEnergy(bulk_configuration).evaluate(),
        )

if os.path.isfile('akmc_kmc.hdf5'):
    kmc = nlread('akmc_kmc.hdf5')[-1]
else:
    kmc = None

constraints = [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
                13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
                26, 27, 28, 29, 30, 31, 32, 33, 34, 35]

akmc = AdaptiveKineticMonteCarlo(
    markov_chain=markov_chain,
    kmc_temperature=300.0*Kelvin,
    md_temperature=2000.0*Kelvin,
    calculator=bulk_configuration.calculator(),
    kmc=kmc,
    superbasin_threshold=20,
    htst_parameters=htst_parameters,
    constraints=constraints,
    confidence=0.99,
    filename_prefix='akmc',
    )

akmc.run(max_searches=200, max_kmc_steps=10000)
```

You can also download the input file `akmc.py` to start the run for the AKMC simulation.

Running the Simulation

This script will run 200 saddle searches each time it is called. The first time the script is run, it creates a new AKMC simulation, while subsequent runs resume the simulation. Each saddle search takes about 15 seconds. So, in serial, 200 saddle searches will take under 1 hour to complete. The calculation can also be run in parallel using MPI, but note that you need at least 3 processes to get a speed-up. The goal of this simulation is to find the time it takes to form a compact heptamer (7 atom) island. This is the lowest energy configuration of the system. The AKMC script will have to be run several times to reach this point. After each run, you should follow the analysis below to monitor the progress of the simulation.

Note: You will probably need to run more searches to find a heptamer island with the AKMC simulation. If you do not find it within 200 searches, you can restart the simulation to search for more saddle points by using the files: `akmc_markov_chain.hdf5` and `akmc_kmc.hdf5`, which the script is set up to do automatically. Here we ran it again up to 1000 searches.

Analyzing the AKMC Simulation

Since AKMC simulations are stochastic, your results will be slightly different from what is shown here. You can download the current results for the same analysis. `akmc_log.hdf5`, `akmc_markov_chain.hdf5` and `akmc_kmc.hdf5`.

There are three output files to examine: `akmc_log.hdf5`, `akmc_markov_chain.hdf5` and `akmc_kmc.hdf5`. The `akmc_log.hdf5` file contains information about the results of each saddle search. If there are errors they will be reported here. The file `akmc_markov_chain.hdf5` contains the energies and geometries of the different configurations and how the different configurations are connected through saddle points and their energy barriers. It also contains the data needed to compute rate constants using harmonic transition state theory. The file `akmc_kmc.hdf5` stores the Kinetic Monte Carlo simulation data. This includes the current state identifier and the simulation time.

AKMC Log File

The log file should be looked at first to make sure that the simulation ran correctly. Below is an example of what the log file will look like. The log file can be viewed by clicking on the `akmc_log.hdf5` file on the **LabFloor** and clicking the **Text Representation** button in the panel on the right.

The first column shows the state *id* number that is used to identify each configuration in the Markov Chain. The second column shows the search number, which is a unique identifier for each saddle search. The third column shows the confidence estimate, which is a measure of the probability that all relevant saddle points have been found in that state. Saddle searches are run in each until the confidence exceeds the chosen threshold passed to the AKMC simulation. In the log file below, you can see that the initial state 0 is visited until the confidence meets 0.99, then the next state 2 is reached.

The fifth column contains a message about the saddle search. The message *found new state* represents a new reaction mechanism being discovered, while the message *Found saddle connecting to state* i *again* represents state *i* having been found again. In the state *id* 0, there are 7 *Found new state* and several *Found saddle connecting to state* i *again*. There is only one search that ended in an error in this example: search number 17. In this search a new saddle point was found, but it does not connect back to the current state. This means it is a saddle point that describes a reaction between two different states. This is expected to happen sometimes, but if a large number of the saddle searches have this message, then it probably means that the MD temperature is too high.

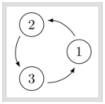
```
# Item:  0
# File:  C:\tutorial\akmc_log.nc
# Title: akmc_log.nc - gID000
# Type:  AKMCLog
state id search number confidence  message
       0             0   0.000000  Found new state
       0             1   0.000000  Found new state
       0             2   0.000000  Found new state
       0             3   0.000000  Found new state
       0             4   0.655097  Found saddle connecting to state 2 again
       0             5   0.801496  Found saddle connecting to state 4 again
       0             6   0.866310  Found saddle connecting to state 2 again
       0             7   0.927713  Found saddle connecting to state 1 again
       0             8   0.866170  Found new state
       0             9   0.507710  Found new state
       0            10   0.520759  Found saddle connecting to state 2 again
       0            11   0.878596  Found saddle connecting to state 6 again
       0            12   0.914000  Found saddle connecting to state 6 again
       0            13   0.918800  Found saddle connecting to state 2 again
       0            14   0.931824  Found saddle connecting to state 6 again
```

(continues on next page)

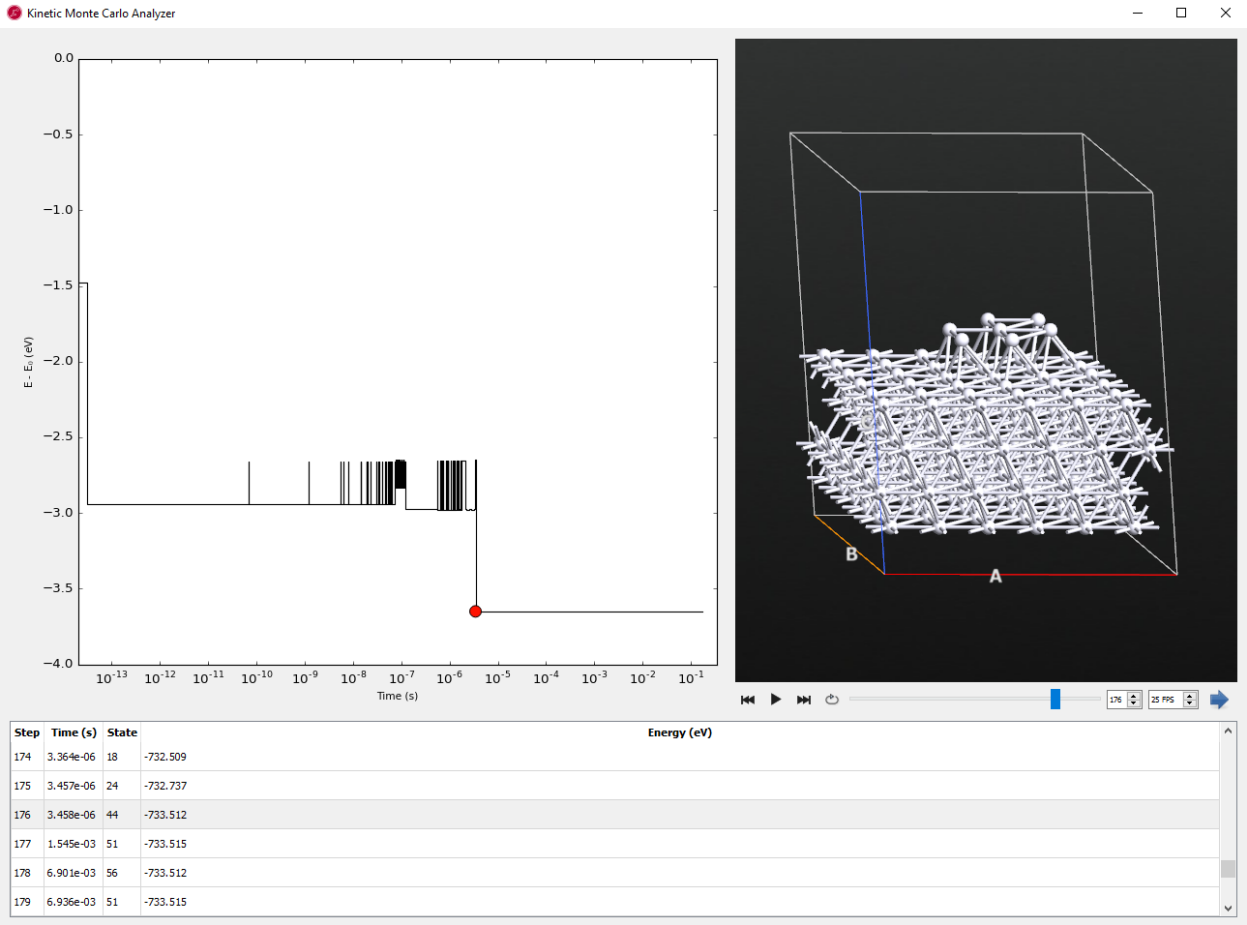
(continued from previous page)

```
      0             15   0.933590  Found saddle connecting to state 2 again
      0             16   0.854401  Found new state
      0             17   0.854401  No barrier was found between the end points of the␣
↪NEB calculation. Check to see if there are enough images and that the NEB is converged.
      0             18   0.858786  Found saddle connecting to state 6 again
      0             19   0.932129  Found saddle connecting to state 7 again
      0             20   0.962899  Found saddle connecting to state 5 again
      0             21   0.963494  Found saddle connecting to state 2 again
      0             22   0.966538  Found saddle connecting to state 5 again
      0             23   0.966757  Found saddle connecting to state 2 again
      0             24   0.966837  Found saddle connecting to state 2 again
      0             25   0.967957  Found saddle connecting to state 5 again
      0             26   0.968369  Found saddle connecting to state 5 again
      0             27   0.968399  Found saddle connecting to state 2 again
      0             28   0.968550  Found saddle connecting to state 5 again
      0             29   0.975805  Found saddle connecting to state 4 again
      0             30   0.978474  Found saddle connecting to state 4 again
      0             31   0.978529  Found saddle connecting to state 5 again
      0             32   0.978540  Found saddle connecting to state 2 again
      0             33   0.978544  Found saddle connecting to state 2 again
      0             34   0.985801  Found saddle connecting to state 7 again
      0             35   0.985802  Found saddle connecting to state 2 again
      0             36   0.985803  Found saddle connecting to state 2 again
      0             37   0.986687  Found saddle connecting to state 3 again
      0             38   0.986708  Found saddle connecting to state 5 again
      0             39   0.989750  Found saddle connecting to state 1 again
      0             40   0.991364  Found saddle connecting to state 6 again
      2             41   0.000000  Found new state
      2             42   0.000000  Found new state
```

Kinetic Monte Carlo Analyzer

The main way to view the results of the AKMC simulation is with the **Kinetic Monte Carlo Analyzer**. To use this tool, select both the KMC object in the `akmc_kmc.hdf5` and the MarkovChain object  in the `akmc_markov_chain.hdf5` files (if you hold the `ctrl` key, you can select on multiple files in the LabFloor) and then click on **Kinetic Monte Carlo Analyzer** in the panel on the right.

The Kinetic Monte Carlo Analyzer shows a plot of energy vs. time during the AKMC simulation on the left and the geometry of the currently selected configuration/time step on the right.



Tip: You can click on the image to see a version with higher resolution.

The above image shows the results after 1000 saddle searches, which results in the simulation of about 0.17 s of dynamics. The compact island was formed at KMC step 176, which corresponded to a simulation time of 3.4 ms. After the compact island is formed, the entire island diffuses across the surface using a concerted sliding mechanism. When you click the play button, you see the dynamics along time.

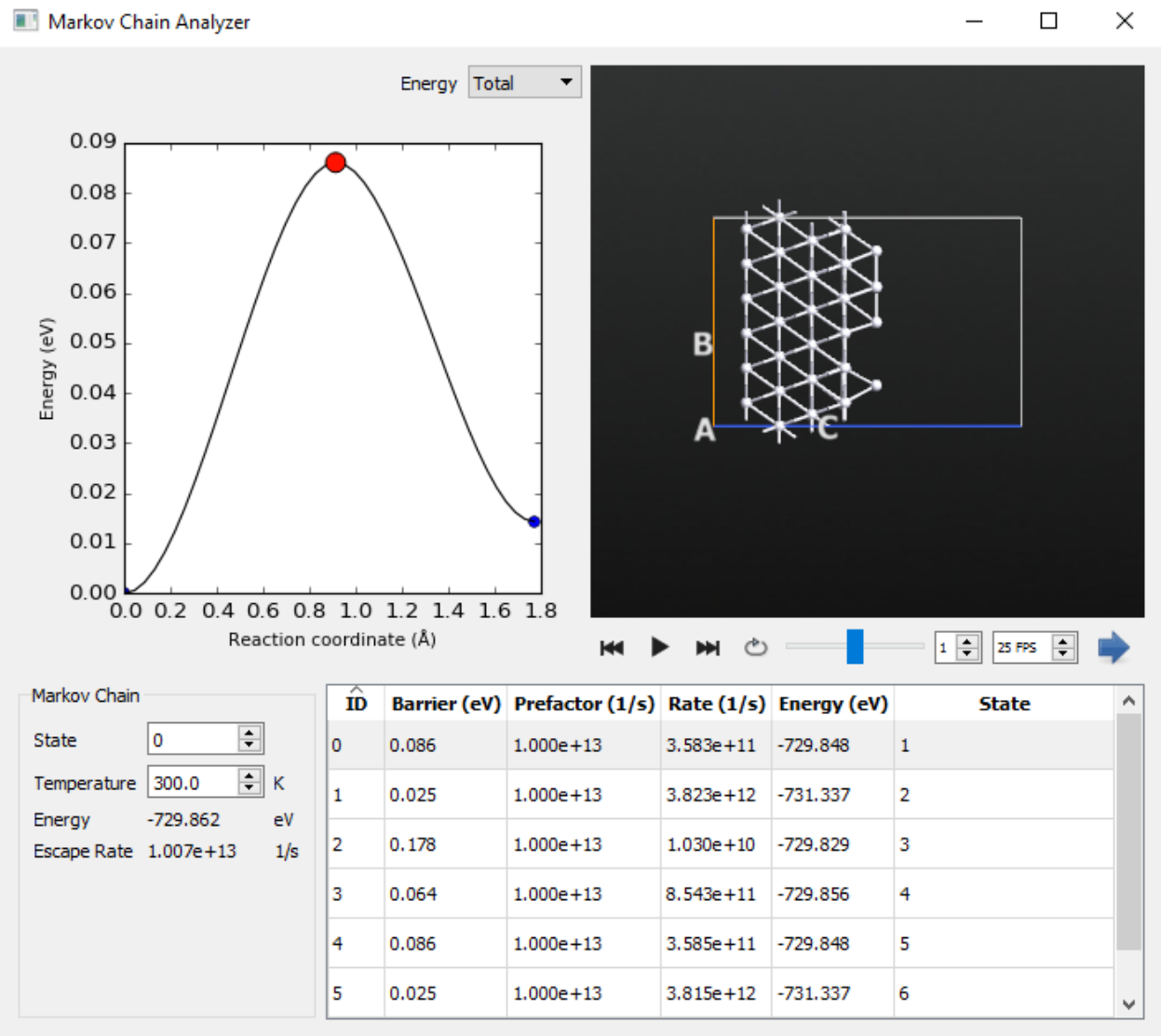
Markov Chain Analyzer

The Markov Chain Analyzer allows you to look at which reactions are present at each step of the KMC simulation. It can be invoked in two ways:

1)From the **Kinetic Monte Carlo Analyzer** by double clicking on a entry in the table.

2)By selecting the object in the `akmc_markov_chain.hdf5` on the **LabFloor** and clicking the **Markov Chain Analyzer** button in the panel.

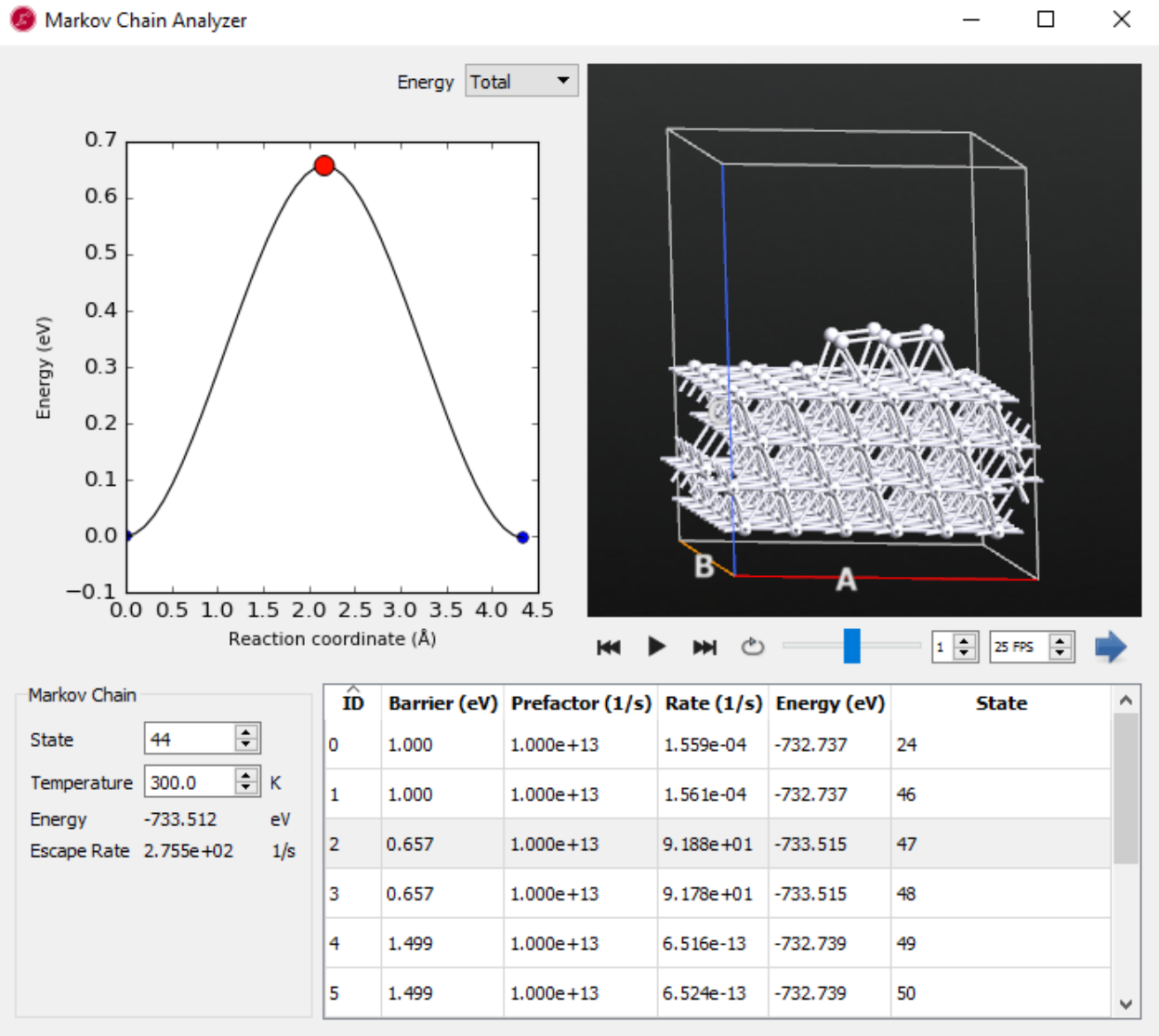
If we go to the first line in the table in the Kinetic Monte Carlo Analyzer and double click the line, it will open the Markov Chain Analyzer for the initial state.



ID	Barrier (eV)	Prefactor (1/s)	Rate (1/s)	Energy (eV)	State
0	0.086	1.000e+13	3.583e+11	-729.848	1
1	0.025	1.000e+13	3.823e+12	-731.337	2
2	0.178	1.000e+13	1.030e+10	-729.829	3
3	0.064	1.000e+13	8.543e+11	-729.856	4
4	0.086	1.000e+13	3.585e+11	-729.848	5
5	0.025	1.000e+13	3.815e+12	-731.337	6

Each of these reaction mechanisms has a very small barrier, indicating that the initial configuration is very unstable. By selecting different mechanisms from the table, each can be visualized. Now we will examine what reaction mechanisms are present for the compact island. Double-click on the line with the 176th step, which is where we found the island formation in the Kinetic Monte Carlo Analyzer. This will take us to state 44, which is the first state containing the compact island.

Note: Remember that the step number is most likely different in your simulation because of its randomly selected initial structure and dynamics.



The compact island shows several reaction mechanisms. Each corresponds to the entire 7 atom island diffusion in concert. The barriers are much larger (> 0.65 eV) than those from the initial state and explain why once the compact island is formed a large amount of time passed during the KMC simulation.

The increase in reaction barriers from the initial state with disconnected adatoms to the final compact island shows that the diffusion coefficient of the adatoms decreases dramatically over the course of the simulation.

Conclusion

In this tutorial, we looked at how to model surface diffusion of Pt adatoms using a simple Morse interaction potential. The initial configuration was 7 lone Pt adatoms and the final configuration was a compact 7 atom island. The AKMC simulation modeled longer than 0.1 s of dynamics and the formation time of the compact island took 3.4 ms at 300 K. If a molecular dynamics simulation with a 1 fs timestep had been performed instead, it would have taken 3.4 billion steps to reach the compact island formation. Thus the use of AKMC allowed us to model reaction dynamics on a timescale that is virtually unaccessible using molecular dynamics.

References

3.11.19 Adaptive Kinetic Monte Carlo Simulation of Pt on Pt(100)

Version: 2017

Downloads & Links

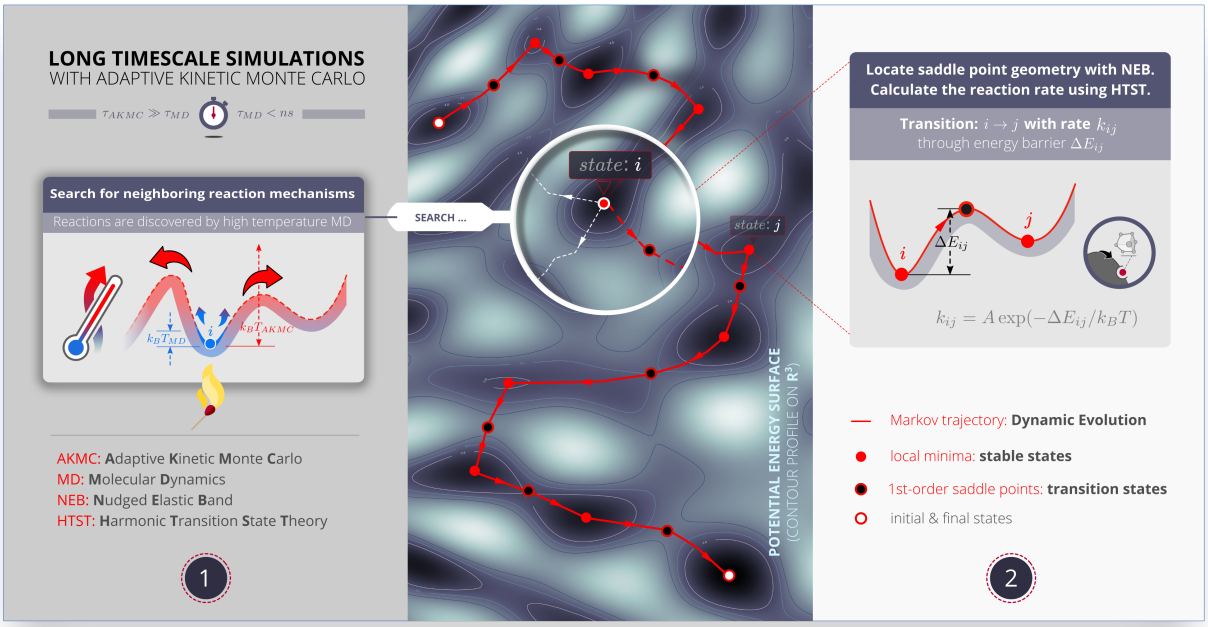
AKMC_ATK2017.py AKMC_ATK2016.py AKMC_ATK2015

In this tutorial you will learn how to run an Adaptive Kinetic Monte Carlo (AKMC) simulation to investigate the diffusion of a Pt adatom on a Pt(100) surface.

Introduction

How an atom diffuses on the surface is important to understand material properties and processes such as adsorption phenomena, thin-film growth, and sintering. Especially defect sites with vacancies or adsorbates on the surface are critical to determining the reaction pathway and rate. In general, an atom has two possible diffusion paths on a surface; by the exchange mechanism or the hopping mechanism. In this tutorial, we examine the diffusion path of a Pt adatom on the Pt(100) surface.

However, there is significant difference between the timescale of these rare events and the timescale of nano- or microseconds which is achievable with direct classical dynamics simulations. There are several approaches to bridging this time scale gap. One of these methods is *Adaptive Kinetic Monte Carlo (AKMC)* which can be used to speed up the saddle point searches along the potential energy surface quite effectively. In the next section, we will introduce the details of the AKMC method.



The AKMC method

In solid state systems, atoms reside near their equilibrium positions and reactions are rare events. Because the reactions occur on a much longer timescale than the typical vibrational period of the system (~100 fs), it is not possible to use molecular dynamics (MD) simulations to model solid-state reactions. MD simulations can only routinely access the nanosecond timescale while room temperature solid-state reactions of interest may occur on the millisecond timescale or longer.

Adaptive Kinetic Monte Carlo (AKMC) is a powerful tool for simulating the dynamics of solid-state systems that works by identifying the bottleneck regions between the stable minima and then modeling the reactions between them using statistical mechanics. More specifically, the bottleneck regions are the saddle points on the potential energy surface. These points lie on the minimum energy pathway between two minima, which is the statistically most likely path for a reaction to occur along. The primary computational expense in an AKMC simulation is in locating these saddle points.

In the AKMC implementation in QuantumATK, the saddle points are located by running high temperature MD trajectories. These trajectories should be hot enough that a few picoseconds of dynamics will be sufficiently long for a reaction to occur. The saddle points are located by periodically minimizing the MD trajectory to detect when the trajectories minimizes to a new geometry. Once the new minimum is located, the saddle point is found by running a nudged elastic band (NEB) calculation.

Once the saddle points are identified, the reaction rates can be estimated using harmonic transition state theory (HTST). In HTST, the reaction rate is given by:

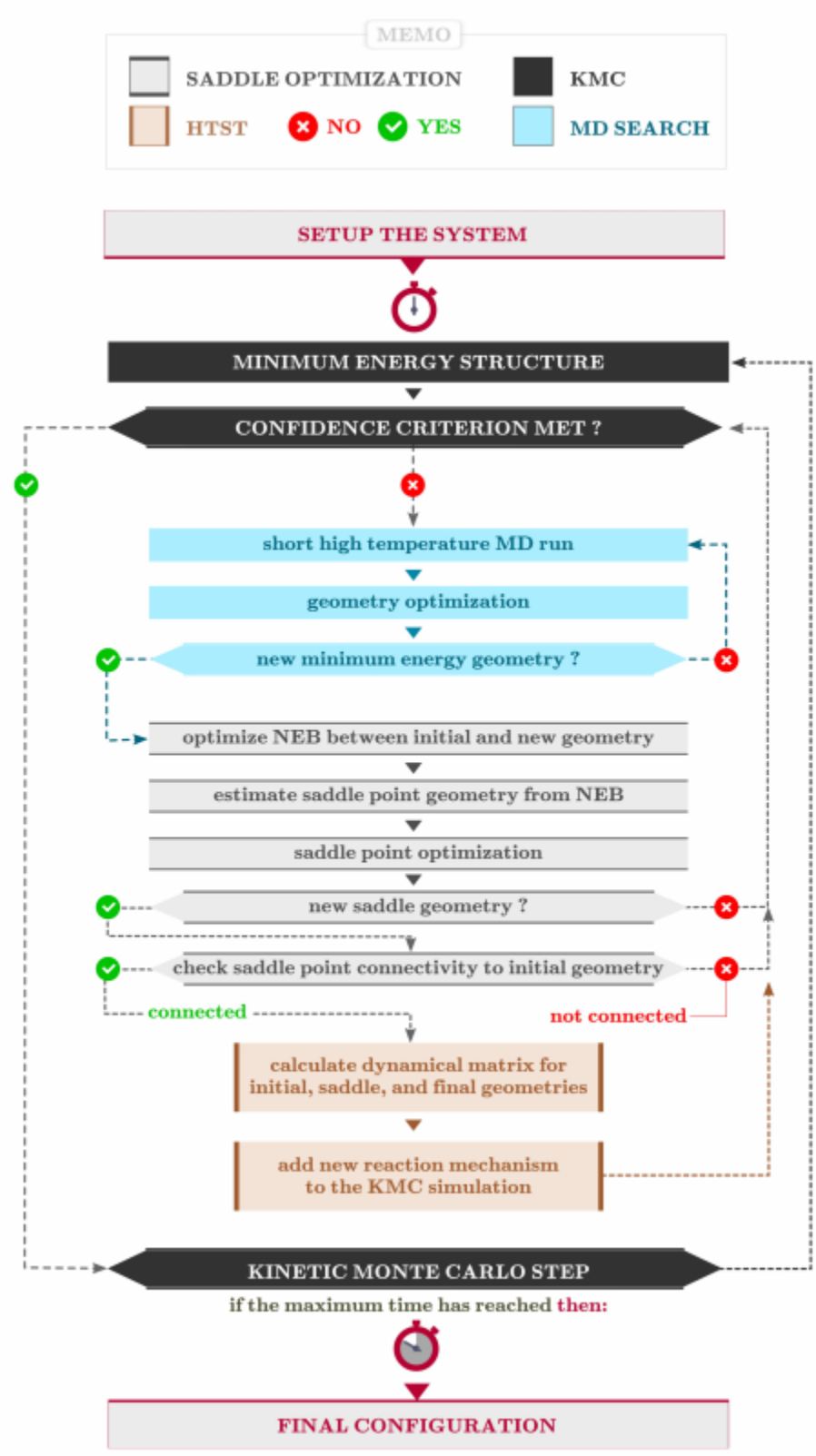
$$k = A \exp\left(- \frac{E}{k_B T} \right)$$

where k is the reaction rate, A is the vibrational prefactor, E is the difference in energy between the saddle point and the minimum (energy barrier), k_B is Boltzmanns constant, and T is the temperature.

The HTST rates can then be used to run a kinetic monte carlo (KMC) simulation. In KMC, the system is represented using a Markov chain. Each state of the Markov chain represents a stable state of the system (a local energy minimum) and the transition probabilities between the states are related to the reaction rates (determined from the saddle points). The state of the system is evolved from minimum to minimum. The next state is chosen with a probability proportional to its rate. The amount of time taken to transition to the next state is drawn from an exponential distribution with a rate parameter equal to the total escape rate from the state. When the dynamics are modeled in this way, the computational cost to model one step of the dynamics is very small. Tens of thousands of steps can be performed per second on a typical computer.

For the AKMC simulation to be accurate, it is crucial that the fastest reaction mechanisms are discovered as they represent the most likely events to be chosen in the KMC simulation. To ensure the accuracy of the simulation, each state in the Markov chain has an associated confidence. This confidence is an estimate of the probability that for each KMC step taken from that state, the correct event was chosen. Saddle searches are executed in each state until a user defined confidence level is reached. Once the confidence level is reached, the KMC simulation will proceed until a state is reached where the confidence is less than the target level.

Below is a basic flowchart for the AKMC algorithm. The *adaptive* part finds saddle points in inner loops. The *KMC* part updates the total time and the current configuration.

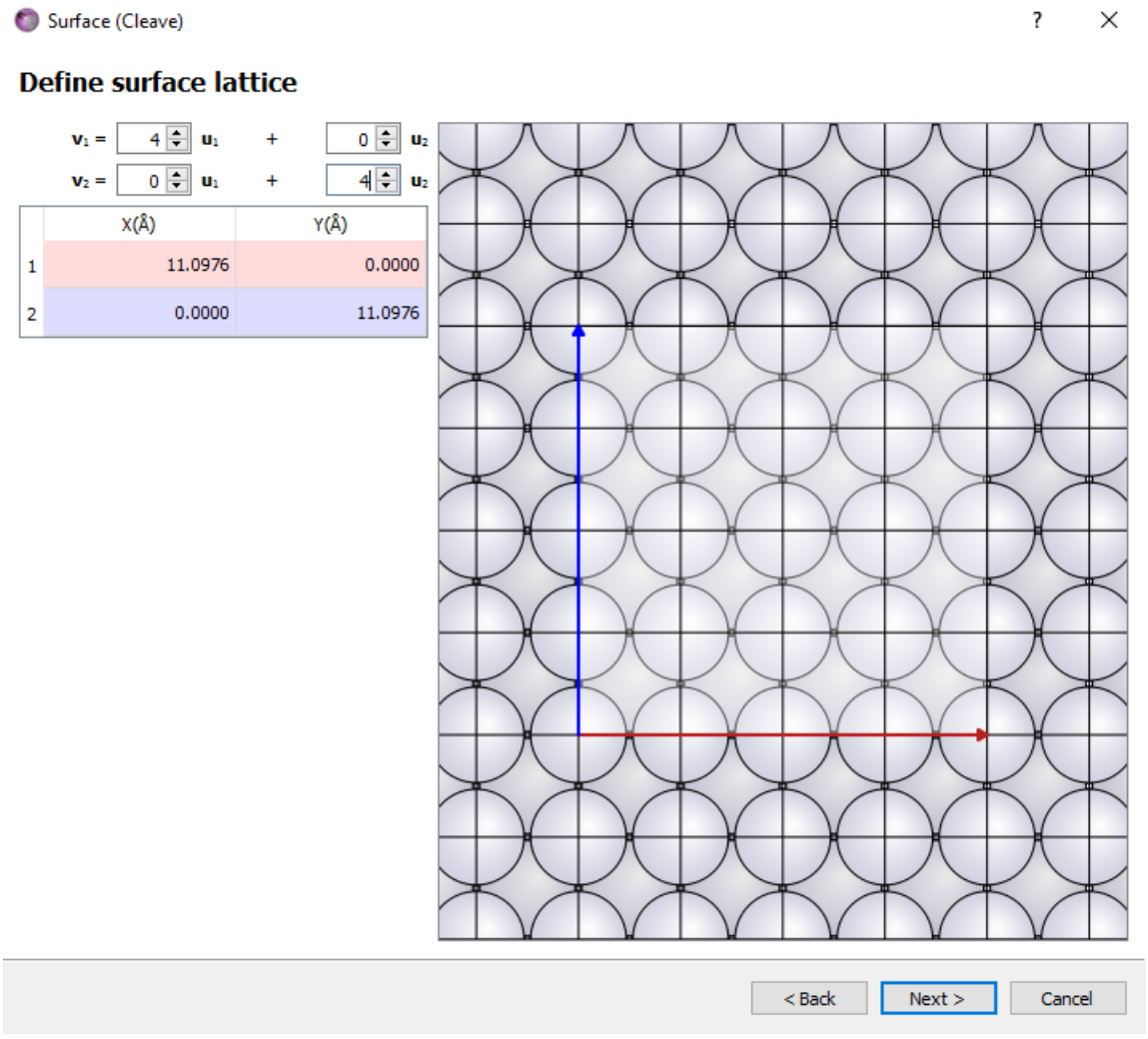

MEMO
SADDLE OPTIMIZATION
KMC
HTST
NO
YES
MD SEARCH
SETUP THE SYSTEM
MINIMUM ENERGY STRUCTURE
CONFIDENCE CRITERION MET ?
short high temperature MD run
geometry optimization
new minimum energy geometry ?
optimize NEB between initial and new geometry
estimate saddle point geometry from NEB
saddle point optimization
new saddle geometry ?
check saddle point connectivity to initial geometry
connected
not connected
calculate dynamical matrix for initial, saddle, and final geometries
add new reaction mechanism to the KMC simulation
KINETIC MONTE CARLO STEP
if the maximum time has reached then:
FINAL CONFIGURATION

For more details on AKMC and HTST see f.ex.[1] and[2]. For an introduction to KMC see[3].

Creating the initial configuration

The initial configuration will be a 4 layer *p* (4x4)-Pt(100) slab with a single adatom occupying a four-fold hollow site.

1. As the first step, import the Pt *fcc* bulk configuration by clicking on *Add* → *From Database* and search for *Platinum*. The Pt primitive cell will appear in the Stash.
2. Click on *Builder* → *Surface (Cleave)*
3. In the new window choose the (100) Miller indices.
4. Define the *p* (4x4) surface unit cell.

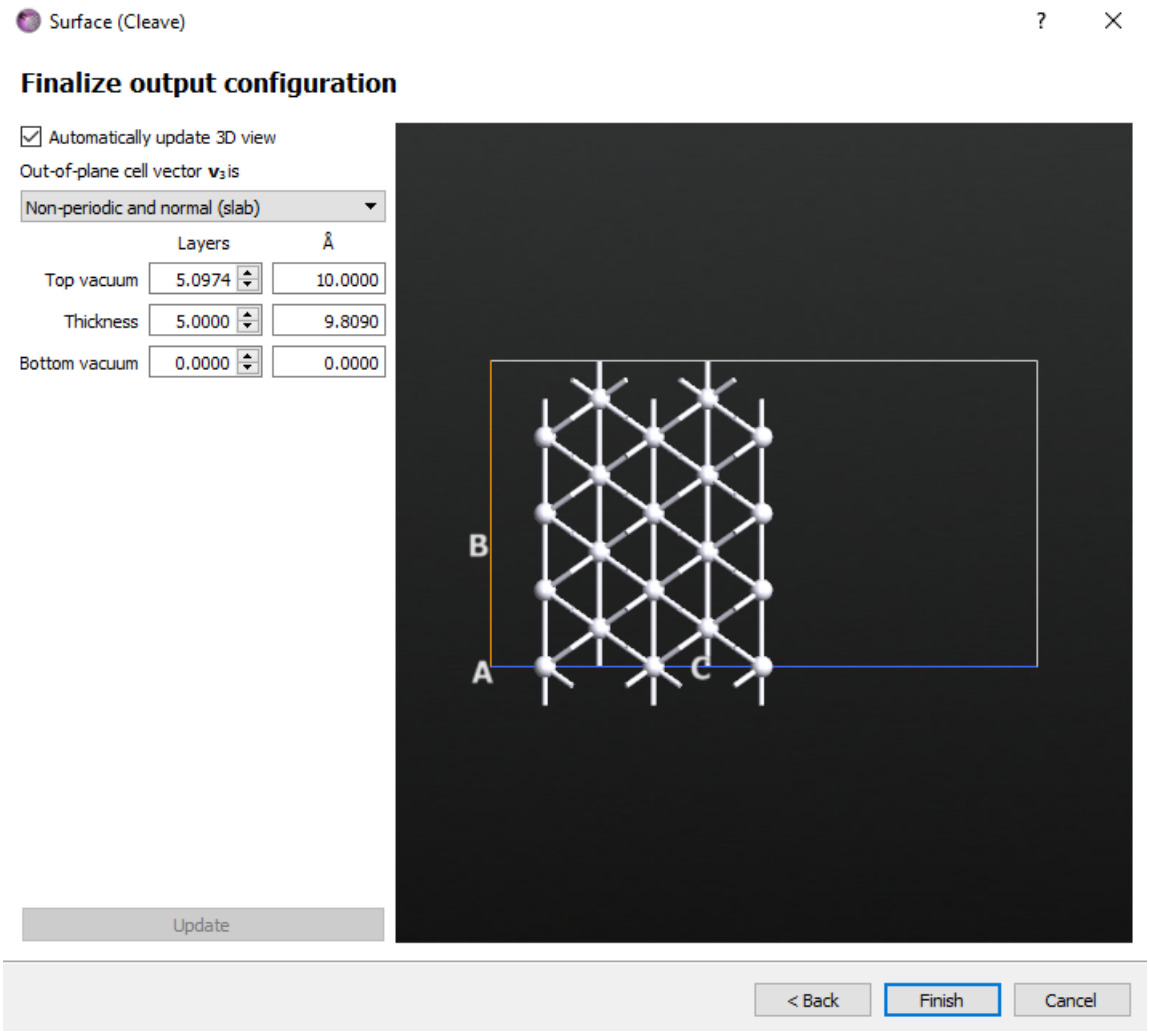


5. Choose the *non-periodic and normal (slab)* configuration with five layers, 10 Å of top vacuum. Click Finish.

[1] SamuelăT. Chill and Graeme Henkelman. Molecular dynamics saddle search adaptive kinetic Monte Carlo. *The Journal of Chemical Physics*, 140:214110, 2014. doi:10.1063/1.4880721.

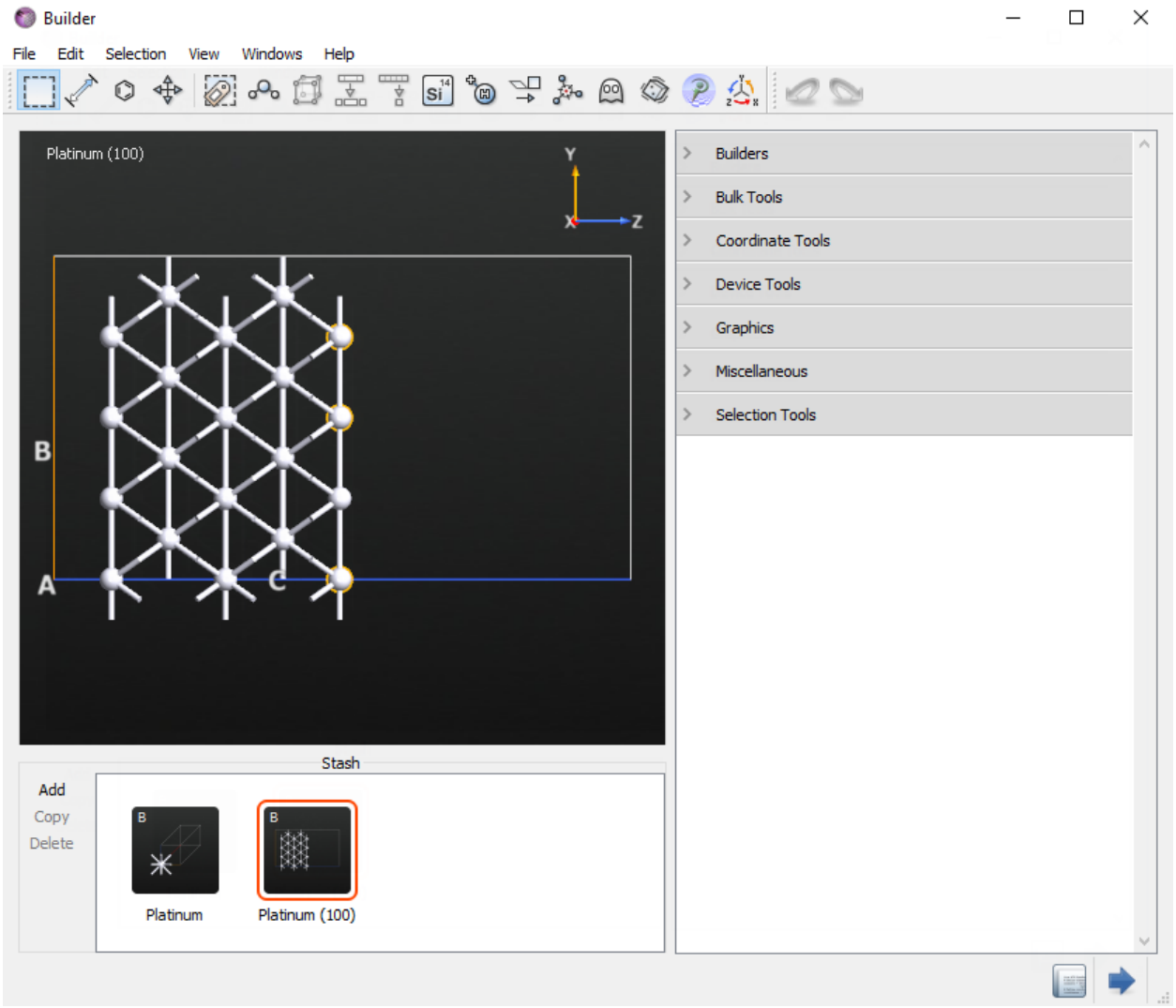
[2] L.ăXu and G.ăHenkelman. Adaptive kinetic Monte Carlo for first-principles accelerated dynamics. *Journal of Chemical Physics*, 129(11):114104, 2008. doi:10.1063/1.2976010.

[3] ArthurăF. Voter. *Introduction to the Kinetic Monte Carlo Method*, pages 1–23. Springer Netherlands, Dordrecht, 2007. URL: http://dx.doi.org/10.1007/978-1-4020-5295-8_1, doi:10.1007/978-1-4020-5295-8_1.

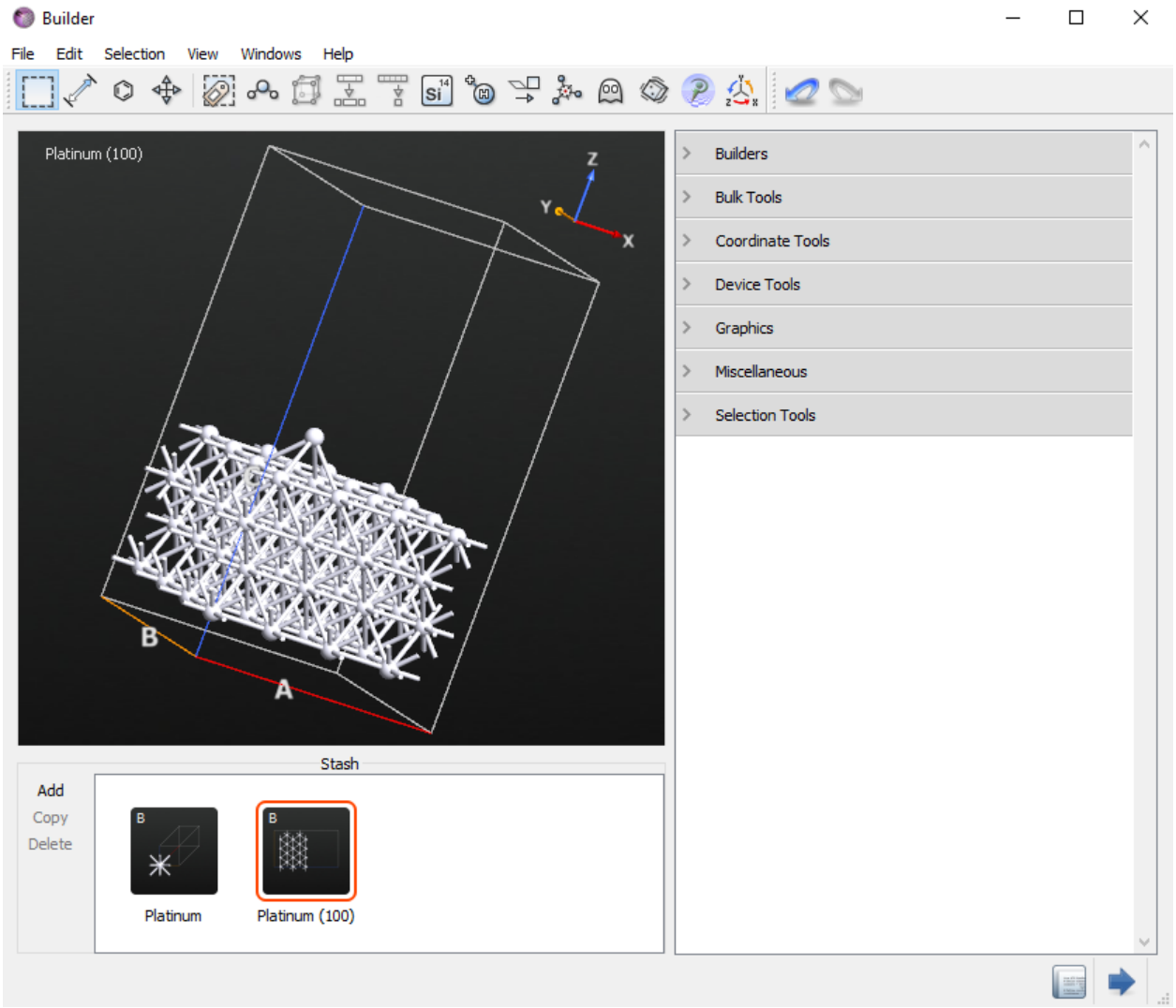


Your Pt(100) slab is now present in the **Stash**. The next step is to add the Pt adatom in a four-fold hollow position. Instead of adding a new atom on top of the slab, we will instead delete all but one atom from the topmost layer.

1. Select three of the four top rows of atoms as shown below and then press the delete key.



2.Now rotate the configuration so that the remaining row of four surface atoms is visible. Select three of the four atoms and then press the delete key. The final surface should have just one adatom in a four-fold hollow site.



Creating the AKMC script

Now that we have the initial configuration, we can start the AKMC simulation.

1. Send the initial configuration to the **Script Generator** by clicking the button.

2. Add **New Calculator** and *Optimization* → *OptimizeGeometry* blocks.

3. Change the output name to `akmc.nc`.

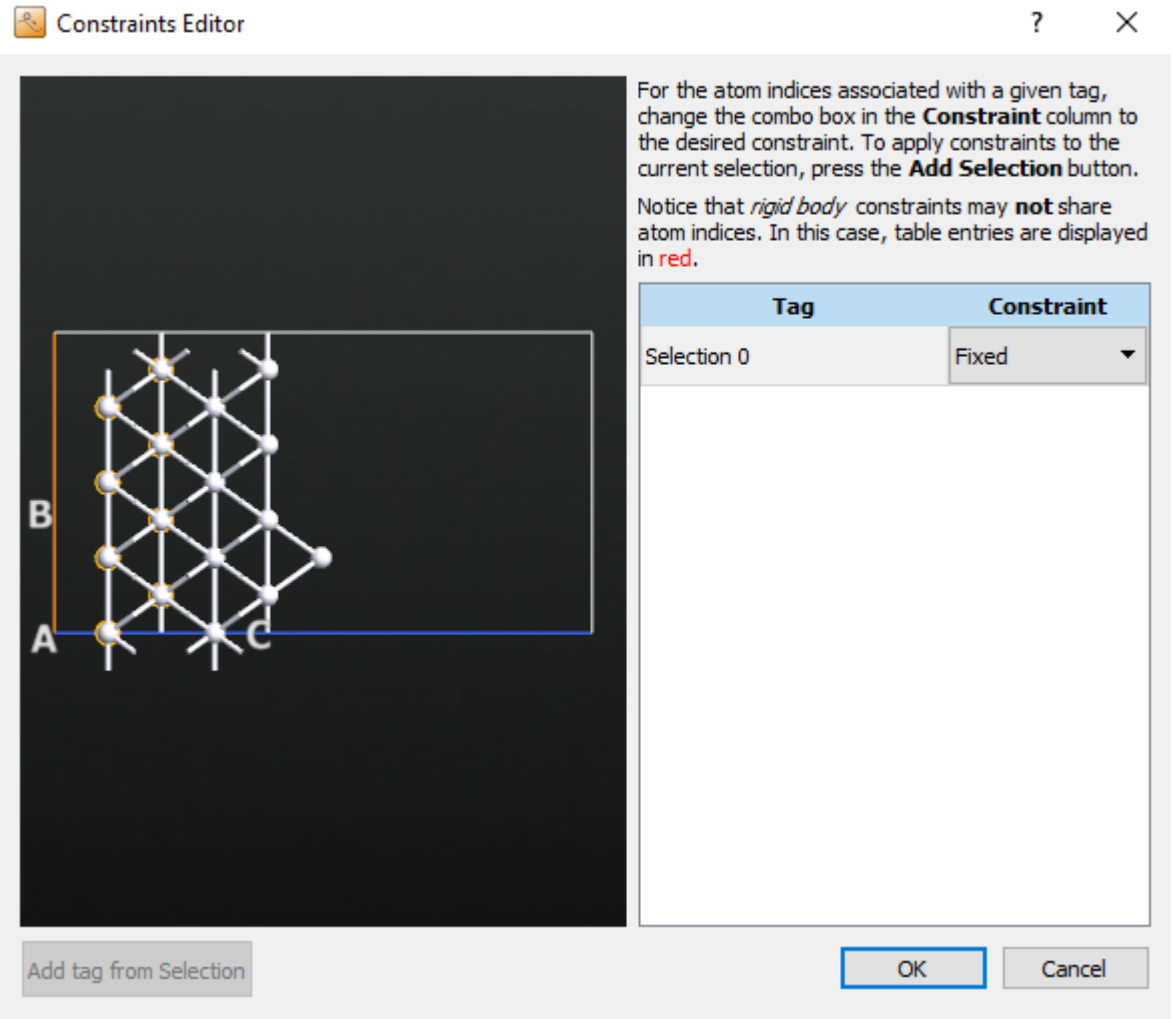
4. Double-click the **New Calculator** block and modify the following parameters:

   - Select the ATK-ForceField calculator
   - Choose the EAM_Pt_2004 Parameter set

5. In the **OptimizeGeometry** block, change the following parameters:

   - Set the maximum force to 0.01 eV/Å

•Click the *Atomic Constraint Editor* button, select the 2 bottom layers, click *Add tag from Selection*, and change the constraint type from None to *Fixed*



Now, you will add the AKMC part of the script.

Note: Starting from the **ATK 2017** version, you can use the **AdaptiveKineticMonteCarlo** block in the Script Generator. While if you have an older version, you need to follow the below instructions on appending the AKMC code in the editor. *Link to 2016 and 2015 versions*.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.

In the **2017** version, add the *OptimizeGeometry* → *AdaptiveKineticMonteCarlo* object, and double click the **AdaptiveKineticMonteCarlo** block. You can see the warning for the Atomic Constraint Editor setting. You need to define the atomic constraint *again*, usually in the same way as in the Optimization, but not necessarily

•In **Kinetic Monte Carlo**, change the *confidence* to 0.95

•In **Saddle Search** Settings, set the *number of searches* to 100

•Click the *Atomic Constraint Editor* in the **Optimization**, select the two bottom layers, change to *Fixed* for the constraint.

•For **Nudged Elastic Band**, *Maximum number of images* is 5 and check *the Climbing image method*

Note: If you need more information to setup the condition, click the **Help** button in the Adaptive Kinetic Monte Carlo widget.

If you now send the script to the editor, you will see the following at the end of the script.

```
# ------------------------------------------------------------
# Adaptive Kinetic Monte Carlo
# ------------------------------------------------------------

saddle_search_parameters = SaddleSearchParameters(
    max_neb_images=5,
    neb_climbing_image=True,
    )

if os.path.isfile('akmc_markov_chain.nc'):
    markov_chain = nlread('akmc_markov_chain.nc')[-1]
else:
    markov_chain = MarkovChain(
        configuration=bulk_configuration,
        configuration_energy=TotalEnergy(bulk_configuration).evaluate(),
        )

if os.path.isfile('akmc_kmc.nc'):
    kmc = nlread('akmc_kmc.nc')[-1]
else:
    kmc = None

constraints = [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
                13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
                26, 27, 28, 29, 30, 31]

akmc = AdaptiveKineticMonteCarlo(
    markov_chain=markov_chain,
    kmc_temperature=300.0*Kelvin,
    md_temperature=2000.0*Kelvin,
    calculator=bulk_configuration.calculator(),
    kmc=kmc,
    saddle_search_parameters=saddle_search_parameters,
    constraints=constraints,
    confidence=0.95,
    filename_prefix='akmc',
    )

akmc.run(max_searches=100, max_kmc_steps=10000)
```

Note: If you look at the **2017** version of the script, there are two *if* blocks for repeating runs; it reuses the already

completed saddle searches.

In **ATK 2016** and **ATK 2015**, send the script to the **Editor** using the **Send To** icon. Then append the below code to the bottom of the script.

•ATK 2016

```
# ------------------------------------------------------------
# AKMC
# ------------------------------------------------------------
saddle_search_parameters = SaddleSearchParameters(max_neb_images=5)
markov_chain = MarkovChain(bulk_configuration, TotalEnergy(bulk_configuration).
↪evaluate())
akmc = AdaptiveKineticMonteCarlo(markov_chain,
                                 kmc_temperature=300.0*Kelvin,
                                 md_temperature=2000*Kelvin,
                                 calculator=calculator,
                                 saddle_search_parameters=saddle_search_parameters,
                                 write_searches=True,
                                 constraints=fix_atom_indices_0,
                                 confidence=0.95)
akmc.run(100)
```

•ATK 2015

```
# ------------------------------------------------------------
# AKMC
# ------------------------------------------------------------
saddle_search_parameters = SaddleSearchParameters(max_neb_images=5)
markov_chain = MarkovChain(bulk_configuration, TotalEnergy(bulk_configuration).
↪evaluate())
akmc = AdaptiveKineticMonteCarlo(markov_chain,
                                 kmc_temperature=300.0*Kelvin,
                                 md_temperature=2000*Kelvin,
                                 calculator=calculator,
                                 saddle_search_parameters=saddle_search_parameters,
                                 write_searches=True,
                                 constraints=constraints,
                                 confidence=0.95)
akmc.run(100)
```

Note: The only difference between the **2016** and **2015** versions is how the constraints are defined.

This code defines an AKMC simulation that will model the dynamics of the system at 300 K (kmc_temperature). The MD saddle searches will be performed at 2000 K (md_temperature). The confidence is set to 95% (confidence).

Save the script as `akmc.py` and send it to the **Job Manager** and run it.

In this case, the calculation will take around 2 hours to complete when run in serial. Additionally, the AKMC simulation can be parallelized using MPI. When run in parallel one MPI process is reserved as the *master* process that schedules the work. The remaining *worker* processes are then used to carry out the saddle searches. This means that if two MPI

processes are used no speedup will be observed since there will only be one worker process. There should be a linear speedup as the number of MPI processes is increased.

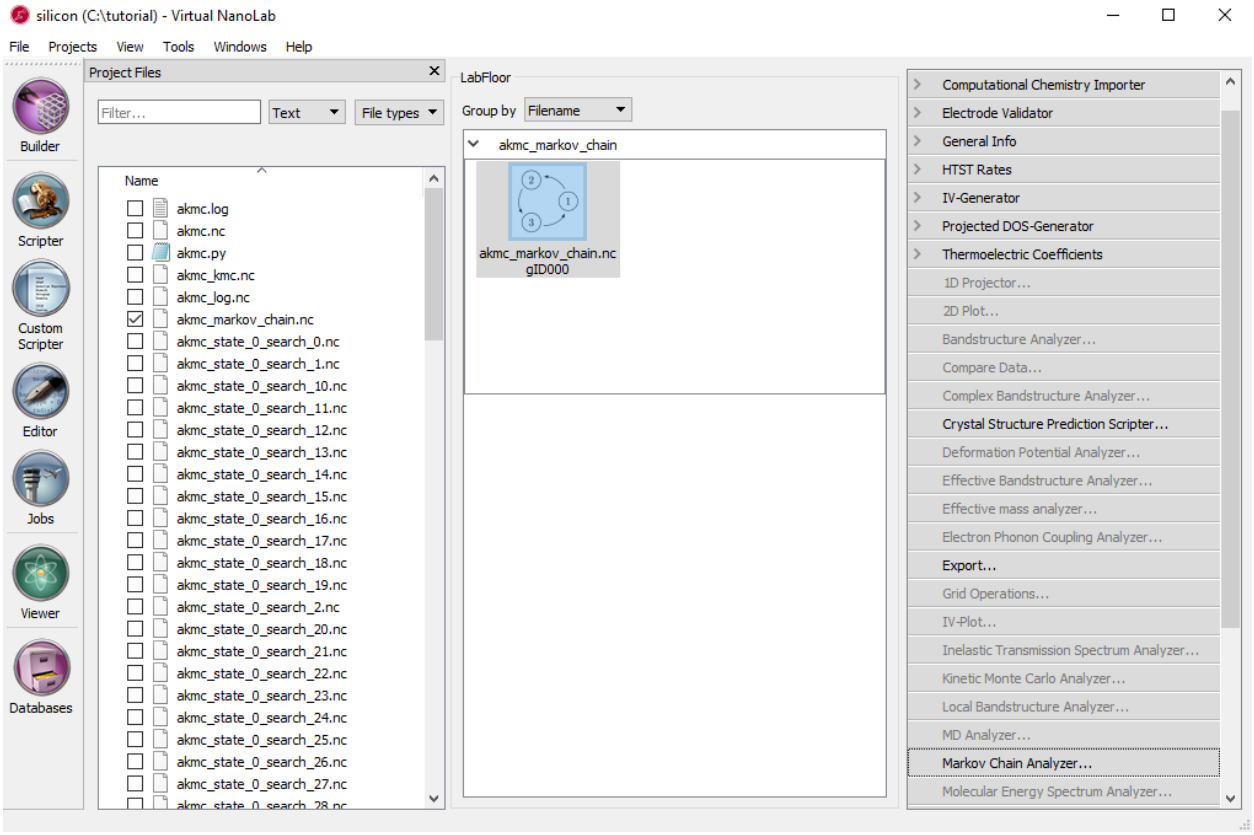
Analyzing the results

Note: Since AKMC is a stochastic method, your results may differ slightly from what is shown here.

Since a large number of files are created during the calculation, it may be useful to click on the list of project files and click *Uncheck*. But if you uncheck the *write searches* box in the *I/O settings* in the AKMC widget or set *write_searches = False* in the script, fewer files will be written.

Markov Chain Analyzer

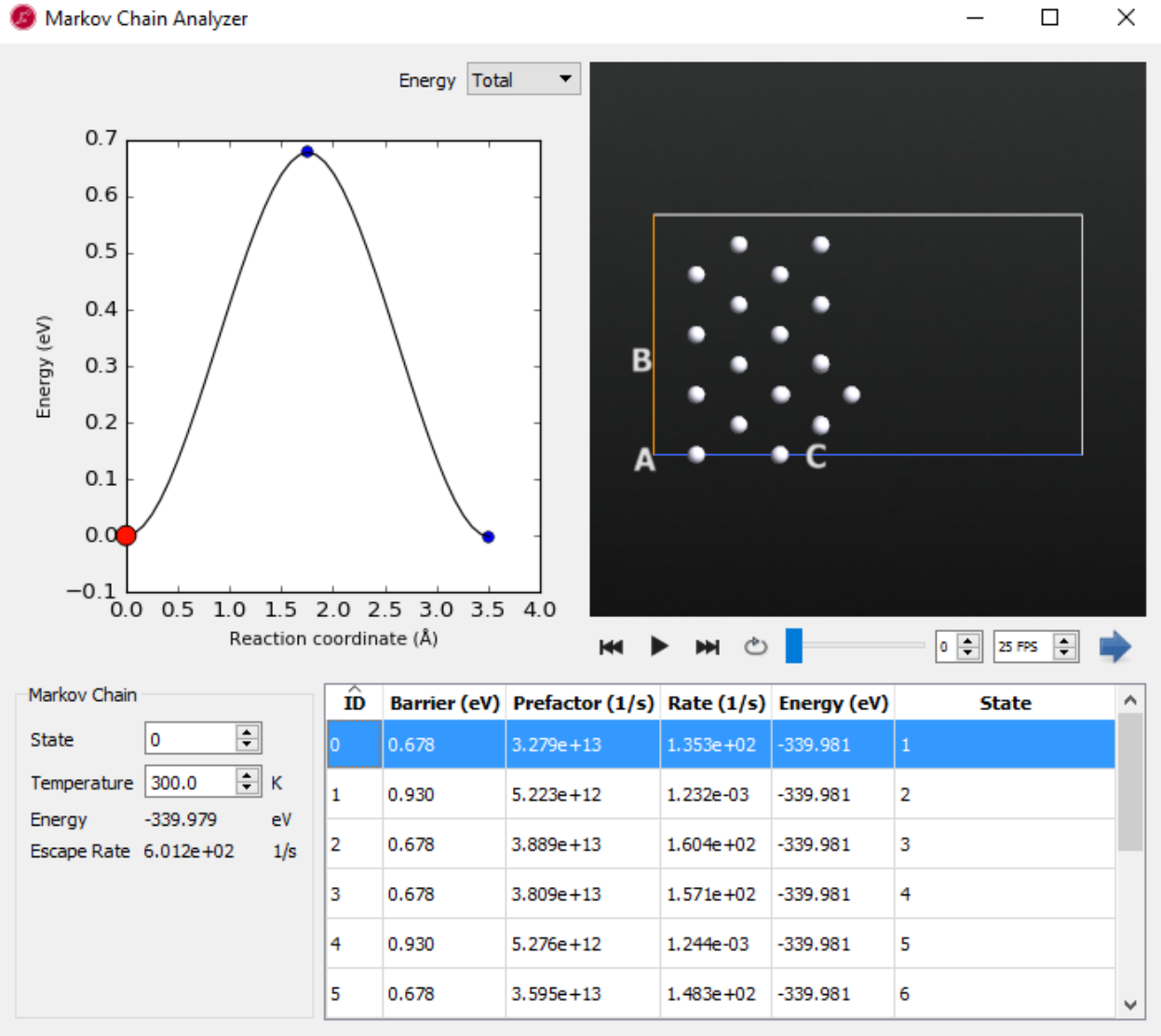
The file `akmc_markov_chain.nc` contains all of the saddle points and minima that were found during the calculation. Check the box next to the filename to load it onto the **LabFloor**. Then select it and click on the **Markov Chain Analyzer** button in the panel.



The new window that popped up can be used to examine all of the minima and saddle points that make up the Markov chain.

State 0 is shown in the below table, this corresponds to the initial configuration made at the beginning of this tutorial. By clicking on item in the table, different events can be visualized. In the top left, an energy coordinate diagram is shown for the event and in the top right is a viewer for the trajectory of the minima and saddle point configurations.

For each event, the energy barrier, harmonic prefactor (the variable A in the HTST equation above), reaction rate at the current temperature, and the energy of the final state as well as the state number for the final state are displayed.



ID	Barrier (eV)	Prefactor (1/s)	Rate (1/s)	Energy (eV)	State
0	0.678	3.279e+13	1.353e+02	-339.981	1
1	0.930	5.223e+12	1.232e-03	-339.981	2
2	0.678	3.889e+13	1.604e+02	-339.981	3
3	0.678	3.809e+13	1.571e+02	-339.981	4
4	0.930	5.276e+12	1.244e-03	-339.981	5
5	0.678	3.595e+13	1.483e+02	-339.981	6

In these results you can see that there are 4 identical reactions with a barrier of 0.678 eV. By using the viewer to look at the saddle point configuration it is apparent that these correspond to the so-called exchange mechanism of adatom diffusion where the adatom pushes a surface atom up onto the surface.

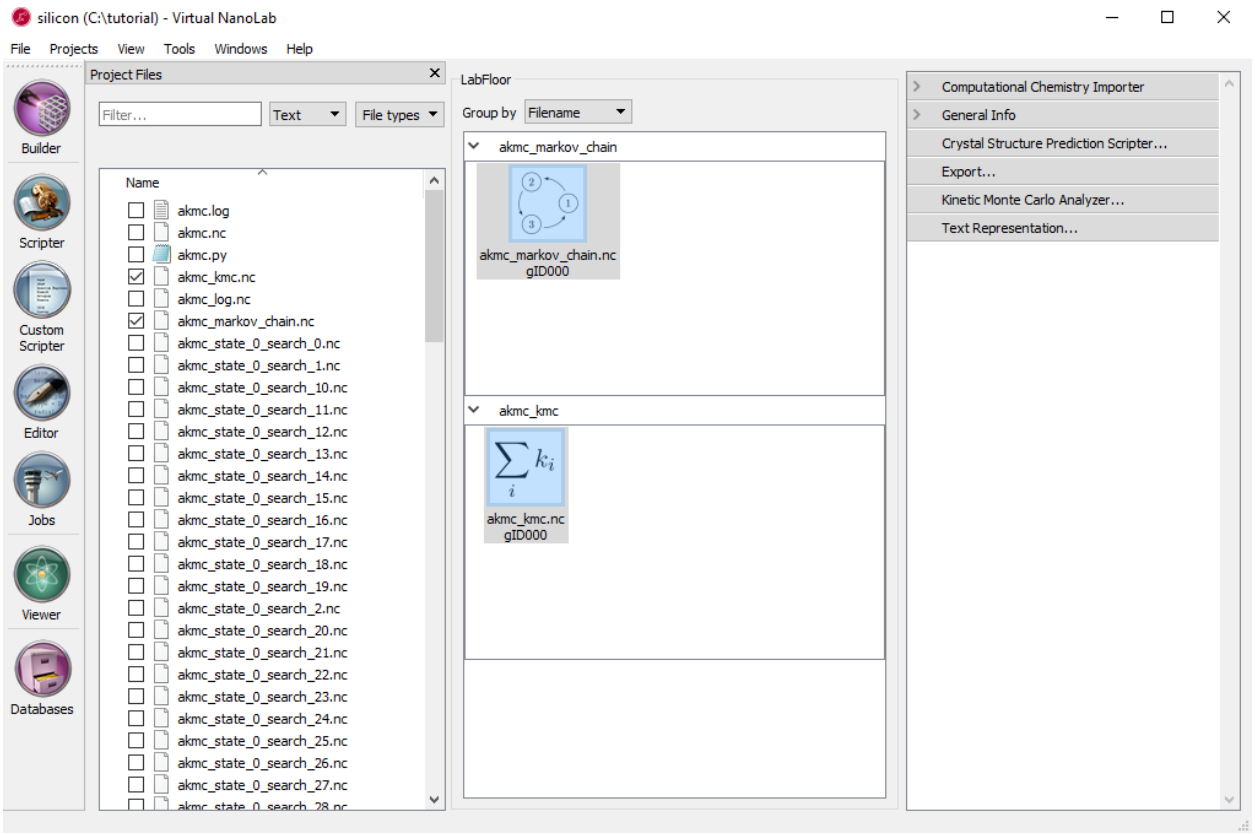
This is in contrast to the most obvious diffusion mechanism - the hop - where the adatom directly moves to an adjacent four-fold hollow site. The relative rates of these two diffusion mechanisms show that at room temperature the exchange mechanism is the primary way that the adatom diffuses. This is because the rate of the exchange mechanism is 5 orders of magnitude larger than the hop mechanism.

Kinetic Monte Carlo Analyzer

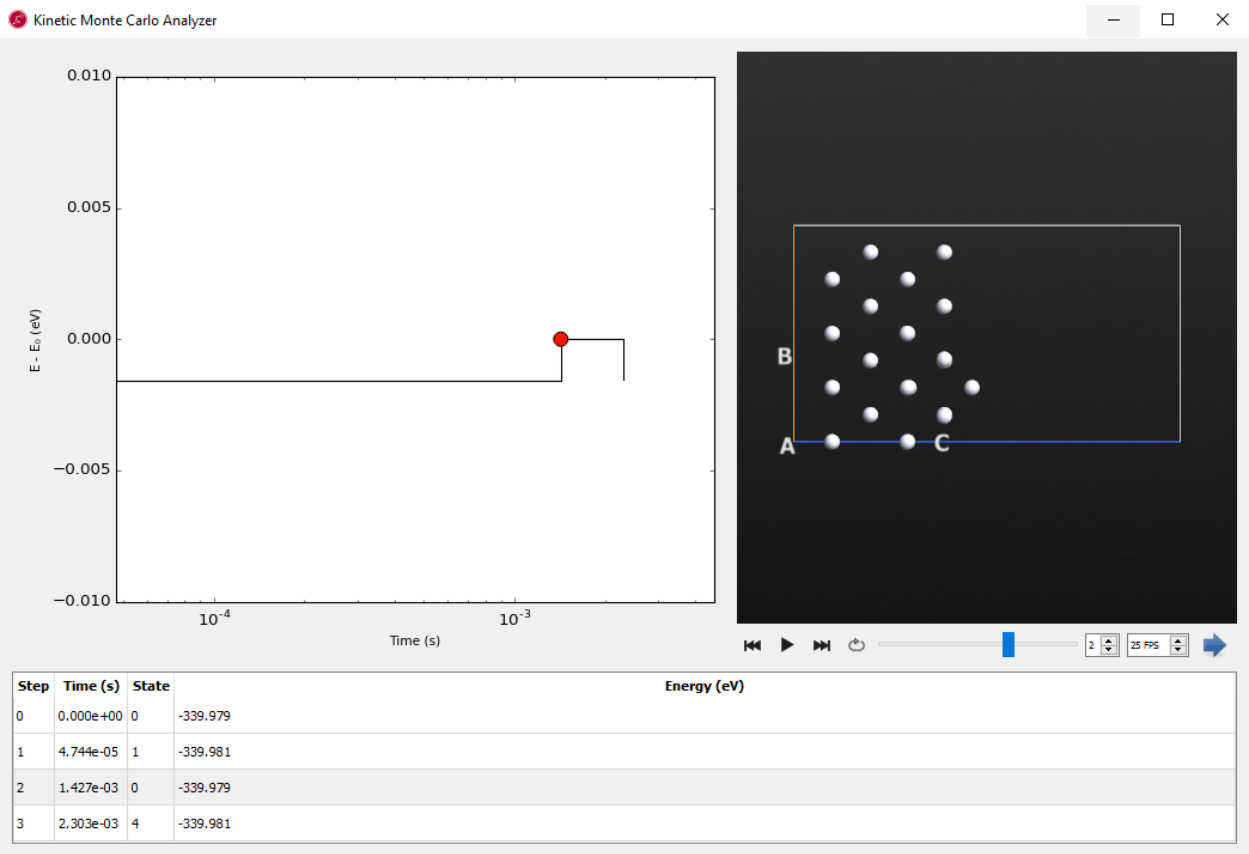
The file named `akmc_kmc.nc` contains the state-to-state KMC trajectory. Click the checkbox next to the filename to load it onto the **LabFloor**. Click on its icon and click **Text Representation** in the panel. The popup window will display the state of the KMC simulation as a function of time. The state number in the right-hand column corresponds to the state numbers in the Markov chain file we just looked at. In this example, the simulation took one step from state 0 to state 1 and 47 microseconds of time elapsed.

```
# Item:  0
# File:  C:\tutorial\akmc_kmc.nc
# Title: akmc_kmc.nc - gID000
# Type:  KineticMonteCarlo
     Step #   Time (seconds)  State
        0   0.00000000e+00      0
        1   4.74388502e-05      1
        2   1.42688935e-03      0
        3   2.30287130e-03      4
```

A graphical representation of the KMC trajectory is also available. In order to use it, both the *KineticMonteCarlo* and *MarkovChain* objects on the **LabFloor** must be selected. Then you may click on **Kinetic Monte Carlo Analyzer** in the panel.



The Kinetic Monte Carlo Analyzer displays the same information shown in the text representation in the table at the bottom of the windows. However, it also makes a plot of the energy versus time and shows the geometry of the currently selected state. Since the energy plot is logarithmic on the time axis the first state at zero seconds cannot be seen, and there are only three data points. The energies are plotted by the difference between the first state 0 and searched state versus time.



Step	Time (s)	State	Energy (eV)
0	0.000e+00	0	-339.979
1	4.744e-05	1	-339.981
2	1.427e-03	0	-339.979
3	2.303e-03	4	-339.981

AKMC Log

The next file to examine is `akmc_log.nc`. This file contains a brief summary of the outcome of each of the saddle searches used to find the states shown above. Click the checkbox next to the filename to load it onto the **LabFloor**. Click on its icon and click **Text Representation** in the panel.

```
# Item:  0
# File:  C:\tutorial\akmc_log.nc
# Title: akmc_log.nc - gID000
# Type:  AKMCLog
state id search number confidence  message
       0             0   0.000000  Found new state
       0             1   0.864665  Found saddle connecting to state 1 again
       0             2   0.864657  Found new state
       0             3   0.395535  Found new state
       0             4   0.864661  Found saddle connecting to state 3 again
       0             5   0.564633  Found new state
       0             6   0.564631  Found new state
       0             7   0.590186  Found saddle connecting to state 1 again
       0             8   0.590186  First order saddles have exactly one negative␣
↪frequency, not 2 negative frequencies.
       0             9   0.590188  Found saddle connecting to state 5 again
       0            10   0.590188  First order saddles have exactly one negative␣
↪frequency, not 2 negative frequencies.
       0            11   0.444580  Found new state
```

(continues on next page)

(continued from previous page)

```
      0            12   0.670587  Found saddle connecting to state 4 again
      0            13   0.670587  First order saddles have exactly one negative␣
↪frequency, not 2 negative frequencies.
      0            14   0.670587  First order saddles have exactly one negative␣
↪frequency, not 2 negative frequencies.
      0            15   0.670587  First order saddles have exactly one negative␣
↪frequency, not 2 negative frequencies.
      0            16   0.692947  Found saddle connecting to state 4 again
      0            17   0.700029  Found saddle connecting to state 1 again
```

The output above shows the first 17 lines of output. Each line corresponds to a single saddle search.

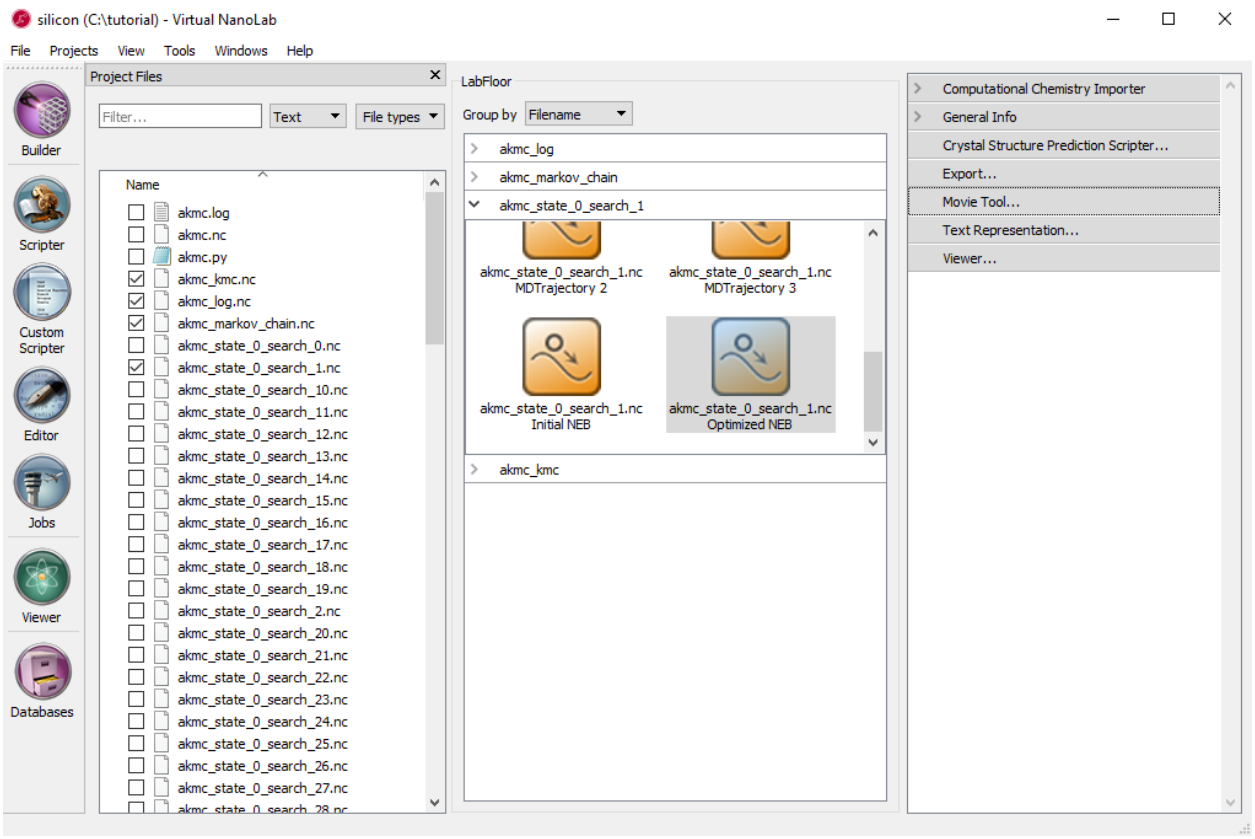
The *state id* column represents the id number for the state that the saddle searches are started from. The *search number* is unique identifier for each search. The *confidence* represents an estimate of the confidence in the KMC simulation for the state. The confidence is used as the stopping criteria for the saddle searches. The *message* column summarizes the result of the saddle search.

The message *Found new state* means that a new reaction mechanism was discovered. The message *Found saddle connecting to state 1 again* means that a previously discovered saddle point was found.

There are a number of possible error messages that can occur. In this case the message *First order saddles have one negative frequency not 2 negative frequencies* means that a saddle point was found, but when the vibrational modes were calculated more than one negative frequency was identified. There could be a number of reasons for this issue. Often times classical potentials are not fit near saddle points and can have unexpected behavior.

Diagnosing Errors

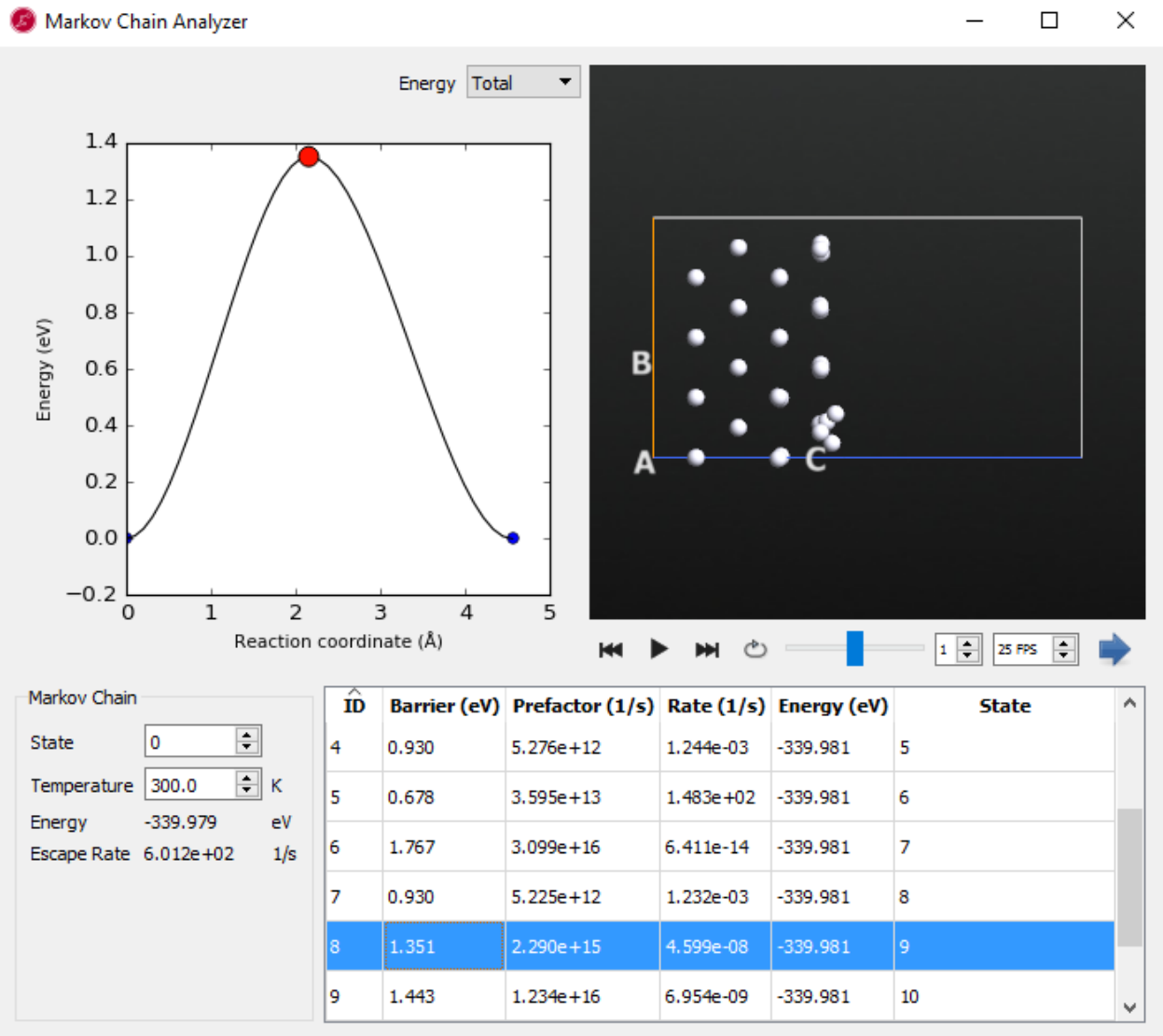
When there is an issue with a saddle search, an error message will be displayed in the log. The *state id* and *search number* columns can then be used to locate a file with more information about the search, if the parameter *write_searches=True* was passed to the *AdaptiveKineticMonteCarlo* class. In this case we look at search number 1. The filename is `akmc_search_0_search_1.nc`. Remember, in your simulation the numbers may be different. Click the checkbox by the filename of akmc search you wish to inspect. Then you can click on the *Optimized NEB* item and then click on **Movie Tool** in the panel to visualize the reaction pathway.



In general, if it was lower energy barrier, it will be an important reaction path. But in this case, we see that this is another type of exchange diffusion mechanism even higher in energy than the hopping mechanisms that we already showed was too slow to contribute to adatom diffusion.

By using the *Movie Tool* in the **Markov Chain Analyzer**, we can examine each state to understand the diffusion mechanism.

In a real study it is always important to verify the results of a classical potential and re-optimizing this NEB using DFT would be a good idea to determine if it was a real pathway in the system.



Conclusion

In this tutorial we examined how to construct a Pt(100) slab model with a single adatom. We then performed an AKMC simulation to model the diffusion of the adatom at 300 K. The results showed two different diffusion mechanisms: the exchange and hop processes. The exchange process was found to be the dominant mechanism for adatom diffusion.

This simulation was a very simple case. No matter how the adatom diffuses the system is the same by symmetry (i.e. the adatom will always have the same set of 4 exchange and 4 hop diffusion mechanisms). So once one state has been explored, there is nothing left to learn. However, in the current implementation, the AKMC code does not take symmetry into account and must do saddle searches for each new state. Most systems of interest do not contain such high levels of symmetry.

References

3.11.20 Crystal Structure Prediction Scripter: Phases of TiO_2

Version: 2016.2

In this tutorial, we will use the Crystal Structure Prediction method to investigate the possible crystal structures of TiO_2. At zero temperature and pressure, the *rutile* phase of TiO_2 is the most stable one, while the *anatase* and *brookite* phases are meta-stable. Both rutile and anatase can be described in a unit cell of 6 atoms, while brookite requires a much larger cell. QuantumATK uses a genetic algorithm for crystal structure prediction.

Note: A **genetic algorithm** is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The algorithm repeatedly modifies/refines a population of individual solutions.

In practice, the algorithm starts out from an initial population of crystals (a generation), and iteratively refines the generation by producing new generations through application of **genetic operators** to the previous generation, e.g. *promotion* (lowest-energy crystals live on), *heredity* (mixing of good genes from two different low-energy crystals), as well as *mutation* and *permutation* (ensures genetic diversity).

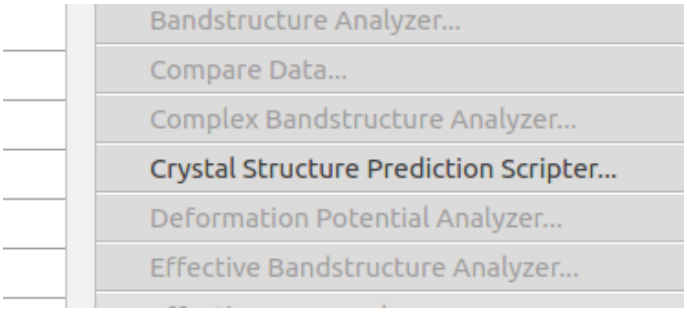
Please refer to[1],[2] and[3] for more information.

Setting up the calculation

We will use the **Crystal Structure Prediction Scripter**, which is available in recent versions of QuantumATK. This custom scripter widget is used to produce a template script for running the crystal structure prediction algorithm. The scripter does not add script lines defining the specific calculator engine to be used – they must be added manually.

Crystal Structure Prediction Scripter

Open the Crystal Structure Prediction Scripter from the right-hand plugins panel in QuantumATK.



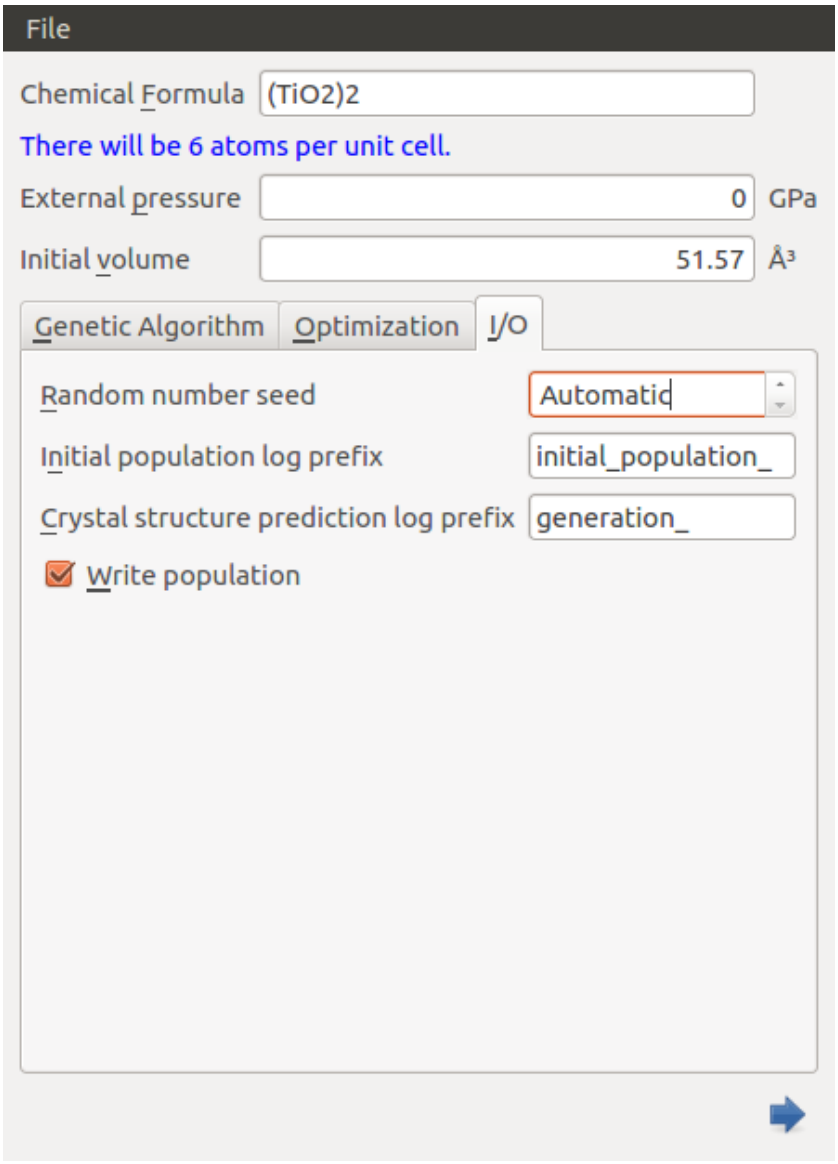
The widget will open, and will initially display the I/O tab.

[1] WilliamăW Tipton and RichardăG Hennig. A grand canonical genetic algorithm for the prediction of multi-component phase diagrams and testing of empirical potentials. *J. Phys. Condens. Matter*, 25(49):495401, 2013. doi:10.1088/0953-8984/25/49/495401.

[2] ArtemăR. Oganov. *Modern Methods of Crystal Structure Prediction*. Wiley, 2011. ISBN 9783527643776.

[3] Eva Zurek. *Discovering New Materials via A Priori Crystal Structure Prediction*, pages 274–326. John Wiley & Sons, Inc, 2016. doi:10.1002/9781119148739.ch5.

First of all, you need to write the **chemical formula** of the crystal to investigate. In this case, you should write (TiO2)2 (or Ti2O4) as we need two TiO_2 formula units to describe the unit cells of both anatase and rutile. Leave the external pressure at 0 GPa. The initial volume will be set automatically, based on the expected volume of the elements. The window should now look as seen below.



Genetic algorithm settings

Next, set up the parameters for the genetic algorithm, which is used to search for viable crystal structures. Click the tab named Genetic Algorithm to start. It has the following parameters:

Population size
The number of crystal structures in each generation. Set this to 10.

Number of generations
The number of steps the algorithm will run. Set this to 20.

Selection pressure
Determines how strongly the fitness score (see note below) will be taken into account when individuals are selected for

application of a genetic operator. A high value will make it much more likely that genetic operators are applied to the individuals with the highest fitness, while a value of 0 will make all (elite, see below) individuals equally likely. Leave this value at 2.00.

Number of elites
The number of individuals which will be considered when creating the next generation. The difference between Population size and Number of elites indicates the number of individuals with lowest fitness that will simply be discarded after each iteration step. Leave this value at 10, so that no individuals will be discarded.

Number to promote
The number of individuals, with the highest fitness scores, which are promoted (copied) to the next step without any alteration. Set this to 4.

Note: The fitness score is defined as minus the enthalpy. This ensures that the algorithm searches for the thermodynamically most stable structures.

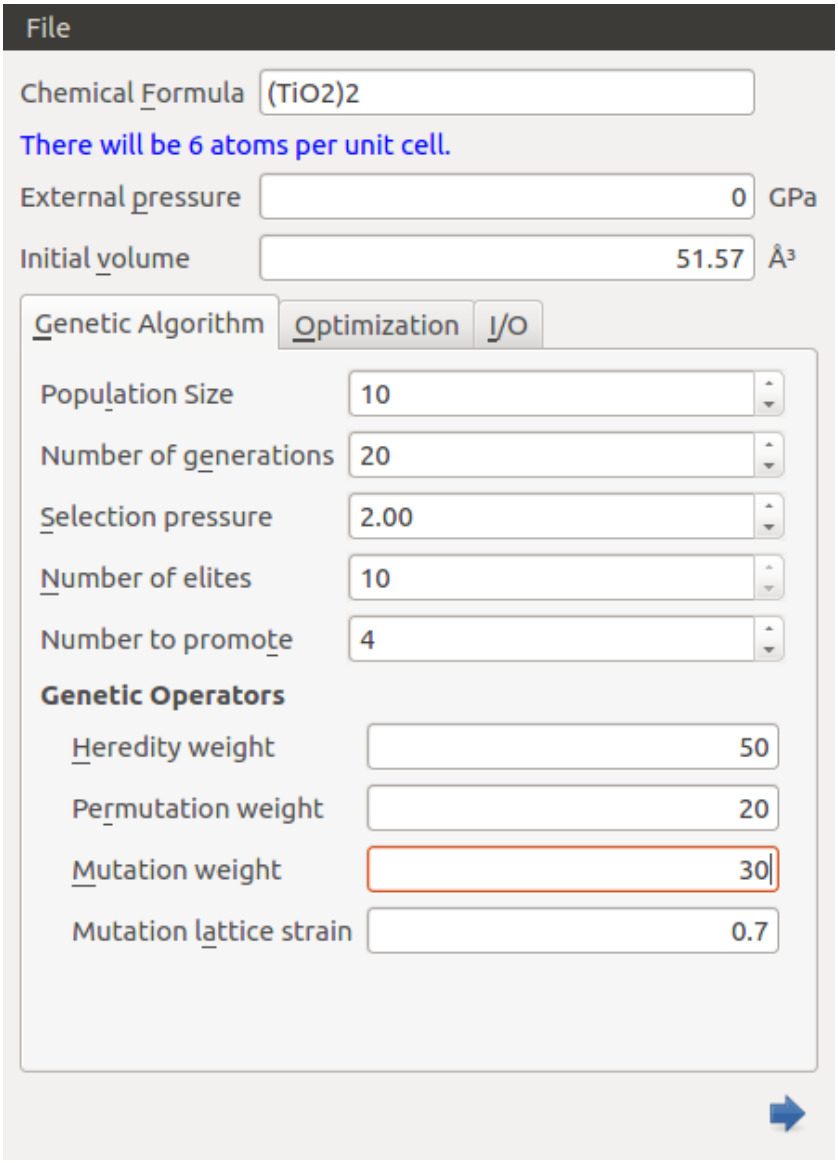
The remaining members of each new generation are created by applying **genetic operators** to the elite individuals from the previous generation. There are three genetic operators, with an additional setting for one of them. The probability of applying each operator is calculated from the following **weights**, which may be considered to be unnormalized probabilities.

Heredity weight
The unnormalized probability that the *Heredity* operation is used to create the next individual in the next generation. This operator uses two structures, and combines one-half of each as determined by a randomly chosen cut-plane. Set this value to 50.

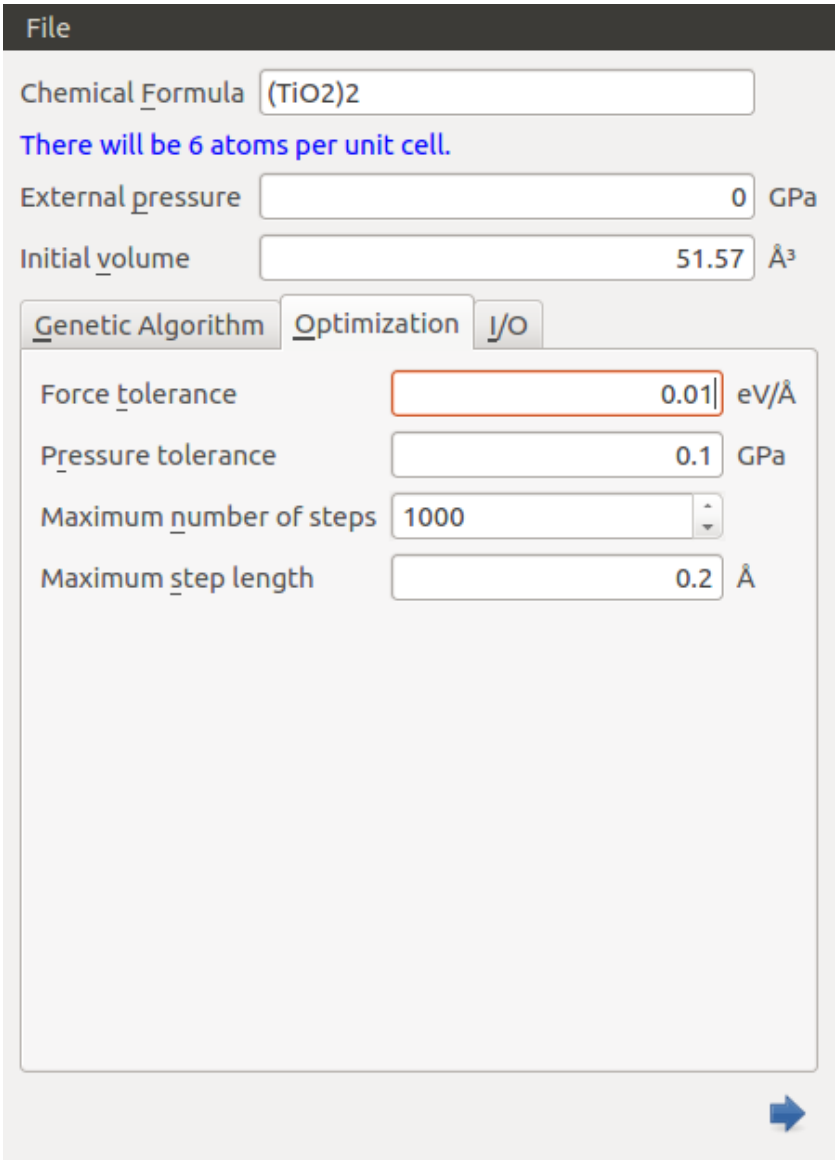
Permutation weight
The unnormalized probability of using the *Permutation* operator. This operator will exchange the positions of two randomly selected atoms of different elements. Set this value to 20.

Mutation weight
The unnormalized probability of using the *Mutation* operator. This operator applies a symmetric strain matrix with elements drawn randomly from a gaussian distribution with a standard deviation of *Mutation lattice strain*. This will change the lattice parameters and angles of the unit cell. Set the *Mutation weight* to 30 and leave the *Mutation lattice strain* at 0.7.

The Crystal Structure Prediction Scripter should now look like this:



Next, go to the **Optimization** tab and verify that it looks like this:

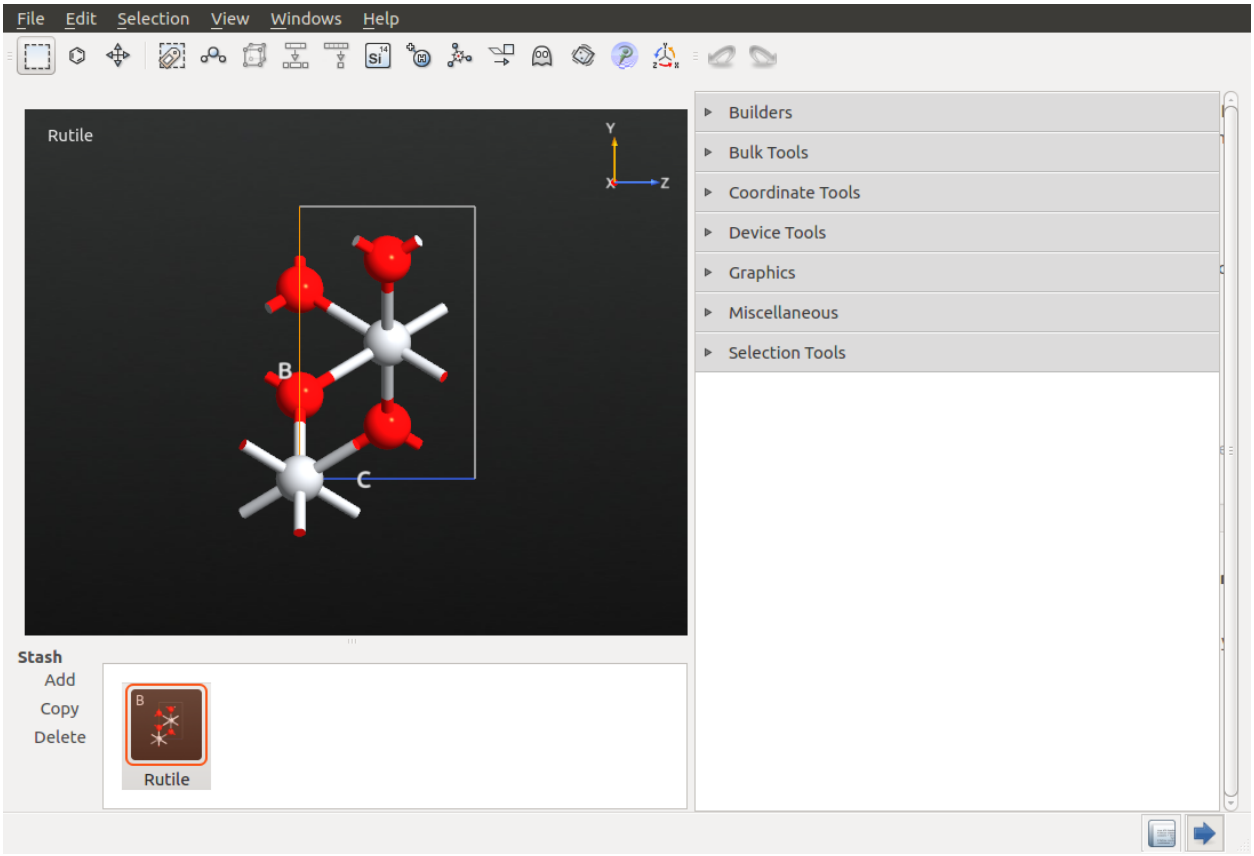


Tip: You can of course change the geometry optimization settings, e.g. to tighten the force tolerance, but the default values should work well for most purposes.

Send the script to the **Editor** using the button. The produced script should be identical to this one: `csp-script.py`. Note that `calculator = None`. You therefore need to manually add the script lines defining the calculator to be used (could be an *ATK-DFT* or *ATK-ForceField* calculator).

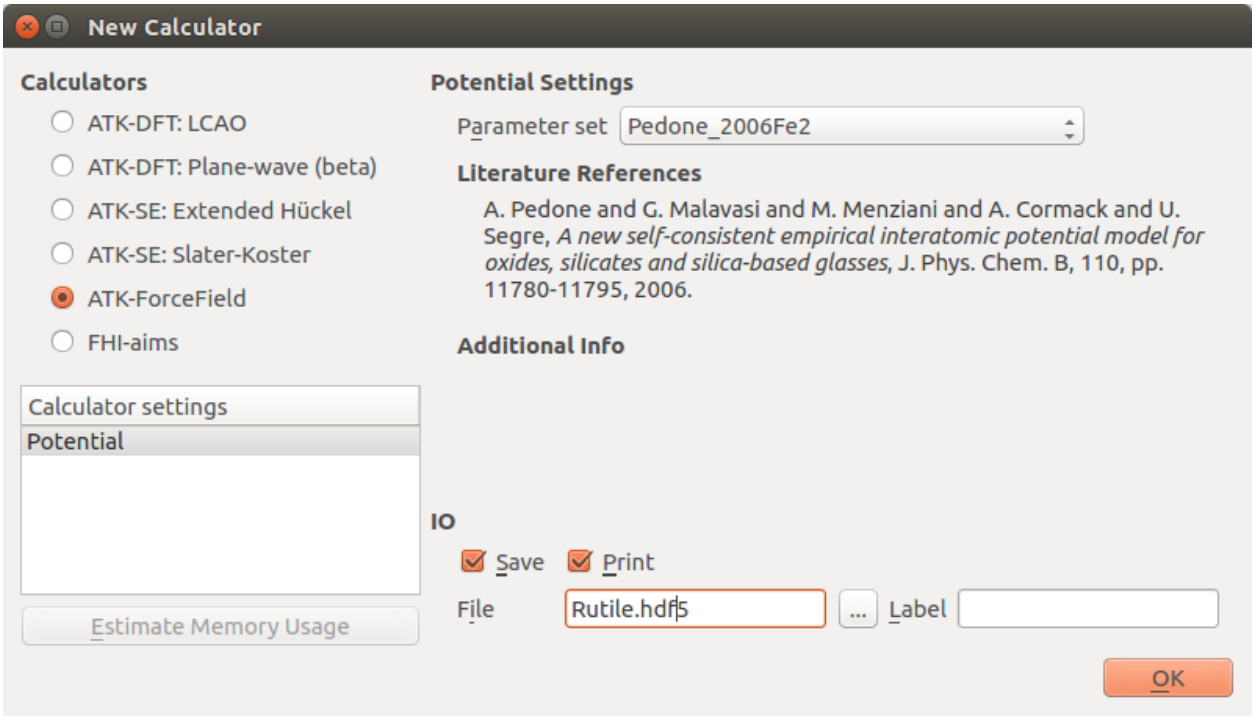
Adding a calculator

In principle, you can just add a calculator to the script, particularly if you already have an appropriate one defined in another script or can write it up manually. However, here we will go through the steps of defining it using QuantumATK. Open the **Builder** and find the rutile structure of TiO_2:



Send this configuration to the **Script Generator** and set up an **ATK-ForceField** calculator. QuantumATK will automatically suggest relevant parameter sets. In this case, choose **Pedone_2006Fe2**.

Note: For QuantumATK-versions older than 2017, the **ATK-ForceField** calculator can be found under the name **ATK-Classical**.



Send the script to the **Editor**, and copy the script lines defining the calculator:

```
potentialSet = Pedone_2006Fe2()
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
```

Insert those lines into the script generated using the Crystal Structure Prediction Scripter, and save it. You can also download the complete script here: `csp-script-with-calculator.py`

Running the calculations and analyzing results

Run the script using the **Job manager** or from the command line. It will take no more than a few minutes to run.

The script will generate a significant number of files, since it saves an `.nc` file and 6 log-files for each generation. The `.nc` files contain the *BulkConfigurations* for each individual in the given generation, and the log-files contain information from the 6 geometry optimizations carried out in each generation. This is the total population minus the number of promoted individuals, as they do not need to be re-optimized. However, we will focus on the overall log-file, in this case called `csp-script-with-calculator.log`. We are particularly interested in the information about the final generation:

```
+-------------------------------------------------------------------------+
| Niching                                                                 |
+-------------------------------------------------------------------------+
| Killing individual  0                                                   |
| Killing individual  1                                                   |
| Killing individual  4                                                   |
| Killing individual  6                                                   |
+-------------------------------------------------------------------------+
| Population for Final Niched Generation                                  |
```

(continues on next page)

(continued from previous page)

```
+-------------------------------------------------------------------------------+
| Individual   0: Fitness:   94.238940                                          |
|                 Symmetry:  P4_2/mnm      Volume:     63.5550                  |
|                 Max Force: 0.0056        Max Stress Error: 0.0929             |
|                 History: IPPPPPPPPPPPPPPPPPPP                                 |
| Individual   1: Fitness:   93.978255                                          |
|                 Symmetry:  I4_1/amd      Volume:     70.0359                  |
|                 Max Force: 0.0025        Max Stress Error: 0.0380             |
|                 History: IPPPPPPPPPPPPPPPPPPP                                 |
| Individual   2: Fitness:   93.796305                                          |
|                 Symmetry:  Fd-3m         Volume:     139.6049                 |
|                 Max Force: 0.0099        Max Stress Error: 0.0357             |
|                 History: IXPPPPPPPPPPPPPPPPPP                                 |
| Individual   3: Fitness:   93.625090                                          |
|                 Symmetry:  Cmcm          Volume:     87.9469                  |
|                 Max Force: 0.0074        Max Stress Error: 0.0533             |
|                 History: IPPPPPPPPPPPPPPPPPPP                                 |
| Individual   4: Fitness:   93.388660                                          |
|                 Symmetry:  Imcm          Volume:     100.2970                 |
|                 Max Force: 0.0092        Max Stress Error: 0.0912             |
|                 History: IPPPPPPPPPPPPPPPPPPH                                 |
| Individual   5: Fitness:   92.364891                                          |
|                 Symmetry:  Imm2          Volume:     86.5277                  |
|                 Max Force: 0.0042        Max Stress Error: 0.0761             |
|                 History: IPPPPPPPPPPPPPPPPPPH                                 |
+-------------------------------------------------------------------------------+
| Legend: I=Initial P=Promotion H=Heredity M=Mutation X=Permutation             |
|         fitness in units of eV                                                |
|         force   in units of eV/Angstrom                                       |
|         stress  in units of GPa                                               |
+-------------------------------------------------------------------------------+
```

First, the genetic algorithm performs **Niching**, which is removal of any duplicate structures from the generation. Below that, we see the final generation, sorted by decreasing fitness, which is simply the negative of the enthalpy. We see that the first two structures (those with lowest energy) have *P4_2/mnm* and *I4_1/amd* symmetries, consistent with the rutile and anatase phases of TiO_2.

The **History** shows how the algorithm arrived at this particular generation, and it turns out that, in this case, both the rutile and anatase structures have simply been promoted all the way from the initial population. Note also how the volume varies quite a bit between the different crystal structures, showing how the genetic algorithm is able to find highly different local minima on the potential energy surface of this system.

References

3.11.21 DFT-D and basis-set superposition error

In this tutorial you will calculate the distance between two graphene layers. To calculate this distance you will need to account properly for van der Waals interactions (also termed dispersion interactions). The latter are underestimated by local (LDA) and semi-local (GGA) exchange-correlation functionals. As a result, using these functionals the layers do not bind (or bind too weakly) and the minimal total energy configuration becomes one where the layers are infinitely separated. The long-range van der Waals interaction will be included through the semi-empirical corrections

by Grimme D2[1] and D3[2]. These corrections do not attempt to describe the actual source of the interaction (fluctuating dipoles) but rather its effect on the DFT mean-field effective potential.

You will also learn that incompleteness of the LCAO basis set can give rise to a so-called basis set superposition error (BSSE). The BSSE will contribute with an artificial attraction between the graphene layers, which actually partly compensates for the lack of van der Waals attraction, so with some exchange-correlation potentials the graphene layers will bind, but for the wrong reason. Thus, if you introduce the Grimme correction but do not account for BSSE, the layers will instead bind too strongly and a too small layer separation will be obtained. To remove the BSSE you will use the counterpoise correction[3].

The BSSE is mostly noticeable when computing cohesive energies or binding energies where two subsystems can be clearly defined, like in the case of a molecule on a surface[4] or, as in this tutorial, in layered systems. It is however also an issue for vacancy formation energies, where the material surrounding the vacancy has a lower number of basis orbitals than available in the vacancy case[5].

The DFT-D dispersion corrections

The corrections by developed by Grimme an co-workers[Page 1659, 1 2] add an additional term E_{disp} to the DFT total energy E_{DFT} in order to account for van der Waals interactions:

$$E_{\mathrm{DFT+D}} = E_{\mathrm{DFT}} + E_{\mathrm{disp}}$$

The D2 and D3 corrections differ in the way the term E_{disp} is evaluated.

D2 correction

In the D2 correction[1] the E_{disp} term include only two-body energies and is given by an attractive semi-empirical pair potential V^{PP} which accounts only for the lowest-order dispersion term:

$$E_{\mathrm{disp}} = -S_6 \sum_{A<B} V^{\mathrm{PP}}(Z_A, Z_B, R_{AB}),$$

where R_A (R_B) and R_{AB} are the atomic number of atom A (atom B) and the interatomic distance, respectively. The global scaling parameter S_6 has been fitted to reproduce the thermochemical properties of a large training set of molecules and depends on the exchange-correlation functional employed.

The pair potential between two atoms with atomic numbers A and B separated by a distance R_{AB} is:

$$V^{\mathrm{PP}} = \frac{\sqrt{C_6^A C_6^B}}{R_{AB}^6} f(R_{AB})$$

where

$$f(R_{AB}) = \frac{1}{1 + e^{-d_6\left[\frac{R_{AB}}{R_0^A + R_0^B} - 1\right]}}$$

is a damping function and C_6 and R_0 are element-specific parameters. Typically, the damping factor d_6 is set to 20.

[1] S.ăGrimme. Semiempirical GGA-type density functional constructed with a long-range dispersion correction. *J. Comput. Chem.*, 27(15):1787–1799, 2006. doi:10.1002/jcc.20495.

[2] S.ăGrimme, J.ăAntony, S.ăEhrlich, and H.ăKrieg. A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu. *J. Chem. Phys.*, 2010. doi:10.1063/1.3382344.

[3] S.ăF. Boys and F.ăBernardi. The calculation of small molecular interactions by the differences of separate total energies. Some procedures with reduced errors. *Molecular Physics*, 19:553–566, 1970. doi:10.1080/00268977000101561.

[4] Kurt Stokbro and Soren Smidstrup. Electron transport across a metal-organic interface: simulations using nonequilibrium greens function and density functional theory. *Phys. Rev. B*, 88:075317, 2013. doi:10.1103/PhysRevB.88.075317.

[5] H.ăWu, N.ăZhang, H.ăWang, and S.ăHong. First-principle study of oxygen-vacancy Cu2O(111) surface. *J. Theor. Comput. Chem.*, 11:1261, 2012. doi:http://dx.doi.org/10.1142/S0219633612500848.

D3 correction

In the D3 correction[2], the E_{disp} term includes both two- and three-body energies:

$$E_{\mathrm{disp}} = E^{(2)} + E^{(3)}.$$

Two-body energy term

The two-body energy include dispersion terms up to the 8th order:

$$E^{(2)} = -S_6 \sum_{A<B} V_6^{\mathrm{PP}}(Z_A, Z_B, R_{AB}) - S_8 \sum_{A<B} V_8^{\mathrm{PP}}(Z_A, Z_B, R_{AB}).$$

In the equation above, $S_6 = 1$ for all elements, whereas S_8 depends on the exchange-correlation functional employed. The 6th order pair potential is

$$V_6^{\mathrm{PP}} = \frac{C_6^{AB}}{R_{AB}^6} f_6(R_{AB}),$$

where

$$C_6^{AB} = \frac{\sum_i^{N_A} \sum_j^{N_B} C_{6,\mathrm{ref}}^{AB}\, e^{-4[(n^A - n_i^A)^2 + (n^B - n_j^B)^2]}}{\sum_i^{N_A} \sum_j^{N_B} e^{-4[(n^A - n_i^A)^2 + (n^B - n_j^B)^2]}},$$

and the 8th order pair potential is constructed using the 6th order data as:

$$V_8^{\mathrm{PP}} = \frac{3 C_{6,\mathrm{ref}}^{AB} \left(\sqrt{Z_A} \frac{\langle r^4 \rangle_A}{\langle r^2 \rangle_A} \sqrt{Z_B} \frac{\langle r^4 \rangle_B}{\langle r^2 \rangle_B} \right)^{\frac{1}{2}}}{R_{AB}^8} f_8(R_{AB}).$$

$C_{6,\mathrm{ref}}^{AB}$ are reference dispersion coefficients derived from time-dependent DFT (TDDFT) calculations using a training set of N_A (N_B) molecules for atom A (B). $\langle r^2 \rangle$ and $\langle r^4 \rangle$ are geometrically averaged multipole-type expectation values derived from atomic densities. n^A and n_i^A (n^B and n_j^B) are the atomic fractional coordination number for atom A in the system and in the i-th (j-th) molecule of training set N_A (N_B),

$$n_A = \sum_{B \neq A} \frac{1}{1 + e^{-16[\frac{4}{3}(R_{A,\mathrm{cov}} + R_{B,\mathrm{cov}})/R_{AB} - 1]}},$$

where $R_{A,\mathrm{cov}}$ and $R_{B,\mathrm{cov}}$ have been taken from[6]. Finally, the damping functions for 6th ($f_6(R_{AB})$) and 8th ($f_6(R_{AB})$) orders terms are, respectively:

$$f_6(R_{AB}) = \frac{1}{1 + 6[R_{AB}/(d_6\, R_0^{AB})]^{-14}},$$

and

$$f_8(R_{AB}) = \frac{1}{1 + 6[R_{AB}/(d_8\, R_0^{AB})]^{-16}},$$

with R_0^{AB} being the cutoff radius for the AB element pair, and d_6 and d_8 are damping factors depending on the exchange-correlation functional.

Note: As it is evident from the equations above, in the D3 method V_6^{PP} and V_8^{PP} depend not only on the interatomic distance but also on the chemical enviroment of each atom via the fractional coordination number. This is one of the major differences compared to the D2 method.

---

[6] P.ăPyykkö and A.ăMichiko. Molecular Single-Bond Covalent Radii for Elements 1-118. *Chemistry: a European Journal*, 15:186–197, 2009. doi:10.1002/chem.200800987.

Three-body energy term

The additional three-body energy contribution is evaluated as:

$$E^{(3)} = \sum_{A<B<C} V^{\mathrm{TBP}}(Z_A, Z_B, Z_C, R_{AB}, R_{AC}, R_{BC}, \cos\theta_a, \cos\theta_b, \cos\theta_c),$$

where the three-body potential V^{TBP} is constructed starting from the two-body terms,

$$V^{\mathrm{TBP}} = f(\bar{R}_{ABC}) \frac{\sqrt{C_6^{AB} C_6^{AC} C_6^{BC}}(3\cos\theta_a \cos\theta_b \cos\theta_c + 1)}{(R_{AB} R_{BC} R_{CA})^3},$$

using the following damping function,

$$f(\bar{R}_{ABC}) = \frac{1}{1 + 6\left[\bar{R}_{ABC}/(\frac{4}{3}\bar{R}_0^{ABC})\right]^{-16}}$$

where the geometrically averaged radii are

$$\bar{R}_{ABC} = (R_{AB} R_{BC} R_{CA})^{\frac{1}{3}} \qquad \text{and} \qquad \bar{R}_0^{ABC} = \left(R_0^{AB} R_0^{BC} R_0^{CA}\right)^{\frac{1}{3}}.$$

Note: The evaluation of $E^{(3)}$ changes the scaling behavior of the D3 algorithm from $O(N^2)$ to $O(N^3)$, where N is the number of atoms. Computing the three-body energy term can therefore be extremely expensive for large systems. Since the magnitude of the $E^{(3)}$ energy correction is much smaller than the two-body energy correction $E^{(2)}$, the former is switched-off by default in QuantumATK.

BSSE and the counterpoise correction

In polyatomic systems, the so-called basis set superposition error (BSSE) arises due to the localised nature of the LCAO basis set. To understand the origin of the BSSE, consider a system formed by two subunits A and B. For simplicity, consider each subunit to be a single atom. When A and B are far apart, they are described only by the basis orbitals centered at their respective atomic nuclei. However, when A couples to B the basis functions centered at A overlap with those centered at B. As a result, part of the basis orbitals centered at B are available to describe A. This gives rise to an artificial attraction between A and B.

The BSSE can be minimized by using the counterpoise (CP) correction[Page 1659, 3]:

$$E = E_{AB} + E^{\mathrm{cp}},$$

where E_{AB} is the total energy of system AB and E is the counterpoise energy correction.

The counterpoise correction is obtained through:

$$E^{\mathrm{cp}} = (E_A - E_{A\tilde{B}}) + (E_B - E_{\tilde{A}B}),$$

where $E_{A\tilde{B}}$ is the energy of system A in the AB basis, which is obtained by taking the full system at its equilibrium geometry and defining the atoms of B as ghost atoms.

Note: Ghost atoms have no charge and no mass, but they have basis orbitals, defined by whichever element (and basis set) that defines the atom if it was not a ghost.

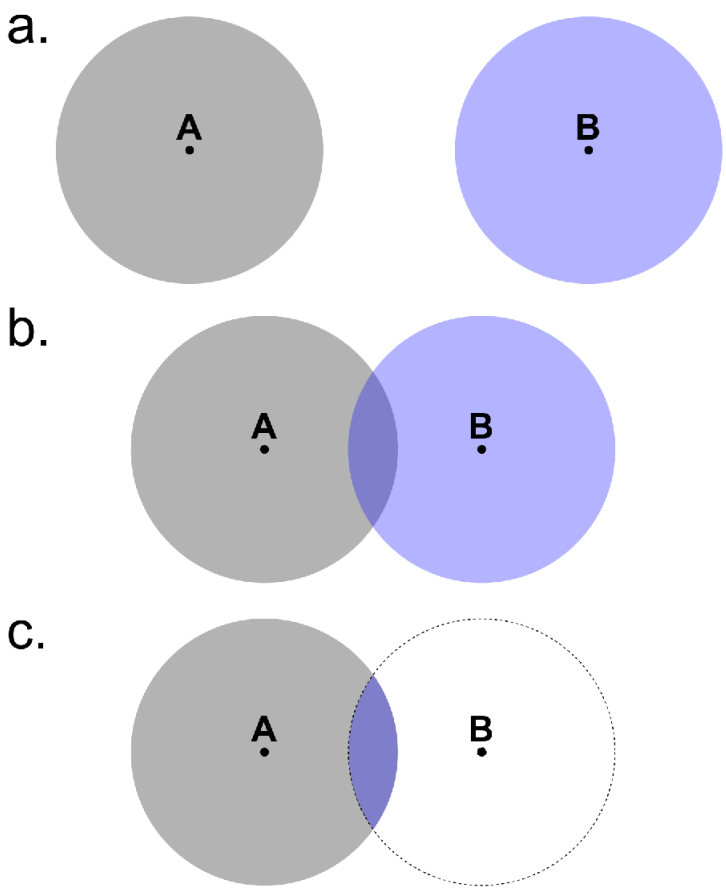


Fig. 3.115: Origin of the basis set superposition error. (a) A and B far apart from each other; (b) basis orbitals overlap when A couples to B; (c) the region in blue indicated the part of the basis orbitals centered at B that are available to describe A.

The counterpoise correction is readily extended to several subunits, A_i , through:

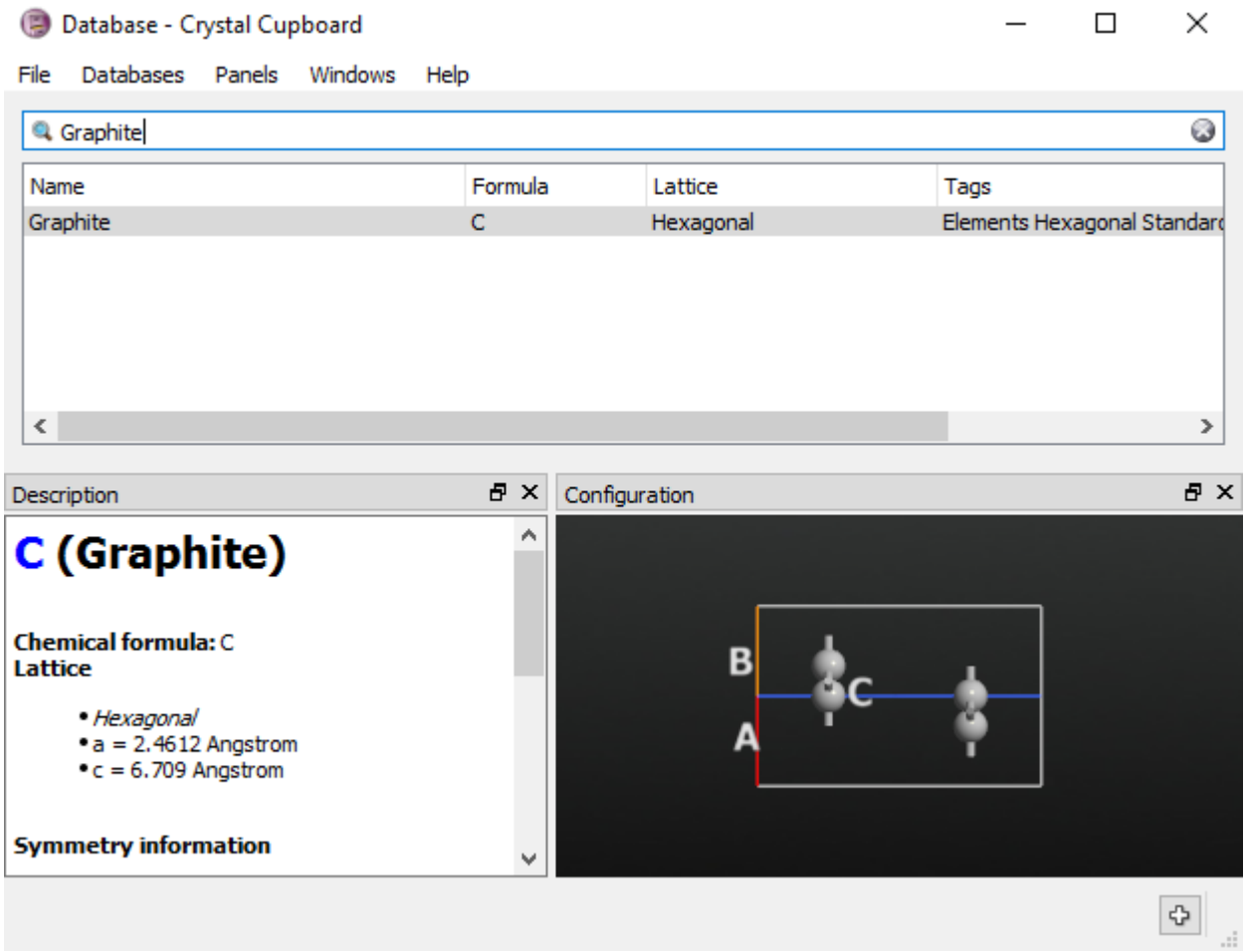
$$E^{\mathrm{cp}} = \sum_i (E_{A_i} - E_{A_i \tilde{C}_i}),$$

where C_i is the A_i complement, i.e. the atoms not in region A_i.

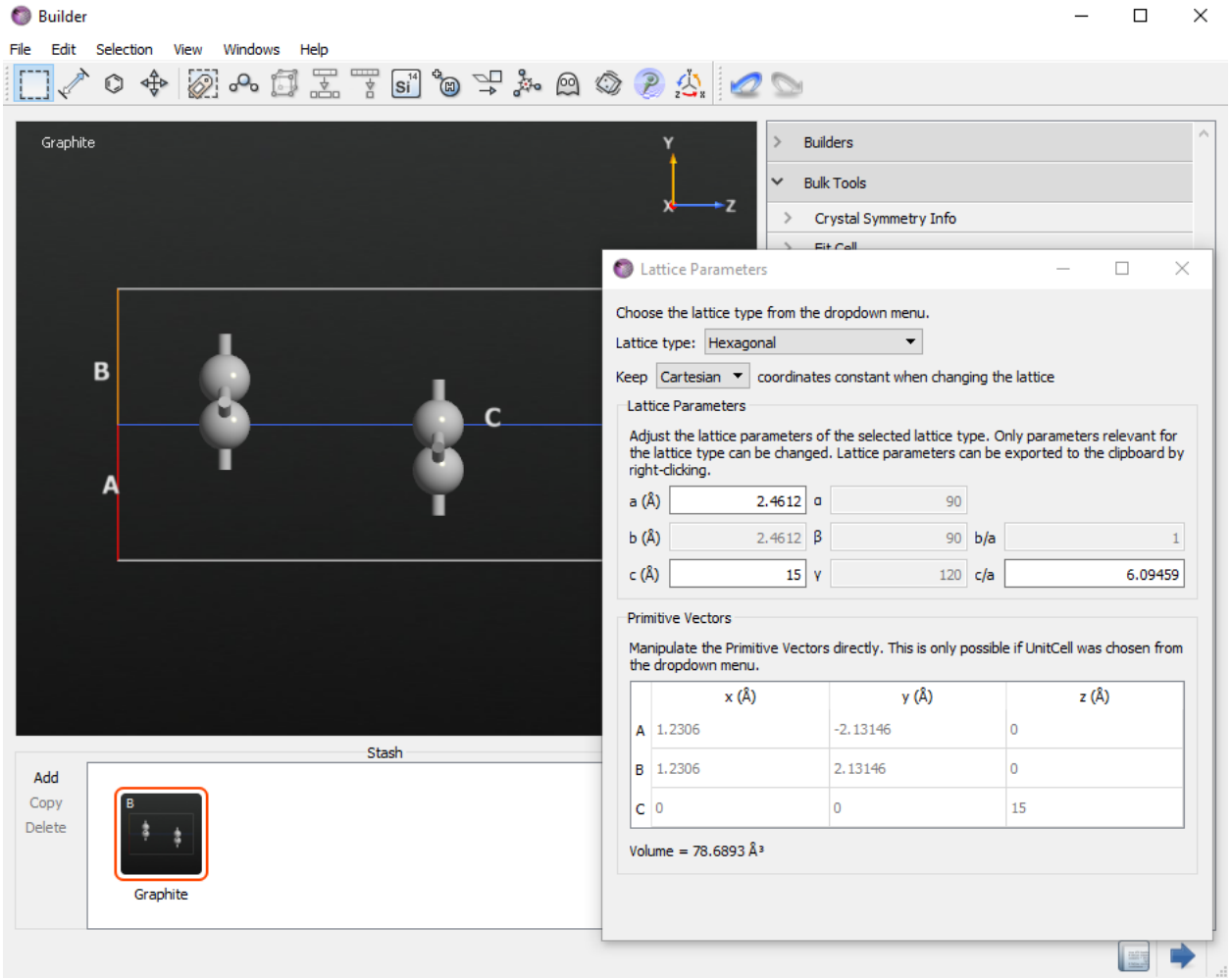
In the following section you will explicitly see the effects of these two corrections on the calculated equilibrium distance between two graphene layers.

Set-up the graphene bilayer system

In the **Builder**, click on *Add* → *From Database*, search for *Graphite* and add the structure to the stash.

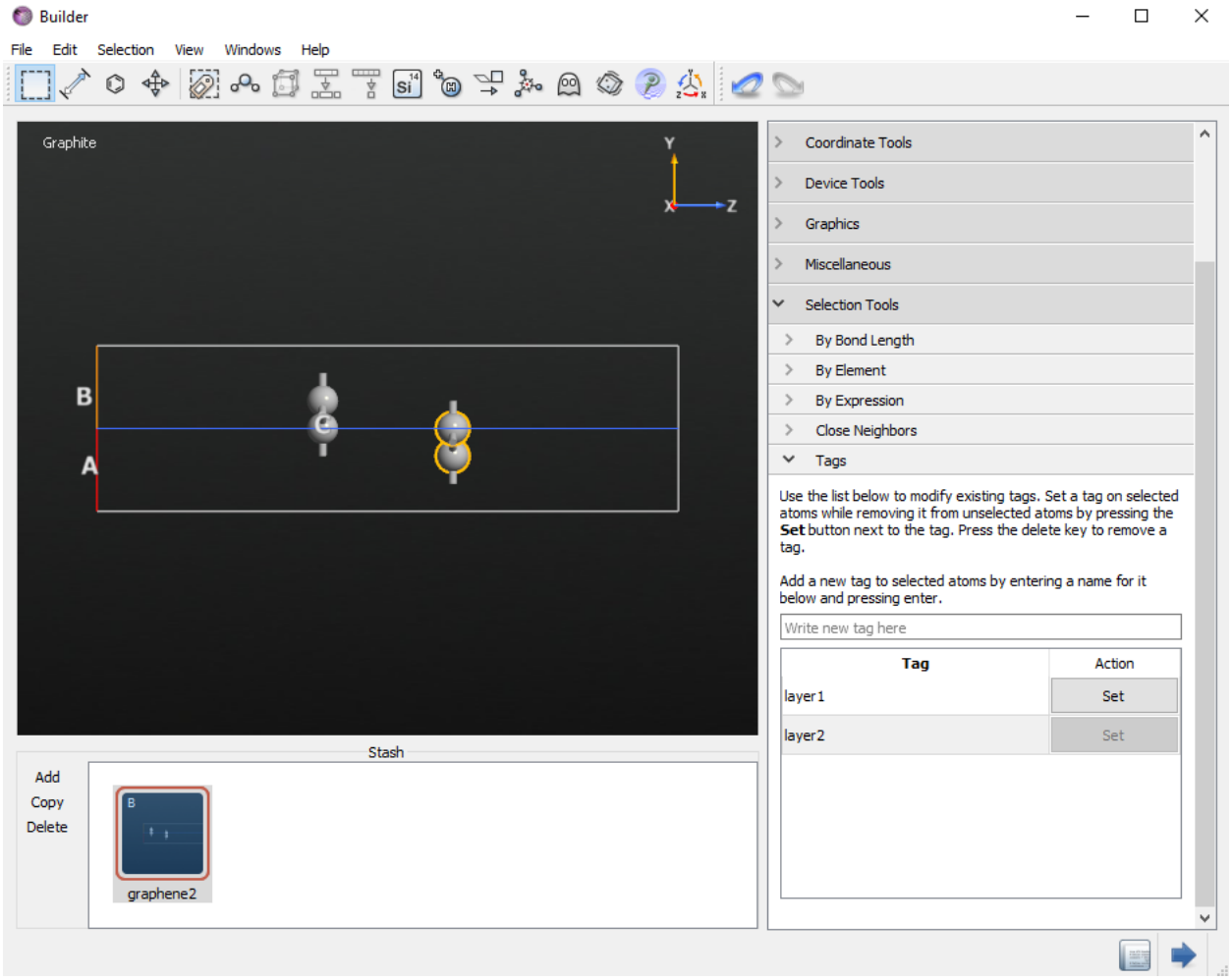


Rename the structure as *graphene2*, then open *Bulk Tools* → *Lattice parameters* and set the length of the C-vector to 15 Å. This transforms the graphite crystal into a bilayer graphene model.



Finally, center the geometry by using *Coordinate Tools* → *Center*.

To finish the setup of the structure you must add a tag to each graphene layer. Tags are used to distinguish the two layers, in order to define the counterpoise correction. In order to add the tags, select the two atoms in the first graphene layer, open *Selection Tools* → *Tags*, write *layer1* in the bottommost field and press `enter`. Then, repeat the same steps for the second layer and tag it as *layer2*.



Tip: You can check that the tags are correctly set using the Selection by Tag tool on the toolbar on the top of the **Builder**.

Geometry optimization without counterpoise correction

In the **Builder**, use the button to send the graphene bilayer structure to the **Script Generator**. Change the default output name to *graphene2.nc* and add the following blocks:

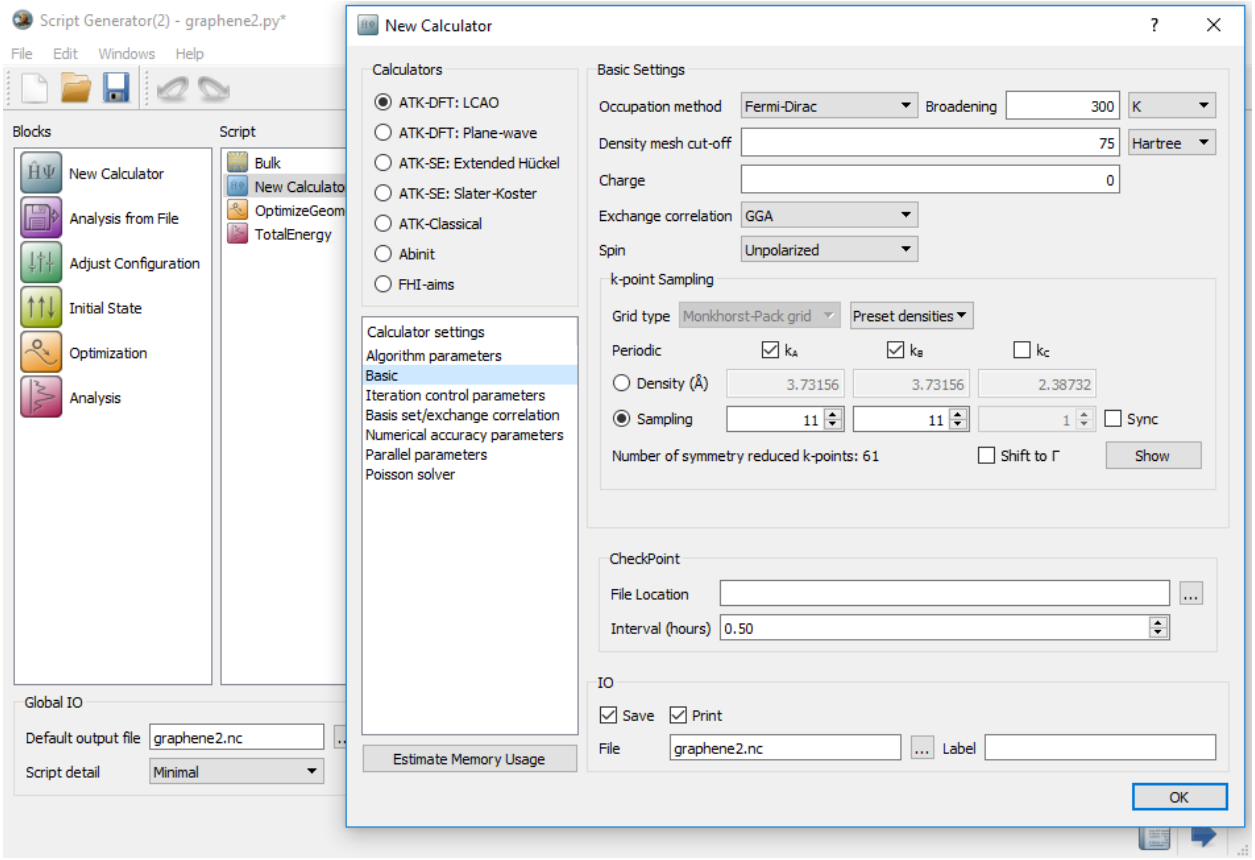
- *New Calculator*
- *Optimization → Optimize Geometry*
- *Analysis → Total Energy*

Click on the *New Calculator* block and modify the following parameters:

•Select the PBE functinal

•Set the k-point sampling to a grid of $11 \times 11 \times 1$ k-points.

Then, click on the *Optimize Geometry* block and set the force tolerance to 0.01 eV/Å.



To start the calculation, click the **Send To** icon located in the lower-right of the scripter and select **Job Manager** from the pop-up menu. **Save** the script as `graphene2.py` in the project directory.

The Job Manager will now pop up. Select the dropped script and press icon to start the job.

Analysis of the optimized structure

To inspect the relaxed configuration, expand graphene on the **LabFloor**. The file contains two configurations *gID000* and *gID001*. These are the input and relaxed structures, respectively.

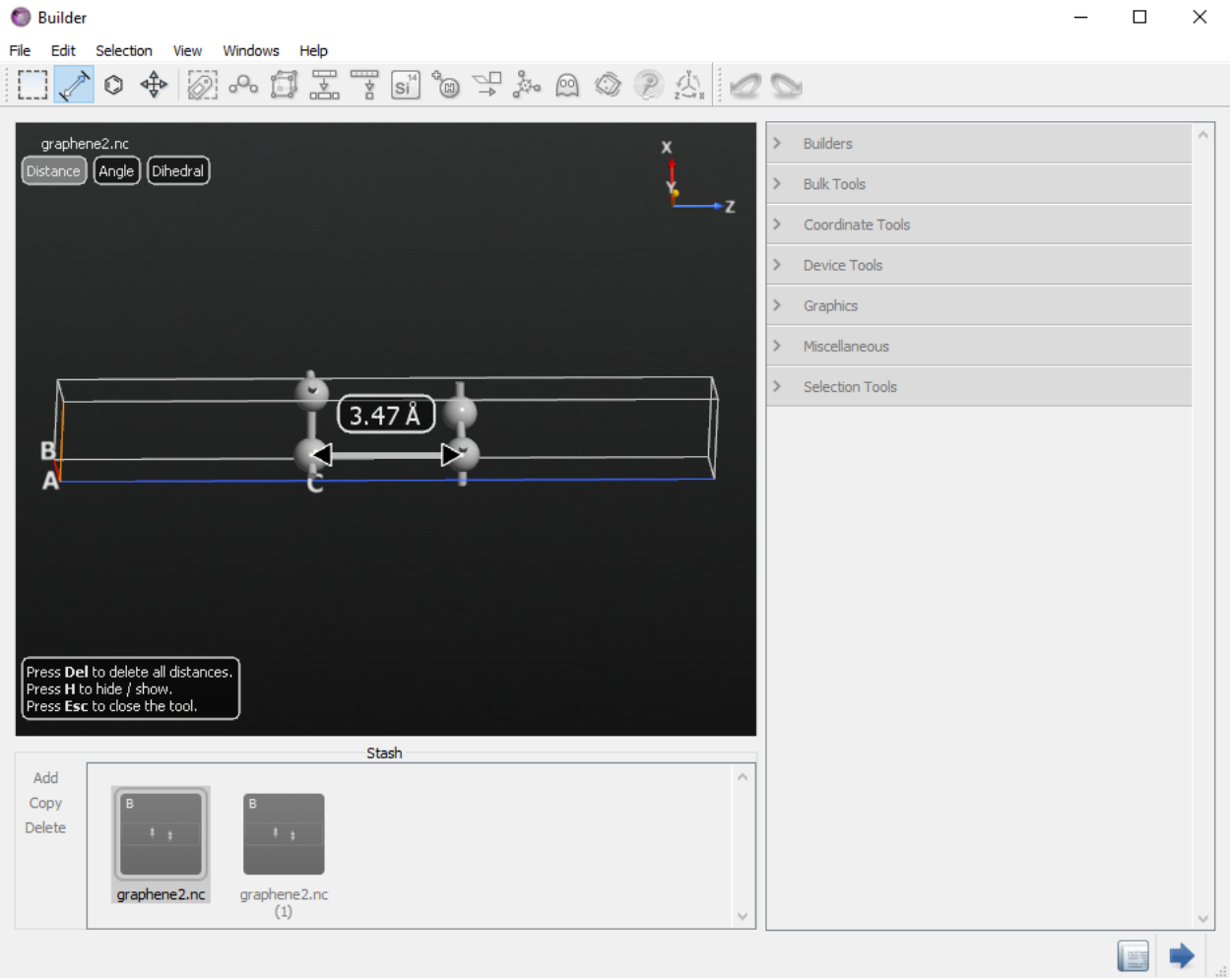
Drag and drop the relaxed structure (*gID001*) to the **Builder**.

In the **Builder**, click the measurements option for distance, and measure the distance between the two carbon atoms that sit on top of each other in the Z direction. The interatomic distance is 3.47 Å, which is fairly close to the experimental value 3.32 Å[7]. As you will soon see, this is however a result of a cancellation of two errors!

Note: Most literature experimental values for the graphene interlayer distance are valid for graphite but not for the bilayer graphene system discussed in this tutorial. Hence there can be small differences between the calculated results

[7] N.N. Greenwood and A.ăEarnshaw. Chemistry of the elements. *Pergamon*, 1984.

and experimental data.

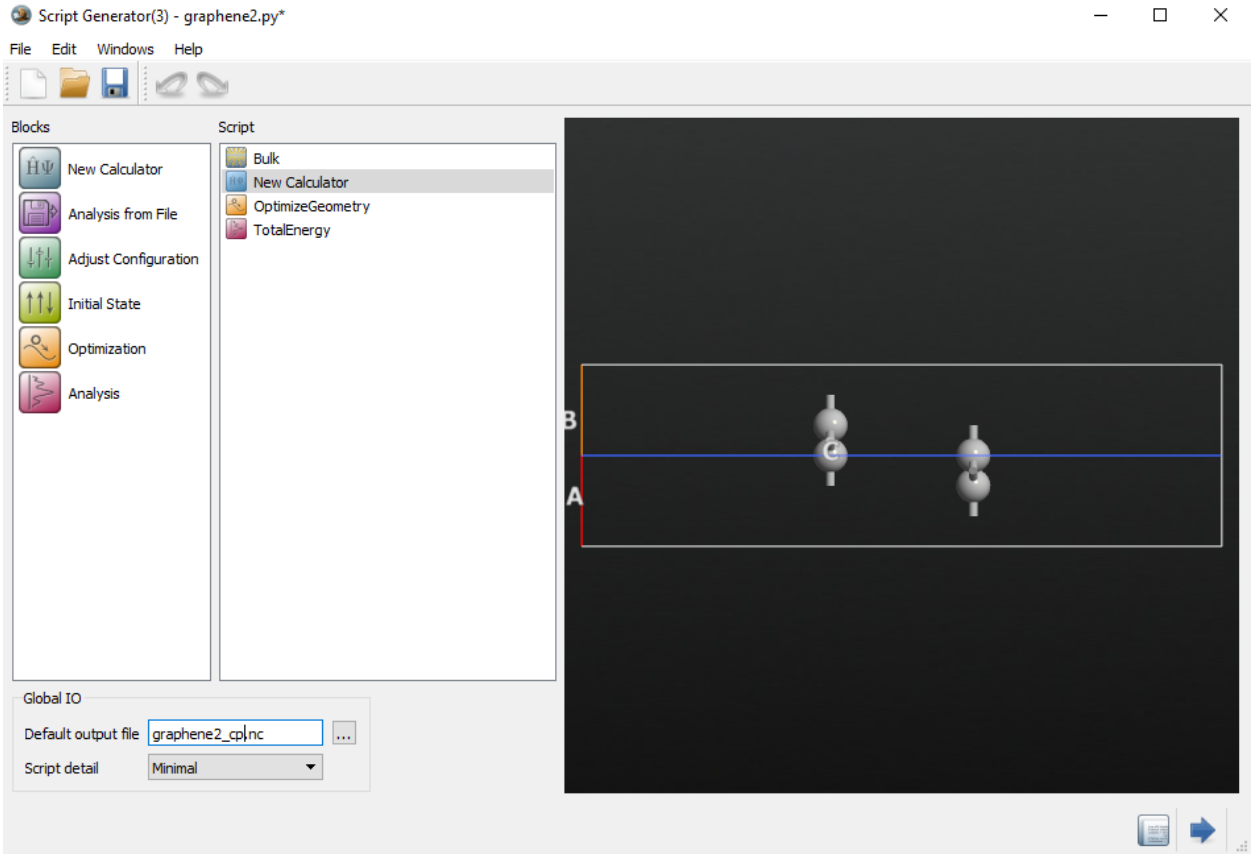


Including the counterpoise correction

You will now relax the geometry using the CP correction to correct for the basis set superposition error. To this end, open again the **Scripter** window where you set up the previous calculation.

Tip: All active tools are available through the Windows menu at the top of each tool.

Change the name of the output file to `graphene2_cp.nc`.



It is currently not possible to set up the CP correction in the Scripter. The CP correction must therefore be manually added by editing the QuantumATK script. Therefore, transfer the script to the **Editor** using the **Send To** icon.

In the **Editor**, change the definition of the Calculator as follows:

```
#calculator = LCAOCalculator(
#    basis_set=basis_set,
#    exchange_correlation=exchange_correlation,
#    numerical_accuracy_parameters=numerical_accuracy_parameters,
#    correction_extension=correction_extension,
#    )

#BSSE CP correction
bsse_calculator = counterpoiseCorrected(LCAOCalculator, ["layer1", "layer2"])
calculator = bsse_calculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    correction_extension=correction_extension,
    )
```

Note how the system is separated into two parts (*layer1* and *layer2*) using the tags. The system can be divided into an arbitrary number of parts, and the counterpoise correction is applied to each couple of parts. The only requirement is that the parts are adjoint and sum up to the entire system.

Save the script as `graphene2_cp.py`, send it to the **Job Manager** and start the calculation. The run time will be

2–5 times longer than before, since now at each geometry optimization step, 5 different systems need to be calculated, *viz.* $AB, A, A\tilde{B}, B, \tilde{A}B$.

When the calculation is done, inspect the interlayer distance and you will find it to be 4.23 Å. This is considerably larger than the previous result, and far from the experimental value (3.32 Å). As mentioned in the introduction, the BSSE creates an artificial attraction between the graphene layers, which reduces the interlayer distance. The counterpoise correction eliminates this error, but you are now left with the problem that the layers are essentially unbound, due to the omission of van der Waals forces.

Including the D2 dispersion correction

Since the interlayer distance calculated with the CP correction is too large, there must be a missing attractive contribution to the total force. This is the van der Waals (dispersion) force which is not correctly described with the PBE functional. The vdW contribution to the total forces can be approximately added using the semi-empirical DFT-D2 approach.

To add the D2 correction, drag and drop the `graphene2_cp.nc` file to the **Scripter** :

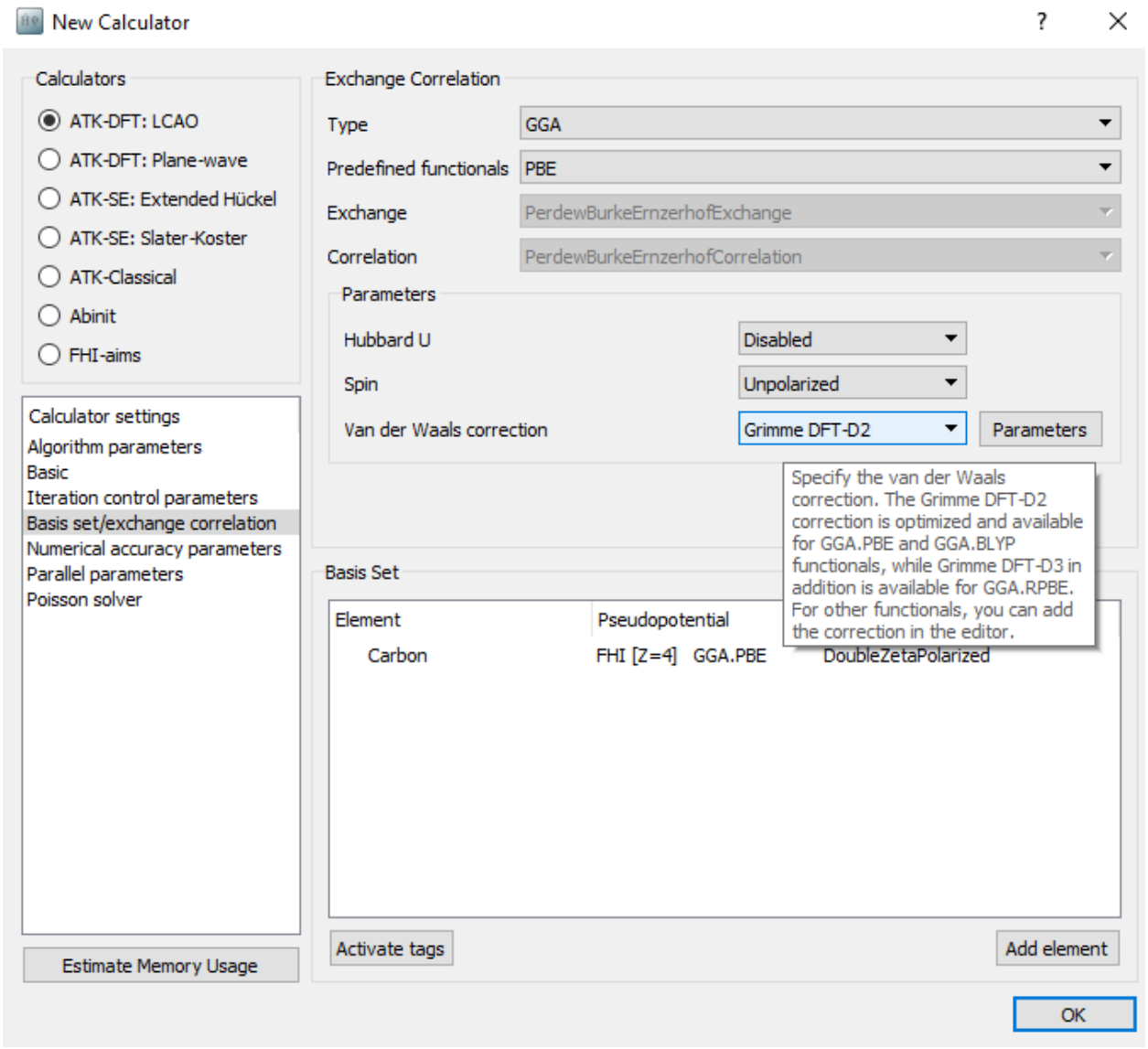
•Change the output file to `graphene2_cp_d2.nc`.

•Double-click **Optimization** and select **OptimizeGeometry**.

•Double-click **Analysis** and select **TotalEnergy**.

Now double-click the **New Calculator** block in the Script panel to open the calculator widget.

Under Calculator setting and Basis set/Exchange correlation select the Grimme DFT-D2 van der Waals correction. Note that you can modify all the specific parameters by clicking the *Parameters* button.



Double-click on the **OptimizeGeometry** block and set the force tolerance to 0.01 eV/Å.

Send the script to the **Editor** and edit the python script in order to add the counterpoise correction as you did before.

```
#-----------------------------------------
# Grimme DFTD2
#-----------------------------------------
correction_extension = GrimmeDFTD2(
    global_scale_factor=0.75,
    damping_factor=20.0,
    maximum_neighbour_distance=30.0*Ang,
    element_parameters={
        Carbon: [1.452*Ang, 1.75*J*nm**6/mol],
        },
```

(continues on next page)

(continued from previous page)

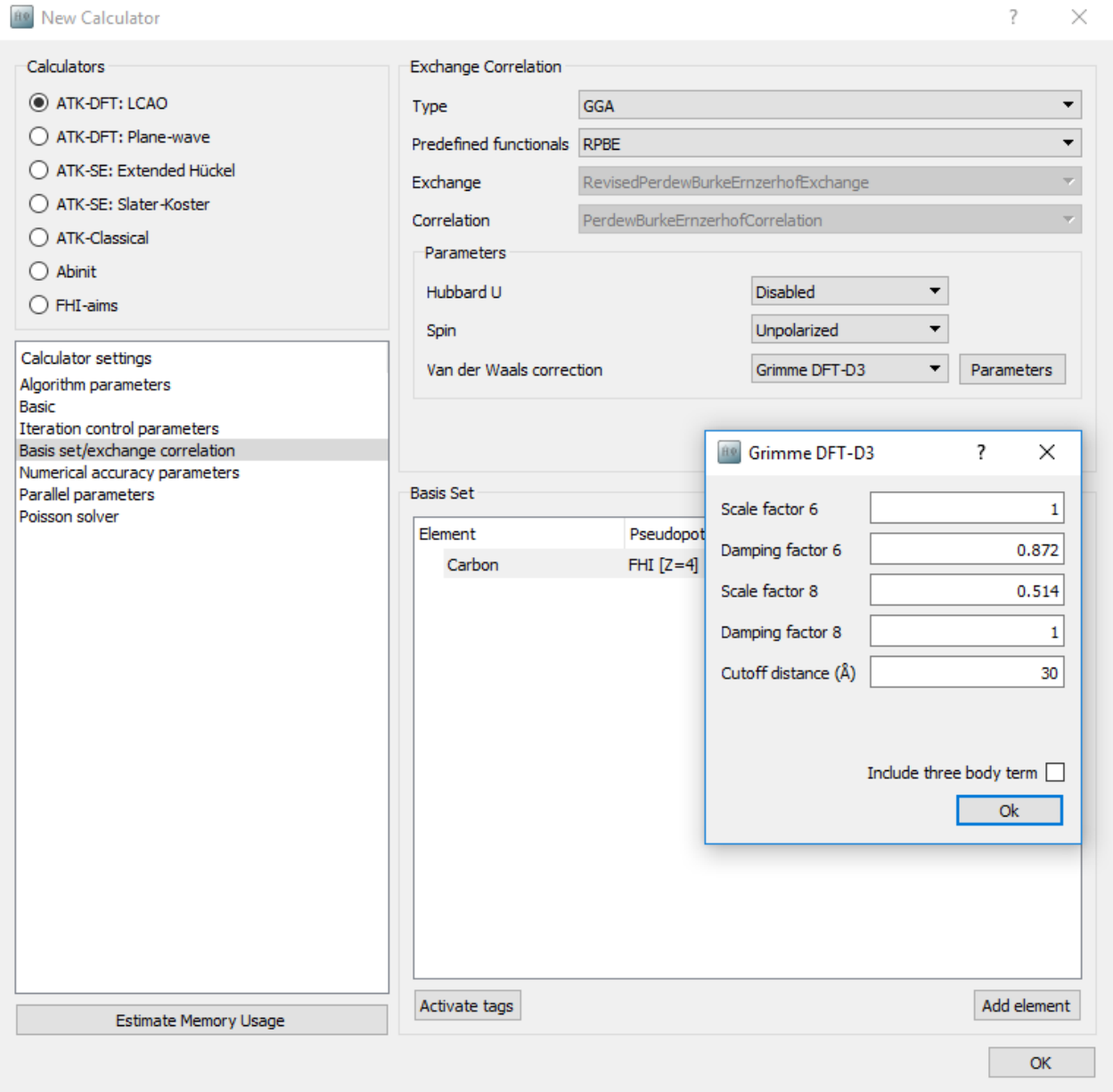
```
    )

#BSSE CP correction
bsse_calculator = counterpoiseCorrected(LCAOCalculator, ["layer1", "layer2"])
calculator = bsse_calculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    correction_extension=correction_extension,
    )
```

Press the **save** button and save the script as `graphene2_cp_d2.py` in the project directory.

Run the script and when the calculation is done, inspect the interlayer distance. You will now find it to be 3.31 Å. Thus, including the dispersion correction and accounting for BSSE, an interlayer distance close to the experimental result can be obtained.

Including the D3 dispersion correction

The D3 correction can be applied in a similar way as shown for the D2 correction. However, as discussed in[Page 1659, 2] in this case the best results for the interlayer distance and binding energy are obtained using a revised PBE-type functional (revPBE) instead of the original PBE. Therefore, when adding the D3 correction, one also need to change the exchange-correlation functional from PBE to revPBE as shown below (RPBE in the list of predefined functionals).



As discussed in the method section the D3 correction can be supplemented with a three-body energy term $E^{(3)}$ for an even more accurate evaluation of the vdW forces. However, evaluation of this term is computationally demanding, and the $E^{(3)}$ term is therefore not computed by default. If one wants to used DFT-D3 including three-body interactions, we suggest to first optimize the structure using DFT-D3 without the $E^{(3)}$ term, and then perfom a single-point calculation on the optimized geometry including the $E^{(3)}$ term.

Summary of the results

The result of the calculations are summarized in the following table.

Table 3.9: Results for the distance, d, and binding energy per atom, E_B, of bilayer graphene.

	E_B (meV/atom)	d (Å)
PBE	-15	3.47
PBE+CP	-0.6	4.23
PBE-D2+CP	-22.9	3.31
revPBE-D3+CP	-22.6	3.35
revPBE-D3+CP+3body	-18.6	3.36

The binding energy ΔE of the bilayer system was calculated as:

$$\Delta E = E_{\mathrm{tot}}^{\mathrm{bilayer}} - 2E_{\mathrm{tot}}^{\mathrm{monolayer}}$$

where $E_{\mathrm{tot}}^{\mathrm{bilayer}}$ and $E_{\mathrm{tot}}^{\mathrm{monolayer}}$ are the total energies of bilayer and monolayer graphene respectively.

Note: The calculation of the binding energies must be consistent in terms of exchange-correlation functional and dispersion correction. For example, to obtain the PBE-D2+CP result, the single layer graphene reference calculation must be performed using DFT-D2.

References

3.12 Miscellaneous

3.12.1 The DFTB model in ATK-SE

The purpose of this tutorial is to show you how to install new parameter sets into the Density Functional based Tight Binding (DFTB) model[1],[2] implemented in **ATK-SE**.

ATK is shipped with Slater-Koster parameter files from the CP2K and Hotbit consortia, for use in the semi-empirical DFTB model. This tutorial will show how to download additional parameters from the DFTB website and install them such that they work as a native part of **ATK**.

Once the parameters are installed, they can be used for modeling molecules, bulk and device systems. In this tutorial, you will first download and install a DFTB parameter set, and then learn how to calculate properties of spin polarized bulk systems using a Slater-Koster calculator.

Note: You will primarily use the graphical user interface **QuantumATK** for setting up and analyzing the results. If you are not familiar with **QuantumATK**, we recommend you first go through a few of the **QuantumATK Tutorials**.

---

[1] M.Elstner, D. Porezag, G. Jungnickel, J. Elsner, M. Haug, T. Frauenheim, S. Suhai, and G. Seifert, Phys. Rev. B **58**, 07260 (1998)

[2] C.Kohler, T. Frauenheim, B. Hourahine, G. Seifert, and M. Sternberg, J. Phys. Chem. A **111**, 5622 (2007)

The underlying calculation engine for this tutorial is **ATK-SE**, the semi-empirical part of **QuantumATK**. A complete description of all the parameters, and in many cases a longer discussion of their physical relevance, can be found in the ATK Reference Manual.

In order to run this tutorial, and in general use the semi-empirical models in **ATK**, you must have a **license for ATK-SE**. If you do not have one, you may obtain a time-limited demo license by contacting us via our website: Contact QuantumATK.

Installing DFTB parameters

Getting a DFTB license

The dftb.org website has a number of parameters for the DFTB method, which can be used with **ATK-SE**. In order to download the parameters, you will need to fill out their registration form. When your registration has been processed, you will receive a username and password to their website.

Downloading the parameters

On the DFTB download page, you can get an overview of the different DFTB parameter sets. The primary ones are named mio, pbc, and matsci; the mio set can further be extended by a number of specialized sets listed further below. The page also lists the papers that must be referenced when using the parameter sets.

Log into dftb.org and download the parameter sets of your interest. The downloaded files will be in a compressed tar ball (`.tar.gz`).

Installing the parameters

ATK has a special folder for each of the three DFTB parameters sets. For instance, to install the mio parameter set, unpack the downloaded tar ball and copy the `.skf` files to the directory `share/tightbinding/dftb/mio/` in the directory where **ATK** is installed. In Linux you can use the following commands:

```
tar zxvf mio-1-1.tar.gz
cp mio-1-1/* [ATKPATH]/share/tightbinding/dftb/mio
```

where `[ATKPATH]` is the path to your installation of QuantumATK. Similarly, the pbc and matsci parameter files should be copied to the `dftb/pbc` and `dftb/matsci` directories. Note that these directories already exist, but are empty.

> **Warning:** If you use **QuantumATK** on a laptop or workstation to set up the calculations, and then a separate installation of **ATK** on a cluster for running the calculations, then the parameter files must be installed on both systems separately.

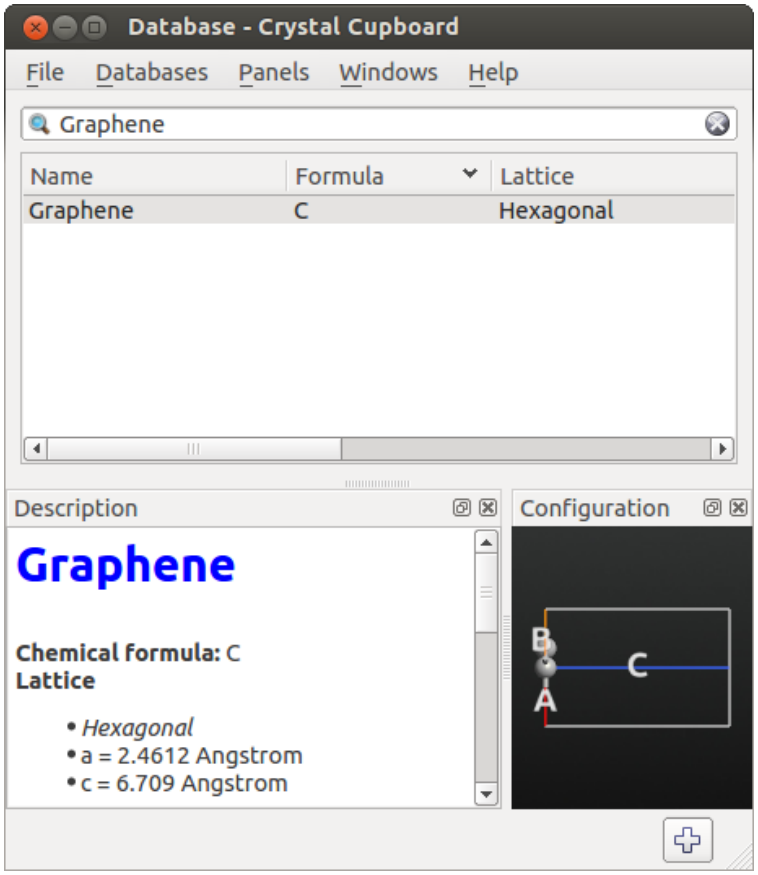
Testing the installation

Bandstructure calculation

To test that the parameters are correctly installed you, can perform the following bandstructure calculation for graphene.

•Start **QuantumATK** and create a new project, then click **Open**.

•Launch the **Builder** and click *Add* → *From Database*. Find Graphene.



•Add the structure to the **Stash** (double-click or use the icon).

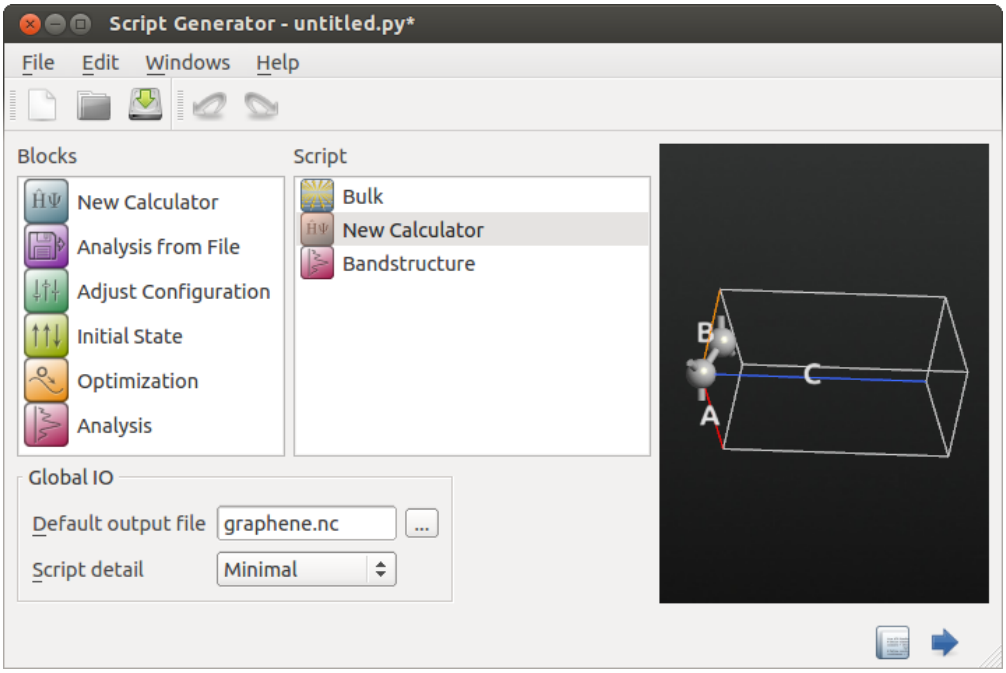
•Send the structure to the **Script Generator** using the button.

•In the **Script Generator**, add the following blocks:

1. **New Calculator**
2. **Bandstructure**

•Change the output filename to `graphene.nc`.

The **Script Generator** should now have the following settings:

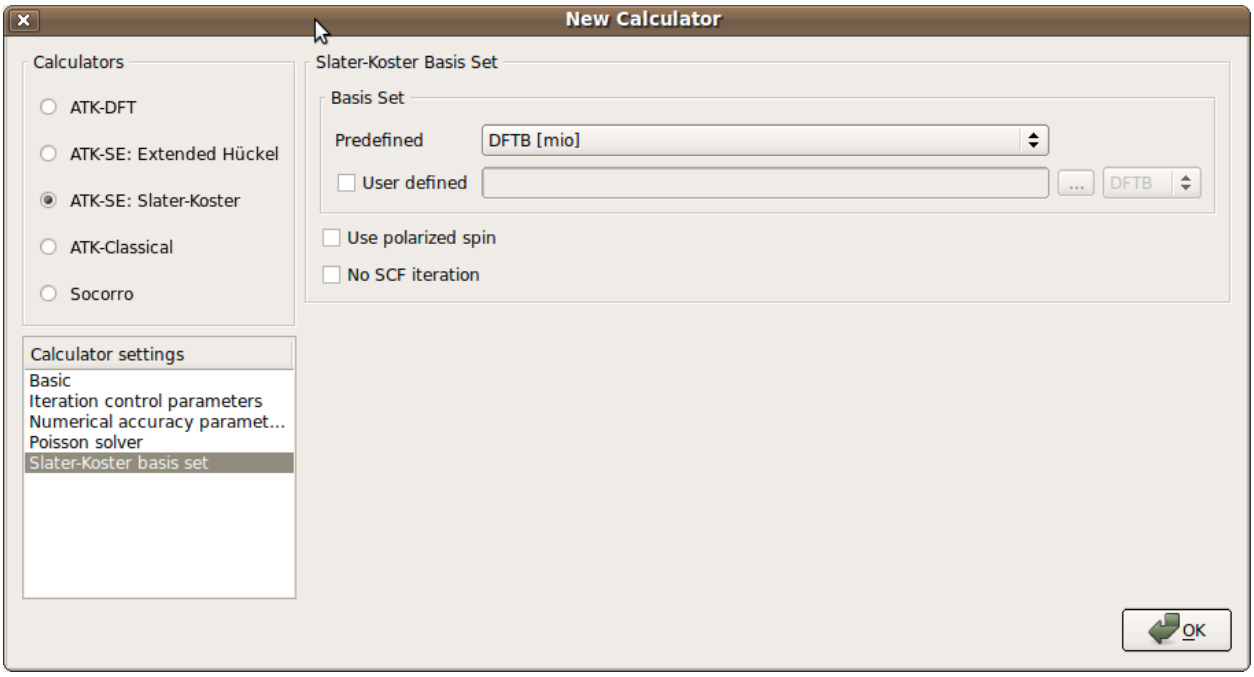


Now open the **New Calculator** block and make the following changes:

- Select the **ATK-SE: Slater-Koster** calculator.
- Change the **k-point sampling** to (5, 5, 1).
- Go to the **Slater-Koster basis set**, and check that the installed basis sets are in the basis set list. Select the DFTB [mio] basis set.

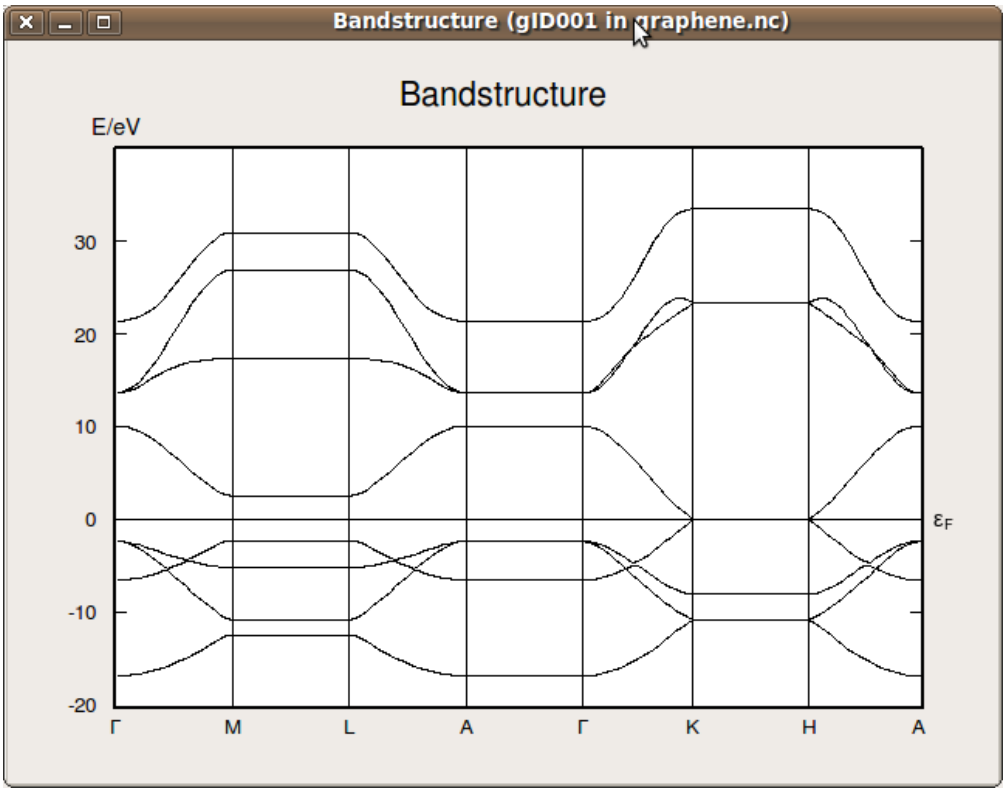
Note: If you do not see the DFTB [mio] basis set in the list, something went wrong in the installation process. Check that the file `atkpython/share/tightbinding/dftb/mio/C-C.skf` exists in your **ATK** installation directory.

- Uncheck the **No SCF iteration** box, in order to invoke a self-consistent DFTB calculation.



Attention: The default assumption is that a tight-binding model is non-self-consistent. It is the responsibility of the user to uncheck the **No SCF iteration** box for models which are to be self-consistent. Most DFTB models are self-consistent.

- Transfer the script to the **Job Manager** using the **Send To** button ( ), and start the calculation.
- When the job has finished, locate the file `graphene.nc` under project files in the main **QuantumATK** window, and plot the bandstructure using the *Bandstructure Analyzer* from the right-hand tool bar. You should get the result shown below:



Note: If the test went well, the DFTB parameters are properly installed. You are then able to use DFTB parameters instead of DFT for some of the other **ATK** tutorials available on the QuantumATK website (including quantum transport calculations), in order to save computational time.

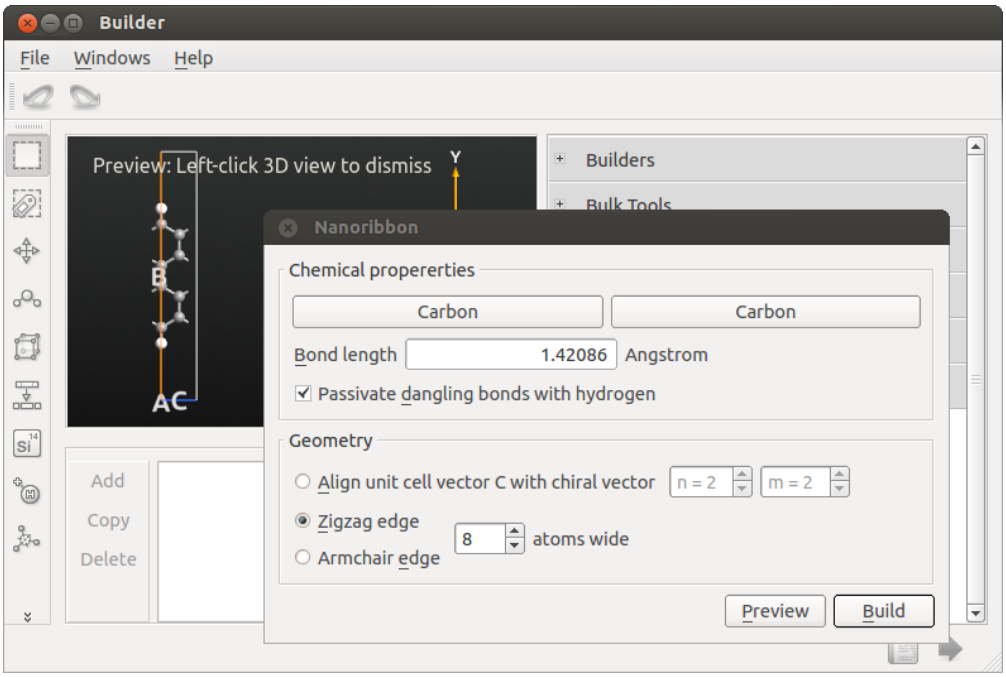
Obviously, this only works if the elements used are covered by the DFTB basis sets.

Spin polarized calculations with DFTB

Calculating the spin polarization of a graphene ribbon

In the following you will set up a graphene nanoribbon, and perform a spin-polarized calculation using the DFTB [mio] model.

- Open the Builder and click *Add* → *From Plugin* → *Nanoribbon*.
- Build a zigzag ribbon that is 8 atoms wide.



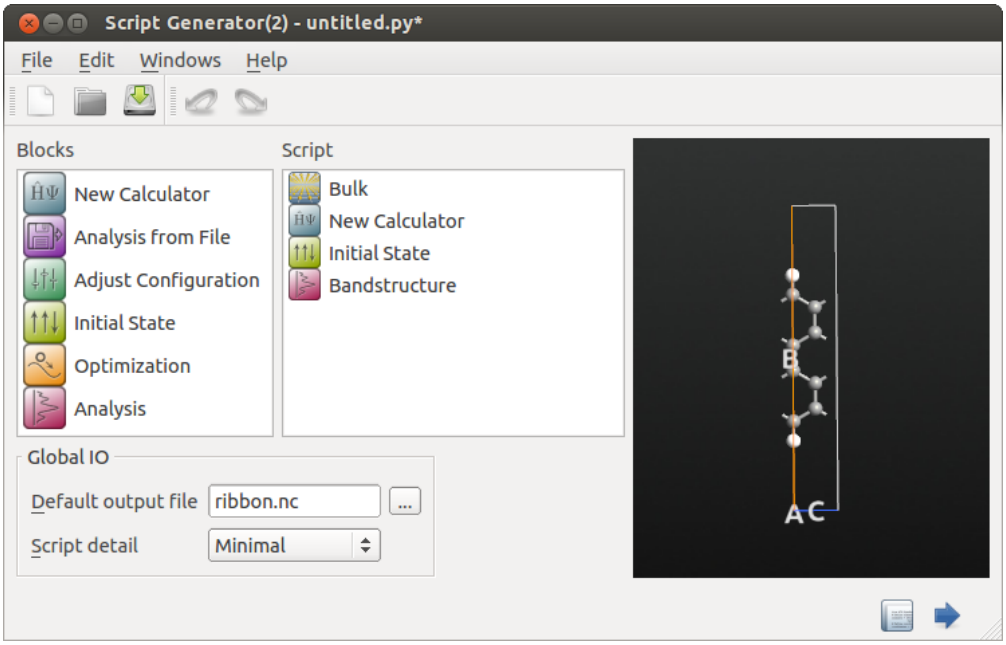
To set up a calculation of the band structure, send the structure to the **Script Generator** using the button in the lower right-hand corner of the builder window.

In the **Script Generator**, add the following blocks:

- **New Calculator**
- **Initial State**
- **Bandstructure**

and change the output filename to `ribbon.nc`.

The **Script Generator** should now have the following settings:



Open the **New Calculator** block.

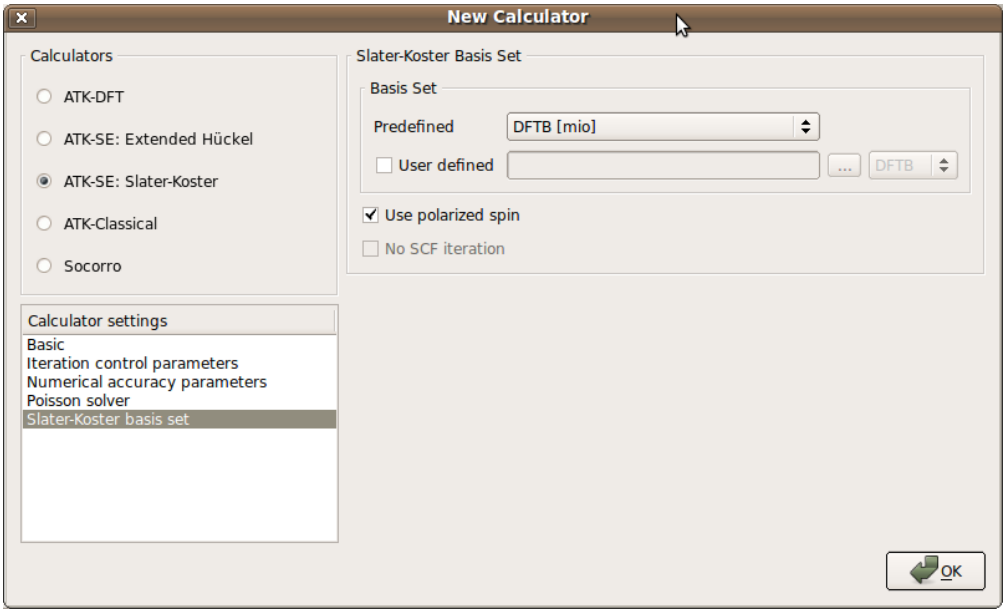
•Select the **ATK-SE: Slater-Koster** calculator.

•Change the **k-point sampling** to (1,1,11).

•Go to the **Slater-Koster basis set** and select the DFTB [mio] basis set.

•Check the **Use polarized spin** box, in order to perform a spin-polarized DFTB calculation.

Note: The DFTB parameter set does not include spin polarization parameters. Instead, spin polarization parameters are obtained from the ATK_W database.



Finish by clicking **OK**.

Next, set up the initial spin state of the system:

•Open the **Initial State** block by double-clicking it.

•Set the **Initial state type** to *User spin*.

•Under Spin, set the default spin of both Carbon and Hydrogen to 0.

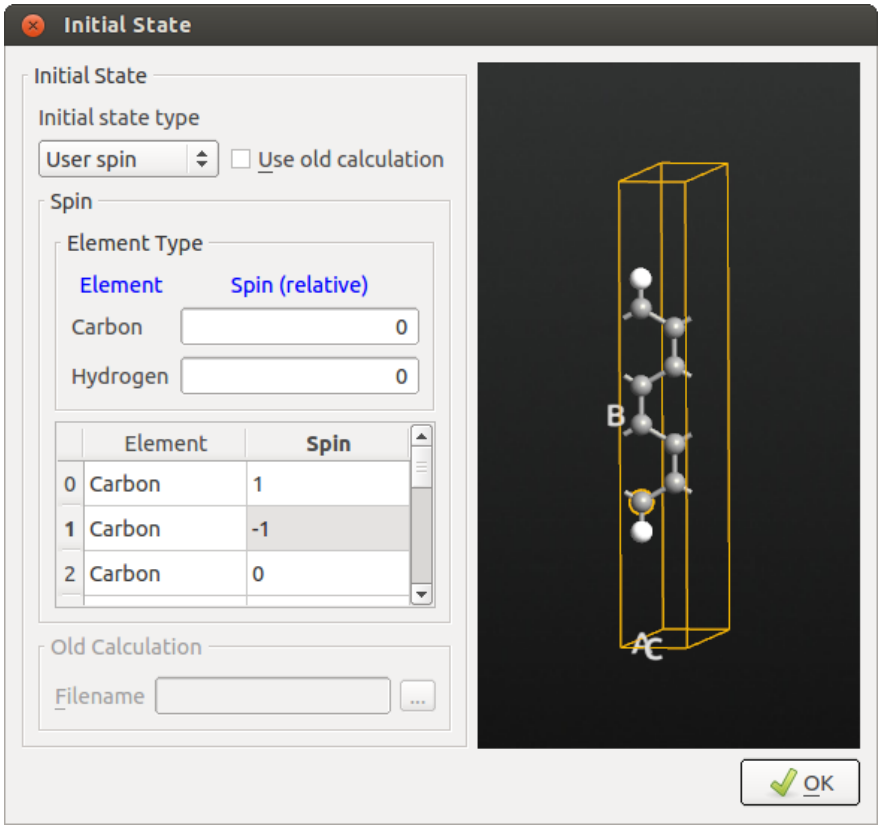
•In the 3D view:

–Click the upper carbon atom (index 0 in the list of atoms) and set its **initial relative spin** to 1.0.

–Repeat this procedure for the lower carbon atom (number 1 in the list), setting its **initial relative spins** to -1.0. (Dont be confused by the fact that the carbon atoms are not ordered by Y coordinate).

–Leave the hydrogen atoms and the other carbon atoms with **initial relative spin** values of 0.0.

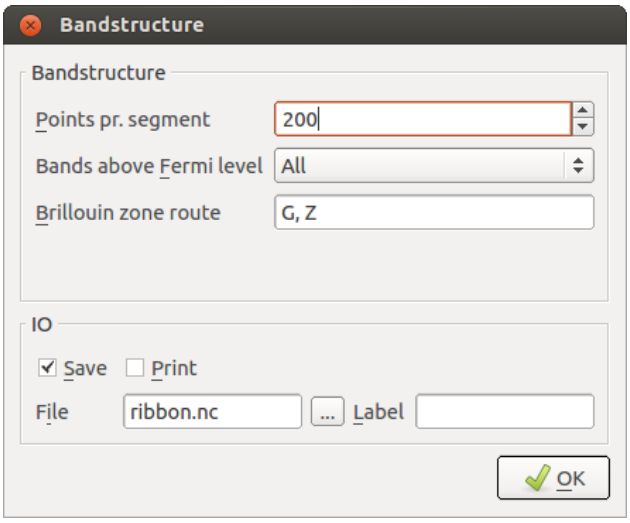
The dialogue should now look as shown below.



Finally, open the **Bandstructure** block:

•Change **Points pr. segment** to 200. This implies that each band will be calculated with a resolution of 200 points.

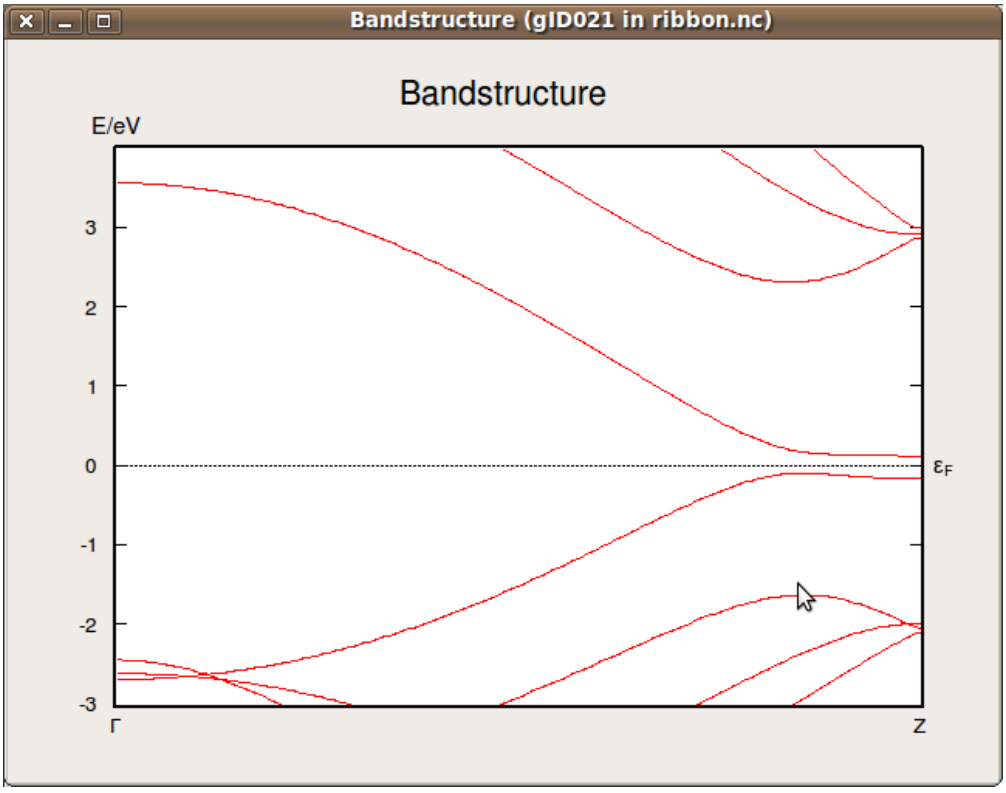
•Also notice that the default suggested route in the Brillouin zone (from the Γ-point G=(0,0,0) to Z=(0,0,1/2) in units of the reciprocal lattice vectors) is the appropriate one.



•Send the script to the **Job Manager** and run it.

•Return to the main **QuantumATK** window and select the file `ribbon.nc` and plot the bandstructure.

•By using the zoom functionality, you should be able to obtain the following figure:



Calculate the properties of a series of configurations using scripting

If you want to calculate the bandstructure of a number of nanoribbons, it is very convenient to use Python scripting. The script below performs a calculation of the bandstructure of nanoribbons with chiral index (n,n), where n runs from 1 to 10.

Attention: The *chiral index* notation for nanoribbons in **ATK** is based on the linear combination of the planar graphene unit vectors that are used to form the unit cell. Thus, (n,n) have zigzag edges, and actually correspond to an unrolled armchair nanotube with the same chiral indices.

For armchair edge nanoribbons, the chiral indices are (n,0).

```
# Setup the DFTB Calculator
basis_set = DFTBDirectory("dftb/mio/")
pair_potentials = DFTBDirectory("dftb/mio/")

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(1, 1, 51),
    )

iteration_control_parameters = IterationControlParameters()

calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    pair_potentials=pair_potentials,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    spin_polarization=True,
    )

#loop over different NanoRibbons
for n in range(1,10):
    #generate the nanoribbon
    bulk_configuration = NanoRibbon(n,n)

    #Determine the initial spin of the configuration
    elements = bulk_configuration.elements()
    coords = bulk_configuration.fractionalCoordinates()
    scaled_spins = numpy.zeros(len(elements))

    #find the index of the two edge Carbon atoms,
    ymin = 0.5
    ymax = 0.5
    imin=0
    imax=0
    for i in range(len(elements)):
        if coords[i][1] < ymin and elements[i] == Carbon:
            ymin = coords[i][1]
            imin = i
        if coords[i][1] > ymax and elements[i] == Carbon:
            ymax = coords[i][1]
            imax = i
```

(continues on next page)

(continued from previous page)

```
    # set opposite spins on the edge Carbon atoms
    scaled_spins[imin] = 1
    scaled_spins[imax] = -1

    #attach calculator, and set the initial spin
    bulk_configuration.setCalculator(
        calculator(),
        initial_spin=InitialSpin(scaled_spins=scaled_spins),
        )
    bulk_configuration.update()

    # Calculate Bandstructure
    bandstructure = Bandstructure(
        configuration=bulk_configuration,
        route=['G', 'Z'],
        points_per_segment=200,
        bands_above_fermi_level=All
    )
    nlsave('bandgap.nc',bandstructure)
```

•Download the script to the project folder (`bandgap.py`) and send it to the **Job Manager** by drag-and-drop in the **QuantumATK** window.

•This will generate the file `bandgap.nc`.

•To analyze the data, download (`analyze_bandgap.py`) and drop the following script on the **Job Manager**.

```
w_list = []
gap0_list = []
gap1_list = []
# read a list with the bandstructure data
bandstructure = nlread('bandgap.hdf5', Bandstructure)

for i in range(len(bandstructure)):
    # get all the bandlines
    energies = bandstructure[i].evaluate().inUnitsOf(eV)
    #calculate the number of occupied bands
    occupied_bands = numpy.sum(numpy.array(energies[0]) < 0)
    #calculate the minimum direct band gap
    gap0 = numpy.min(energies[:,occupied_bands]- energies[:,occupied_bands-1])
    #calculate the band gap at the zone boundary
    gap1 = energies[-1,occupied_bands]- energies[-1,occupied_bands-1]
    #determine the coordinates in the y direction
    y_values = NanoRibbon(i+1,i+1).cartesianCoordinates().inUnitsOf(Ang)[:,1]
    # calculate the width, as distance between furthest carbon atoms
    # i.e. calculate the  H-H distance and subtracting the C-H distance
    w = numpy.max(y_values)-numpy.min(y_values)-2.2
    #append calculated values to the list
    w_list.append(w)
    gap0_list.append(gap0)
```

(continues on next page)

(continued from previous page)

```
    gap1_list.append(gap1)

#print the data
import pylab
pylab.figure()
pylab.plot(w_list,gap0_list, 'k-o')
pylab.plot(w_list,gap1_list, 'k-o', markerfacecolor='white')
pylab.xlabel(r"$w_z$ ($\AA$)", size=16)
pylab.ylabel("$\Delta_z$ (eV)", size=16)
pylab.show()
```

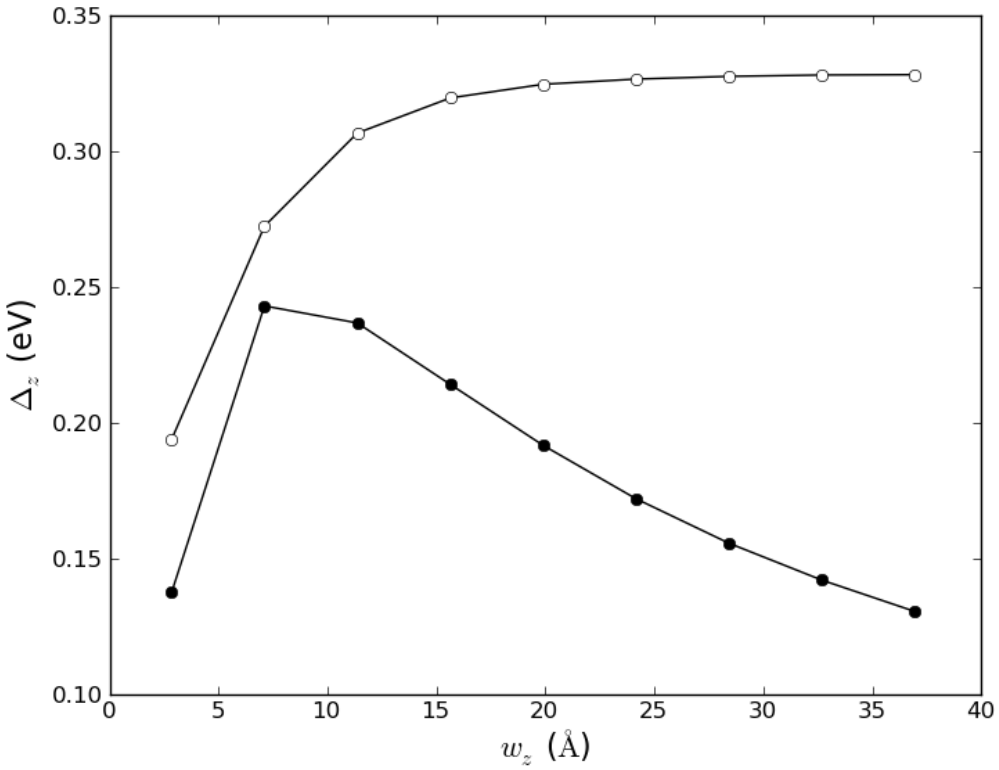


Fig. 3.116: Direct band gap (solid circles) and the band gap at the zone boundary as function of the width of the nanoribbon.

Attention: Except for the smallest ribbon, the plot shows the same qualitative behavior as Fig. 4c in Ref.[3]. The discrepancy in the results are related to the accuracy of the DFTB method compared with a DFT-LDA calculation. To test this, you can modify the script and change the definition of the calculator:

```
calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    exchange_correlation = LSDA.PZ
    )
```

Remember to also modify the name of the output file in order to separate the new data from the old data. Then execute the script.

References

3.12.2 Accessing QuantumATK internal variables

Version: 2016.0

The purpose of this tutorial is to illustrate how to extract internal quantities from QuantumATK. In the following, the quantities are briefly described and several sections then illustrate with examples how the information can be used to compute different sorts of transport coefficients.

Internal matrices accessible in QuantumATK

The table below lists some of the internal matrices in QuantumATK, e.g. the Hamiltonian H and the overlap matrix S. The commands listed in the left-hand column in the table allow users to extract these matrices through the so-called low level entities module in QuantumATK.

Table 3.10: Internal matrices accessible in QuantumATK. Further description of the variables can be found in the ATK Reference Manual.

Command	Symbol
`orbitalInfo()`	$[(i,l,m),\cdots]$
`calculateHamiltonianAndOverlap()`	$H(\mathbf{k}), S(\mathbf{k})$
`calculateDensityMatrix()`	$D(\mathbf{k})$
`calculateSelfEnergy()`	$\Sigma^{L/R}(E,\mathbf{k})$
`calculateRetardedGreenFunction()`	$G^{r}(E,\mathbf{k})$
`calculateGreenFunctionComponent()`	$G^{L/R}(E,\mathbf{k})$
`calculateDynamicalMatrixAndOverlap()`	$C(\mathbf{k}), M(\mathbf{k})$
`calculatePhononSelfEnergy()`	$\Pi^{L/R}(E,\mathbf{k})$
`calculatePhononRetardedGreenFunction()`	$D^{r}(E,\mathbf{k})$
`calculatePhononGreenFunctionComponent()`	$D^{L/R}(E,\mathbf{k})$

You will in this tutorial see examples of how to extract these matrices and use them to compute transport coefficients. The underlying calculation engine to be used here is ATK-SemiEmpirical (ATK-SE), but could also have been ATK-DFT.

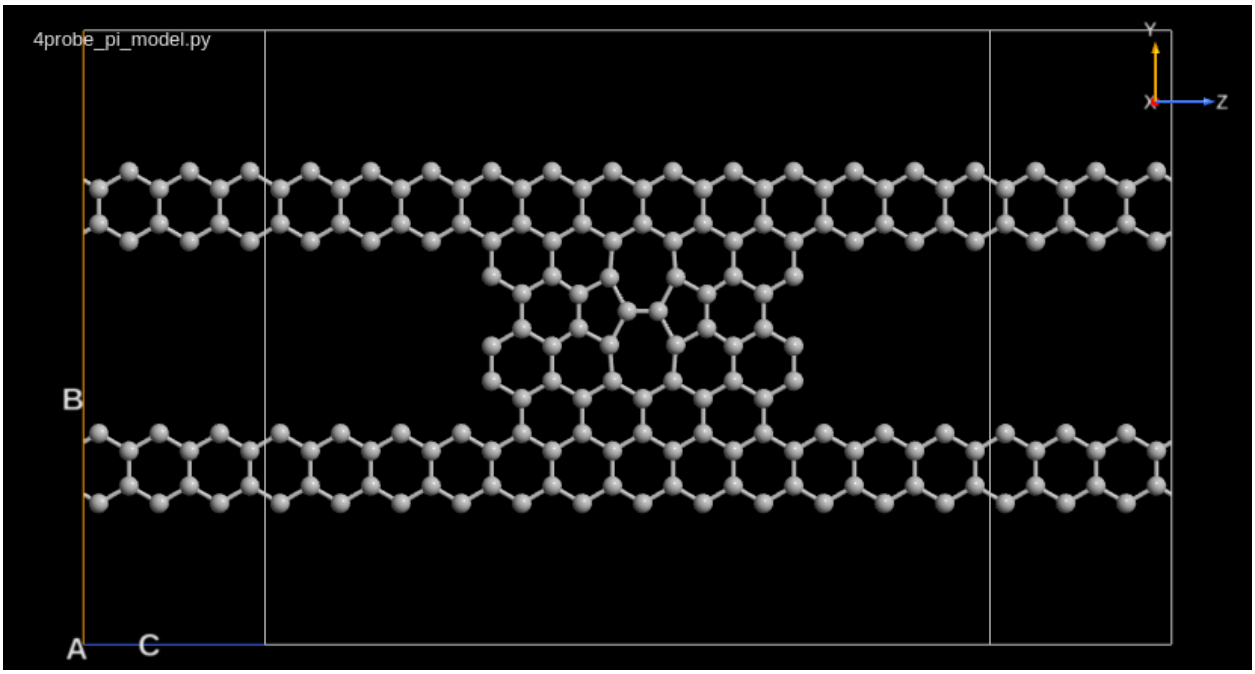
[3] Y.W. Son, M.L. Cohen, and S.G. Louie, Phys. Rev. Lett. **97**, 216803 (2006)

Multi-terminal conduction

You will in this section consider a 4-probe graphene device, and calculate the transmission between each of the four probes. For the calculation, you will extract the self energies of each of the four probes and implement the Fisher–Lee transmission function in a small QuantumATK Python script. The example is inspired by the 4-probe setup in[1].

Device calculations

You should first use the Slater–Koster tight-binding method to compute the ground state and zero-bias transmission at the Fermi level for the 4-terminal graphene device illustrated below.



Start up QuantumATK with a new project, for instance named 4probe graphene. Then download this pre-made QuantumATK Python script and save it in the project folder: `4probe_pi_model.py`. It sets up the device configuration with a Slater–Koster calculator, and then performs the calculation and computes the transmission at the Fermi level. All output is saved in `4probe_pi_model.hdf5`.

Run the script, either using the **Job Manager** or from command line:

```
$ atkpython 4probe_pi_model.py > 4probe_pi_model.log
```

The script will finish very quickly, and reports the Fermi level transmission in the log file:

```
+-----------------------------------------------------------+
| Transmission Spectrum Report                              |
| --------------------------------------------------------- |
| Left electrode Fermi level  =  3.851459e-01 eV            |
| Right electrode Fermi level =  3.851459e-01 eV            |
| Energy zero                 =  3.851459e-01 eV            |
+-----------------------------------------------------------+
```

(continues on next page)

[1] M.ăKoleini, M.ăPaulsson, and M.ăBrandbyge. Efficient Organometallic Spin Filter between Single-Wall Carbon Nanotube or Graphene Electrodes. *Phys. Rev. Lett.*, 98:197202, May 2007. doi:10.1103/PhysRevLett.98.197202.

(continued from previous page)

```
  energy       T(up)
    eV
0.000000e+00   7.553672e-01
```

Calculating the transmission between the 4 leads

The calculated transmission in the previous section is from the two leads on the left-hand side of the device, which are tagged lead1 and lead2, to the two leads on the right-hand side of the device, tagged lead3 and lead4.

Tip: You can check the tags if you send the device configuration to the Builder.

ATK has no analysis functions for calculating transmission between individual leads. For such a calculation, you need to calculate the broadening function of each lead, Γ_i, and then evaluate the Fisher–Lee relation,

$$T_{ij} = \mathrm{Tr}[\Gamma_i G^r \Gamma_j (G^r)^\dagger],$$

where G^r is the retarded Green function of the device.

The downloadable script `4probe_trans.py` performs this calculation. The script extracts internal QuantumATK quantities for the calculation, and the following section discusses details of the script.

Details of the script

The first part of the script is a utility function, which can zero out all entries in the Γ matrix, except for the orbitals belonging to the indices of the specified atoms. To obtain a list of the orbitals on the atoms, the function uses a `ProjectionList` object.

```
# Utility function to project out Gamma of the lead1, lead2, lead3, lead4
def projectGamma(configuration, gamma, indices):
    """Zero all components in Gamma which are not in the indices list
    @param configuration : The configuration giving the full Gamma
    @param gamma         : The full gamma
    @param indices       : Indices of the atoms to project onto

    @return              : The projected gamma
    """

    projection_list = ProjectionList(indices)
    orbital_index = projection_list.orbitalIndex(configuration)
    # Make a filter matrix consisting of 1's at the orbital_index
    filter = 0*gamma
    for i in orbital_index:
        for j in orbital_index:
            filter[i,j] = 1
    return gamma*filter
```

Another important part is the lines shown below, which use the tags on the device configuration to determine the indices of the atoms in each electrode.

```
# Project onto Lead 1, 2, 3, 4
lead1_index = device_configuration.indicesFromTags('lead1')
Gamma1 = projectGamma(device_configuration, Gamma_L, lead1_index)
lead2_index = device_configuration.indicesFromTags('lead2')
Gamma2 = projectGamma(device_configuration, Gamma_L, lead2_index)
lead3_index = device_configuration.indicesFromTags('lead3')
Gamma3 = projectGamma(device_configuration, Gamma_R, lead3_index)
lead4_index = device_configuration.indicesFromTags('lead4')
Gamma4 = projectGamma(device_configuration, Gamma_R, lead4_index)
```

The rest of the script should be self explanatory to experienced QuantumATK users.

Running the script

Save the script and execute it. It should produce the following output:

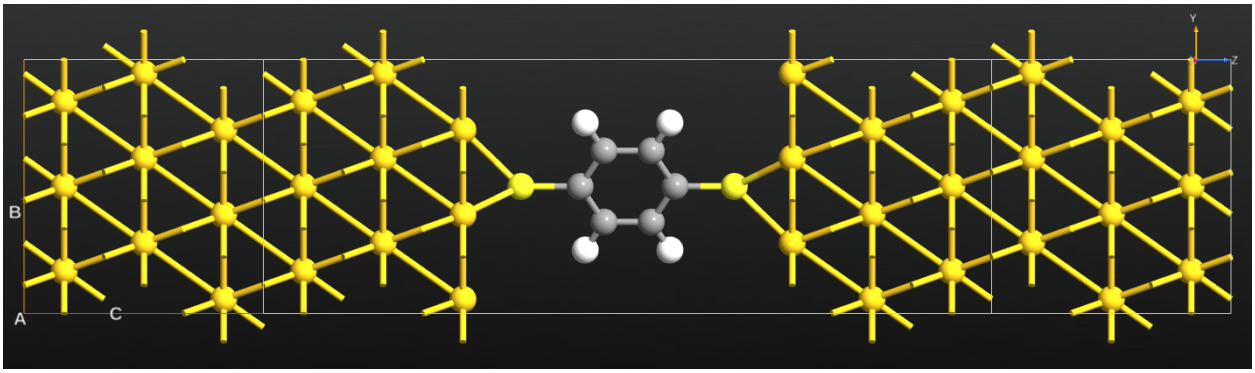
```
T: L (1,2) -> R    (3,4)   : 0.756157362529
T: Up(1,3) -> Down (2,4)   : 0.232988936132

Transmission Matrix
[[ 3.99103191  0.05825146  0.04799848  0.05832079]
 [ 0.05825146  4.60578343  0.05853133  0.59130676]
 [ 0.04799848  0.05853133  3.74982393  0.05788536]
 [ 0.05832079  0.59130676  0.05788536  1.96636417]]
```

The script reports the transmission matrix, T_{ij}. By summing up $i \in L$, $i \in R$ it is possible to get the transmission from left to right, which is identical (within numerical noise), to the value from the bare QuantumATK transmission calculation.

By summing up the transmission coefficients between the upper leads and the lower leads, it is possible to calculate the up-down transmission, as in[Page 1687, 1].

Transmission projection

You will here learn how to resolve a transmission calculation into molecular projected self-consistent Hamiltonian (MPSH) eigenstates. In this way, you can analyze how large a fraction of a transmission function is propagating through a particular eigenstate or part of the system. As an example, you will investigate a dithiol-benzene (DTB) ring between two gold surfaces, see[2].



[2] K.ăStokbro, J.ăTaylor, M.ăBrandbyge, J.-L. Mozos, and P.ăOrdejón. Theoretical study of the nonlinear conductance of Di-thiol benzene coupled to Au(111) surfaces via thiol and thiolate bonds. *Computational Materials Science*, 27(1-2):151–160, 2003. doi:10.1016/S0927-0256(02)00439-1.

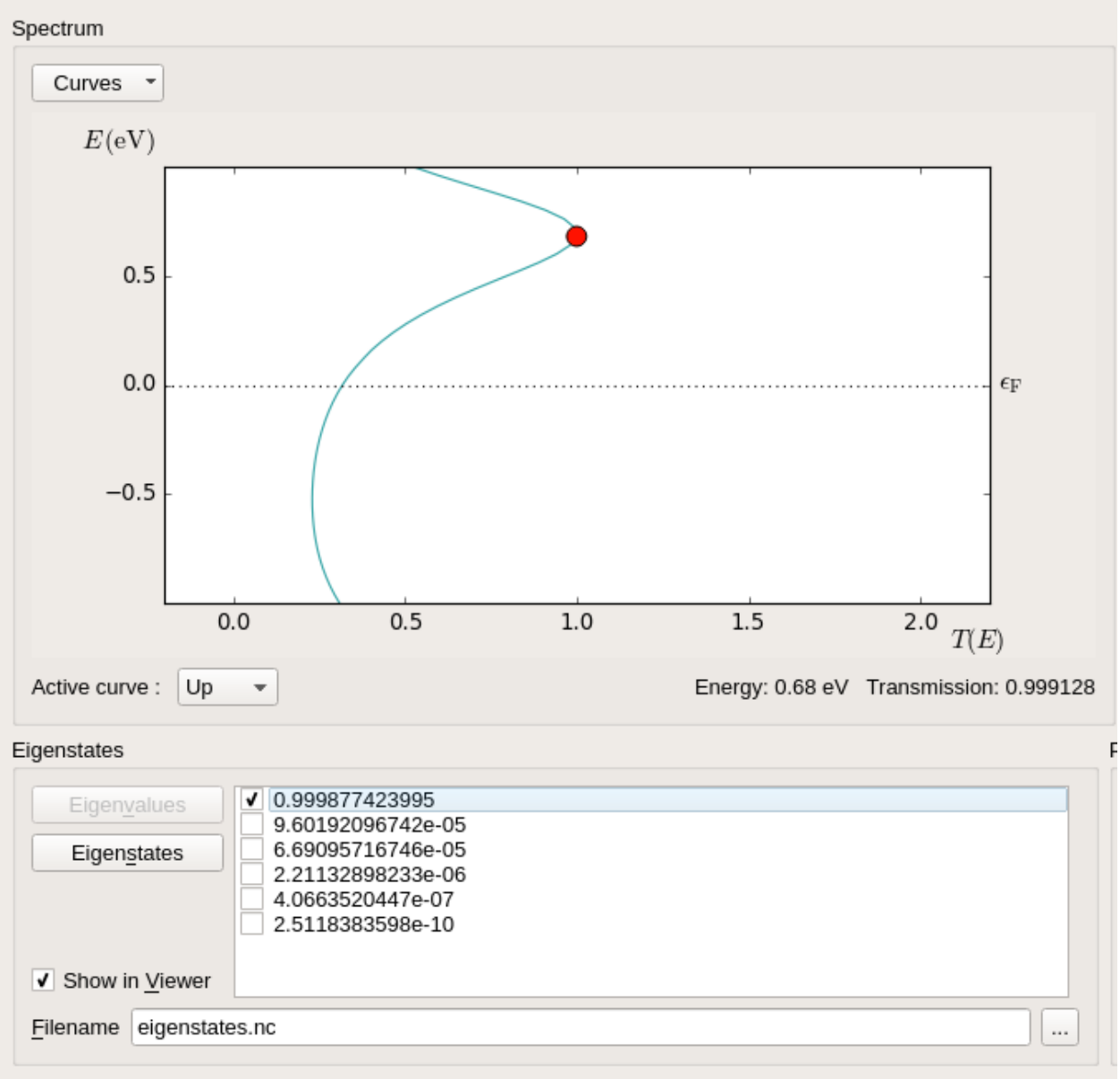
Running the calculations

Use the pre-made QuantumATK Python script `au_dtb_au.py`, which defines the Au–DTB–Au device configuration, performs a DFT-SE calculation using the extended Hückel method, and computes the -point transmission in an energy range ± 1 eV around the Fermi level. The calculations will finish in a few minutes.

Analyzing the transmission

Once the calculation has finished, the file `au_dtb_au.hdf5` will appear on the LabFloor. Select the **Transmission-Spectrum** item and open the **Transmission Analyzer**.

The -point transmission spectrum has a peak 0.68 eV above the Fermi level. Select this peak using the mouse, and click **Eigenvalues** to calculate the transmission eigenvalues. There is one dominating eigenvalue very close to 1, and several negligible eigenvalues.



Remove the tick from all except the highest eigenvalue, and click the **Eigenstates** button. This will calculate the scattering eigenstate in real space, and the Viewer will pop up with a visualization of it – choose the isosurface visualization option if prompted. You can then use the *Properties* → *Isosurfaces* options to tune the visualization.

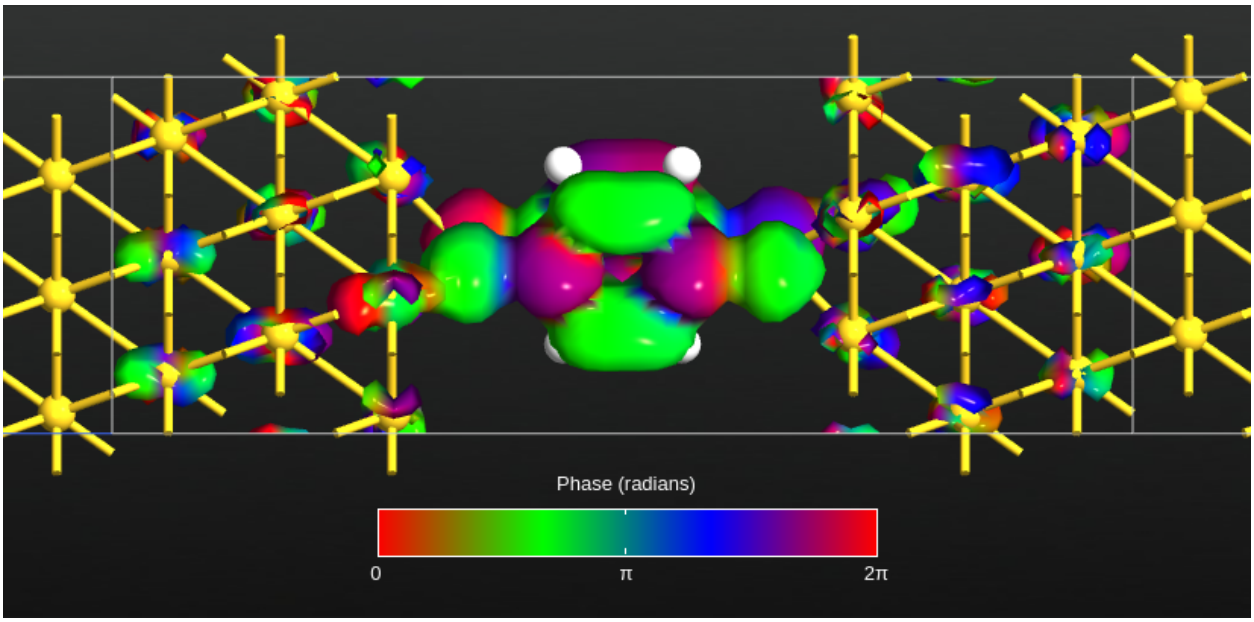


Fig. 3.117: Largest transmission eigenstate for energy 0.68 eV. The isosurface indicates an absolute value of 0.05 Å$^{-3/2}$, and the colors indicate the phase of the eigenstate.

Projecting the transmission

We would now like to project the transmission eigenstate onto the MPSH states of the molecule, to find the orbitals that carry the transmission.

The scattering state, $\psi(\mathbf{r})$, is expanded in basis orbitals, $\phi_i(\mathbf{r})$, through expansion coefficients v_i,

$$\psi(\mathbf{r}) = \sum_i v_i \phi_i(\mathbf{r}).$$

We now diagonalize the self-consistent Hamiltonian projected onto DTB,

$$H^{\mathrm{DTB}}\mathbf{c}_\alpha = \varepsilon_\alpha S^{\mathrm{DTB}}\mathbf{c}_\alpha,$$

where $\mathbf{c}_\alpha$ are the expansion coefficients of the MPSH states.

Next, we expand the projection of the scattering state of the DTB molecule in the MPSH states,

$$\mathbf{v} = \sum_\alpha a_\alpha \mathbf{c}_\alpha,$$

where the expansion coefficients are given by $a_\alpha = \mathbf{c}_\alpha^\dagger S^{\mathrm{DTB}}\mathbf{v}$. Through the magnitude of each a_α, we can get the relevance of each MPSH state, and

$$\sum_\alpha |a_\alpha|^2 = \mathbf{v}^\dagger S^{\mathrm{DTB}}\mathbf{v}.$$

Below is given a script, `projection.py`, which calculates the largest eigenvalue scattering state[3] at energy 0.68 eV, and calculates the projection weight of each MPSH state,

$$p_\alpha = \frac{|a_\alpha|^2}{\mathbf{v}^\dagger S^{\mathrm{DTB}} \mathbf{v}}.$$

```
from QuantumATK import *
from utilities import vectorToGrid, scatteringStates, averageFermiLevel
import scipy

# Read the configuration
device_configuration = nlread('au_dtb_au.hdf5', DeviceConfiguration)[0]
# Get H and S
H, S = calculateHamiltonianAndOverlap(device_configuration)
H = H.inUnitsOf(eV)

# Calculate average Fermi level
average_fermi_level = averageFermiLevel(device_configuration)
energy = average_fermi_level+0.68*eV

# Get index of orbitals on the Phenyl ring
projection_list = ProjectionList(elements = [Carbon, Hydrogen])
orbital_index = projection_list.orbitalIndex(device_configuration)
# Project H, S onto the Phenyl ring (MPSH)
H_dtb = H[orbital_index,:][:, orbital_index]
S_dtb = S[orbital_index,:][:, orbital_index]

# Calculate the mpsh eigenfunctions
w, v = scipy.linalg.eigh(H_dtb, S_dtb)

# Calculate the scattering eigenstates
T, c = scatteringStates(device_configuration, energy)

# Take the highest eigenstate
eigenvector = c[:,0]
# Project the eigenstate onto the DTB molecule
ev_dtb = eigenvector[orbital_index]
# Get the norm of the projected eigenstate
norm_ev = numpy.dot(numpy.conj(ev_dtb.transpose()), numpy.dot(S_dtb, ev_dtb))

# Loop over all MPSH
for i in range(len(w)):
    # Resolve the eigenstate into the MPSH states
    coeff = numpy.dot(numpy.conj(v[:,i].transpose()), numpy.dot(S_dtb, ev_dtb))
    # Find the strength of the MPSH projection
    p = numpy.conj(coeff)*coeff
    p = numpy.abs(p/norm_ev)
    # Print out the weight of each non negligible projection
    if p > 0.001:
        print(i, p)
```

The script uses some utility functions from the script `utilities.py`.

[3] M.ăPaulsson and M.ăBrandbyge. Transmission eigenchannels from nonequilibrium Greens functions. *Phys. Rev. B*, 76:115117, Sep 2007. doi:10.1103/PhysRevB.76.115117.

Running the scripts

Save both scripts, and execute `projection.py` using the Job Manager or from command line. It will generate the following output:

```
8 0.00686834365658
11 0.00467610628863
15 0.981647336721
17 0.00670447660685
```

The MPSH state number 15 is clearly the one with the largest projection.

Plotting the MPSH states

The script `mpsh.py` shown below will save MPSH states 8, 11, 15, and 17 into the HDF5 data file `mpsh.hdf5`. The script uses the function `vectorToGrid()`, which performs the folding of the eigenvector with the basis functions.

```
from QuantumATK import *
from utilities import vectorToGrid, averageFermiLevel
import scipy

# Read the configuration
device_configuration = nlread('au_dtb_au.hdf5', DeviceConfiguration)[0]
# Get H and S
H, S = calculateHamiltonianAndOverlap(device_configuration)
H = H.inUnitsOf(eV)

# Calculate average Fermi level
average_fermi_level = averageFermiLevel(device_configuration)

# Get index of orbitals on the Phenyl ring
projection_list = ProjectionList(elements=[Carbon, Hydrogen])
orbital_index = projection_list.orbitalIndex(device_configuration)
# Project H, S onto the Phenyl ring
H_dtb = H[orbital_index,:][:, orbital_index]
S_dtb = S[orbital_index,:][:, orbital_index]

# Calculate the eigenfunctions
w, v = scipy.linalg.eigh(H_dtb, S_dtb)
# Calculate eigen energies relative to the fermi level
eigen_energies =  (w*eV-average_fermi_level)

# Save eigenstates number 13-17
for i in [8, 11, 15, 17]:
    print('eigenenergy ', i,  eigen_energies[i])
    # Put the eigenvector into a vector of length of all orbitals
    number_orbitals = H.shape[0]
    eigenvector = numpy.zeros(number_orbitals, dtype=complex)
    eigenvector[orbital_index] = v[:,i]
    grid = vectorToGrid(eigenvector, device_configuration)
    nlsave('mpsh.hdf5', grid)
```

Run the script, and when finished drag and drop the object with ID `gID002` from `mpsh.hdf5` onto the Viewer.

Then add the device configuration to the plot by drag and dropping it from `au_dtb_au.hdf5`. By adjusting the plot properties, you should be able to get the image shown below.

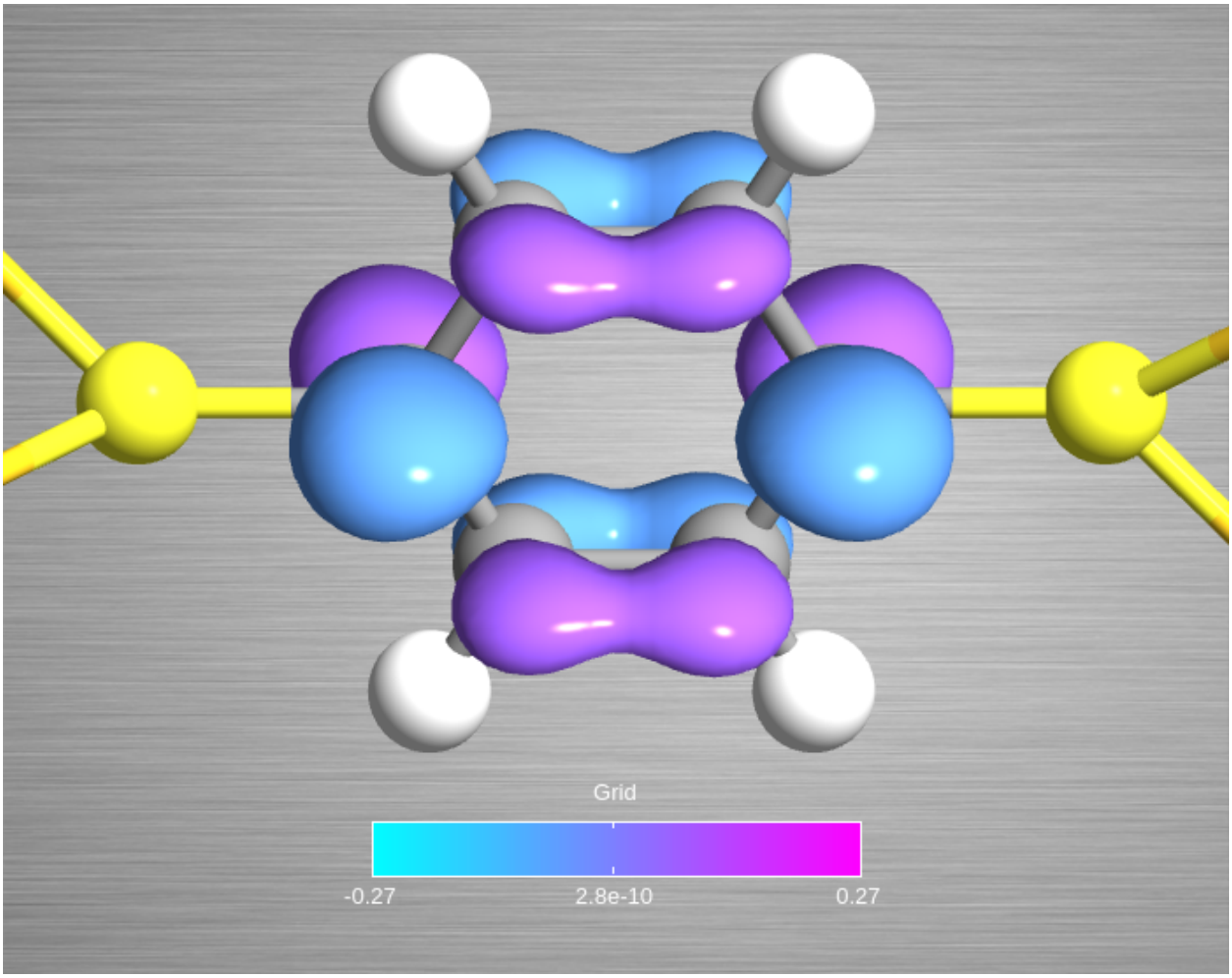


Fig. 3.118: Isosurface with isovalue 0.07 $Å^{-3/2}$ for MPSH state number 15.

Note: The `vectorToGrid()` method uses a finer grid spacing than the value set by the HuckelCalculator. This is why the image above has high-quality resolution.

AC conductance

The scripts presented in this section can be used to calculate the AC conductance of a nanodevice within the wide-band limit. The implementation follows closely the work by Yamamoto *et al.*[4].

We will here consider the (10,10) carbon nanotube device illustrated below, and use again the extended Hückel method with a π-model. Use `cnt_device.py` to perform the ground state device calculation.

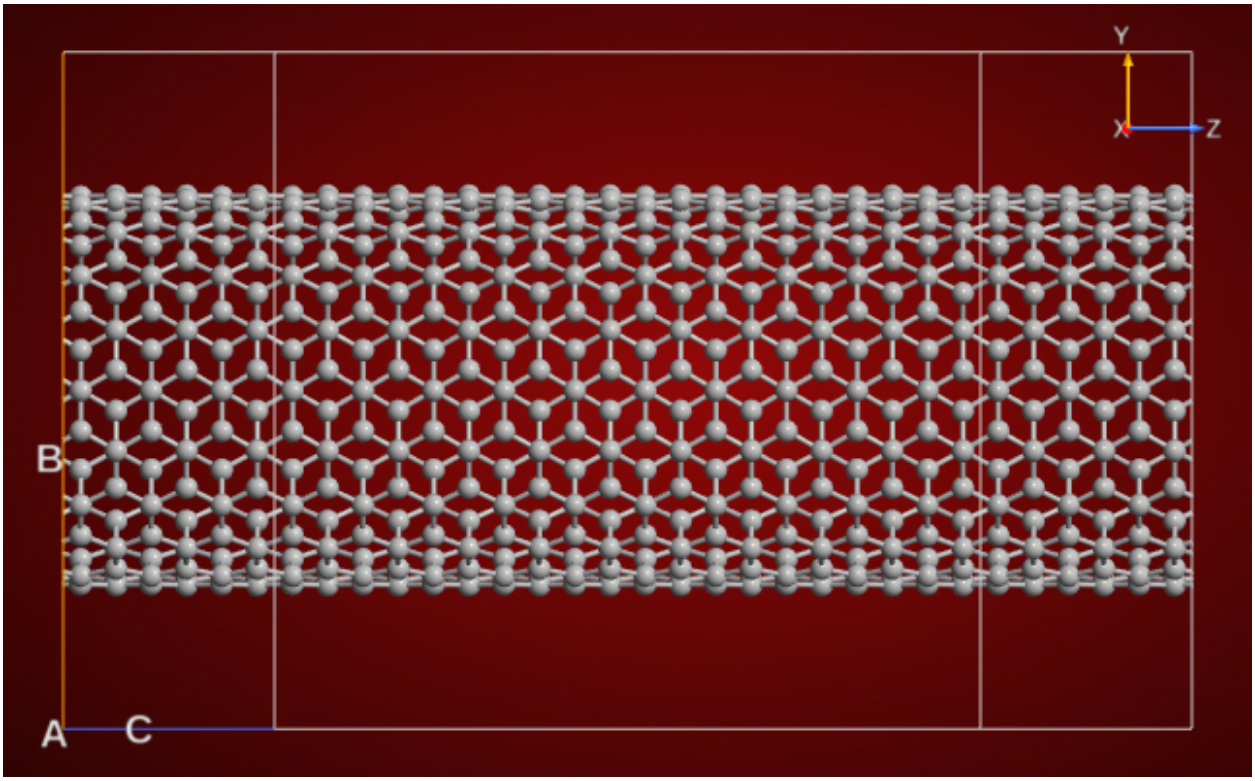


Fig. 3.119: (10,10) CNT device configuration with a central region length of 24.61 Å.

The **admittance** is defined as the inverse of the impedance, and is a measure of how easily a device will allow a current to flow. Download the scripts `cnt_admittance.py` and `admittance.py` to your QuantumATK project folder. Then execute `cnt_admittance.py`, which computes and plots the real (G) and imaginary (B) parts of the AC admittance for the CNT device.

Note: Since the implementation of the AC conductance is done in Python and uses dense matrices, the calculation is computationally inefficient, and for systems with a large number of orbitals the calculation can take substantial time.

---

[4] T.ăYamamoto, K.ăSasaoka, S.ăWatanabe, and K.ăWatanabe. Two chirality classes of ac quantum transport in metallic carbon nanotubes. *Phys. Rev. B*, 81:115448, Mar 2010. doi:10.1103/PhysRevB.81.115448.

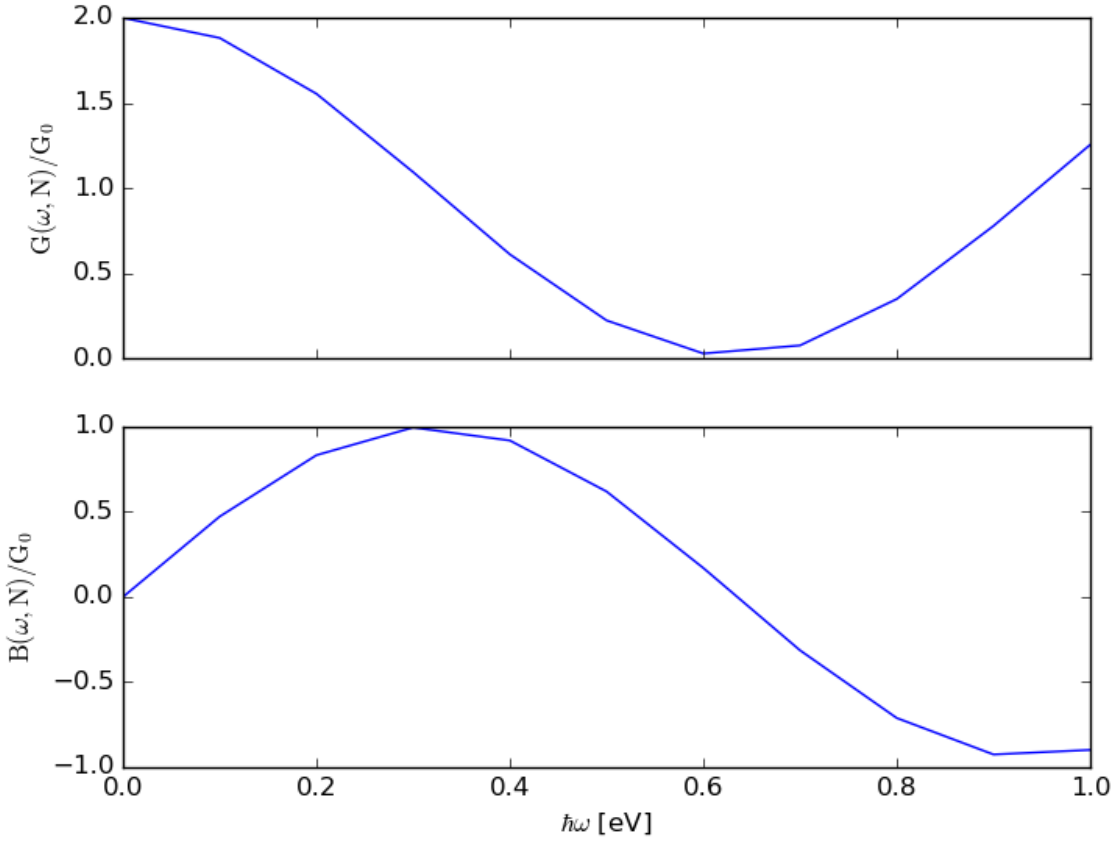


Fig. 3.120: Real (G) and imaginary (B) parts of the AC admittance for a (10,10) CNT device.

References

3.12.3 Slater-Koster tight-binding models in ATK-SE

Version: 2017.0

ATK-SemiEmpirical (**ATK-SE**) comes preloaded with Slater-Koster parameters for simple systems like Si and GaAs, using a couple of different nearest-neighbor models like sp^3s^* and $sp^3d^5s^*$. One very important feature of ATK-SE is, however, that it is possible to **manually** define custom Slater-Koster parameters, e.g. found in scientific literature, and this will be demonstrated in this tutorial.

The tutorial considers a nearest-neighbor Slater-Koster model for hydrogen-passivated silicon, which can for example be used to simulate H-terminated Si nanowires.

Introduction

There are two primary ingredients in a Slater-Koster (SK) model for electronic structure calculations:

1. The values of the **onsite** matrix elements for each element (or atom types) in the model.
2. The values of the **offsite** (hopping) matrix elements for each pair of atom types in the model, and how these depend on the distance between the atoms.

These points will be discussed extensively in this tutorial.

Nearest-neighbor model for silicon with placeholder hydrogen atoms

We will use the Boykin-type $sp^3d^5s^*$ model by Zheng *et al.*[1], which is a commonly used Slater-Koster parametrization for studying Si nanowires. The model extends an earlier parametrization of Si and Ge[2], which in turn builds on the original $sp^3d^5s^*$ models by Jancu *et al.*[3]. The latter is available as a built-in basis set in QuantumATK (denoted Bassani).

Note: The Boykin-type Slater-Koster model implemented in this tutorial will be available as a built-in basis set in QuantumATK as of the 2018 release.

The first step is to define the SK model for Si according to Ref.[Page 1697, 2], then the hydrogen matrix elements will be added to the model in a second step according to Ref.[1]. These are used to remove spurious electronic states due to dangling Si surface bonds, so they basically constitute a numerical trick.

[1] Y.ăZheng, C.ăRivas, R.ăLake, K.ăAlam, T.ăB. Boykin, and G.ăKlimeck. Electronic properties of silicon nanowires. *IEEE Transactions on Electron Devices*, 52(6):1097–1103, 2005. doi:10.1109/TED.2005.848077.

[2] T.ăB. Boykin, G.ăKlimeck, and F.ăOyafuso. Valence band effective-mass expressions in the sp3d5s* empirical tight-binding model applied to a Si and Ge parametrization. *Phys. Rev. B*, 69:115201, 2004. doi:10.1103/PhysRevB.69.115201.

[3] J.-M. Jancu, R.ăScholz, F.ăBeltram, and F.ăBassani. Empirical spds* tight-binding calculation for cubic semiconductors: General method and material parameters. *Phys. Rev. B*, 57:6493–6507, Mar 1998. doi:10.1103/PhysRevB.57.6493.

Onsite matrix element

First, consider the onsite matrix element. Here you need to specify the **angular momenta**, **number of valence electrons**, **ionization potentials** and **spin-orbit split** of the basis orbitals. The onsite **Hartree shift** and **spin split** are taken from built-in databases in QuantumATK.

The ionization potentials and spin-orbit split can be taken directly from Ref.[2]. There are four types of valence orbitals (3s, 3p, 3d, and an excited s-type orbital, commonly denoted s* in literature). The onsite Hartree shifts can be extracted from a built-in database; for the *unoccupied* orbitals 3d and s* will use the values from the *occupied* valence orbitals. The spin splits are also obtained from a built-in database.

The onsite matrix elements for each element in the basis set are specified as instances of the *SlaterKosterOnsiteParameters* ATKPython class. For the present silicon SK model, we use the variable `onsite` for an instance of this class:

```
# Define ionization potentials
Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV

# Define spin-orbit split
split =  0.01989*eV

onsite = SlaterKosterOnsiteParameters(
            element                     = Silicon,
            angular_momenta             = [0,1,2,0],
            number_of_valence_electrons = 4,
            filling_method              = SphericalSymmetric,
            ionization_potential        = [Es,Ep,Ed,Es1],
            onsite_hartree_shift=ATK_U(PeriodicTable.Silicon, ["3p"], 'ncp')[0],
            onsite_spin_split=ATK_W(PeriodicTable.Silicon, [ "3p", "3p", "3p", "3s" ]),
            onsite_spin_orbit_split=[0.0,2*split, 0.0, 0.0]*eV,
           )
```

Note that the `element` for which these onsite parameters apply to is a **required argument**.

Tip: Occupations: What actually happens when specifying `number_of_valence_electrons = 4` is that the four electrons are equally distributed among all the 10 available orbitals. This gives the following electron occupations of the Slater-Koster orbitals:

3s
(4/10) electrons * 1 s-type orbital = 0.4

3p
(4/10) electrons * 3 p-type orbitals = 1.2

3d
(4/10) electrons * 5 d-type orbitals = 2.0

s*
(4/10) electrons * 1 s-type orbital = 0.4

These occupations could also have been directly specified in the *SlaterKosterOnsiteParameters* instead of specifying the total number of electrons:

```
onsite = SlaterKosterOnsit+eParameters(
            element                       = Silicon,
            angular_momenta               = [0,1,2,0],
            # number_of_valence_electrons = 4,
            occupations                   = [0.4, 1.2, 2.0, 0.4],
```

Offsite matrix elements

A Slater-Koster model is in general not based on the concept of neighbors – the offsite (hopping) matrix elements are instead defined in a range of distances around each atom, possibly using a scaling function to specify the distance-dependence of the Hamiltonian matrix element. A cut-off distance should also be specified, above which atoms do not interact (the matrix element is set to zero beyond this distance).

The hopping matrix elements are therefore tabulated in an interval around the silicon nearest-neighbor distance, using values from Ref.[Page 1697, 2]:

```
from math import sqrt

# ----------------------------------------------------#
#                     Silicon                         #
#-----------------------------------------------------#

# a is the cubic lattice constant
a = 5.431*Ang

# Nearest neighbor distance
d_n1 = sqrt(3.0)/4.0*a
# Second-nearest neighbor distance
d_n2 = sqrt(2.0)/2.0*a

# Setup list of distances
epsilon = numpy.linspace(-0.20, 0.20, 41)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

# Setup hopping elements
si_si_sss   = [ -1.95933*eV for x in epsilon ]
si_si_s1s1s = [ -4.24135*eV for x in epsilon ]
si_si_ss1s  = [ -1.52230*eV for x in epsilon ]
si_si_sps   = [  3.02562*eV for x in epsilon ]
si_si_s1ps  = [  3.15565*eV for x in epsilon ]
si_si_sds   = [ -2.28485*eV for x in epsilon ]
si_si_s1ds  = [ -0.80993*eV for x in epsilon ]
si_si_pps   = [  4.10364*eV for x in epsilon ]
si_si_ppp   = [ -1.51801*eV for x in epsilon ]
si_si_pds   = [ -1.35554*eV for x in epsilon ]
si_si_pdp   = [  2.38479*eV for x in epsilon ]
si_si_dds   = [ -1.68136*eV for x in epsilon ]
si_si_ddp   = [  2.58880*eV for x in epsilon ]
si_si_ddd   = [ -1.81400*eV for x in epsilon ]
```

Note that the cut-off distance is taken at the midpoint between the first and second nearest neighbors (the separate element added to the list), where the corresponding matrix element is set to zero.

Important: The hopping matrix elements defined above are only valid for silicon with a lattice constant of 5.431 Å.

However, the scaling function for the matrix elements could also be fitted to allow calculations for more irregular structures. In that approach, it is commonly assumed that the matrix elements $ijk(d)$ follow a power law,

$$ijk(d) = ijk(d_0)\left(\frac{d_0}{d}\right)^{n_{ijk}},$$

where d_0 is the ideal neighbor distance. Harrison[4] originally used a general exponent of $n_{ijk} = 2$ for all matrix elements ijk, but later work, e.g.[Page 1697, 3] and[5], extends this to fit an individual exponent for each matrix element.

Defining the full Slater-Koster table

In QuantumATK, the Slater-Koster basis set should be implemented as an instance of the *SlaterKosterTable* ATKPython class. As explained in detail in the *ATK Manual*, the onsite and offsite matrix elements in the *SlaterKosterTable* are defined using special keyword arguments.

Onsite parameters

The onsite matrix element for Si is simply passed to the *SlaterKosterTable* using the label `silicon`:

```
# Create Slater-Koster table
basis_set = SlaterKosterTable(silicon     = onsite,
```

Offsite parameters

All the offsite matrix elements defined above are given as individual keyword arguments when setting up the *SlaterKosterTable*. As explained in the *ATK Manual*, these arguments must be on the form `element1_element2_XYZ=list`, where `X` and `Y` are the labels of the orbitals on the first and second elements (both can be `s`, `p`, `d`, or `f`), while `Z` is the type of interaction (`s`, `p`, or `d` for σ, π, or δ, respectively).

Thus, for example, the $\mathrm{sd}\pi$ offsite matrix element between carbon and silicon in some hypothetical SK model would be set with the keyword `si_c_sdp`.

In the present case of pure silicon, the full Slater-Koster table definition, including both onsite and offsite parameters, becomes as follows:

```
# Create Slater-Koster table
basis_set = SlaterKosterTable(silicon     = onsite,
                              si_si_sss   = zip(distances,si_si_sss),
                              si_si_s1s1s = zip(distances,si_si_s1s1s),
                              si_si_ss1s  = zip(distances,si_si_ss1s),
                              si_si_sps   = zip(distances,si_si_sps),
                              si_si_s1ps  = zip(distances,si_si_s1ps),
                              si_si_sds   = zip(distances,si_si_sds),
                              si_si_s1ds  = zip(distances,si_si_s1ds),
```

(continues on next page)

[4] WalterăA. Harrison. *Electronic Structure and the Properties of Solids: The Physics of the Chemical Bond*. W. H. Freeman & Company (San Francisco), 1980.

[5] T.ăB. Boykin, M.ăLuisier, M.ăSalmani-Jelodar, and G.ăKlimeck. Strain-induced, off-diagonal, same-atom parameters in empirical tight-binding theory suitable for [110] uniaxial strain applied to a silicon parametrization. *Phys. Rev. B*, 81:125202, 2010. doi:10.1103/PhysRevB.81.125202.

(continued from previous page)

```
                                si_si_pps   = zip(distances,si_si_pps),
                                si_si_ppp   = zip(distances,si_si_ppp),
                                si_si_pds   = zip(distances,si_si_pds),
                                si_si_pdp   = zip(distances,si_si_pdp),
                                si_si_dds   = zip(distances,si_si_dds),
                                si_si_ddp   = zip(distances,si_si_ddp),
                                si_si_ddd   = zip(distances,si_si_ddd),
                               )
```

Note: **Some additional remarks:**

- The order of the element labels `X` and `Y` matters, if there is more than one element in the model. The orbital `i` is associated with the element `X` orbital, and `j` with `Y`.
- The element labels are not case-sensitive and may be abbreviated. Thus, `Si`, `si`, or `Silicon` are all accepted for silicon.

Tip: This present SK basis set contains two s-type orbitals (3s and s^*). To distinguish orbitals of the same angular momentum, there are two things to keep in mind:

- For the onsite elements, the identification is made by the ordering of the ionization potentials vs. the occupations and angular momenta, etc.
- For the offsite elements, a running index is added, so the first s-orbital is s0, the second one s1, etc. The 0 for the first orbital can be omitted to reduce the notation, so as you see above, the $s * p\sigma$ matrix element will be defined by the keyword `si_si_s1ps`, etc.

Full script

The full script defining the silicon Slater-Koster basis set combines all of the script snippets shown above:

```
from QuantumATK import *

'''
Generate a Slater-Koster basis set for silicon based on
T. B. Boykin et al., PRB 69, 115201 (2004)
http://dx.doi.org/10.1103/PhysRevB.69.115201
'''

from math import sqrt

# ------------------------------------------------#
#                     Silicon                     #
#-------------------------------------------------#

# a is the cubic lattice constant
a = 5.431*Ang

# Nearest neighbor distance
d_n1 = sqrt(3.0)/4.0*a
# Second-nearest neighbor distance
```

(continues on next page)

(continued from previous page)

```
d_n2 = sqrt(2.0)/2.0*a

# Setup list of distances
epsilon = numpy.linspace(-0.20, 0.20, 41)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

# Setup hopping elements
si_si_sss   = [ -1.95933*eV for x in epsilon ]
si_si_s1s1s = [ -4.24135*eV for x in epsilon ]
si_si_ss1s  = [ -1.52230*eV for x in epsilon ]
si_si_sps   = [  3.02562*eV for x in epsilon ]
si_si_s1ps  = [  3.15565*eV for x in epsilon ]
si_si_sds   = [ -2.28485*eV for x in epsilon ]
si_si_s1ds  = [ -0.80993*eV for x in epsilon ]
si_si_pps   = [  4.10364*eV for x in epsilon ]
si_si_ppp   = [ -1.51801*eV for x in epsilon ]
si_si_pds   = [ -1.35554*eV for x in epsilon ]
si_si_pdp   = [  2.38479*eV for x in epsilon ]
si_si_dds   = [ -1.68136*eV for x in epsilon ]
si_si_ddp   = [  2.58880*eV for x in epsilon ]
si_si_ddd   = [ -1.81400*eV for x in epsilon ]

# Define ionization potentials
Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV

# Define spin-orbit split
split =  0.01989*eV

onsite = SlaterKosterOnsiteParameters(
            element                    = Silicon,
            angular_momenta            = [0,1,2,0],
            number_of_valence_electrons = 4,
            filling_method             = SphericalSymmetric,
            ionization_potential       = [Es,Ep,Ed,Es1],
            onsite_hartree_shift=ATK_U(PeriodicTable.Silicon, ["3p"], 'ncp')[0],
            onsite_spin_split=ATK_W(PeriodicTable.Silicon, [ "3p", "3p", "3p", "3s" ]),
            onsite_spin_orbit_split=[0.0,2*split, 0.0, 0.0]*eV,
           )

# Create Slater-Koster table
basis_set = SlaterKosterTable(silicon     = onsite,
                              si_si_sss   = zip(distances,si_si_sss),
                              si_si_s1s1s = zip(distances,si_si_s1s1s),
                              si_si_ss1s  = zip(distances,si_si_ss1s),
                              si_si_sps   = zip(distances,si_si_sps),
                              si_si_s1ps  = zip(distances,si_si_s1ps),
                              si_si_sds   = zip(distances,si_si_sds),
                              si_si_s1ds  = zip(distances,si_si_s1ds),
                              si_si_pps   = zip(distances,si_si_pps),
```

(continues on next page)

(continued from previous page)

```
                                si_si_ppp   = zip(distances,si_si_ppp),
                                si_si_pds   = zip(distances,si_si_pds),
                                si_si_pdp   = zip(distances,si_si_pdp),
                                si_si_dds   = zip(distances,si_si_dds),
                                si_si_ddp   = zip(distances,si_si_ddp),
                                si_si_ddd   = zip(distances,si_si_ddd),
                               )
```

SiBasisSet.py

Silicon band structure

A simple test case for the custom silicon Slater-Koster model is to compute the band structure of silicon:

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Silicon
# ------------------------------------------------------------
lattice = FaceCenteredCubic(5.431*Angstrom)
elements = [Silicon, Silicon]
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
from SiBasisSet import basis_set
k_point_sampling = MonkhorstPackGrid(9,9,9,force_timereversal=False)
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=10.0*Hartree,
    )
calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    spin_polarization=SpinOrbit,
    )
bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('si_bs.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
```

(continues on next page)

(continued from previous page)

```
    route=[['L', 'G', 'X', 'W', 'K', 'L', 'W', 'X', 'U'],['K', 'G']],
    points_per_segment=50
    )
nlsave('si_bs.hdf5', bandstructure)
```

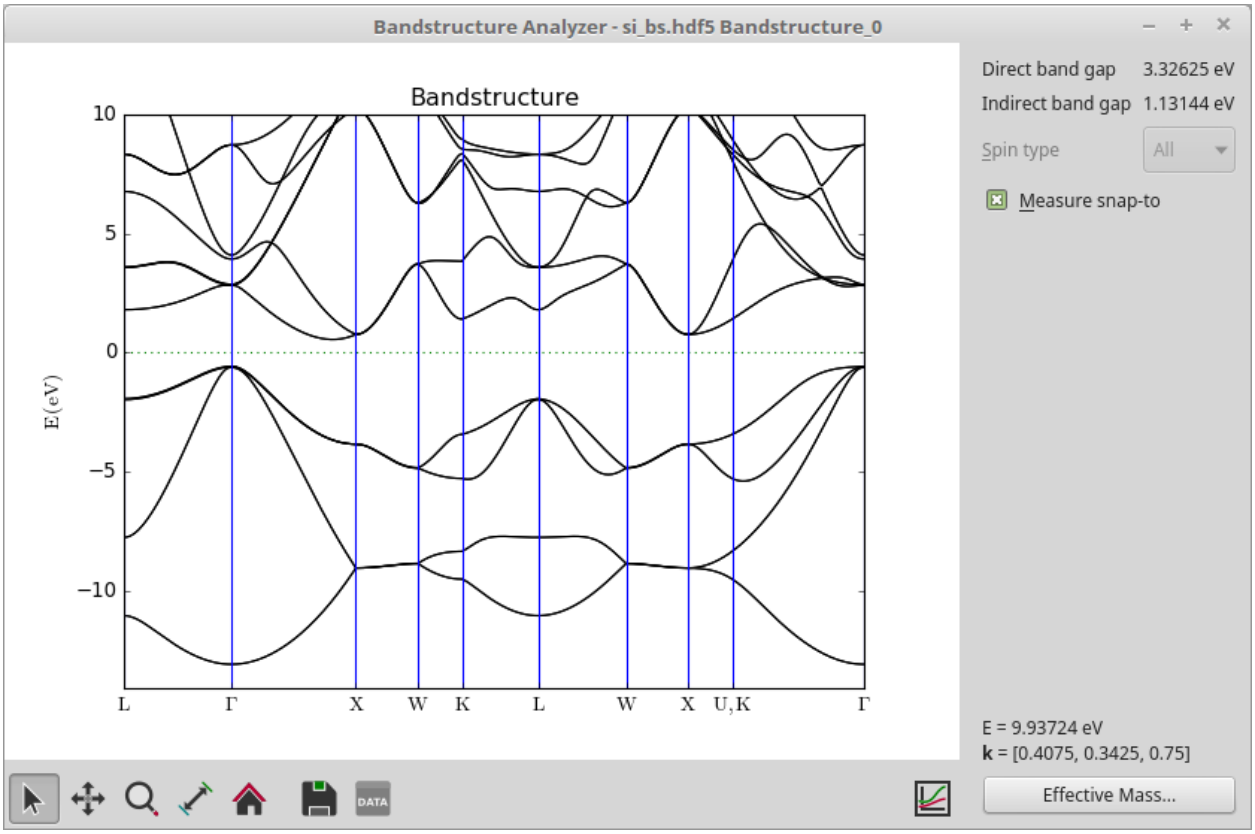
`si_bs.py`

Download this script, as well as the script `SiBasisSet.py` containing the Si basis set, and save them in your QuantumATK Project Folder.

Use the **Job Manager** to run the script `si_bs.py`, or run it from a terminal:

```
atkpython si_bs.py
```

The job should finish in a few seconds, and produces a very accurate band structure of silicon with an indirect band gap of 1.13 eV.



Note: The energy bands in Si are degenerate at the U and K points, and by the special way we construct the band structure route in the script, the band structure plot makes a discontinuous break in the route between these points, i.e. it goes from X to U, then jumps to K and continues to Γ.

Adding hydrogen

The goal of this tutorial is to enable modeling of H-terminated Si nanowires, so we need to expand the Slater-Koster table with the parameters for hydrogen from Ref.[Page 1697, 1]. As previously discussed, the role of the hydrogen atoms is to eliminate the effects of the unsaturated Si bonds on the surface of the wire, not to represent real hydrogen atoms. In an experiment it is of course quite possible that hydrogen atoms, or other atoms, will attach themselves to these dangling bonds, but that is not the point here.

The **onsite** matrix element for hydrogen is added like this:

```
# Define ionization potentials
Es_H    =  0.999840*eV

onsite_H = SlaterKosterOnsiteParameters(
                element                 = Hydrogen,
                angular_momenta         = [0],
                occupations             = [1.0],
                ionization_potential    = [Es_H],
                filling_method          = SphericalSymmetric,
                onsite_hartree_shift    = ATK_U(PeriodicTable.Hydrogen, ['1s']),
                onsite_spin_split       = [[0.0]]*eV,
                onsite_spin_orbit_split = [0.0]*eV,
                vacuum_level            = 0.0*Hartree,
```

Specifying the **offsite** parameters is done like this:

```
# reference distances
d_h_n1 = 1.4*Angstrom
d_h_n2 = 2.2*Angstrom

# Setup list of distances
h_distances = [d_h_n1 * (1.0+r) for r in epsilon] + [0.5 * (d_h_n1+d_h_n2)]

# Setup hopping elements
h_si_sss  = [ -3.999720*eV / 1.0 for r in epsilon ]
h_si_ss1s = [ -1.697700*eV / 1.0 for r in epsilon ]
h_si_sps  = [  4.251750*eV / 1.0 for r in epsilon ]
h_si_sds  = [ -2.105520*eV / 1.0 for r in epsilon ]
```

The onsite and offsite parameters are then simply added to the already prepared silicon *SlaterKosterTable*:

```
from QuantumATK import *

'''
Generate a Slater-Koster basis set for silicon based on
T. B. Boykin et al., PRB 69, 115201 (2004)
http://dx.doi.org/10.1103/PhysRevB.69.115201
'''

from math import sqrt

# ------------------------------------------------#
#                     Silicon                     #
#-------------------------------------------------#
```

(continues on next page)

(continued from previous page)

```
# a is the cubic lattice constant
a = 5.431*Ang

# Nearest neighbor distance
d_n1 = sqrt(3.0)/4.0*a
# Second-nearest neighbor distance
d_n2 = sqrt(2.0)/2.0*a

# Setup list of distances
epsilon = numpy.linspace(-0.20, 0.20, 41)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

# Setup hopping elements
si_si_sss   = [ -1.95933*eV for x in epsilon ]
si_si_s1s1s = [ -4.24135*eV for x in epsilon ]
si_si_ss1s  = [ -1.52230*eV for x in epsilon ]
si_si_sps   = [  3.02562*eV for x in epsilon ]
si_si_s1ps  = [  3.15565*eV for x in epsilon ]
si_si_sds   = [ -2.28485*eV for x in epsilon ]
si_si_s1ds  = [ -0.80993*eV for x in epsilon ]
si_si_pps   = [  4.10364*eV for x in epsilon ]
si_si_ppp   = [ -1.51801*eV for x in epsilon ]
si_si_pds   = [ -1.35554*eV for x in epsilon ]
si_si_pdp   = [  2.38479*eV for x in epsilon ]
si_si_dds   = [ -1.68136*eV for x in epsilon ]
si_si_ddp   = [  2.58880*eV for x in epsilon ]
si_si_ddd   = [ -1.81400*eV for x in epsilon ]

# Define ionization potentials
Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV

# Define spin-orbit split
split =  0.01989*eV

onsite = SlaterKosterOnsiteParameters(
            element                     = Silicon,
            angular_momenta             = [0,1,2,0],
            number_of_valence_electrons = 4,
            filling_method              = SphericalSymmetric,
            ionization_potential        = [Es,Ep,Ed,Es1],
            onsite_hartree_shift=ATK_U(PeriodicTable.Silicon, ["3p"], 'ncp')[0],
            onsite_spin_split=ATK_W(PeriodicTable.Silicon, [ "3p", "3p", "3p", "3s" ]),
            onsite_spin_orbit_split=[0.0,2*split, 0.0, 0.0]*eV,
           )

# ------------------------------------------------#
#                    Hydrogen                     #
#------------------------------------------------#
```

(continues on next page)

(continued from previous page)

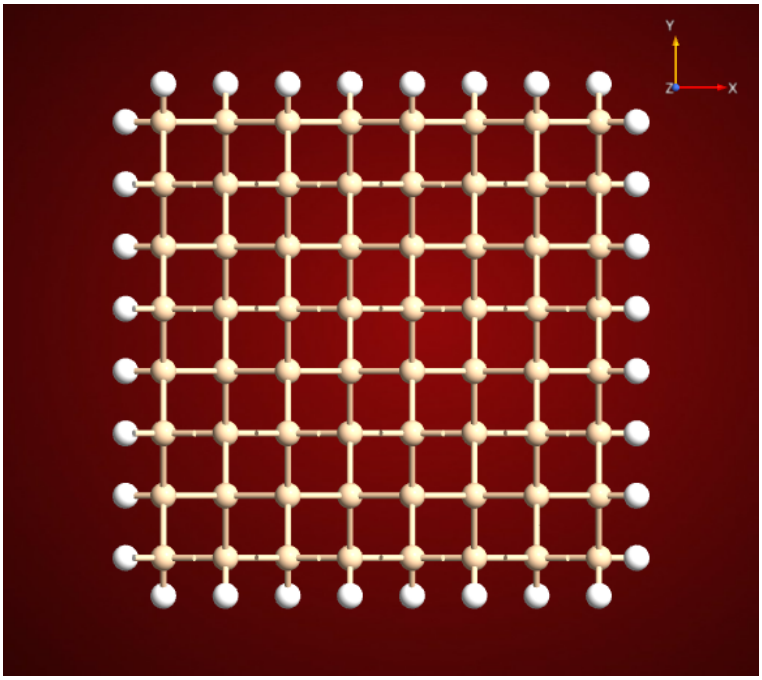
```
# reference distances
d_h_n1 = 1.4*Angstrom
d_h_n2 = 2.2*Angstrom

# Setup list of distances
h_distances = [d_h_n1 * (1.0+r) for r in epsilon] + [0.5 * (d_h_n1+d_h_n2)]

# Setup hopping elements
h_si_sss  = [ -3.999720*eV / 1.0 for r in epsilon ]
h_si_ss1s = [ -1.697700*eV / 1.0 for r in epsilon ]
h_si_sps  = [  4.251750*eV / 1.0 for r in epsilon ]
h_si_sds  = [ -2.105520*eV / 1.0 for r in epsilon ]

# Define ionization potentials
Es_H    =  0.999840*eV

onsite_H = SlaterKosterOnsiteParameters(
                element                 = Hydrogen,
                angular_momenta         = [0],
                occupations             = [1.0],
                ionization_potential    = [Es_H],
                filling_method          = SphericalSymmetric,
                onsite_hartree_shift    = ATK_U(PeriodicTable.Hydrogen, ['1s']),
                onsite_spin_split       = [[0.0]]*eV,
                onsite_spin_orbit_split = [0.0]*eV,
                vacuum_level            = 0.0*Hartree,
               )

# Create Slater-Koster table
basis_set = SlaterKosterTable(silicon     = onsite,
                              hydrogen    = onsite_H,
                              si_si_sss   = zip(distances,si_si_sss),
                              si_si_s1s1s = zip(distances,si_si_s1s1s),
                              si_si_ss1s  = zip(distances,si_si_ss1s),
                              si_si_sps   = zip(distances,si_si_sps),
                              si_si_s1ps  = zip(distances,si_si_s1ps),
                              si_si_sds   = zip(distances,si_si_sds),
                              si_si_s1ds  = zip(distances,si_si_s1ds),
                              si_si_pps   = zip(distances,si_si_pps),
                              si_si_ppp   = zip(distances,si_si_ppp),
                              si_si_pds   = zip(distances,si_si_pds),
                              si_si_pdp   = zip(distances,si_si_pdp),
                              si_si_dds   = zip(distances,si_si_dds),
                              si_si_ddp   = zip(distances,si_si_ddp),
                              si_si_ddd   = zip(distances,si_si_ddd),
                              h_si_sss    = zip(h_distances,h_si_sss),
                              h_si_ss1s   = zip(h_distances,h_si_ss1s),
                              h_si_sps    = zip(h_distances,h_si_sps),
                              h_si_sds    = zip(h_distances,h_si_sds),
                             )
```

SiHBasisSet.py

Band gaps of passivated silicon nanowires

Due to quantum confinement, the band gap of silicon nanowires usually increase as the wire thickness decreases. You will here demonstrate this using the Si-H Slater-Koster model defined in the sections above.

The script `wires.py` contains 4 silicon nanowire configurations of different thickness. They are all hydrogen passivated, as exemplified below. The wires are perdiodic along the Z direction.



The script `nw_bs.py` loopes over all 4 nanowires and does a nonself-consistent Slater-Koster calculation and a band-structure analysis for each wire. Results are saved in individual HDF5 data files.

```
from SiHBasisSet import basis_set
from wires import wire_2x2, wire_3x3, wire_4x4, wire_5x5

for i, wire in zip([2,3,4,5],[wire_2x2,wire_3x3,wire_4x4,wire_5x5]):
    out = 'wire_%ix%i.hdf5' % (i,i)

    # ------------------------------------------------------------
    # Calculator
    # ------------------------------------------------------------
    k_point_sampling = MonkhorstPackGrid(nc=9,force_timereversal=False)
    numerical_accuracy_parameters = NumericalAccuracyParameters(
        k_point_sampling=k_point_sampling,
        density_mesh_cutoff=10.0*Hartree,
        )
    calculator = SlaterKosterCalculator(
        basis_set=basis_set,
        numerical_accuracy_parameters=numerical_accuracy_parameters,
        spin_polarization=SpinOrbit,
        )

    wire.setCalculator(calculator)
```

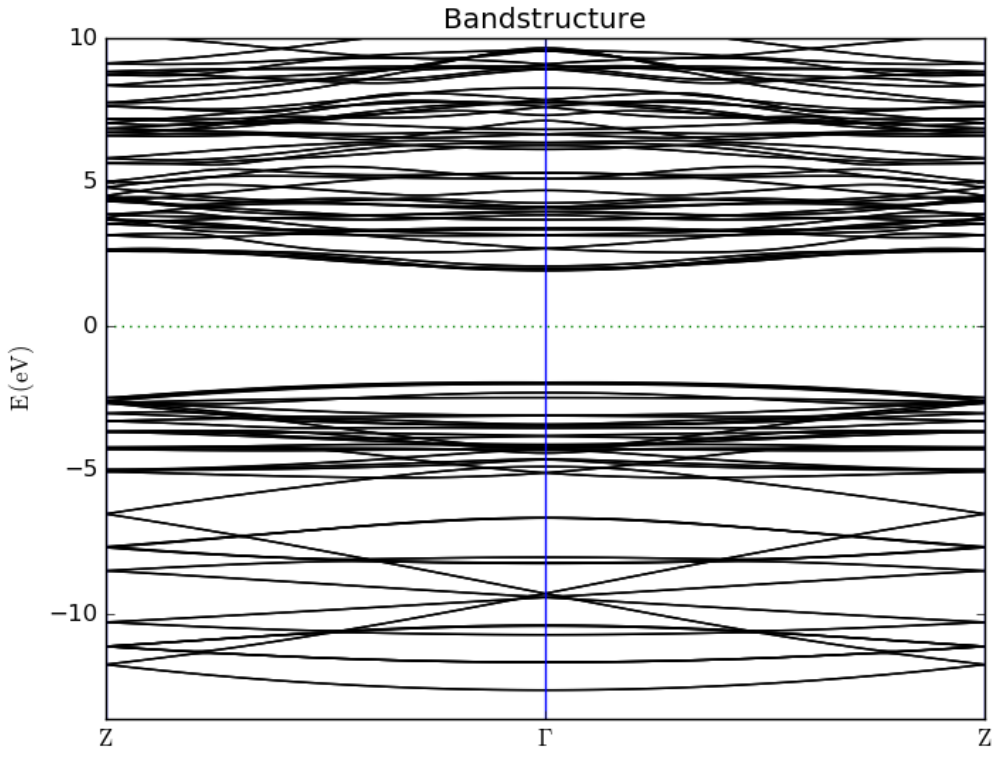
(continues on next page)

(continued from previous page)

```
nlprint(wire)
wire.update()
nlsave(out, wire)

# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=wire,
    route=['Z', 'G', 'Z'],
    points_per_segment=50
    )
nlsave(out, bandstructure)
```

Download both `wires.py` and `nw_bs.py`, and save them in your QuantumATK Project Folder, which should also contain `SiHBasisSet.py`. Then execute the script `nw_bs.py` using the Job Manager or a terminal – the calculation may take up to 10 minutes to run.

The resulting bandstructure analysis objects should appear on the LabFloor, and you can use the **Bandstructure Analyzer** to visualize the nanowire band structures. You will find for all 4 wires that the fundamental gap is located at the Γ point, and is significantly larger than the (indirect) band gap of bulk Si.

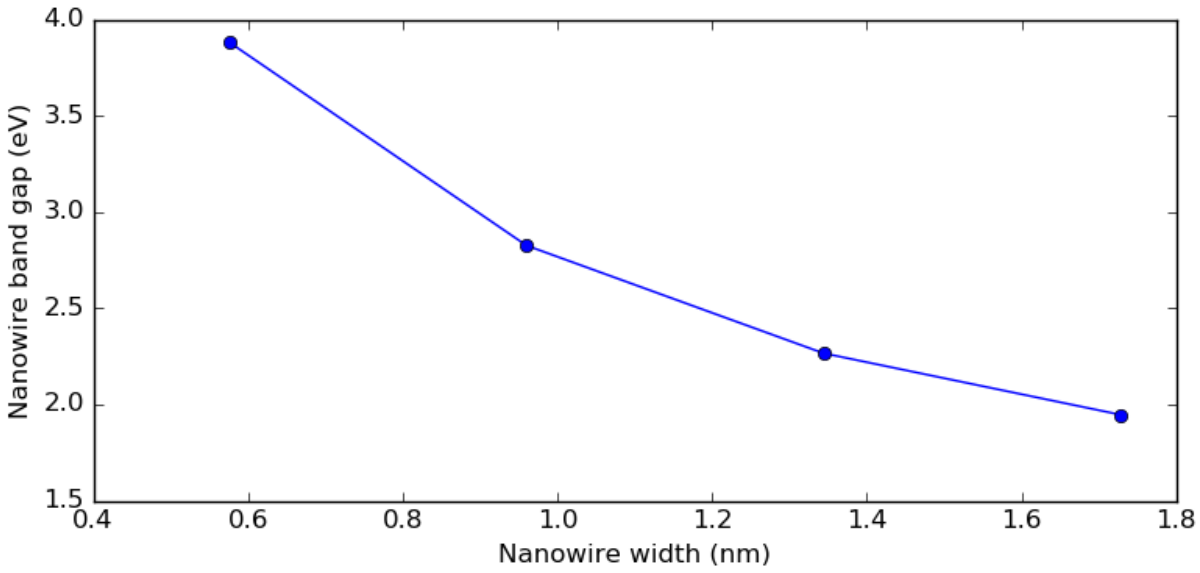


You can also use the script `nw_gaps.py` to extract the direct band gaps and compute the nanowire thicknesses, and to plot the data.

```
# Read data
gaps = []
lengths = []
for i in [2,3,4,5]:
    out = 'wire_%ix%i.hdf5' % (i,i)
    bandstructure = nlread(out, Bandstructure)[0]
    gap =bandstructure._directBandGap().inUnitsOf(eV)
    gaps.append(gap)
    wire = nlread(out, BulkConfiguration)[0]
    coordinates = wire.cartesianCoordinates().inUnitsOf(nanoMeter)
    indices = numpy.where(numpy.array(wire.symbols())=='Si')[0]
    coords = coordinates[indices][:,0]
    length = max(coords)-min(coords)
    lengths.append(length)

# Plot data
import pylab
pylab.figure(figsize=(8,4))
pylab.plot(lengths, gaps, 'bo-')
pylab.xlabel('Nanowire width (nm)')
pylab.ylabel('Nanowire band gap (eV)')
pylab.tight_layout()
pylab.savefig('nw_gaps.png')
pylab.show()
```

`nw_gaps.py`

Running the script produces the plot shown below. The nanowire band gap increase significantly for thicknesses of 1 nm or less, also also found in Ref.[Page 1697, 1].



References

3.12.4 Linear response current – how to compute it, and why it is often not a good idea

Version: 2016.2

In this tutorial, you will learn a simple way to calculate the **linear response current** and how it is different from the self-consistent result, the latter of which is necessary to exhibit **negative differential resistance** (NDR).

The question sometimes arises, why it is necessary to compute the I-V curve using finite-bias NEGF calculations – cant one just use the zero-bias transmission spectrum (which is much cheaper to compute) and calculate the finite-bias current from that? This is indeed possible, but the result – the linear response current – is in most cases radically different from the current evaulated from a finite-bias self-consistent solution. The reason is fairly simple: **The transmission spectrum is bias-dependent**.

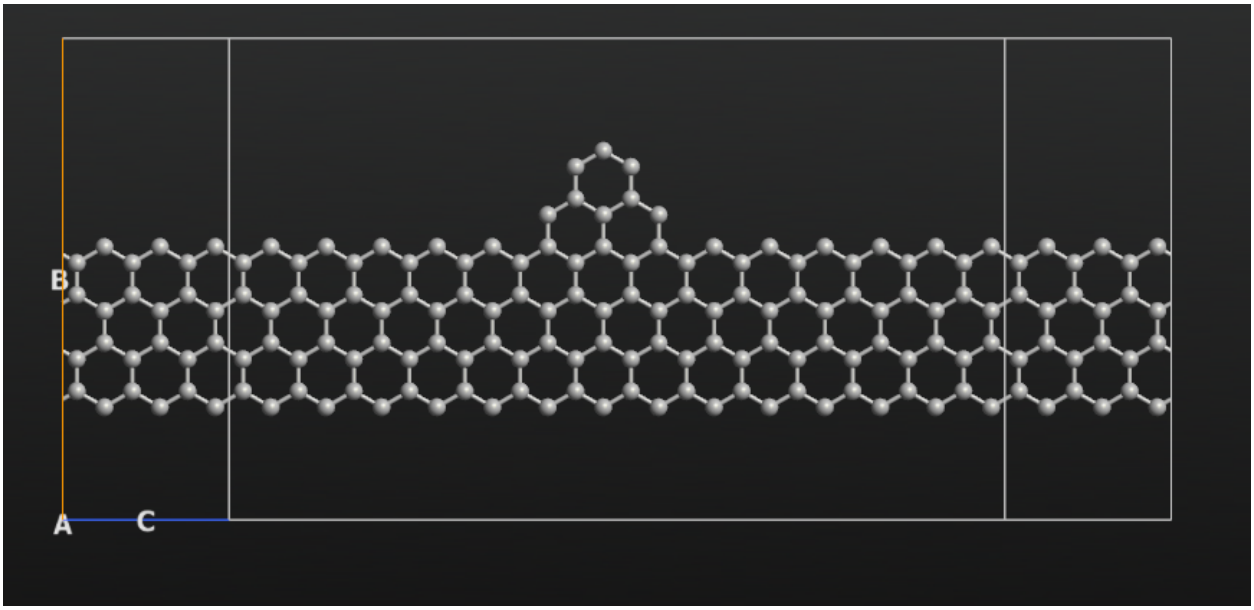
Important: In the linear response case, it is not possible to obtain negative differential resistance, since the current can only increase as the bias window opens up and you integrate an increasingly large part of the transmission spectrum. NDR is an important effect for applications, and is observed in a wide range of cases, but can only be obtained using a self-consistent finite-bias NEGF calculation.

Here well will demonstrate with a relatively simple example how to obtain the linear response current, and how it is different from the self-consistent result. The inspiration is taken from a recent work performed with QuantumATK by Yipeng An *et al.*[1]; in the article the authors used DFT, but we will consider the same system (specifically, the GE1 geometry) using a simple Slater-Koster tight-binding π-model.

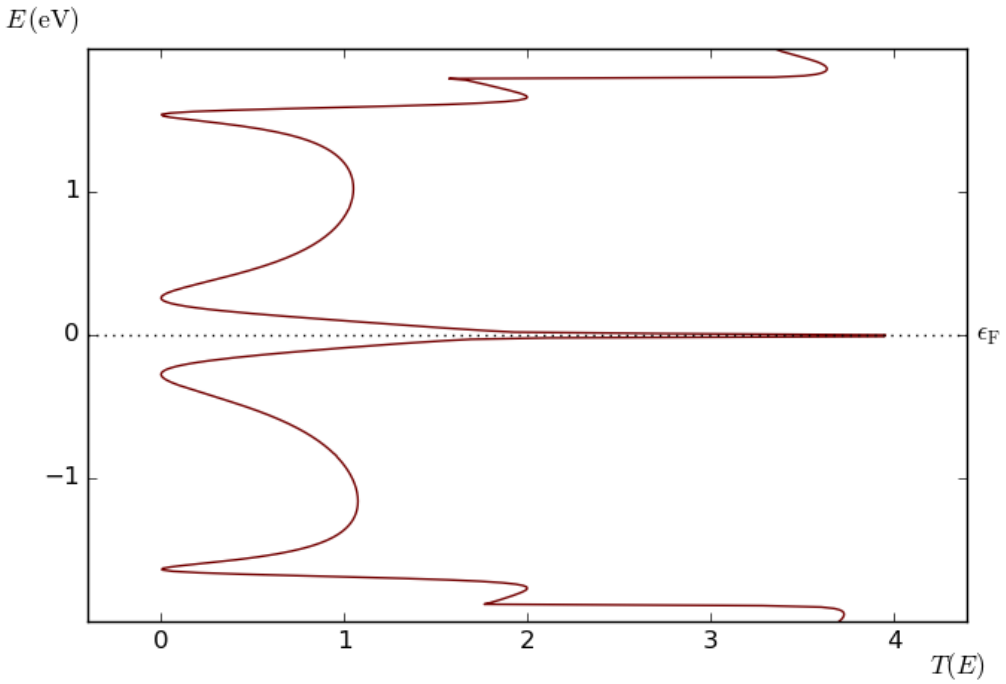
The geometry is shown below. We will not go into the details of how to create it; its farily easy, using the **Builder**:

Create a large graphene sheet and remove the atoms you dont need. Note that for the Slater-Koster model we dont use any terminating hydrogen atoms.

[1] Yipeng An, Wei Ji, and Zhongqin Yang. Z-like Conducting Pathways in Zigzag Graphene Nanoribbons with Edge Protrusions. *JPC C*, 116(9):5915–5919, 2012. doi:10.1021/jp3003646.



We can now compute the zero-bias transmission spectrum of this device. Use a pre-made QuantumATK Python script for this: `zero_bias.py`. The script takes less than a minute to run, even if there are 120 atoms in the central region. The resulting transmission spectrum is shown below (select the `Sum` spin channel in the *Curves* drop-down menu). The spectrum exhibits a pronounced peak at the Fermi level, which is to be expected since the electrodes are zigzag graphene nanoribbons.



At this point, we can compute the linear response I-V curve by integrating the transmission spectrum in an increasingly wide bias interval. QuantumATK supports this calculation via a built-in method on the *TransmissionSpectrum* object. Normally when you ask for the current,

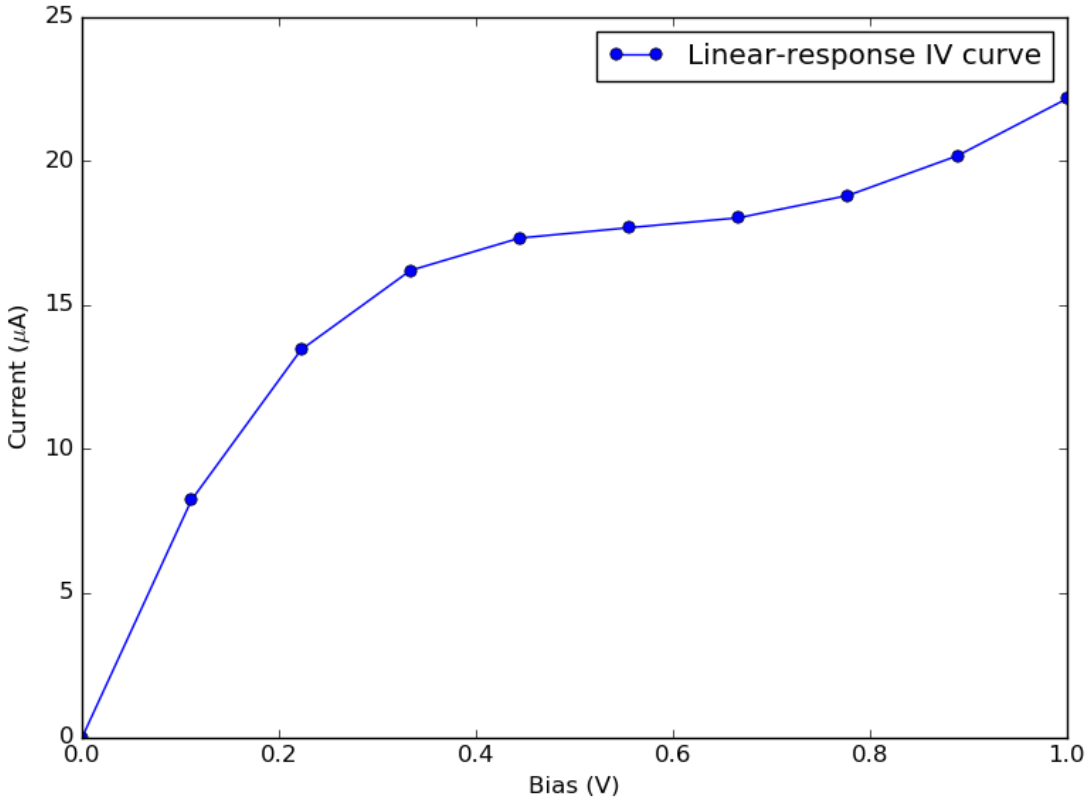
```
current = transmission_spectrum.current()
```

it will be evaluated for the bias that was used when computing the transmission spectrum. However, it is also possible to specify another set of electrode voltages, which will then be used instead. Thus, if the transmission spectrum, as in our case, was computed for zero bias, you can use

```
current = transmission_spectrum.current(electrode_voltages=[-0.5,0.5]*Volt)
```

to get the linear response current at 1 V bias. You can therefore extract an entire linear response I-V curve by looping over a range of bias points and compute the linear response current at each bias.

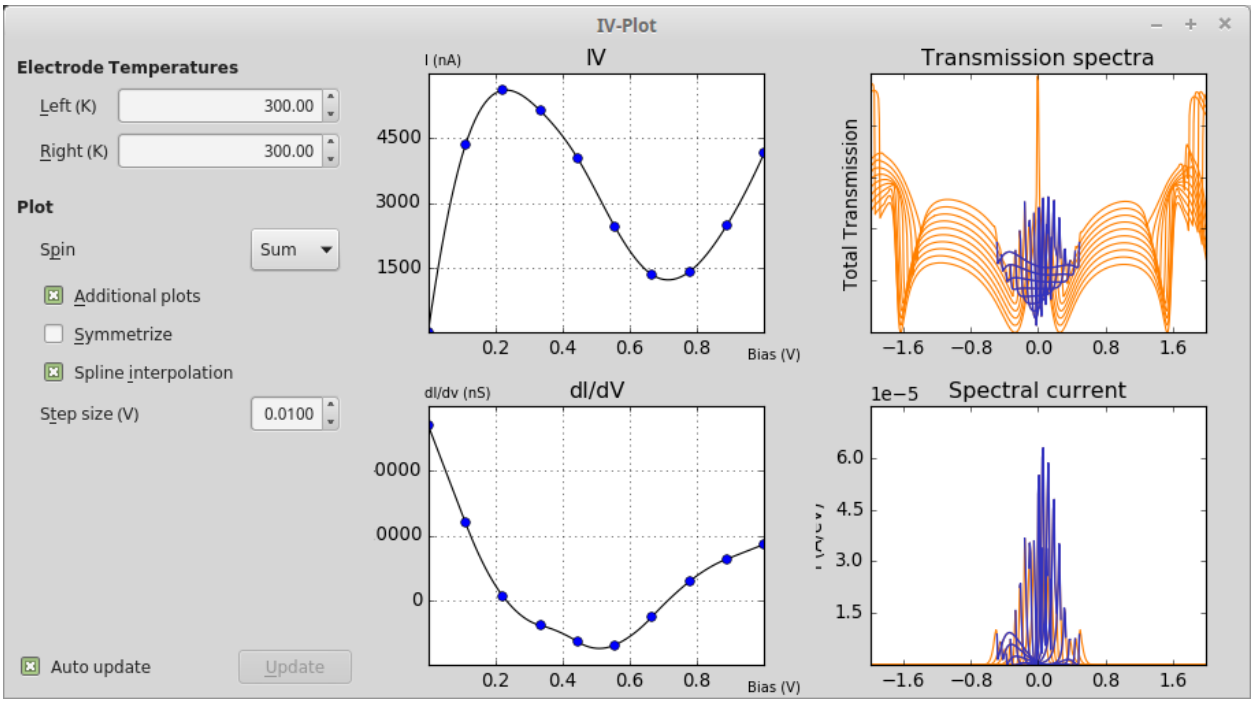
Download the script `linear_response.py` and execute it. The script computes and plots the linear response I-V curve. The result is shown below.



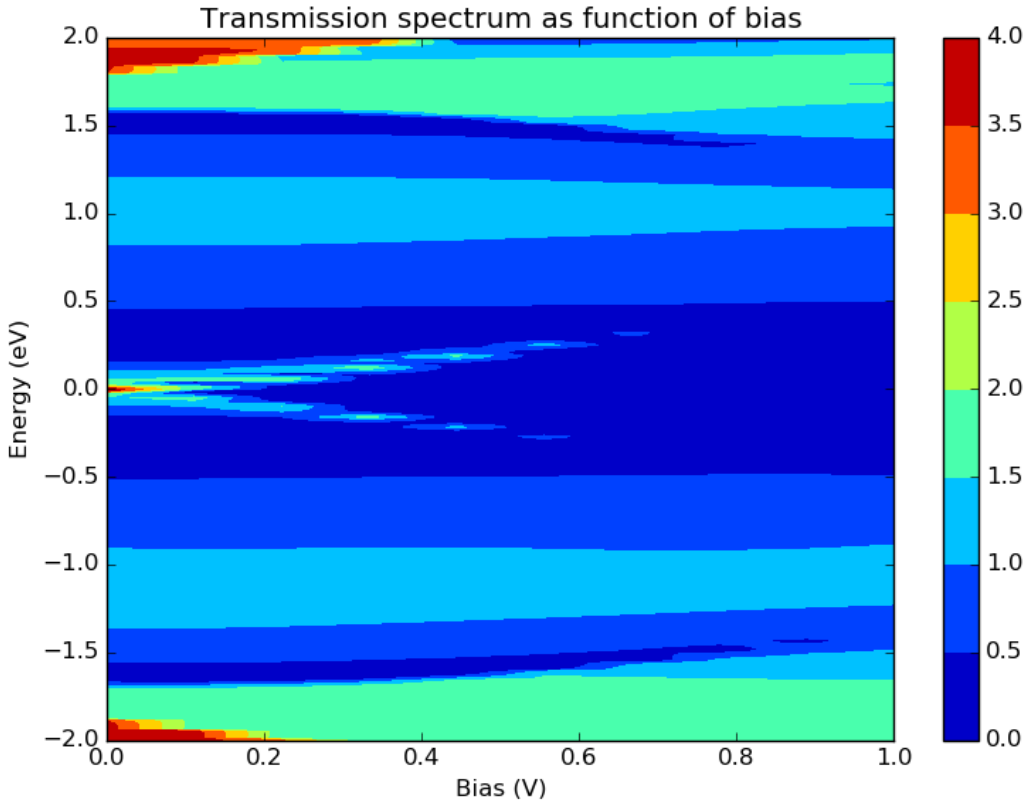
As expected, the current rises monotonically with the voltage – this is the only possible I-V curve that can be obtained this way, since we are integrating a non-negative function (the transmission spectrum) over an increasingly broad interval.

However, we know from[Page 1711, 1] that this system exhibits negative differential resistance, so something is not correct. Therefore, we will next compute the self-consistent transmission spectrum for each bias. Use this script: `iv.py`. It should take less than 5 minutes to run on a modern PC.

The script uses the `IVcurve` analysis object. From the QuantumATK LabFloor, select the generated **IVCurve** item and click the **IV-Plot** plugin from the right-hand plugins bar to visualize the results. Tick the boxes named *Additional plots* and *Spline interpolation*.



In this case we do indeed see NDR – the differential current is shown in the lower-left plot, and it does become negative – and from the upper-right plot of the transmission spectra we can see why. The individual transmission spectra are strongly bias-dependent. To illustrate this point even more clearly, we generate a contour plot of the transmission spectra as a function of energy and bias, using a simple script: `transmission_contour.py`.



Clearly, the assumption of a bias-independent transmission spectrum is invalid in this system, and as mentioned above indeed in most device configurations.

References

3.12.5 Make Movies from QuantumATK Trajectory Files

ATK: gif animations Platforms: Prerequisites: Know how to create NEB or MD structures in QuantumATK `NEBmovie.hdf5` *In this tutorial you will learn how to make nice movies from Trajectory, NudgedElasticBand (NEB) or MDTrajectory objects.*

Warning: You must already have an existing trajectory file.

Here we report the steps to follow to make a movie from an existing trajectory file. The example is a NEB calculation for a buckled graphene nanoribbon (GNR), but it works exactly the same way for optimization and MD trajectories. There are other tutorials describing how to create and calculate NEB trajectories. Also note that you can also make movies for trajectories computed with other software packages, like VASP, as long as they can be imported.

Creating Animated GIF

You can import your NEB object (here we use the NEB object in `NEBmovie.hdf5`) by simply dragging and dropping the file on the **Viewer**. There you can decorate your structure as you prefer by clicking on the **Properties** button in the right-hand panel. Left-clicking on the structure you have the following four extraction options:

NEBmovie.hdf5

You can download the NEB file from here: `NEBmovie.hdf5`.

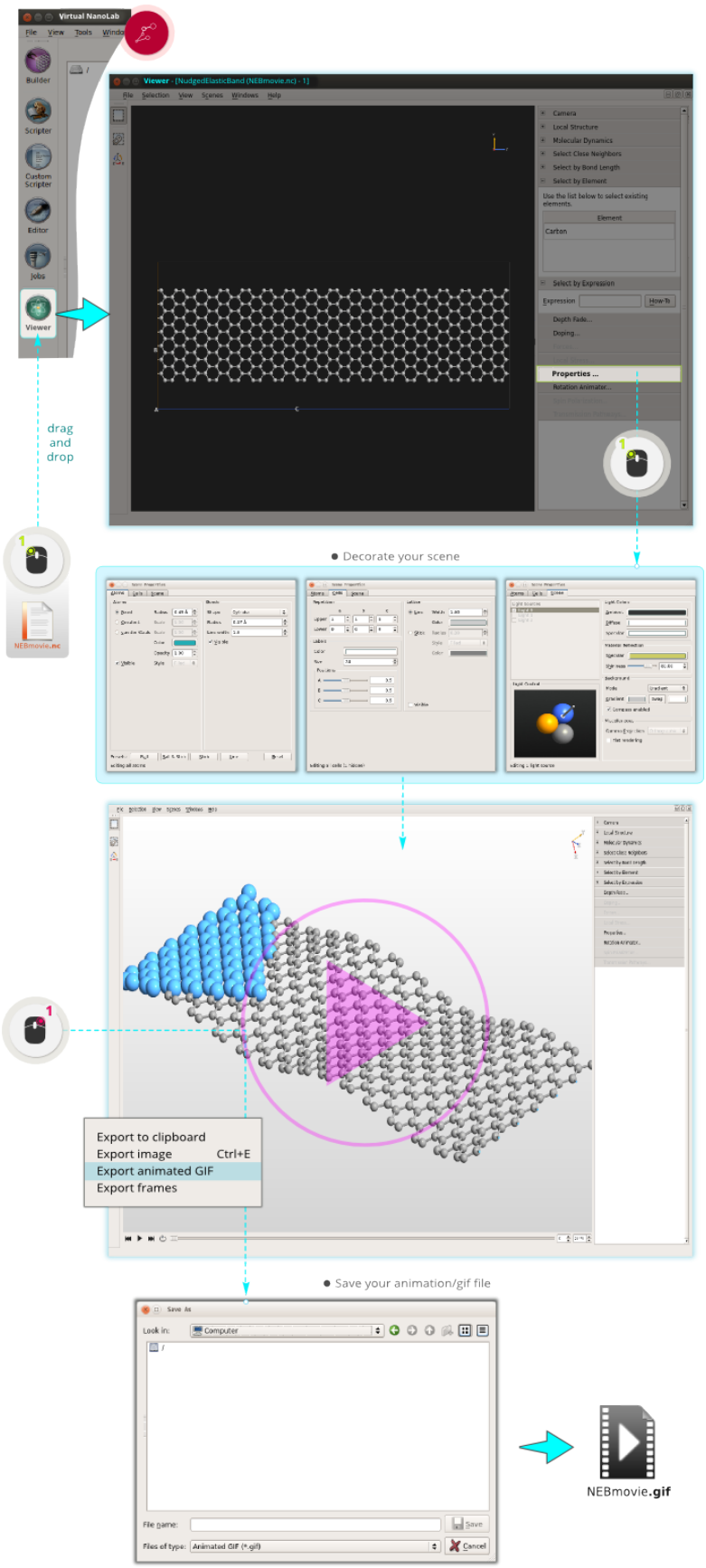
•Export to clipboard

•Export image

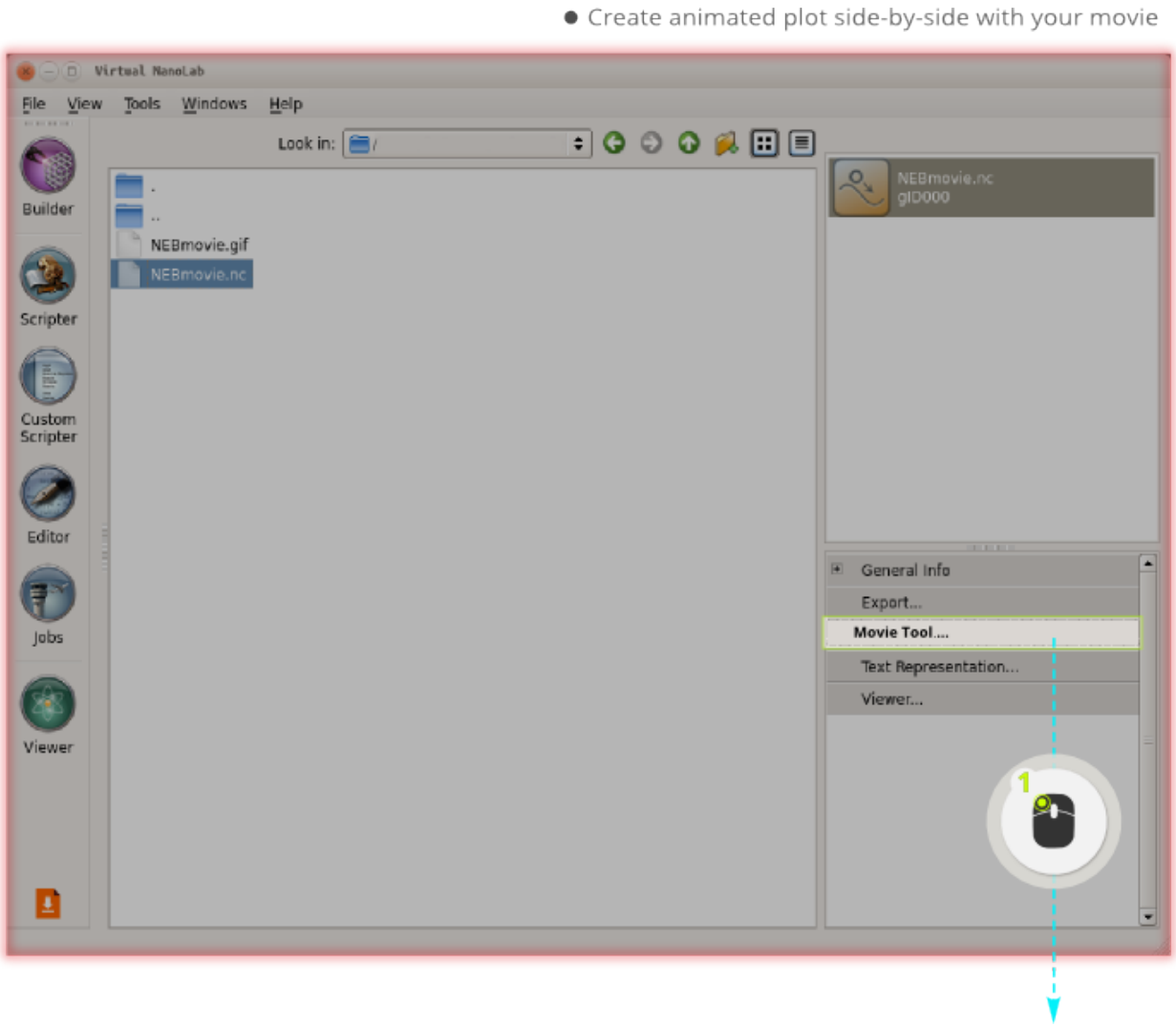
•**Export animated GIF**

•Export frames

Use the following workflow for exporting an animated GIF:

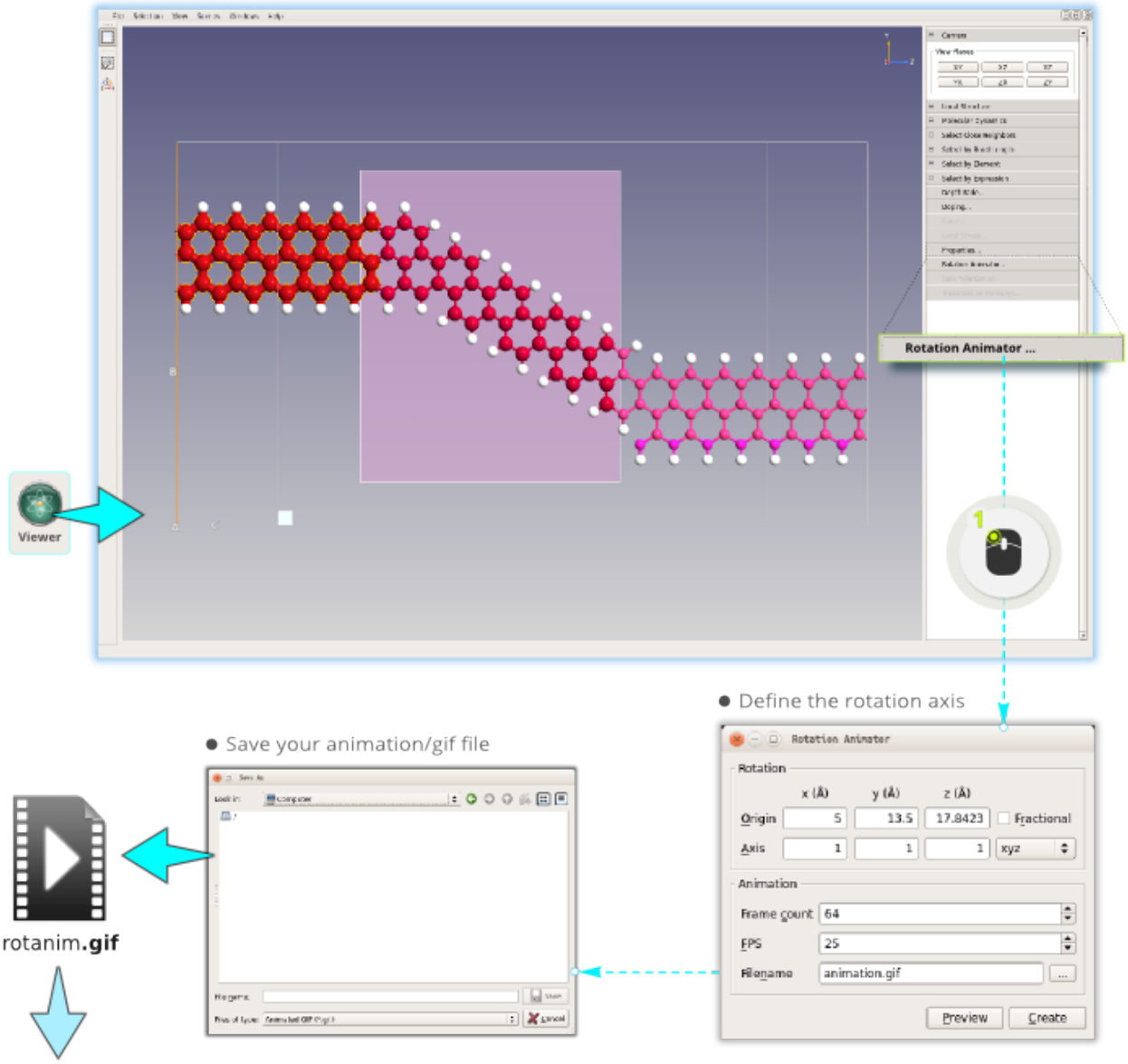

Virtual NanoLab
Builder
Scripter
Custom Scripter
Editor
Jobs
Viewer
drag and drop
Properties ...
Decorate your scene
NEBmovie.nc
Export to clipboard
Export image Ctrl+E
Export animated GIF
Export frames
Save your animation/gif file
Save As
Look in:
Computer
File name:
Save
Files of type:
Animated GIF (*.gif)
Cancel
NEBmovie.gif

The possibilites at this point are endless. If you analyze your NEB or MD simulation with the **Movie Tool**, for example, you can view an animated plot of the relevant quantity:



Rotation Animator

In QuantumATK you can also create a rotation animation (in `.gif` format) of a structure around a specific axis (GIF animation from a static configuration). Just got to *Viewer* → *Rotation Animator*. The functionality of the **Rotation Animator** is quite obvious and easy to understand:

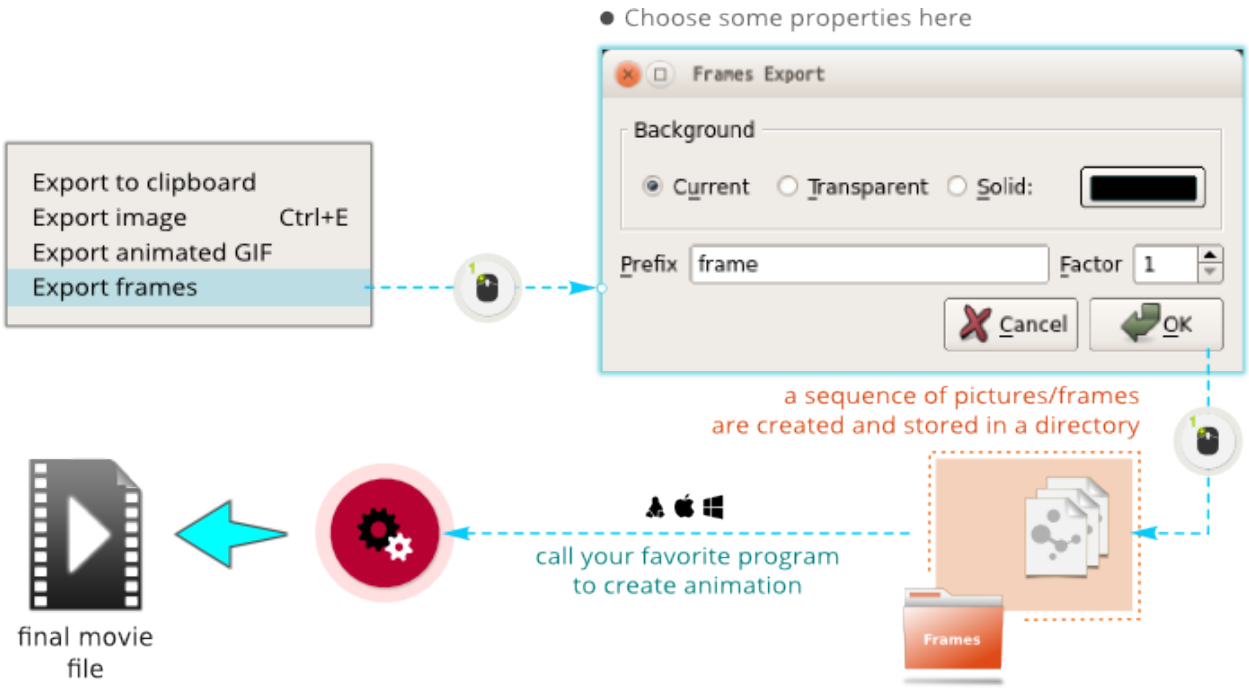


Movie from Trajectory Files

Combining QuantumATK with other (external programs) it is possible to create your animation from a sequence of pictures (in various formats like MP4, FLV, etc.).

Note: Export Frames

In QuantumATK you can extract the set of pictures that correspond to an animated structure. Just right click on the animation/structure in the Viewer and choose **Export frames**.



It may be wise to change the prefix name in case you will export another set of frames later; then you will not overwrite or intermix the image files. When you click **OK**, the movie will start to play in the 3D Viewer. While playing, each frame is actually exported as an image into a directory called **Frames** in your current project. Those images can be stapled together in a movie using some external software! See below for a brief description of making the movie using MPlayer on Linux.

Example: Create a movie with **MPlayer Platforms: Prerequisites:** Linux terminal and MPlayer **Links:** MPlayer *Here we give an example of how to build a movie/animation file using the MPlayer software from the Linux command-line.*

•Open a terminal and navigate to the directory with the frames. Here you will find the PNG files corresponding to your frames, and a file containing the command needed to create the movie (`encode`).

•An example of the encode file is shown below.

•You may run the encode file directly (*./encode*), or choose to modify some option first.

```
$ mencoder mf://GNR_buckle*.png -o atkFrames.avi -ovc lavc -lavcopts \
$ vcodec=msmpeg4v2:autoaspect:vbitrate=2160000:mbd=2:keyint=132:vqblur=1.
↪0:cmp=2:subcmp=2:dia=2:mv0:last_pred=3 \
$ -mf w=800:h=600:fps=5:type=png
```

Some options to consider changing are the aspect ratio (use -mf to set the width (w) and height (h) matching the size of the 3D Viewer window) and the frame rate (fps), and of course the filename of the final movie.

•Now you will have the file **atkFrames.avi** in the Frames directory. The movie should play in most players, but if in doubt, always look to VLC or MPlayer.

•To make an **FLV** file (smaller file size, suitable for online stuff), try:

```
$ mencoder mf://atkFram*.png -o atkFrame.flv -of lavf -oac mp3lame \
$ -lameopts abr:br=56 -srate 22050 -ovc lavc \
```

(continues on next page)

(continued from previous page)

```
$ -lavcopts vcodec=flv:vbitrate=500:mbd=2:mv0:trell:v4mv:last_pred=3 \
$ -mf w=512:h=512:fps=10:type=png
```

Have fun creating cool videos with QuantumATK!

Note: MPlayer is a cross-platform program and you can use it in Windows and Mac as well.

3.12.6 Converting lattices: Rhombohedral to hexagonal and back

Version: 2015.1

As we know, there are 14 Bravais lattices in $\mathbf{R}^3$ space. These determine the translational symmetry properties of a crystal unit cell, and thereby also the symmetry properties in reciprocal space (high-symmetry **k**-points, etc.). Additionally, the space group determines the rotational and other internal symmetry operations of the basis (the atomic coordinates). For the complete symmetry specification we need both, although the lattice symmetry is usually determined by the space group, i.e. each space group is associated with one and only one Bravais lattice.[1]

However, the trigonal crystal system is special, because one can often choose between rhombohedral and hexagonal representation. Trigonal structures found in online databases are often hexagonal, but for computational purposes, the smaller rhombohedral cell will usually be more efficient. In that case, conversion between the two representations is advisable.

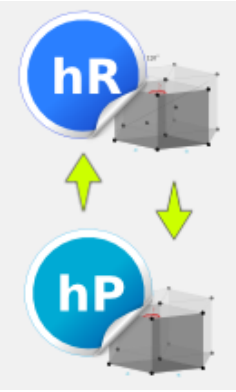
Note: hR (hexagonal) and hP (rhombohedral) crystal systems

The space groups are classified by crystal systems, and any standard book on solid state physics will tell you that there are **7 crystal systems** and **6 crystal families**. Here we focus on the special situation in the hexagonal family, which is the only one that can be subdivided into two crystal systems; the hexagonal and the trigonal.

The hexagonal system comprises 27 space groups. All of these have hexagonal Bravais lattices, labeled hP. The trigonal system is the tricky one, because its 25 space groups (143-167) belong either to the hexagonal (hP, 18 space groups) or the rhombohedral (hR, 7 space groups) Bravais lattice. Now, to make things a bit more confusing, all trigonal crystals with rhombohedral lattices (space groups 146, 148, 155, 160, 161, 166, and 167), can be represented as an equivalent hexagonal system; there is a choice of using a hexagonal or a rhombohedral representation.

Further information on this subject is available in the section *Crystal classifications*.

Conversion between hP and hR representations



[1] ITA: International Tables for Crystallography, Volume A: Space-Group Symmetry, 5th, revised edition, Springer 2005

The hexagonal setting is in fact a supercell with three irreducible rhombohedral units. In the modern so-called *obverse* representation, the origins of these three subsystems are placed at the $\boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3$ fractional coordinates of the hexagonal lattice.

Tip:

- $\{\boldsymbol{A}_H, \boldsymbol{B}_H, \boldsymbol{C}_H\}_{hP}$: hexagonal (**hP**) lattice vectors
- $\{\boldsymbol{A}_R, \boldsymbol{B}_R, \boldsymbol{C}_R\}_{hR}$: rhombohedral (**hR**) lattice vectors.

Taking into account the symmetries of the crystal structures we can find the following relations among the above vectors:

$$\begin{pmatrix} \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \\ \boldsymbol{d}_3 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 0 & 0 & 0 \\ 2 & 1 & 1 \\ 1 & 2 & 2 \end{pmatrix}\begin{pmatrix} \boldsymbol{A}_H \\ \boldsymbol{B}_H \\ \boldsymbol{C}_H \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}\begin{pmatrix} \boldsymbol{A}_R \\ \boldsymbol{B}_R \\ \boldsymbol{C}_R \end{pmatrix}$$

The lattice vectors are connected via the following transformation (direct and inverse):

$$\begin{pmatrix} \boldsymbol{A}_R \\ \boldsymbol{B}_R \\ \boldsymbol{C}_R \end{pmatrix} = \underbrace{\frac{1}{3}\begin{pmatrix} -1 & 1 & 1 \\ 2 & 1 & 1 \\ -1 & -2 & 1 \end{pmatrix}}_{:=\boldsymbol{M}}\begin{pmatrix} \boldsymbol{A}_H \\ \boldsymbol{B}_H \\ \boldsymbol{C}_H \end{pmatrix} \Leftrightarrow \begin{pmatrix} \boldsymbol{A}_H \\ \boldsymbol{B}_H \\ \boldsymbol{C}_H \end{pmatrix} = \underbrace{\begin{pmatrix} -1 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & 1 \end{pmatrix}}_{:=\boldsymbol{M}^{-1}}\begin{pmatrix} \boldsymbol{A}_R \\ \boldsymbol{B}_R \\ \boldsymbol{C}_R \end{pmatrix}$$

Choosing now Cartesian coordinates one obtains the following relations:

$$\text{hP:} \begin{cases} \boldsymbol{A}_H = a_H(1, -\sqrt{3}, 0)/2 \\ \boldsymbol{B}_H = a_H(1, +\sqrt{3}, 0)/2 \\ \boldsymbol{C}_H = c_H(0, 0, 1) \end{cases}, \text{ hR:} \begin{cases} \boldsymbol{A}_R = (0, +a_H\sqrt{3}, c_H)/3 \\ \boldsymbol{B}_R = (3a_H, -a_H\sqrt{3}, 2c_H)/6 \\ \boldsymbol{C}_R = (-3a_H, -a_H\sqrt{3}, 2c_H)/6 \end{cases}$$

Warning: The orientation of the last cell with respect to the cartesian axes is different from the standard QuantumATK choice for the rhombohedral cell!

For later use, let us compute the lattice parameters of the rhombohedral cell, i.e. the lengths of the lattice vectors (a_H, c_H, a_R) and their angles (α_R, β). They are all the same, as expected for a rhombohedral crystal, and after some algebra we obtain

$$\begin{cases} a_R = a_H\beta/3 \\ \alpha_R = 2\arcsin(3\beta/2) \\ \text{where: } \beta := \sqrt{3 + (c_H/a_H)^2} \end{cases} \Rightarrow \begin{cases} a_H = 2a_R\sin(a_R/2) \\ c_H = a_R 3\sqrt{1 + 2\cos a_R} \end{cases}$$

Now, what to do in practice, if you have recevied a rhombohedral crystal in the hexagonal supercell representation, but want to convert it to the smaller rhombohedral system? We give the recipe in the next section.

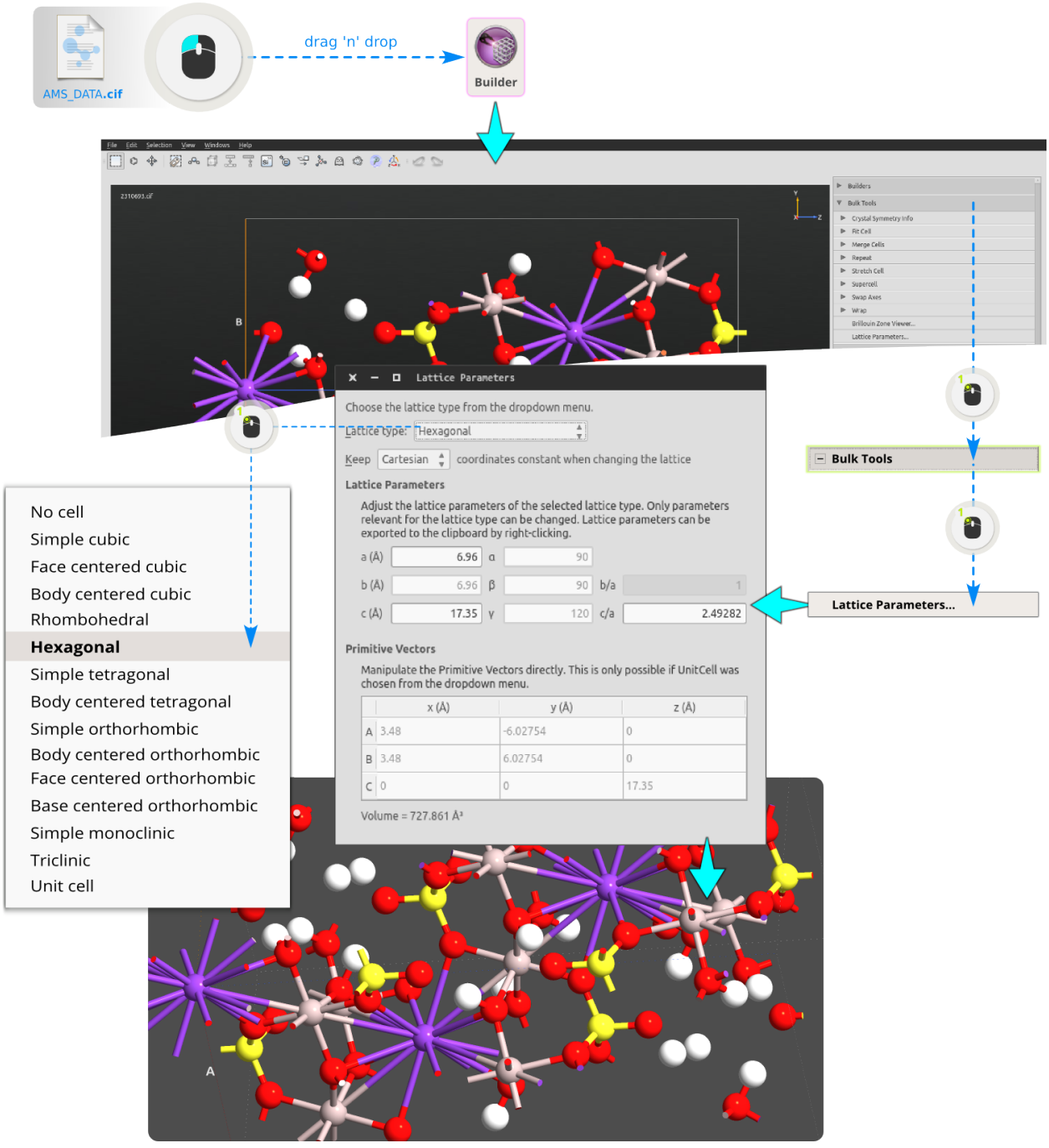
Converting hP supercell to hR primitive cell

Assume you have found a structure in some online database: Hilgardite, $KS_2Al_3O_{14}H_6$. Download the CIF file and save it: `AMS_DATA.cif`.

- **Step 1:** Drag and drop (or use *Add* → *From Files*) the `AMS_DATA.cif` file on the **Builder** .

- **Step 2:** You can read off the lattice parameters via *Bulk Tools* → *Lattice Parameters* (or you can read them in the CIF file):

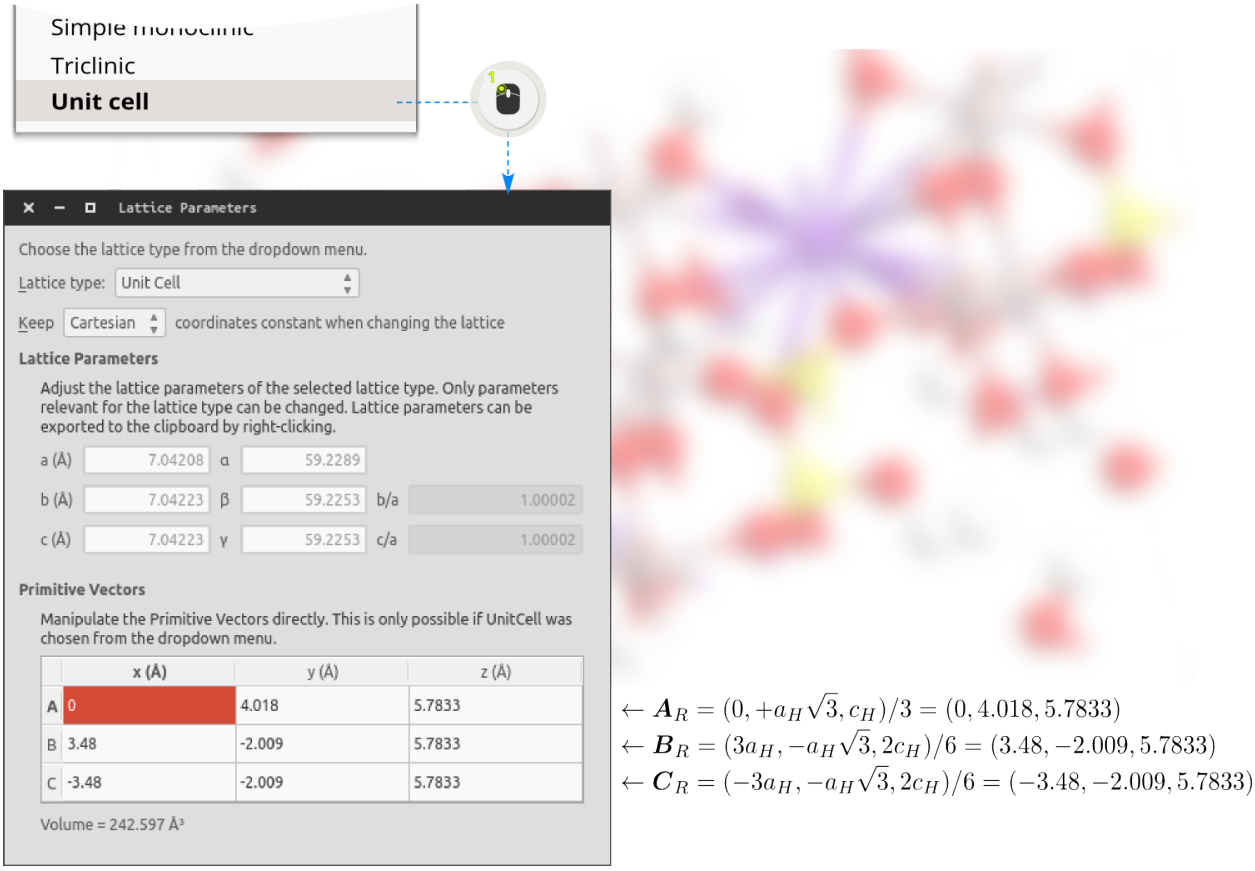
$$a_H = 6.96 \text{ Angstroms}$$
$$c_H = 17.35 \text{ Angstroms}$$



•**Step 3:** To change the lattice to rhombohedral (hR), start by changing the Lattice type to Unit cell, and select to Keep Cartesian coordinates constant when changing the lattice. The rhombohedral primitive vectors are given by the relations from above and inserting the hexagonal lattice constants, you obtain:

$$\text{hR:} \begin{cases} \boldsymbol{A}_R = (0, +a_H\sqrt{3}, c_H)/3 \\ = (0, 4.018, 5.7833) \\ \boldsymbol{B}_R = (3a_H, -a_H\sqrt{3}, 2c_H)/6 \\ = (3.48, -2.009, 5.7833) \\ \boldsymbol{C}_R = (-3a_H, -a_H\sqrt{3}, 2c_H)/6 \\ = (-3.48, -2.009, 5.7833) \end{cases}$$

Insert these numbers, component by component, in the Primitive Vectors table. (At this point you will see that the K atoms sit close to the corners of the unit cell, indicating that we have indeed made a reasonable transformation.)

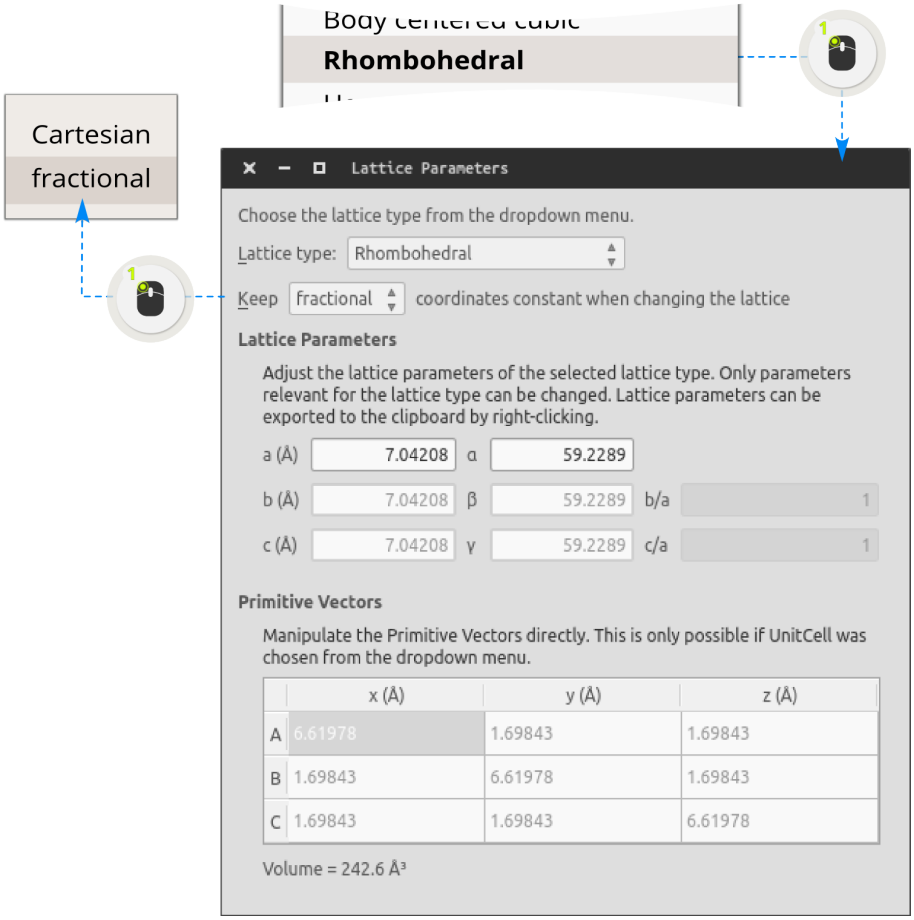


$\leftarrow \boldsymbol{A}_R = (0, +a_H\sqrt{3}, c_H)/3 = (0, 4.018, 5.7833)$
$\leftarrow \boldsymbol{B}_R = (3a_H, -a_H\sqrt{3}, 2c_H)/6 = (3.48, -2.009, 5.7833)$
$\leftarrow \boldsymbol{C}_R = (-3a_H, -a_H\sqrt{3}, 2c_H)/6 = (-3.48, -2.009, 5.7833)$

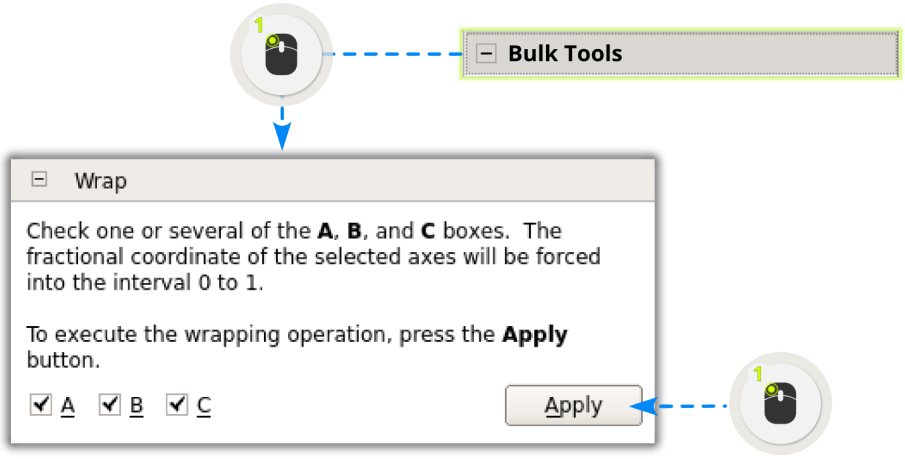
- **Step 4:** As mentioned above, the orientation of this new cell with respect to the Cartesian axes is different from the standard QuantumATK choice for the rhombohedral cell. The next step is therefore to **reorient the cell to fit the standard cell**.
- To do this, switch to Keep fractional coordinates constant when changing the lattice. This ensures we do not distort the geometry while reorienting the cell.
- Change the Lattice type to Rhombohedral. This operation preserves the lengths of the primitive vectors, but aligns them into the standard orientation (and also uses the Rhombohedral lattice class, so you have access to the relevant symmetry points for band structure calculations, etc). As a verification of this, you can compute the lattice constants by hand:

$$\begin{aligned} \beta &= 3.03548 \\ a_R &= a_H\beta/3 = 7.0423 \text{ Angstroms} \\ \alpha_R &= 2\arcsin(3\beta/2) = 59.228^\circ \end{aligned}$$

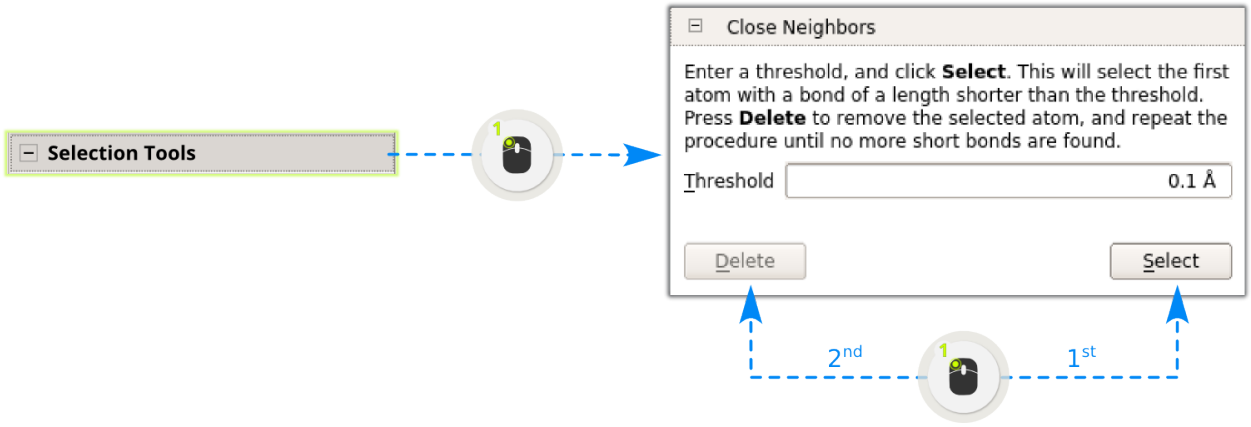
which is close to the values shown in the Lattice Parameters widget (some rounding error will occur due to the number of decimals used for the lattice vector components).



•**Step 5:** There are still too many atoms (three times too many). These extra atoms are however all sitting in symmetry equivalent positions of the rhombohedral lattice, and thus by **wrapping them into the cell**, they will fall on top of the minimal basis. Thus, go to *Bulk Tools* → *Wrap*, ensure all directions A, B, C are ticked, and click **Apply**.

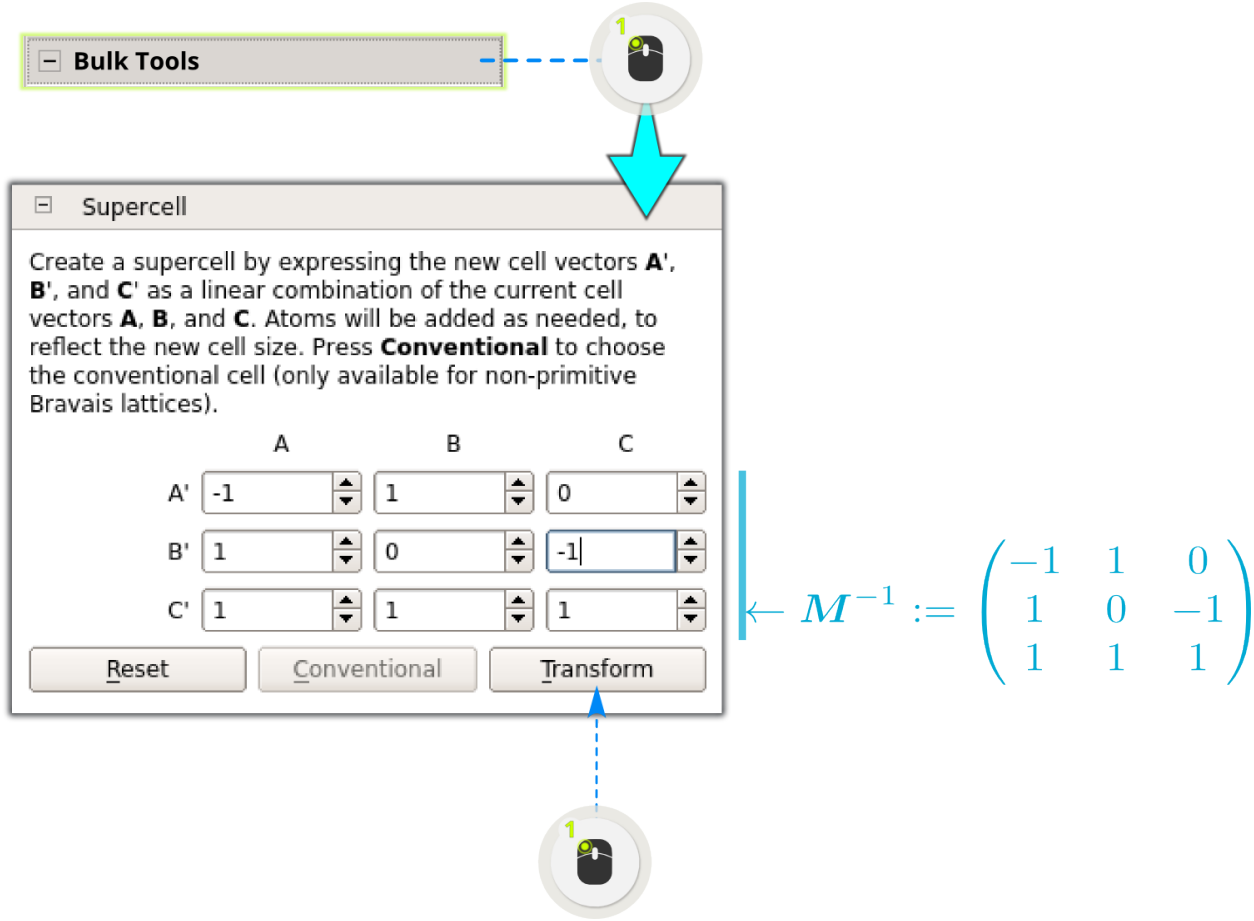


•**Step 6:** Now it may appear as if magically the superfluous atoms were removed. However, they are still there - there are atoms on top of each other now. Such a configuration is not valid for calculations, so the final step is to remove the equivalent atoms. For this, use *Select* → *Close Neighbors*, and click the buttons **Select** and **Delete**.



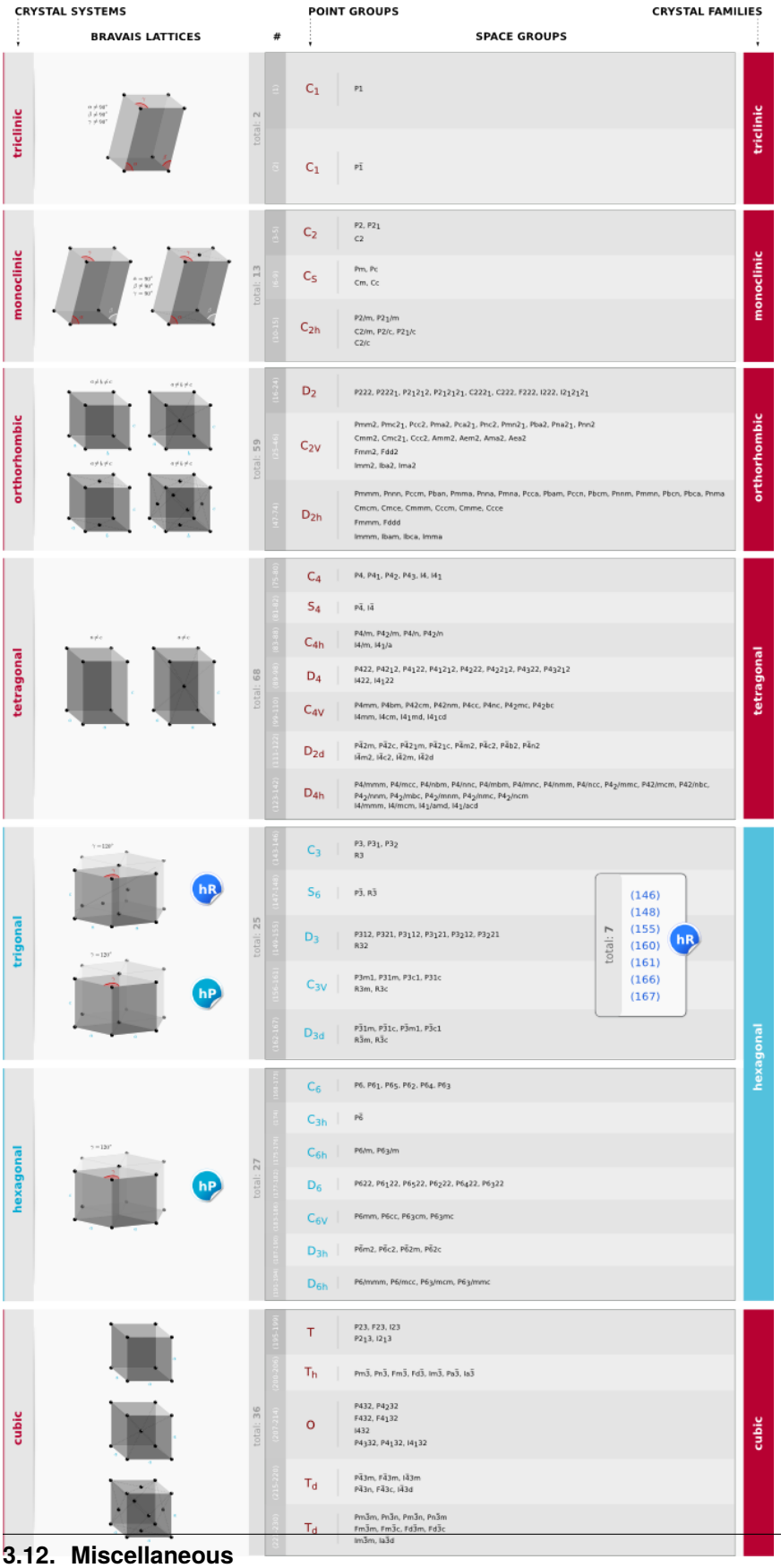
Inverse case

You can of course also convert the other way, i.e. to the hexagonal supercell from the rhombohedral minimal cell. This is simpler. Looking above, we see that the hP vectors are obtained by hR after the action of the matrix M^{-1} as defined above. This matrix can be implemented **directly in ATK** by choosing *Bulk Tools* → *Supercell* and enter this transformation:



You now have a hexagonal cell, its just not oriented in the usual way, but this is easily fixed just like before: Use *Tools* → *Lattice Parameters*, choose to keep fractional coordinates constant, then change the Lattice type to Hexagonal. The structure is now back to the originally imported file.

Crystal classifications



References

3.12.7 Reusing electrodes in device calculations

A typical study of electronic devices with QuantumATK involves systematically studying the influence of various modifications of the device region. The variation can be atomic positions or types of dopants/vacancies in a semiconductor wire, or a different molecule attached to a nanotube, etc. In any case, the electrodes often remain unchanged. One can therefore calculate the electronic structure of the electrodes only once, and then reuse them in other calculations to save computational time.

In this tutorial you will therefore learn how to reuse the devices electrodes for different calculations, thus saving time by only having to calculate the converged state of the electrodes once.

Note: **Methodology** An QuantumATK device calculation is usually a two-step process: First the self-consistent states of the electrodes are calculated, and then the actual device calculation is done. Any post-SCF analysis and other operations are done after those two steps.

Reusing the initial electrode calculations is done like this: Starting out from a Python script designed for a normal device calculation as outlined just above, you split that calculation into two scripts: One script that calculates and saves the self-consistent state of the electrodes, and another script that reuses those electrodes and starts directly on the device calculation.

Warning: **Calculator settings** It is important that the parameters used for the electrode calculators are the same as those that will be used in the final device calculation, or at least very similar. Otherwise the electrode Fermi level alignment vs. the device average Fermi level may change, which can cause problems for SCF convergence.

Therefore, first make a decision on the numerical accuracy parameters, such as exchange–correlation functional, temperature, mesh cut-off, basis set, etc., and then stick to these for the whole sequence of calculations.

Moreover, it is absolutely necessary that the electrode and device **k-point sampling** match each other in the A and B directions! You can not use 3 k-points along A in the device but 7 k-points along A in the electrodes.

Separate scripts for electrodes and device

A bit of scripting is required, but mostly copying/pasting existing lines of Python code:

1. Start by setting up the full device calculation as you would normally with some representative central region. Use the Script Generator for this task, and make some good chices for the numerical accuracy parameters to be used.
2. Save two copies of the script: `electrode.py` and `device.py`.
3. Open `electrode.py` in the Editor and scroll down to where the `device_configuration` is defined. It usually looks like illustrated below. **Remove** those lines.

```
device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode]
    )
```

Tip: The script `electrode.py` is only used to compute the electrodes. You may want to remove the definition of the

central region altogether to keep things tidy, but its not strictly necessary.

4. Now scroll down towards the end of the script, and remove everything that has to do with the device calculator. That is, **remove** all lines from the following comment lines to the end of the script:

```
#----------------------------------------
# Device Calculator
#----------------------------------------
```

5. **Add** instead the following lines at the end of the script, and save the script as `electrodes.py`.

```
left_electrode.setCalculator(left_electrode_calculator)
left_electrode.update()
nlsave('left_electrode.nc', left_electrode)
right_electrode.setCalculator(right_electrode_calculator)
right_electrode.update()
nlsave('right_electrode.nc', right_electrode)
```

Note: In the case of identical electrodes, delete the three lines involving the right_electrode, and remove the left prefix from all the remaining variables:

```
electrode.setCalculator(electrode_calculator)
electrode.update()
nlsave('electrode.nc', electrode)
```

6. **Run** the electrode script, e.g. Job Manager .

7. Now start modifying the script `device.py`. Starting from the top, remove all the lines involving the left and right electrode configurations, i.e. **delete** all lines until (but not including)

```
# ------------------------------------------------------------
# Central region
# ------------------------------------------------------------
```

8. Instead, **add** the following lines at the top of the script, which will read the saved electrodes from file:

```
left_electrode = nlread("left_electrode.nc")[0]
right_electrode = nlread("right_electrode.nc")[0]
```

Hint: In the case of identical electrodes, if only a single electrode calculation was executed and saved, in e.g. `electrode.nc`, use this file for loading *both* electrodes.

9. It is recommended, but not strictly necessary, to also remove the definition of the electrode calculators, including the lines in the device calculator:

```
electrode_calculators=
    [left_electrode_calculator, right_electrode_calculator],
```

Thats it! You have successfully separated the electrode and device calculation into two scripts. When the electrode calculation is done, you can run `device.py`, and see that it completely bypasses the electrode calculation.

Tip: Whats next? For each new central region geometry to be computed with these electrodes, just replace the definition of the `central_region` in `device.py`, and run the device calculation. Make sure to check the device parameters such as bias voltage, etc.

Alternatively, you can apply steps 7–9 above to any complete device calculation script, keeping the points about the numerical accuracy parameters in mind.

3.12.8 Initialize from a converged state

Version: 2016.0

In this tutorial, you will learn how to save time in calculations by using a previously converged calculation as the starting point of a new calculation. This is useful, for example, when scanning the lattice constant, applied bias, or checking the convergence of various parameters, like k-point or mesh cut-off.

In many calculations you can save a lot of time by using a converged calculation as a starting guess for another calculation. In fact, for finite bias calculations of complicated transport systems, it may sometimes be the only way to obtain stable convergence.

Introduction

The default start guess for a self-consistent calculation in **ATK** is based on neutral atoms, i.e. the basis orbitals are populated as if each atom was isolated in space. This can, in many cases, be quite far from the real state of the atoms, even in a simple crystal; so it seems obvious that you could save a lot of time by providing a better starting guess. Similar systems, with only one parameter slightly different, will commonly be a better starting guess of a calculation and is especially useful in the following scenarios:

- Looping over a particular variable to see how much the results change. Therefore, we can apply this method to obtain good convergence at finite bias, which is done automatically when using the IVCurve object
- Checking convergence in various parameters, like k-points or mesh cut-off.
- Determining equilibrium lattice constant for a bulk system, or a coordinate for one or several atoms.
- As a special case: to initialize the anti-parallel configuration in a magnetic tunnel junction.

Since the initial guess is applied to the density matrix, it is a requirement that the original configuration/calculation (to be used to initialize the new one) and the new one have exactly the same sets of basis orbitals. Thus, the two configurations must have:

- the same atoms, element by element and in the same order
- the same basis set, atom by atom.

The atoms do not have to be in the same geometric positions, but the electronic density will not be adjusted to accordingly, so if the atoms have moved too far between configurations, the initial guess is likely to be worse than using neutral atoms.

The way to initialize one calculation from another is to add a keyword *initial_state* to the *setCalculator()* method on a configuration. The initial state object is typically read from a HDF5 file, or a variable in the script if you are looping over e.g. k-points or bias. We will also show how to do this using the **QuantumATK** GUI.

It is probably easiest to understand how to do this by looking at a couple of examples.

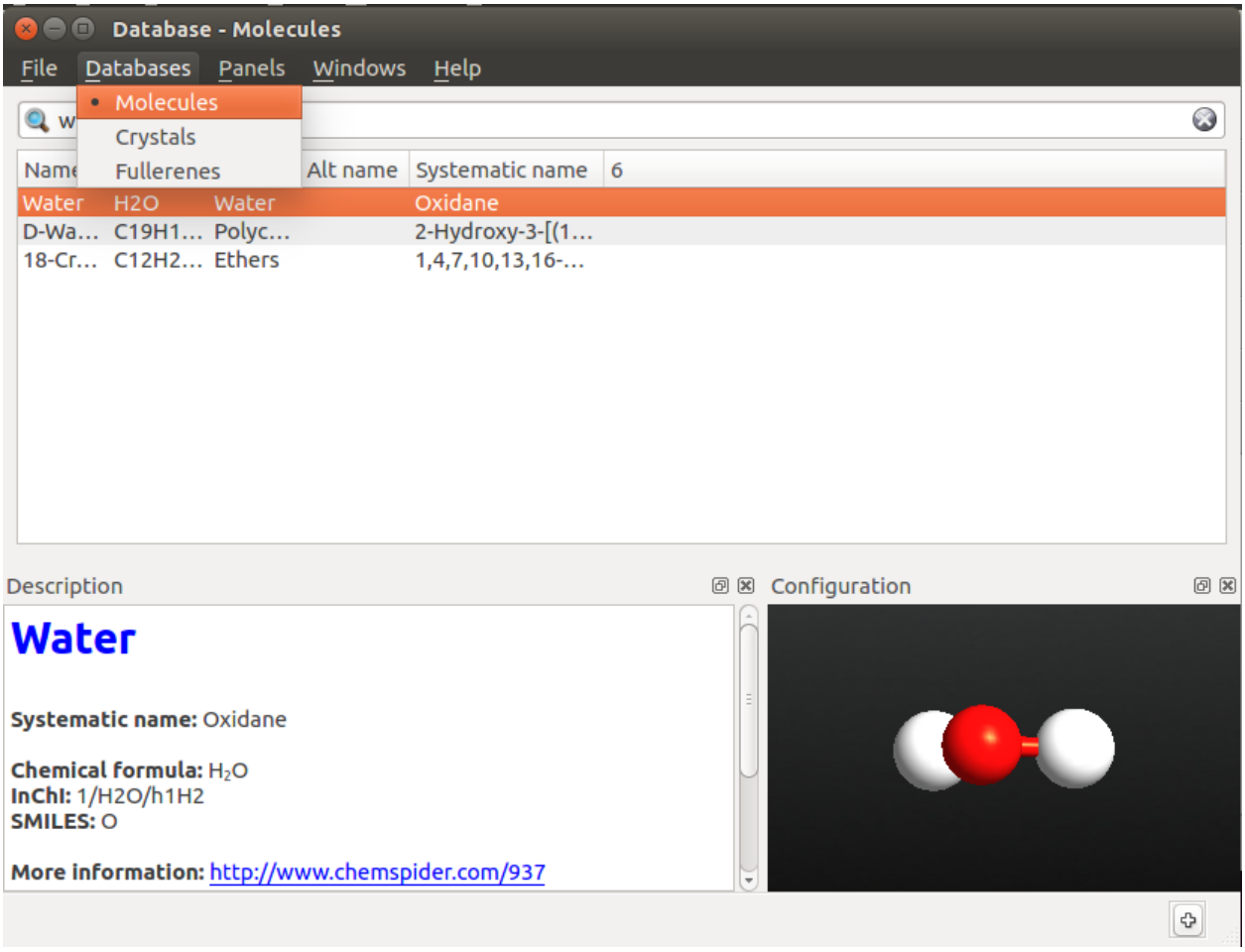
Examples

Water molecule example

We will demonstrate how to save computational time for a high accuracy calculation by doing it in two steps, for a simple case of a water molecule.

Calculation with low accuracy:

•Open the **Builder** and add water from *Database* → *Molecule*.



•Send the structure to the **Script Generator** with the **Send To** icon at the lower right corner.

•Add a **New Calculator** and change the *mesh cut-off* to 40 Hartree.

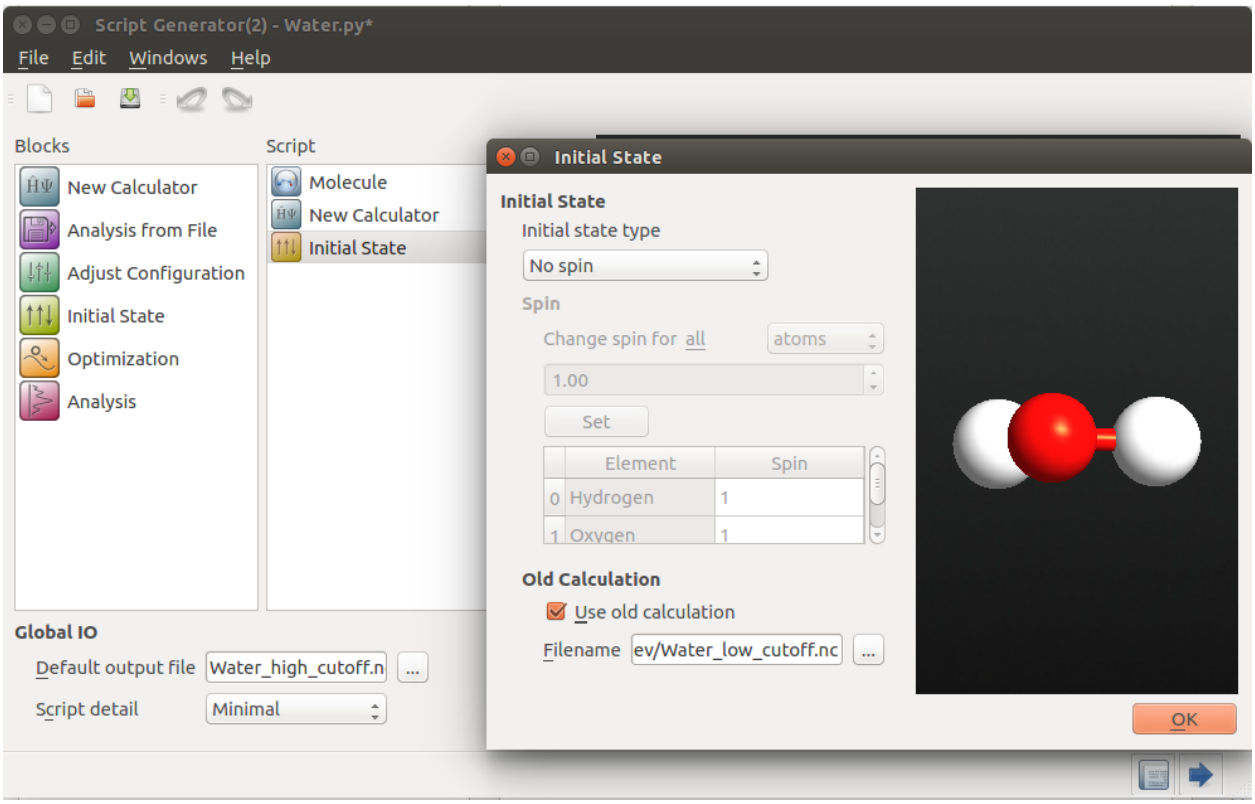
•Change the output name to `water_low_cutoff.hdf5`.

•Send the job to the **Job Manager** and run the calculation.

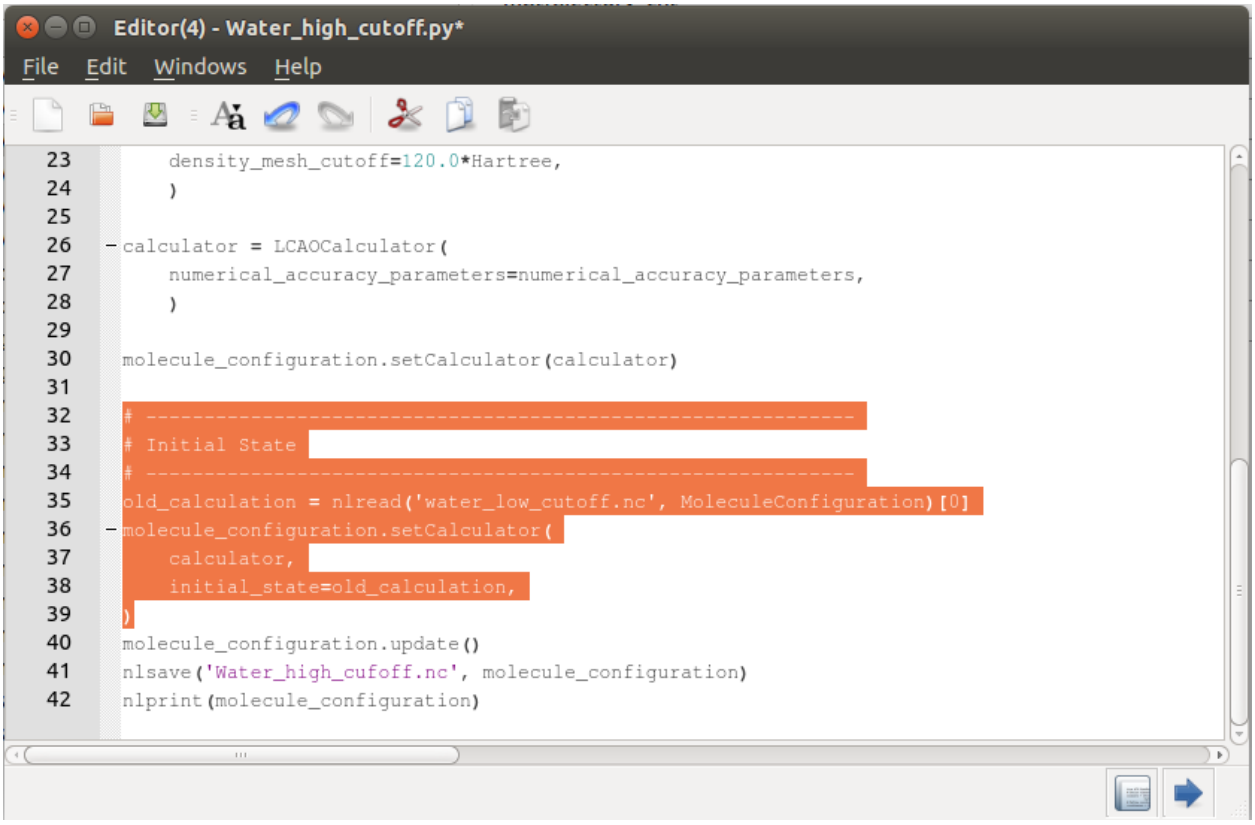
Calculation using the low accuracy result

•In the previous **Script generator** , open the **New Calculator** block and change the *mesh cut-off* to 120 Hartree.

•Add an **Initial State**. Tick the checkbox *Use old calculation*, and locate the file `water_low_cutoff.hdf5`.



This inserts an *initial_state* variable to *setCalculator()*, as mentioned above. Send the script to the **Editor** by the **Send To** icon at the lower right corner. You will find the lines in the script. Send it to the **Job Manager** and run the calculation.



You will find that the low accuracy calculation takes 6 self-consistent iterations to converge and the high accuracy calculation takes only a single iteration. If you run the high-accuracy calculation without initialization, it also needs 6 iterations. In this example, as the system is very small, there is not any time save by using this method, due to overhead from initialization etc. But for a realistic system, the high accuracy calculation could be much more time consuming than the low accuracy ones, saving time overall by using this method. We can save a lot of computational time by reducing the number of high accuracy iterations using this strategy.

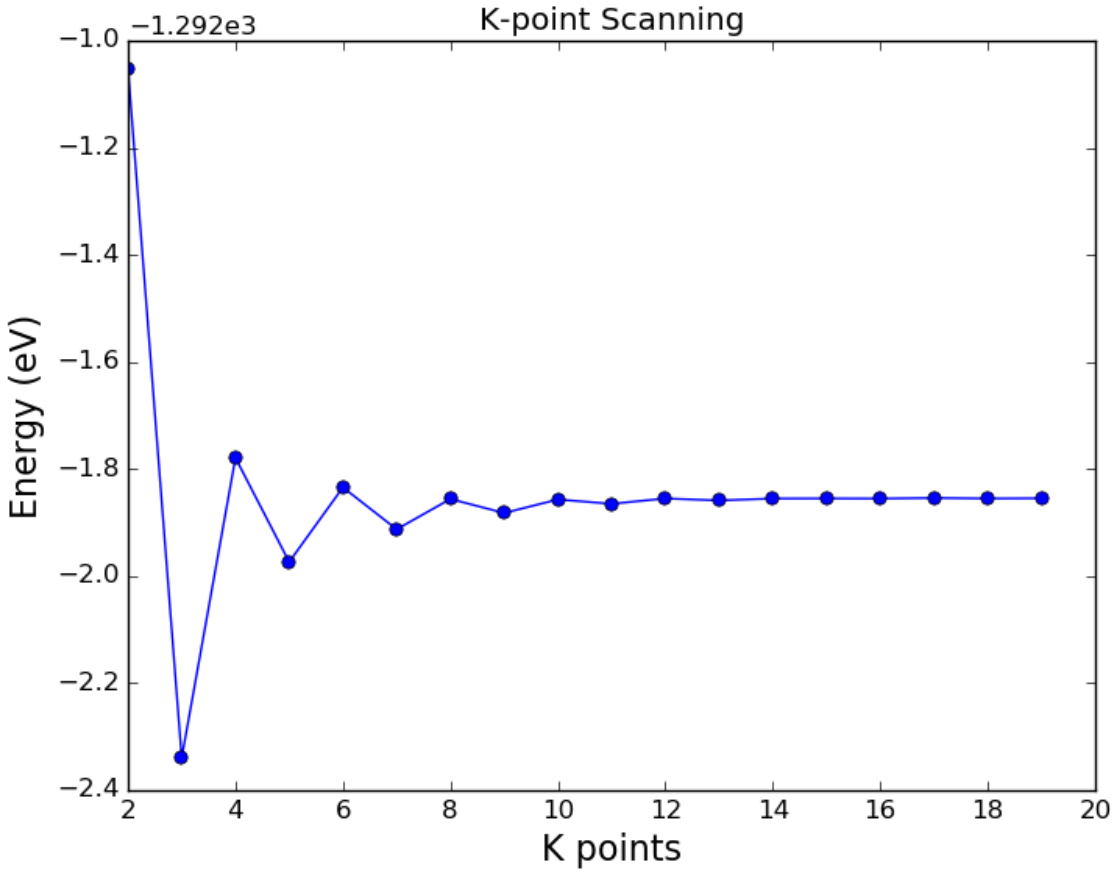
Note: The first calculation also takes some time and the efficiency of this method varies strongly from case to case. It is only effective if the result from the first calculation is good enough to reduce the number of iterations in the second calculation significantly.

Looping over parameters

Usually, there are numerical parameters which you do not initially know how to choose optimally, that is, the values of parameters which gives a desired balance between calculation time and accuracy. This applies to parameters like k-point sampling and mesh cut-off. Besides using some reasonable physical insight to get a rough idea, a more reliable way is to try different values and see how the results change. This can often be done on a small testing system, and the results can then be applied to a real and larger system of interest.

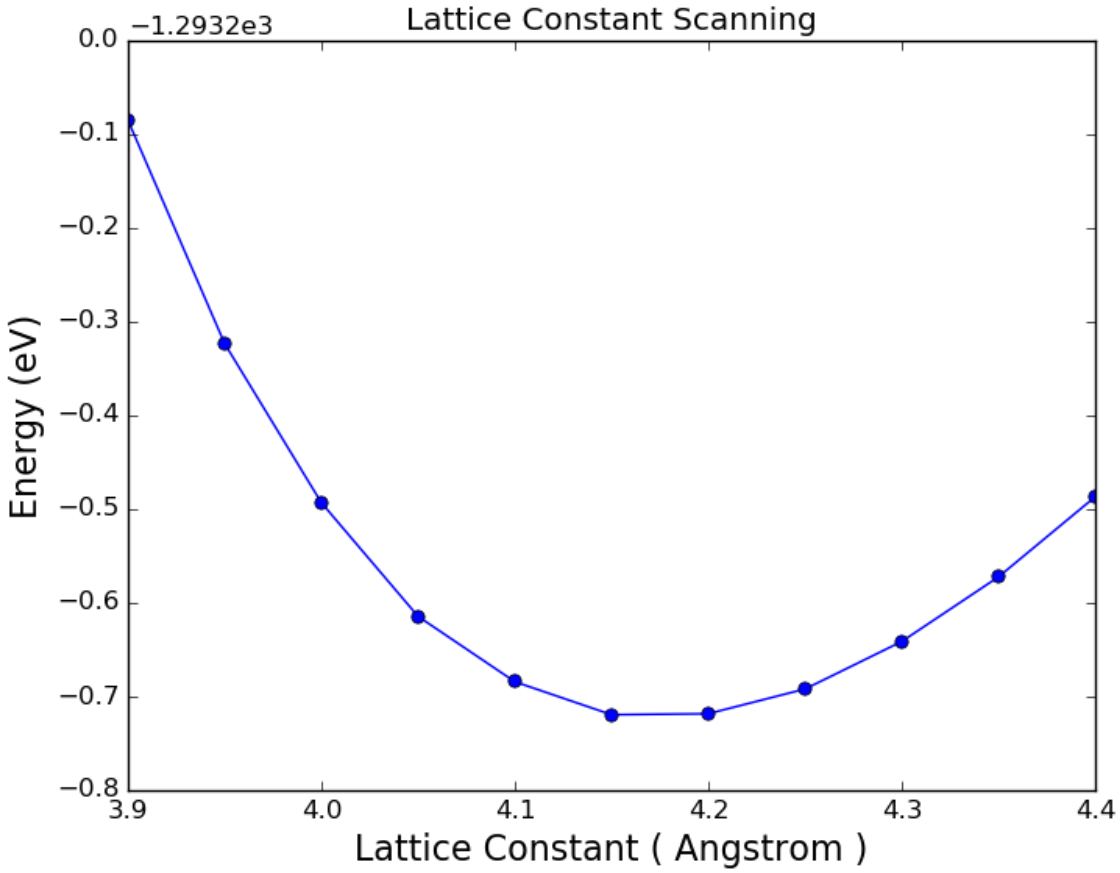
The script `kpoint_looping.py` checks how many k-points are needed to get a good value for the total energy of a gold crystal. When doing parameter scanning, it is a good idea to lower the tolerance of the self-consistent loop, which increases the accuracy of the solution. In this script we set it two orders of magnitude lower than default.

Running this script, you will get the following plot, from where you find that a sampling of 12 or 13 k-points along each axis is sufficient for good total energies for gold (valid for this specific combination of basis set, etc).



If you compare to the same calculation without any initial state, i.e. if each calculation is started from neutral atoms, the number of self-consistent iterations is typically more than two times larger, especially when the number of k-points is large (i.e. the most time-consuming parts of the script); thus the benefit of using an initial state in this type of loop is very obvious.

Another very common task is to loop over a variable, like a lattice constant, to find the lowest total energy for a configuration. This script `lattice_const_looping.py` is for finding the equilibrium lattice constant of gold.



Note: To optimize the unit cell size for a system like this, it is often more efficient to use the **OptimizeGeometry** method in **ATK**, which can optimize th lattice parameter(s) using an efficient algorithm. The calculation above is still meaningful, however, since it also gives an idea of the error we make by having a non-optimized cell. Furthermore, the values can be used to compute the bulk modulus if you do a more proper 3rd degree polynomial fitting.

In these examples the results are not saved. If you want to do some further processing, like compute the band structure for each configuration and compare those, add such statements inside the loop and store the results in a HDF5 file.

The result of fitting the parabola is a =4.17 Å; if we use GGA instead of LDA ,the value changes to 4.19 Å, i.e. in both cases about 2-3% above the experimental value of 4.08 Å. It is not expected that DFT always gives the experimental value, so this is not an error. A more relevant comparison is, for instance, what a corresponding high-accuracy plane-wave calculation gives, and these typically also overestimate the lattice constant of gold by 1-3%[1].

[1] K.ăLeung, S.ăB. Rempe, P.ăA. Schultz, E.ăM. Sproviero, V.ăS. Batista, M.ăE. Chandross, and C.ăJ. Medforth. Density Functional Theory and DFT+U Study of Transition Metal Porphines Adsorbed on Au(111) Surfaces and Effects of Applied Electric Fields. *Journal of the American Chemical Society*, 128(11):3659–3668, 2006. doi:10.1021/ja056630o.

Stepping up in bias

In the tutorial of Molecular Device, the I-V curve is obtained by climbing in bias using the converged previous state as the initial state.

Special example: anti-parallel spin in MTJs

This strategy has also been used in the magnetic tunnel junction (MTJ) calculation, where it is used initialize a calculation with a new spin configuration compared to the original calculation. This allows you to converge the calculation with a simpler spin-state and use that as a starting point. **ATK** will rescale the converged density matrix into an initial density matrix for the new calculation according to the distribution of spin up and spin down as given by the *initial spin* setting of the new calculation.

References

3.12.9 Restarting a stopped calculation

Version: P-2019.03

Saving the checkpoint file

It may happen that a calculation terminates before converging, for instance due to a power outage, or if a job runs over its allocated walltime in a queue. It may also be that convergence is not reached within the set number of maximum iterations. In these cases you will want to restart the job from the point it stopped.

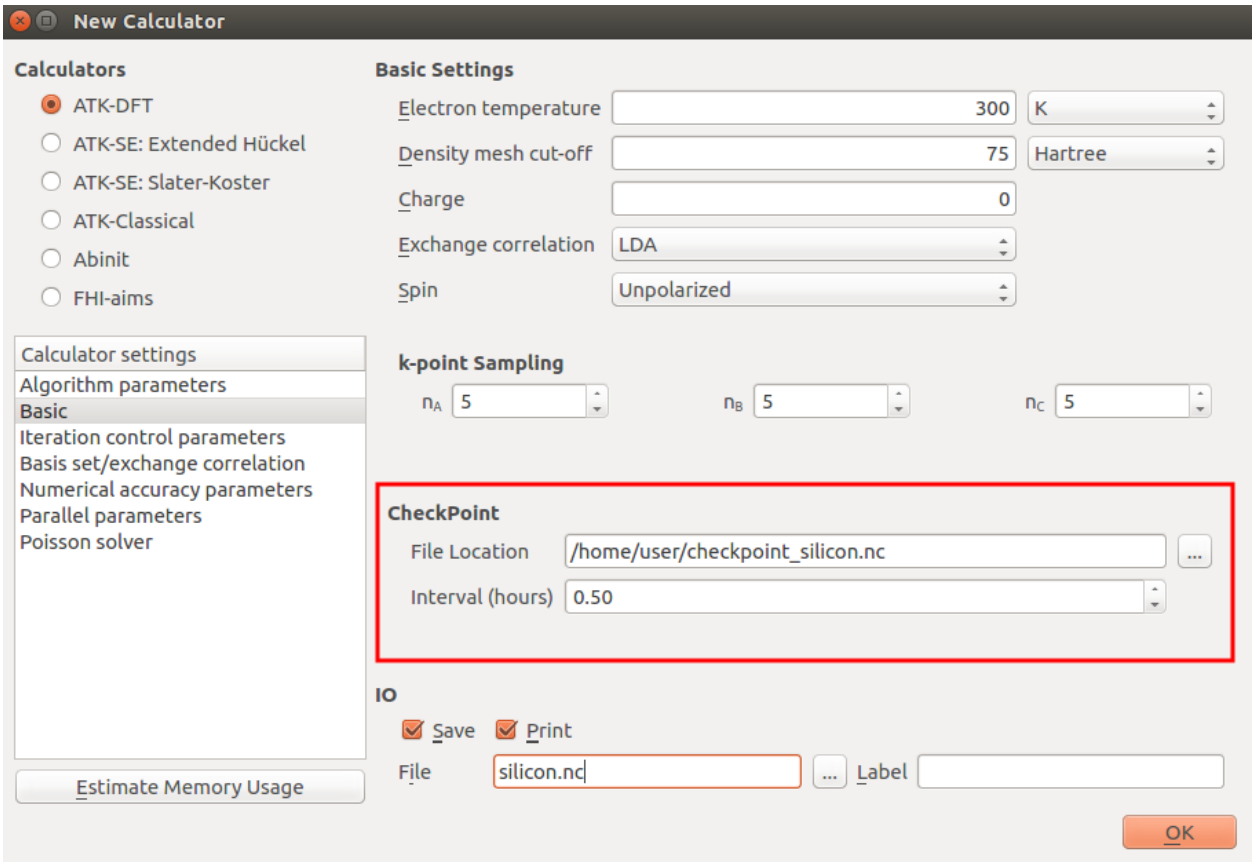
For this purpose, QuantumATK saves the current state of the calculation to a **checkpoint file** at regular intervals. The default is to save it every 30 minutes. The name of the checkpoint file is always written in the log file.

```
+------------------------------------------------------------------------------+
| Checkpoint Handler                                                           |
| Filename : /tmp/checkpoint28146777.nc                                        |
| Interval : 0.5 h                                                             |
+------------------------------------------------------------------------------+

                            |-------------------------------------------------|
Calculating Eigenvalues    : ================================================
Calculating Density Matrix : ================================================

+------------------------------------------------------------------------------+
| Density Matrix Report                       DM[U]     DM[D]      DD          |
+------------------------------------------------------------------------------+
|   0  Fe   [  0.717 ,  0.717 ,  0.716 ]    5.39789   2.59547  -0.00664        |
```

Note: The default location of the checkpoint file is in the directory specified by the environment variable TEMP. If you are running on a large cluster, you may not have permission to write to the TEMP directory, and even if you do, any files you create in this directory may be deleted automatically when your job finishes – even if the QuantumATK calculation did not converge. In this case it is important to specify the location of the checkpoint file manually, e.g. in your HOME directory.

Specifying the location of the checkpoint file

The image below shows how you can set the saving interval and the name of the checkpoint file using the **New Calculator** block in the QuantumATK **Script Generator**.



If you wish, you can refer to the QuantumATK Reference Manual to get more information on how to set these parameters or turn off the checkpoint handler.

Restarting the calculation from the checkpoint file

The quickest way to restart a calculation from a checkpoint file is to read the configuration from the checkpoint file and then call update on the configuration.

```
configuration = nlread("checkpointfile.hdf5")[0]
configuration.update()
nlsave("file.hdf5", configuration)
```

The argument to `nlread()` should of course be set to the actual checkpoint file name.

The disadvantage of this approach is that if the *original* script contained any analysis blocks (e.g. to compute the band structure), you need to manually insert those blocks to the bottom of the restart script.

Restarting the original script

Conceptually, a better approach would be to rerun the script you already have, but tell it to start not from scratch, but from the checkpoint file. This would also retain all analysis blocks, as defined in the original script. This is possible; you just need to insert the lines of code shown above in the appropriate way.

Let us assume that you have a standard script, produced by the Script Generator, without too many elaborate steps. That is, a straightforward sequence of Configuration and New Calculator, followed by analysis blocks. In other cases, you can always modify the script in the same way as described here, but you have to take more care to preserve the logic. Special care needs to be take if the script contains an **InitialState** block.

Open your original script in the **Editor** and locate the line

```
device_configuration.update()
```

Tip: For bulks or molecular calculations, the variable will be called `bulk_configuration` or `molecule_configuration` instead.

Then add the following line before that line:

```
device_configuration = nlread("checkpointfile.hdf5")[0]
```

Again, the argument to `nlread()` should of course be the actual checkpoint file name.

Now you can rerun the script.

Note:

- The checkpoint file is not written exactly at the specified interval, but only when a step in the self-consistent loop has been completed and the requested time interval has passed.
- The history of the self-consistent loop is not written to the checkpoint file. Therefore, convergence might become more difficult when restarting, since the mixing algorithm has less information to work with than normally.

Restarting geometry optimizations

Restarting a geometry optimization is much more complicated. For a lengthy relaxation it is therefore always a good idea to use a trajectory file; if the calculation is interrupted you can take out some of the later images and set up a new optimization using this geometry as a starting point. Note, however, that some images in a QuasiNewton geometry optimization are test balloons, which may correspond to very large forces (i.e. a very bad guess), especially during the first 5–10 steps. So, it can be important to choose an image that does not have too large forces.

3.12.10 Compute quantities from converged simulations

Version: 2016.0

In this tutorial, you will learn how to start a converged calculation to add additional functions. In this way a lot of simulation time can be saved.

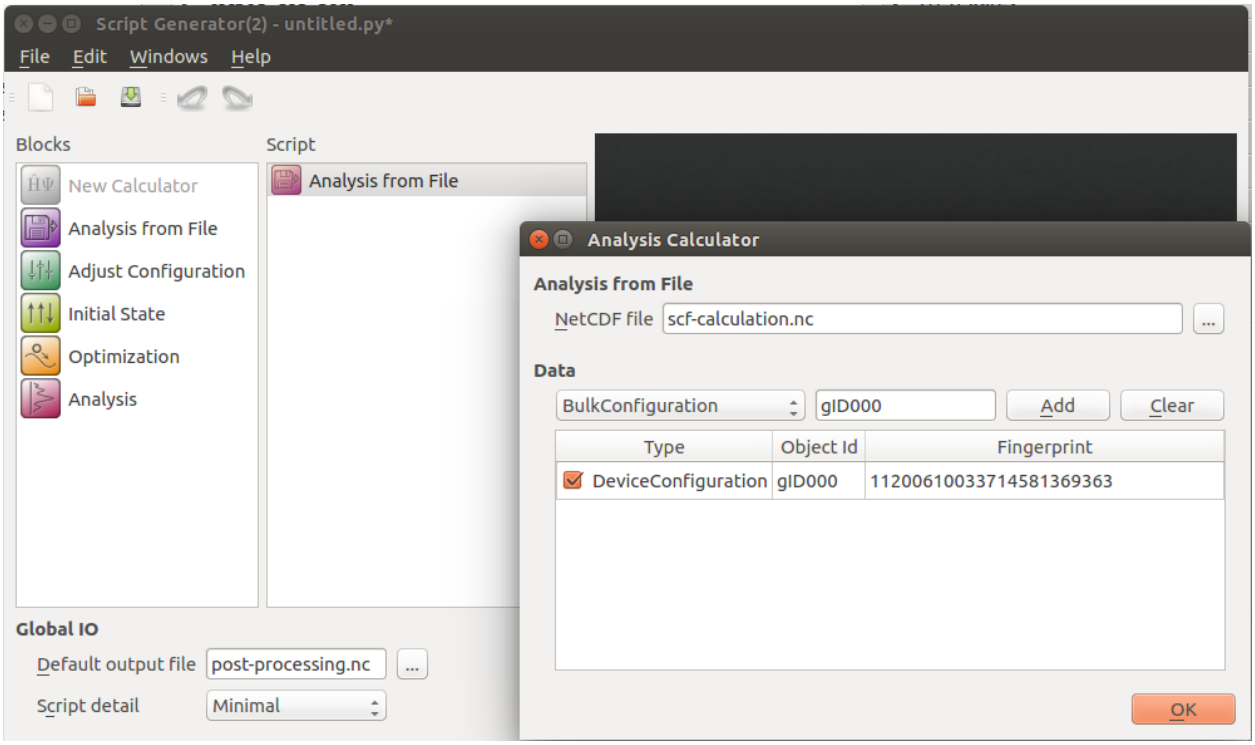
What can you do if you forgot to add the Bandstructure analysis to your script when you set it up? Or you discover you probably need more k-points to get a converged transmission spectrum? Do not worry - with **ATK** you dont have to compute all quantities together with the self-consistent calculation. You can always compute them later on, by reading the converged state of the calculation from the HDF5 file.

Using the **Script Generator** is usually the easiest way to do post-processing analysis, especially for small calculations. The post-processing can take some time, however, especially to compute the transmission spectrum or DOS with many k-points. In those cases you might want to run the script in parallel on a cluster (the analysis calculations usually parallelize linearly up to very many nodes). If so, it is still very handy to set up the script using the **Script Generator**, and then save it. That is - just like you would do it for the main self-consistent calculation itself.

Start a new **Script Generator** from the **QuantumATK** toolbar. When you open it without dropping any configuration on it, the only available block is *Analysis from File*, which is precisely what we want - to perform analysis based on a HDF5 file. Double-click this block to insert it into the script.

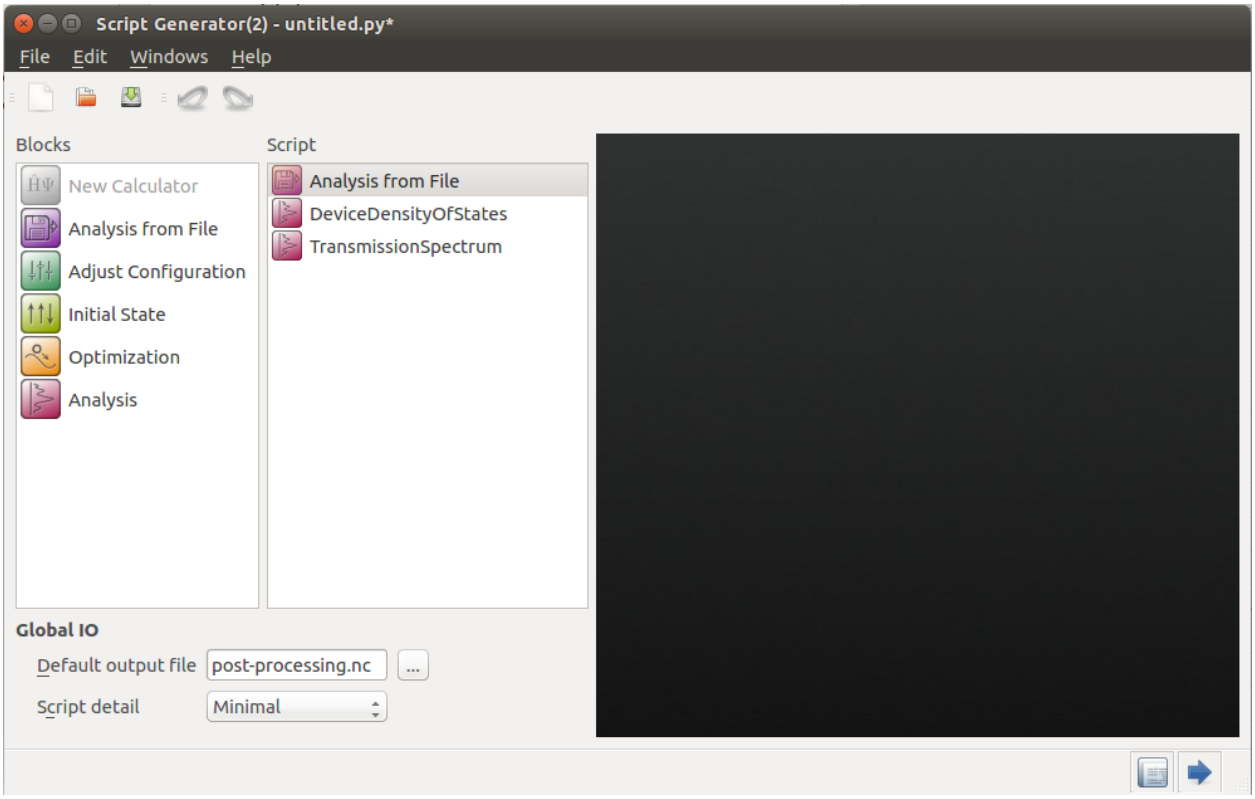
Open it, then browse and select the relevant HDF5 file.

Note: If you intend to run the script on another computer, from the command line (e.g. in parallel) you must naturally copy both the script and the HDF5 file to it. Make sure to place the two files in the same directory, and only specify the name of the input HDF5 file, without any path.



Next, check that the Object id corresponds to the self-consistent calculation you want to base the analysis on. If your calculation was a standard run, then most likely the default *gID000* is the one you want. However, if you did a geometry optimization, *gID000* corresponds to the initial geometry, and the optimized geometry will be *gID001*. You can inspect the contents of the HDF5 file, and the object ids, in the *LabFloor* in the main **QuantumATK** window.

Click **OK**, and then add all the *Analysis* quantities you want to compute, and set the parameters as appropriate.



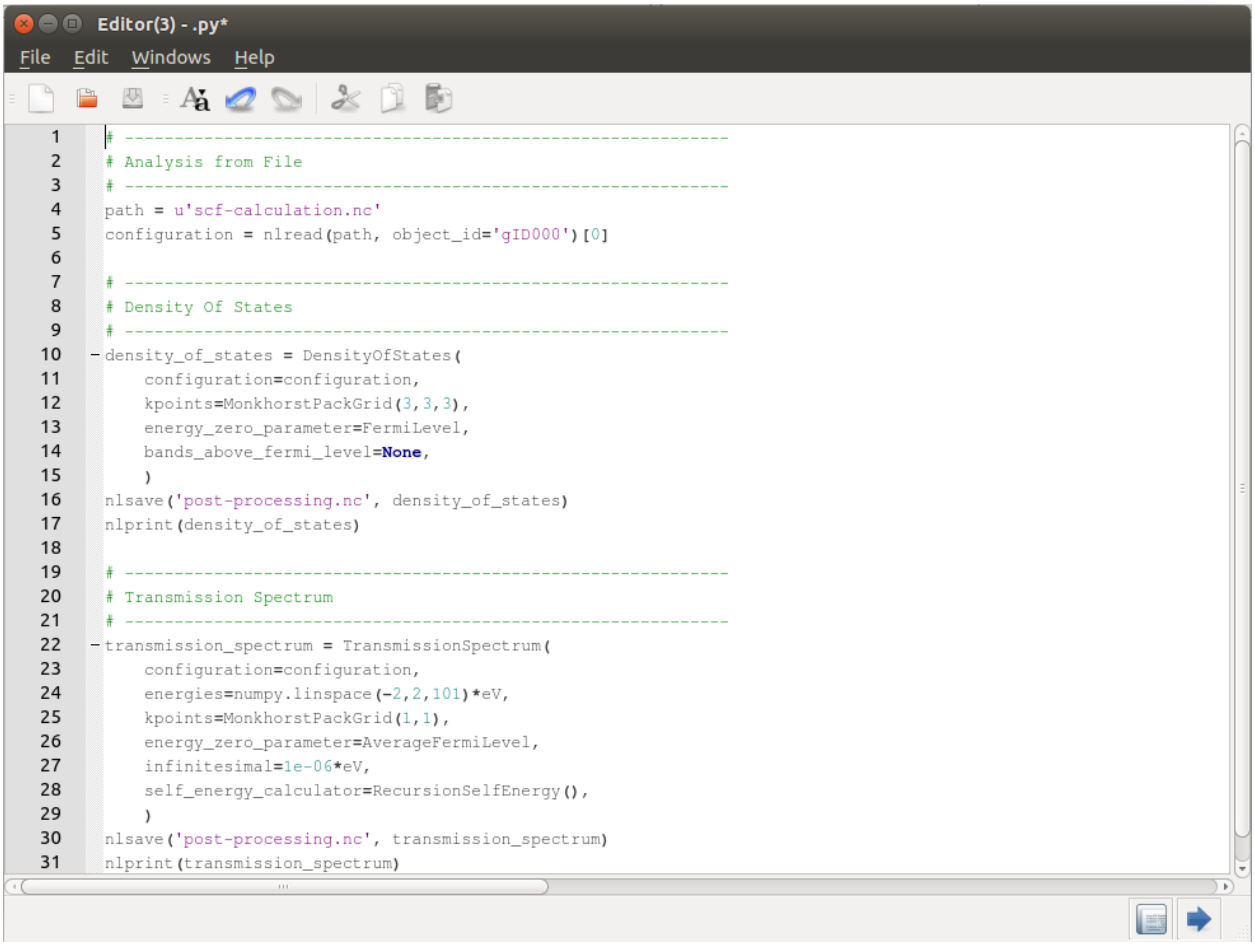
Finally, give a name and location at the blank *Default output file*.

Note: As above, if you will run the script from the command line on a different computer, just set a name without a path, in order to save the file in the running directory.

Tip: You might want to save the computed analysis quantities in a separate HDF5 file, in order not to make the original HDF5 file too big.

Now either send the script to the **Job Manager** by using the *Send To* button at the lower right corner, as usual, or save the script and copy it (with the HDF5 file!) to the cluster and run it from the command line.

If you havent looked into the Python scripts used by **ATK**, this is an excellent type of script to start with. If you send the script to the **Editor** , you will see the following.



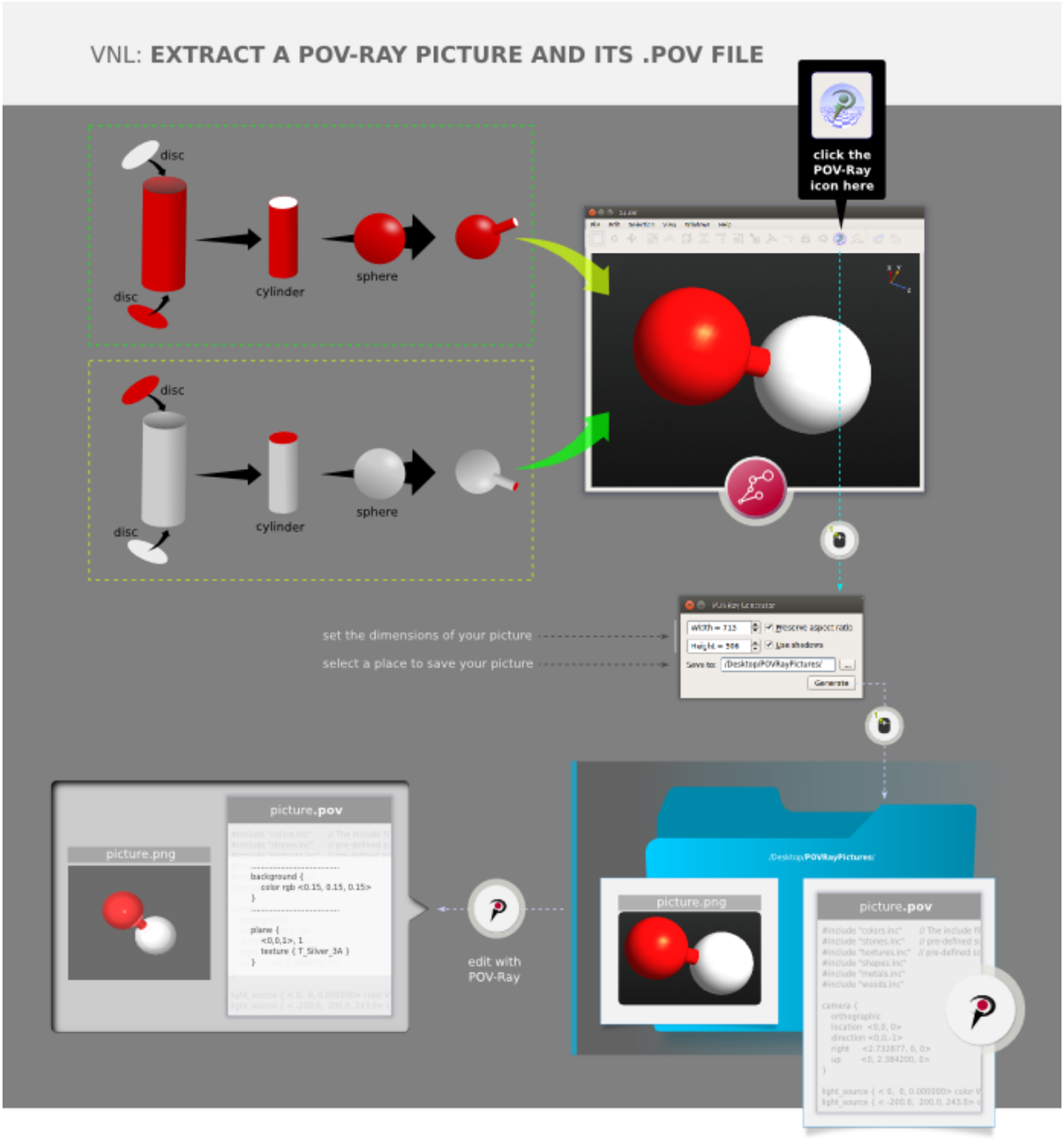
You will see that each function of the script is separated into blocks and under each block, there are lines to setup the detail parameters of the simulation. The line that does the main trick is at the very top, using the function *nlread()* to read the configuration from the HDF5 file.

Note: *nlread()* can in principle read many objects at once from a HDF5 file. Object ids must however be unique, so by using the object_id keyword, you are guaranteed to get only one object back. However, nlread() always returns a list of objects, even if there is only one match, so we need the [0] to pick out the first/only configuration matching the object_id.

3.12.11 POV-Ray images from QuantumATK

Elementary functionalities through an example

In this simple example (diatomic molecule) we are going to present how to edit an image of a molecule composed by two atoms (red and white). Also, it is a good opportunity to introduce some basic POV-Ray concepts we are going to use. The whole process is shown schematically below:



•**Step 1: Create a structure in QuantumATK.** The first step is to create this structure using the QuantumATK Builder. At this point it is good to analyze what our structure has. Here in this example we have two atoms represented by two spheres (white and red colors) connected with a single bond that represented by a thin cylindrical object. This object is composed by two cylinders (white and red cylindrical surfaces) and their top and bottom caps (white and red discs). Namely,

Structure Composition:

1.sphere (red atom)

2.sphere (white atom)

3.cylinder (red atom bond-part)

4.cylinder (white atom bond-part)

5.disc (red bottom cap of the red atom bond-part)

6.disc (white top cap of the red atom bond-part)

7.disc (red bottom cap of the white atom bond-part)

8.disc (white top cap of the white atom bond-part)

•**Step 2: Extracting POV-Ray files.** Once we are ready with the molecule in QuantumATK we are ready to extract it as a POV-Ray file (POV-Ray files have the **.pov** extension). It is quite easy to do that; just click the POV-Ray icon at the top-right of the Builders toolbar. A pop-up window appears asking you to give a name to the .pov file. Lets name it as **picture.pov**. Clicking the Generate button we extract our diatomic molecule in a POV-Ray file and our rendering in picture.png format.

> **Attention:** *It is important to mention here that the extracted .pov files (from QuantumATK) cannot be modified with QuantumATK. We can do that only with the POV-Ray program changing and importing parameters and properties.*

Examining the .pov file

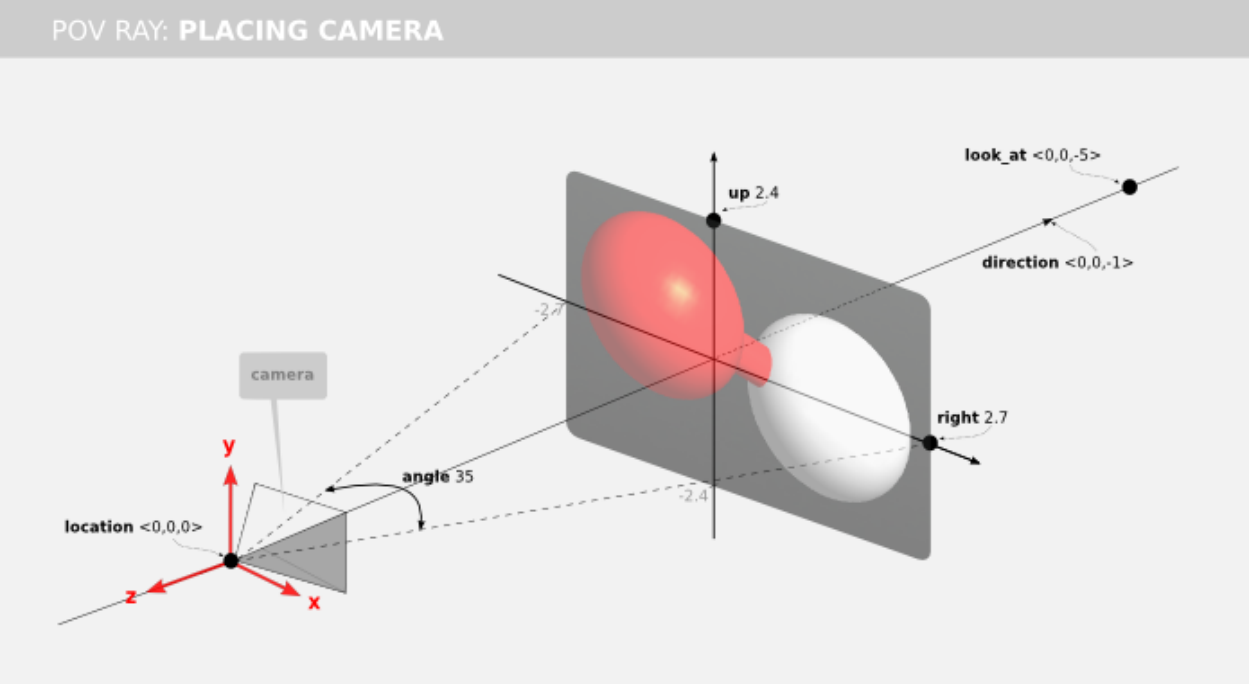
•**Header Files.** *Opening the picture.pov file we see a C++ code that describes each element and every detail of our picture. In the first lines of the code we include some .inc header files that are responsible for handling various colors, textures and shapes. Their role is obvious and it is not necessary to get into more details about that now:*

```
#include "colors.inc"
#include "stones.inc"
#include "textures.inc"
#include "shapes.inc"
#include "metals.inc"
#include "woods.inc"
```

•**Camera(s).** *The next few lines describe the properties of the camera. The camera element is very important because the rendering of the picture is depending on it.*

```
camera {
    orthographic
    location  <0,0, 0>
    direction <0,0,-1>
    right     <2.7, 0, 0>
    up        <0, 2.4, 0>
}
```

The above lines of code describe the camera view that we have used to extract our diatomic molecule from QuantumATK. The following picture shows some basic camera properties in the 3D space. For more information on editing cameras and their properties look at .



•**Light Source(s).** *The next important element is the light_source element. One may control this element by importing and/or changing various light source attributes. In the present example we have two light sources where their locations and their color lights have defined:*

```
light_source { < 0,  0, 0> color White }
light_source { < -200.0,  200.0, 243.0> color White shadowless}
```

•**Background(s).** *This element is used to assign a color to all rays that do not hit any object. Here we have defined a grey background providing the proper RGB color coordinates:*

```
background { color rgb <0.15, 0.15, 0.15> }
```

•**QuantumATK Object(s).** *The next part of the code includes the description of the various objects that we have created and rendered in QuantumATK. Namely, the elements and their properties (like textures, colors, relevant positions, radii, etc.) that are mentioned in step 1 above. It is easy to see in these code lines that we have indeed: two spheres, two cylinders and four discs and their properties are defined between the curly brackets. For example, the red atom/sphere is described by the following sphere:*

```
sphere {
    <-0.3, 0.2, -2.1>,  0.4     // center and radius
    texture {
        pigment { color rgb <1. 0.06 0.06> }
        finish {
             ambient .2
             diffuse .4
             specular .75
             roughness .001
             reflection { .6, metallic}
        }
    }
}
```

In line 2 we can see the location (in Cartesian <x,y,z> coordinates) of the center of the sphere. The next number

after the comma is just the radius of the sphere. The rest lines describe the texture of the sphere. In **pigment** we can clearly see the color definition (red) while with the (staring in line 7) we have a more realistic impression of the object. **Finish** is quite useful identifier since it gives us the option to control many properties (like roughness, diffusion and reflection effects, and many more) of the surface(s) of an object in a more detailed level. The final result of all these texture-terms are highly correlated with the light sources and the camera view of course. This means that when we change a parameter in **finish** identifier we must always take into account the light properties that we have already set in our scenery according to our camera view (different camera views give us different impressions of the same object since the light reflections are changing). Cylinders and discs are defined in a similar way.

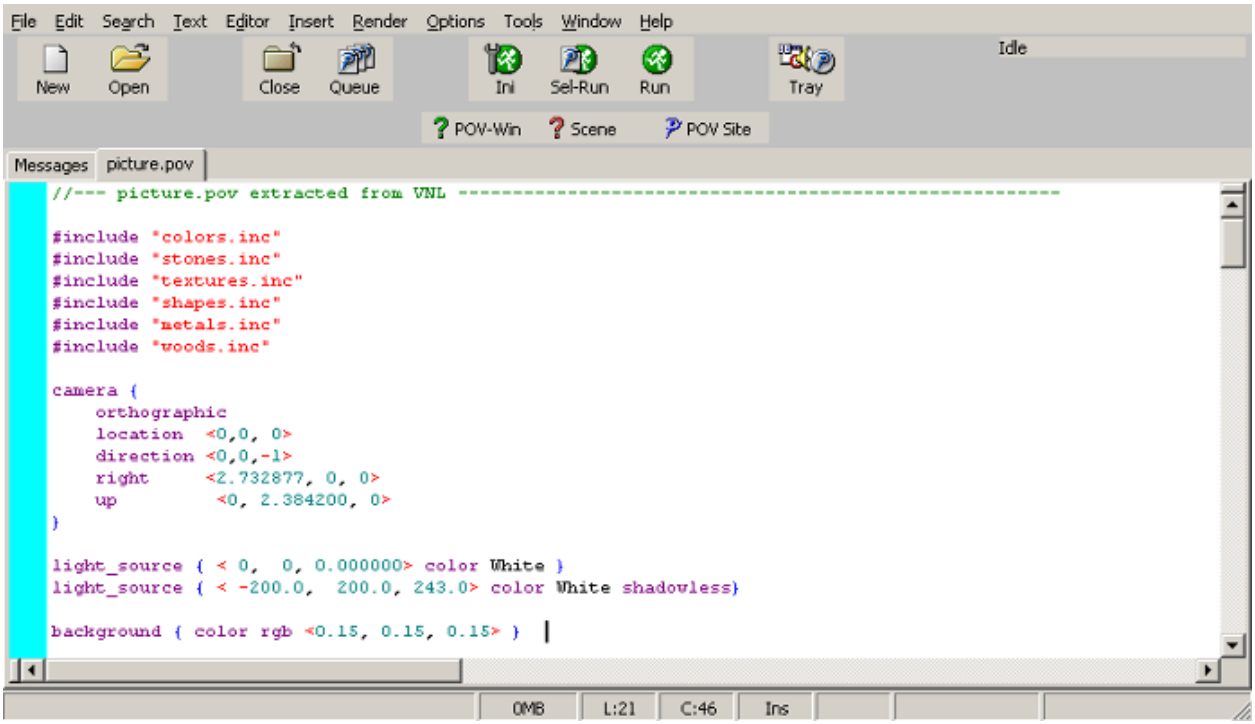
•**Step 4: Editing POV-Ray Files.** Now we are ready to update our picture. Editing the example1.pov file we are able to control many parameters, to add objects and to create many artistic effects in general (using the POV-Ray program). In our example we just add a plane. The we want to have is described by the following lines of code:

```
plane {
    <0,0,1>, 1                          //This represents the plane 0x+0y+z=1
    texture { T_Silver_3A }   //The texture comes from the file "metals.inc"
}
```

Exporting pictures with POV-Ray

Note: The content of this paragraph refers to the general use of POV-Ray as an independent program and does not restricted by QuantumATK or any other software. QuantumATK just has the ability to export complex structures as .pov and .png extensions.

In order to be able to export an edited picture you must have already installed POV-Ray on your system. You can find detailed instructions about this topic at the . So, assuming that you have already a [filename].pov file (in our example here [filename] stands for picture) follow the steps below:



| POV-Ray on Windows

•**Step 5.1:** Click the POV-Ray application icon and the POV-Ray interface opens (see picture above).

•**Step 5.2:** Click **Open** button, and navigate to your files to import your [filename].pov file you prefer (picture.pov for instance). Now you can see and edit the code according to your preferences. Once you are ready just click the green **Run** button and in some seconds you will create your [filename].png picture.

| POV-Ray on Linux

•**Step 6.1:** Open the [filename].pov with a text editor, make the changes you want (if any) and save it.

•**Step 6.2:** Open a terminal and type `povray [path/to/pov/file]`. For example if the picture.pov file is on the Desktop, then type:

```
$ cd Desktop/
$ povray Desktop/picture.pov
```

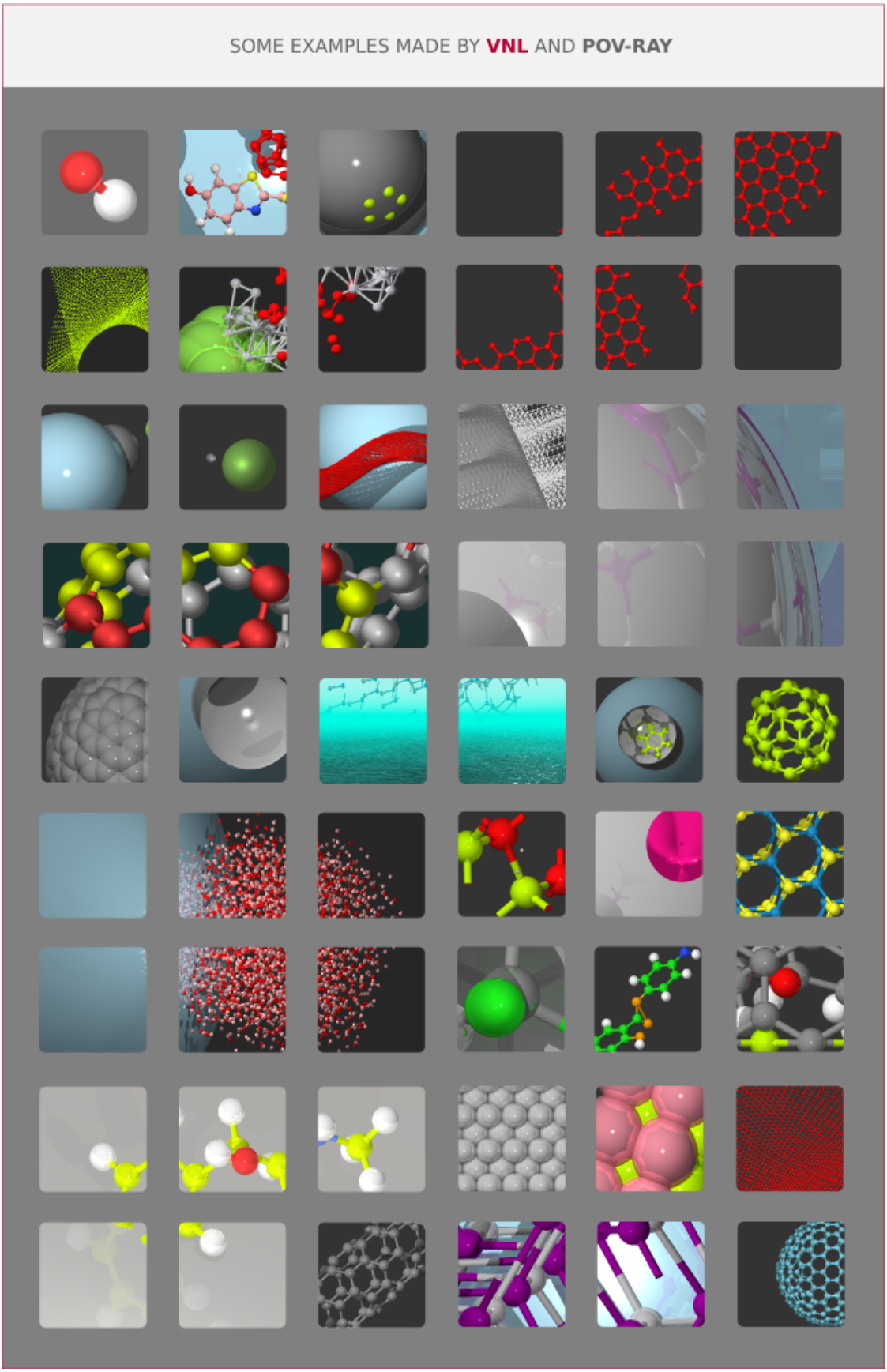
or directly

```
$ povray Desktop/picture.pov
```

If you want specific picture size type:

```
$ cd Desktop/
$ povray picture.pov -w320 -h240 +a0.3
```

Where **-w**, **-h**, **+a** numbers define the **width**, the **height** and the **aspect ratio** respectivelly.


SOME EXAMPLES MADE BY VNL AND POV-RAY

3.12.12 Transport calculations with QuantumATK

Version: 2015

Downloads & Links

`zn-zno-zn.py` `zero-bias.py` `analysis.py` `finite-bias.py` `iv-curve.py`

This tutorial introduces you to one of the most important functionalities in **QuantumATK**, namely electron transport calculations.

The system you will investigate is a Zn|ZnO|Zn junction, which is selected for demonstration purposes, in order to make calculations relatively fast. However, this system does encapsulate many of the relevant and interesting physical effects present in more realistic devices, and these will be discussed in the tutorial.

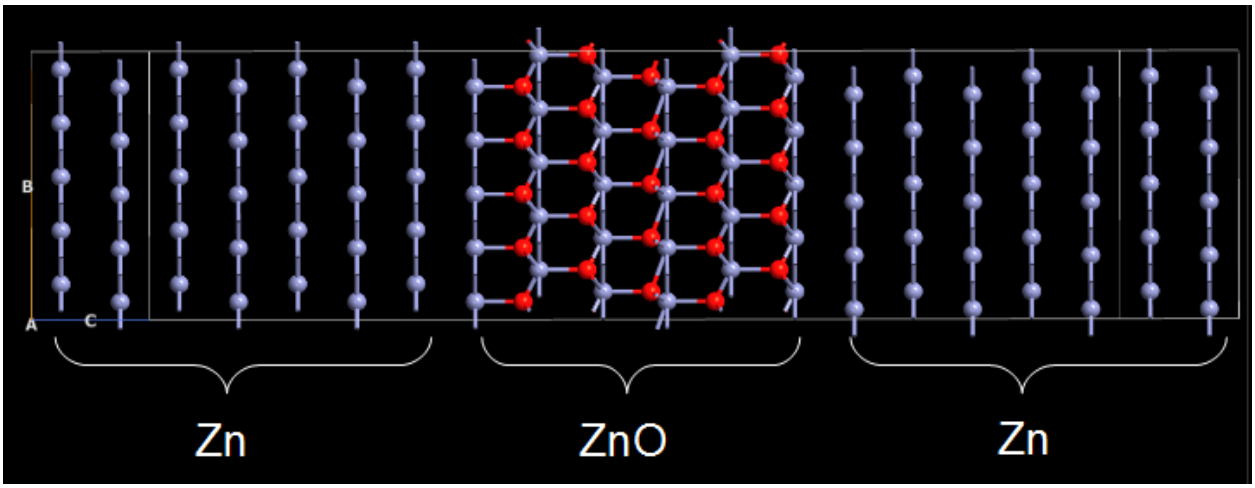


Fig. 3.121: The Zn-ZnO-Zn system to be studied in this tutorial. Note that the configuration is repeated 5 times along the B axis to improve the visual clarity.

Note: In order to run the calculations in this tutorial, you must have a license for the **ATK-SE** tight-binding engine. If you do not have one, you may obtain a time-limited demo license by contacting us via our website: Contact QuantumATK.

Attention: This tutorial was prepared using QuantumATK version 2016.3. Downloadable scripts and output files may therefore not work with QuantumATK 2015 and earlier.

Introduction

In this tutorial, the electronic structure of the Zn-ZnO-Zn junction will be calculated using a **tight-binding** (TB) model in combination with the **non-equilibrium Greens function** (NEGF) technique for the actual calculation of the electron transport through this junction. A detailed description of technical aspects of the computational parameters and models can be found in the ATK Reference Manual.

To set up the calculations, you will use the graphical interface **QuantumATK** (QuantumATK). If you have never worked with QuantumATK before, it is highly recommended to first go through the Basic QuantumATK Tutorial to become familiar with the basics of how to operate the software.

You will here use a predefined device configuration, but we have many other tutorials that show you how to set up other device geometries.

Tutorial objectives

We will first introduce the central concepts involved in a device calculation, and then present a range of tools that can be used to analyze the electronic transport properties of the device.

After reading this tutorial and performing the calculations, you will be able to:

•Understand the general concept of a device configuration.

•Test convergence of physical quantities with respect to computational parameters.

•Check if a given bulk configuration is suitable as an electrode.

•Calculate the transmission spectrum at zero and finite bias voltage.

•Analyze the transmission spectrum using the Transmission Analyzer.

•Calculate and plot transmission eigenstates.

•Calculate and plot the projected device density of states.

•Calculate the potential profile (voltage drop) across the device.

•Calculate the I–V curve and analyze it.

Geometry for transport calculations

The type of atomic configuration needed for an electron transport calculation is fundamentally different from the configurations used in atomic-scale simulations of periodic and finite structures. This is because an electron transport calculation requires **open boundary conditions** in the transport direction. Open boundary conditions make it possible for incident electrons coming from the source electrode to be partially transmitted through the central part of the device and end up in the drain electrode, and partially reflected back to the source.

One may see the device configuration as a combination of periodic and finite configurations, as shown in the figure below. The left and right bulk (periodic) **electrodes** (source and drain, in a transistor perspective) are attached to the **central region** of the device, which is finite (non-periodic) along the transport direction. The central region is often called the scattering region, because the charge carriers traveling between the two electrodes are scattered by changes to the physical and chemical environment in the central region, for example by interfaces and defects. The central region therefore determines the *functionality* of the device, while the left and right electrodes are modeled as perfect leads that connect the device to the external source and drain.

For reasons of numerical stability during QuantumATK device calculations, there is a region in both ends of the central region where the element types and ionic positions are fixed to match the electrodes – these are the **electrode extensions**. Both must always be an exact replica of the corresponding electrode, and are therefore not allowed to change during geometry optimization of the device.

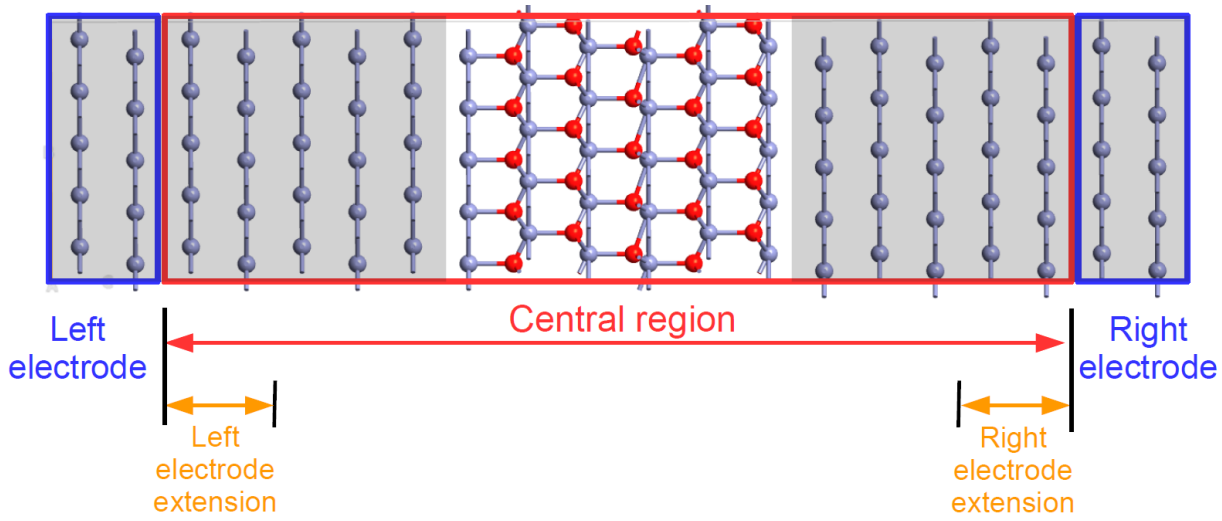


Fig. 3.122: A two-probe device configuration consists of two electrodes and a central region. Both ends of the central region, called the electrode extensions, must be exact replicas of the corresponding electrode structures.

Hint: In both figures above, the device configuration is periodically repeated 5 times along the B axis (which points up in both figures) to improve visual clarity. However, just like in standard atomistic simulations one should always use device configuration with the smallest lateral unit cell area, taking advantage of any structural periodicity of the device in the A and B directions (orthogonal to the transport direction, C).

Important notes

- The C-direction in the device configuration is the **transport direction**, meaning that the electrodes are always connected to the central region in this direction.
- The electrodes are by default assumed to be periodic in the A and B directions, i.e., perpendicular to the transport direction. When simulating transport through 1D or 2D confined systems, such as wires, tubes, ribbons, and sheets, a sufficiently large vacuum padding should be added along the confined direction(s). This allows the electron density and electrostatic potential to decay correctly in the vacuum region, and this also avoids spurious interactions with repeated images of the system.
- The central region can take many different forms. For example, it can be a metal–semiconductor–metal sandwich, as in this tutorial, a molecule between the two electrodes, or a piece of a nanotube or a graphene flake with a defect.
- In the Zn|ZnO|Zn device treated here, the two electrodes are identical, but this is not a general requirement. For example, QuantumATK can easily simulate the electron transport through an interface between two different materials.
- In addition to source and drain electrodes, QuantumATK supports inclusion of **electrostatic gates** for transistor simulations. This will not be covered here, but is addressed in several other tutorials.

The screening region

The central region should normally involve some structural inhomogeneity (interfaces, surfaces, molecules, defects, etc.) that leads to rearrangements in the electron density and hence in the self-consistent potential felt by the electrons. This electronic rearrangement may be further enhanced by structural relaxation effects inside the central region (excluding the electrode extensions, which are not allowed to relax). This deviation of the central region potential from the bulk electrode potential causes the electrons to be scattered, which will cause a voltage drop across the central region at finite bias.

Note: Simulating the transport properties of a perfectly periodic structure is only possible at zero bias (since there is no place for the voltage to naturally drop in a perfectly periodic structure, as there is no scattering), and can moreover be done in a simpler way, using just a periodic bulk configuration

---

The NEGF formalism assumes that the potential in the central region converges smoothly towards the electrode potentials, such that there is no scattering at the two electrode–central region interfaces. In the sketch of the Zn|ZnO|Zn device shown below, the Zn metal regions in both ends of the central region are on a gray background, just like in the electrodes, in order to indicate the so-called **screening regions**, in which the potential and charge rearrangements at the Zn–ZnO interfaces are screened and smoothly approach the electrode potentials. If the screening regions are shorter than the actual (physical) screening length of the system, spurious scattering may take place at the electrode–central region interfaces. This may give rise to poor convergence of the self-consistent calculation and inaccurate results in finite-bias calculations.

The figure above shows the potential profile due to an applied bias voltage of U = -0.5 Volt in the Zn|ZnO|Zn device considered here, meaning that the voltage on the left electrode is 0.5 V lower than that on the right electrode. Most of the voltage drop occurs in the ZnO region while the potential in the metallic screening regions is virtually flat, so no scattering takes place in these screening regions. In the section *Finite-bias calculations* you will reproduce this potential profile.

Attention: In systems with semiconductor electrodes, the screening length can be quite long, and a very long central region may be required before the potential approaches the bulk value. However, in most real situations, the semiconductor electrodes are doped, which brings the screening length down to manageable simulation sizes, provided the doping level is high enough.

Note:

- The entire treatment of electron transport is in this context coherent and elastic, i.e. there is no scattering by phonons.
- The NEGF formalism assumes that the electrodes are in thermal equilibrium, which formally requires that the electrical current through the device is low. This always holds true at low voltage, and may also be a valid assumption at higher voltage, provided that the transmission probability through the central region is sufficiently low.

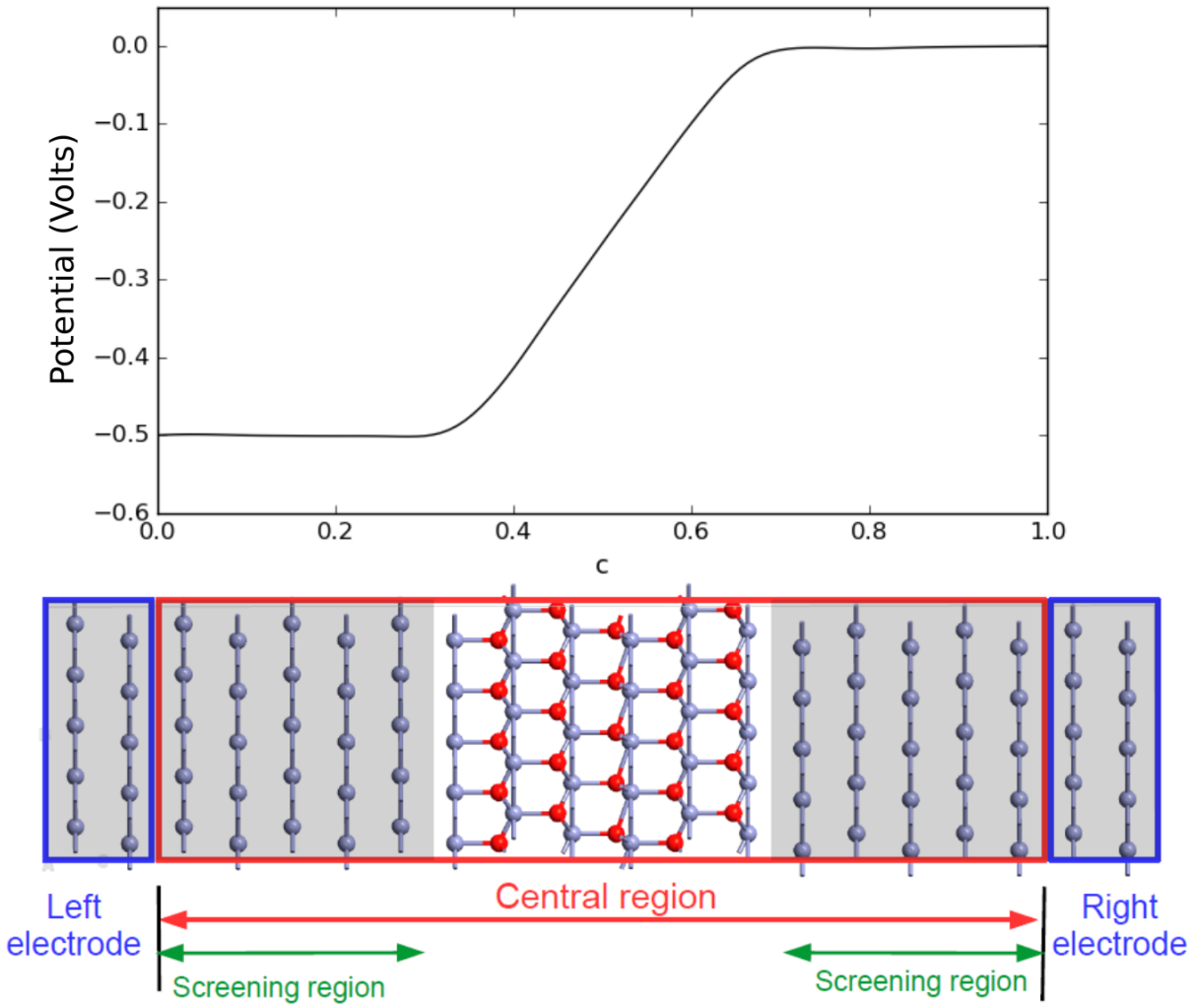
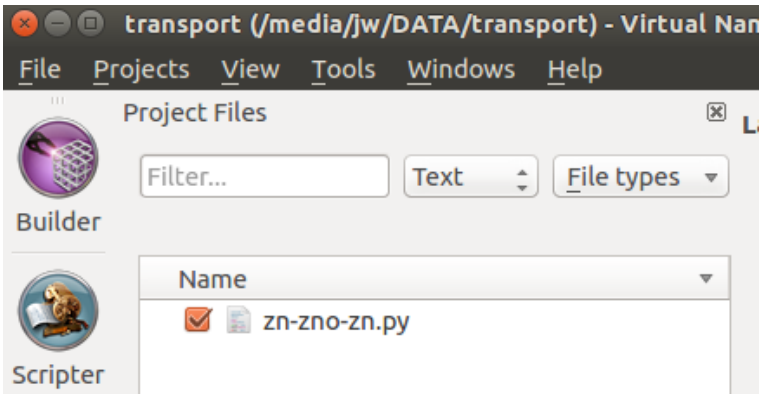


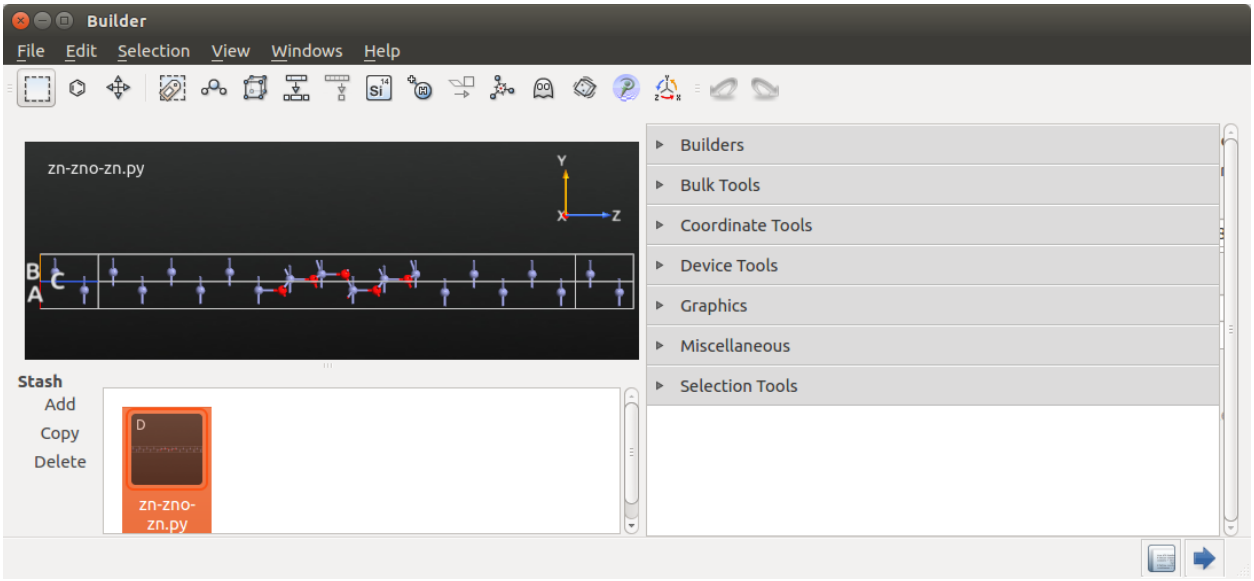
Fig. 3.123: Top: Electrostatic potential along the transport direction. Bottom: Device geometry. The screening regions in the left and right parts of the central region, indicated by gray background, must be longer than the screening length of the system to ensure a smooth transition between the potential of the central region and electrodes.

Getting started

Start up **QuantumATK** and create a new project. You would normally use the **Builder** to create the device configuration, but here we will use the Zn|ZnO|Zn device already defined in the QuantumATK Python script `zn-zno-zn.py`. Download the script and save it in the project folder. It will then show up on in the Project Files list.



Drag the script onto the Builder to access a large palette of tools for creating new configurations and modifying existing ones.



Convergence of electrode parameters

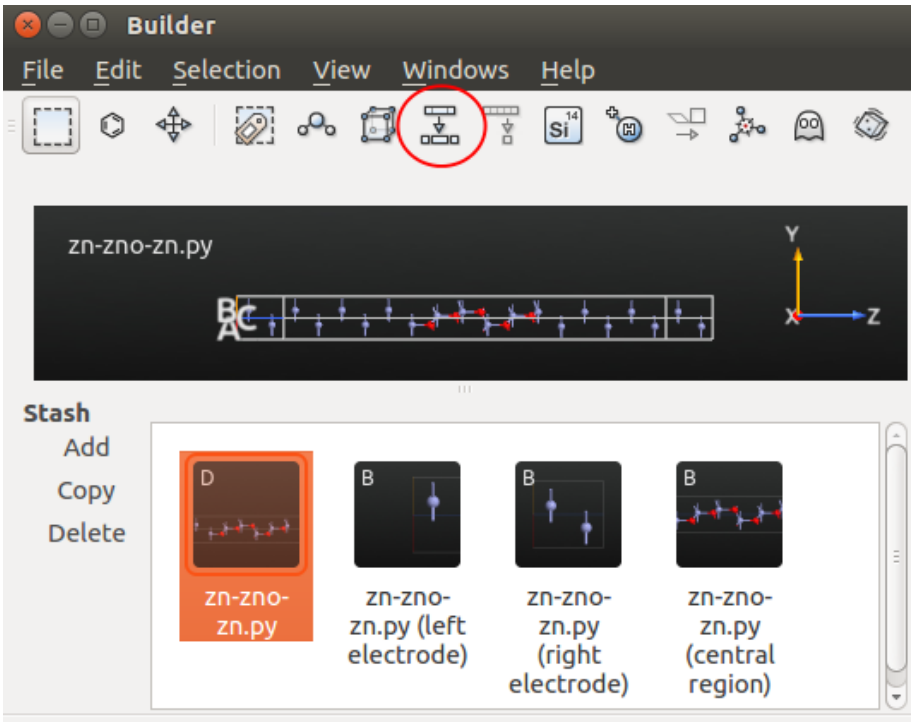
An QuantumATK device calculation is performed in multiple steps. First, the electrode(s) and the central region are treated as (periodic) bulk systems, and the self-consistent electron density is calculated for each of them. The device calculation is then started, using the central region bulk electron density as an initial guess, and matching the electronic structure at the two electrode/central region boundaries to that of the electrodes. Finally, the self-consistent electronic structure of the central region is determined, and thereby of the full device.

It is important to the procedure outlined above that the electrode bulk calculations are well converged, both in electrode lengths and numerical accuracy. The purpose of this section is to demonstrate the convergence of two very important numerical accuracy parameters; the mesh cut-off and the number of k-points in the transverse directions (A and B).

The convergence test will here be performed only on the left electrode, since the two electrodes are (physically) identical. In case the electrodes are not identical, the required mesh cut-off and k-point sampling should be determined separately for both of them. However, since these numerical parameters must be the same in the two electrodes in the device calculation, in the end they need to be chosen large enough that both electrodes are accurately described.

Split the device configuration

Use the *Splitter* tool in the top toolbar to split the device into electrodes and central region. This will add 3 new configurations to the **Stash**, corresponding to the left and right electrodes and the central region.



You can now highlight the left electrode configuration and use the botton to send it to the **Script Generator** and set up a range of bulk calculations for the electrode using different density mesh cut-offs and k-point samplings to study the numerical convergence.

For convenience, scripts have already been prepared for the calculations needed in the two convergence studies. In the following, you should simply download and run them.

Mesh cut-off

The first numerical parameter to converge is the density grid cut-off. This controls the density of the real-space grids in QuantumATK, which represent e.g. the electrostatic potential and real-space electron density. The density cut-off is specified as an energy E; the corresponding distance between the grid points is given by

$$\Delta x = \frac{\pi\hbar}{\sqrt{2mE}},$$

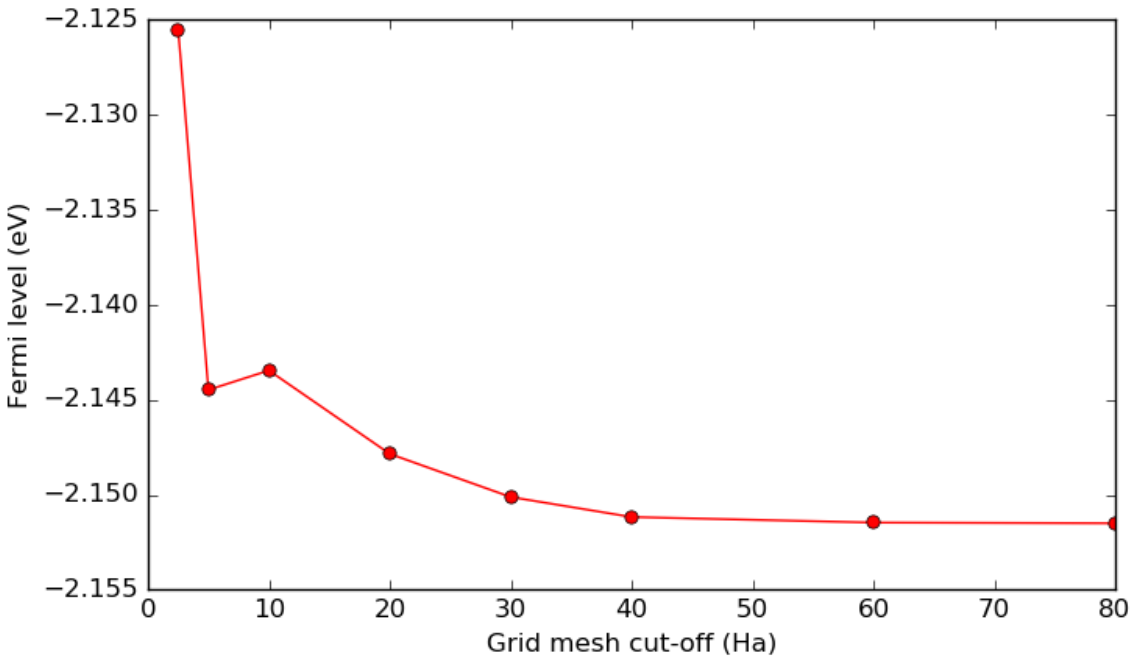
where m is the electron mass.

The script `mesh_loop.py` performs calculations of the ground state of the electrode for different mesh cut-offs ranging from 2.5 Ha to 80 Ha. All other numerical accuracy parameters are kept constant. The convergence observable is the Fermi level (chemical potential) of the electrode.

The script simply loops over the mesh cut-offs, performs the calculation for each value, and finally plots the results. Download the script, and then run it using the **Job Manager**. If you are not familiar with the QuantumATK Job Manager, please refer to the tutorial *Job Manager for local execution of QuantumATK scripts*. You can of course also run the script from command line:

```
$ atkpython mesh_loop.py > mesh_loop.log
```

The plot will automatically appear once the calculations are done (in less than a minute). The plot is shown below. It is clear that the default 10 Ha mesh-cut-off for tight-binding calculations is a perfectly sufficient default, since the Fermi level changes by less than 10 meV when using larger values (which are computationally more expensive), and we will keep it for further calculations.



Attention: This weak dependence of computed quantities on the mesh cut-off is typical of tight-binding models. The same would generally not be true for DFT calculations, where the default mesh cut-off with the cheapest pseudopotentials is 75 Ha.

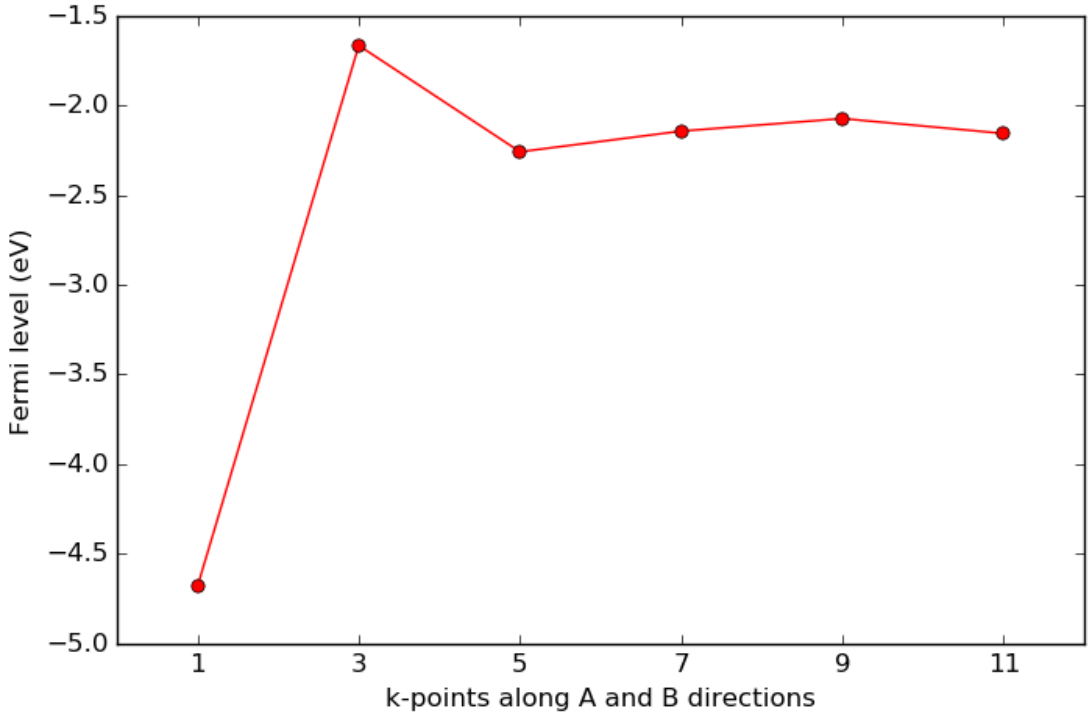
Transverse k-points

In the same fashion as above, you will now study the electrode Fermi level as a function of the number of k-points in the transverse directions (A and B), which are orthogonal to the transport direction (C).

The number of k-points in the transport direction should be quite large in a device calculation, so keep it at n_C = 51. All other numerical accuracy parameters should be kept at their default values, including the grid mesh cut-off (10 Ha).

Use the script `kpts_loop.py` for this convergence study. It is identical to the one used above for the mesh cut-off, but loops over odd transverse k-points in the range 1 to 11.

Download and run the script, which will produce the plot shown below. It is quite clear that the calculation with only 1 k-point along the A and B directions fails completely in describing the periodic Zn electrode bulk, while the calculations with 5 or more transverse k-points are much better.

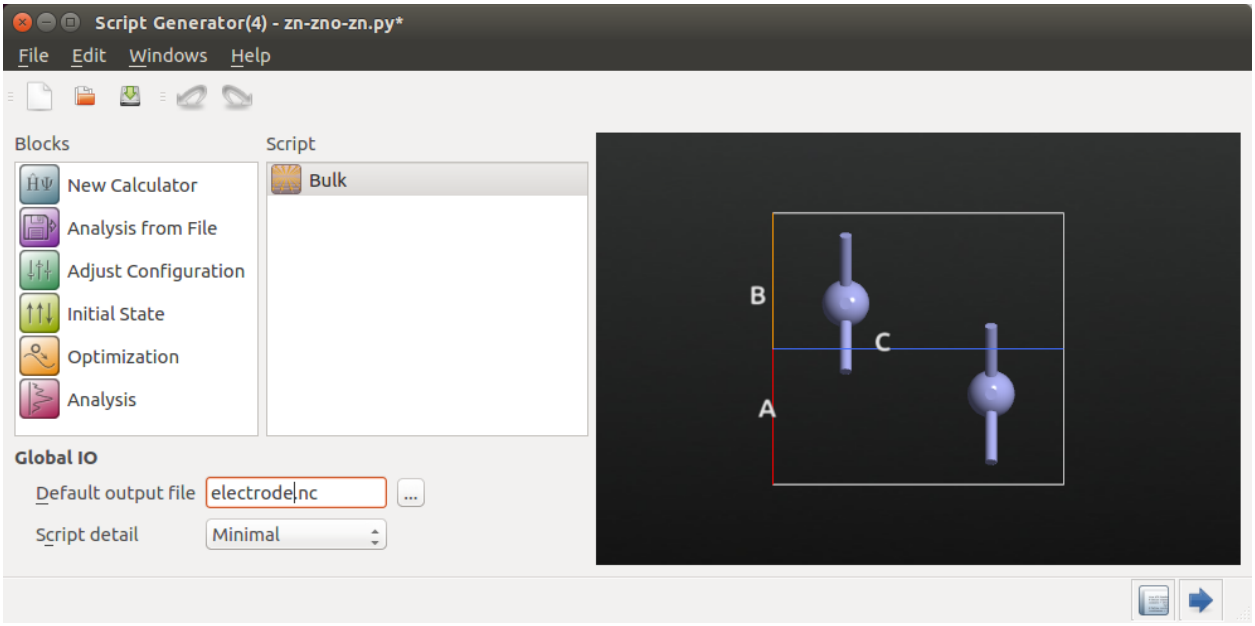


Check the electrode geometry and size

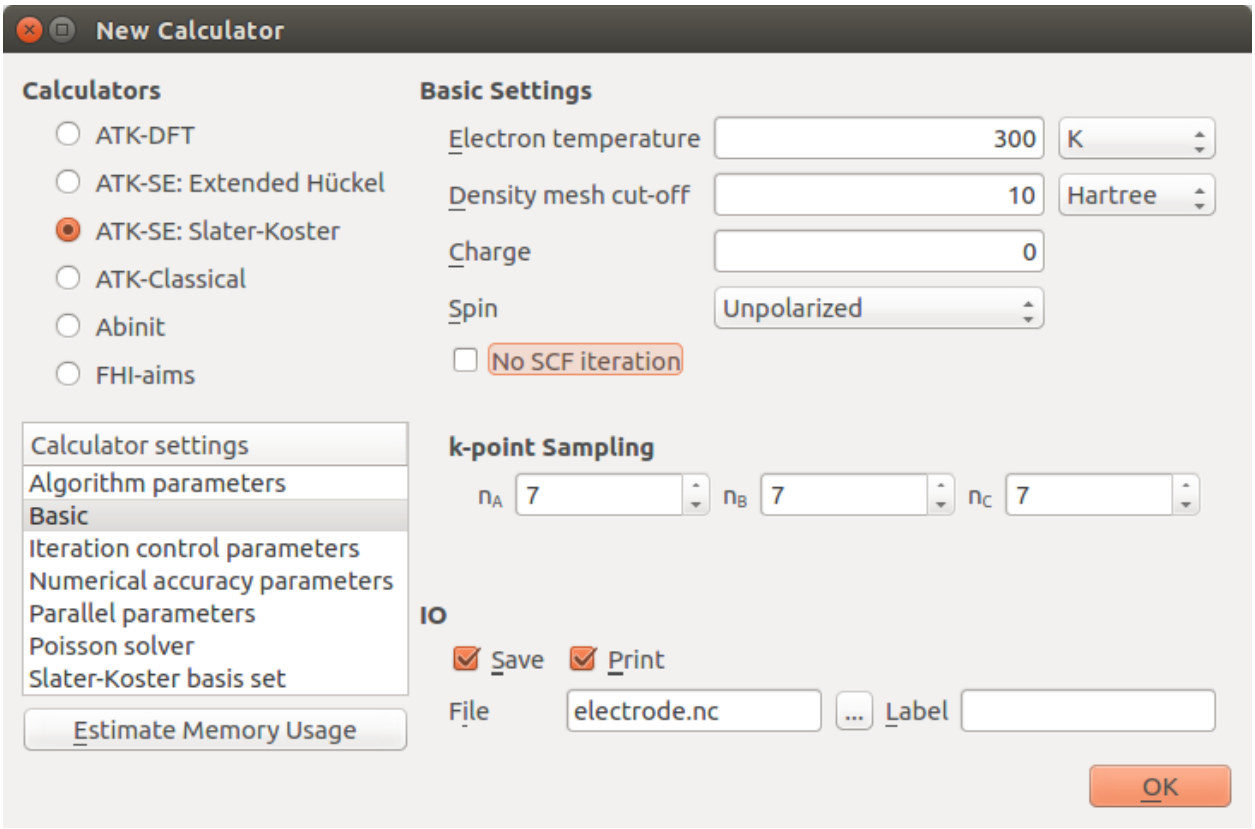
After having performed the electrode calculator convergence tests, the final preparatory step consists of checking the electrode geometry, using the **Electrode Validator** plugin. This tool checks if the electrode configuration has the appropriate geometry (*viz.* if the C axis is parallel to the Z coordinate axis, and if the AB plane is perpendicular to C) and if the electrode is long enough along C.

You first need to do a ground state calculation for the electrode. Follow these steps to set up the calculation:

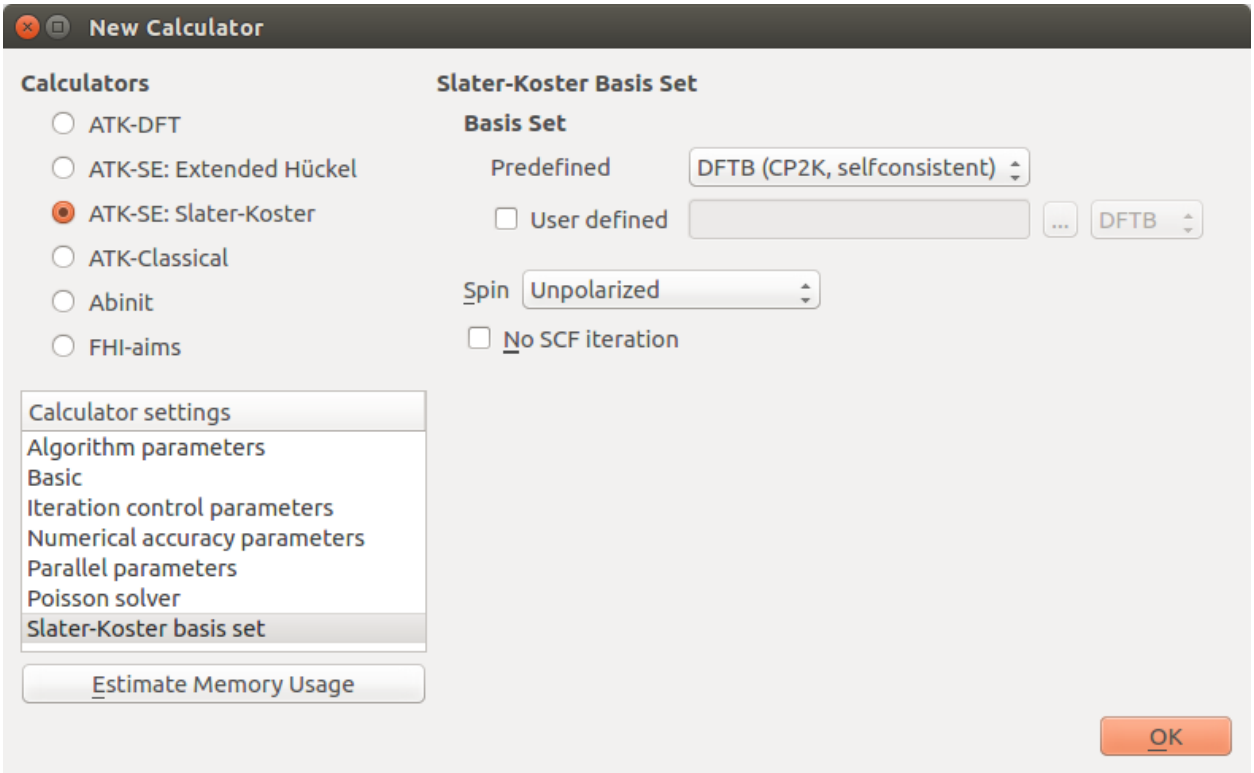
•Open the Builder and select the left electrode. Then use the botton to send the electrode to the **Script Generator**. Change the default output file name to `electrode.hdf5`.



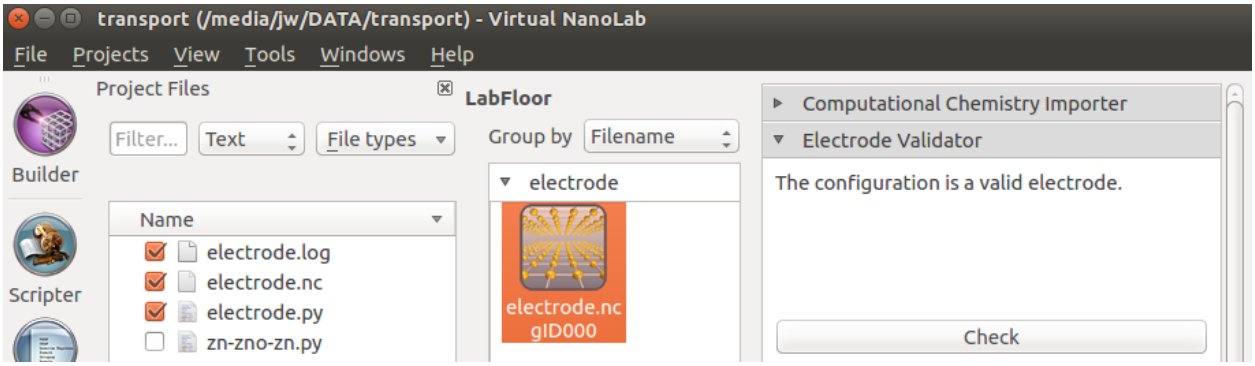
•Add a **New Calculator** block and double-click it to open it. Select the **Slater-Koster** tight-binding calculator engine, make sure No SCF iteration is not checked, and set up a 7x7x7 k-point grid.



•Finally, in the *Slater-Koster basis set* calculator settings, choose the self-consistent DFTB CP2K basis set.



Save the script as `electrode.py` and run the calculation. The script can also be obtained here: `electrode.py`. The results should soon appear on the **LabFloor**. Select the bulk configuration saved in the output file (`electrode.hdf5`), open the Electrode Validator plugin in the right-hand plugins toolbar, and click *Check*.



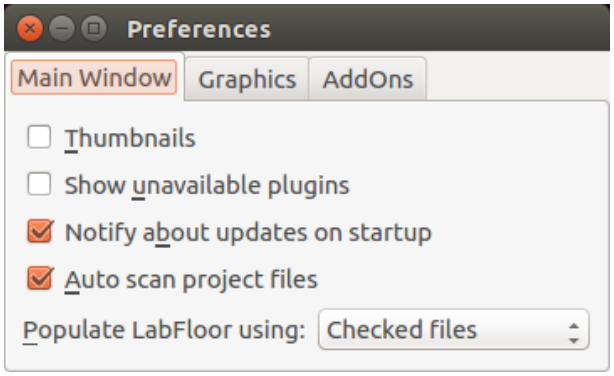
You should see a message saying *The configuration is a valid electrode*, indicating that the electrode geometry and length are fine. In case the electrode for some reason fails the check, a warning and an explanation will be presented.

Note: Whether an electrode is long enough in the C direction depends to some extend on the basis set used. The Electrode Validator uses the criterion that any discarded matrix elements of the Hamiltonian should not exceed 10^{-10} eV.

The reason for this is that only the nearest-neighbor interactions of the electrode along C are kept in the device calculation. Specifically, this is the interaction between the electrode and the electrode extension. This is necessary in order to describe the electrodes as semi-infinite rather than fully periodic along C).

If the electrode is deemed too short, you can still proceed with the device calculation, but, depending on how large the discarded matrix elements are, you risk that the calculation will not converge or that the results are inaccurate.

Tip: The screenshots for this tutorial have been made after unticking the option *Show unavailable plugins* in the *File* → *Preferences* menu in the QuantumATK main window. This causes QuantumATK to dynamically change the list of available plugins in the right-hand plugins bar, such that plugins that are not compatible with the selected LabFloor item are not shown.



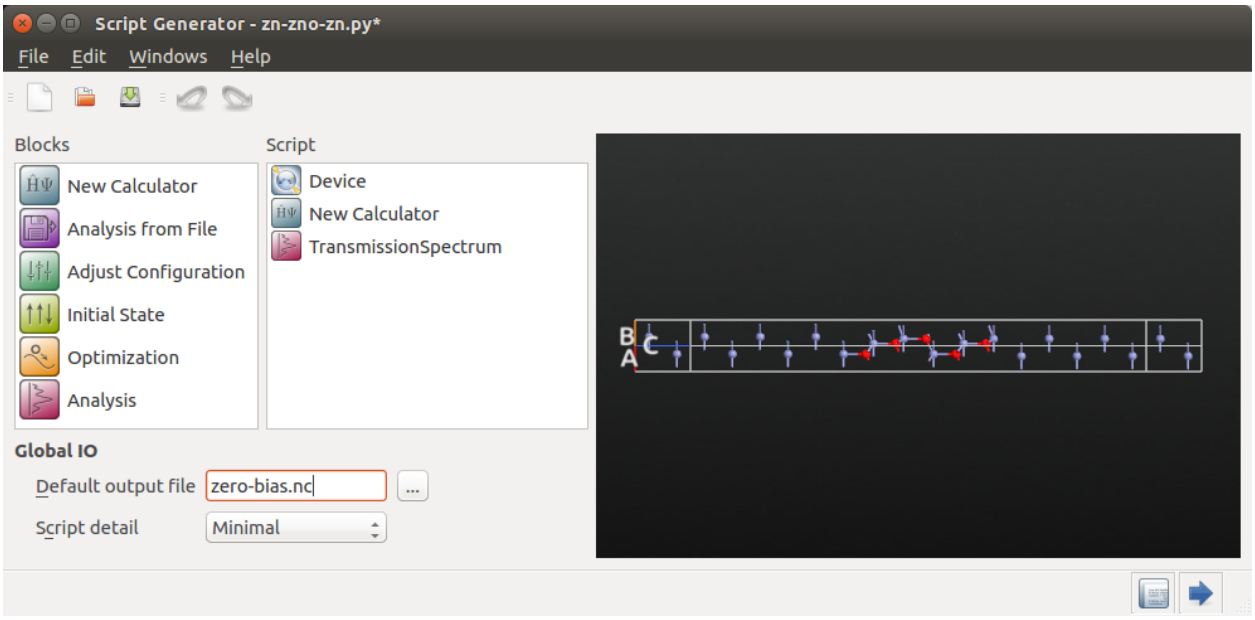
Zero-bias analysis

Having confirmed that the geometry of the electrodes is indeed suitable for QuantumATK device calculations, you are now ready to continue with the actual transport calculations. In the **Builder**, Select the Zn|ZnO|Zn device configuration, named `zn-zno-zn.py`, and send it to the **Script Generator**.

Transmission spectrum analysis

In the Script Generator, follow these steps to set up a script that calculates the electronic transmission spectrum of the Zn|ZnO|Zn device:

- Change the default output file name to `zero-bias.hdf5`.
- Add a New Calculator block to the script.
- Add also the *Analysis* → *TransmissionSpectrum* block.



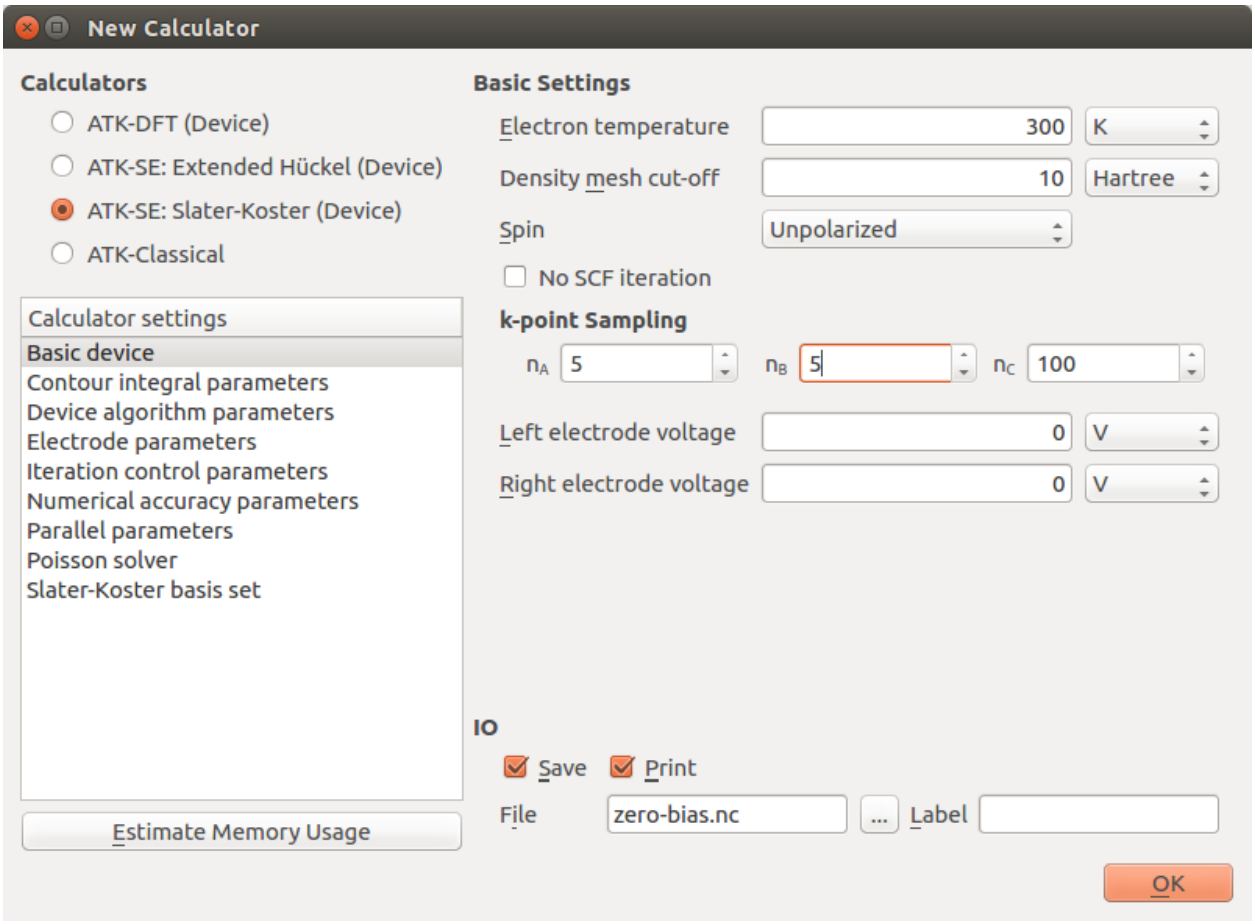
Then edit the calculator settings by double-clicking in the script:

•Choose the **ATK-SE: Slater-Koster (Device)** tight-binding calculator engine.

•Set the transverse k-point samplings (along A and B) to 5.

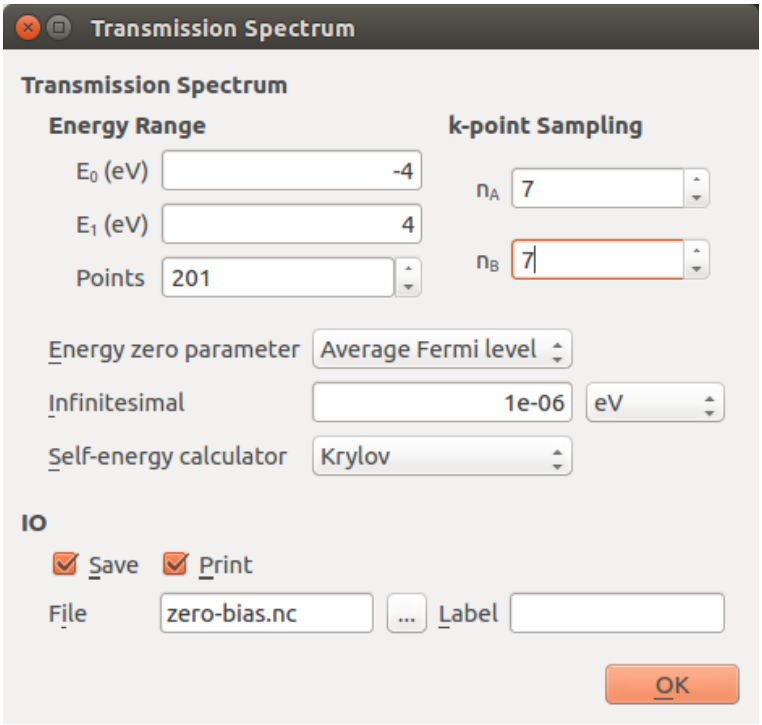
•Uncheck the box *No SCF iteration* to run a self-consistent calculation.

•Verify that the **Slater-Koster basis set** is set to *DFTB CP2K*.



Next, edit the settings in the TransmissionSpectrum block:

- Increase the energy window such that it ranges from -4 eV to +4 eV.
- Increase the number of sampled energy points to 201.
- Set the transverse k-point sampling to 7x7.
- Change the self-energy calculator to *Krylov*.



Note:

- The density of energy points will determine the accuracy of the computed current at finite bias.
- For the energy window, it is recommended to choose a wide interval at zero bias to get a picture of where the transmission peaks are located, but for finite bias, a proper convergence study should be performed, since a too wide energy spacing (too low density of energy points) may mean that narrow peaks in the transmission spectrum are not captured.
- The k-point sampling used in the transmission analysis (and in most post-SCF analysis calculations) does not need to be the same as used in the calculator. Typically, you need more k-points in the transmission analysis than in the SCF calculation.
- The Krylov self-energy calculator is chosen here because it is the fastest method available, but in some cases it can give negative transmission values due to some approximations made.

The script is now ready, so save it as `zero-bias.py`. If needed, you can also download the final script here: `zero-bias.py`.

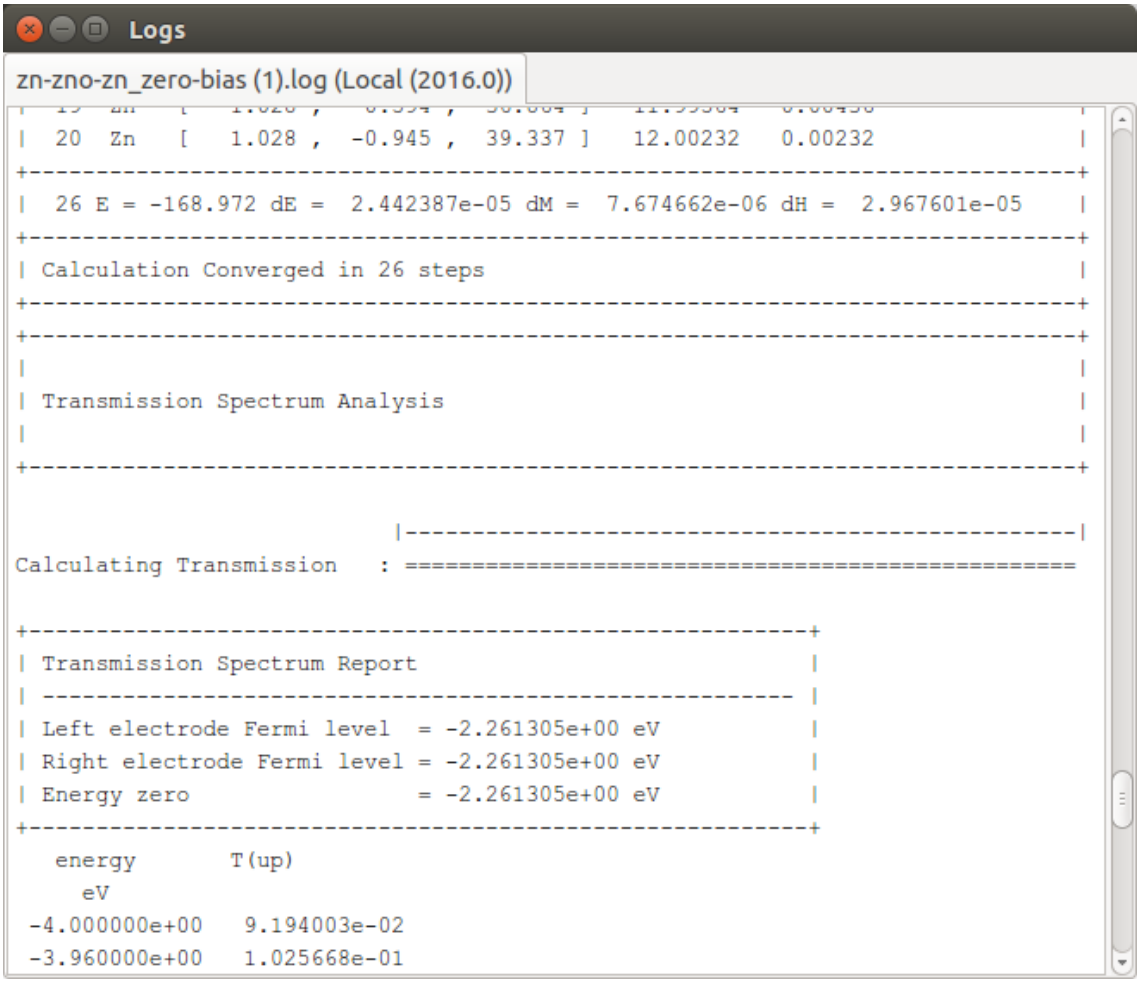
Running the calculation

Execute the script using either the Job Manager or from command line. The calculation should finish in a few minutes.

You should have a look through the job log file to follow the progress of the device calculation. You will see that the following sequence of calculations are carried out:

1. Left electrode SCF (bulk).
2. Right electrode SCF (bulk).
3. Device SCF (NEGF).

4.Transmission spectrum analysis.



Congratulations! You have now performed a complete transport calculation with QuantumATK. The basic work flow is similar for all transport calculations. Please proceed to the next section to visualize and analyze the results.

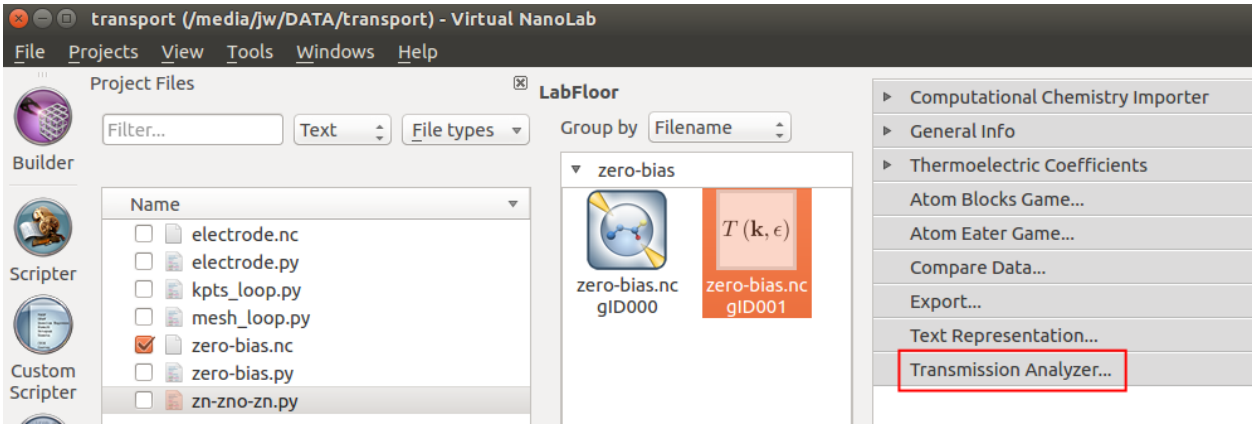
Transmission Analyzer

The calculation output should now have appeared on the **LabFloor**. Locate and select the file `zero-bias.hdf5`. It contains two objects:

DeviceConfiguration contains the two-probe device configuration and the self-consistent state of the attached calculator.

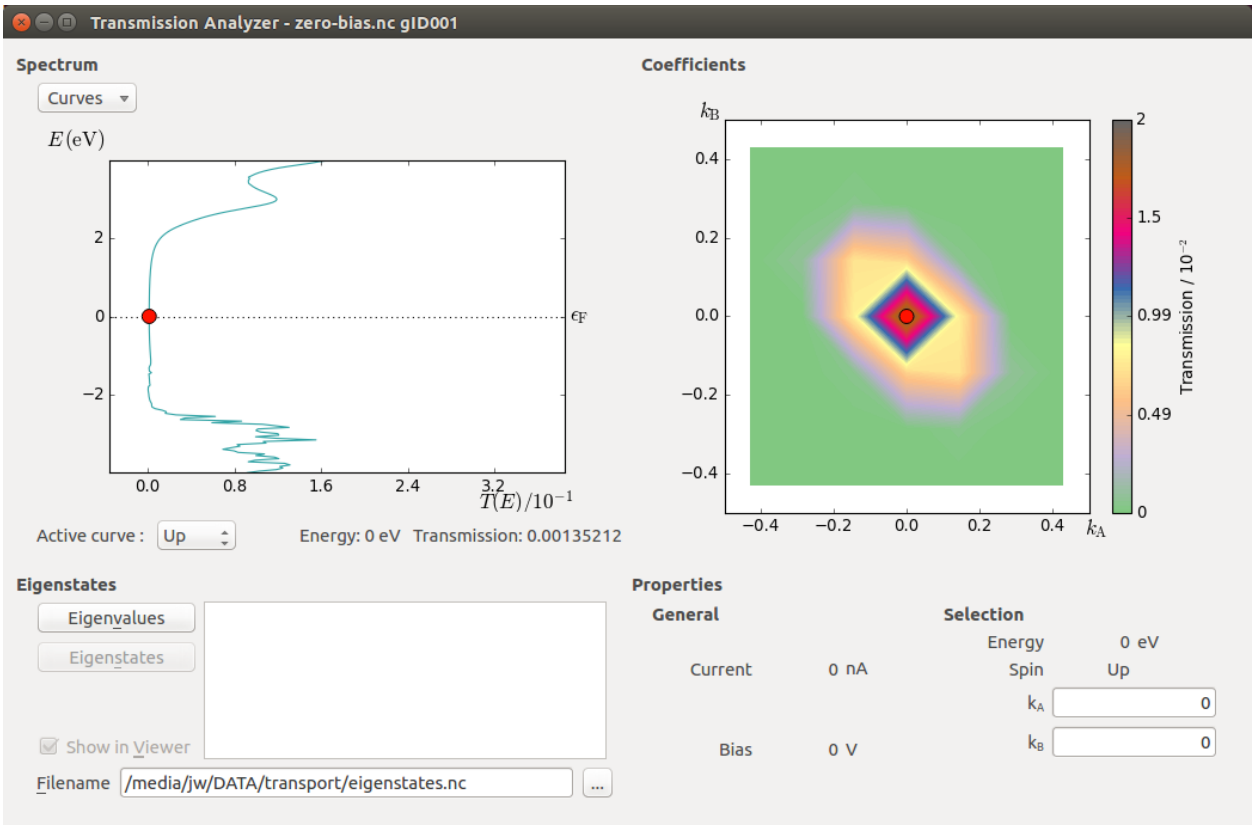
$T(\mathbf{k}, \epsilon)$

TransmissionSpectrum contains all information about the computed transmission spectrum, including settings and results.



Select the TransmissionSpectrum object, and click the **Transmission Analyzer** in the right-hand panel bar.

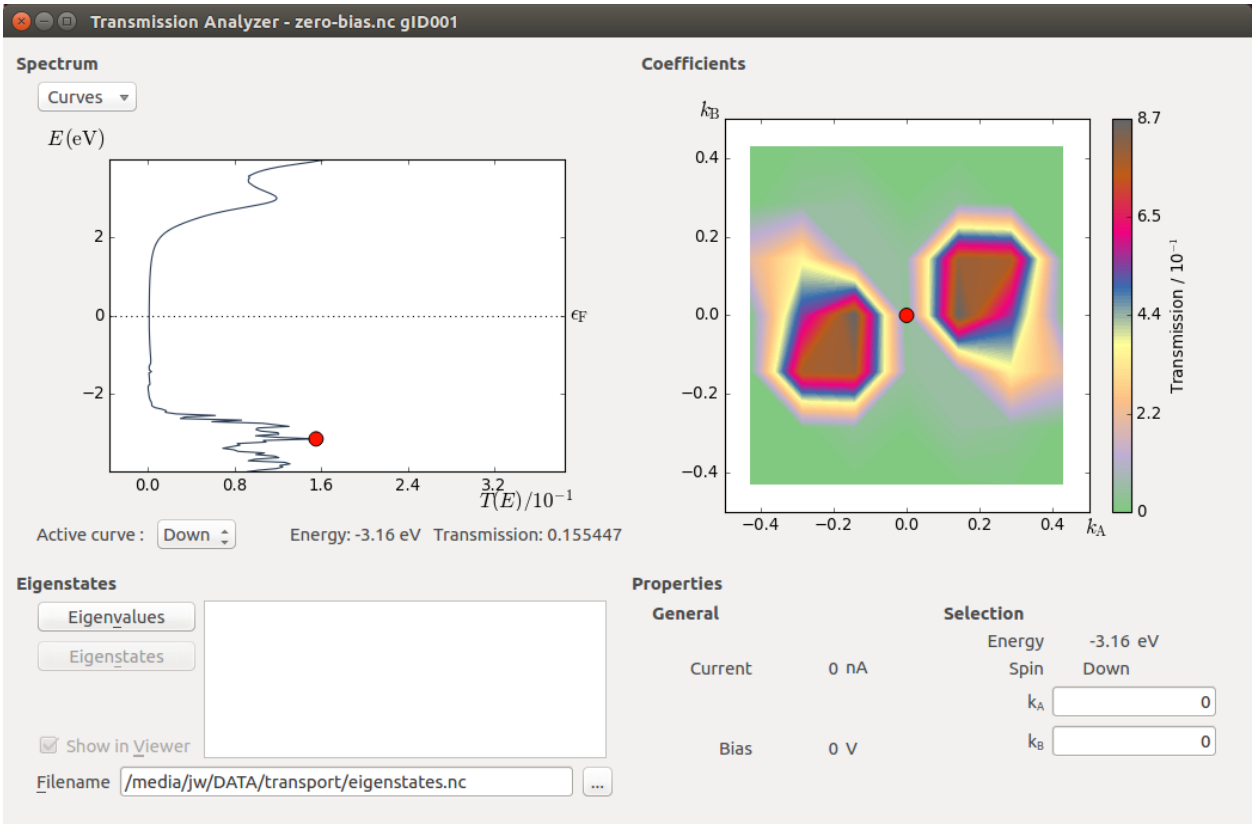
The Transmission Analyzer plugin opens. On the left, named **Spectrum**, is shown the k-point averaged transmission spectrum as function of energy. The average Fermi energy of the two electrodes is set to zero and indicated with a dashed line. It is evident that there is a broad energy range from -2.5 eV to +2.5 eV with very low transmission values. This corresponds to the band gap in the semiconducting ZnO.



The transmission and energy values in the point marked by the red dot are shown below the plot. You can use the mouse to move the dot along the $T(E)$ curve and read off the values. The *Curves* drop-down menu lets you select the spin components to be plotted. Note that the transmission for spin types X, Y, and Z are only non-zero for calculations with noncollinear spin.

At any particular energy (defined by the red dot), the total transmission is computed as an average over the transmission coefficients in the sampled k-points. The right-hand plot, named **Coefficients**, shows an interpolated contour plot of the transmission coefficients vs. reciprocal vectors k_A and k_B.

In the figure above, it is clear that the main contributions to electronic transmission through the Zn|ZnO|Zn device at the average Fermi level come from k-points around the Γ-point, where $k_A = k_B = 0$. The situation is very different at other energies, such as e.g. E = -3.16 eV, as shown in the figure below.



Tip: By increasing the number of k-points sampled in the transmission spectrum calculation, the resolution of the contour plot can be improved – and thereby also the accuracy of the total transmission spectrum. In many devices, a large number of k-points may be required to obtain a converged spectrum.

Transmission eigenstates

By clicking on the contour plot you can also move the red dot to a desired k-point. The chosen k-point is shown in the **Selection** box. At a particular energy and spin (chosen in the **Spectrum** plot) and a particular k-point (chosen on the **Coefficients** contour plot) you can obtain the transmission eigenvalues and eigenstates:

- Select a k-point with high transmission value; (k_A, k_B) = (0.15, 0.05) at energy -3.16 eV.
- Click *Eigenvalues* in the lower left part of the window.

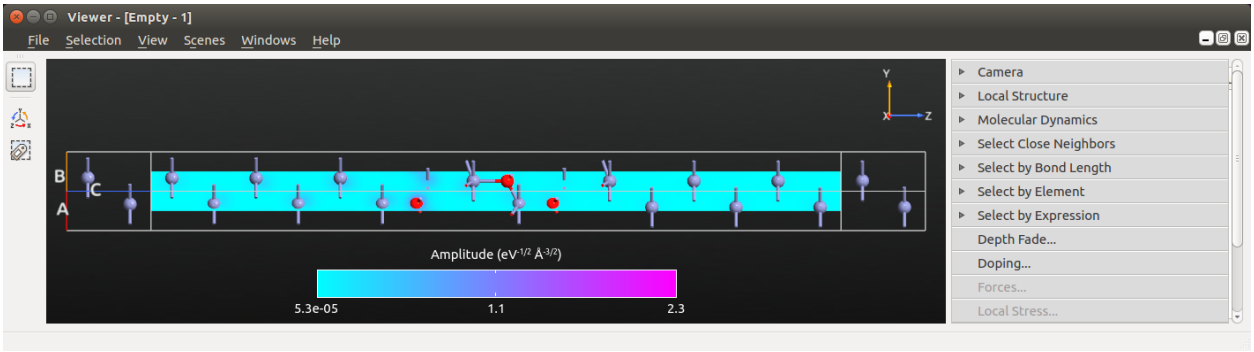
The transmission eigenvalues for the selected energy, k-point and spin are calculated, and after a couple of seconds the eigenvalues are listed.

Hint: If several transmission channels contribute to the transmission, more than one eigenvalue will be shown. Each eigenvalue is restricted to the range [0, 1], but the sum of the eigenvalues, which equals the k-point resolved transmission $T(E, \mathbf{k})$, may be larger than 1, just like the total transmission $T(E)$ can be.

You can now calculate and visualize the transmission eigenstate for each eigenvalue:

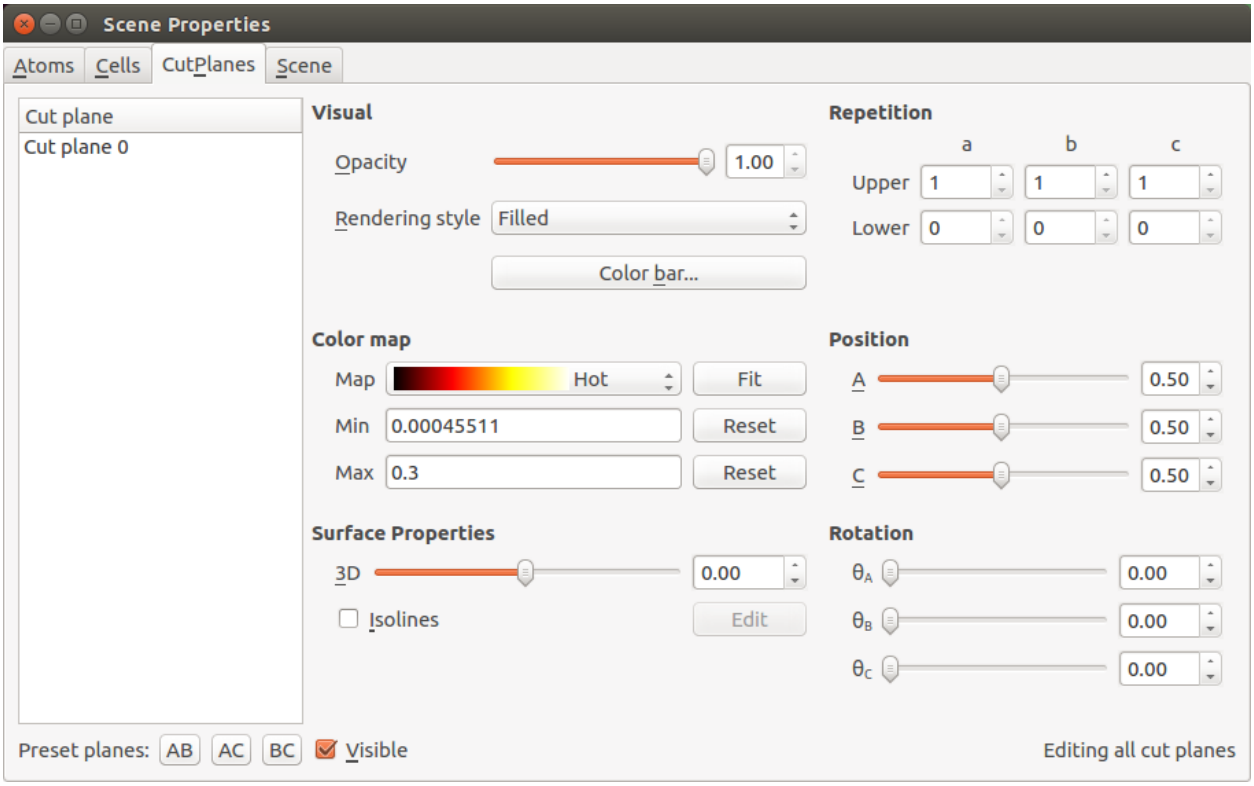
- Select an eigenvalue and click *Eigenstates*. The **3D viewer** opens after a few seconds.
- Select the **Cut plane** plotting type and click *OK*.

The Viewer now displays the Zn|ZnO|Zn device configuration and a cut-plane representation of the transmission eigenstate throughout the device central region.

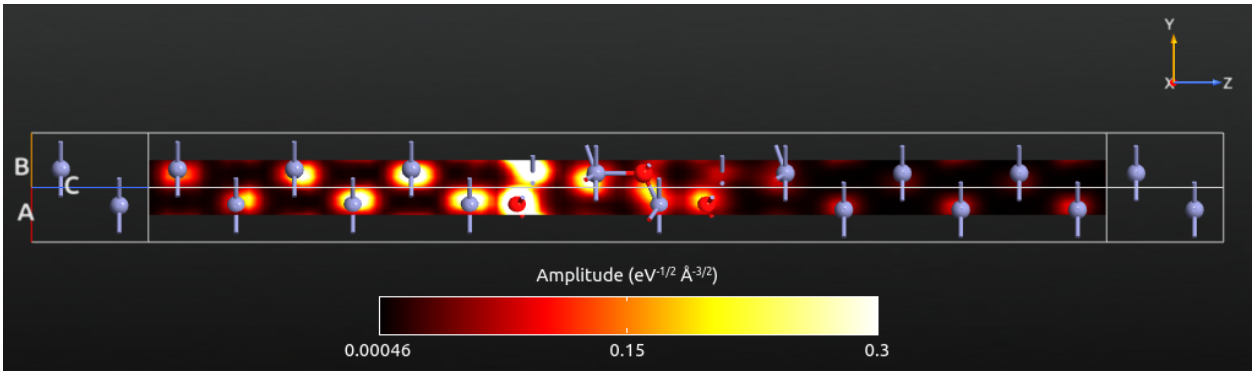


Next, edit the visual properties of the cut-plane to focus the color palette on the finer features of the eigenstate:

- Open the **Properties** editor in the right side of the Viewer.
- In the *CutPlanes* tab, change the colormap to Hot, then click *Fit*, and enter 0.3 for the maximum eigenstate value to be visualized.

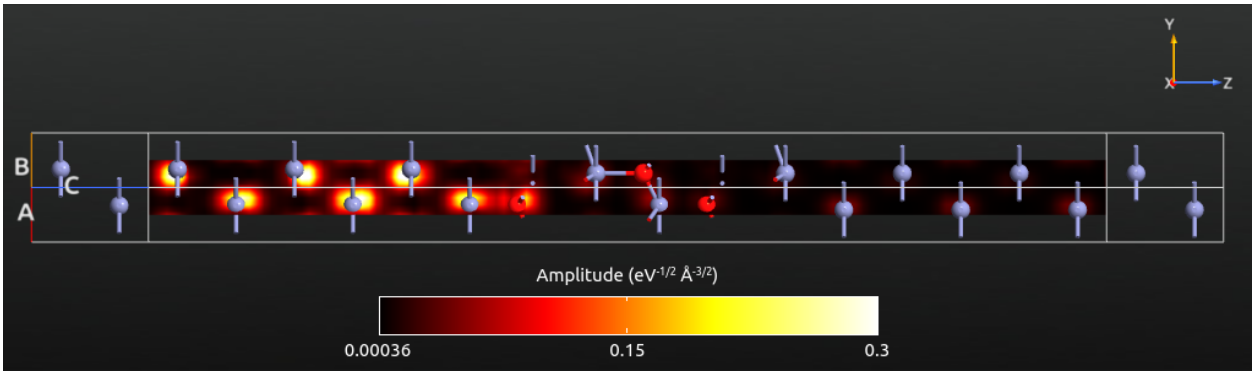


The cut-plane visualization should now look as illustrated below. The eigenstate corresponds to a scattering state coming from the left electrode and traveling towards the right. The eigenstates will therefore always have a large amplitude in the left-hand part of the central region. Eigenstates with a relatively high transmission eigenvalue will also have a relatively large amplitude in the right-hand part of the scattering region. This simply reflects the relatively high transmission probability of an incoming state (at a specific energy, spin, and k-point) that attempts to travel through the central region and into the right electrode.



The transmission eigenstates in the left part of the scattering region are composed of an incoming (right-moving) and a reflected (left-moving) state. This leads to interference effects, which can be observed in the figure above: The eigenstate amplitude around the first Zn atom is relatively small due to destructive interference between the incoming and reflected states. To the right of the ZnO, the eigenstate consists only of the transmitted (right-moving) states, and no such interference effects are observed.

Try yourself to plot the transmission eigenstate at the Fermi level (energy E = 0 eV) and $k_A = k_B = 0$. By adjusting the plot properties you can get a plot similar to the one shown below. Now the eigenstate has a vanishing amplitude in the middle and right parts of the scattering region, due to the low transmission eigenvalue, i.e. only a tiny part of the scattering eigenstate is transmitted, and most of it is reflected at the Zn–ZnO interface.



Tip: If you instead plot the transmission eigenstate as an isosurface, the isovalue corresponds to the magnitude of the wave function, and the phase is used to color the isosurface.

Convergence of k-points for transmission spectrum

As already mentioned, more k-points are typically needed to obtain a well-converged transmission spectrum than for the SCF calculation. As an example, consider the figure below, which shows the total transmission (averaged over all k-points) evaluated at three different energies and plotted as function of the number of k-points in the transverse A and B directions. The figure was generated using the script `transmission_kpts.py`.

The transmission converges very differently at different energies. While the transmission at the Fermi energy is converged with just 5 or 7 k-points in the A and B directions, at -3 eV and +3 eV it looks like at least 13 k-points are needed.

You may recall from above that the transmission spectrum has many sharp peaks in the region around -3 eV. It is typically such, that many k-points are needed when sharp features are present. However, there may also be isolated peaks that are hidden by too low k-point density, so one really needs to study the k-resolved transmission plot to get an idea of the needed k-point sampling.

Additionally, in a region with many (sharp) transmission peaks as function of energy, a large number of energy points are required to converge the integral over energy needed to compute the current.

Further analysis

This section will introduce some further ways to analyze the zero-bias calculation. An important feature in QuantumATK is that you do not have to define all analysis quantities from the outset; you can always calculate additional quantities without having to redo the self-consistent calculation from scratch.

You will now compute the device density of states (DDOS) of the Zn|ZnO|Zn configuration, and the electrostatic difference potential:

- Open a new **Script Generator** window.
- Change the output file name to `analysis.hdf5`, and add three blocks to the script:
  - Analysis from File
  - DeviceDensityOfStates

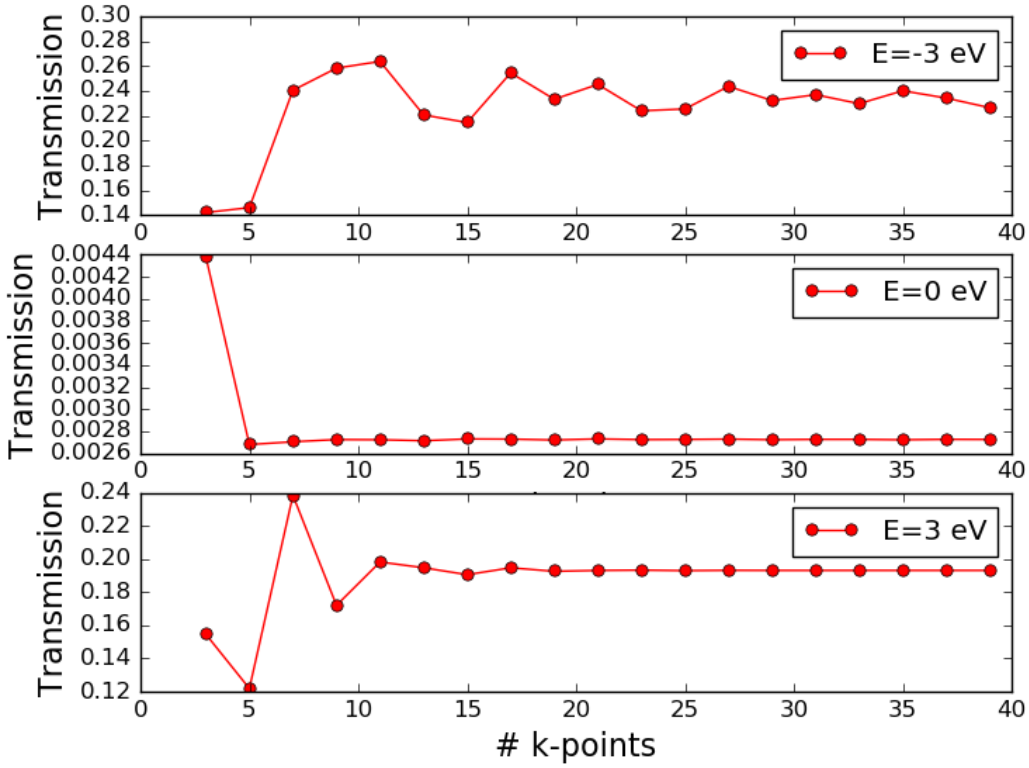
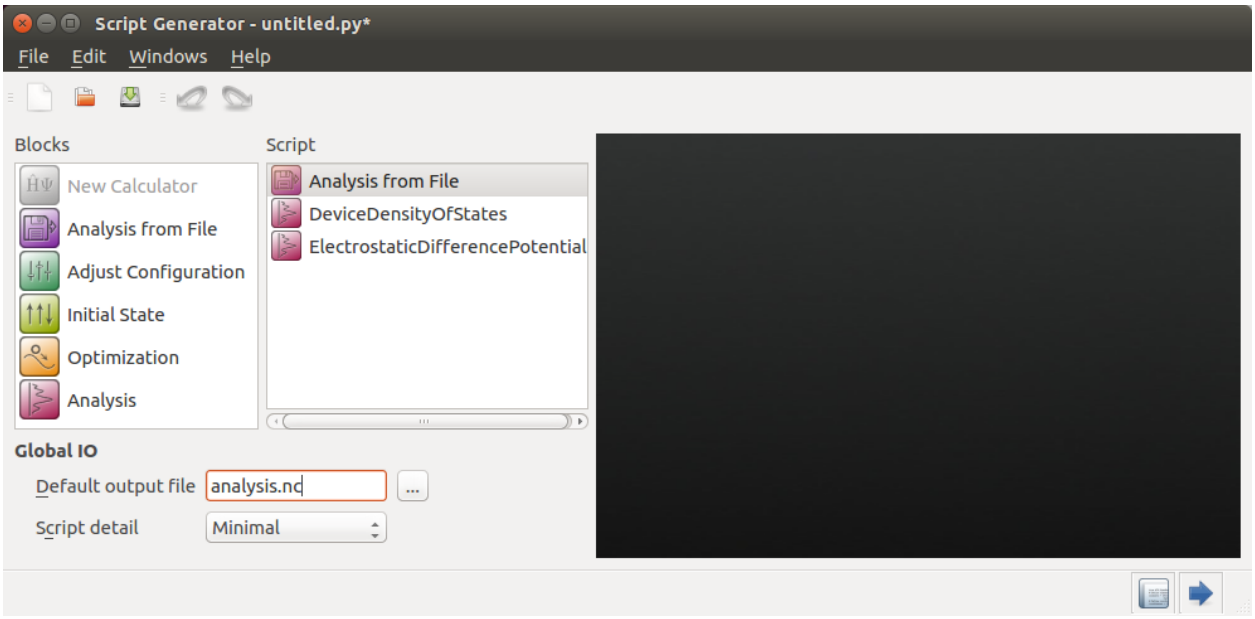


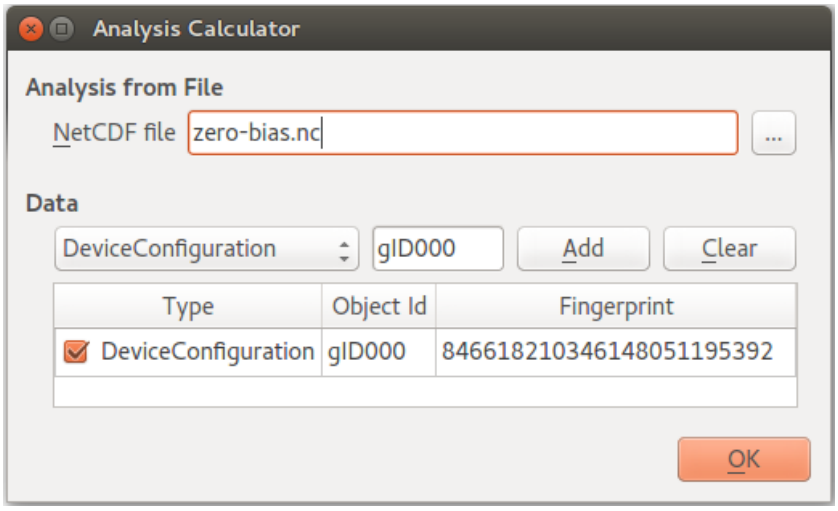
Fig. 3.124: Total electronic transmission vs. the number of k-points in the transverse directions for the Zn|ZnO|Zn device evaluated at three different energies: -3 eV (top), 0 eV (middle), and 3 eV (bottom).

– ElectrostaticDifferencePotential



•Open the Analysis from File block. This is the part of the QuantumATK script that is responsible for reading the converged state of the zero-bias calculation. You need to specify which file and which saved configuration you want to base the following analysis on (there may be several configurations in a single file).

•Select the saved file `zero-bias.hdf5` – the configurations stored in that data file are then automatically detected and listed. Only one configuration is in this case available; it has object ID gID000. Make sure that configuration is ticked, and click *OK*.



Tip: The ID number of each object is indicated below its icon on the LabFloor. To distinguish different configurations in the same file, you can use the **General information** plugin to inspect the most important parameters of the calculation.

•Next, edit the DeviceDensityOfStates settings, which should be similar to the settings used for computing the

transmission spectrum:

–energy range from -4 eV to +4 eV with 201 points.

–7x7 k-point sampling.

–Krylov self-energy calculator.

•No changes need be made to the ElectrostaticDifferencePotential analysis.

The analysis script is now ready. Save it as `analysis.py` and execute it using the Job Manager or from command line – it takes only a few minutes to run. You can also download the script here: `analysis.py`.

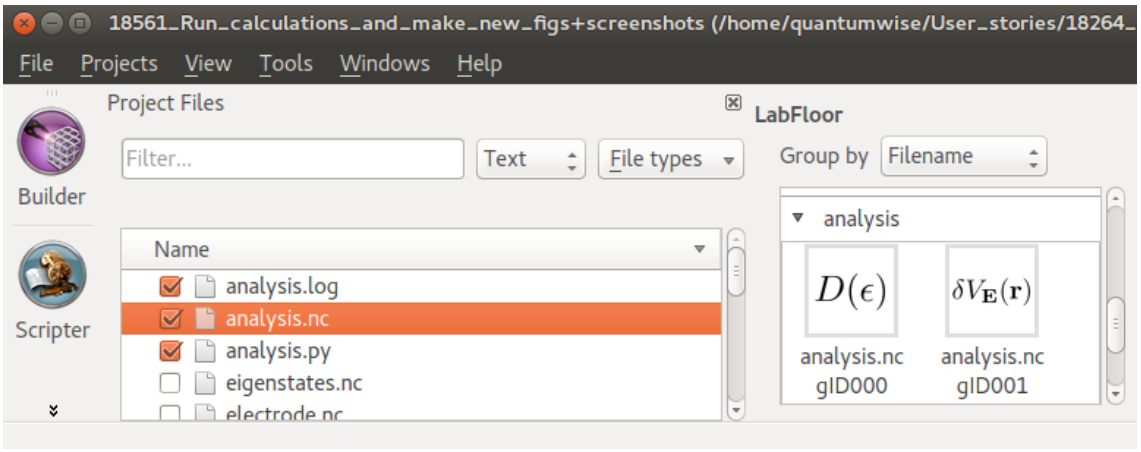
When the job is done, observe that the analysis results have appeared on the LabFloor:

DeviceDensityOfStates

$D(\epsilon)$

ElectrostaticDifferencePotential

$\delta V_{\mathbf{E}}(\mathbf{r})$



Device density of states

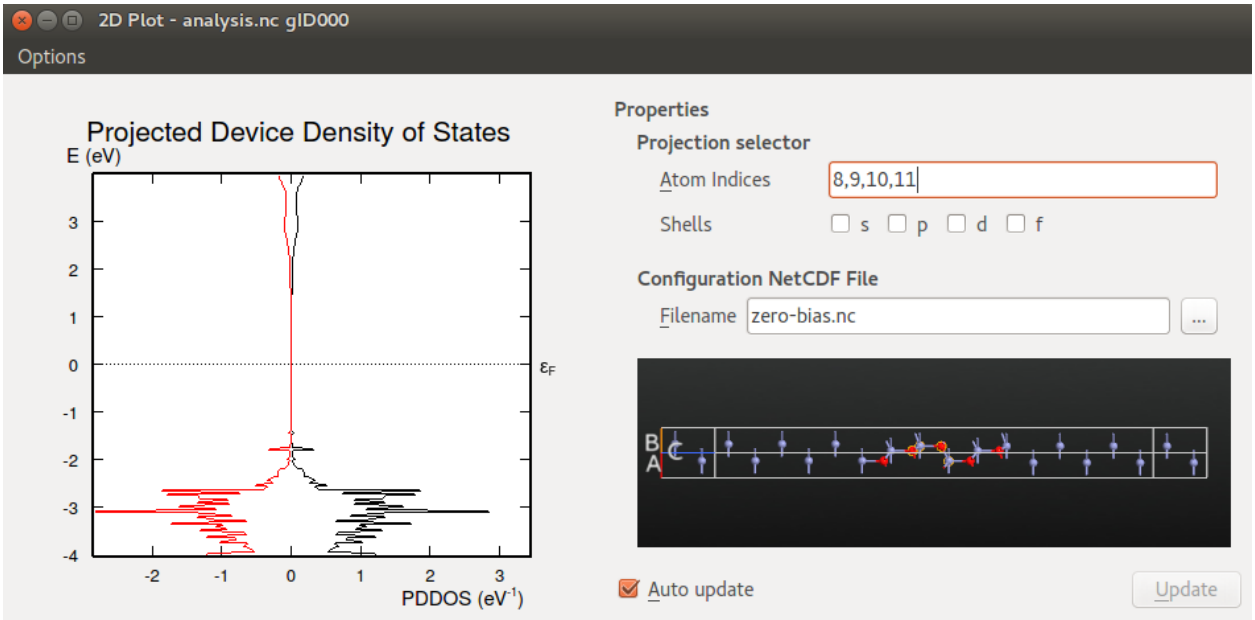
Select the DeviceDensityOfStates object and click **2D plot** in the right-hand plugins bar to visualize it.

The 2D Plot window opens with the DDOS plottet on the left. The right-hand panel allows you to also visualize the configuration and select atoms and orbitals to project the DOS onto:

•Under **Configuration HDF5 File**, select the output file `zero-bias.hdf5` for visualizing the device configuration (which is not saved in `analysis.hdf5`).

•Under **Projection selector**, select atom indices 8,9,10,11 in order to preject the DOS onto the middle part of the ZnO layer.

•Then zoom in on the projected DOS like shown in the figure below.



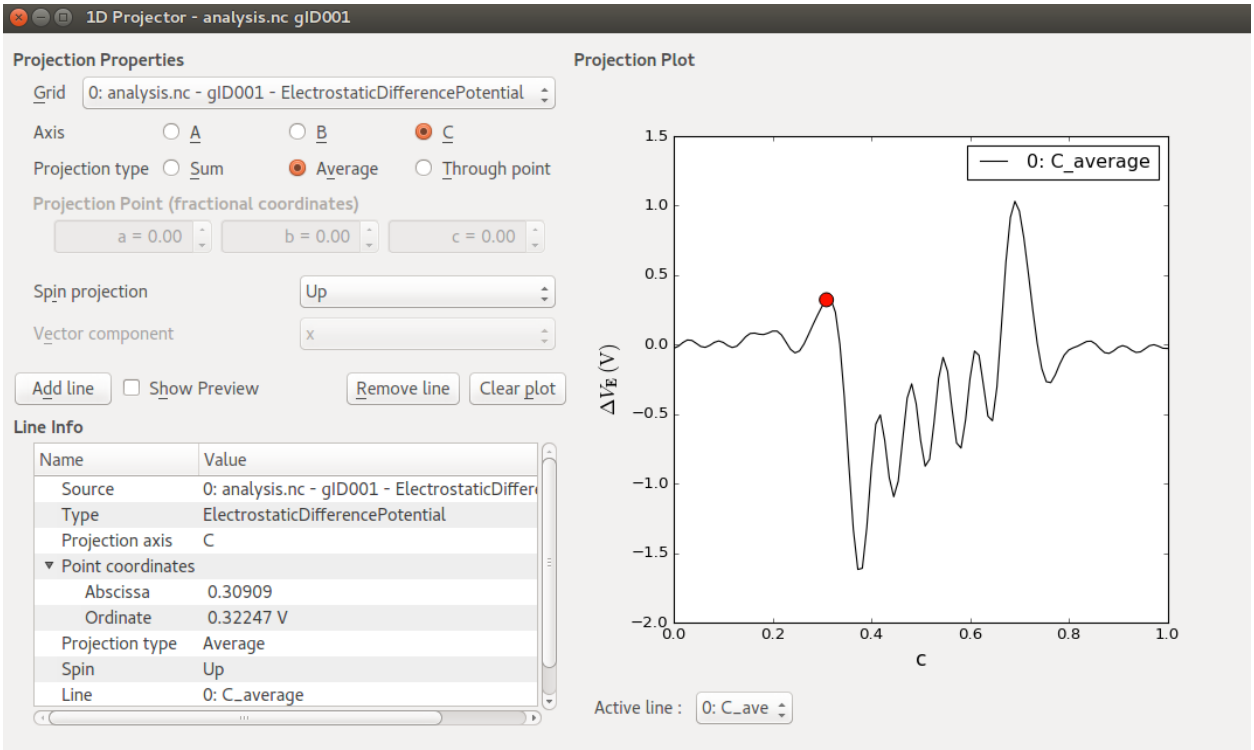
You should notice that the shape of the DDOS resembles the shape of the transmission spectrum analyzed above: Both plots show a broad energy region around the Fermi energy with low values, reflecting the semi-conducting nature of ZnO; for energies below -2 eV there are many sharp peaks; finally, both the transmission spectrum and the DDOS display a broad peak around +3 eV. The correspondence between the transmission spectrum and DDOS is a very general feature often observed in device calculations.

Electrostatic difference potential

The ElectrostaticDifferecePotential object contains a 3D grid that represents the difference between the electrostatic potential of the self-consistent valence charge density (the solution to the Poisson equation with this density) and the electrostatic potential from a superposition of atomic valence densities. You will use this object later when calculating the voltage drop.

You can of course use the **Viewer** to visualize such 3D grids, just like you did for the transmission eigenstates a few sections up. Another way to visualize a quantity represented on a 3D grid is to use the **1D Projector**. With this tool you can project a potential or density along a certain direction, integrated or averaged over the two directions perpendicular to it. For devices, it is often most interesting to plot the potential along the transport direction (C) and to average over the A and B directions.

- Select the ElectrostaticDifferencePotential object on the LabFloor, and open the 1D Projector from the right-hand plugins bar.
- Change Projection type to *Average* and click *Add line*.



You should now see a plot like shown in the figure above, where the electrostatic difference potential, ΔV_E, is plotted against the fractional C-coordinate (ranging from 0 to 1).

A significant charge transfer seems to take place at both Zn–ZnO interfaces, causing the self-consistent electrostatic potential to differ significantly from that of the atomic valence densities. In the metallic Zn regions near the electrodes, oscillations are rather small, indicating that the self-consistent charge distribution is not much different from the atomic valence charges in those regions.

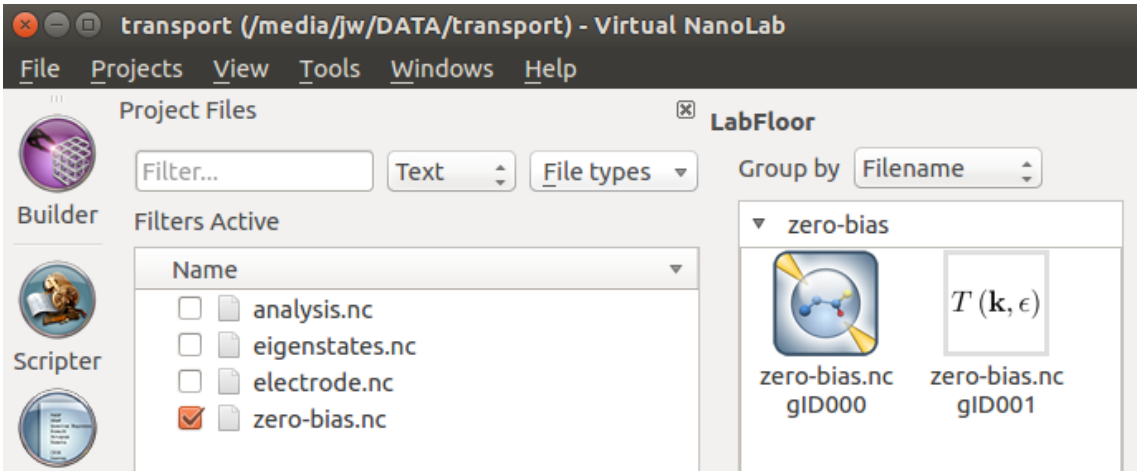
Finite-bias calculations

All calculations performed so far have been with zero voltage on the left and right electrodes, i.e. at zero bias. In this section you will use the zero-bias calculation as a starting point for finite-bias calculations, where the left and right electrode voltages are different. This shifts the left and right electrode Fermi levels relative to each other – a basic mode of operation for electronic devices.

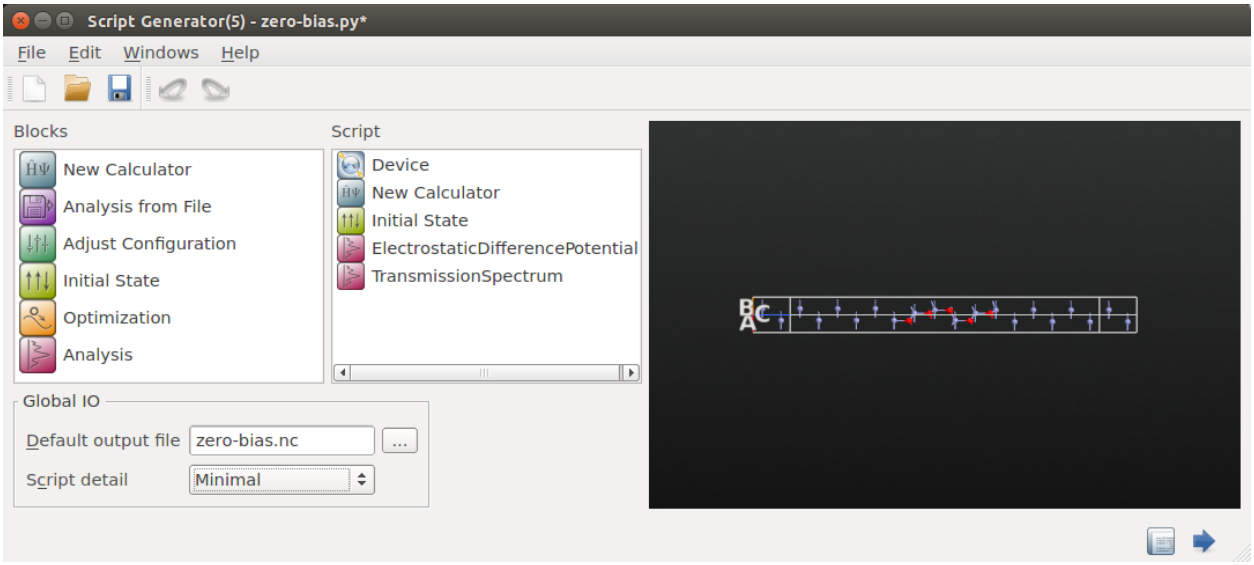
The recommended method for obtaining fast and reliable SCF convergence for finite-bias QuantumATK transport calculations is to start out from a zero-bias calculation and change the bias in small steps. You will now perform a device calculation at -0.5 V bias.

Set up the calculation

In the QuantumATK Project Files list, tick the file `zero-bias.hdf5` to see its contents on the LabFloor, and drag the DeviceConfiguration (*gID000*) onto the **Script Generator**. This will setup the Slater-Koster calculator parameters identically to those used for the zero-bias calculation.

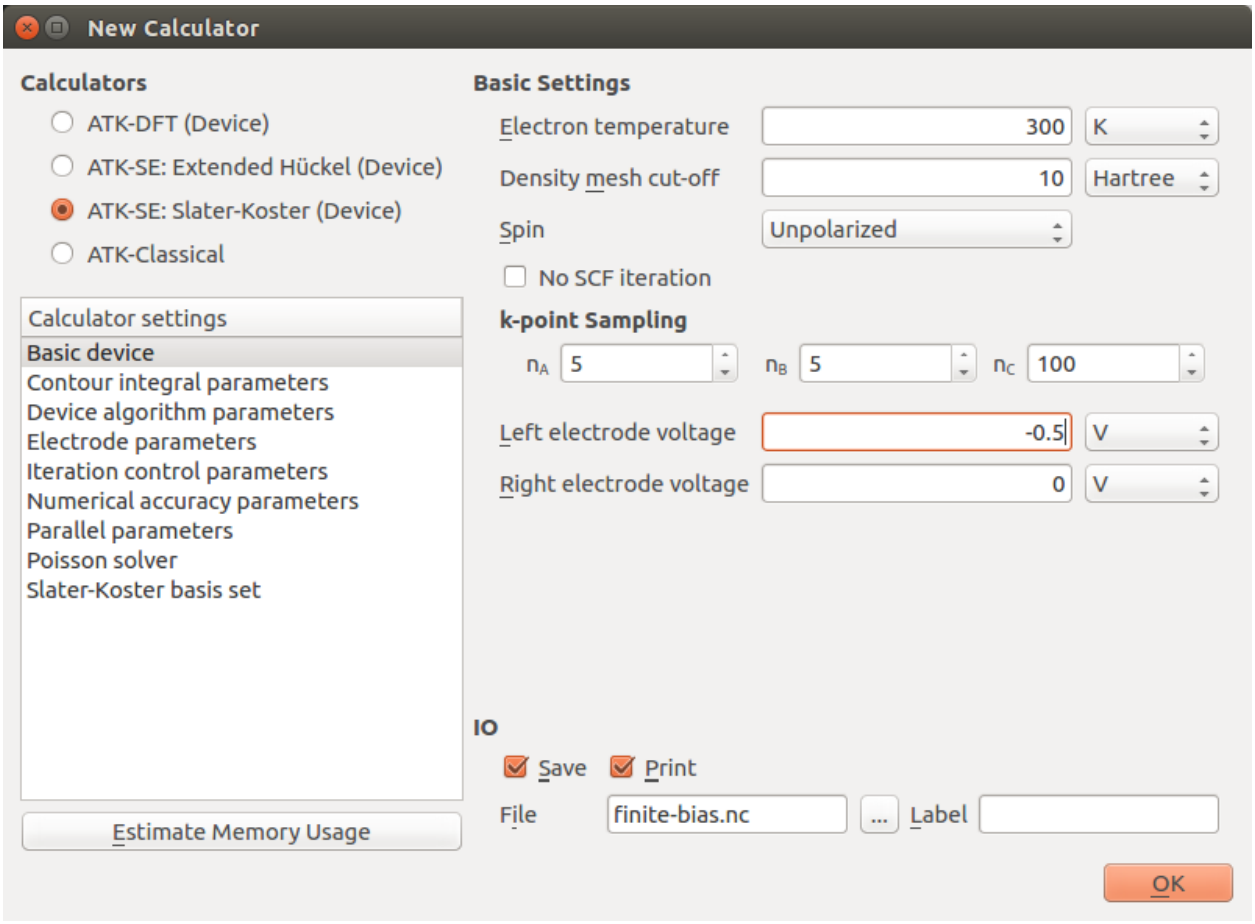


Add three blocks to the script: Initial State, *Analysis → ElectrostaticDifferencePotential*, and *Analysis → TransmissionSpectrum*.



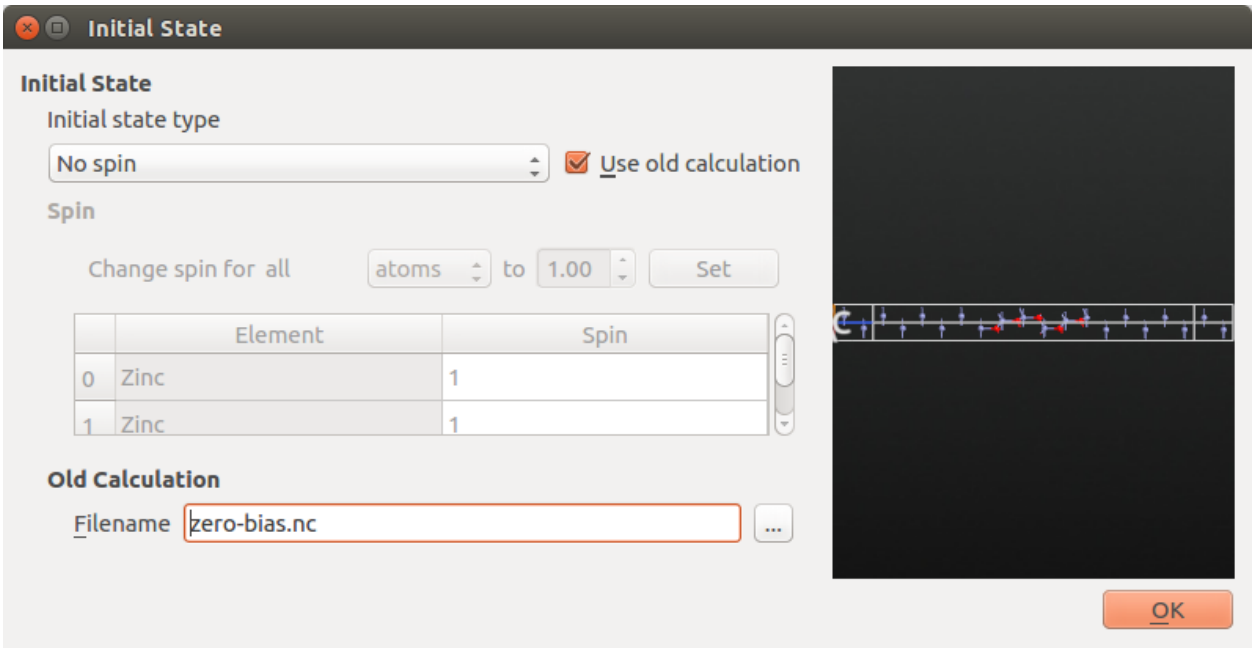
Then change the default output file to `finite-bias.hdf5`, and open the New Calculator to apply a bias:

•Set the **Left electrode voltage** to -0.5 V.



Next, open the Initial State block to instruct QuantumATK to use the zero-bias state as an initial guess for the finite-bias calculation:

- Tick **Use old calculation**, and choose `zero-bias.hdf5` as the filename of the *Old Calculation*.



Lastly, set up the TransmissionSpectrum parameters identically to those used for the zero-bias calculation:

- Energy window of -4 eV to +4 eV with 201 energy points.
- 7x7 transverse k-points.
- Krylov self-energy calculator.

Save the script as `finite-bias.py` and run it. The calculation will take a little more time than the zero-bias calculation. If needed, you can download the final script here: `finite-bias.py`.

Note: Electrode voltages and current In a zero-bias calculation, the local chemical potential is constant throughout the device, including the electrodes. Applying a potential difference $U = V_{\mathrm{L}} - V_{\mathrm{R}}$ between the electrodes is identical to shifting the electrode Fermi levels with respect to each other,

$$eU = e(V_{\mathrm{L}} - V_{\mathrm{R}}) = \mu_{\mathrm{R}} - \mu_{\mathrm{L}},$$

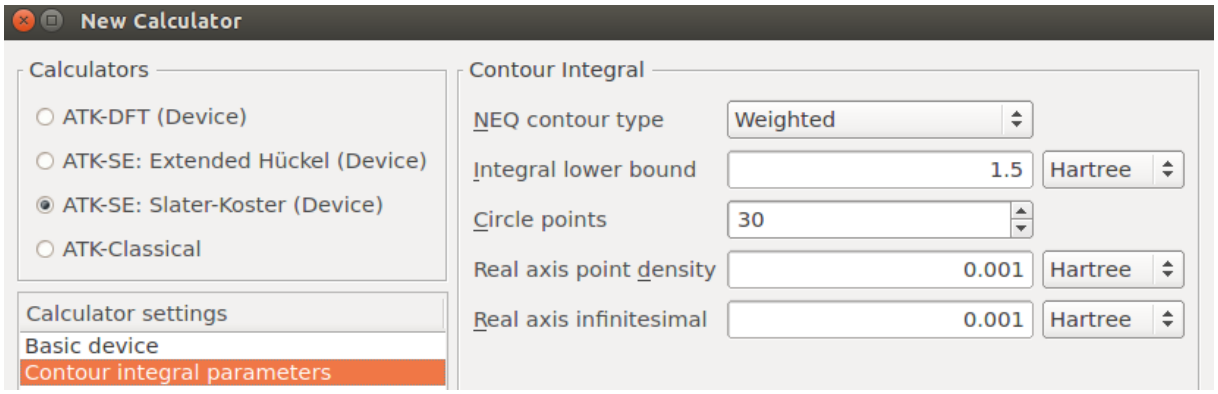
where μ_{L} and μ_{R} are the left and right electrode Fermi levels, respectively, and e the (positive) elementary charge.

The **bias voltage** U is therefore positive when the applied potential on the left electrode is higher than on the right electrode. This situation is also called *forward bias*, and implies that the chemical potential in the left electrode is moved down such that $\mu_{\mathrm{L}} < \mu_{\mathrm{R}}$. Conversely, under *reverse bias*, we have $V_{\mathrm{L}} < V_{\mathrm{R}}$, and therefore a negative bias voltage and opposite ordering of the chemical potentials, $\mu_{\mathrm{L}} > \mu_{\mathrm{R}}$.

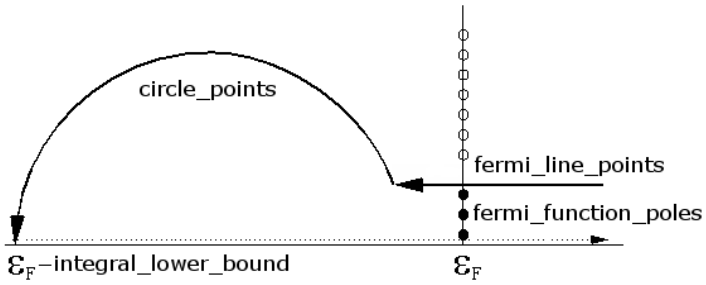
A positive **current** corresponds to charges moving from left to right, meaning that electrons move from right to left (since electrons have a negative charge). The current is therefore *positive under forward bias* and *negative under reverse bias*, so you should expect to get a negative current with the bias setting specified above.

Type	Bias voltage	Electrode potentials	Chemical potentials	Current
Forward	$U > 0$	$V_{\mathrm{L}} > V_{\mathrm{R}}$	$\mu_{\mathrm{L}} < \mu_{\mathrm{R}}$	$I > 0$
Reverse	$U < 0$	$V_{\mathrm{L}} < V_{\mathrm{R}}$	$\mu_{\mathrm{L}} > \mu_{\mathrm{R}}$	$I < 0$

Attention: NEGF contour integral For finite-bias calculations, a potentially important convergence parameter is the *Real axis point density*, which is specified under *Contour integral parameters*. This parameter determines how many energy points are used when integrating the left and right spectral density matrices along the real energy axis between the chemical potentials of the left and right electrodes. For the present calculation you can proceed with the default value, but be aware that a convergence test of this parameter should in general be performed.

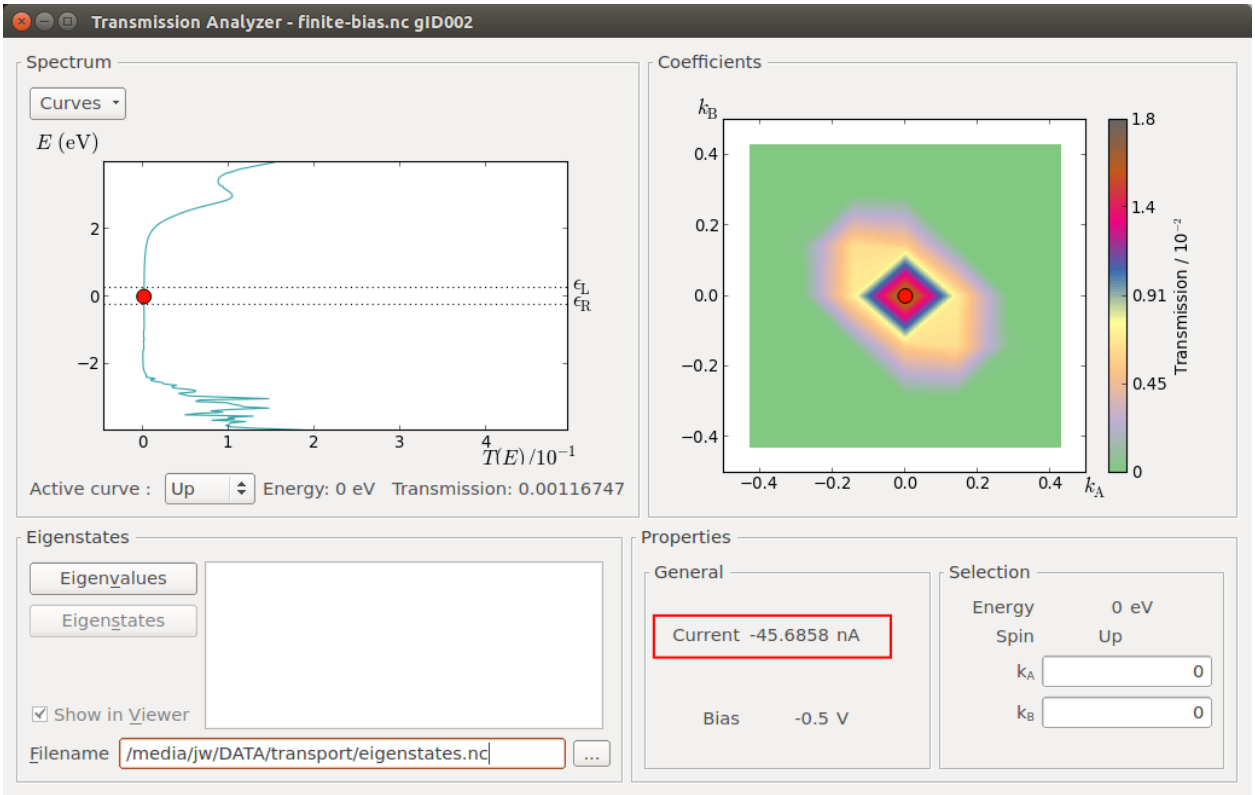


The settings *Integral lower bound* and *Circle points* are another important set of parameters. The NEGF contour integral needs to include all eigenvalues on the energy axis, but these are in general not known *a priori*. The default lower bound for the integral is 1.5 Ha below the Fermi level, but for DFT calculations with hard pseudopotentials (many valence electrons, possibly semi-core states), there may well be eigenvalues deeper than that. In that case, the *Integral lower bound* should be increased, and possibly also the number of circle points. Poor device SCF convergence is often due to such issues.



Analyzing the results

Go to the QuantumATK main window and open `finite-bias.hdf5` on the LabFloor. Use again the **Transmission Analyzer** to plot the transmission spectrum. Notice that the chemical potential of the left and right electrodes are separated by the bias of 0.5 V and distributed evenly around the average electrode Fermi levels, which still is the energy zero reference. Also, you can see that there is a (very small) negative current flowing in this system, consistent with the direction in which the bias was applied.



The finite-bias transmission looks essentially the same as found for zero bias. Recall from the analysis in section *Device density of states* that you found similar shapes of the ZnO DDOS and the transmission function. The shape of the transmission function is thus in this system largely determined by the density of states of the ZnO, and not so much by the electronic structure of the electrodes. Since the transmission values are very small in the bias window, the current is low and hence the change in DDOS in the ZnO is small. This is not necessarily the case in other systems, especially not if the bias is large enough to bridge the band gap of the scattering region.

Voltage drop

Your next task is to calculate the potential profile along the device transport direction at finite bias, also known as the voltage drop or the induced electrostatic potential. This can be computed as the difference between the electrostatic difference potentials at finite and zero bias,

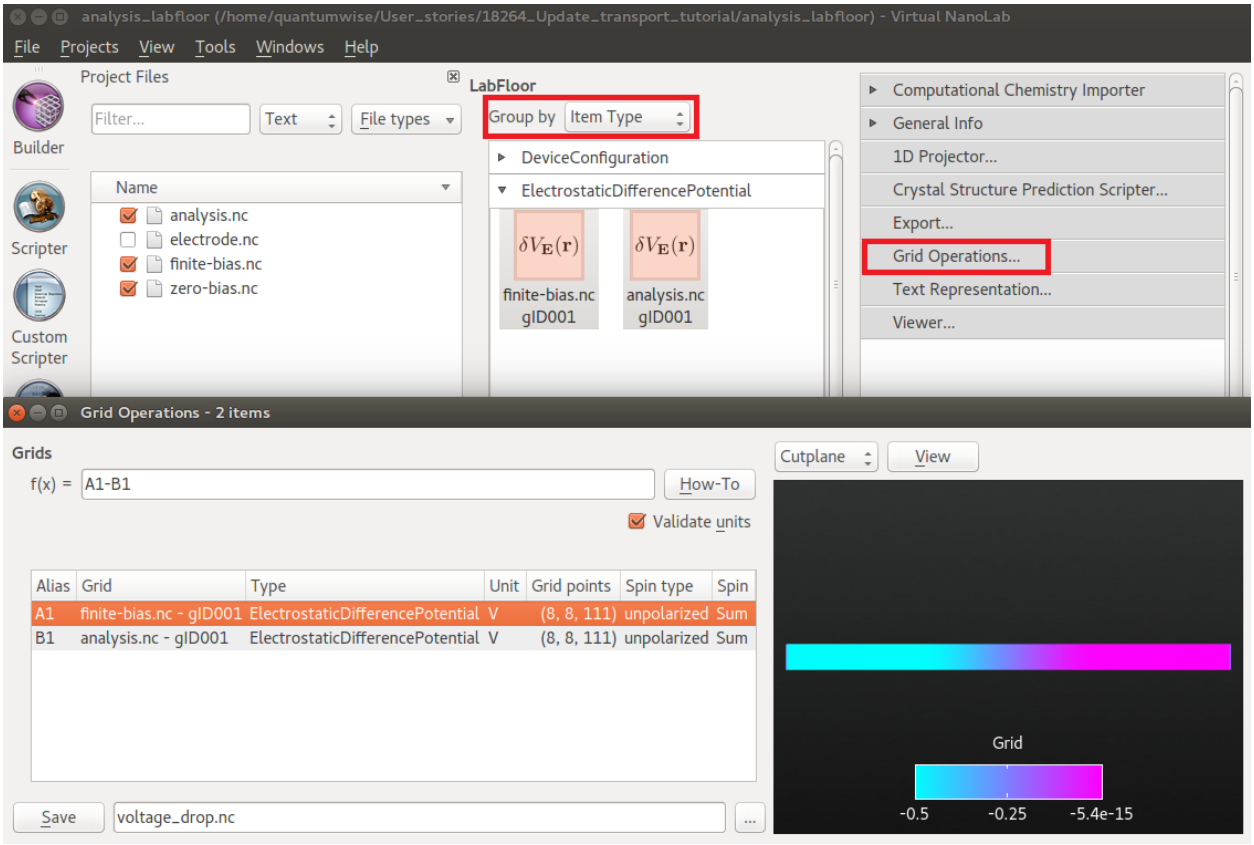
$$\Delta V_E(\mathbf{r}) = \delta V_E^{\mathrm{finite-bias}} - \delta V_E^{\mathrm{zero-bias}}.$$

Hint: Each electrostatic difference potential is the difference between the electrostatic potential of the self-consistent valence charge density and the electrostatic potential from a superposition of atomic valence densities. Since the atomic valence charge densities are the same for the zero and finite bias calculations, the calculated difference, $\Delta V_E(\mathbf{r})$, is the potential due to the differences between the two self-consistent charge densities, which is exactly the voltage drop.

The QuantumATK **Grid Operations** plugin is a very convenient tool. It allows you to perform arithmetic operations on 3D grids, such as addition and subtractions of one grid from another, and also multiplication and division. You will now use the plugin to compute $\Delta V_E(\mathbf{r})$, save it in a file, and then plot it in the **Viewer**:

•By default, the Labfloor groups all items by filename. Change the option **Group by** to *Item Type*.

•Then select the two ElectrostaticDifferencePotential items, and open the Grid Operations tool.



In the Grid Operations tool, the grids are assigned aliases. Depending on the order in which you selected the two items, the finite and zero bias potentials will be called A1 and B1, or opposite. To compute the voltage drop (the induced potential), subtract the zero-bias potential from the finite-bias potential by entering A1-B1 (or opposite) in the box next to *f(x)=* and press *Enter* on your keyboard.

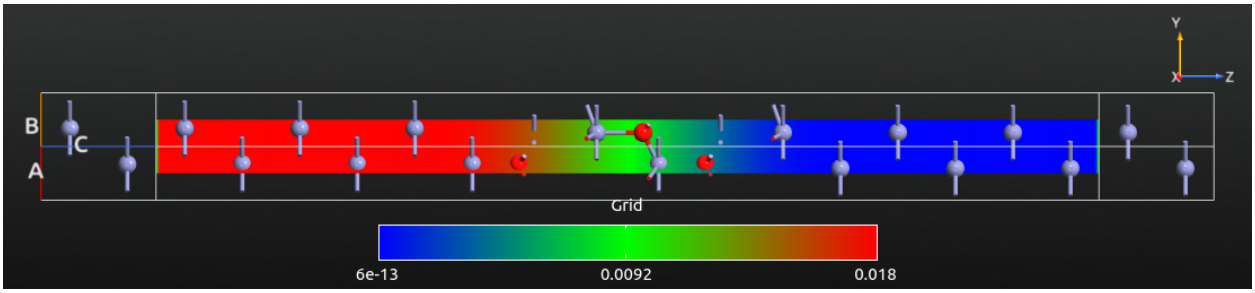
You can then immediately plot the result as a cut-plane or an iso-value contour in the 3D window to the right, and/or save the difference grid in a new HDF5 file by clicking the *Save* button in the lower left corner.

Save the difference grid as `voltage_drop.hdf5`, and observe the GridValues object appear on the LabFloor.

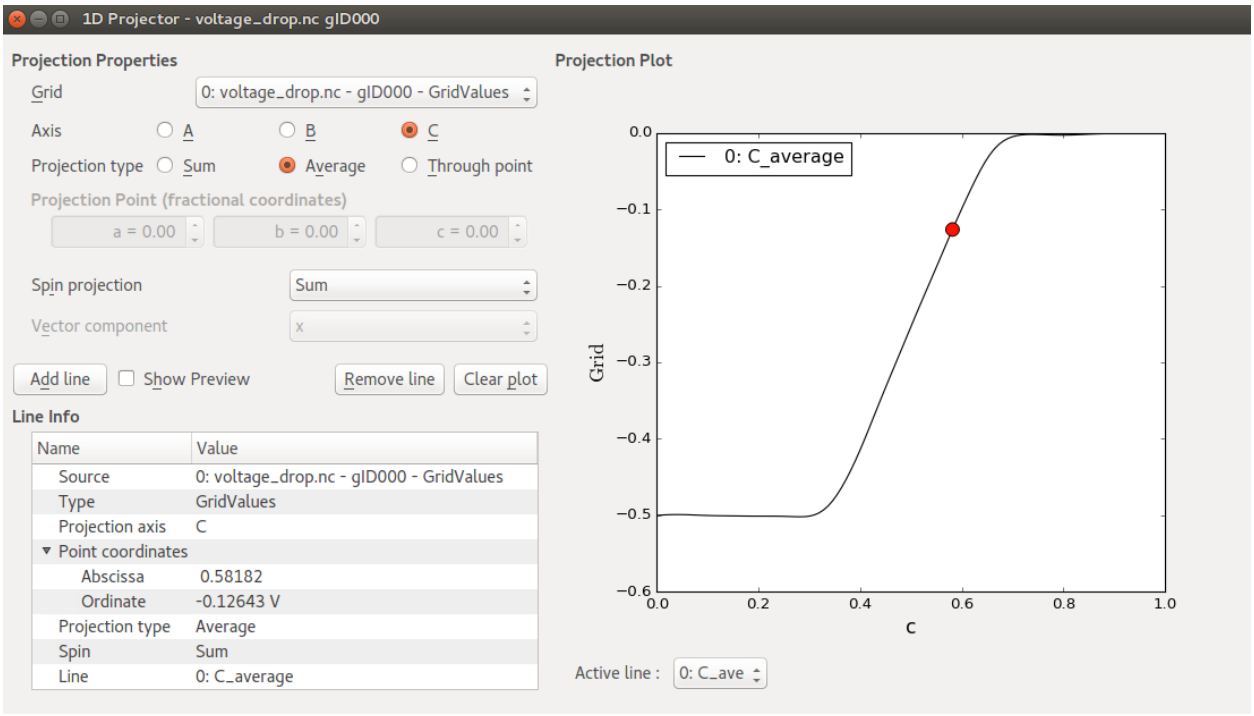
Note: $f(\mathbf{r})$ **GridValues** is a general container for quantities represented on 3D grids.

Select one of the DeviceConfigurations and open it with the **Viewer**. Then drag and drop the GridValues object onto the Viewer canvas in order to visualize it on top of the device configuration. Choose the *Cut-plane* visualization type, and set the color map to *RGB*.

The visualization should look like shown below. The potential drop occurs in the center of the device region, around the ZnO layers, showing that this is where scattering of the electrons takes place.



For a more quantitative analysis, open the saved difference grid with the **1D Projector**, choose *Average* as projection type, and click *Add line*. You should now see a plot like the one shown below. The potential values are given in Volt. The right-to-left potential drop is -0.5 V, which is exactly the applied bias voltage.



Also note that the induced potential is flat towards the electrodes, indicating that the Zn layers in the left and right ends of the central region provide sufficient screening.

I–V curve

The last finite-bias device analysis in this tutorial is the I–V curve. You will first compute it using the **IV Characteristics** analysis in the **Study Objects**, and then visualize it using the **IV-Characteristics Analyzer** plugin.

Attention: The *Study Objects → IV Characteristics* analysis has been implemented since the O-2018.06 version. If you use an older version, you can use the *Analysis → IVCurve* object instead.

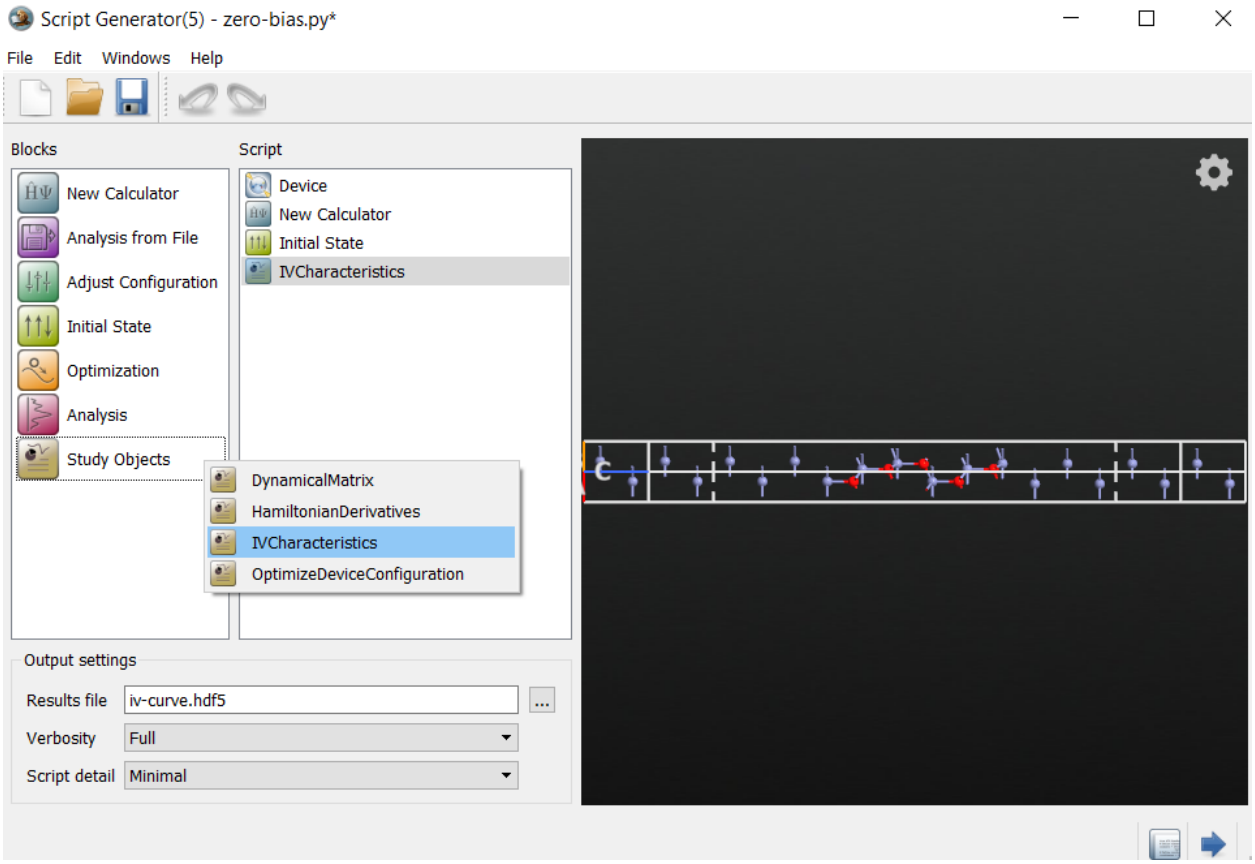
Calculations

Drag and drop the `zero-bias.hdf5` or `zero-bias.hdf5` file onto the **Scripter**. It automatically keeps the settings of the zero-bias calculation. Now do the following:

•Add the Initial State block.

•Add the *Study Objects* → *IVCharacteristics* block.

•Change the default output filename to `iv-curve.hdf5`.

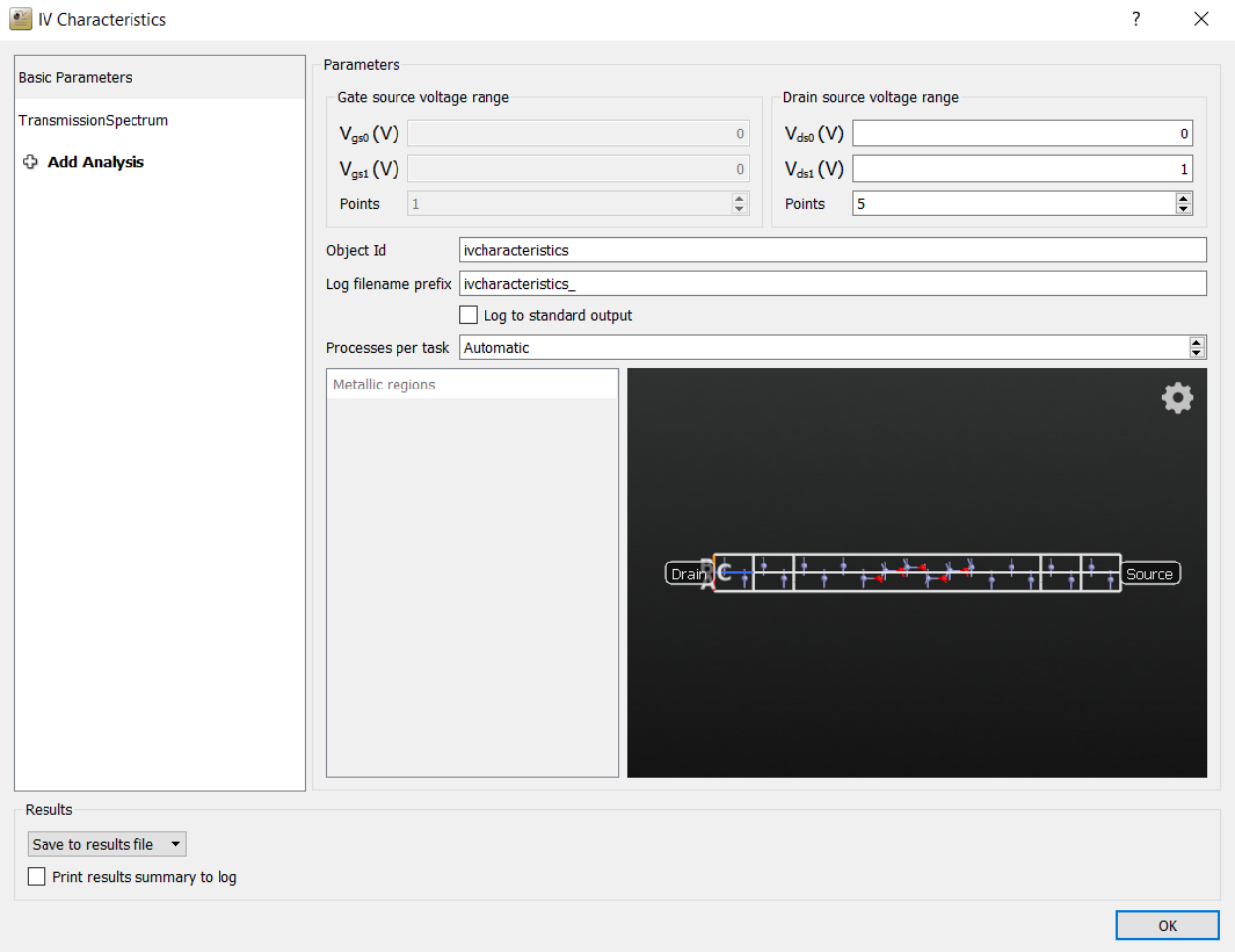


Next, open the Initial State block to instruct QuantumATK to use the zero-bias state as an initial guess for the finite-bias calculation:

•Tick **Use old calculation**, and choose `zero-bias.hdf5` or `zero-bias.hdf5` as the filename of the *Old Calculation*.

Edit the IVCharacteristics block:

•Decrease the number of voltage points to 5 (for speed).



The QuantumATK Python script is now done. Save it as `iv-curve.py` and execute it using the **Job Manager** or from the command line. This calculation will take some time, because a self-consistent calculation and a transmission spectrum analysis must be done at each bias point, so probably around 10 minutes in 16 cpus.

If needed, you can download the script here: `iv-curve.py`. If you do not want to perform the calculation yourself, you can also just download the results: `iv-curve.hdf5`.

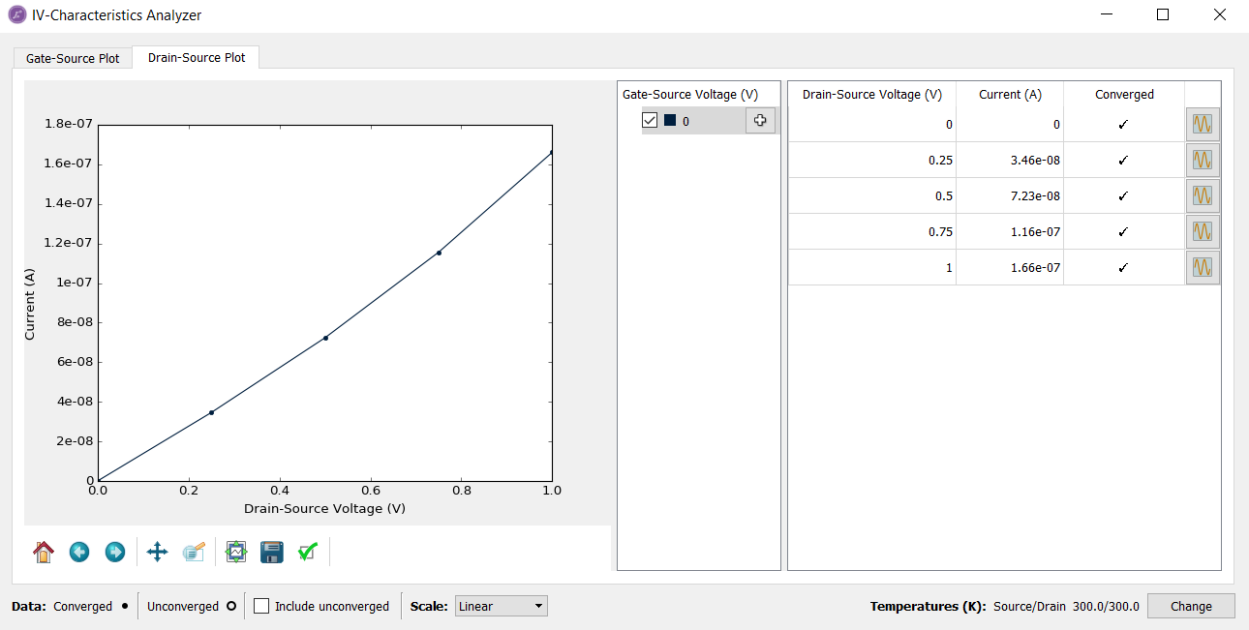
Visualization

The calculation should now be done, and the $I(V_g, V)$ *IVCharacteristics* data item should now be included in the `iv-curve.hdf5` output file. Five new output files were also created, prefixed `ivcharacteristics`; they contain the log file at each bias point.

Note: The $I(V_g, V)$ *IVCharacteristics* item contains the calculated I–V curve, including the transmission spectra computed at each bias voltage.

Select the *IVCharacteristics* item on the LabFloor and click the **IV-Characteristics Analyzer** tool in the $I\,(V_g, V)$ right-hand plugins bar. The I–V curve is plotted in the window that pops up.

Select the *Drain-Source Plot* tab and choose the scale of current by right-clicking and choosing *Customize*. Change the maximum for the y-axis to *1.8e-07* to get the plot shown below.

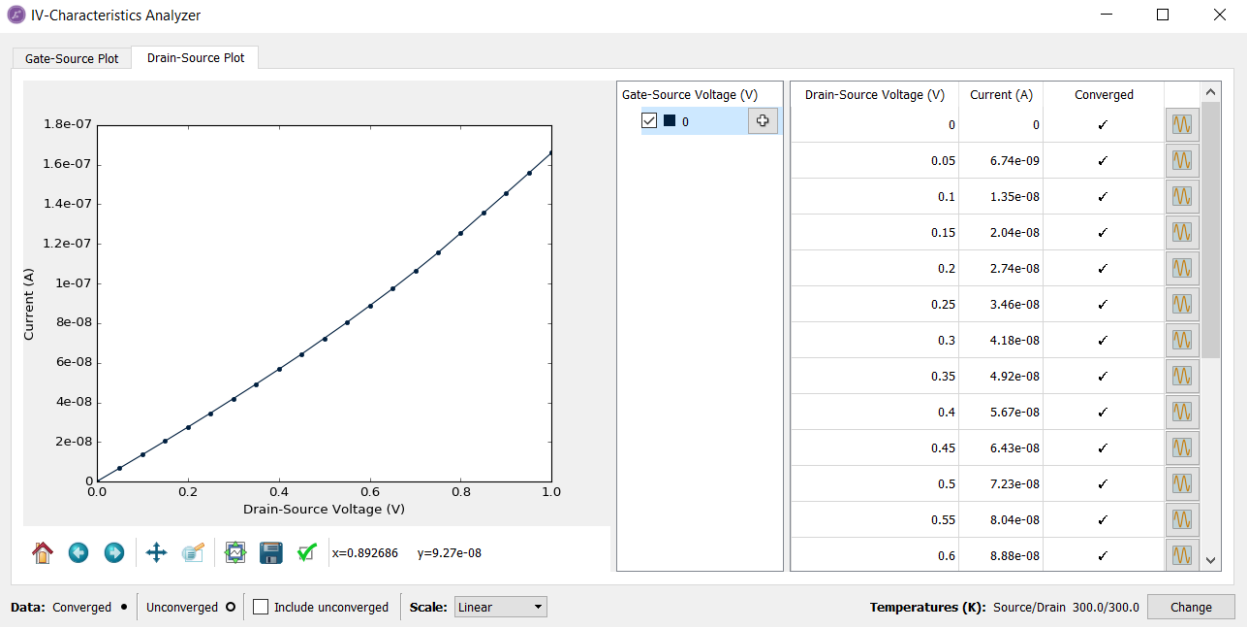


The table in the **IV-Characteristics Analyzer** shows the current and status of convergence at each finite voltage. Once you click the right-hand side plot icon ( ) in the table for the first time, the Analyzer will load in the Transmission spectra from the file. You can the visualize the transmission spectrum corresponding to a specific bias voltage point by clicking the icon and choosing *Transmission spectrum*.

A somewhat finer bias-point sampling would yield a more accurate current. To this end, you should increase the number of bias points to 21, for example, and re-run the script with **IVCharacteristics** study object. It will automatically re-calculate the new points. You can also download the result here: `iv-curve_fine.hdf5`.

Note: The **IVCharacteristics** study object enables addition of more bias points. Simply add more voltage points in the **IVCharacteristics** object with the same calculator settings, then run it in the existing output file directory. The **IVCharacteristics** recognizes the existing bias points smartly, and just runs the additional voltage points. It will be faster and not waste the computational cost. See also the dedicated tutorial: *Electrical characteristics of devices using the IVCharacteristics study object* for more information.

The finely sampled I–V curve is illustrated below. Compare it to the coarsely sampled one shown above even though the IV-curves look very similar.



Summary

You have now been introduced electronic transport calculations with QuantumATK.

- You have gained an understanding of the general structure of a device geometry, and can continue with setting up other device geometries.
- You have learned about important convergence parameters, such as transverse k-points, density mesh cut-off energy, and NEGF contour integral settings.
- You are now able to calculate the transmission spectrum at zero and finite bias voltage, and analyze the results using the Transmission Analyzer.
- You have also been introduced to other analysis tools like projected device density of states, electrostatic difference potential, transmission eigenvalues and eigenstates, and I–V curve using the IV characteristics study object.
- You are also able to visualize 3-dimensional objects with the 3D Viewer or by using 1D projections.

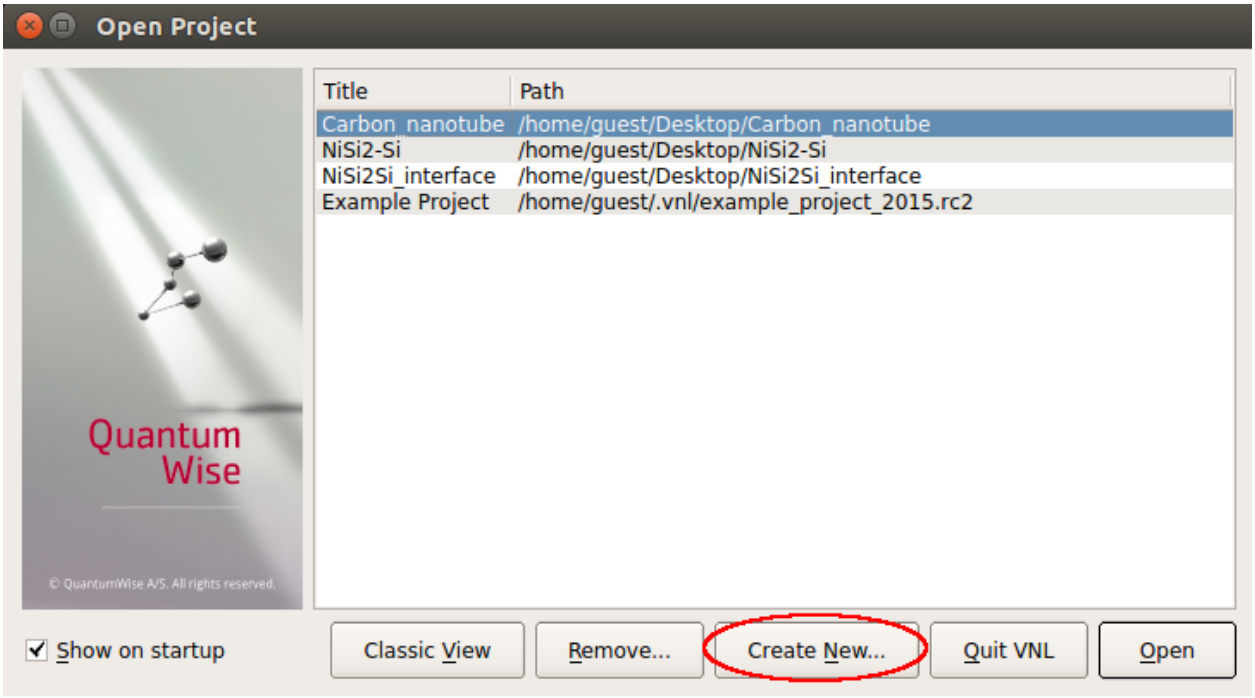
Tip: Whats next? Now that you know the basics of *electron* transport calculations, it would be wise to continue to the tutorial *Phonons, Bandstructure and Thermoelectrics* to learn about simulating *phonon* transport with QuantumATK.

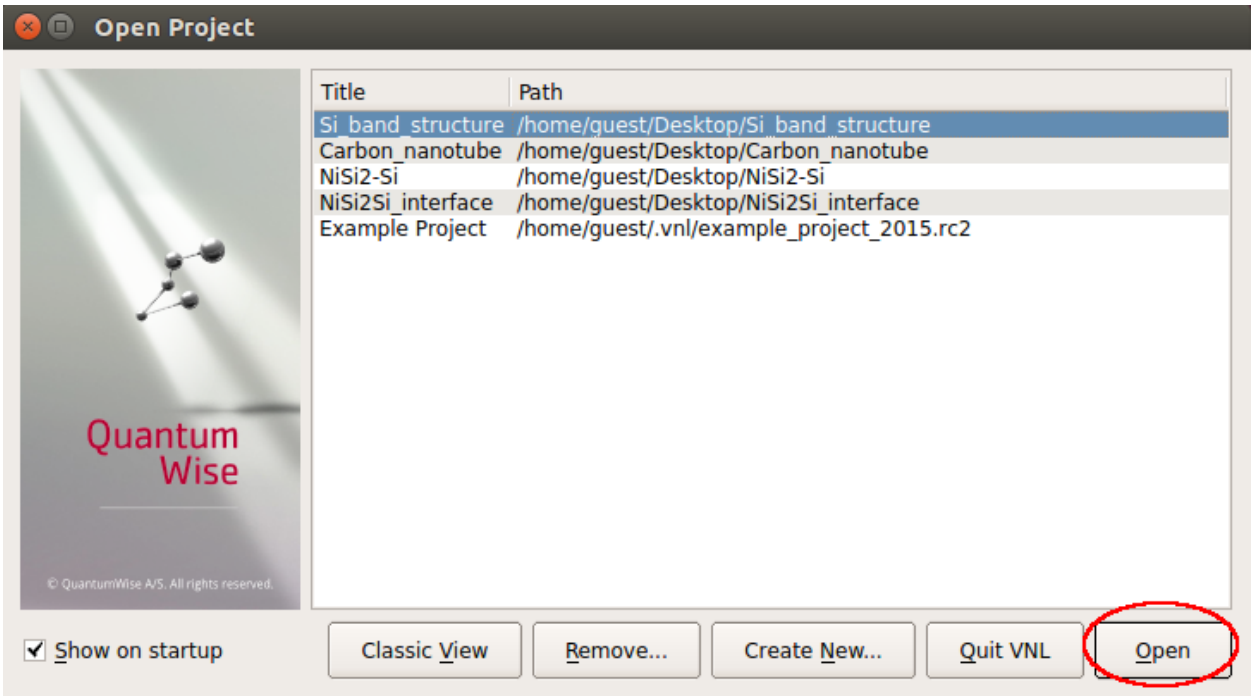
3.12.13 Calculate the band structure of a crystal

In this tutorial, you will learn how to calculate the band structure of silicon by using the extended Hückel method.

Start QuantumATK and create a new project

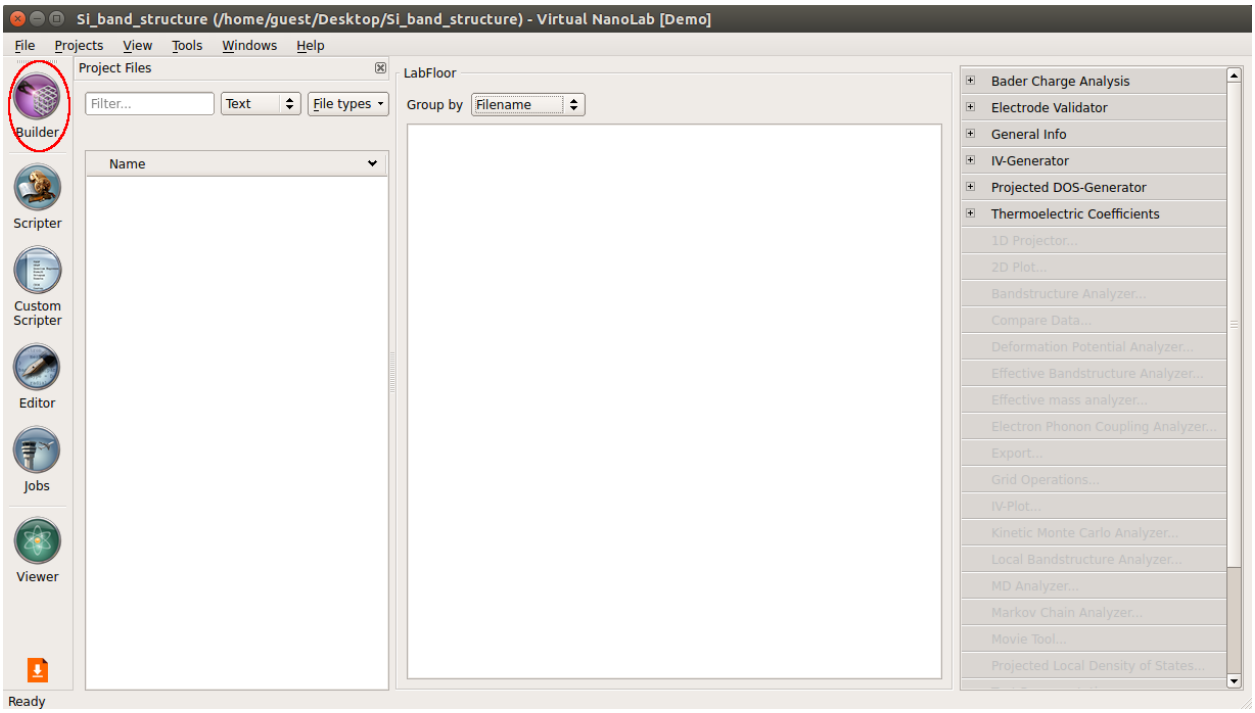
Open **QuantumATK** and create a new project by clicking **Create New**. Give the project a *Title* (here: Silicon_band_structure), select the folder where data will be stored, then click **OK** and **Open** to start the project.



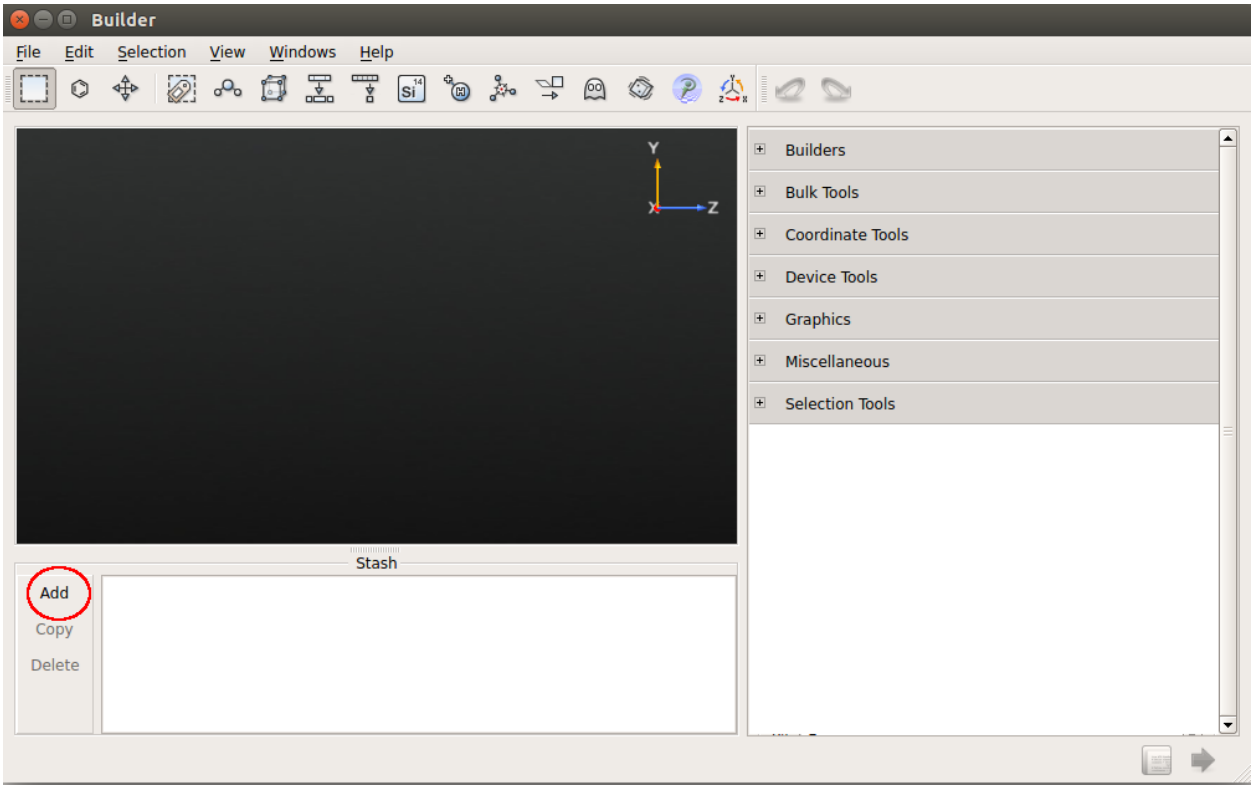


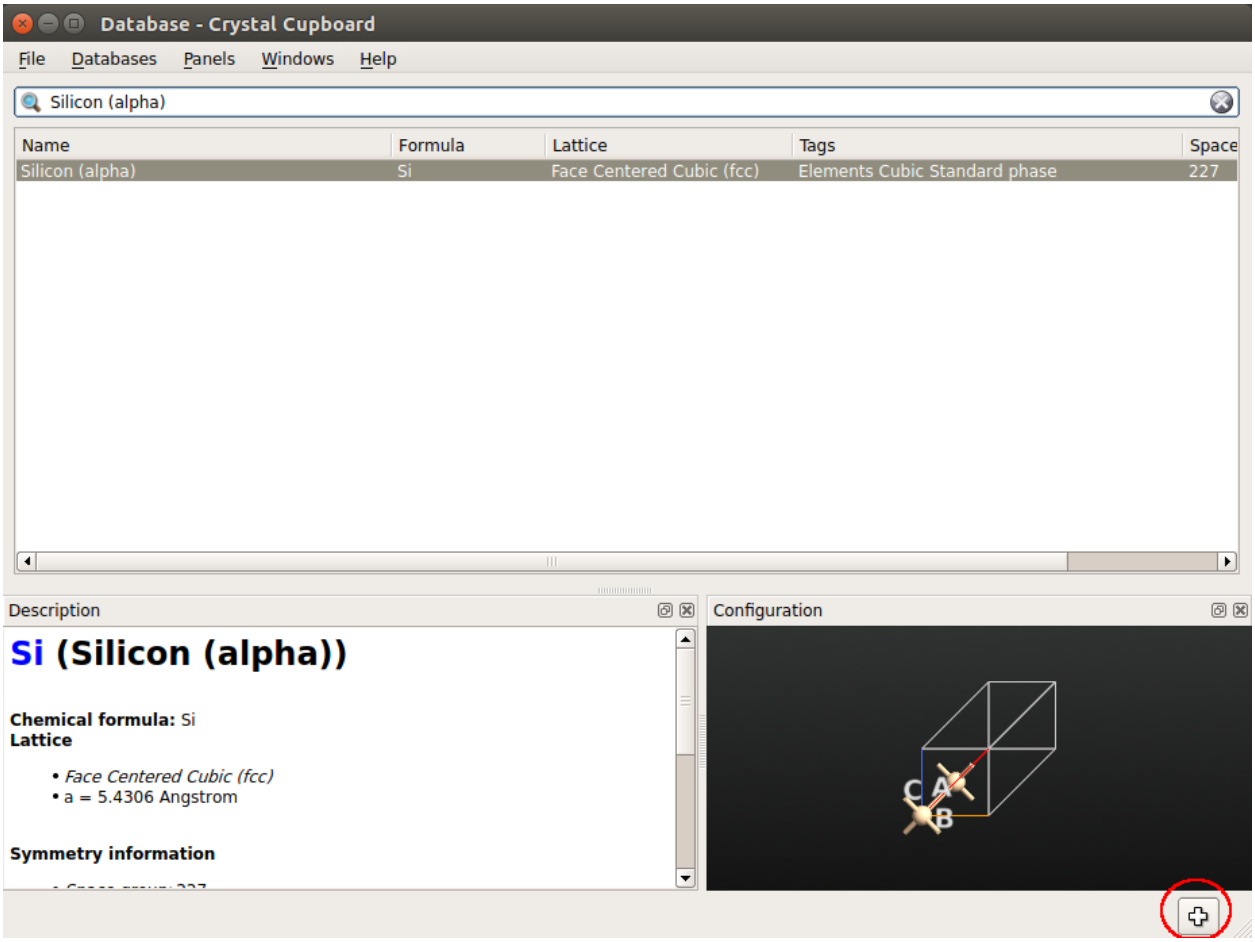
Import the Silicon structure from the Database and send it to the Scripter

1.In the **QuantumATK** main window, click on the icon to open the **Builder**.

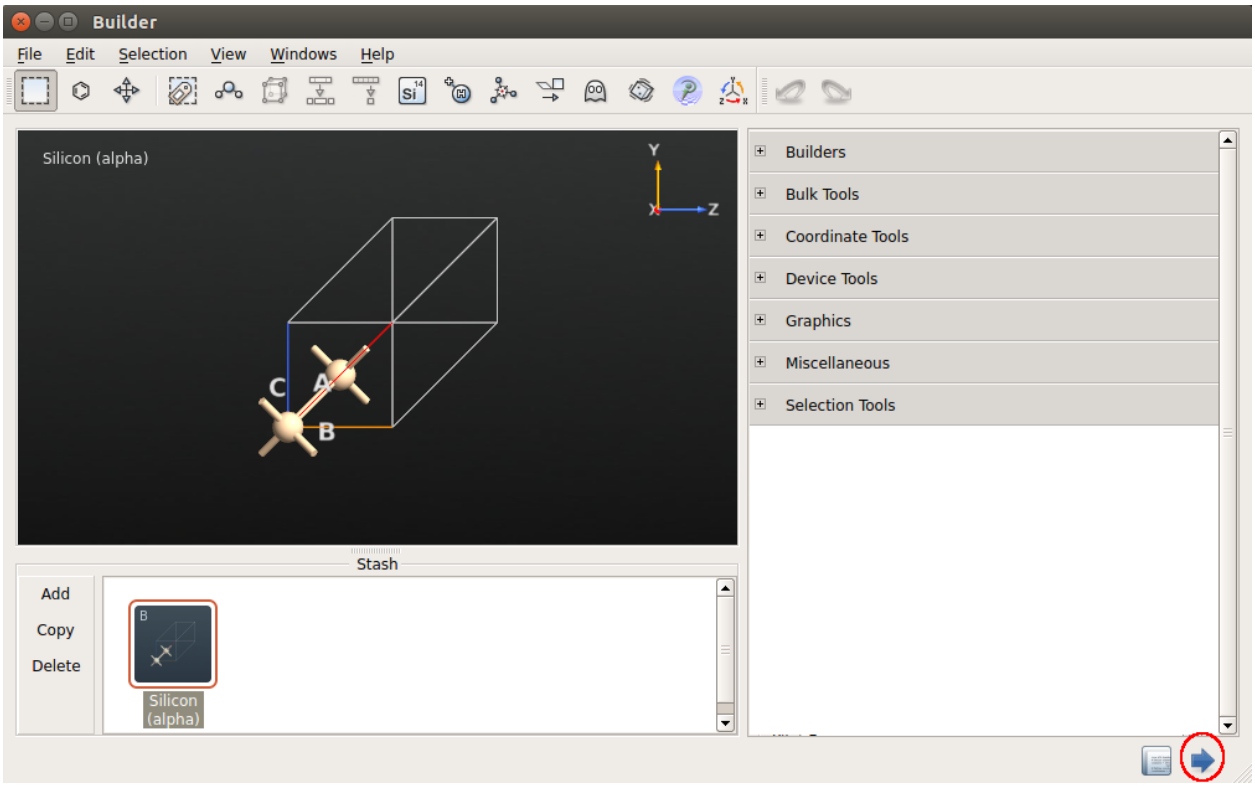


2.Go to the **Stash**, click *Add* → *From Database*, and choose *Databases* → *Crystals* from the menu. Type *Silicon* in the filter field at the top and select *Silicon (alpha)* in the list. Click the icon, or double-click its line in the list, to send it to the **Stash**.


Builder
File
Edit
Selection
View
Windows
Help
Y
X
Z
Builders
Bulk Tools
Coordinate Tools
Device Tools
Graphics
Miscellaneous
Selection Tools
Stash
Add
Copy
Delete



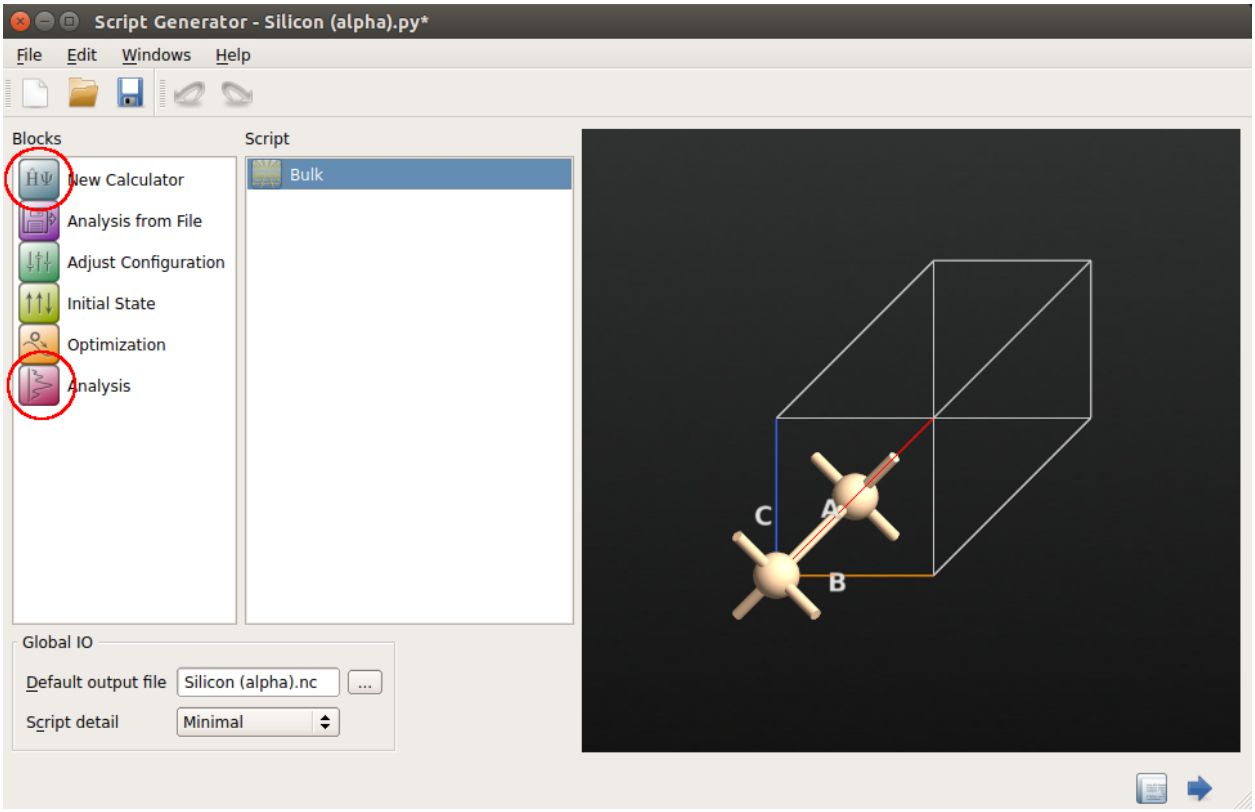
3. Send the bulk structure to the **Scripter**. In order to do that, click the button and choose **Script Generator** from the menu that appears.



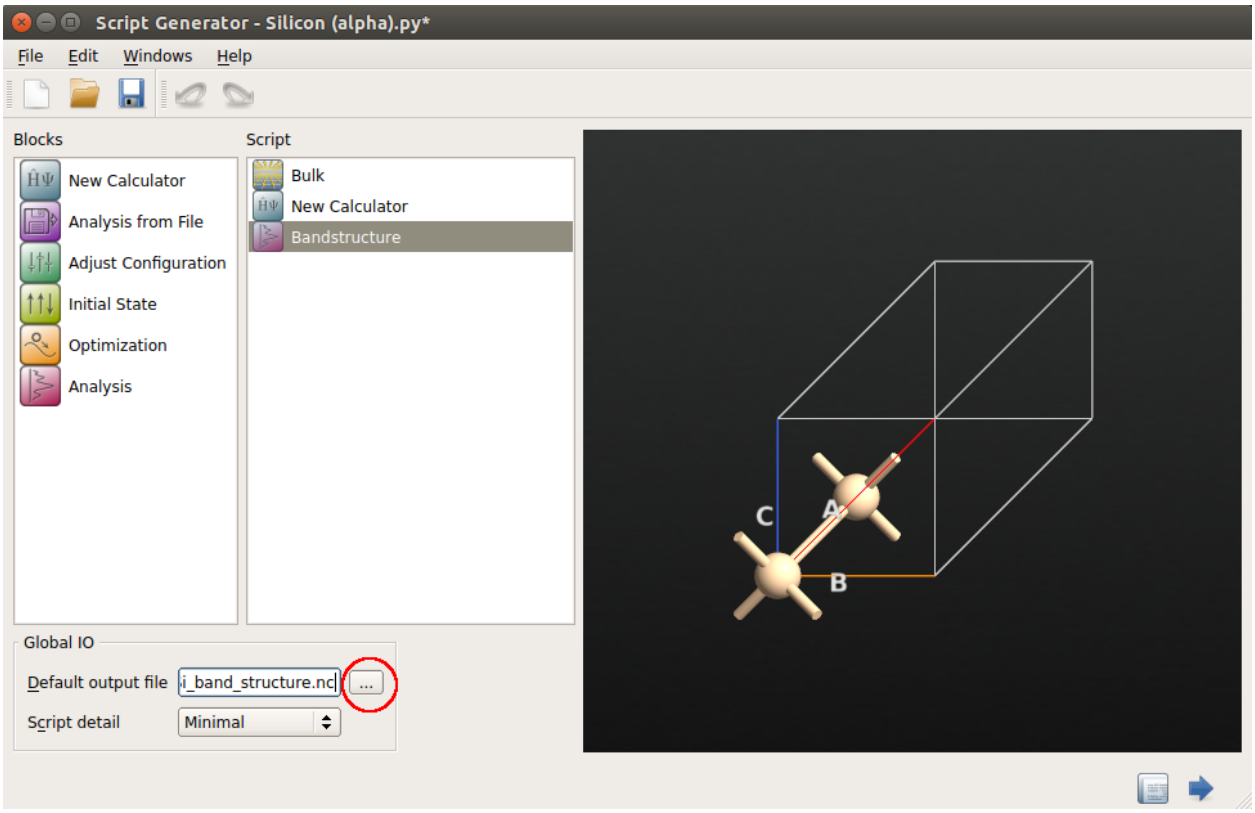
Set up the calculation and analyse the band structure

1.In the Script Generator, use double-clicks to add a New Calculator block.

2.Also add one *Analysis* → *Bandstructure* block.

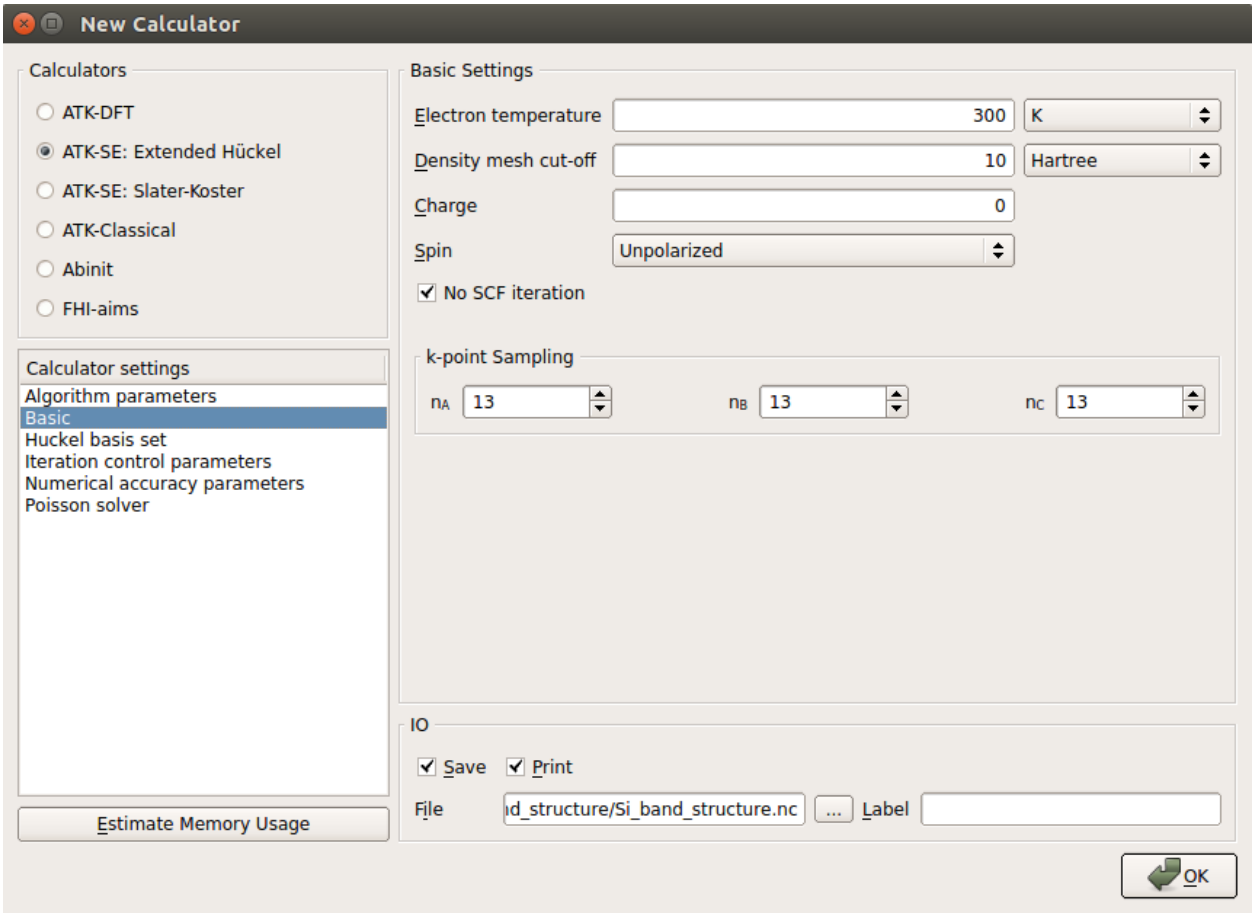


3.Change the default name of the HDF5 file in the **Global IO** options: Type in `Si_band_structure.hdf5`.

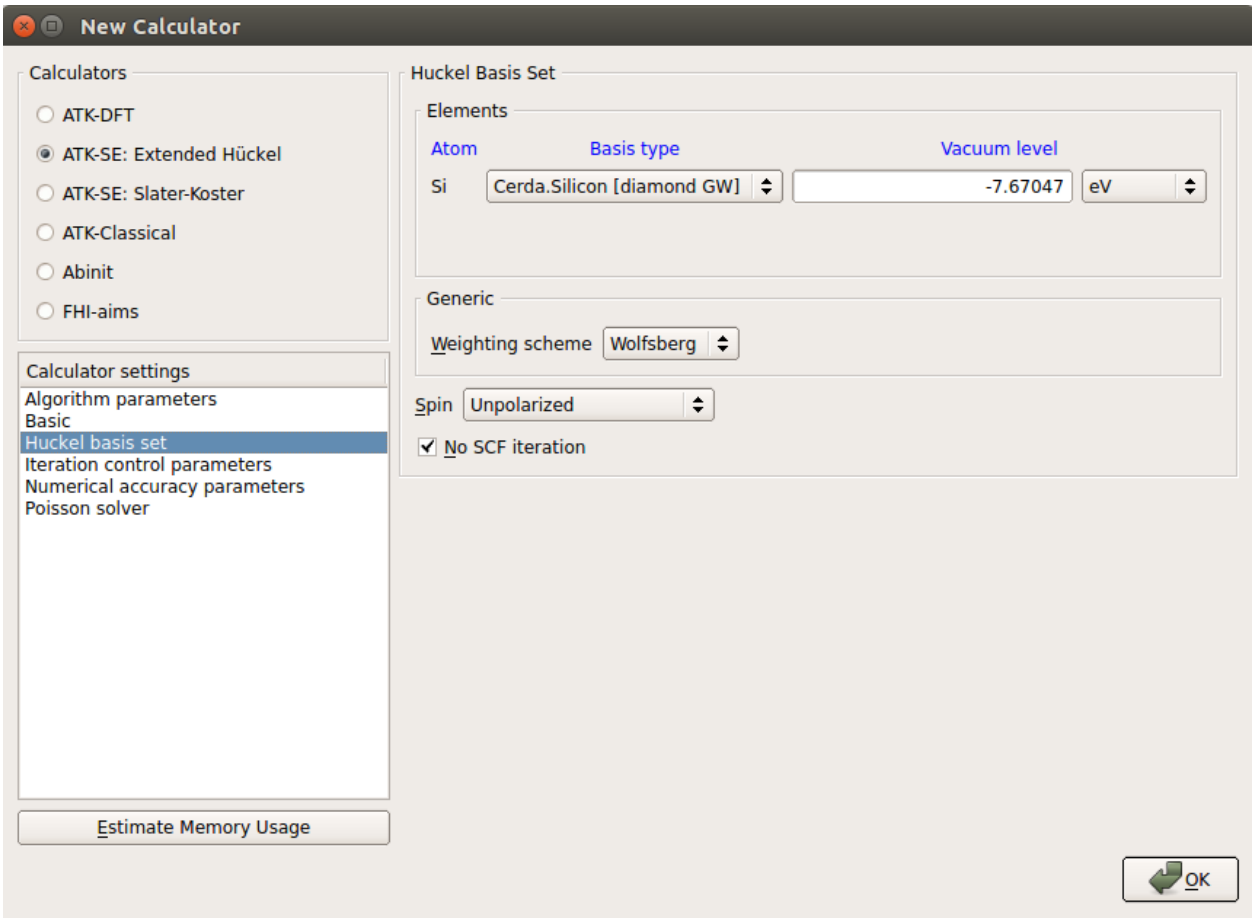


4.Double-click the New Calculator block to define the parameters for the quantum-chemical method:

1.Set the calculator to Extended Hückel (or leave it at DFT if you so prefer), and specify a k-point sampling of $n_A = n_B = n_C = 13$.

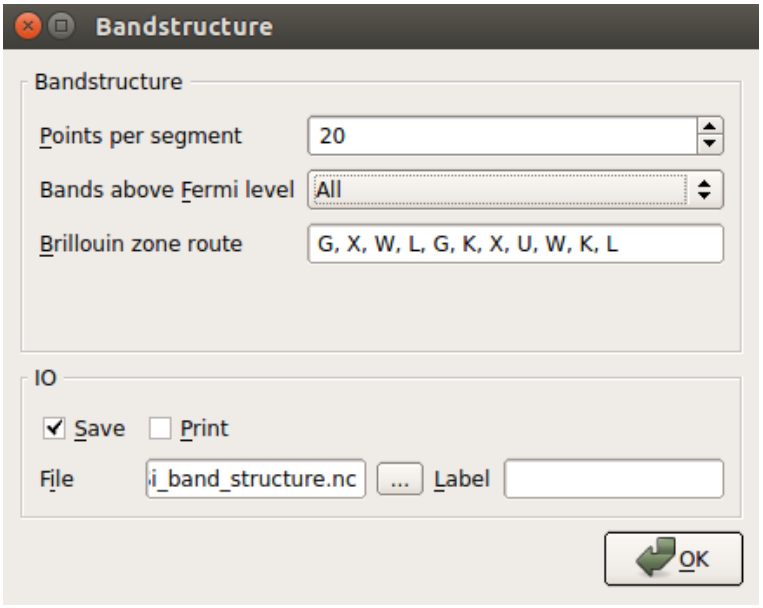


2.If you use the Extended Hückel, you must also change the basis set to *Cerda.Silicon* to get a proper description of the band gap.



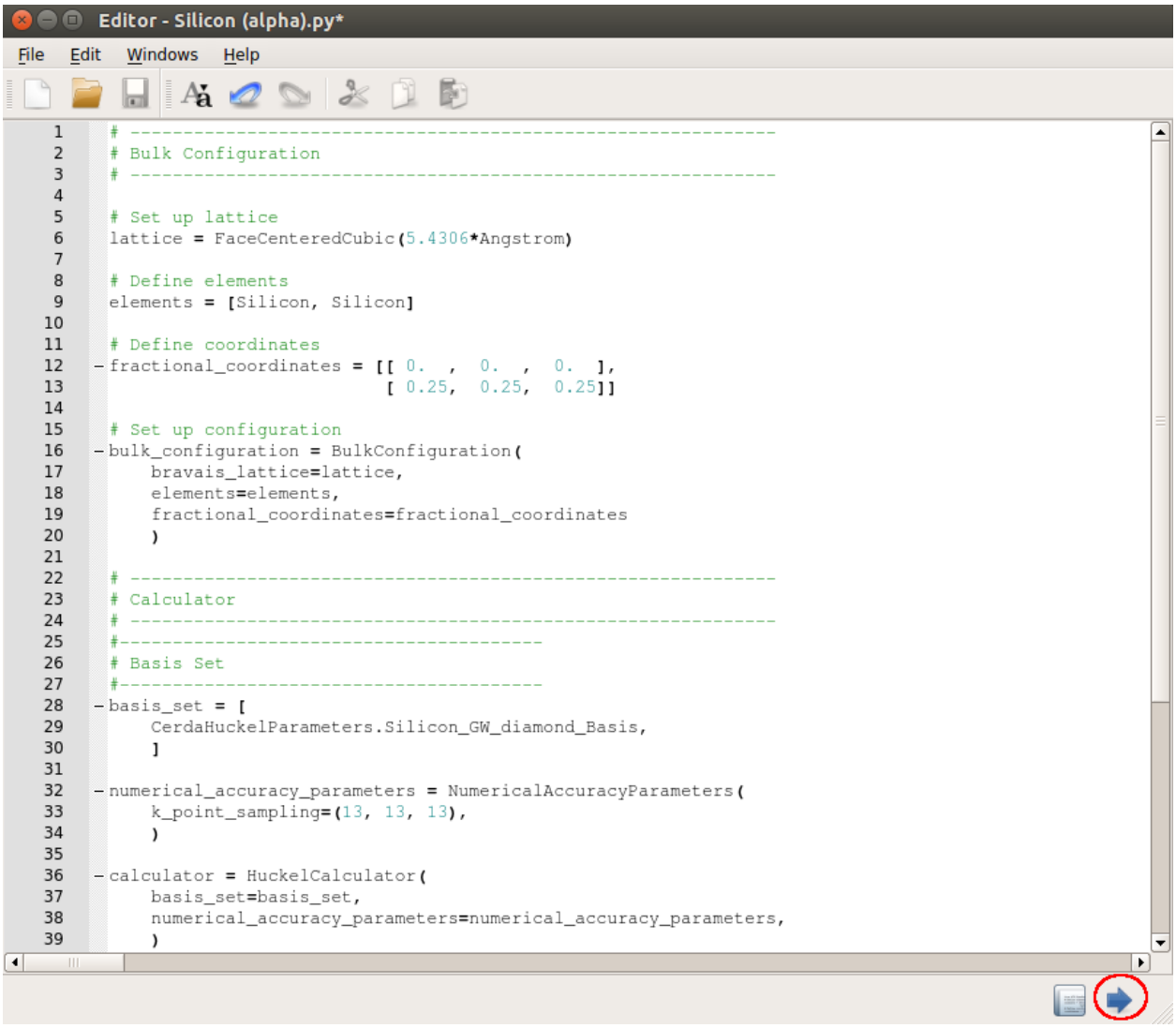
3.Click **OK** to save the changes.

5.Now open the **Bandstructure** block. We will not change anything, but you can see how the path in the face-centered cubic Brillouin zone between the symmetry points G, X, W, L, G, K, X, U, W, K, L, is set up. Click **OK**.

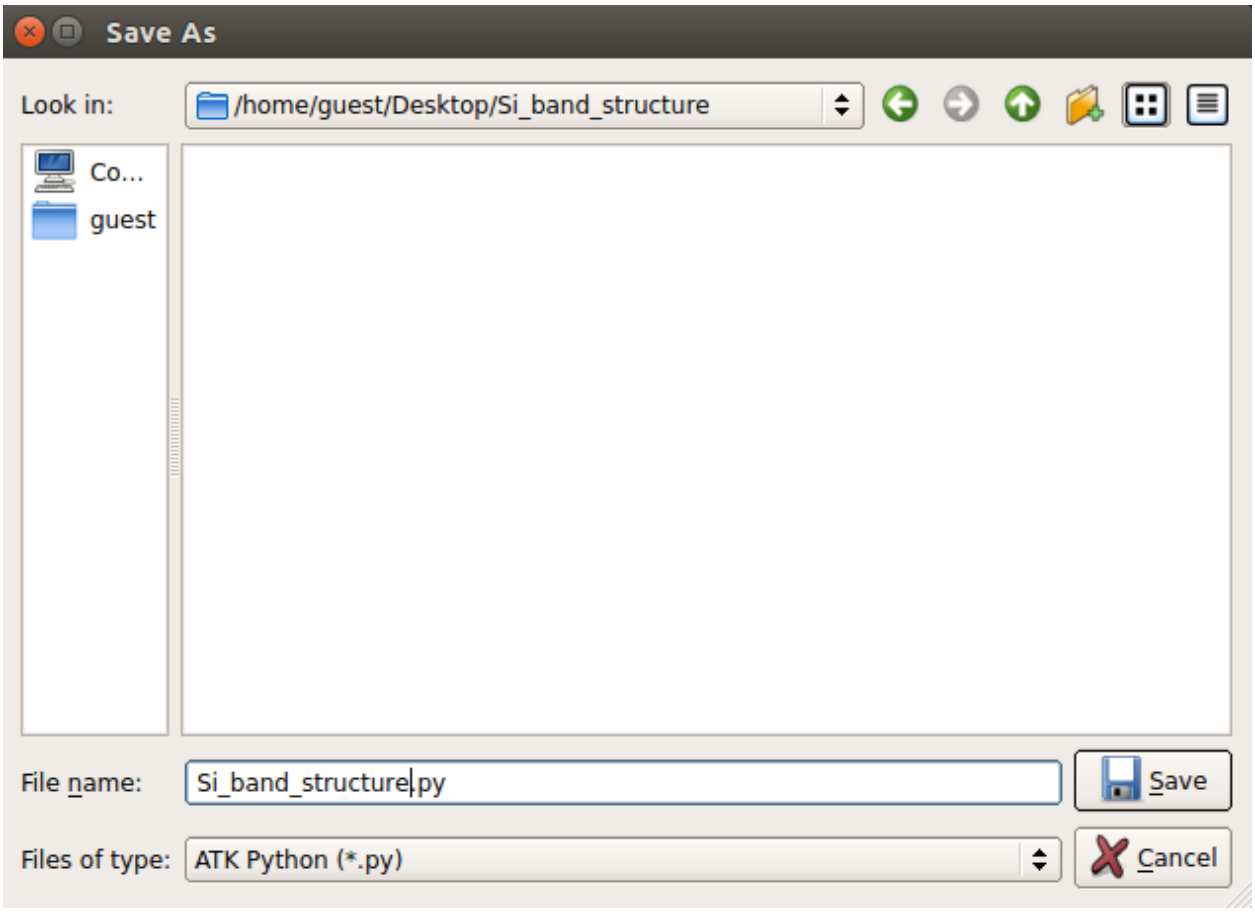


6.If you are interested in inspecting the actual Python script, you can now send the scipt to the **Editor**: Use the button or simply drag and drop it onto . Note that this will minimize the Script Generator window.

7.In order to run the script, you simply send it to the **Job Manager**, again using the button.

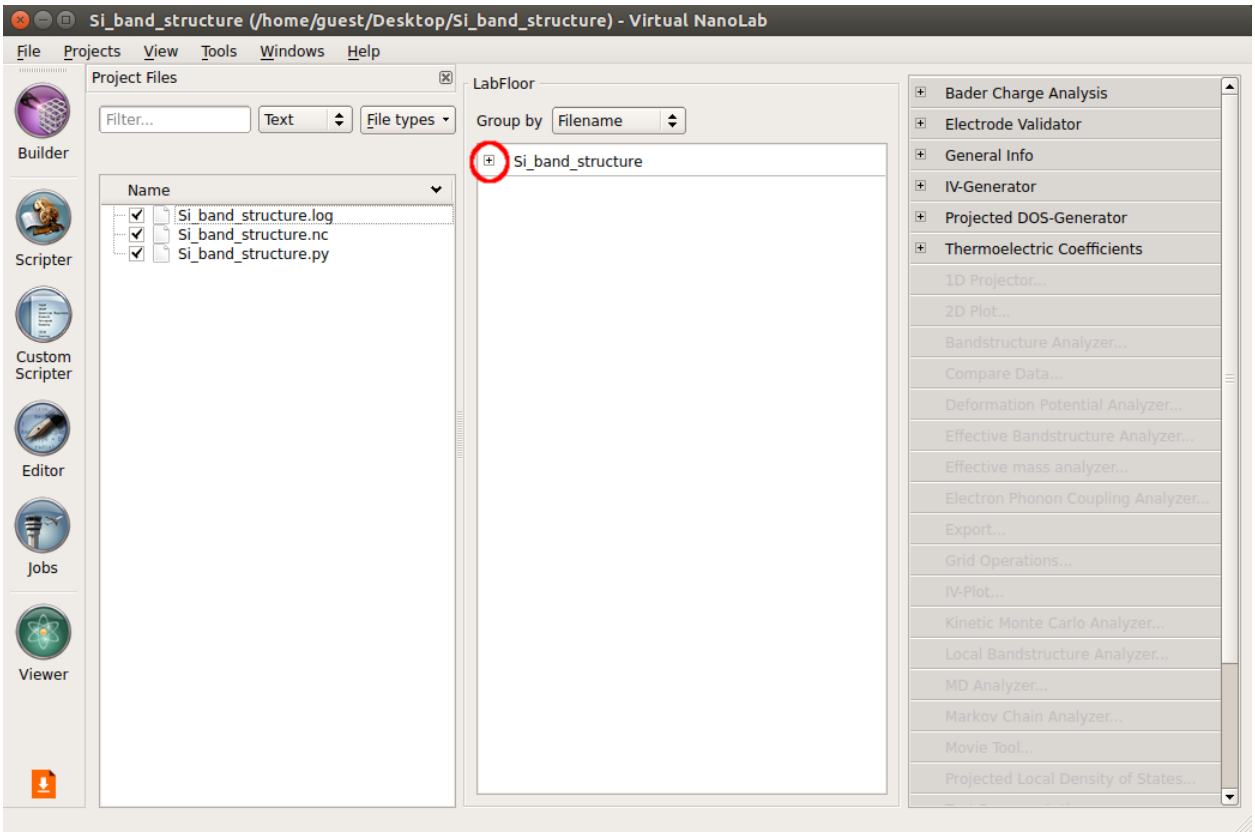


8.Save the script in the window that appears.

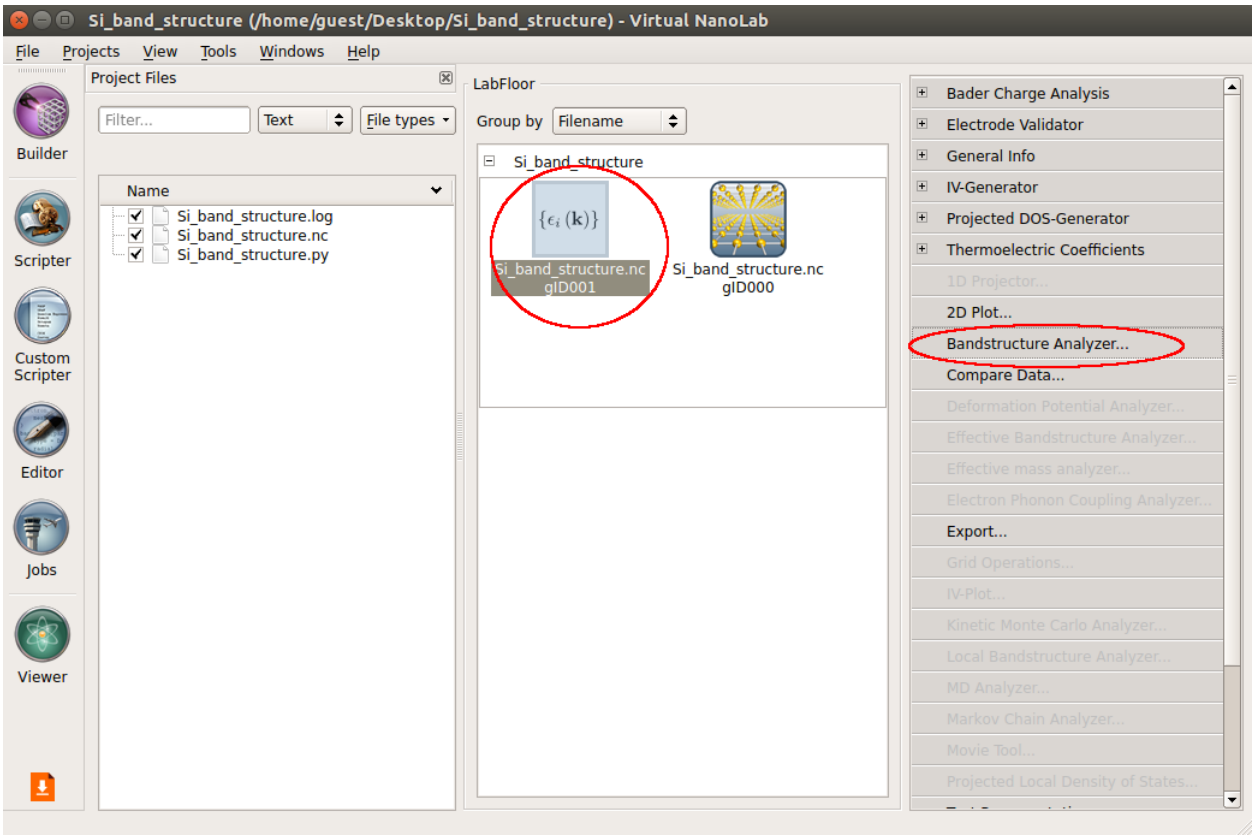


9. In the Job Manager, click ▶ to run the script.

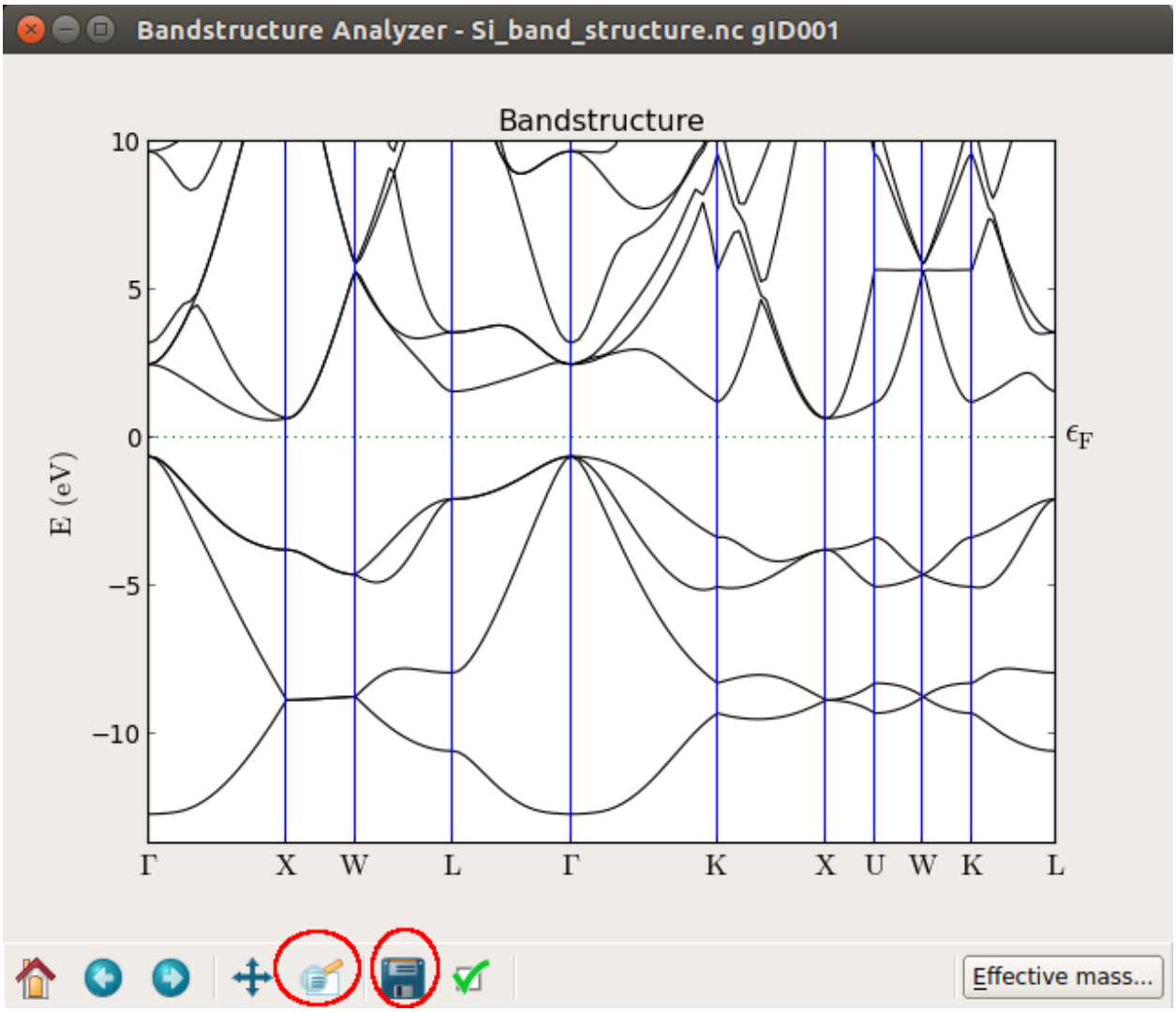
10. When the script has finished (it should only take a few seconds), go back to the main QuantumATK window, and click the tab (⊞) next to your HDF5 file, `Si_band_structure.hdf5`, to see the objects contained in it.



11.Select the Bandstructure object and plot it with the **Bandstructure Analyzer** in the right-hand plugins panel.



12. In the band structure plot that opens, you can zoom in using the **Zoom to rectangle** tool. You can also save the figure by clicking the button.



3.12.14 Introducing the QuantumATK plane-wave DFT calculator

Version: 2017.0

Downloads & Links

`e-cut-conv.py` `restart-e-cut-conv.py` `plot-e-cut-conv.py` `cu-bulk.py`

The ATK-DFT plane-wave calculator was introduced in QuantumATK 2017.0, and this tutorial gives a brief introduction to how to use it. We will look at a simple bulk example to introduce the basic functionalities of this calculator.

Introduction

The central difference between the plane-wave (PW) calculator and the LCAO calculator is that the one-electron eigenstates are expanded in a plane-wave basis set instead of atom-centered basis functions. The plane-wave representation of electronic structure has a long tradition in computational studies of periodic bulk systems, and has the key advantage that the basis set can **systematically be improved** by simply increasing the number of plane waves included in the calculation (albeit at a significantly increased computational cost). This makes the PW calculator an interesting alternative to the usual LCAO calculator for studies of (not too large) bulk systems, and may also be used as a benchmark in investigations of the accuracy of an LCAO basis set for a specific system.

As an example, we will find the lattice constant and band structure of bulk copper, but first we need to converge the size of the PW basis to make sure our results are accurate.

Warning: Note that in the 2017 version of QuantumATK, the plane-wave calculator is still a beta version, so not all functionality has been implemented, and it has a higher risk of bugs. From QuantumATK O-2018.06 it is a finished product with full functionality.

Converging the PW basis

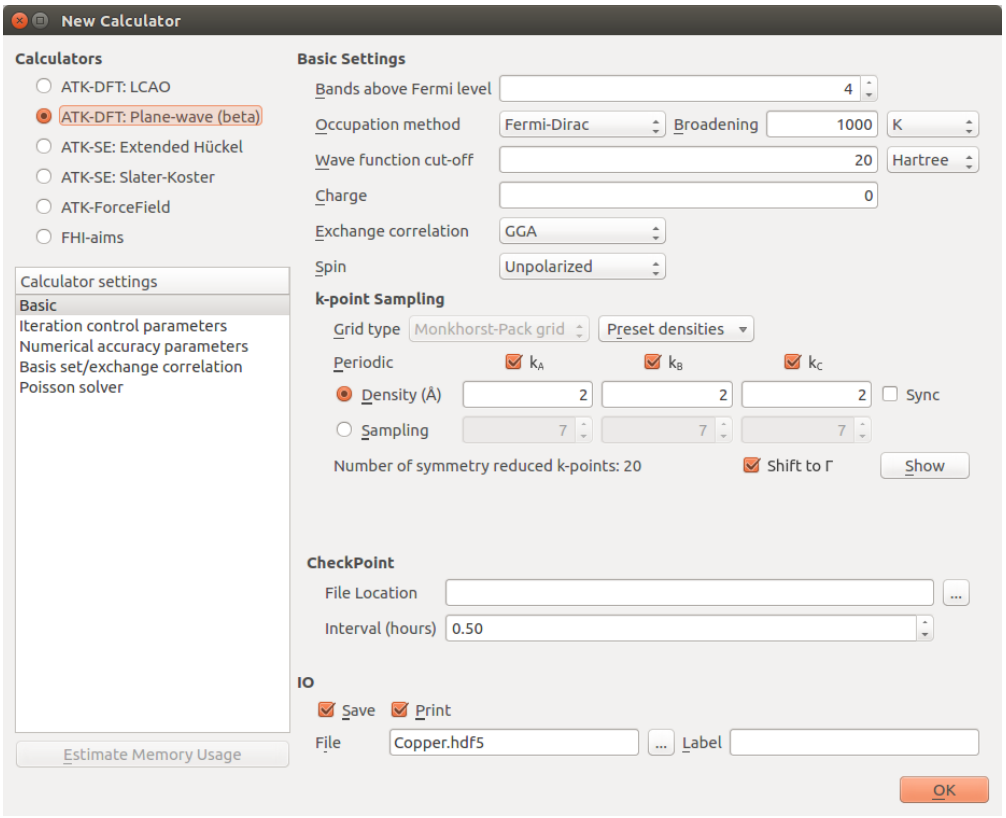
The number of plane waves included in the PW calculation depends on the plane-wave cutoff energy E_{cut}. The accuracy of the PW calculation increases as the number of plane waves increase (larger E_{cut}), but so does also the computational cost. The desired trade-off between computational accuracy and cost must therefore be chosen, and we use a convergence study for this.

To converge the PW basis, we compute some desired property for a range of cut-off energies, here the total energy, in order to investigate how large E_{cut} needs to be in order to get sufficiently accurate results.

To do this, follow these steps:

- Open the **Builder** and click on *Add* → *From database*.
- Type copper to search for pure copper, select it and press the button.
- Now send the configuration to the **Script generator**
- Add **New Calculator** and **Total energy** objects.
- Open the **New Calculator** and change it to *ATK-DFT: Plane-wave (beta)*.

Your **New Calculator** window should look like this:



•Click *Ok* and send the script to the **Editor** by using the .

We now need to create the loop over the PW cutoffs. Insert these lines at the top of the script, after the line specifying the encoding, to calculate the total energy for cutoffs ranging from 10 to 100 Hartree:

```
e_cut_list = range(20,110,10)*Hartree

for e_cut in e_cut_list:
```

You also need to indent everything after these lines.

Tip: You can easily indent many lines in the Editor by selecting them, and pressing the `Tab` key.

Finally, we need to change the values for the cutoffs to the variable that is being used in the `for`-loop. Find the line where the density mesh cutoff is defined, it should be line 37 now, and replace it with the following:

```
        density_mesh_cutoff=e_cut*4,
```

Find the line where the calculator is defined, and insert `wave_function_cutoff=e_cut,` to make the calculator look like this:

```
    calculator = PlaneWaveCalculator(
        wave_function_cutoff=e_cut,
        numerical_accuracy_parameters=numerical_accuracy_parameters,
        )
```

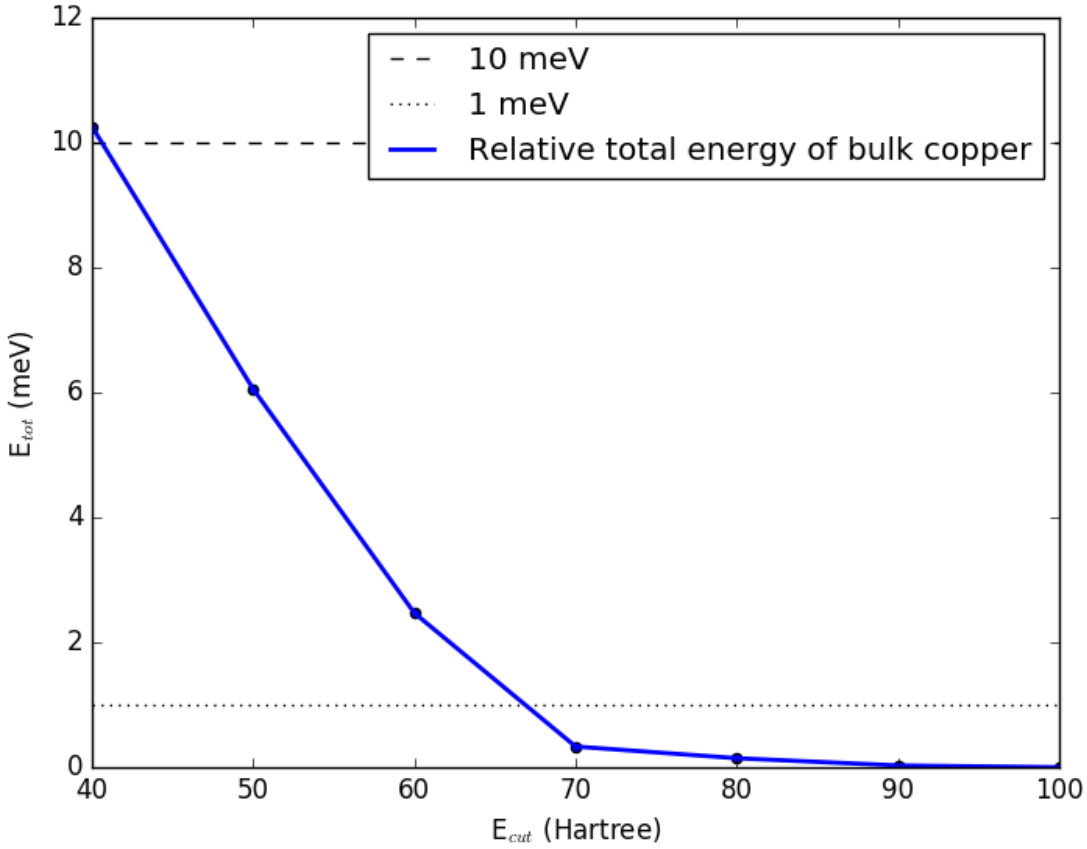
Note: The default mesh cutoff is 4 times the PW cutoff, which ensures that there is no loss of accuracy from this and no unnecessary computational overhead.

Your script should now look like this: `e-cut-conv.py`, and you can run it. It should take no more than half an hour on a modern laptop, and significantly less if MPI parallelization is used.

Tip: We plan to introduce a new feature in 2018 which allows you to set up this process from the Script Generator.

In some cases where you want to redo a previous calculation, but with a higher PW cutoff, it can be beneficial to restart the calculation. It is slightly more complicated to set up the script this way, but in our case it will make the calculation run a little faster, reducing the time required by about a factor of 2. An example of this script can be seen here: `restart-e-cut-conv.py`.

After running the script, we can now plot the total energy of the system vs. the PW cutoff. You can use this simple script to plot the data: `plot-e-cut-conv.py`. It will generate two plots. One for all values of the cutoff, and one where the first values are omitted, to allow for a clearer picture in the region of convergence. The second plot should look like this:



It shows that the total energy has converged to within 10 meV per atom with a cutoff of 50 Hartree, while 70 Hartree is required to converge the energy within 1 meV of the chosen reference calculated with a cutoff of 100 Hartree. The choice of reference and convergence criteria is a decision for the user, and should be based on the needed accuracy, balanced against the computational cost. In this case, we are mostly interested in fast calculations, and so we choose a

cutoff of 50 Hartree.

Tip: The required cutoff depends strongly on both the element and the specific pseudopotential, so you should always test convergence when using a new pseudopotential.

Calculating properties

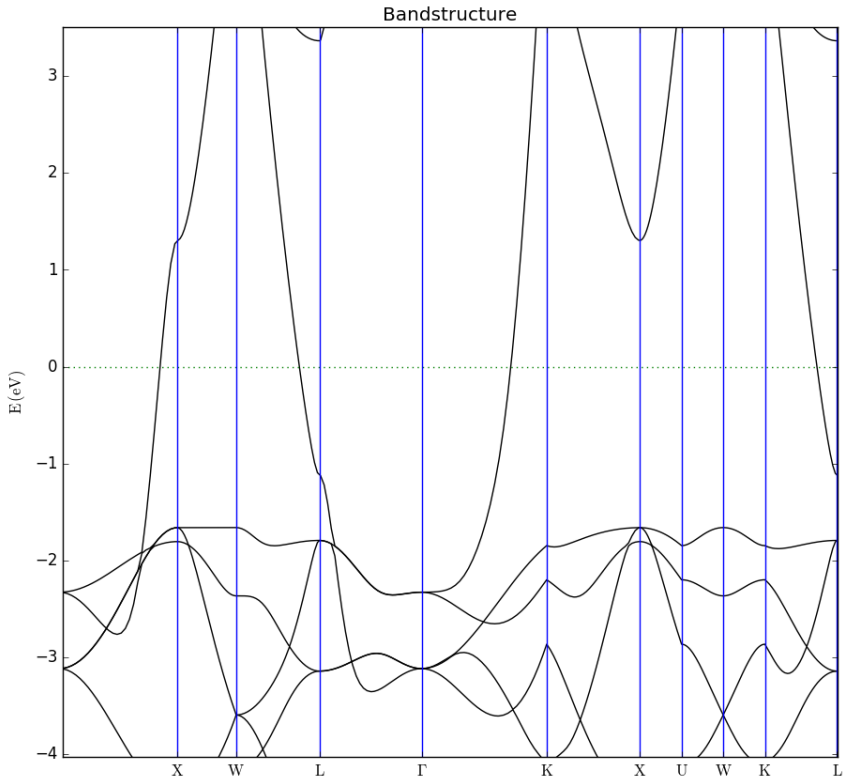
Now that we have found the PW cutoff needed for satisfactory accuracy, we can calculate some properties for bulk Cu. We will do a simple calculation of the lattice constant and the bandstructure. Go back to the Script Generator window.

- Add **OptimizeGeometry** and **Bandstructure** objects.
- Move the objects around to ensure that the is just after the .
- Open the **New Calculator** object and change the cutoff to the 50 Hartree we found previously.
- Open the **OptimizeGeometry** object and un-check *Constrain Lattice Vectors*
- Send the script to the **Job Manager** to run it. It should look like this: `cu-bulk.py`

Running the script took about 10 minutes using 8 MPI processes on an 8-core machine, so anything up to a couple of hours is possible if run on a laptop.

Tip: The ATK-PW calculator uses symmetry to reduce the number of k-points that must be calculated, and it parallelizes over the reduced k-points. In order to get the most efficient parallelization, you should check this reduced number in the Script Generator, and use a number of MPI processes which is as close to a divisor of this number as possible.

You can now inspect the log-file `Copper.log` and the data-file `Copper.hdf5` and verify that we have found a lattice constant for copper of 3.660650 Å after stress and force optimization, a total energy of -4968.92 eV and the following bandstructure, if zoomed in on a range around the Fermi level.



Other analysis objects

Not all QuantumATK analysis objects support the PW calculator in QuantumATK 2017.0, as it is still in beta. However, the **Script Generator** is aware of this fact, and will cross out any unsupported analysis objects, and will omit them when the actual script is generated and sent to the Editor or the Job Manager. If the script is created manually, QuantumATK will give an error message when it encounters an analysis object which does not support the plane-wave calculator.

3.12.15 Build a graphene nanoribbon transistor

In this tutorial, you will learn how to build an advanced device using the QuantumATK **Builder**. We shall consider a nano-scale graphene nanoribbon transistor with a metallic gate separated from the ribbon by a dielectric. This transistor design is inspired by the work presented in[1], where QuantumATK was used to investigate the properties of a similar system.

The device is illustrated in the figure above. It consists of three regions and forms a metal–semiconductor–metal junction. By applying a gate potential across the central region, the system can function as a field effect transistor (FET).

[1] Qimin Yan, Bing Huang, Jie Yu, Fawei Zheng, JiăZang, Jian Wu, Bing-Lin Gu, Feng Liu, and Wenhui Duan. Intrinsic CurrentVoltage Characteristics of Graphene Nanoribbon Transistors and Effect of Edge Doping. *Nano Letters*, 7:1469–1473, 2007. doi:10.1021/nl070133j.

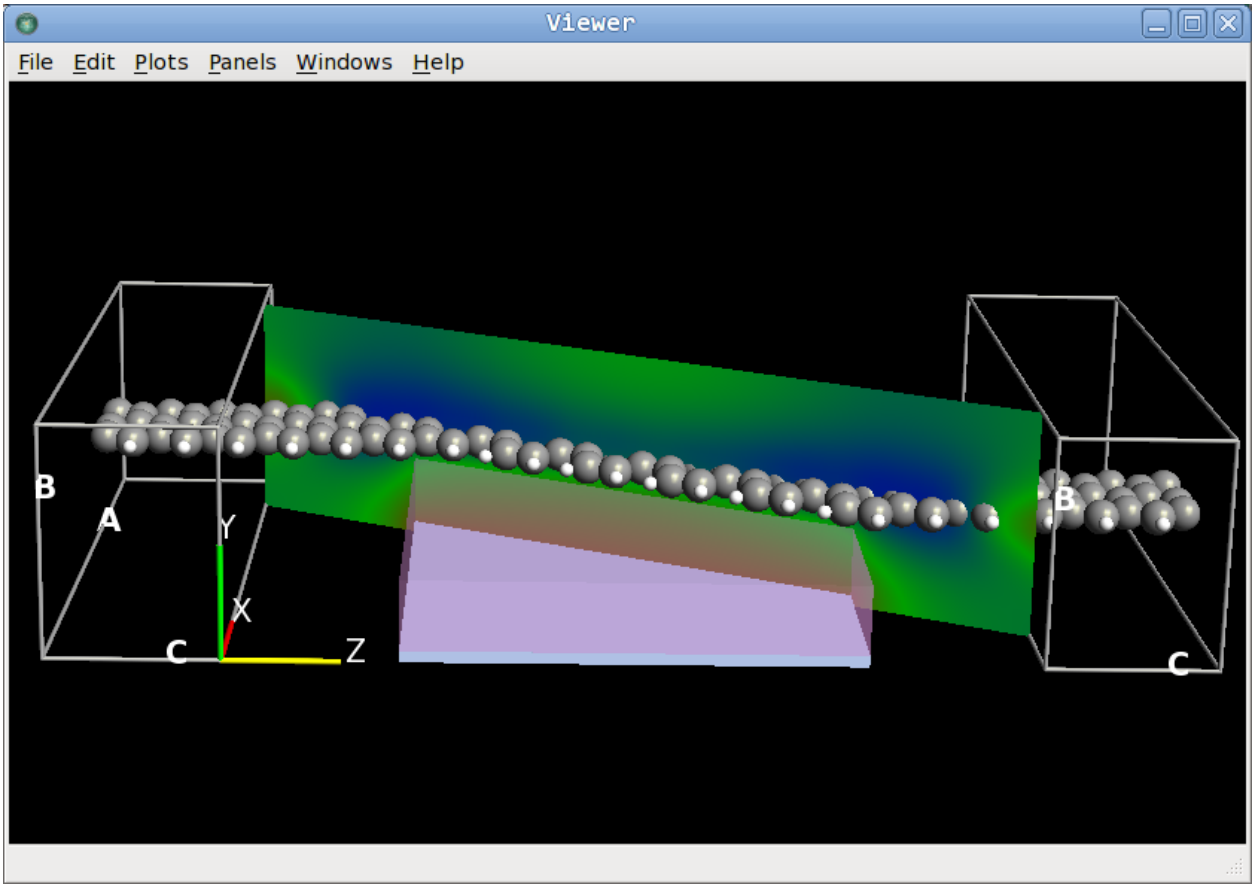


Fig. 3.125: The device system considered in this tutorial consists of a z-shaped graphene nanoribbon on top of a dielectric and controlled by a metallic gate. The contour plot illustrates the electrostatic potential along the transport direction.

Small nanoribbon transistor

The graphene nanoribbon junction device will be of the zigzag–armchair–zigzag type. You will first build the central region (an armchair nanoribbon) and the two electrodes (zig-zag nanoribbons).You will then proceed to merge together the three components and build the complete junction device.

Central region

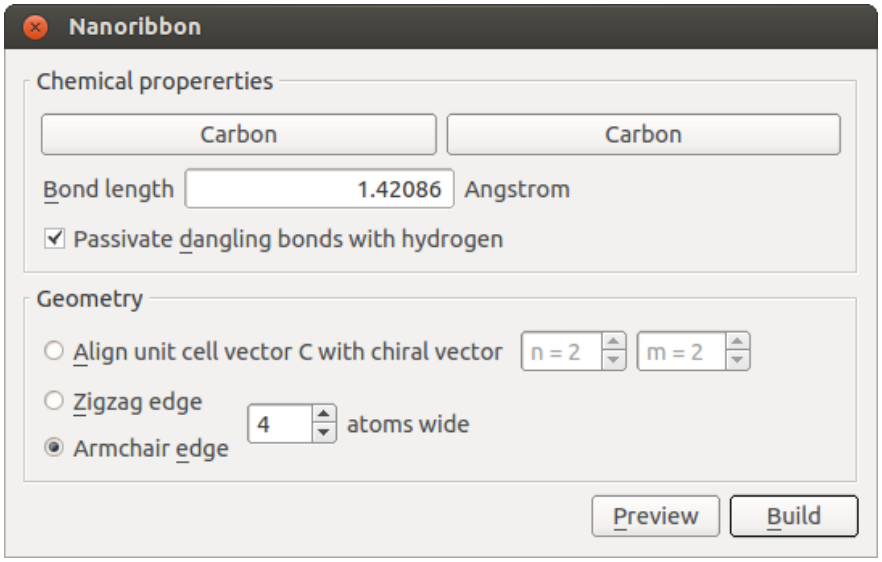
Open the **Builder** and do the following:

•Use the **Nanoribbon plugin** via *Add* → *From plugin* → *Nanoribbon* to create a graphene nanoribbon.

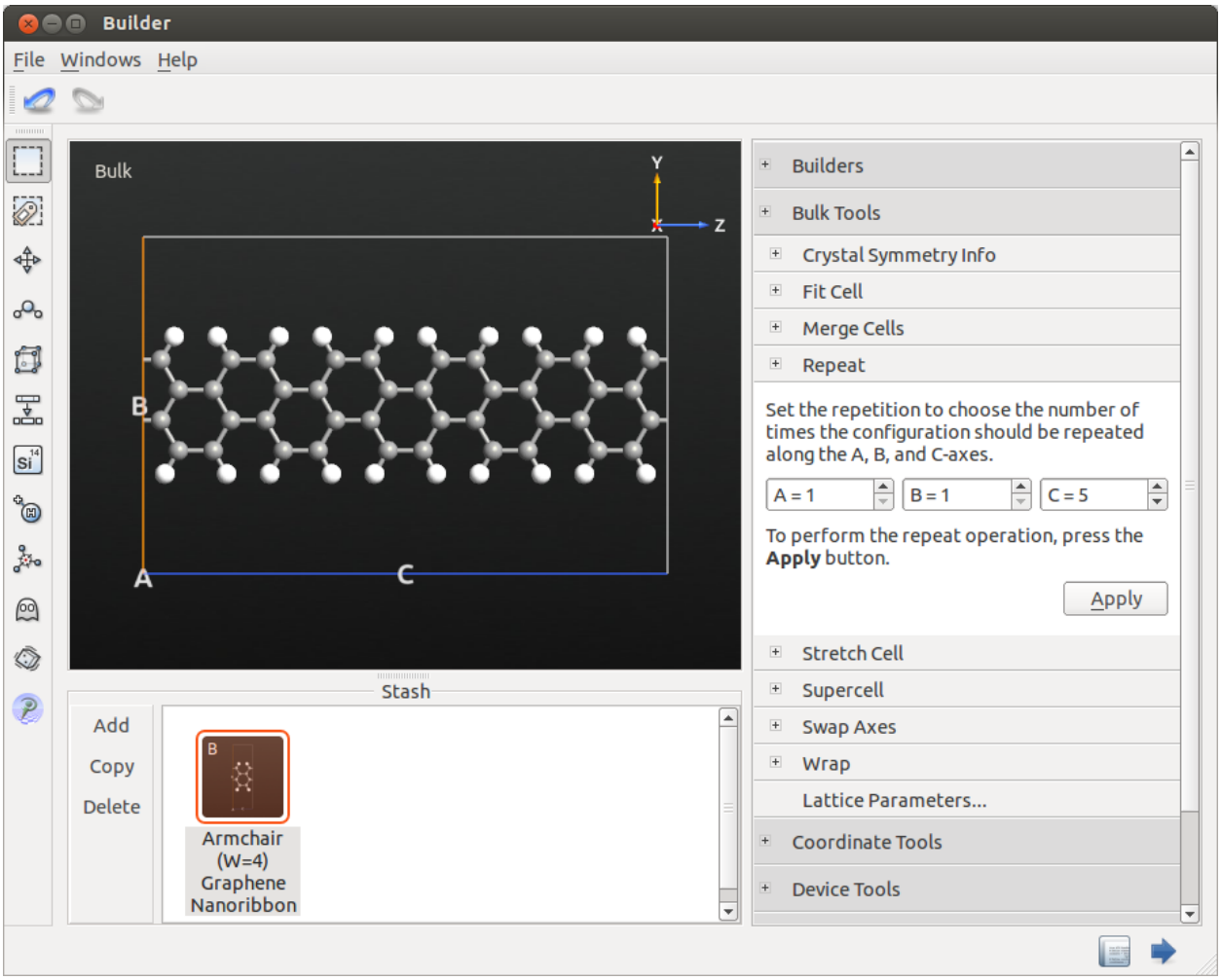
•Tick passivate dangling bonds with hydrogen atoms.

•Choose a width of 4 atoms.

•Choose armchair edge as the ribbon chirality.



•Use *Bulk Tools* → *Repeat* to repeat the ribbon 5 times along the C axis to create the central region.



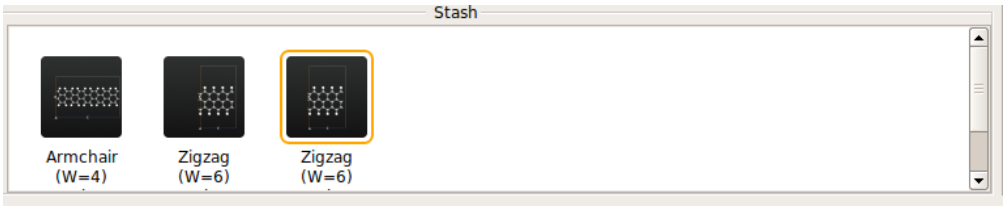
Electrodes

You are now ready to set up the two electrodes, consisting of two identical zig-zag nanoribbons.

•Again, create a ribbon from *Add* → *From Plugin* → *Nanoribbon*. This time you should select a 6 atoms wide zigzag edge nanoribbon.

•Repeat the bulk configuration 4 times along the C axis.

•In order to create two distinct electrodes, make a copy of the newly created zigzag nanoribbon in the Stash.



Merge the three components

Two steps a required:

1. You first create the z-shaped nanoribbon bulk configuration.
2. Then turn the bulk configuration into a **device configuration** used in first principles transport calculations.

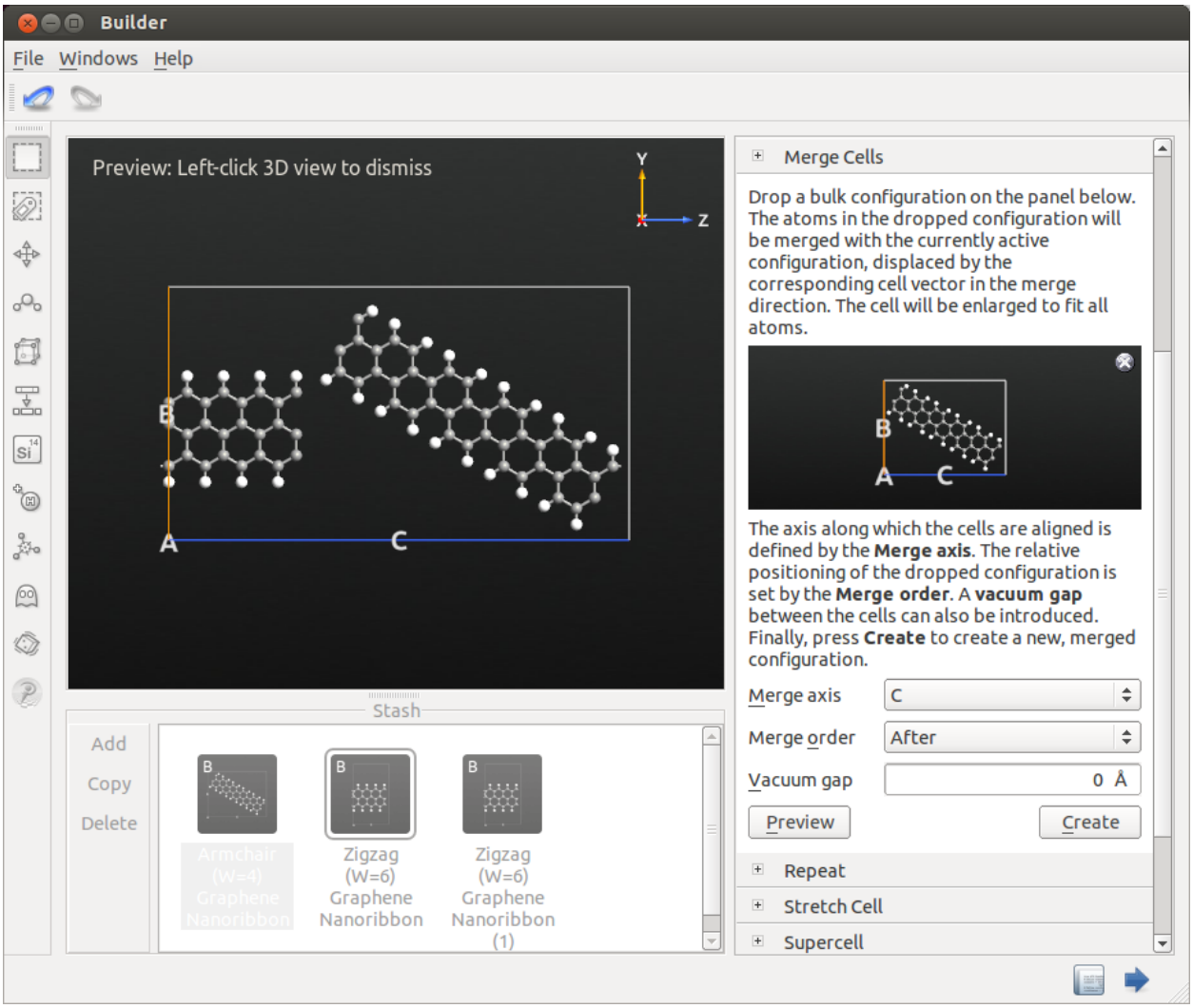
Z-shaped nanoribbon structure

You need to connect the armchair nanoribbon (central part) with the two zigzag nanoribbons (electrodes). First, rotate the central part by 30 degrees:

- Select the armchair nanoribbon from the Stash.
- Use *Coordinate Tools* → *Rotate* to rotate the central region 30 degree around the x axis.
- Go to *Bulk Tools* → *Fit Cell* and fix the cell along the B axis.

Now you can merge the three parts together:

- Select the left electrode by double clicking the configuration on the Stash.
- Select *Bulk Tools*→ *Merge* and drag the central region configuration from the Stash to the small window on the right side as shown below.
- Click **Create**.



•Select all the atoms belonging to the central region and activate the **Move Tool** in the tool bar.

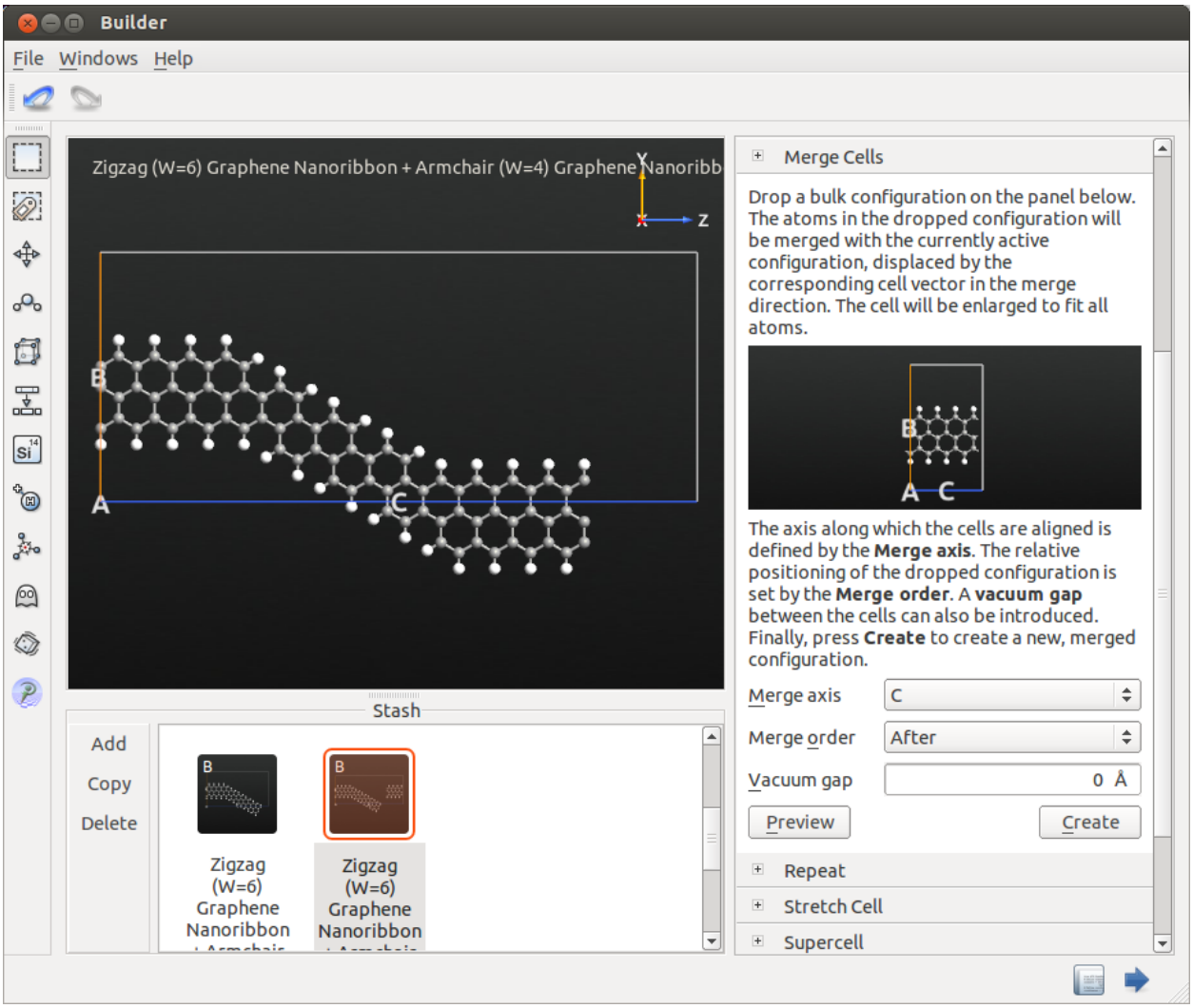
•Select the lower-left edge atom of the central region as shown below. This atom is now the anchor atom which means that all the other atoms will follow this atom when they are moved.

•Be sure that the **Snap** function is selected then drag your anchor atoms close to the right edge atom of your electrode.

•On the Move window you should now be able to see that two atoms are overlapping. Click **Fuse** to delete one atom for each pair of overlapping atoms.

•Close the Move window and **delete** the extra H between the two carbon atoms.

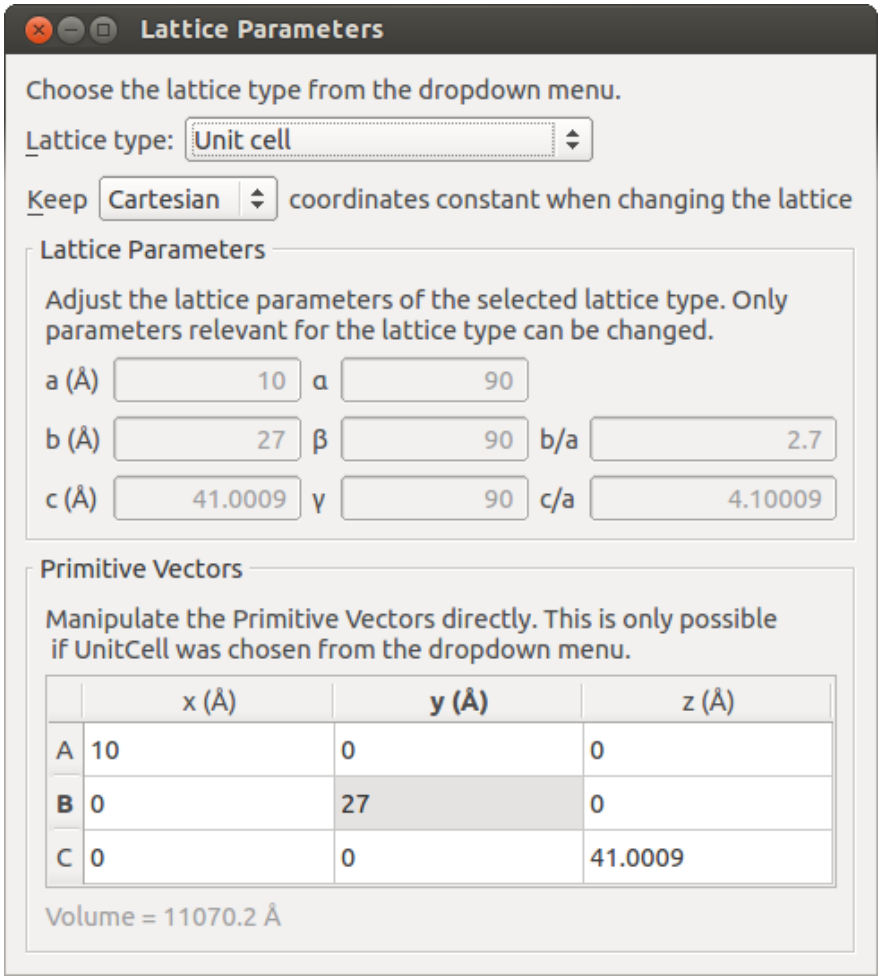
You now have a perfect junction between the left electrode and the central region. Repeat the procedure outlined above to attach the right electrode to the new configuration, and you will have finished the merge.



A proper unit cell

Create a proper unit cell and center the configuration in it:

•Open *Bulk Tools* → *Lattice parameter*, and increase the dimension of the B vector to 27 Å in order to assure sufficient vacuum above and below the configuration.



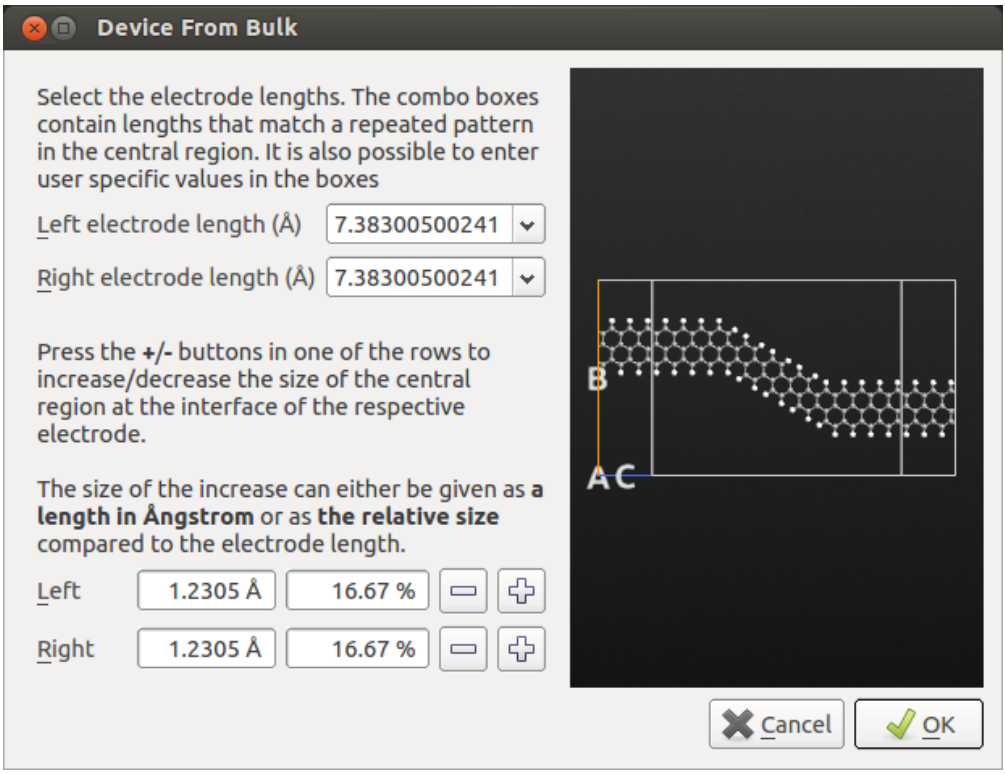
•Use *Coordinate* → *Center* plugin to center the configuration.

The bulk configuration is now finished.

Device from bulk

Transport calculations is a key feature of the QuantumATK platform. The **Device Tools** plugins help you to construct the device with electrodes and gates.

•Highlight the bulk configuration in the Stash, and use the *Device Tools* → *Device From Bulk* tool to construct the graphene nanoribbon device. The tool will automatically select appropriate electrode lengths.



Note: The required length of the electrode cell depends on the computational model employed. Strictly speaking, the length of the cell in the the C-direction should be longer than the interaction distance of the model. For most models a length of about 7 Å will suffice.

Gate and dielectric

Next, add the metallic gate and the dielectrice separator using the **Spatial Regions** tool:

•Open *Miscellaneous* → *Spatial Regions*.

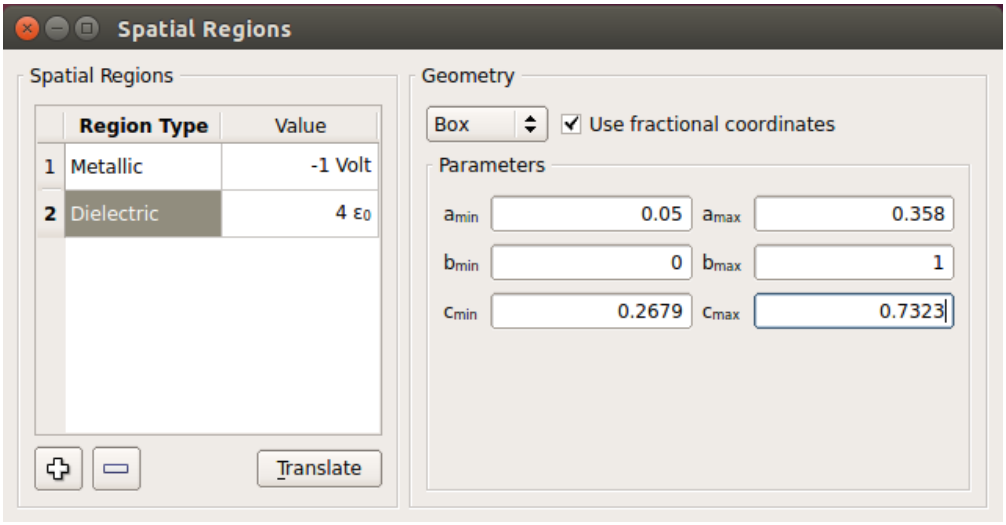
•Click to insert a new spatial region.

•Set the spatial region to be a *dielectric* region and set the dielectric constant to 4.0.

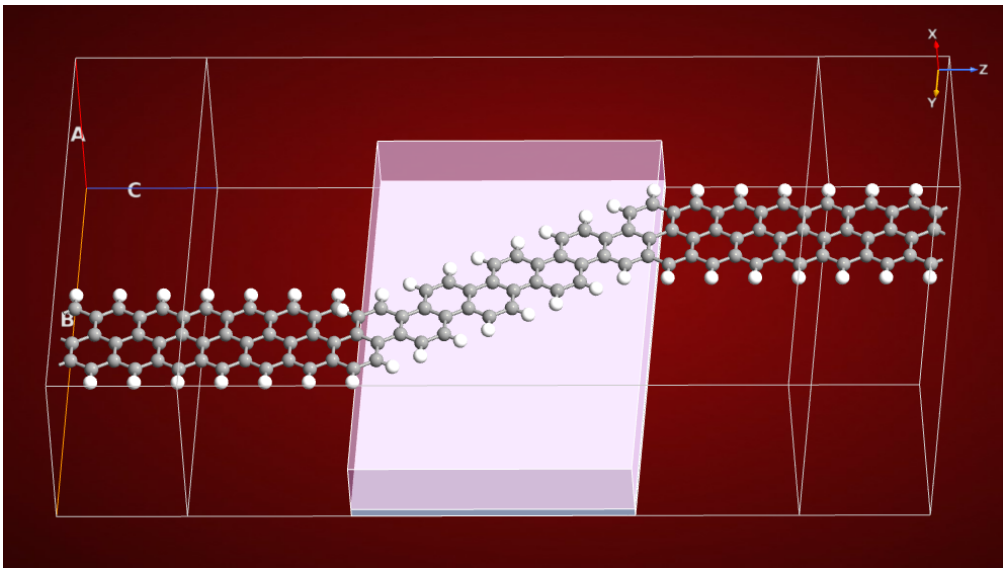
•Use the **fractional coordinate** fields to define the boundaries of the dielectric region: *a:0.05-0.358, b:0-1, c:0.2679-0.7323*.

•Add a second spatial region, set it to be metallic and set the gate potential to -1 V.

•Use the **fractional coordinate** to define the size of the metallic gate: *a:0-0.05, b:0-1, c:0.2679-0.7323*.



Finally, your device should look like illustrated below. You can download a script defining the device configuration here: `device.py`.



Tip: **What next?** You are now ready to perform some transport calculations! For example, check out the tutorial *Graphene nanoribbon device: Electric properties*. It shows you how to compute electron transport properties of the gated graphene nanoribbon device, including the electron transmission spectrum, I–V curve, electron thermal transport, and conductance.

You can also proceed to the next section to learn another way to build a nanoribbon transistor.

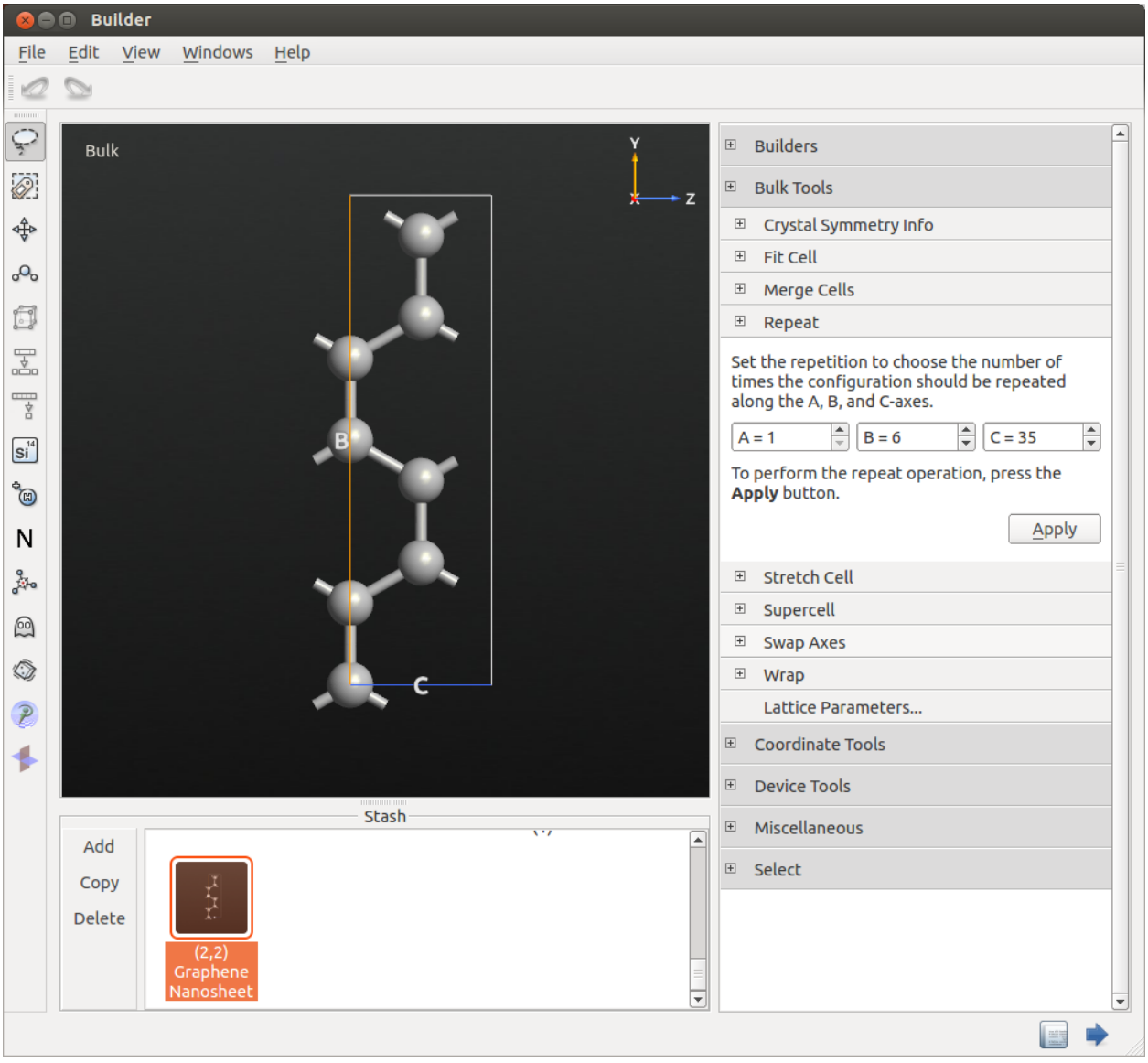
A longer nanoribbon transistor

There are of course several different ways to build a z-shaped graphene nanoribbon device with the QuantumATK Builder. The approach outlined above constructs the three parts of the device separately and then merge them together. Another possibility is to create the device by tailoring an extended graphene sheet. This approach is more flexible, because you can use it to cut out any kind of device based on your needs and imagination.

Use the following this procedure to build a long graphene nanoribbon transistor:

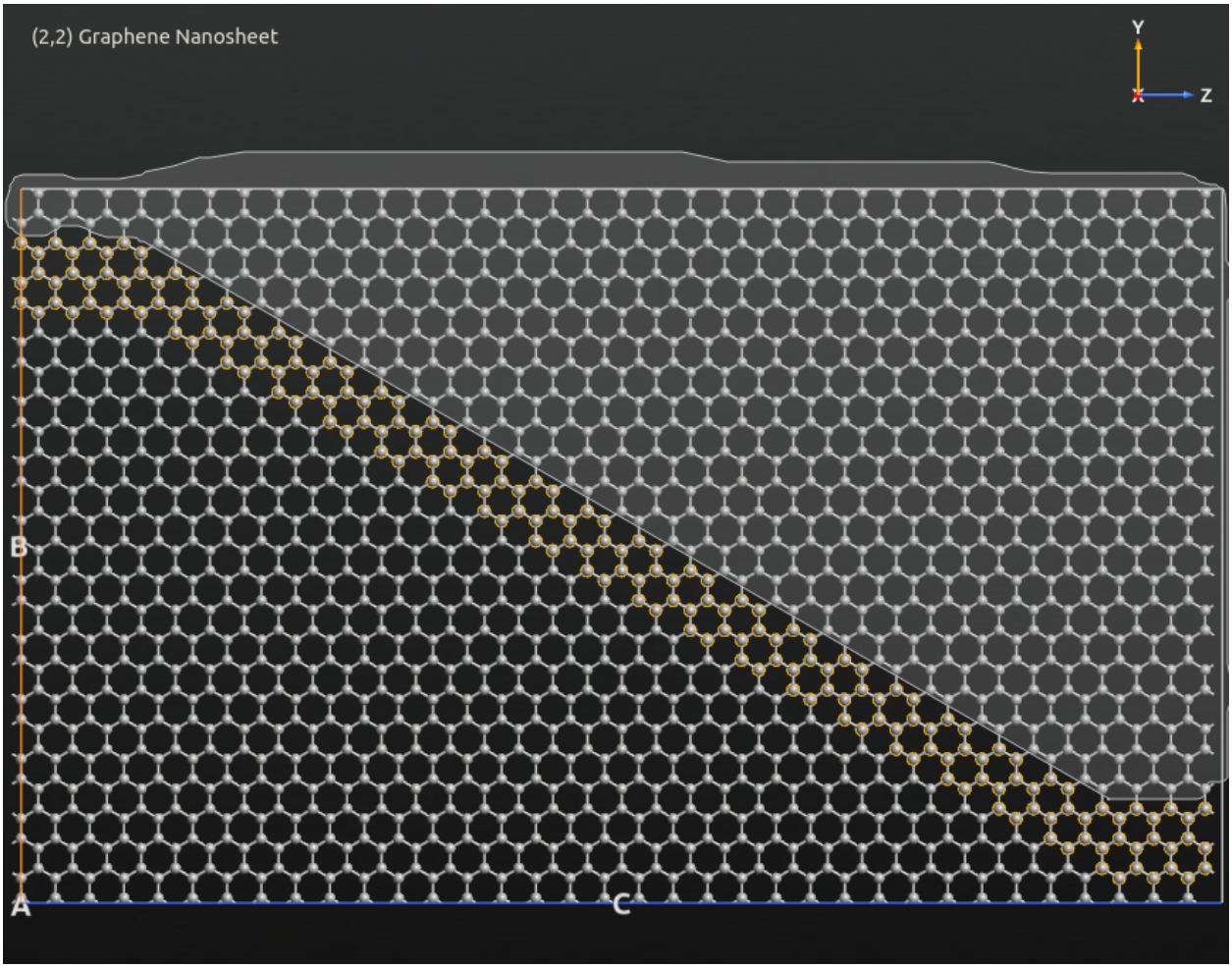
•In the Builder, insert a graphene sheet from *Add* → *From Plugin* → *Nanosheet*. Leave the default parameters unchanged and click **Build**.

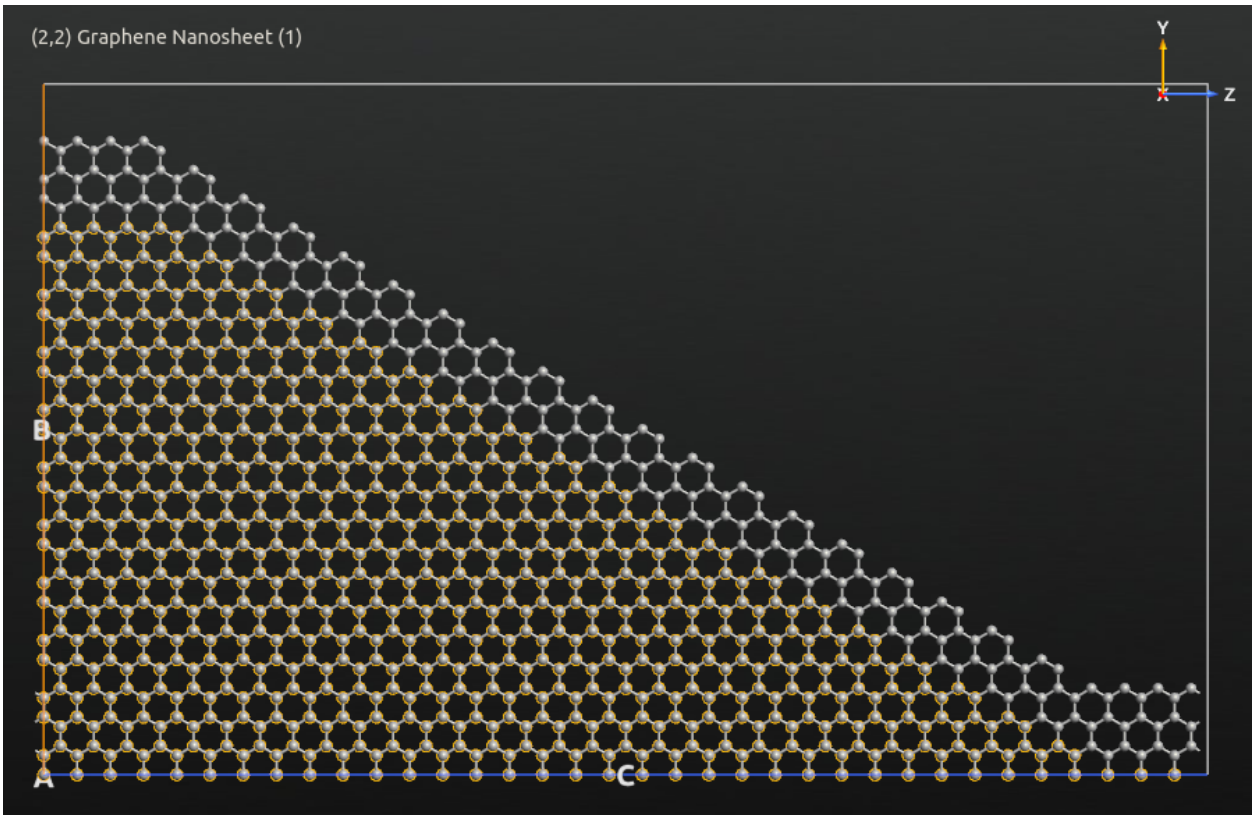
•Use the *Bulk Tools* → *Repeat* plugin to repeat the sheet 1, 6 and 35 times along the A, B and C axes, respectively.



•Open the **Selection Tool** by clicking it and choose the **Lasso** from the pop-up list.

•Use the Lasso to select the regions of atoms you want to cut out, as shown below.


(2,2) Graphene Nanosheet
Y
Z
B
A
C



Tip: You can also use the *Select* → *By Expression* plugin. You will find this tool very handy in case of complicated but geometrically regular shapes. Click the How-To button for more informations about how to use and combine expressions.

To select the atoms that do not belong to the graphene device, use the two following expressions:

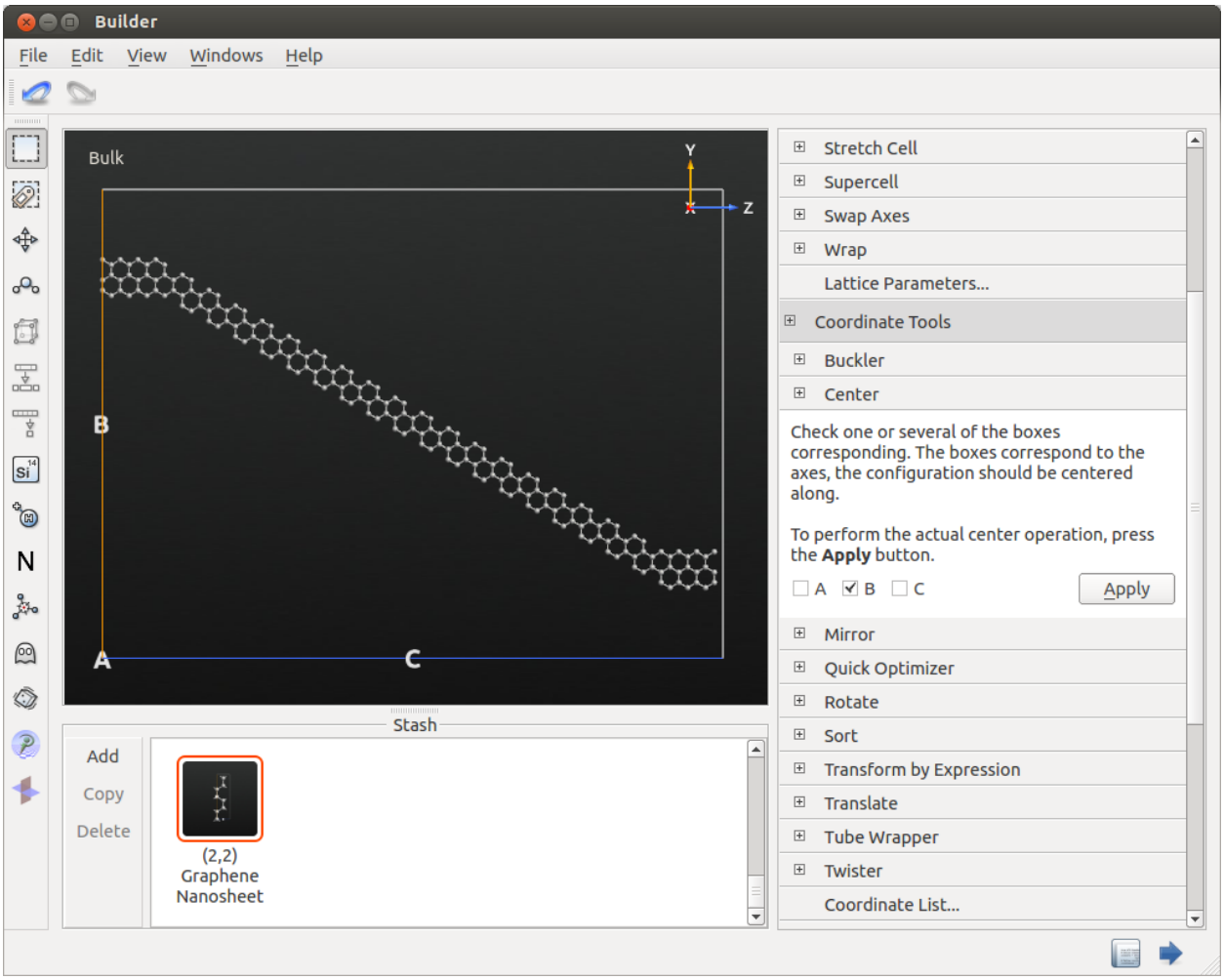
```
(y > 47) or (y > -0.577*z+52 and 7.39 < z and z < 77.5216) or (y > 6.40 and z > 77.5216)
```

```
(y < 1) or (y < -0.577*z+46 and 8.65 < z and z < 77.5216) or (y < 41.91 and z < 8.62)
```

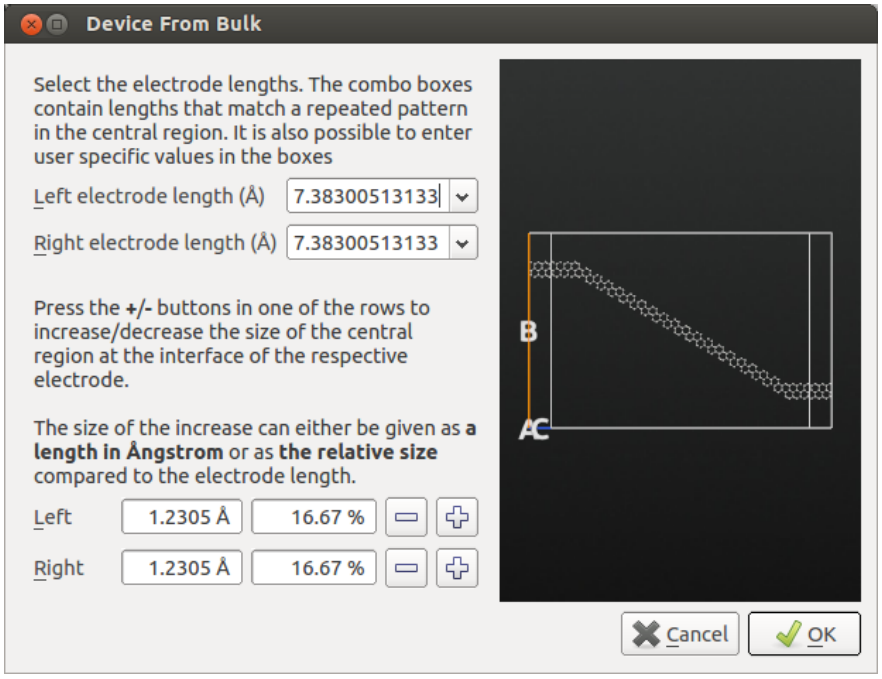
The first expression selects the atoms to delete in the upper part of the device, while the second expression selects the atoms to delete in the lower part.

Whatever method you choose to cut out the z-shaped device, you should now have a nice z-shaped ribbon. Add a bit of vacuum along the B direction:

- Open *Bulk Tools* → *Lattice Parameters* and increase the B vector dimension to 65 Å. Make sure to keep the Cartesian coordinates fixed in this step.
- Use the *Coordinate Tools* → *Center* tool to center the configuration along the B vector.

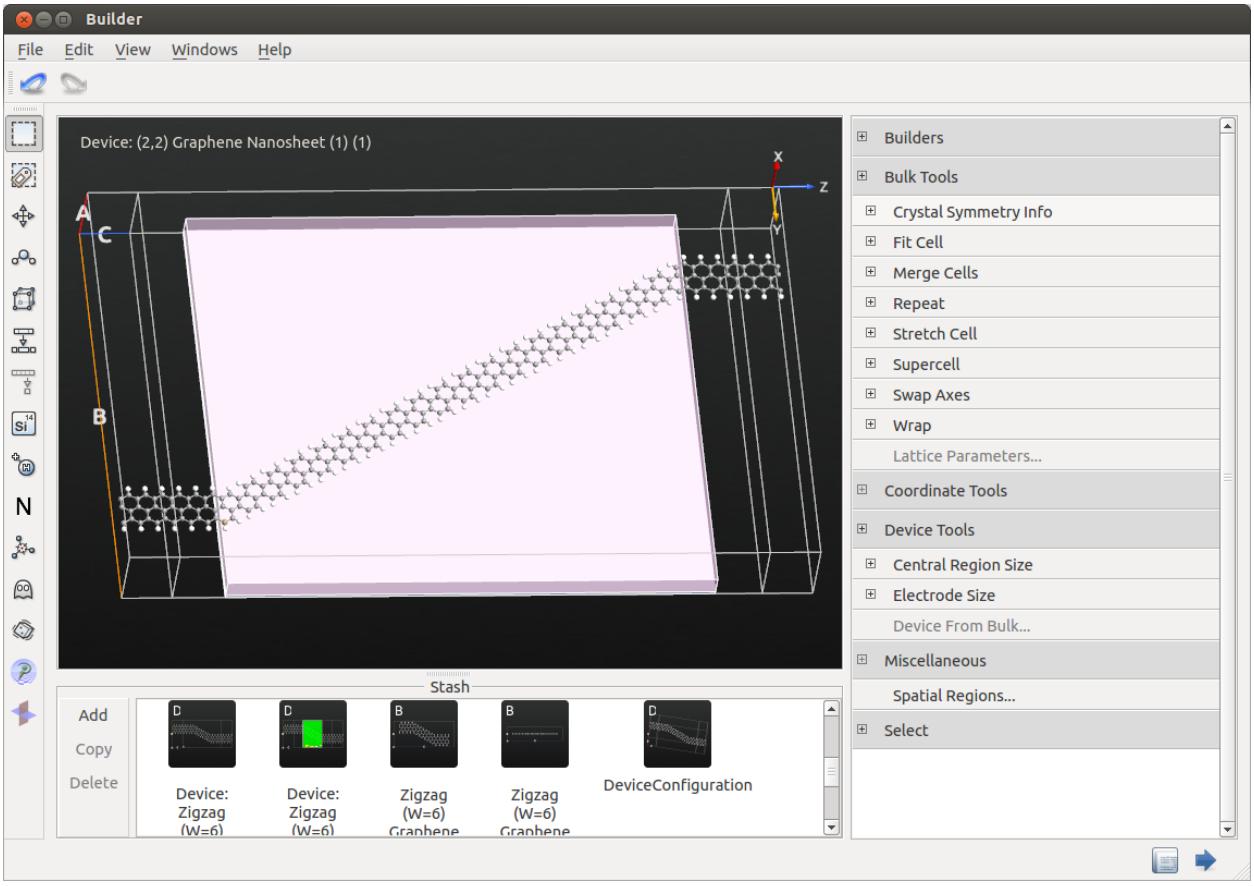


•Next, use the *Device Tools* → *Device From Bulk* tool to create the device configuration. The default electrode lengths are fine.



- The graphene nanoribbon needs to be hydrogen terminated at the edges. Use the Passivate tool to saturate all bonds by hydrogen atoms.
- Next, you should add the metallic gate and dielectric spacer. Open the *Miscellaneous* → *Spatial Regions* plugin and add two regions:
  - Metallic, -1 Volt, fractional coordinates: a:0-0.05, b:0-1, c:0.08-0.92
  - Dielectric, 4, fractional coordinates: a:0.05-0.045, b:0-1, c:0.08-0.92

  You have now created a long z-shaped graphene nanoribbon device on top of a dielectric and controlled by a metallic gate.

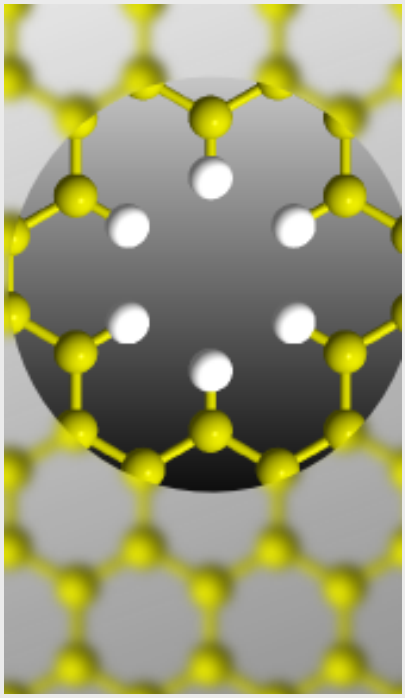


Tip: **What next?** The transistor you have just built is used in the second part of the tutorial *Graphene nanoribbon device: Electric properties*. Go directly to the relevant section by following this link: *Comparison to results for a longer device*.

References

3.12.16 Nanosheet with a hole

QuantumATK: Nanosheet Platforms: *This quick tutorial shows you how to make a small defect in graphene, viz. a hole, and passivate it with hydrogen atoms to saturate all the bonds.*



Here are the instructions:

•**Step 1:** Click *Add* → *From Plugin* → *Nanosheet*.

•**Step 2:** Set **n=1, m=1** and click **Build**. This inserts a simple orthorhombic unit cell of graphene.

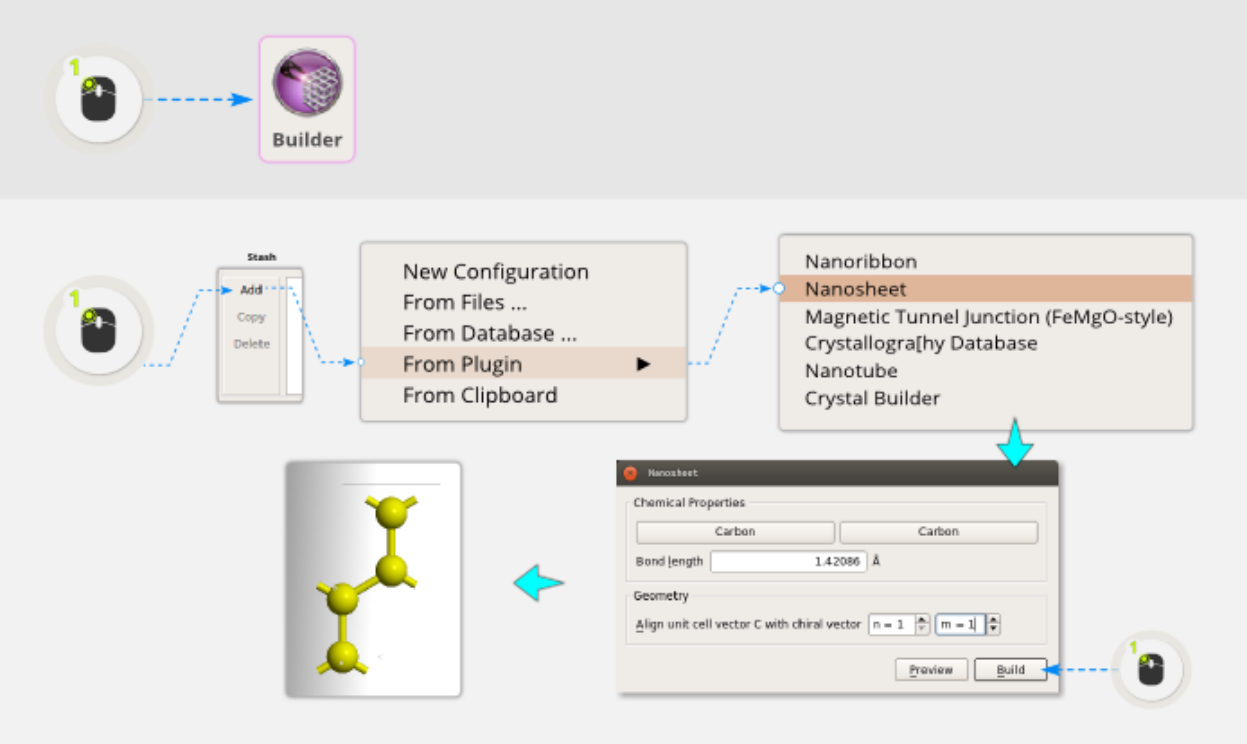
•**Step 3:** Use *Bulk Tools* → *Repeat* to make a larger structure. A repetition of 1x6x6 is suitable.

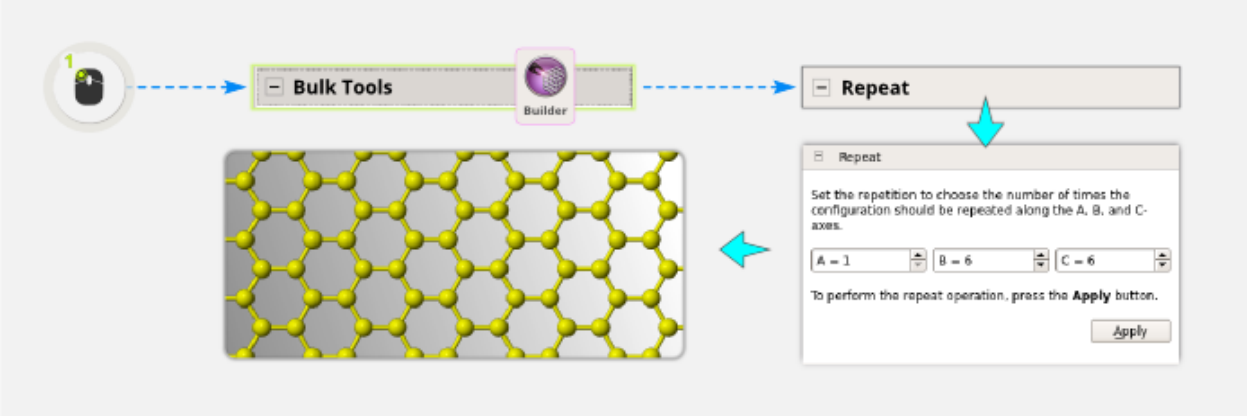
•**Step 4:** Press Ctrl+R in the plot window, to center the view.

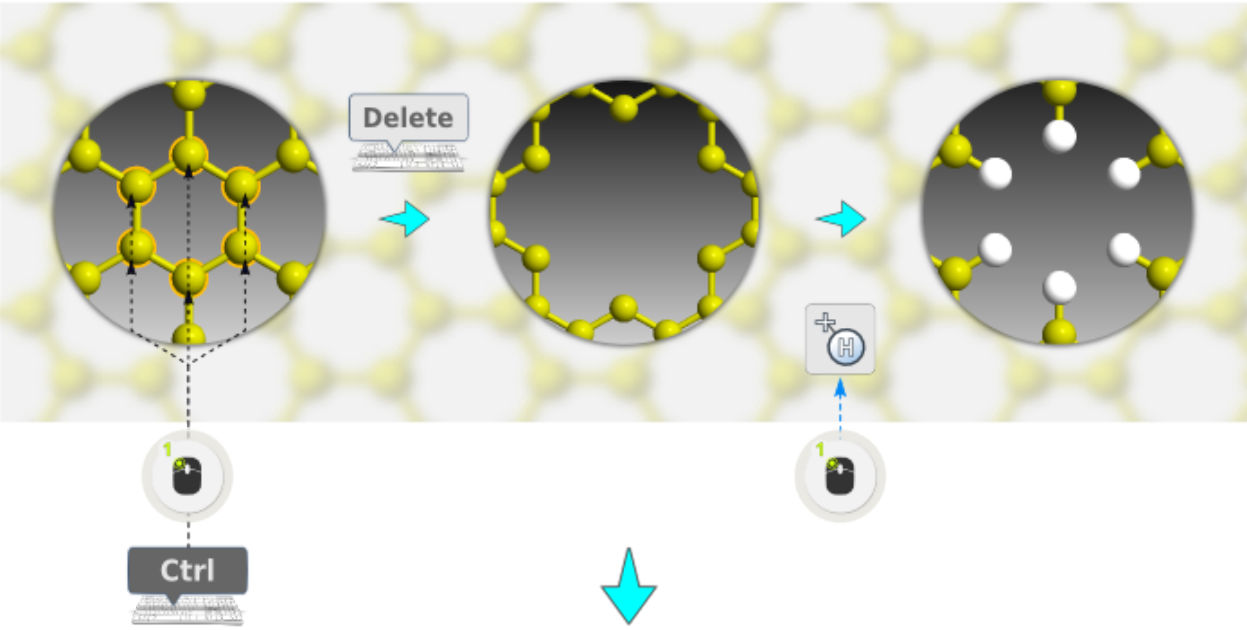
•**Step 5:** Switch to the circle select tool .

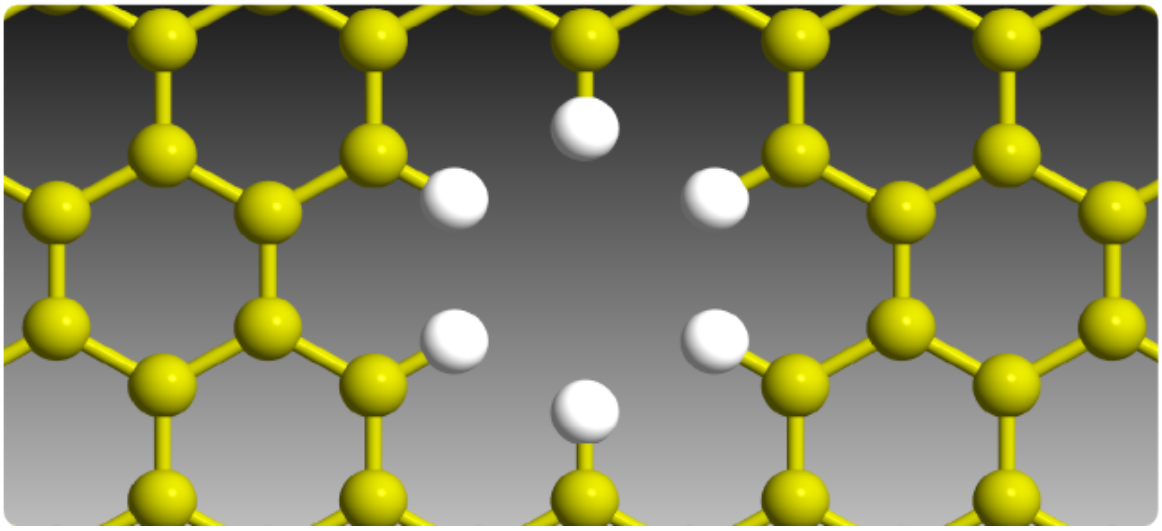
•**Step 6:** Select a hexagon, press Delete on the keyboard.

•**Step 7:** Click the passivate button on the toolbar.


Builder
Stash
Add
Copy
Delete
New Configuration
From Files ...
From Database ...
From Plugin
From Clipboard
Nanoribbon
Nanosheet
Magnetic Tunnel Junction (FeMgO-style)
Crystallogra[hy Database
Nanotube
Crystal Builder
Nanosheet
Chemical Properties
Carbon
Carbon
Bond length
1.42086
Å
Geometry
Align unit cell vector C with chiral vector
n = 1
m = 1
Preview
Build


Bulk Tools
Builder
Repeat
Repeat
Set the repetition to choose the number of times the configuration should be repeated along the A, B, and C-axes.
A = 1
B = 6
C = 6
To perform the repeat operation, press the Apply button.
Apply


Delete
Ctrl



3.12.17 MoS_2 Nanotubes

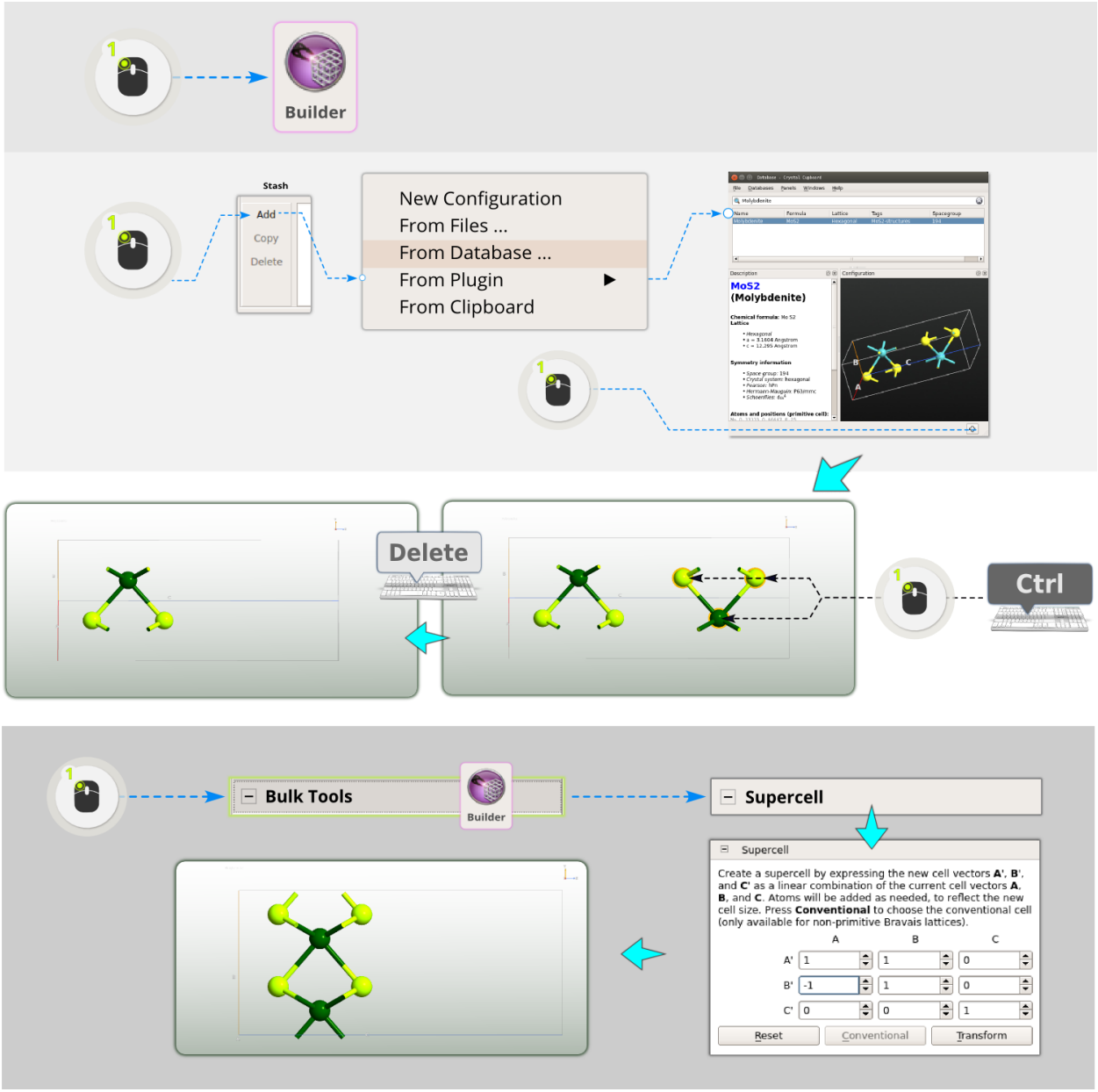
A logical approach seems to be to first make a hexagonal sheet, just like graphene, and then wrap it into the shape of a tube. Indeed, such a **TubeWrapper addon** has been developed, and in this tutorial you will learn how to use it to build MoS_2 and other transition metal sulfide TS_2 nanotubes (T = W, Nb, Re), see Refs.[1] and[2].

> **Attention:** The recipe is simple enough:
>
> 1. Create a bulk crystal of MoS_2.
> 2. Change it into a monolayer.
> 3. Wrap it using the **TubeWrapper addon**.
>
> This approach can then easily be adapted for other TS_2 structures.

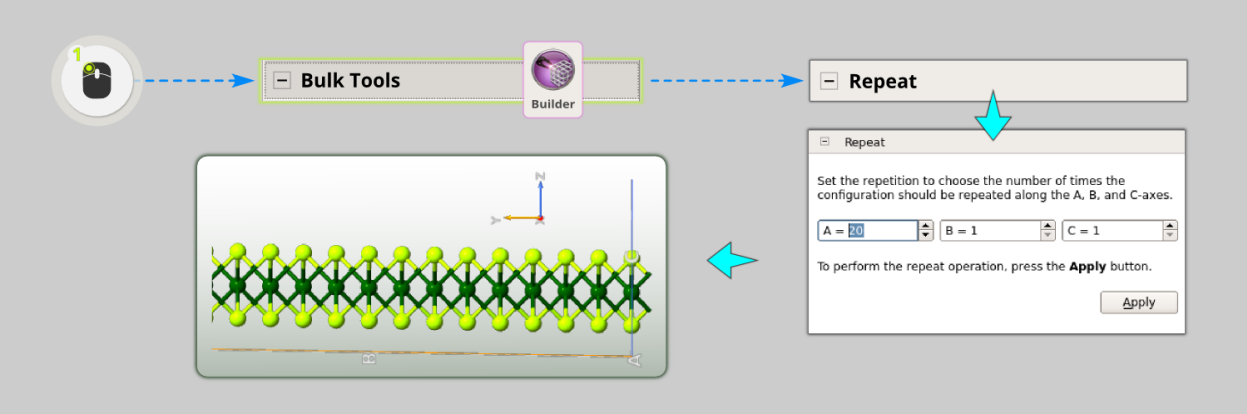
- **Step 1:** Download the `TubeWrapper.zip` file and install the addon following the instructions in the *AddOns* page.
- **Step 2:** Click *Add* → *Add from Database* and locate **molybdenite**. Add it to the Stash (clicking). Use the mouse to select and delete the three atoms to the right in the cell. This leaves a single layer of MoS_2. The cell is hexagonal cell, but to make the tube it is necessary to have an orthorhombic supercell to work with. Therefore, open *Bulk Tools* → *Supercell* and use the settings illustrated below.

[1] A.ăKuc, N.ăZibouche, and T.ăHeine. Influence of quantum confinement on the electronic structure of the transition metal sulfide TS2. *Phys. Rev. B*, 83:245213, Jun 2011. doi:10.1103/PhysRevB.83.245213.

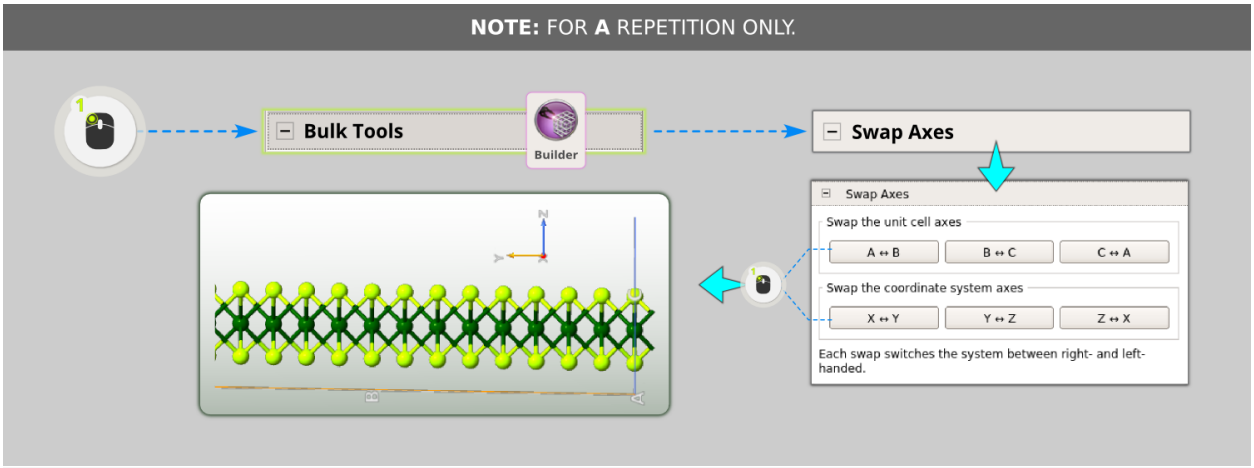
[2] Gotthard Seifert, Humberto Terrones, Mauricio Terrones, Gerd Jungnickel, and Thomas Frauenheim. Structure and Electronic Properties of MoS2 Nanotubes. *Phys. Rev. Lett.*, 85:146–149, Jul 2000. doi:10.1103/PhysRevLett.85.146.



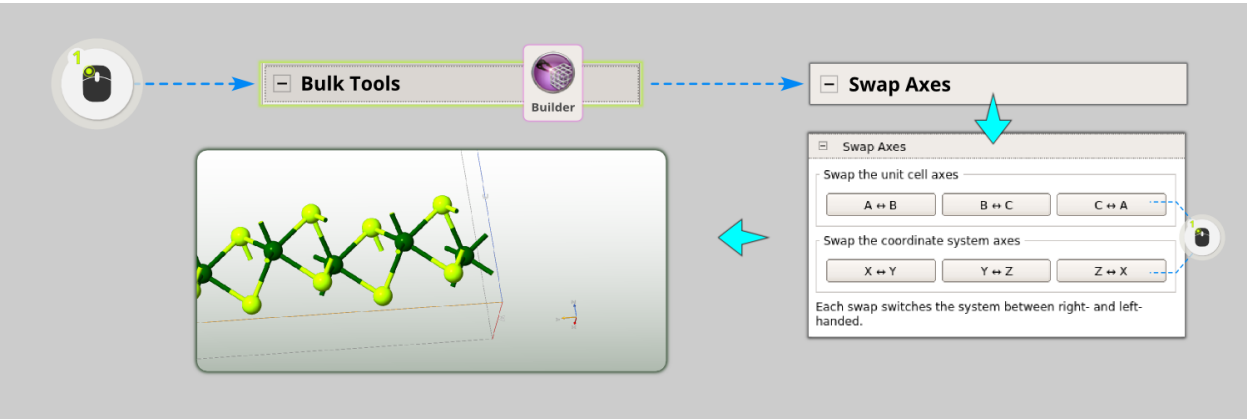
•**Step 3:** Open *Bulk Tools* → *Repeat*. You now have two choices: Repeat in **A** or **B** – this will give either armchair (A) or zigzag (B) tubes. Repeat some decent number, like 20 – the new size of the system in the repeated direction will directly determine the circumference of the tube.



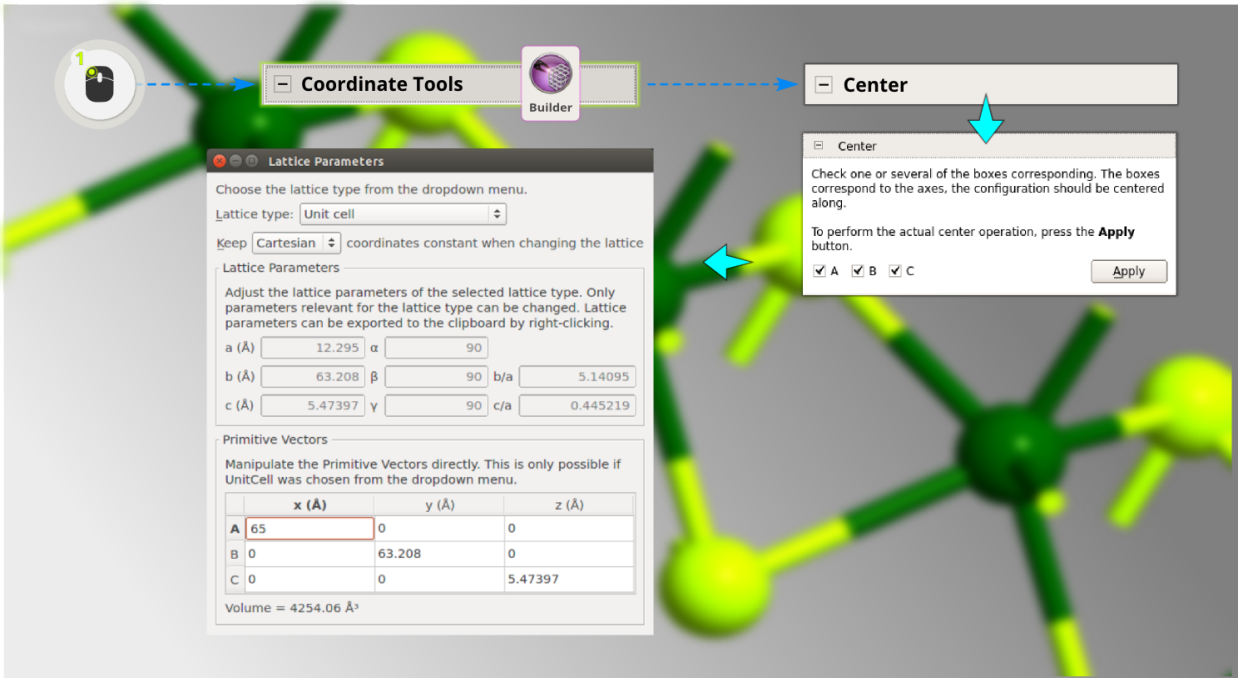
Note: **If you repeated in A**, use *Bulk Tools* → *Swap Axes* and click **A<->B** then **X<->Y** to reorient the system. If you repeated in B, skip this step.



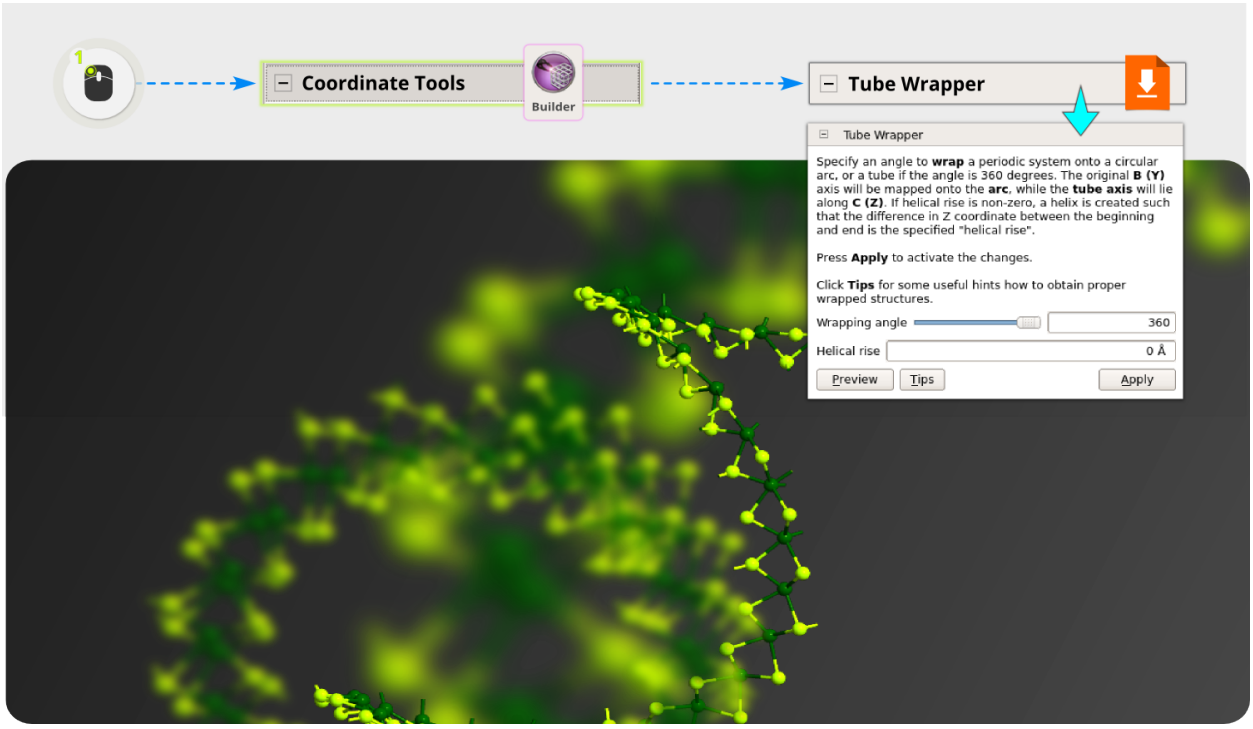
•**Step 4:** Using *Bulk Tools* → *Swap Axes*, click **C<->A** then **X<->Z** (do this regardless of whether you repeated in **A** or **B**):



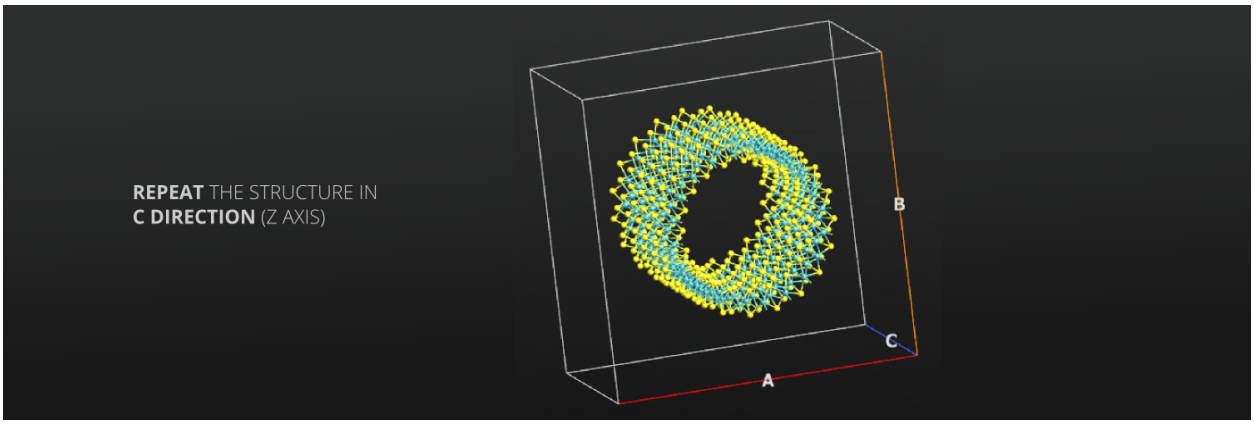
•**Step 5:** Open *Bulk Tools* → *Lattice Parameters* and select **Cartesian** coordinates. Then change the first cell vector **A** so that the size of the cell in X is at least as large as in Y. Open *Coordinate Tools* → *Center* and center the system.



•**Step 6:** Finally, open the plugin you just installed, *Coordinate Tools* → *Tube Wrapper* and wrap the tube. You can use 360 degrees directly to get a fully wrapped tube, or perhaps 180 degrees for some interesting novel work!



•**Step 7:** In the end, its a good idea to adjust the cell size (it was necessary to keep it large during the building procedure) and then center the system in **A** and **B** and afterwards repeat in z direction:



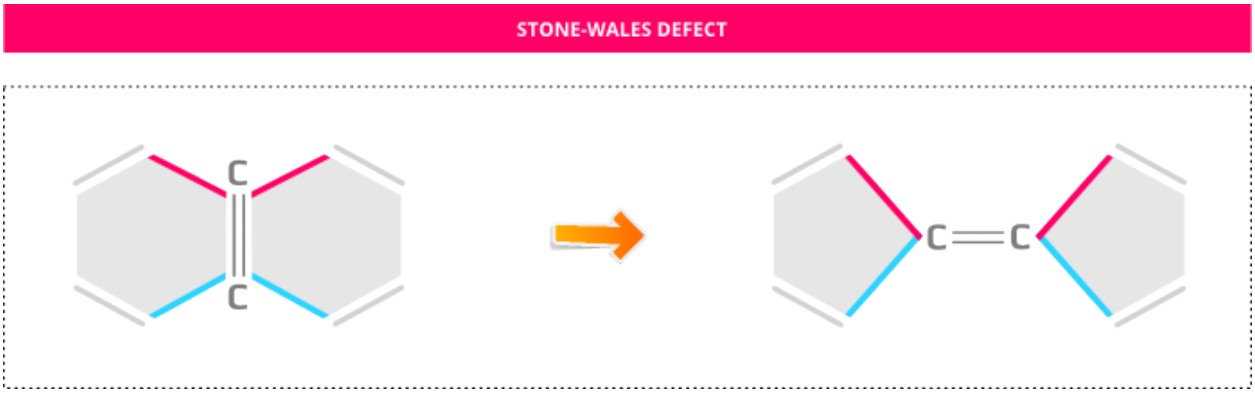
Tip: Whats next? Now you can perform calculations on MoS_2 nanotubes just as easily as CNTs. The DFTB model used in Ref.[Page 1821, 2] is also available in QuantumATK, and you can request the Mo/S parameters from , and run QuantumATK with those. How to install DFTB parameters is described in another tutorial: *The DFTB model in ATK-SE*.

You can of course also use the method outlined here to make carbon nanotubes, even though the dedicated plugin *Add → Add from Database → Nanotube* is much faster to use for a perfect, simple CNT. An interesting example where the Tube Wrapper plugin is anyway useful also for CNTs is for the creation of Stone–Wales defects, which are much easier to define in for a mololayer sheet than for the rolled-up tube. Or, you can make open or partially wrapped nanotubes.

References

3.12.18 Stone–Wales Defects in Nanotubes

A Stone–Wales defect changes the connectivity of two π-bonded carbon atoms by rotating them 90ř with respect to the midpoint of their bond, as illustrated below:



It is a crystallographic defect that occurs in e.g. nanotubes and graphene, and can have a strong influence on the electronic, chemical, and mechanical properties. It is named after Anthony Stone and David Wales of Cambridge University, who described it in a 1986 paper on fullerenes[1], but a similar defect was described much earlier by Peter Thrower in a paper on defects in graphite[2].

Note: You will create the Stone–Wales defect in graphene and then wrap it into a tube. You will therefore need the

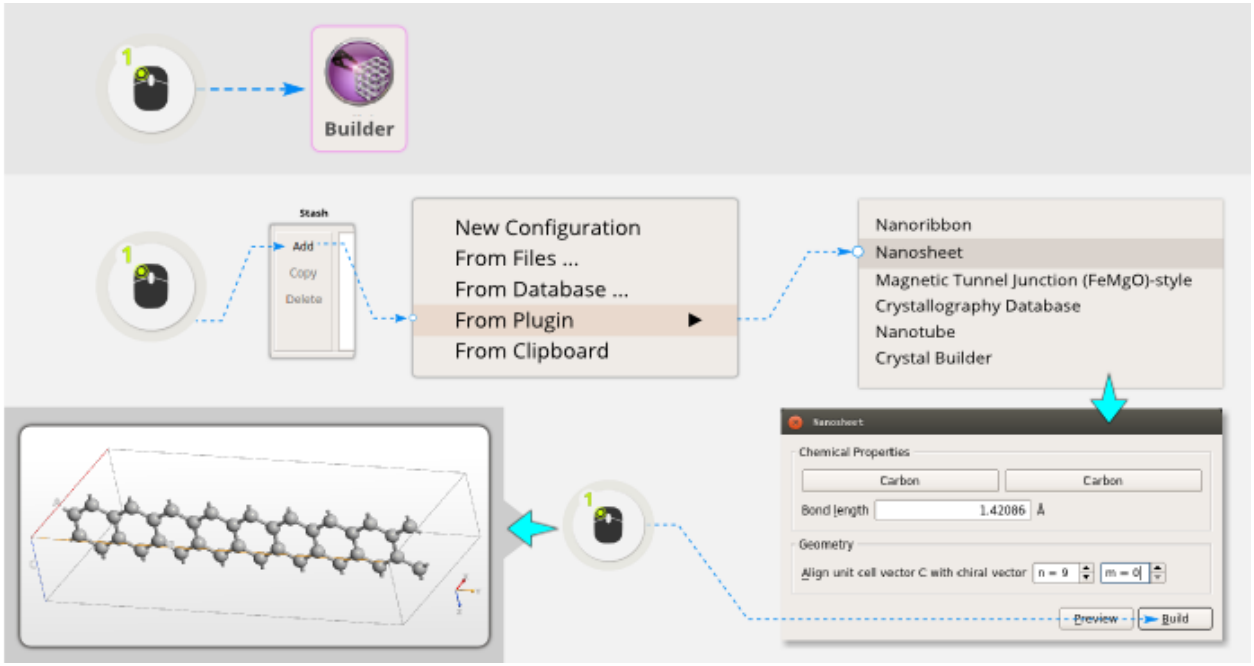
[1] A.J. Stone and D.J. Wales. Theoretical studies of icosahedral C60 and some related species . *Chemical Physics Letters*, 128(5-6):501–503, 1986. doi:10.1016/0009-2614(86)80661-3.

[2] P.ăThrower. The study of defects in graphite by transmission electron microscopy. *Chemistry and Physics of Carbon*, 5:217–320, 1969.

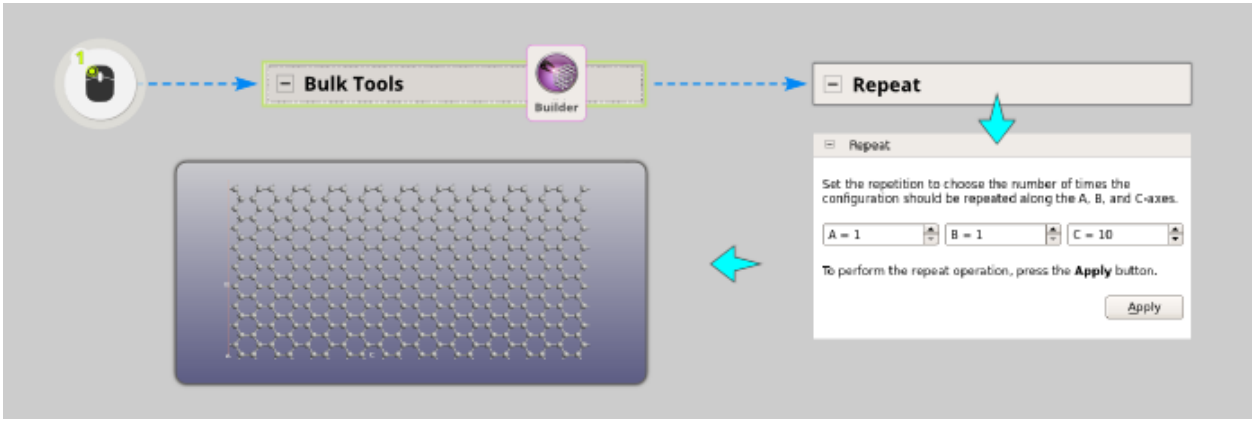
TubeWrapper addon. Download the `TubeWrapper.zip` file and install the addon following the instructions in the *AddOns* page.

Creating the defect and wrapping the tube

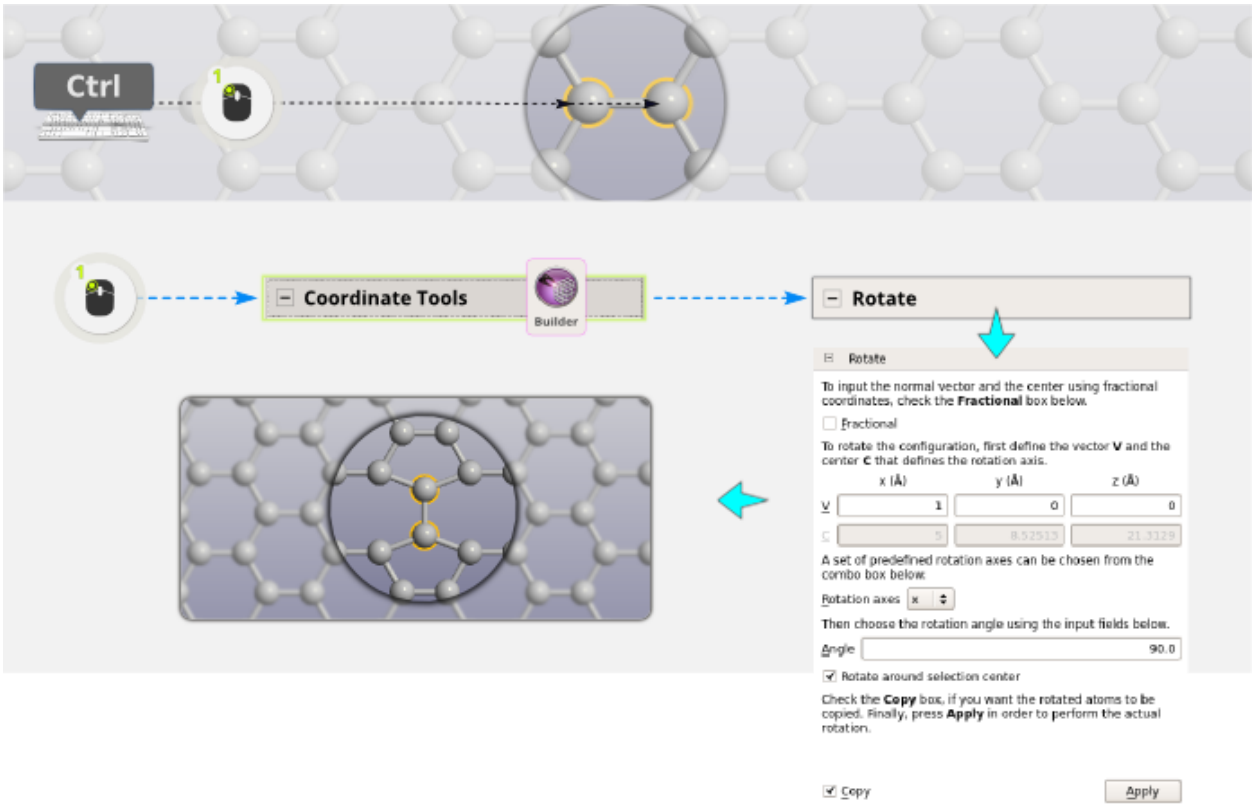
•**Step 1:** First create a graphene sheet by using the *Add* → *Add From Plugin* → *Nanosheet* plugin. Select **n=9** and **m=0** for the chiral indices, and click **Builder**.



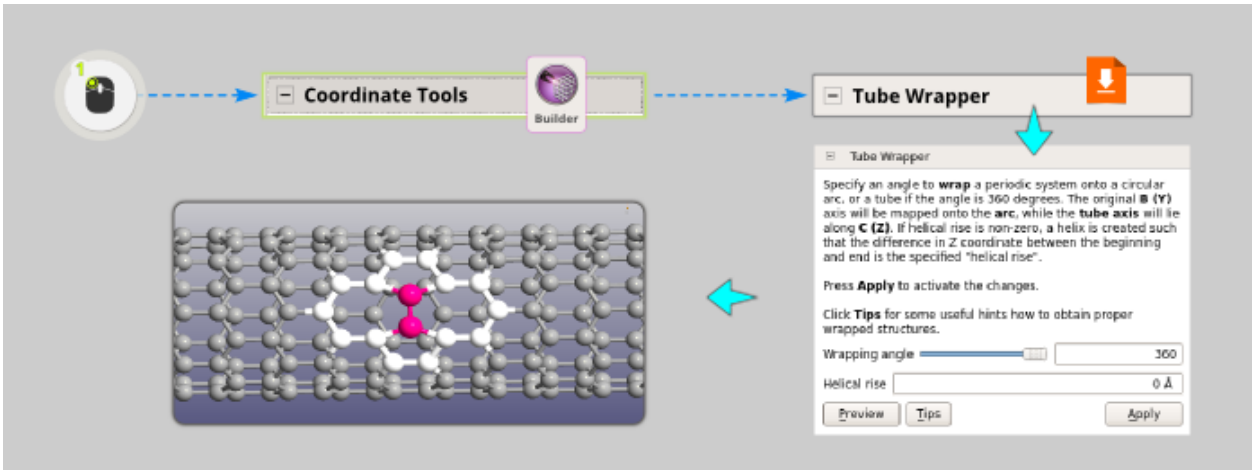
•**Step 2:** The purpose is to study an isolated defect, so it is necessary to make a large structure where a single carbon–carbon bond can be rotated. **Repeat** the configuration 10 times along the C direction using the *Bulk Tools* → *Repeat* tool, and press Ctrl+R to reset the view.



•**Step 3:** Next, **create the defect**: Select two atoms in the middle of the structure, and open the *Coordinate Tools* → *Rotate* tool. Select the rotation axis to be *X* (this is the default) and enter 90 degrees for the rotation angle. It is important that the checkbox *Rotate around selection center* is ticked. Then click **Apply**.



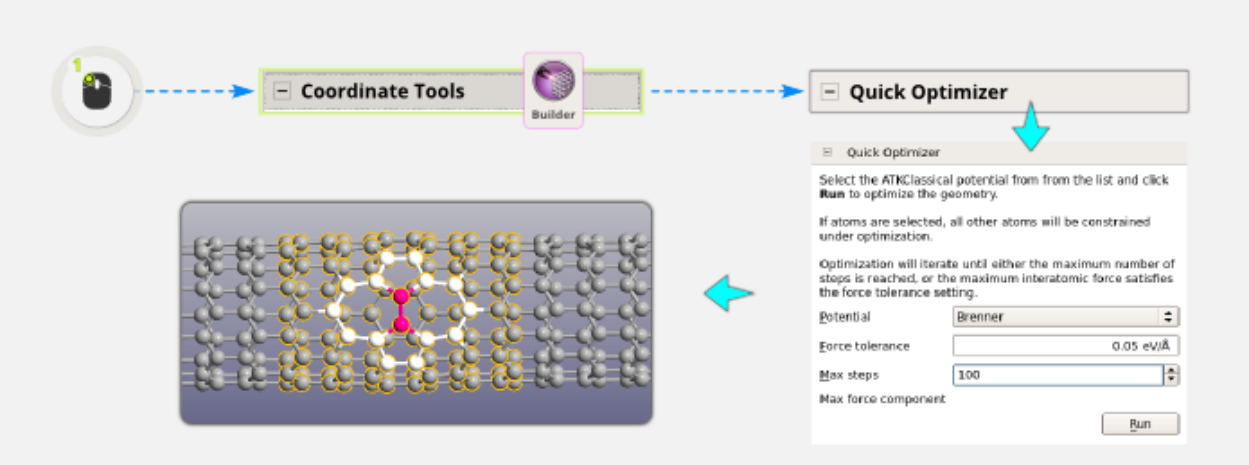
•**Step 4:** You can now **wrap** the graphene sheet into a tube. The sheet is periodic along the B and C directions. The tube axis should be along C, so the graphene sheet should be rolled up along B. Use the *Coordinate Tools* → *Tube Wrapper* tool to accomplish this; set the angle to 360 degrees and click **Apply**.



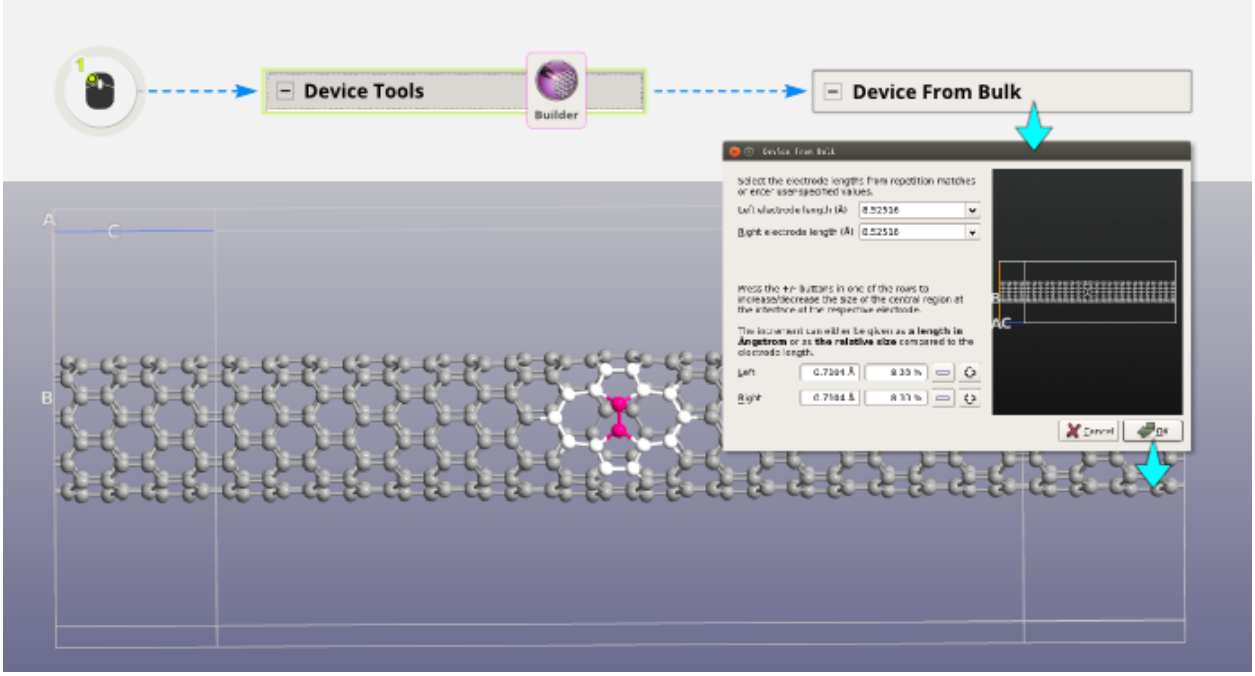
Optimizing the structure

The defect was created by a simple rotation of two carbon atoms, and is most likely not in its equilibrium shape. You should therefore geometry optimize the defect. The **ATK-ForceField** engine with the Brenner potential will provide a great starting point (conveniently available in *Coordinate Tools* → *Quick Optimizer*), but for a truly correct structure, you should run a full DFT optimization. In both cases, the ends of the tube should be constrained so they remain perfectly periodic, in order to use it for a device calculation later on.

Anyway, as a first approximation, select the defect atoms (white and pink in the figure below), and atoms immediately around the defect, to locally optimize the defect geometry. Use the **Quick Optimizer** with the Brenner potential (you will need to increase the number of steps to 100).



Then, convert the structure to a **device** using *Device Tools* → *Device From Bulk* (just click Create, the default settings are fine).



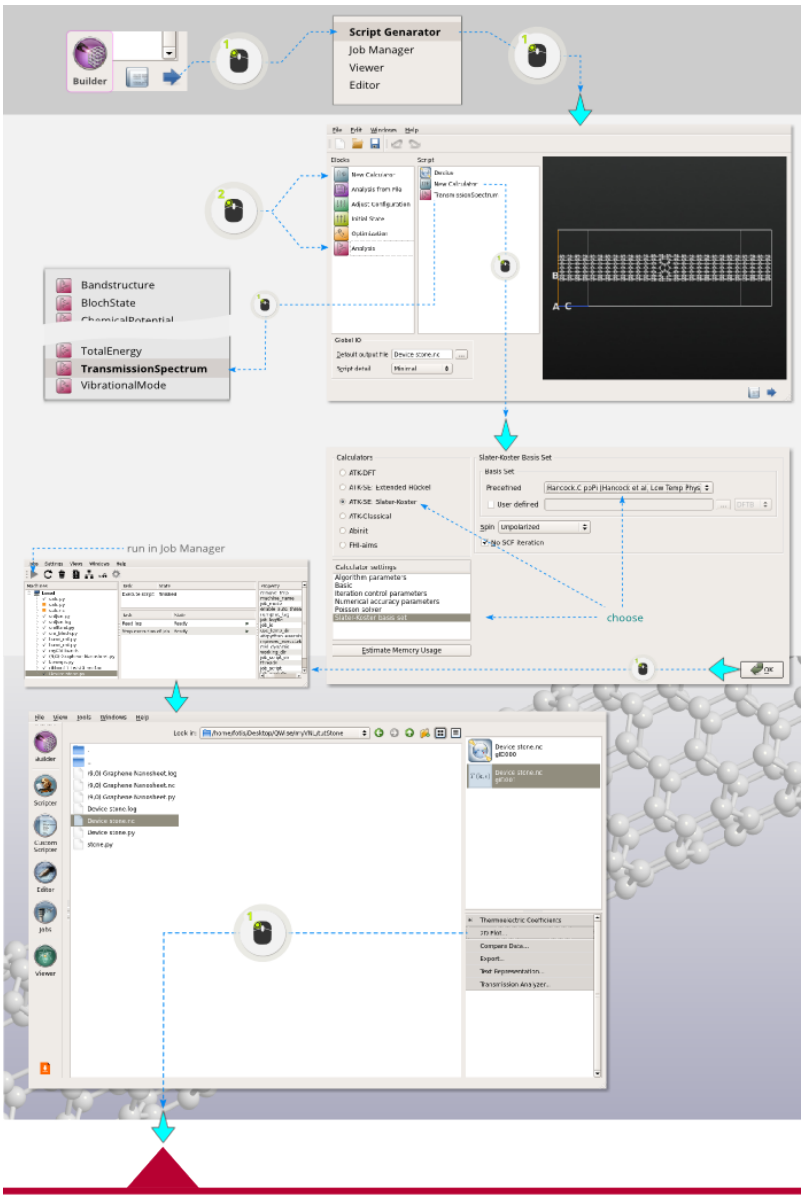
Save the device in a file (say) as **Device_stone.hdf5** in your project directory for other future calculations.

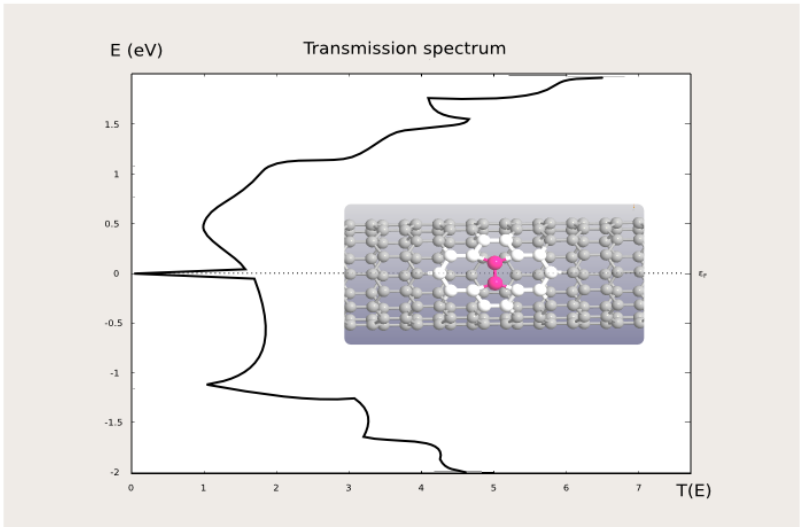
Transmission spectrum

Lets extract the transmission spectrum of the last device structure. This is a rather large device – the central region contains 360 atoms – but with the **ATK-SE** engine you can calculate the electron transmission spectrum in just a few minutes.

Use the Script Generator to create a Python script including a *Slater–Koster calculator* with the *Hancock.C ppPi* basis set and a *TransmissionSpectrum* analysis block. Then run the calculation using the Job Manager, and use the 2D Plot plugin to visualize the calculated transmission spectrum.

The resulting transmission spectrum is also available for download: `Device_stone.hdf5`.


Builder
Script Genarator
Job Manager
Viewer
Editor
Bandstructure
BlochState
TotalEnergy
TransmissionSpectrum
VibrationalMode
run in Job Manager
choose


E (eV)
Transmission spectrum
T(E)

The transmission spectrum is shown above. It was computed using the Hancock single-band tight-binding model with up to 3rd nearest neighbor interactions from Ref.[3]. A perfect (9,0) tube would have a step-like transmission spectrum around the Fermi level, but the defect introduces strong scattering and significantly modifies the electronic transmission. This will most certainly have a noticeable influence on the I-V curve.

Tip: You can easily calculate the corresponding I-V curve: Simply start a new Script Generator session and use **Analysis from File** and the **IVCurve** analysis object to create the script. Plot the results using the **IV-Plot** plugin.

You can also calculate the transmission spectrum and I-V curve for the defect free (9,0) nanotube and compare to the results found above.

References

[3] Y.ăHancock, K.ăSaloriutta, A.ăUppstu, A.ăHarju, and M.ăJ. Puska. Spin-Dependence in Asymmetric, V-Shaped-Notched Graphene Nanoribbons. *J. Low Temp. Phys.*, 153(5):393–398, 2008. doi:10.1007/s10909-008-9838-y.

CHAPTER

FOUR

MANUAL

4.1 General

4.1.1 Introduction

QuantumATK packages a powerful set of tools for calculating properties of nano-scale systems. These *atomic-scale calculators* are based on density functional theory, semi-empirical tight binding, and classical potentials. The non-equilibrium Greens function method is a unique feature of QuantumATK; it allows simulations on nano-scale devices and interfaces, including support for non-zero bias between the electrode leads and device components such as electrostatic gates and dielectrics. More information may be found at the QuantumATK website.

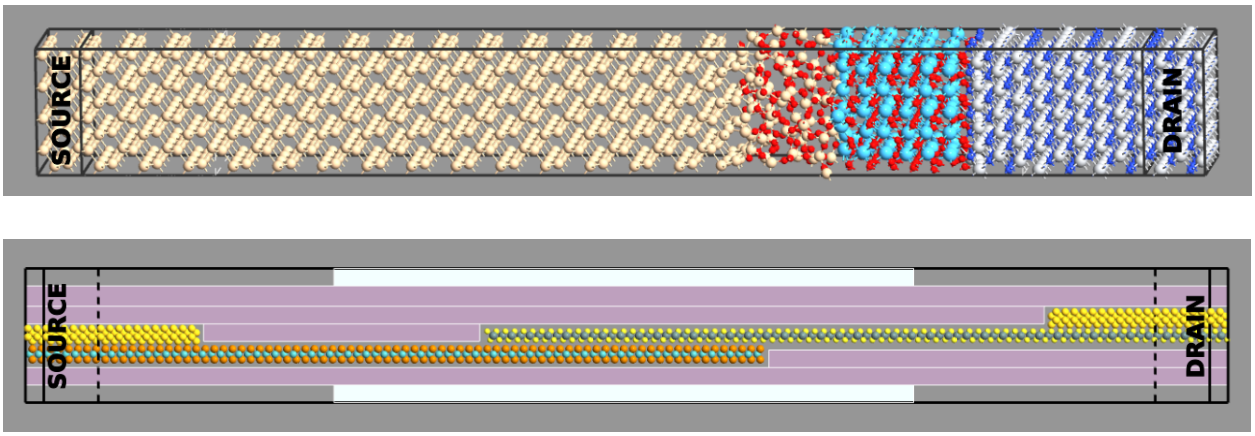


Fig. 4.1: Typical device systems; a high-k metal gate stack (top) and a field-effect transistor made of 2D materials (bottom), both with left and right semi-infinite electrode leads. One unique capability of QuantumATK is to simulate the electron transport through such two-probe systems. Gate electrodes may be included as non-current-carrying electrostatic gates (bright regions in the 2D-FET above), and regions of a particular dielectric constant may also be defined (pink regions in the 2D-FET above). One-probe systems with only a single electrode are also supported.

ATK is controlled using *ATK-Python*, which is an extension to the well-established Python scripting language. Setting up and executing QuantumATK calculations is therefore done in an ATK-Python script or directly from the command line in an interactive Python shell.

The main purpose of this manual is to document all QuantumATK functionality. The *QuantumATK Reference Manual* therefore gives a detailed summary of all input and output parameters, as well as notes on relevant theory and usage examples.

We also provide a thorough exposition of the theoretical background for the atomic-scale simulators implemented in QuantumATK (see *Atomic-Scale Calculators*), and a detailed introduction to Python scripting and using *ATK-Python* to control QuantumATK (see *Python in QuantumATK*).

For tutorials on how to use QuantumATK we refer to the Tutorials website.

New in QuantumATK X-2025.06

The X-2025.06 release of QuantumATK introduces a range of new features and performance improvements and an overview is given here: Release Overview. For a comprehensive list, see the full release notes, which can be found on the SolvNet portal.

Tip: For the full list of changes, see the release notes for the feature release, i.e. X-2025.06. The release notes for a service pack release will only list the changes relative to the feature release.

Installing and running the software

The QuantumATK binary installer is available for registered users at the SolvNet software portal. Access to a valid QuantumATK license is also needed. Detailed instructions are given in the Installation Guide. Note that a license is required and if your institution does not already have a license, you can request an evaluation license via the Synopsys Eval Portal.

When QuantumATK has been installed on your machine you can run it from the command line using the **atkpython** executable and a QuantumATK Python script. The executable should be in your `PATH` and is available here: `.../QuantumATK-VERSION/bin/atkpython`.

Parallel execution

ATK supports multi-level parallelism, using the Message Passing Interface (MPI) available on most supercomputing clusters for distributed memory parallelism, and OpenMP for shared memory parallelism. QuantumATK ships with Intel MPI, which you can find here: `.../QuantumATK-VERSION/mpi/bin/mpiexec.hydra`

How to read this manual

This manual is typeset using in-line references to QuantumATK Python objects and functions, and contains several script examples. The following style conventions are used:

- All QuantumATK objects and functions appear as links, e.g. *`MoleculeConfiguration`*. The link will take you to the relevant section of the *QuantumATK Reference Manual*, where a detailed description of the object MoleculeConfiguration is provided.
- References to particular chapters and sections are also links. For example, the links sect1.python.loops and sect1.python.tuples direct you to specific sections in the chapter *Python in QuantumATK*.
- Boldface letters are used to highlight specific words, e.g. **atkpython**, while in-line names of Python variables, functions, parameters, and methods are in general typeset using a monospace serif, e.g. `list_of_atoms` for a Python variable and `cartesianCoordinates()` for a method of the *`MoleculeConfiguration`* object.
- In-line names of files and directories are also typeset using a monospace serif, e.g. `file.txt` and `$HOME/QuantumATK/`.
- Python structures are visually enclosed in a box and typeset using a monospaced serif:

```
# This is a comment

for i in range(3):
    print(i)
```

Scripts can often be downloaded by following a link, e.g. `example_code.py`.

•Instructions for using the command line are indicated by the $ prefix:

```
$ atkpython script.py
```

while instructions for using an interactive Python session is indicated by the »> prefix:

```
>>> myList = [1, 2, 3, 4, 5]
>>> print(myList)
[1, 2, 3, 4, 5]
>>> print(len(myList))
5
```

4.2 Atomic-Scale Calculators

4.2.1 DFT: LCAO

Introduction

QuantumATK can model the electronic properties of closed and open quantum systems within the framework of **density functional theory (DFT)** using numerical LCAO basis sets (linear combination of atomic orbitals). For periodic systems, QuantumATK can also use the plane-wave basis set for DFT calculations, as discussed in *DFT: Plane Wave*.

The key parameter in the self-consistent calculation of the Kohn–Sham equations is the **density matrix**, which defines the electron density. For open systems, the density matrix is calculated using non-equilibrium Greens functions (NEGFs), see *NEGF: Device Calculators*, while for closed or periodic systems it is calculated by diagonalization of the Kohn–Sham Hamiltonian, as described in the present chapter. The electron density then sets up an effective potential, which is given by the Hartree, exchange-correlation, and external potentials. Knowing the effective potential allows us to obtain the Kohn–Sham Hamiltonian.

Information about provided pseudopotentials (SG15, PseudoDojo), pre-built LCAO basis sets sizes (medium, high, ultra) and their speed and accuracy against all-electron reference and DFT-PlaneWave calculations can be found in the QuantumATK reference paper by Smidstrup *et al.*[1]. The next section describes the mathematical formalism behind the DFT-LCAO model.

Background information

The *LCAOCalculator* provides a description of electronic structure using DFT and norm-conserving pseudopotentials. The method is based on an expansion of the single-particle wave functions in a basis of numerical atomic orbitals with compact support. In this section, we describe the mathematical formalism of the DFT-LCAO calculator, which closely follows the work of Soler *et al.*[2].

[1] S.ăSmidstrup, T.ăMarkussen, P.ăVancraeyveld, J.ăWellendorff, J.ăSchneider, T.ăGunst, B.ăVerstichel, D.ăStradi, P.ăA. Khomyakov, U.ăG. Vej-Hansen, M.-E. Lee, S.ăT. Chill, F.ăRasmussen, G.ăPenazzi, F.ăCorsetti, A.ăOjanperä, K.ăJensen, M.ăL.ăN. Palsgaard, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. QuantumATK: An integrated platform of electronic and atomic-scale modelling tools. *Journal of Physics: Condensed Matter*, 32(1):015901, Oct 2019. URL: https://doi.org/10.1088/1361-648X/ab4007 (visited on 2019-10-10), doi:10.1088/1361-648x/ab4007.

[2] J.ăM. Soler, E.ăArtacho, J.ăD. Gale, A.ăGarcía, J.ăJunquera, P.ăOrdejón, and D.ăSánchez-Portal. The SIESTA method for ab initio order-N materials simulation. *J. Phys.: Condensed Matter*, 14(11):2745, 2002. URL: http://stacks.iop.org/0953-8984/14/i=11/a=302.

Kohn–Sham Hamiltonian

Within density functional theory, the many-body electronic structure of the system is described in terms of the one-electron Kohn–Sham Hamiltonian:

$$\hat{H}_{\mathrm{1el}} = -\frac{\hbar^2}{2m}\nabla^2 + V^{\mathrm{eff}}[n](\mathbf{r}).$$

In this equation, the first term is the kinetic energy of the electron, while the second term (the effective potential) is the potential energy of the electron moving in the mean field created by the other electrons as well as in any external potential field, e.g. the electrostatic potential of ions or any other external field. The electrons are described in terms of the total electron density, $n = n(\mathbf{r})$.

The electron density is discussed in detail in the section *Electron density*, and the effective potential in the section *Effective potential*.

Solving the Kohn–Sham equations by means of a basis set expansion

We calculate the one-electron eigenfunctions of the Kohn–Sham Hamiltonian, ψ_α, by solving the one-electron Schrödinger equation,

$$\hat{H}_{\mathrm{1el}}\psi_\alpha(\mathbf{r}) = \varepsilon_\alpha\psi_\alpha(\mathbf{r}).$$

This differential equation is called the Kohn–Sham equation within DFT. To solve it, we expand the eigenfunctions $\psi_\alpha(\mathbf{r})$ in a set of basis functions, ϕ_i:

$$\psi_\alpha(\mathbf{r}) = \sum_i c_{\alpha i}\phi_i(\mathbf{r}).$$

This allows us to represent the differential equation as a matrix equation for determining the expansion coefficients, $c_{\alpha i}$:

$$\sum_j H_{ij}c_{\alpha j} = \varepsilon_\alpha \sum_j S_{ij}c_{\alpha j},$$

where the Hamiltonian matrix, $H_{ij} = \langle\phi_i|\hat{H}_{\mathrm{1el}}|\phi_j\rangle$, and overlap matrix $S_{ij} = \langle\phi_i|\phi_j\rangle$ are given by the multiple integrals with respect to the electron coordinates.

Electron density

The electron density of the many-electron system is given by the occupied eigenstates of the Kohn–Sham Hamiltonian:

$$n(\mathbf{r}) = \sum_\alpha f_\alpha|\psi_\alpha(\mathbf{r})|^2,$$

where f_α is the occupation of the level denoted by α. For finite temperature calculation the occupations are determined by the Fermi-Dirac distribution $f_\alpha = \frac{1}{1+e^{(\epsilon_\alpha-\epsilon_F)/kT}}$ but other smooth distributions may be introduced in order to help speed convergence (see *Occupation Methods*).

The electron density can then be expressed in terms of the density matrix:

$$n(\mathbf{r}) = \sum_{ij} D_{ij}\phi_i(\mathbf{r})\phi_j(\mathbf{r}),$$

where the density matrix is given by the basis set expansion coefficients $c_{\alpha i}$:

$$D_{ij} = \sum_\alpha f_\alpha c^*_{\alpha i}c_{\alpha j}.$$

Note: For open systems (*DeviceConfiguration*), the density matrix is calculated using non-equilibrium Greens functions, see *NEGF: Device Calculators*.

Electron difference density

It is often convenient to compare the electron density of the many-body system to a superposition of individual atom-based electron densities, $n^{\mathrm{atom}}(\mathbf{r}-\mathbf{R}_\mu)$, where $\mathbf{R}_\mu$ is the position of atom μ in the many-body system:

$$\Delta n(\mathbf{r}) = n(\mathbf{r}) - \sum_\mu n^{\mathrm{atom}}(\mathbf{r}-\mathbf{R}_\mu).$$

Δn is called the electron difference density, and it is calculated using the *ElectronDifferenceDensity* analysis object.

Effective potential

The effective potential, $V^{\mathrm{eff}}[n]$, has three contributions:

$$V^{\mathrm{eff}}[n] = V^H[n] + V^{\mathrm{xc}}[n] + V^{\mathrm{ext}}.$$

The first two terms are due to electron–electron interactions, which depend on the electron density. The first term, $V^H[n]$, is the Hartree potential due to the mean-field electrostatic interaction between the electrons, while the second term, $V^{\mathrm{xc}}[n]$, is the exchange-correlation potential, which arises from the quantum mechanical nature of the electrons.

The potential V^{ext} represents any other electrostatic fields in the system. It can be separated into two contributions; the electrostatic potential of ions (given by norm-conserving pseudopotentials) and external electrostatic fields (given by one or more external sources).

External potential

The external potential is given by the pseudopotentials and an external electrostatic field. Such a field may arise from the inclusion of metallic gates:

$$V^{\mathrm{ext}} = \sum_\mu V_\mu^{\mathrm{pseudo}} + V^{\mathrm{gate}}.$$

The pseudopotential has two contributions:

$$V^{\mathrm{pseudo}} = V^{\mathrm{local}} + \sum_{n,n'} |\chi_n\rangle B_{n,n'} \langle\chi_{n'}|,$$

where the first and second terms correspond to the local and nonlocal part of the pseudopotential, respectively.

The term V^{gate} is the electrostatic potential due to external gates, calculated with zero electron density. This term will be returned by the Analysis object `ExternalPotential` and is calculated using the `MultigridSolver`, `DirectSolver` and `ParallelConjugateGradientSolver` Poisson solvers. The Analysis object `ElectrostaticDifferencePotential` also includes this term, among others.

Hartree potential

Using the electron density, we can calculate the classical electrostatic potential, the so-called Hartree potential. The actual calculation of the Hartree potential is described in detail in the section *Poisson solvers*.

Exchange-correlation potential

In the DFT method, the quantum mechanical part of the electron–electron interaction is approximated by the exchange-correlation term, and a large number of different approximate exchange-correlation density functionals exist. ATK supports many of these, see the section *Exchange-correlation energy*.

The exchange-correlation potential is defined as the functional derivative of the exchange-correlation energy with respect to the electron density, which corresponds to a mean-field quantum mechanical interaction potential between the electrons:

$$V^{\mathrm{xc}}[n](\mathbf{r}) = \frac{\delta E^{\mathrm{XC}}}{\delta n}(\mathbf{r}).$$

Total energy and forces

The DFT total energy of a many-electron system is a functional of the electron density, n:

$$E[n] = T[n] + E^{\mathrm{xc}}[n] + E^{H}[n] + E^{\mathrm{ext}}[n],$$

where $T[n]$ is the kinetic energy of a non-interacting electron gas with density n, $E^{\mathrm{xc}}[n]$ the exchange-correlation energy, $E^{H}[n]$ the Hartree potential energy, and $E^{\mathrm{ext}}[n]$ the interaction energy of the electrons in the electrostatic field created by ions and other external sources.

The electron kinetic energy may be defined as

$$T[n] = \sum_{\alpha} f_{\alpha} \langle \psi_{\alpha} | \frac{-\hbar^2}{2m} \nabla^2 | \psi_{\alpha} \rangle.$$

The total energy is calculated using *TotalEnergy*.

First-principles forces are calculated by differentiating the total energy with respect to the ionic coordinates of atom i at position $\mathbf{R}_i$:

$$\mathbf{F}_i = -\frac{dE[n]}{d\mathbf{R}_i}.$$

Pseudopotentials

QuantumATK uses norm-conserving pseudopotentials and PAW potentials, and is shipped with a database for the entire periodic table. See the full list here: *Pseudopotentials*. This database is reviewed and updated on a regular basis to provide increasingly accurate and general-purpose pseudopotentials and PAW potentials.

QuantumATK uses the unified pseudopotential format (`upf`) defined by the Quantum ESPRESSO consortium. From their website, one may download tools to convert several different pseudopotential formats into a `upf` file.

LCAO basis set

The eigenfunctions of the Kohn–Sham Hamiltonian can be expanded in a Linear Combination of Atomic Orbitals (LCAOs):

$$\phi_{nlm}(\mathbf{r}) = R_{nl}(r)Y_{lm}(\hat{\mathbf{r}}),$$

where Y_{lm} are spherical harmonics, and R_{nl} are radial functions with compact support, being exactly zero outside a confinement radius.

Note: The basis set functions have a finite range, but the interaction range of the Hamiltonian is larger than that of the basis set. Theoretically, the Hamiltonian interaction range may exceed twice the basis set range, but is usually smaller in practice.

The basis orbitals have a number of parameters that determine the shape of the orbitals. It is possible to assemble the basis orbitals into your own basis set through the use of the *BasisSet* keyword.

QuantumATK comes with a number of pre-built basis sets for each chemical element and for each type of pseudopotential. See the full list here: *LCAO basis sets*

Exchange-correlation energy

The QuantumATK package use the libxc library for providing a large suite of exchange-correlation functionals. For each functional it is possible to add a Hubbard correction, as described in the section *XC+U mean-field Hubbard term*.

The main families of exchange-correlation functionals supported in QuantumATK are the *Local-density approximation (LDA)*, the *Generalized-gradient approximation (GGA)*, and the *Meta-GGA*. Each functional comes in four spin variants:

- non-polarized,
- spin-polarized (collinear),
- noncollinear,
- noncollinear with spin-orbit coupling.

Important: The choice of spin variant for the exchange-correlation functional determines the spin type used in all of the DFT-LCAO calculation.

Furthermore, a Hubbard correction can be added to all exchange-correlation variants, regardless of spin type, see *XC+U mean-field Hubbard term*. Several standard GGA functionals can also be extended with the *DFT-D2* and *DFT-D3* dispersion corrections by Grimme and co-workers.

Initial spin

Both collinear and noncollinear calculations may have local minima of the total energy that correspond to different spin configurations. It is therefore important to prepare the system in the correct initial spin configuration, such that the DFT self-consistent calculation will end up in the lowest energy spin configuration. This is done through the use of the methods *InitialSpin* and *RandomSpin*. The initial spin direction on each atom is specified in physical spherical coordinates (r, θ, ϕ), where θ is the angle with the z-axis, and ϕ the polar angle in the x-y plane relative to the x-axis. The collinear case is $\theta = 0$ Radians or $\theta = \pi$ Radians.

Local-density approximation (LDA)

In the LDA approximation, the exchange-correlation energy is taken as a functional of the local electron density,

$$E^{\mathrm{LDA}}[n] = \int n(\mathbf{r})\varepsilon^{\mathrm{LDA}}(n(\mathbf{r}))d\mathbf{r},$$

where $\varepsilon^{\mathrm{LDA}}(n(\mathbf{r}))$ is the exchange-correlation energy density of a homogeneous electron gas with density $n(\mathbf{r})$.

It is possible to derive an exact, analytical expression for the exchange energy of the homogeneous electron gas, the so-called Dirac–Bloch exchange energy, which is used for all the LDA functionals. The correlation energy of the electron gas cannot be calculated exactly, so a number of different approximations have been proposed over the years. Most of them give almost identical results; the most commonly used variant is the parametrization by Perdew and Zunger, which is also the default LDA functional in *DFT: LCAO*.

For a list of the parametrizations for the LDA correlation functional implemented in QuantumATK, see the documentation of the *ExchangeCorrelation* class. In particular, the section *Abbreviations* explains how to select a particular exchange-correlation functional. See also Table 4.7, Table 4.10, and Table 4.11.

Generalized-gradient approximation (GGA)

The GGA functionals is a large family of semi-local approximations for the exchange-correlation energy, where the functional depends on both the local value and the local gradient of the electron density,

$$E^{\mathrm{GGA}}[n] = \int n(\mathbf{r})\varepsilon^{\mathrm{GGA}}(n(\mathbf{r}), \nabla n(\mathbf{r}))d\mathbf{r}.$$

For a list of the GGA exchange-correlation functionals available in QuantumATK, see the documentation of the *ExchangeCorrelation* class. In particular, the section *Abbreviations* explains how to select a particular exchange-correlation functional. See also Table 4.8, Table 4.10, and Table 4.11.

Meta-GGA

The meta-GGA (MGGA) family of functionals extend the GGA approximation by additionally depending on one or both of the following quantities: the Laplacian of the density

$$\nabla^2 n(\mathbf{r})$$

and the so-called kinetic energy density

$$\frac{1}{2}\sum_{\alpha} f_\alpha \left|\nabla\psi_\alpha(\mathbf{r})\right|^2.$$

For a list of the MGGA exchange-correlation functionals available in QuantumATK, see the documentation of the *ExchangeCorrelation* class. In particular, the section *Abbreviations* explains how to select a particular exchange-correlation functional. See also Table 4.9, Table 4.10, and Table 4.11.

Different formulations of the MGGA can produce functionals with different uses. The SCAN functional has been found to give improved energetics over the LDA and GGA in several cases; however, band gaps are similar.

Conversely, the TB09 functional can provide a much more accurate description of band gaps than ordinary LDA and GGA. In fact, the accuracy of semiconductor band gaps obtained with TB09 are often comparable to those from calculations using GW or hybrid functionals, which have a significantly higher computational cost. It is even possible to tune the TB09 method such that it works optimally on both sides of an interface (see *ExchangeCorrelation*). Alternatively, one can use the *Hubbard U model* within the LDA or GGA with the U parameters for the semiconductor elements fitted against TB09 results.

However, TB09 is not meant for energetics, and, hence, geometry optimizations are disabled with this functional. For this, the GGA or other MGGA functionals should be used.

XC+U mean-field Hubbard term

The local approximations to the exchange-correlation energy have a number of shortcomings. Two of these are of particular interest:

- **Self-interaction:** The electron is formally allowed to interact with itself. This can prevent electrons from localizing properly.
- **Excited states:** The LDA and GGA description of conduction-band energy levels is often poor, so band gaps are often too low.

The mean field Hubbard correction by Dudarev *et al.*[3] and Cococcioni *et al.*[4], often denoted XC+U, DFT+U, LDA+U, or GGA+U, is a semi-empirical correction which attempts to improve on these deficiencies of the local exchange-correlation functionals by adding an extra term to the exchange-correlation functional:

$$E_U = \frac{1}{2}\sum_{\mu} U_\mu (n_\mu - n_\mu^2).$$

In this equation, n_μ is the projection onto an atomic shell and U_μ is the **Hubbard U** for that shell. The E_U energy term is zero for a fully occupied or unoccupied shell, while positive for a fractionally occupied shell.

The energy is thereby lowered if states become fully occupied or empty. This may happen if the energy levels move away from the Fermi Level, increasing the band gap, or if the broadening of the states is decreased, the electrons are then more localized. Thus, the Hubbard U method improves on the deficiencies of the local exchange-correlation functionals listed above.

The value of U is often used as an empirical parameter, which is varied in order to improve the comparison between DFT and experimental data. However, it is also possible to calculate the U using either linear response theory with the function *calculateFiniteDifferenceLinearResponseHubbardU* or using the local screened Coulomb correction (LSCC) approximation with the function *calculateLocalScreenedCoulombCorrectionHubbardU* .

Some caution must be taken when using the XC+U correction, since for large U values the electron density may have several local minima where some of them are unphysical, and it is often necessary to select an anisotropic initial electron state to reach the correct local minimum.

[3] S.ăL. Dudarev, G.ăA. Botton, S.ăY. Savrasov, C.ăJ. Humphreys, and A.ăP. Sutton. Electron-energy-loss spectra and the structural stability of nickel oxide: An LSDA+U study. *Phys. Rev. B*, 57:1505–1509, Jan 1998. doi:10.1103/PhysRevB.57.1505.

[4] M.ăCococcioni and S.ădeăGironcoli. Linear response approach to the calculation of the effective interaction parameters in the LDA+U method. *Phys. Rev. B*, 71:035105, Jan 2005. doi:10.1103/PhysRevB.71.035105.

XC+U implementations in QuantumATK

The XC+U implementation in QuantumATK comes in four main variants; a standard local-orbital formulation, so-called **Onsite** representation, and the **Dual** representation introduced by Han *et al.*[5]. The difference between the two implementations is in the definition of the local occupation matrix. The occupations may be summed over a single orbital or over an entire angular momentum shell, the latter corresponds to the **OnsiteShell** and **DualShell** representations.

Onsite representation

In the onsite representation, the local occupation matrix is given by

$$n^{\sigma}_{\mu,mm'} = D^{\sigma}_{\mu m,\mu m'},$$

where D is the density matrix, μ a basis orbital index, and σ a spin index.

OnsiteShell representation

In the onsite shell representation, the occupation is obtained by summing together all the basis functions in each angular momentum shell, l:

$$n^{\sigma}_{l,mm'} = \sum_{\mu \in l} n^{\sigma}_{\mu,mm'}.$$

Dual representation

In the dual representation, the local occupation matrix is given by

$$n^{\sigma}_{\mu,mm'} = \sum_{i,n} \left[S_{\mu m,in} D^{\sigma}_{in,\mu m'} + D^{\sigma}_{\mu m,in} S_{in,\mu m'} \right],$$

where S is the overlap matrix.

DualShell representation

Similar to the onsite shell representation, the dual shell occupation is obtained by summing up all dual occupations in each angular momentum shell, l:

$$n^{\sigma}_{l,mm'} = \sum_{\mu \in l} n^{\sigma}_{\mu,mm'}.$$

[5] M.ăJ. Han, T.ăOzaki, and J.ăYu. O(N) LDA+U electronic structure calculation method based on the nonorthogonal pseudoatomic orbital basis. *Phys. Rev. B*, 73:045110, Jan 2006. doi:10.1103/PhysRevB.73.045110.

Which model to use?

The onsite representation corresponds to projecting into a local orbital, and since the local orbitals are non-orthogonal, there are no sum rules for the occupation matrix and it is not guaranteed that a fully occupied shell has occupation 1.

In the dual representation, the occupation matrix has the form of a Mulliken population, and this form has the advantage that sum rules apply, i.e. the trace of the occupation matrix sums up to the total number of electrons.

The onsite shell and dual shell representations often give a better description with multiple-zeta basis sets compared to the onsite and dual representation, where the occupation is obtained by projecting into single basis functions.

Input format for XC+U

The Hubbard U approximation in the onsite representation may be selected through keywords of the types `LDAU.XCTYPE`, `LSDAU.XCTYPE`, `GGAU.XCTYP`, and `SGGAU.XCTYPE`.

The keyword `LSDAU.PZ` gives the default LSDA functional with the default Hubbard U method. The specification

```
exchange_correlation = LSDAU.PZ
```

is therefore identical to

```
exchange_correlation = ExchangeCorrelation(
    exchange=DiracBloch,
    correlation=PerdewZunger,
    hubbard_term=Onsite,
    number_of_spins=2,
    )
```

Use this form with the option `hubbard_term=Dual` to select the dual representation.

The value of the U term is specified through the basis set:

```
basis_set = [LDABasis.Nickel_SingleZeta(hubbard_u=[4.6, 0.0]*eV,
                                        filling_method=Anisotropic)]
```

which will give a single-zeta basis set for nickel, consisting of a 3d and a 4s orbital, where the 3d orbital has U=4.6 eV. The 3d orbitals are filled anisotropically, i.e, there is 1 electron in in orbitals with m=-2,-1,0,1, leaving m=2 empty. The alternative is `filling_method=SphericalSymmetric`, which puts 4/5 electron in each orbital.

DFT-1/2 method

The DFT-1/2 method (often also denoted LDA-1/2 or GGA-1/2) is a semi-empirical approach to correct the self-interaction error in local and semi-local exchange-correlation functionals for extended systems. As such, it can be viewed as an alternative to the *XC+U mean-field Hubbard term* with broadly the same aims: to improve the description of conduction-band energy levels and band gaps.

This method, introduced by Ferreira *et al.*[6], is based on the much older Slater half-occupation scheme for molecules[7] (also known as the transition state method). Slaters original method consists in carrying out a self-consistent calculation

[6] LuizăG. Ferreira, Marcelo Marques, and LaraăK. Teles. Approximation to density functional theory for the calculation of band gaps of semiconductors. *Phys. Rev. B*, 78:125116, Sep 2008. doi:10.1103/PhysRevB.78.125116.

[7] J.ăC. Slater and K.ăH. Johnson. Self-Consistent-Field $X\alpha \,$ Cluster Method for Polyatomic Molecules and Solids. *Phys. Rev. B*, 5:844–853, Feb 1972. doi:10.1103/PhysRevB.5.844.

with half an electron removed from the system, and taking the eigenvalue of the half-filled state as an estimate for the ionization energy. This approach was later formalized in the theorem by Janak[8]:

$$\frac{\partial E}{\partial f_\alpha} = \varepsilon_\alpha \left(f_\alpha \right),$$

where E is the total energy of the system, f_α is the occupation of state α (between 0 and 1), and ε_α is the eigenvalue of the state. The success of the half-occupation scheme is based on the fact that $\varepsilon_\alpha \left(f_\alpha \right)$ is known to be almost precisely linear for many cases, which makes the relationship with the ionization energy exact.

The DFT-1/2 method makes use of the theoretical insights from the half-occupation scheme and to tackle the fundamental problem of the self-interaction error for the case of extended systems with a band gap. The method attempts to correct for this by defining an atomic self-energy potential which cancels the electron-hole self-interaction energy. This potential is calculated for atomic sites in the system, and is defined as the difference between the potential of the neutral atom and that of a charged ion resulting from the removal of a fraction of its charge, between 0 and 1 electrons. The total self-energy potential is the sum of these atomic potentials.

The addition of the DFT-1/2 self-energy potential to the DFT Hamiltonian has been found to greatly improve band gaps for a wide range of semiconducting and insulating systems[9].

It is important to note that the method is not entirely free of empirical parameters. Much like the value of U for the Hubbard U method, there are two values which much be fixed for every species in the system: the fractional charge f_α removed from the neutral atom, and a cutoff radius r^{cut} beyond which to trim the atomic self-energy potential. The latter is needed to avoid excessive overlap of the potentials from different sites in the crystal.

The original authors of the method suggest to fix r^{cut} variationally, by choosing the value which maximizes the band gap[Page 1843, 69].

The choice of f_α is less well-defined: although a value of 0.5 is standard (hence the name of the method), this is known not to be the optimal choice for various materials (e.g., silicon). It may therefore be appropriate to treat it as an empirical parameter, to be varied by comparison with experiment.

It is also important to note that not all species in the system necessarily require the DFT-1/2 correction; it is generally advisable only to add this to the anionic species, and leave the cationic species as normal[Page 1843, 69].

Shell-DFT-1/2

Following the paper by Xue et al.[10] QuantumATK also supports calculations with the slightly modified shell-DFT-1/2 method. The difference between normal DFT-1/2 and shell-DFT-1/2 is that in the latter, the self-energy potential is added in a shell around the atom, and not in a sphere as in normal DFT-1/2. The shell is specified by an additional inner radius parameter in addition to the cutoff radius r^{cut}.

[8] J.ăF. Janak. Proof that $\partial E/\partial n_i=\varepsilon \,$ in density-functional theory. *Phys. Rev. B*, 18:7165–7168, Dec 1978. doi:10.1103/PhysRevB.18.7165.

[9] LuizăG. Ferreira, Marcelo Marques, and LaraăK. Teles. Slater half-occupation technique revisited: The LDA-1/2 and GGA-1/2 approaches for atomic ionization energies and band gaps in semiconductors. *AIP Adv.*, 1(3):032119, 2011. doi:10.1063/1.3624562.

[10] Kan-Hao Xue, Jun-Hui Yuan, LeonardoăR.C. Fonseca, and Xiang-Shui Miao. Improved LDA-1/2 method for band structure calculations in covalent semiconductors. *Computational Materials Science*, 153:493–505, 2018. URL: https://www.sciencedirect.com/science/article/pii/S0927025618304166, doi:https://doi.org/10.1016/j.commatsci.2018.06.036.

Input format for DFT-1/2

The DFT-1/2 method may be selected by specifying one of the pre-defined exchange-correlation functionals: `LDAHalf`, `LSDAHalf`, `NCLDAHalf`, `SOLDAHalf`, `GGAHalf`, `SGGAHalf`, `NCGGAHalf`, `SOGGAHalf`.

The keyword `LDAHalf.PZ` gives the default LDA functional with the DFT-1/2 correction. The specification

```
exchange_correlation = LDAHalf.PZ
```

is therefore identical to

```
exchange_correlation = ExchangeCorrelation(
    exchange=DiracBloch,
    correlation=PerdewZunger,
    dft_half_enabled=True
    )
```

In general, the keyword `dft_half_enabled=True` can be specified for any user-defined exchange-correlation functional.

The value of the DFT-1/2 parameters are specified through the basis set:

```
dft_half_parameters = DFTHalfParameters(
    element=Arsenic,
    fractional_charge=[0.3, 0.0],
    cutoff_radius=4.0*Bohr,
    inner_radius=0.0*Bohr)

basis_set = [
    LDABasis.Arsenic_DoubleZetaPolarized(
        dft_half_parameters=dft_half_parameters),
    LDABasis.Gallium_DoubleZetaPolarized(
        dft_half_parameters=Disabled)
]
```

The above example shows a typical case for gallium arsenide, in which the anion (As) is given a DFT-1/2 self-energy potential calculated with $f_\alpha = 0.3$ and $r^{\mathrm{cut}} = 4\ a_0$, while the cation (Ga) has its DFT-1/2 correction turned off with the keyword `Disabled`.

If `dft_half_parameters` is not specified for a species, the default `Automatic` is used. This will set the *DFT-1/2 default parameters* from the table of optimized values given below.

Warning: When the DFT-1/2 method is enabled, the self-energy potential is included in the calculation of the total energy, forces and stress. However, the method is only intended as a scheme for calculating band structures, not for structural relaxations. We recommend that structural properties be calculated before applying the DFT-1/2 correction.

DFT-1/2 default parameters

The default DFT-1/2 parameters used by the `Automatic` keyword have been optimized against a wide range of materials, and should improve upon the standard DFT band gap in most cases. The complete list is given in the table below. As explained above, we only include elements which tend to have anionic character in compounds of interest.

Currently, only Ge (GGA) has an `inner_radius` different from zero, meaning that this is the only element using the shell-DFT-1/2 method as default.

Important: All elements not present in the table default to `Disabled`, i.e., no DFT-1/2 correction.

Important: For meta-GGA exchange-correlation functionals, **all** elements default to `Disabled`.

Table 4.1: Optimized DFT-1/2 parameters.

Element	LDA functionals			GGA functionals		
	fractional_char	cutoff_radius | inner_radius		fractional_char	cutoff_radiı	inner_radius
C	0.3	2.5 Bohr	0.0 Bohr	0.4	2.5 Bohr	0.0 Bohr
N	0.4	3.0 Bohr	0.0 Bohr	0.4	3.0 Bohr	0.0 Bohr
O	0.5	2.5 Bohr	0.0 Bohr	0.4	3.0 Bohr	0.0 Bohr
F	0.7	3.0 Bohr	0.0 Bohr	0.6	2.5 Bohr	0.0 Bohr
Si	0.2	4.0 Bohr	0.0 Bohr	0.2	4.0 Bohr	0.0 Bohr
P	0.3	4.0 Bohr	0.0 Bohr	0.3	3.5 Bohr	0.0 Bohr
S	0.6	3.5 Bohr	0.0 Bohr	0.4	3.5 Bohr	0.0 Bohr
Cl	0.7	3.5 Bohr	0.0 Bohr	0.7	3.5 Bohr	0.0 Bohr
Zn	0.5	2.0 Bohr	0.0 Bohr	0.5	2.0 Bohr	0.0 Bohr
Ge	0.5	4.0 Bohr	0.0 Bohr	0.25	3.4 Bohr	2.35 Bohr
As	0.3	4.0 Bohr	0.0 Bohr	0.3	4.0 Bohr	0.0 Bohr
Se	0.5	3.5 Bohr	0.0 Bohr	0.5	3.5 Bohr	0.0 Bohr
Br	0.7	4.0 Bohr	0.0 Bohr	0.7	4.0 Bohr	0.0 Bohr
Sn	0.7	5.0 Bohr	0.0 Bohr	0.9	4.5 Bohr	0.0 Bohr
Sb	0.3	4.5 Bohr	0.0 Bohr	0.3	4.5 Bohr	0.0 Bohr
Te	0.4	4.0 Bohr	0.0 Bohr	0.4	4.0 Bohr	0.0 Bohr
I	0.6	4.5 Bohr	0.0 Bohr	0.6	4.0 Bohr	0.0 Bohr

Spin-orbit coupling (SOC)

Background

The spin-orbit coupling can lead to energy level splits of bandstructures and molecular energy spectra. As a rule of thumb, the magnitude of SOC scales as Z^4/n^3l^2, where Z, n, and l are the atomic charge, principle, and angular quantum number of the considered energy level, respectively. The SOC is fundamentally a relativistic effect, and can be described by a second-order expansion of the relativistic Hamiltonian in terms of the fine structure constant α[11].

In *DFT: LCAO*, the SOC is taken into account via the *NormConservingPseudoPotential*. For example, the **SG15 pseudopotentials**, shipped with QuantumATK as of the 2016 release, come in two versions for each element; a standard

[11] L.ăFernandez-Seivane, M.ăA. Oliveira, S.ăSanvito, and J.ăFerrer. On-site approximation for spin–orbit coupling in linear combination of atomic orbitals density functional methods. *J. Phys.: Condensed Matter*, 18(34):7999, 2006. URL: http://stacks.iop.org/0953-8984/18/i=34/a=012.

scalar-relativistic pseudopotential (denoted SG15), and one that is generated by mapping the solution to the Dirac equation, which naturally includes the SOC, to a scalar-relativistic pseudopotential (denoted SG15-SO):

$$V_{ps} = V_L + V_{NL}^{+1/2} + V_{NL}^{-1/2},$$

with a local contribution V_L and non-local contributions from total angular momenta $j = l + 1/2$ and $j = l - 1/2$.

Each non-local term is expanded in spin-orbit projector functions $P_{\alpha\beta}^{l\pm1/2,\xi}$,

$$V_{NL}^{\pm1/2} = \sum_{l,\xi,\alpha,\beta} \nu_{l\pm1/2,\xi} P_{\alpha\beta}^{l\pm1/2,\xi},$$

where $\nu_{l\pm1/2,\xi}$ are normalization constants. The indices α, β denote the possible spin orientations (up, down). Each nonlocal pseudopotential term requires four projector functions per expansion order ξ.

Tip: Use SG15-SO or OpenMX pseudopotentials for *DFT: LCAO* calculations with spin-orbit.

Usage

The SOC terms are enabled by choosing a pseudopotential containing spin-orbit terms (SG15-SO or OpenMX), and appropriate settings for the *ExchangeCorrelation* keyword. For example, an QuantumATK Python script for silicon with SOC could contain the following specification for the calculator:

```
#------------------------------------------
# Basis Set
#------------------------------------------
basis_set = [
    BasisGGASG15SO.Silicon_Medium,
    ]
#------------------------------------------
# Exchange-Correlation
#------------------------------------------
exchange_correlation = SOGGA.PBE
#------------------------------------------
# Calculator
#------------------------------------------
k_point_sampling = MonkhorstPackGrid(na=9,nb=9,nc=9)
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=100.0*Hartree,
    )
calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
```

Note the setting `exchange_correlation=SOGGA.PBE`, which switches on the SOC terms in the pseudopotential. Omitting this keyword or setting it to a non-SOC method, e.g. `NCGGA.PBE`, would still result in a calculation with the SG15-SO pseudopotential, but the SOC terms would be disregarded.

Hybrid functionals in the LCAO framework

Hybrid functionals are DFT functionals where part (or all) of the exchange in the exchange correlation functional is replaced by the exact exchange contribution. The exact exchange contribution in an LCAO basis set is defined as:

$$X_{ij} = \sum_{kl} V_{ik;jl} D_{kl} \ ,$$

in which D is the density matrix, and V are the 4-index coulomb integrals:

$$V_{ik;jl} = \int \int \frac{\phi_i^*(\mathbf{r})\phi_k(\mathbf{r})\phi_l^*(\mathbf{r}')\phi_j(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r} d\mathbf{r}' \ .$$

In our LCAO implementation we use numerical orbitals, which makes the above integral very complicated to evaluate. To improve performance of the integral evaluations we introduce the resolution of identity (RI) approach[12] which writes a product of two orbitals as:

$$\phi_i^*(\mathbf{r})\phi_k(\mathbf{r}) = \sum_{\mu} C_{ik}^{\mu} P_{\mu}(\mathbf{r})$$

where we have defined the auxiliary basis functions P_μ and the auxiliary basis coefficients C_{ik}^{μ}. In the general RI approximation, the function P_μ in the above expansion can be located around any of the other atoms in the system. We will limit ourselves to the case where μ lies either on the atom I or K of the original orbitals. This approximation is called the PARI approximation[13], and is the reason for the linear computational scaling with system size.

The accuracy of the numerical evaluation of the integrals can be set by the user using the parameter class `ExactExchangeParameters`.

Dielectric dependent Hybrid functionals.

In HSE06 we mix 0.75 of screened PBE exchange with 0.25 of screened Fock exact exchange. The value of this optimal exchange fraction to use can be shown to be inversely proportionate to the dielectric constant, ϵ, in the material. This makes sense if we look at the two limiting cases. If we have a metal we have perfect screening and an infinite ϵ, so the exchange interaction is screened away fully and you dont want to use Fock exchange. In the vacuum, $\epsilon = 1$, there is no screening and the correct exchange fraction is 1. The factor of 0.25 was chosen to work well for semi-conductors such as Silicon or Germanium, but is not optimal for materials with comparatively large and small band gaps.

To improve on this, various dielectric dependent hybrid functionals (DDH) have been invented. We have chosen to follow the approach suggested in[14]. In this approach they suggest to calculate an estimator for the dielectric constant:

$$\bar{g} = \frac{1}{V} \int d\mathbf{r} \sqrt{\frac{\nabla\rho(\mathbf{r})}{\rho(\mathbf{r})}} \ .$$

It is then assumed that the exchange fraction depends on this property through a fourth order polynomial:

$$\alpha = a_0 + a_1\bar{g} + a_2\bar{g}^2 + a_3\bar{g}^3 + a_4\bar{g}^4 \ .$$

We have fitted these parameters so they give the best possible match to the experimental band gap for a number of semi-conducting and insulating materials. You can choose this functional by setting the exchange correlation:

[12] Xinguo Ren, Patrick Rinke, Volker Blum, Jürgen Wieferink, Alexandre Tkatchenko, Andrea Sanfilippo, Karsten Reuter, and Matthias Scheffler. Resolution-of-identity approach to Hartree–Fock, hybrid density functionals, RPA, MP2 andGWwith numeric atom-centered orbital basis functions. *New Journal of Physics*, 14(5):053020, may 2012. URL: https://doi.org/10.1088/1367-2630/14/5/053020, doi:10.1088/1367-2630/14/5/053020.

[13] SergeyăV. Levchenko, Xinguo Ren, Jürgen Wieferink, Rainer Johanni, Patrick Rinke, Volker Blum, and Matthias Scheffler. Hybrid functionals for large periodic systems in an all-electron, numeric atom-centered basis framework. *Computer Physics Communications*, 192:60–69, 2015. URL: https://www.sciencedirect.com/science/article/pii/S001046551500079X, doi:https://doi.org/10.1016/j.cpc.2015.02.021.

[14] JonathanăH. Skone, Marco Govoni, and Giulia Galli. Nonempirical range-separated hybrid functionals for solids and molecules. *Phys. Rev. B*, 93:235106, Jun 2016. URL: https://link.aps.org/doi/10.1103/PhysRevB.93.235106, doi:10.1103/PhysRevB.93.235106.

```
exchange_correlation = HybridGGA.HSE06DDH
```

If you want to choose a custom parameterization, fitted to a specific set you are interested in, you can do that by manually assigning the fitting parameters:

```
hse06_ddh_parameters = (0.16957336, 0.0, 0.0, 0.0, 0.08940146)
exchange_correlation = HybridGGA.HSE06DDH(ddh_parameters=ddh_parameters)
```

This functional assumes that the whole system has the same exchange fraction, so it is meaningful only when used for homogeneous bulk systems.

For interface system consisting of two or more different materials with different properties, a different approach is needed. To this end the Local DDH functional has been invented[15]. In this approach a local estimator is defined:

$$\bar{g}(\mathbf{r},\sigma) = \frac{1}{(2\pi\sigma)^{\frac{3}{2}}} \int d\mathbf{r}' \sqrt{\frac{\nabla\rho(\mathbf{r}')}{\rho(\mathbf{r}')}} \exp\left(-\frac{|\mathbf{r}-\mathbf{r}'|^2}{2\sigma}\right) ,$$

from which an exchange fraction field is defined as:

$$a(\mathbf{r},\sigma) = a_0 + a_1\bar{g}(\mathbf{r},\sigma) + a_2\bar{g}(\mathbf{r},\sigma)^2 + a_3\bar{g}(\mathbf{r},\sigma)^3 + a_4\bar{g}(\mathbf{r},\sigma)^4.$$

This is then used to modify the 4-center coulomb integrals:

$$V_{ik;jl} = \int\int \alpha(\mathbf{r},\mathbf{r}';\sigma) \frac{\phi_i^*(\mathbf{r})\phi_k(\mathbf{r})\phi_l^*(\mathbf{r}')\phi_j(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}d\mathbf{r}' ,$$

with

$$\alpha(\mathbf{r},\mathbf{r}';\sigma) = \sqrt{a(\mathbf{r},\sigma)a(\mathbf{r}',\sigma)} .$$

Using this functional we can get the best results for the local density of states for interfaces, where in a $Si - SiO_2$ interface a 0.25 exchange fraction will be found for the Si part and a 0.35 for the SiO_2 part.

To use this functional you have to set the exchange correlation:

```
exchange_correlation = HybridGGA.HSE06LocalDDH
```

and once again you can set a custom parameterization by:

```
hse06_ddh_parameters = (0.16957336, 0.0, 0.0, 0.0, 0.08940146)
exchange_correlation = HybridGGA.HSE06LocalDDH(ddh_parameters=ddh_parameters)
```

A caveat for this method is that in the presence of vacuum the estimator blows up, and this has a negative influence on the convergence of the method. So we would recommend only using this functional for 3D systems.

The estimator also has difficulties with metallic systems, for which it should set the exchange fraction to zero. To improve on this we have implemented our own metallic estimator, which identifies metallic regions and sets the exchange fractions to zero in them. To use it just set the exchange correlation to:

```
exchange_correlation = HybridGGA.HSE06MetallicLocalDDH
```

When using this in a metal-semiconductor interface, it is as if you are using PBE for the metallic part and HSE06LocalDDH for the semi-conducting part.

[15] Pedro Borlido, Miguel A.ăL. Marques, and Silvana Botti. Local Hybrid Density Functional for Interfaces. *Journal of Chemical Theory and Computation*, 14(2):939–947, 2018. PMID: 29227686. URL: https://doi.org/10.1021/acs.jctc.7b00853, arXiv:https://doi.org/10.1021/acs.jctc.7b00853, doi:10.1021/acs.jctc.7b00853.

The Auxiliary Density Matrix Method (ADMM)

Using the PARI approximation, the calculation of the exchange matrix scales linearly with system size. There is one problem, however, and that is that the scaling with basis set size is polynomial with $\mathcal{O}(n^6)$. This means that if we need large basis sets to describe a system, the exchange calculation will become prohibitively expensive.

To overcome this issue the auxiliary density matrix method (ADMM) was developed[16], in which you run a DFT calculation with a large basis set, and use a smaller subset to compute the exchange properties.

The idea is simple, we project the density matrix of the system with the big basis set D down to a smaller basis set, and obtain the ADMM density matrix d. It is then postulated that a good approximation to the exchange matrix is:

$$X_{ij}(D) \approx x_{ij}(d) + \langle\phi_i|V_x(D)|\phi_j\rangle - \langle\phi_i|V_x(d)|\phi_j\rangle \ ,$$

in which the latter two terms are the exchange potentials generated by the full density and the ADMM density.

Up and down projections to and from the ADMM basis are done during the diagonalization procedure. This leads to some overhead in the diagonalization procedure. In return we get a significant speedup in the exchange matrix calculation.

The accuracy of the results depends on the ADMM basis set you choose. For the PseudoDojo basis sets we have constructed custom made ADMM basis sets, which are tuned to give accurate results for band gaps and structural properties (Δ-value) for a wide range of semi-conductors and insulators. In particular the PseudoDojo ADMM basis sets have been fitted to work best in combination with PseudoDojo Medium as main basis set, while for larger PseudoDojo basis sets we suggest to follow the hierarchy. For the FHI basis sets, you have a natural hierarchy you can choose from. For the other basis sets we dont recommend using ADMM.

To use it, you need to set:

```
exact_exchange_parameters = ExactExchangeParameters(
    use_admm=True,
    basis_set_admm=BasisGGAPseudoDojo.ADMM)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    exact_exchange_parameters=exact_exchange_parameters
    )

lcao_calculator = LCAOCalculator(
    exchange_correlation=HybridGGA.HSE06,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
```

4.2.2 DFT: Plane Wave

Introduction

QuantumATK can model the electronic properties of periodic quantum systems within the framework of **density functional theory (DFT)** using a plane-wave (PW) basis set. For closed and open systems, QuantumATK can also use the DFT-LCAO calculator, as discussed in *DFT: LCAO*. The *DFT: Plane Wave* calculator is not suitable for open systems, because it assumes periodic boundary conditions when solving the Kohn–Sham equations.

[16] Manuel Guidon, Jürg Hutter, and Joost VandeVondele. Auxiliary Density Matrix Methods for Hartree-Fock Exchange Calculations. *Journal of Chemical Theory and Computation*, 6(8):2348–2364, 2010. PMID: 26613491. URL: https://doi.org/10.1021/ct1002225, arXiv:https://doi.org/10.1021/ct1002225, doi:10.1021/ct1002225.

The key parameter in the self-consistent DFT-PW calculation of the Kohn–Sham equations is the **electron density**. The electron density then sets up an effective potential, which is given by the Hartree, exchange-correlation, and external potential. Knowing the effective potential allows us to obtain the Kohn–Sham Hamiltonian.

The next section describes the similarities and differences of the *PlaneWaveCalculator* and *LCAOCalculator*.

Background information

The *PlaneWaveCalculator* provides a description of electronic structure using *DFT* in combination with norm-conserving *pseudopotentials* or PAW potentials[1]. This method is based on an expansion of the single-particle wave functions in a basis of plane-wave functions to *solve the Kohn–Sham equations*, instead of the *LCAO basis set*. The description for the mathematical formalism of the DFT-PW calculator and PAW potentials can be found in the QuantumATK reference paper by Smidstrup *et al.*[2] and the work by Blöchl[Page 1851, 1], respectively.

Similarly to the *DFT: LCAO* calculator, the *DFT: Plane Wave* calculator allows for calculating basic physical quantities:

- *Electron density*
- *Effective potential*
- *Hartree potential*
- *Exchange-correlation potentials*
- *Pseudopotentials*
- *Total energy, forces and stress*
- *Exchange-correlation energy*
- *Spin-orbit coupling (SOC)*

Unlike the *DFT: LCAO* calculator, the *DFT: Plane Wave* calculator does not support the *XC+U mean-field Hubbard term* and *DFT-1/2 method* yet, as well as some of QuantumATK analysis objects, which are available for the DFT-LCAO calculator.

In addition to norm-conserving *Pseudopotentials*, the *DFT: Plane Wave* calculator supports *PAW potentials*.

4.2.3 Semi Empirical

Introduction

SemiEmpirical (SE) calculator can model the electronic properties of molecules, crystals and devices using both self-consistent and non-self-consistent **tight-binding** models. In this chapter, the implemented tight-binding models based on the **Slater–Koster** model and the **extended Hückel** model are presented.

The **Slater–Koster tight-binding model** follows closely the DFTB formalism described in[1], and it is recommended that this paper, and[2], are cited in publications using the *SemiEmpiricalCalculator* and *DeviceSemiEmpiricalCalculator* with the *SlaterKosterHamiltonianParametrization* in QuantumATK.

[1] P.ăE. Blochl. Projector augmented-wave method. *Phys. Rev. B*, 50:17953–17979, Dec 1994. URL: https://link.aps.org/doi/10.1103/PhysRevB.50.17953, doi:10.1103/PhysRevB.50.17953.

[2] S.ăSmidstrup, T.ăMarkussen, P.ăVancraeyveld, J.ăWellendorff, J.ăSchneider, T.ăGunst, B.ăVerstichel, D.ăStradi, P.ăA. Khomyakov, U.ăG. Vej-Hansen, M.-E. Lee, S.ăT. Chill, F.ăRasmussen, G.ăPenazzi, F.ăCorsetti, A.ăOjanperä, K.ăJensen, M.ăL.ăN. Palsgaard, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. QuantumATK: An integrated platform of electronic and atomic-scale modelling tools. *Journal of Physics: Condensed Matter*, 32(1):015901, Oct 2019. URL: https://doi.org/10.1088/1361-648X/ab4007 (visited on 2019-10-10), doi:10.1088/1361-648x/ab4007.

[1] M.ăElstner, D.ăPorezag, G.ăJungnickel, J.ăElsner, M.ăHaugk, Th. Frauenheim, S.ăSuhai, and G.ăSeifert. Self-consistent-charge density-functional tight-binding method for simulations of complex materials properties. *Phys. Rev. B*, 58:7260–7268, Sep 1998. doi:10.1103/PhysRevB.58.7260.

[2] K.ăStokbro, D.ăE. Petersen, S.ăSmidstrup, A.ăBlom, M.ăIpsen, and K.ăKaasbjerg. Semiempirical model for nanoscale device simulations. *Phys. Rev. B*, 82:075420, Aug 2010. doi:10.1103/PhysRevB.82.075420.

The **extended Hückel model** in SE is described in[2], and it is recommended that this paper is cited in publications using the *SemiEmpiricalCalculator* and *DeviceSemiEmpiricalCalculator* with the *HuckelHamiltonianParametrization* in QuantumATK.

In SE, the non-self-consistent part of the tight-binding Hamiltonian is parametrized using a two-center approximation, i.e. the matrix elements only depend on the distance between two atoms and is independent of the position of the other atoms. In the extended Hückel model, the matrix elements are described in terms of overlaps between Slater orbitals on each site. In this way, the matrix elements can be defined by very few parameters. In the Slater–Koster model, the distance-dependence of the matrix elements is given as a numerical function; this gives higher flexibility but also makes the fitting procedure more difficult.

The self-consistent part of the calculation is identical for both SE models. The density matrix is calculated from the Hamiltonian using non-equilibrium Greens functions for device systems, while for molecules and crystals it is calculated by diagonalization. The density matrix defines the real-space electron density and consequently the Hartree potential can be obtained by solving the Poisson equation. The following describes the details of the mathematical formalism behind the implementation.

For a list of the built-in parameter sets in SE, see *Built-in semiempirical parameter sets in QuantumATK*. In addition to these, it is possible to directly use parameter sets downloaded from the DFTB website or define your own Slater–Koster table or extended Hückel basis set in the QuantumATK input scripts.

Background information

Non-self-consistent Hamiltonian

The Hamiltonian is expanded in a basis of local atomic orbitals (an LCAO expansion),

$$\phi_{nlm}(\mathbf{r}) = R_{nl}(r)Y_{lm}(\hat{\mathbf{r}}),$$

where Y_{lm} is a spherical harmonic and R_{nl} is a radial function. Typically, the atomic orbitals used in the LCAO expansion have a close resemblance to the atomic eigenfunctions.

Onsite terms

With this form of the basis set, the onsite elements are given by

$$S_{ij}^{\text{onsite}} = \delta_{ij},$$

$$H_{ij}^{\text{onsite}} = E_i\delta_{ij},$$

where E_i is an adjustable parameter that is often close to the atomic eigenenergy.

Offsite terms in the extended Hückel model

The central object in the extended Hückel model is the overlap matrix:

$$S_{ij} = \int_V \phi_i(\mathbf{r} - \mathbf{R}_i)\phi_j(\mathbf{r} - \mathbf{R}_j)\ \mathrm{d}\mathbf{r}.$$

To calculate this integral, the form of the basis functions must be specified. In the extended Hückel model, the basis functions are parametrized by Slater orbitals,

$$R_{nl}(r) = \frac{r^{n-1}}{\sqrt{(2n)!}}\left[C_1(2\eta_1)^{n+\frac{1}{2}}e^{-\eta_1 r} + C_2(2\eta_2)^{n+\frac{1}{2}}e^{-\eta_2 r}\right].$$

The LCAO basis is described by the adjustable parameters η_1, η_2, C_1, and C_2. These parameters must be defined for each angular shell of valence orbitals for each element.

The overlap matrix defines the Hamiltonian,

$$H_{ij} = \frac{1}{4}(\beta_i + \beta_j)(E_i + E_j)S_{ij},$$

where E_i is the onsite orbital energy and β_i is a Hückel fitting parameter (often chosen to be 1.75).

Weighting schemes

The are two variants of the weighting schemes for the orbital energies of the offsite Hamiltonian. The scheme used above,

$$\frac{1}{2}\beta(E_i + E_j)S_{ij},$$

where $\beta = \frac{1}{2}(\beta_i + \beta_j)$ is used by Wolfsberg[3], while Hoffmann ([4] and[5]) uses

$$\frac{1}{2}(\beta + \alpha^2 + (1-\beta)\alpha^4)(E_i + E_j),$$

where $\alpha = (E_i - E_j)/(E_i + E_j)$.

Both variants are available in *Semi Empirical* through the parameters `WolfsbergWeighting` and `HoffmannWeighting`, which can be given to the *HuckelHamiltonianParametrization* class.

Offsite Hamiltonian in the Slater–Koster model

The overlap matrix is given by pairwise integrals between the different basis functions. These integrals can be pre-calculated for all relevant distances and different orbital combinations, and stored in so-called Slater–Koster tables. The Slater–Koster table stores the distance-dependent parameters $s(d, Z_1, Z_2, l_1, l_2, m)$, where d is the distance, Z_1, Z_2 the element types, l_1, l_2 the angular momentum of the two orbitals, and the index $m \leq \min(l_1, l_2)$.

From the Slater–Koster tables, the overlap matrix elements are given by

$$S_{ij} = \sum_{m \leq \min(l_i, l_j)} \alpha_{l_i, m_i, l_j, m_j, m}(\hat{R}_{ij}) s(d_{ij}, Z_i, Z_j, l_i, l_j, m),$$

where α are the Slater–Koster expansion coefficients.

In the Slater–Koster, model it is assumed that also the Hamiltonian has a pairwise form and a Slater–Koster table is generated for the Hamiltonian matrix elements. This table may be generated by calculating Hamiltonian matrix elements for a set of dimer distances or by simply fitting matrix elements to the band structure for different lattice constants.

In *Semi Empirical*, the Slater–Koster table is constructed either by providing the path to a directory containing compatible Slater–Koster files (see *DFTBDirectory* and *HotbitDirectory*), or directly using the *SlaterKosterTable* class.

Note that the extended Hückel model is a Slater–Koster model too, but with a special fitting procedure for the Hamiltonian matrix elements.

[3] M.ăWolfsberg and L.ăHelmholz. The Spectra and Electronic Structure of the Tetrahedral Ions MnO4-, CrO4-, and ClO4-. *J. Chem. Phys.*, 1952. doi:10.1063/1.1700580.

[4] M.-H. Whangbo and R.ăHoffmann. Counterintuitive orbital mixing. *J. Chem. Phys.*, 1978. doi:10.1063/1.435677.

[5] J.ăH. Ammeter, H.ăB. Buergi, J.ăC. Thibeault, and R.ăHoffmann. Counterintuitive orbital mixing in semiempirical and ab initio molecular orbital calculations. *J. Am. Chem. Soc.*, 100(12):3686–3692, 1978. doi:10.1021/ja00480a005.

Self-consistent Hamiltonian

In the self-consistent semi-empirical models in QuantumATK, the electron density is computed using the tight-binding model as described above. The density gives rise to a Hartree potential V_H. The calculation of the Hartree potential is described in detail in the section on *Poisson solvers*.

The Hartree potential is included through an additional term in the Hamiltonian:

$$H_{ij}^{\mathrm{SCF}} = \frac{1}{2}(V_H(\mathbf{R}_i) + V_H(\mathbf{R}_j))S_{ij}.$$

Electron density

The electron density is given by the occupied eigenfunctions,

$$n(\mathbf{r}) = \sum_{\alpha} f_\alpha |\psi_\alpha(\mathbf{r})|^2,$$

where f_α is the occupation of the level denoted by α. For finite temperature calculation the occupations are determined by the Fermi-Dirac distribution $f_\alpha = \frac{1}{1+e^{(\epsilon_\alpha - \epsilon_F)/kT}}$ with ϵ_α being the energy of the eigenstate ψ_α, ϵ_F the Fermi level and T the electron temperature. However, other smooth distributions may be introduced in order to help speed convergence (see *Occupation Methods*).

The eigenstates in the Slater orbital basis can be written as

$$\psi_\alpha = \sum_i c_{\alpha i}\phi_i,$$

where the total number of electrons, $N = \int_V n(\mathbf{r})\,\mathrm{d}\mathbf{r}$, is given by

$$N = \sum_{ij} D_{ij}S_{ij},$$

where $D_{ij} = \sum_\alpha f_\alpha c_{\alpha i}^* c_{\alpha j}$ is the density matrix.

An approximate atom-based electron density

In practice, a simple approximation is used for the electron density. To this end, we introduce the Mulliken population,

$$m_l = \sum_{i\in l}\sum_j D_{ij}S_{ij},$$

for shell l of atom number μ, and write the total number of electrons as a sum of atomic contributions, $N = \sum_\mu \sum_{l\in\mu} m_l$. The radial dependence of each atomic-like density is represented by a Gaussian function, and the total induced charge in the system is approximated by

$$\delta n(\mathbf{r}) = \sum_\mu \sum_{l\in\mu} \delta m_l \left(\frac{\alpha_l}{\pi}\right)^{\frac{3}{2}} e^{-\alpha_l|\mathbf{r}-\mathbf{R}_\mu|^2},$$

where $\delta m_l = m_l - Z_\mu$ is the total charge for shell l of atom μ, i.e. the sum of the valence electron charge m_l and the ionic charge $-Z_\mu$.

To see the significance of the width α_l of the Gaussian orbital, consider the electrostatic potential from a single Gaussian density at position $\mathbf{R}_\mu$:

$$V_H(\mathbf{r}) = (m_l - Z_\mu)\frac{\mathrm{erf}(\sqrt{\alpha_l}|\mathbf{r}-\mathbf{R}_\mu|)}{|\mathbf{r}-\mathbf{R}_\mu|}.$$

The onsite value of the Hartree potential is $V_H(\mathbf{R}_\mu) = (m_l - Z_\mu)U_l$, where $U_l = 2\sqrt{\frac{\alpha_l}{\pi}}$ is the onsite Hartree shift. In *Semi Empirical*, it is the specified value of U_l that is used to determine the width α_l of the Gaussian using the above relation.

Onsite Hartree shift parameters

The shell-dependent onsite Hartree shift U_l can be obtained from an atomic calculation. U_l is related to the linear shift of the eigenenergy ε_l of shell l as function of the shell occupation q_l:

$$U_l = \frac{d\varepsilon_l}{dq_l}.$$

Thus, U_l can be obtained by performing atomic calculations with different values of q_l.

In QuantumATK, it is recommended to use the same onsite Hartree shift parameter for the s- and p-shells of each atom.

QuantumATK provides a database for U_l calculated using DFT and the PBE functional (see *Generalized-gradient approximation (GGA)*). Access to the data is through the function *ATK_U*. Due to backwards compatibility, the `HoffmanHückelParameters` and `MullerHückelParameters` do not use the *ATK_U* database, but use special values of the electrostatic parameter U, see *Element data*.

Spin polarization

The inclusion of spin in the tight-binding Hamiltonian follows the scheme in[6]. The following spin dependent term is added to the Hamiltonian:

$$H_{ij}^{\sigma} = \pm\frac{1}{2}S_{ij}\left(dE_{l_i} + dE_{l_j}\right),$$

where the sign in the equation depends on the spin.

The spin splitting dE_l of shell l is calculated from the spin-dependent Mulliken populations μ_l of each shell at the local site:

$$dE_l = \sum_{l' \in \mu_l} W_{ll'}\,(m_{l'\uparrow} - m_{l'\downarrow}).$$

Onsite spin-split parameters

The shell-dependent spin splitting strength $W_{ll'}$ can be obtained from a spin-polarized atomic calculation[Page 1855, 6]:

$$W_{ll'} = \frac{1}{2}\left(\frac{d\varepsilon_{l\uparrow}}{dm_{l'\uparrow}} - \frac{d\varepsilon_{l\uparrow}}{dm_{l'\downarrow}}\right).$$

Since $W_{ll'}$ enters symmetrically in the Hamiltonian, it is convenient to symmetrize it:

$$\bar{W}_{ll'} = \frac{1}{2}(W_{ll'} + W_{l'l}).$$

QuantumATK provides a database for $\bar{W}_{ll'}$. Access to the data is through the function *ATK_W*.

[6] C.ăKohler, T.ăFrauenheim, B.ăHourahine, G.ăSeifert, and M.ăSternberg. Treatment of Collinear and Noncollinear Electron Spin within an Approximate Density Functional Based Method. *J. Phys. Chem. A*, 111(26):5622–5629, 2007. doi:10.1021/jp068802p.

Tight-binding total energy

The calculation of the total energy follows[7] and[Page 1851, 1]. The total energy has five terms:

$$E = E_{\mathrm{H}^0} + E_{\delta\mathrm{H}} + E_{\mathrm{ext}} + E_{\mathrm{spin}} + E_{\mathrm{pp}}.$$

•E_{H^0} is the one-electron energy of the non-self-consistent Hamiltonian, given by

$$E_{\mathrm{H}^0} = \sum_{ij} D_{ij} H^0_{ij}.$$

•$E_{\delta\mathrm{H}}$ is the electrostatic difference energy,

$$E_{\delta\mathrm{H}} = \frac{1}{2} \int V_0^H(\mathbf{r}) \delta n(\mathbf{r}) d\mathbf{r}.$$

•E_{ext} is the electrostatic interaction between the electrons and an external field,

$$E_{\mathrm{ext}} = \int V^{\mathrm{ext}}(\mathbf{r}) \delta n(\mathbf{r}) d\mathbf{r}.$$

•E_{spin} is the spin polarization energy[Page 1855, 6],

$$E_{\mathrm{spin}} = -\frac{1}{2} \sum_{\mu} \sum_{l \in \mu} \sum_{l' \in \mu} W(Z_\mu, l, l') m_l m_{l'}.$$

•E_{pp} is the repulsive energy from a pair-potential between each atom pair, $V^{\mathrm{pp}}(Z_\mu, Z_{\mu'}, R_{\mu,\mu'})$,

$$E_{\mathrm{pp}} = \sum_{\mu < \mu'} V^{\mathrm{pp}}(Z_\mu, Z_{\mu'}, R_{\mu,\mu'}).$$

It is optional to add this term to the tight-binding model; it does not affect the electronic structure. The tight-binding model will, however, not give sensible geometries without a repulsive pair-potential.

Important: In the current version of QuantumATK, only the DFTB and Hotbit parameter sets contain a repulsive pair potential term, and only these methods can be used for geometry optimizations.

Parameters

Parameters for the Slater–Koster method

The Slater–Koster parameters can be provided either through the *SlaterKosterTable* class or through various 3rd-party formats. The supported 3rd-party formats are the DFTB Slater–Koster files from the DFTB consortium, and the Hotbit Slater–Koster files from the Hotbit consortium.

To use parameters in the DFTB or Hotbit format, put the parameter files in a single directory, and setup the basis set and pair potentials using the functions *DFTBDirectory* or *HotbitDirectory*.

Tables showing which elements are covered by the parameters are given here: *Slater–Koster basis sets*.

[7] K.ăKaasbjerg and K.ăFlensberg. Strong Polarization-Induced Reduction of Addition Energies in Single-Molecule Nanojunctions. *Nano Letters*, 8(11):3809–3814, 2008. doi:10.1021/nl8021708.

Shipped DFTB and Hotbit parameters

The current version of SE is shipped with a number of DFTB style parameters from the CP2K and Hotbit consortia. We recommended that these resources are cited if the parameters are used in publications. It is most easy to setup these basis sets using the QuantumATK interface.

Shipped Slater–Koster Table parameters

A number of orthogonal tight-binding parameters are provided. The parameters are from Vogl *et al.*[8] and Jancu *et al.*[9], and it is recommended that these papers are cited if the parameters are used for publications. These basis sets can easily be set up using QuantumATK.

Parameters for the extended Hückel method

The parameters η_1, η_2, C_1, C_2, and E must be defined for each valence orbital, while β and U only depend on the element type. Different parameter sets are provided with *Semi Empirical*, but it is also possible to provide user-defined parameters in the input file using the *HuckelBasisParameters* class.

The tables below provide a mapping between the symbols in the equations and the corresponding keywords.

Table 4.2: HuckelBasisParameters.

Symbol	*HuckelBasisParameters*
E_i	`ionization_potential`
β	`wolfsberg_helmholtz_constant`
U	`onsite_hartree_shift`
W	`onsite_spin_split`
E^{VAC}	`vacuum_level`

Table 4.3: SlaterOrbital parameters.

Symbol	*SlaterOrbital* parameters
n	`principal_quantum_number`
l	`angular_momentum`
η	`slater_coefficients`
C	`weights`

The current version of QuantumATK comes with built-in Hoffmann and Müller parameter sets, which are appropriate for organic molecules. For crystalline structures, both metals and organic materials like graphene, parameters from J. Cerda are provided. When using these parameters,[10] should be referenced. Tables with the available parameter sets can be found here: *Extended Hückel basis sets*.

[8] P.ăVogl, HaroldăP. Hjalmarson, and J.ăD. Dow. A Semi-empirical tight-binding theory of the electronic structure of semiconductors. *J. Phys. Chem. Solids*, 44(5):365–378, jan 1983. URL: http://www.sciencedirect.com/science/article/pii/0022369783900641.

[9] J.-M. Jancu, R.ăScholz, F.ăBeltram, and F.ăBassani. Empirical spds* tight-binding calculation for cubic semiconductors: General method and material parameters. *Phys. Rev. B*, 57:6493–6507, Mar 1998. doi:10.1103/PhysRevB.57.6493.

[10] J.ăCerda and F.ăSoria. Accurate and transferable extended Hückel-type tight-binding parameters. *Phys. Rev. B*, 61:7965–7971, Mar 2000. doi:10.1103/PhysRevB.61.7965.

To combine parameters from different sources, it is important to make sure they use the same energy zero level, in order to obtain correct charge transfers. This can be obtained by ensuring that the crystals have the correct work function and molecules the correct ionisation energies. For this purpose, an additional parameter E^{VAC} is introduced, which shifts the energy of the vacuum level: If a calculation with $E^{\mathrm{VAC}} = 0$ eV has a work function of 6.5 eV, then by setting $E^{\mathrm{VAC}} = -1.5$ eV all bands shift rigidly upwards by 1.5 eV, and the work function becomes 5.0 eV.

Note: The Hückel parameters have been fitted for non-self-consistent calculations. To use the parameters in self-consistent calculations, the self-consistent onsite shifts must be compensated by a reverse shift of the `vacuum_levels`.

4.2.4 Force Field

Table of Contents

•*Introduction*

•*TremoloX*

•*TremoloX potential classes*

•*Neural network based force field parameter sets*

•*How to choose the right pretrained neural network based force field*

•*TremoloX potential parameter sets*

•*Pretrained moment tensor potential (MTP) parameter sets*

Introduction

The **ForceField** module provides calculators for empirical force fields[1]. It includes the following calculators:

•*TremoloXCalculator*

•*BrennerCalculator*

Note: For QuantumATK-version 2018, this module is present under the name **ATK-ForceField**.

TremoloX

The *TremoloXCalculator*, which provides most of the potential classes and parameters sets in the **ForceField** module, is developed by the Fraunhofer Institute for Algorithms and Scientific Computing (SCAI). For details about TremoloX, see also www.tremolo-x.com.

TremoloX uses highly efficient state of the art algorithms for the treatment of short- and long-range potentials. TremoloX provides a large database of pre-defined *TremoloX potential parameter sets*, for modeling of systems in material science and nanotechnology. It also allows the user to set up custom potentials by combining the available *TremoloX potential classes*.

[1] J.ăSchneider, J.ăHamaekers, S.T. Chill, S.ăSmidstrup, J.ăBulin, R.ăThesen, A.ăBlom, and K.ăStokbro. ATK-Classical: A New Generation Molecular Dynamics Software Package. *arXiv*, pages 1701.02495, 2017. URL: https://arxiv.org/abs/1701.02495.

TremoloX potential classes

- *TremoloXCalculator*
- *Angle1Potential*
- *AngleCorrection6Potential*
- *BuckinghamPotential*
- *COMB3FieldCorrection*
- *COMB3PairPotential*
- *COMB3Particle*
- *COMBCoulombOption*
- *COMBMixitPotential*
- *COMBOptimizationOption*
- *COMBOverCoordinationCorrection*
- *COMBPairPotential*
- *COMBSMCoulomb*
- *COMBSelfEnergyCorrection*
- *COMBSelfEnergyCorrectionShan*
- *COMBTriplePotential*
- *ConstantPotential*
- *CoreShellHarmonicPotential*
- *CoreShellMorsePotential*
- *CoreShellOptimizationOption*
- *CosineAnglePotential*
- *CosinePowerTorsionPotential*
- *CosineTorsionPotential*
- *CoulombDSF*
- *CoulombDebye*
- *CoulombERFCPotential*
- *CoulombErf*
- *CoulombErfDSF*
- *CoulombEwald*
- *CoulombN2*
- *CoulombN2Spline*
- *CoulombOption*
- *CoulombQTaperPotential*
- *CoulombReaxFF*
- *CoulombSPME*

- *CoulombScale14Option*
- *CoulombSolver*
- *D3Potential*
- *D4Potential*
- *DampedDispersionPotential*
- *DreidingAnglePotential*
- *EAMfssetflPotential*
- *EAMsetflPotential*
- *EEMOption*
- *EEMSinglePotential*
- *EmtPotential*
- *FENEBondPotential*
- *FourierAnglePotential*
- *FourierImproperPotential*
- *FourierTorsionPotential*
- *GaussPotential*
- *GaussianAnglePotential*
- *General1Potential*
- *General2Potential*
- *General3Potential*
- *GeneralStiwe2Potential*
- *HarmonicAnglePotential*
- *HarmonicBondPotential*
- *HarmonicCosineAnglePotential*
- *HarmonicUreyBradleyAnglePotential*
- *ImproperCosineTorsionPotential*
- *ImproperTorsionPotential*
- *LennardJonesMNPotential*
- *LennardJonesPotential*
- *LennardJonesShiftPotential*
- *LennardJonesSpline2Potential*
- *LennardJonesSplinePotential*
- *MTPPotential*
- *MaddenDispersivePotential*
- *MaddenEwaldOption*
- *MaddenOptimizationOption*

- *MaddenParticle*
- *MaddenPolarizationDampingPotential*
- *MaddenRepulsivePotential*
- *MeamElementPotential*
- *MeamGlobalOption*
- *MeamPairPotential*
- *MeamScreeningPotential*
- *ModifiedStiwe2Potential*
- *ModifiedStiwe3Potential*
- *MolierePotential*
- *MorseBondPotential*
- *MorsePotential*
- *ParticleIdentifier*
- *ParticleType*
- *RSL2Potential*
- ReaxFFBlock
- ReaxFFParameterContainer
- *ReaxFFPotential*
- *Repulsive12Potential*
- *Stiwe2Potential*
- *Stiwe3Potential*
- *SuttonChenPotential*
- *TabulatedBondPotential*
- *TaperedAnglePotential*
- *TersoffBrennerBOPairPotential*
- *TersoffBrennerCorrectionPotential*
- *TersoffBrennerCorrectionPotential2*
- *TersoffBrennerMolierePotential*
- *TersoffBrennerPairPotential*
- *TersoffBrennerPairPotential2*
- *TersoffBrennerPairPotential3*
- *TersoffBrennerSplinePotential*
- *TersoffBrennerTorsionCorrectionPotential*
- *TersoffBrennerTriplePotential*
- *TersoffBrennerTriplePotential2*
- *TersoffBrennerTriplePotential3*

- *TersoffBrennerTriplePotential4*
- *TersoffBrennerTriplePotential5*
- *TersoffDiag2Potential*
- *TersoffDiagPotential*
- *TersoffMixitPotential*
- *TersoffSingleTypePotential*
- *TersoffTriplePotential*
- *TersoffZBLPotential*
- *TorchXPotential*
- *TosiFumiPotential*
- *UserDefinedTabulatedPotential*
- *VFFBondBendingPotential*
- *VFFBondBendingStretchingPotential*
- *VFFBondStretchingPotential*
- *VFFCoplanarBondBendingPotential*
- *VFFCrossBondStretchingPotential*
- *VFFModifiedBondBendingPotential1*
- *VFFModifiedBondBendingStretchingPotential1*
- *VFFModifiedCoplanarBondBendingPotential1*
- *VFFModifiedCrossBondStretchingPotential1*
- *VessalPotential*
- *ZBLPotential*
- *ZBLStiwePotential*

Neural network based force field parameter sets

TorchX_CHGNet_0_3_0 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

B.Deng, P. Zhong, K. Jun, J. Riebesell, K. Han, C. J. Bartel, and G. Ceder, *CHGNet as a pretrained universal neural network potential for charge-informed atomistic modelling*, Nature Machine Intelligence, pp. 1–11, 2023 link

TorchX_M3GNet_MP_DIRECT_PES_2021 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

J.Qi, T. W. Ko, B. C. Wood, T. A. Pham, and S. P. Ong, *Robust training of machine learning interatomic potentials with dimensionality reduction and stratified sampling*, npj Computational Materials, 10, p. 43, 2024 link

TorchX_M3GNet_MP_PES_2021 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

C.Chen and S. P. Ong, *A universal graph deep learning interatomic potential for the periodic table*, Nature Computational Science, 2, pp. 718–728, 2022 link

TorchX_MACE_MPA_0_medium (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

I.Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács, J. Riebesell, X. R. Advincula, M. Asta, W. J. Baldwin, N. Bernstein, A. Bhowmik, S. M. Blau, V. Crare, J. P. Darby, S. De, F. D. Pia, V. L. Deringer, R. Elijoius, Z. El-Machachi, E. Fako, A. C. Ferrari, A. Genreith-Schriever, J. George, R. E. A. Goodall, C. P. Grey, S. Han, W. Handley, H. H. Heenen, K. Hermansson, C. Holm, J. Jaafar, S. Hofmann, K. S. Jakob, H. Jung, V. Kapil, A. D. Kaplan, N. Karimitari, N. Kroupa, J. Kullgren, M. C. Kuner, D. Kuryla, G. Liepuoniute, J. T. Margraf, I. Magdu, A. Michaelides, J. H. Moore, A. A. Naik, S. P. Niblett, S. W. Norwood, N. ONeill, C. Ortner, K. A. Persson, K. Reuter, A. S. Rosen, L. L. Schaaf, C. Schran, E. Sivonxay, T. K. Stenczel, V. Svahn, C. Sutton, C. van der Oord, E. Varga-Umbrich, T. Vegge, M. Vondrák, Y. Wang, W. C. Witt, F. Zills, and G. Csányi, *A foundation model for atomistic materials chemistry*, 2023 link

TorchX_MACE_MP_0_L0_2023 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

I.Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács, J. Riebesell, X. R. Advincula, M. Asta, W. J. Baldwin, N. Bernstein, A. Bhowmik, S. M. Blau, V. Crare, J. P. Darby, S. De, F. D. Pia, V. L. Deringer, R. Elijoius, Z. El-Machachi, E. Fako, A. C. Ferrari, A. Genreith-Schriever, J. George, R. E. A. Goodall, C. P. Grey, S. Han, W. Handley, H. H. Heenen, K. Hermansson, C. Holm, J. Jaafar, S. Hofmann, K. S. Jakob, H. Jung, V. Kapil, A. D. Kaplan, N. Karimitari, N. Kroupa, J. Kullgren, M. C. Kuner, D. Kuryla, G. Liepuoniute, J. T. Margraf, I. Magdu, A. Michaelides, J. H. Moore, A. A. Naik, S. P. Niblett, S. W. Norwood, N. ONeill, C. Ortner, K. A. Persson, K. Reuter, A. S. Rosen, L. L. Schaaf, C. Schran, E. Sivonxay, T. K. Stenczel, V. Svahn, C. Sutton, C. van der Oord, E. Varga-Umbrich, T. Vegge, M. Vondrák, Y. Wang, W. C. Witt, F. Zills, and G. Csányi, *A foundation model for atomistic materials chemistry*, 2023 link

TorchX_MACE_MP_0_L1_2023 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

I.Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács, J. Riebesell, X. R. Advincula, M. Asta, W. J. Baldwin, N. Bernstein, A. Bhowmik, S. M. Blau, V. Crare, J. P. Darby, S. De, F. D. Pia, V. L. Deringer, R. Elijoius, Z. El-Machachi, E. Fako, A. C. Ferrari, A. Genreith-Schriever, J. George, R. E. A. Goodall, C. P. Grey, S. Han, W. Handley, H. H. Heenen, K. Hermansson, C. Holm, J. Jaafar, S. Hofmann, K. S. Jakob, H. Jung, V. Kapil, A. D. Kaplan, N. Karimitari, N. Kroupa, J. Kullgren, M. C. Kuner, D. Kuryla, G. Liepuoniute, J. T. Margraf, I. Magdu, A. Michaelides, J. H. Moore, A. A. Naik, S. P. Niblett, S. W. Norwood, N. ONeill, C. Ortner, K. A. Persson, K. Reuter, A. S. Rosen, L. L. Schaaf, C. Schran, E. Sivonxay, T. K. Stenczel, V. Svahn, C. Sutton, C. van der Oord, E. Varga-Umbrich, T. Vegge, M. Vondrák, Y. Wang, W. C. Witt, F. Zills, and G. Csányi, *A foundation model for atomistic materials chemistry*, 2023 link

TorchX_MACE_MP_0_L2_2024 (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

I.Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács, J. Riebesell, X. R. Advincula, M. Asta, W. J. Baldwin, N. Bernstein, A. Bhowmik, S. M. Blau, V. Crare, J. P. Darby, S. De, F. D. Pia, V. L. Deringer, R. Elijoius, Z. El-Machachi, E. Fako, A. C. Ferrari, A. Genreith-Schriever, J. George, R. E. A. Goodall, C. P. Grey, S. Han, W. Handley, H. H. Heenen, K. Hermansson, C. Holm, J. Jaafar, S. Hofmann, K. S. Jakob, H. Jung, V. Kapil, A. D. Kaplan, N. Karimitari, N. Kroupa, J. Kullgren, M. C. Kuner, D. Kuryla, G. Liepuoniute, J. T. Margraf, I. Magdu, A. Michaelides, J. H. Moore, A. A. Naik, S. P. Niblett, S. W. Norwood, N. ONeill, C. Ortner, K. A. Persson, K. Reuter, A. S. Rosen, L. L. Schaaf, C. Schran, E. Sivonxay, T. K. Stenczel, V. Svahn, C. Sutton, C. van der Oord, E. Varga-Umbrich, T. Vegge, M. Vondrák, Y. Wang, W. C. Witt, F. Zills, and G. Csányi, *A foundation model for atomistic materials chemistry*, 2023 link

TorchX_MACE_MP_0b3_medium (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

I.Batatia, P. Benner, Y. Chiang, A. M. Elena, D. P. Kovács, J. Riebesell, X. R. Advincula, M. Asta, W. J. Baldwin, N. Bernstein, A. Bhowmik, S. M. Blau, V. Crare, J. P. Darby, S. De, F. D. Pia, V. L. Deringer, R. Elijoius, Z. El-Machachi, E. Fako, A. C. Ferrari, A. Genreith-Schriever, J. George, R. E. A. Goodall, C. P. Grey, S. Han, W. Handley, H. H. Heenen, K. Hermansson, C. Holm, J. Jaafar, S. Hofmann, K. S. Jakob, H. Jung, V. Kapil, A. D. Kaplan, N. Karimitari, N. Kroupa, J. Kullgren, M. C. Kuner, D. Kuryla, G. Liepuoniute, J. T. Margraf, I. Magdu, A. Michaelides, J. H. Moore, A. A. Naik, S. P. Niblett, S. W. Norwood, N. ONeill, C. Ortner, K. A. Persson, K. Reuter, A. S. Rosen, L. L. Schaaf, C. Schran, E. Sivonxay, T. K. Stenczel, V. Svahn, C. Sutton, C. van der Oord, E. Varga-Umbrich, T. Vegge, M. Vondrák, Y. Wang, W. C. Witt, F. Zills, and G. Csányi, *A foundation model for atomistic materials chemistry*, 2023 link

TorchX_MACE_Osaka24_large (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

T.Shiota, K. Ishihara, T. M. Do, T. Mori, and W. Mizukami, *Taming Multi-Domain, -Fidelity Data: Towards Foundation Models for Atomistic Scale Simulations*, 2024 link

TorchX_MACE_Osaka24_medium (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

T.Shiota, K. Ishihara, T. M. Do, T. Mori, and W. Mizukami, *Taming Multi-Domain, -Fidelity Data: Towards Foundation Models for Atomistic Scale Simulations*, 2024 link

TorchX_MACE_Osaka24_small (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

T.Shiota, K. Ishihara, T. M. Do, T. Mori, and W. Mizukami, *Taming Multi-Domain, -Fidelity Data: Towards Foundation Models for Atomistic Scale Simulations*, 2024 link

TorchX_MatterSim_v1_0_0_1M (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

H.Yang, C. Hu, Y. Zhou, X. Liu, Y. Shi, J. Li, G. Li, Z. Chen, S. Chen, C. Zeni, M. Horton, R. Pinsler, A. Fowler, D. Zügner, T. Xie, J. Smith, L. Sun, Q. Wang, L. Kong, C. Liu, H. Hao, and Z. Lu, *MatterSim: A Deep Learning Atomistic Model Across Elements, Temperatures and Pressures*, arXiv preprint arXiv:2405.04967, 2024 link

TorchX_MatterSim_v1_0_0_5M (Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Ca, Xe, Yb, Mo, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa)

H.Yang, C. Hu, Y. Zhou, X. Liu, Y. Shi, J. Li, G. Li, Z. Chen, S. Chen, C. Zeni, M. Horton, R. Pinsler, A. Fowler, D. Zügner, T. Xie, J. Smith, L. Sun, Q. Wang, L. Kong, C. Liu, H. Hao, and Z. Lu, *MatterSim: A Deep Learning Atomistic Model Across Elements, Temperatures and Pressures*, arXiv preprint arXiv:2405.04967, 2024 link

How to choose the right pretrained neural network based force field

Selecting the best pretrained model for your system can be challenging. The optimal choice depends on the specific system, desired accuracy, and evaluation speed. It is recommended that users try several models and experiments to determine which suits their specific use case. Here are some guidelines to help you make an informed decision:

Prefer newer or updated models

Newer or updated models are typically more accurate than older ones. For example:

- Updated versions of the medium MACE models (`TorchX_MACE_MP_0b3_medium` and `TorchX_MACE_MPA_0_medium`) are expected to outperform the original medium MACE model (`TorchX_MACE_MP_0_L1_2023`).

Training dataset size

Models trained on larger datasets are typically more accurate than those trained on smaller datasets. For instance:

- `TorchX_MACE_MPA_0_medium` is trained on a larger dataset compared to `TorchX_MACE_MP_0b3_medium`, and is therefore expected to perform better.
- Similarly, MatterSim models are expected to be more accurate than M3GNet models because they are trained on larger datasets.

System type

The type of system you are working with can influence the choice of model. For example:

- For systems containing molecules, the Osaka versions of the MACE models are likely more suitable, as they are the only models trained on molecules.

Speed

Larger models may offer better accuracy but can be slower to evaluate. If speed is a concern, consider using smaller models or those optimized for faster evaluation.

- `TorchX_MACE_MP_0_L0_2023` is the fastest MACE model.
- MatterSim and M3GNet models are generally faster than any of the MACE models.

TremoloX potential parameter sets

Aldermann_TaO_2018 (Ta, O)

O. Alderman, C. Benmore, J. Neuefeind, E. Coillet, A. Mermet, V. Martinez, A. Tamalonis, and R. Weber, *Amorphous tantala and its relationship with the molten state*, Physical Review Materials, 2, p. 043602, 2018

Anwar_NaCl_2003 (Na, Cl)

J. Anwar, D. Frenkel, and M. G. Noro, *Calculation of the melting point of NaCl by molecular simulation*, The Journal of Chemical Physics, 118, pp. 728-735, 2003 link

Billeter_HNOSi_2006 (H, O, Si, N)

R. Billeter, A. Curioni, D. Fischer, and W. Andreoni, *Ab initio derived augmented Tersoff potential for silicon oxynitride compounds and their interfaces with silicon*, Phys. Rev. B, 73, p. 155329, 2006 link

Brenner_CH_1990 (C, H)

D. W. Brenner, *Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films*, Phys. Rev. B, 42, pp. 9458-9471, 1990

Brenner_CH_2002 (C, H)

D. W. Brenner, O. A. Shenderova, J. A. Harrison, S. J. Stuart, B. Ni, and S. B. Sinnott, *A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons*, J, Phys.: Condens. Matter, pp. 783-802, 2002

Broglia_HfOSi_2014 (Hf, O, Si)

G. Broglia, G. Ori, L. Larcher, and M. Montorsi, *Molecular dynamics simulation of amorphous HfO2 for resistive RAM applications*, Modelling and Simulation in Materials Science and Engineering, 22, p. 065006, 2014 link

COMB_NTi_2014 (Ti, N)

Y. Cheng, T. Liang, J. A. Martinez, S. R. Phillpot, and S. B. Sinnott, *A charge optimized many-body potential for titanium nitride (TiN)*, Journal of Physics: Condensed Matter, 26, p. 265004, 2014 link

COMB_OSi_2007 (O, Si)

J. Yu, S. B. Sinnott, and S. R. Phillpot, *Charge optimized many-body potential for the Si/SiO2 system*, Phys. Rev. B, 75, p. 085311, 2007 link

COMB_OSi_2010 (O, Si)

T. Shan, B. D. Devine, J. M. Hawkins, A. Asthagiri, S. R. Phillpot, and S. B. Sinnott, *Second-generation charge-optimized many-body potential for Si/SiO2 and amorphous silica*, Phys. Rev. B, 82, p. 235302, 2010 link

CoulombD4EEQ (Re, Na, N, Co, As, C, Pd, Ge, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, F, Po, Ca, Xe, Yb, Mo, At, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, H, Sr, Rn)

E. Caldeweyher, J. Mewes, S. Ehlert, and S. Grimme, *Extension and evaluation of the D4 London-dispersion model for periodic systems*, Physical Chemistry Chemical Physics, 22, pp. 8499–8512, 2020

E. Caldeweyher, S. Ehlert, A. Hansen, H. Neugebauer, S. Spicher, C. Bannwarth, and S. Grimme, *A generally applicable atomic-charge dependent London dispersion correction*, The Journal of chemical physics, 150, p. 154122, 2019

A. Ghasemi, A. Hofstetter, S. Saha, and S. Goedecker, *Interatomic potentials for ionic systems with density functional accuracy based on charge densities obtained by a neural network*, Physical review B, 92, p. 045131, 2015

E.Caldeweyher, C. Bannwarth, and S. Grimme, *Extension of the D3 dispersion coefficient model*, The Journal of chemical physics, 147, p. 034112, 2017

CoulombD4EEQDSF (Re, Na, N, Co, As, C, Pd, Ge, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, F, Po, Ca, Xe, Yb, Mo, At, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, H, Sr, Rn)

C.Fennell and J. D. Gezelter, *Is the Ewald summation still necessary? Pairwise alternatives to the accepted standard for long-range electrostatics*, The Journal of chemical physics, 124, p. 234104, 2006

E.Caldeweyher, S. Ehlert, A. Hansen, H. Neugebauer, S. Spicher, C. Bannwarth, and S. Grimme, *A generally applicable atomic-charge dependent London dispersion correction*, The Journal of chemical physics, 150, p. 154122, 2019

E.Caldeweyher, J. Mewes, S. Ehlert, and S. Grimme, *Extension and evaluation of the D4 London-dispersion model for periodic systems*, Physical Chemistry Chemical Physics, 22, pp. 8499–8512, 2020

A.Ghasemi, A. Hofstetter, S. Saha, and S. Goedecker, *Interatomic potentials for ionic systems with density functional accuracy based on charge densities obtained by a neural network*, Physical review B, 92, p. 045131, 2015

E.Caldeweyher, C. Bannwarth, and S. Grimme, *Extension of the D3 dispersion coefficient model*, The Journal of chemical physics, 147, p. 034112, 2017

DispersionD3BJ (Fr, Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, Ra, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Po, Ca, Xe, Yb, Mo, At, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa, Rn)

S.Grimme, J. Antony, S. Ehrlich, and H. Krieg, *A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu*, The Journal of chemical physics, 132, p. 154104, 2010

S.Grimme, S. Ehrlich, and L. Goerigk, *Effect of the damping function in dispersion corrected density functional theory*, Journal of computational chemistry, 32, pp. 1456–1465, 2011

DispersionD3Z (Fr, Re, Na, N, Co, Th, As, C, Pd, Ge, Ac, He, Cl, Pb, Ba, Tl, Au, Sc, Zn, I, Fe, Eu, Ga, Be, Pm, Tb, Zr, Cs, Os, Ir, Br, Er, La, U, Al, Ta, Ce, Cr, W, P, Ag, Ru, Sb, Kr, Gd, Ho, Ni, Si, Nd, Li, Y, Sn, Pr, Cd, Hg, Tc, Bi, Ra, V, Ti, Mn, Te, Rh, Ne, Cu, Tm, In, O, Nb, Sm, Pt, Pu, F, Po, Ca, Xe, Yb, Mo, At, B, Ar, S, K, Lu, Se, Dy, Hf, Mg, Rb, Np, H, Sr, Pa, Rn)

S.Grimme, J. Antony, S. Ehrlich, and H. Krieg, *A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu*, The Journal of chemical physics, 132, p. 154104, 2010

Dyson_CHSi_1999 (Si, C, H)

A.Dyson and P. V. Smith, *Improved empirical interatomic potential for CSiH systems*, Molecular Physics, 96, pp. 1491-1507, 1999 link

EAMFS_Ag_1987 (Ag)

G.Ackland, G. Tichy, V. Vitek, and M. Finnis, *Simple N-body potentials for the noble metals and nickel*, Philosophical Magazine A, 56, pp. 735–756, 1987

EAMFS_AlFe_2005 (Al, Fe)

M.Mendelev, D. J. Srolovitz, G. J. Ackland, and S. Han, *Effect of Fe segregation on the migration of a non-symmetric Sigma 5 tilt grain boundary in Al*, J. Mater. Res., 20, pp. 208-218, 2005

EAMFS_AlMg_2009 (Mg, Al)

M.Mendelev, M. Asta, M. Rahman, and J. Hoyt, *Development of interatomic potentials appropriate for simulation of solid–liquid interface properties in Al–Mg alloys*, Philosophical Magazine, 89, pp. 3269–3285, 2009

EAMFS_AlSm_2015 (Sm, Al)

M.Mendelev, F. Zhang, Z. Ye, Y. Sun, M. Nguyen, S. Wilson, C. Wang, and K. Ho, *Development of interatomic potentials appropriate for simulation of devitrification of Al90Sm10 alloy*, Modelling and Simulation in Materials Science and Engineering, 23, p. 045013, 2015

EAMFS_Al_2000 (Al)

B.Sturgeon and B. B. Laird, *Adjusting the melting point of a model system via Gibbs-Duhem integration: Application to a model of aluminum*, Physical Review B, 62, p. 14720, 2000

EAMFS_Al_2008 (Al)

M.Mendelev, M. Kramer, C. Becker, and M. Asta, *Analysis of semi-empirical interatomic potentials appropriate for simulation of crystalline and liquid Al and Cu*, Philosophical Magazine, 88, pp. 1723–1750, 2008

EAMFS_Au_1987 (Au)

G.Ackland, G. Tichy, V. Vitek, and M. Finnis, *Simple N-body potentials for the noble metals and nickel*, Philosophical Magazine A, 56, pp. 735–756, 1987

EAMFS_CFe_2008 (C, Fe)

D.Hepburn and G. J. Ackland, *Metallic-covalent interatomic potential for carbon in iron*, Physical Review B, 78, p. 165115, 2008

EAMFS_CuZr_2007 (Cu, Zr)

M.Mendelev, D. Sordelet, and M. Kramer, *Using atomistic computer simulations to analyze x-ray diffraction data from metallic glasses*, Journal of Applied Physics, 102, pp. 043501–043501, 2007

EAMFS_CuZr_2009 (Cu, Zr)

M.Mendelev, M. Kramer, R. Ott, D. Sordelet, D. Yagodin, and P. Popel, *Development of suitable interatomic potentials for simulation of liquid and amorphous Cu–Zr alloys*, Philosophical Magazine, 89, pp. 967–987, 2009

EAMFS_CuZr_2016 (Cu, Zr)

V.Borovikov, M. I. Mendelev, and A. H. King, *Effects of stable and unstable stacking fault energy on dislocation nucleation in nano-crystalline metals*, Modelling and Simulation in Materials Science and Engineering, 24, p. 085017, 2016

EAMFS_Cu_1987 (Cu)

G.Ackland, G. Tichy, V. Vitek, and M. Finnis, *Simple N-body potentials for the noble metals and nickel*, Philosophical Magazine A, 56, pp. 735–756, 1987

EAMFS_Cu_1990 (Cu)

G.Ackland and V. Vitek, *Many-body potentials and atomic-scale relaxations in noble-metal alloys*, Physical review B, 41, p. 10324, 1990

EAMFS_Cu_2008 (Cu)

M.Mendelev, M. Kramer, C. Becker, and M. Asta, *Analysis of semi-empirical interatomic potentials appropriate for simulation of crystalline and liquid Al and Cu*, Philosophical Magazine, 88, pp. 1723–1750, 2008

EAMFS_FeP_2004 (P, Fe)

G.Ackland, M. Mendelev, D. Srolovitz, S. Han, and A. Barashev, *Development of an interatomic potential for phosphorus impurities in -iron*, Journal of Physics: Condensed Matter, 16, p. S2629, 2004

EAMFS_FeV_2007 (V, Fe)

M.Mendelev, S. Han, W. Son, G. J. Ackland, and D. J. Srolovitz, *Simulation of the interaction between Fe impurities and point defects in V*, Physical Review B, 76, p. 214105, 2007

EAMFS_Fe_1997 (Fe)

G.Ackland, D. Bacon, A. Calder, and T. Harry, *Computer simulation of point defect properties in dilute FeCu alloy using a many-body interatomic potential*, Philosophical Magazine A, 75, pp. 713–732, 1997

EAMFS_Fe_2003 (Fe)

M.Mendelev, S. Han, D. Srolovitz, G. Ackland, D. Sun, and M. Asta, *Development of new interatomic potentials appropriate for crystalline and liquid iron*, Philosophical magazine, 83, pp. 3977–3994, 2003

EAMFS_Fe_2003b (Fe)

M.Mendelev, S. Han, D. Srolovitz, G. Ackland, D. Sun, and M. Asta, *Development of new interatomic potentials appropriate for crystalline and liquid iron*, Philosophical magazine, 83, pp. 3977–3994, 2003

EAMFS_Fe_2010 (Fe)

L.Malerba, M. Marinica, N. Anento, C. Björkas, H. Nguyen, C. Domain, F. Djurabekova, P. Olsson, K. Nordlund, A. Serra, and others, *Comparison of empirical interatomic potentials for iron applied to radiation damage studies*, Journal of Nuclear Materials, 406, pp. 19–38, 2010

EAMFS_Fe_2012 (Fe)

L.Proville, D. Rodney, and M. Marinica, *Quantum effect on thermally activated glide of dislocations*, Nature materials, 11, p. 845, 2012

EAMFS_Mg_2006 (Mg)

D.Sun, M. Mendelev, C. Becker, K. Kudin, T. Haxhimali, M. Asta, J. Hoyt, A. Karma, and D. Srolovitz, *Crystal-melt interfacial free energies in hcp metals: A molecular dynamics study of Mg*, Physical Review B, 73, p. 024116, 2006

EAMFS_Na_2015 (Na)

S.Wilson, K. Gunawardana, and M. Mendelev, *Solid-liquid interface free energies of pure bcc metals and B2 phases*, The Journal of chemical physics, 142, p. 134705, 2015

EAMFS_NiNb_2016 (Ni, Nb)

Y.Zhang, R. Ashcraft, M. Mendelev, C. Wang, and K. Kelton, *Experimental and molecular dynamics simulation study of structure of liquid and amorphous Ni62Nb38 alloy*, The Journal of chemical physics, 145, p. 204505, 2016

EAMFS_NiZr_2012 (Zr, Ni)

M.Mendelev, M. Kramer, S. Hao, K. Ho, and C. Wang, *Development of interatomic potentials appropriate for simulation of liquid and glass properties of NiZr2 alloy*, Philosophical Magazine, 92, pp. 4454–4469, 2012

EAMFS_NiZr_2015 (Zr, Ni)

S.Wilson and M. Mendelev, *Anisotropy of the solid–liquid interface properties of the Ni–Zr B33 phase from molecular dynamics simulation*, Philosophical Magazine, 95, pp. 224–241, 2015

EAMFS_Ni_1987 (Ni)

G.Ackland, G. Tichy, V. Vitek, and M. Finnis, *Simple N-body potentials for the noble metals and nickel*, Philosophical Magazine A, 56, pp. 735–756, 1987

EAMFS_Ni_2012 (Ni)

M.Mendelev, M. Kramer, S. Hao, K. Ho, and C. Wang, *Development of interatomic potentials appropriate for simulation of liquid and glass properties of NiZr2 alloy*, Philosophical Magazine, 92, pp. 4454–4469, 2012

EAMFS_Ru_2008 (Ru)

A.Fortini, M. I. Mendelev, S. Buldyrev, and D. Srolovitz, *Asperity contacts at the nanoscale: Comparison of Ru and Au*, Journal of Applied Physics, 104, pp. 074320–074320, 2008

EAMFS_Ti_1992 (Ti)

G.Ackland, *Theoretical study of titanium surfaces and defects with a new many-body potential*, Philosophical Magazine A, 66, pp. 917–932, 1992

EAMFS_Ti_Mendelev_1_2016 (Ti)

M.Mendelev, T. Underwood, and G. Ackland, *Interatomic Potentials for the Simulation of Defects, Plasticity and Phase Transformations in Titanium*, TBP

EAMFS_Ti_Mendelev_2_2016 (Ti)

M.Mendelev, T. Underwood, and G. Ackland, *Interatomic Potentials for the Simulation of Defects, Plasticity and Phase Transformations in Titanium*, TBP

EAMFS_Ti_Mendelev_3_2016 (Ti)

M.Mendelev, T. Underwood, and G. Ackland, *Interatomic Potentials for the Simulation of Defects, Plasticity and Phase Transformations in Titanium*, TBP

EAMFS_W_2_2014 (W)

M.Marinica, L. Ventelon, M. Gilbert, L. Proville, S. Dudarev, J. Marian, G. Bencteux, and F. Willaime, *Interatomic potentials for modelling radiation defects and dislocations in tungsten*, Journal of Physics: Condensed Matter, 25, p. 395502, 2013

EAMFS_W_3_2014 (W)

M.Marinica, L. Ventelon, M. Gilbert, L. Proville, S. Dudarev, J. Marian, G. Bencteux, and F. Willaime, *Interatomic potentials for modelling radiation defects and dislocations in tungsten*, Journal of Physics: Condensed Matter, 25, p. 395502, 2013

EAMFS_W_4_2014 (W)

M.Marinica, L. Ventelon, M. Gilbert, L. Proville, S. Dudarev, J. Marian, G. Bencteux, and F. Willaime, *Interatomic potentials for modelling radiation defects and dislocations in tungsten*, Journal of Physics: Condensed Matter, 25, p. 395502, 2013

EAM_AgCu_2006 (Cu, Ag)

P.Williams, Y. Mishin, and J. Hamilton, *An embedded-atom potential for the Cu–Ag system*, Modelling and Simulation in Materials Science and Engineering, 14, p. 817, 2006

EAM_AgCu_2009 (Cu, Ag)

H.Wu and D. R. Trinkle, *Cu/Ag EAM potential optimized for heteroepitaxial diffusion from ab initio data*, Computational Materials Science, 47, pp. 577–583, 2009

EAM_AgHPd_Hybrid_2013 (Ag, H, Pd)

M.Hale, B. M. Wong, J. A. Zimmerman, and X. Zhou, *Atomistic potentials for palladium–silver hydrides*, Modelling and Simulation in Materials Science and Engineering, 21, p. 045005, 2013

EAM_AgHPd_Morse_2013 (Ag, H, Pd)

M.Hale, B. M. Wong, J. A. Zimmerman, and X. Zhou, *Atomistic potentials for palladium–silver hydrides*, Modelling and Simulation in Materials Science and Engineering, 21, p. 045005, 2013

EAM_Ag_2004 (Ag)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Ag_2006 (Ag)

P.Williams, Y. Mishin, and J. Hamilton, *An embedded-atom potential for the Cu–Ag system*, Modelling and Simulation in Materials Science and Engineering, 14, p. 817, 2006

EAM_Ag_Sheng_2011 (Ag)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_AlAg_Sheng_2011 (Ag, Al)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_AlCu_Sheng_2011 (Cu, Al)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_AlMg_1997 (Mg, Al)

X.Liu, P. Ohotnicky, J. Adams, C. L. Rohrer, and R. Hyland Jr, *Anisotropic surface segregation in Al Mg alloys*, Surface science, 373, pp. 357–370, 1997

EAM_AlMnPd_2012 (Mn, Al, Pd)

D.Schopf, P. Brommer, B. Frigan, and H. Trebin, *Embedded atom method potentials for Al-Pd-Mn phases*, Physical Review B, 85, p. 054201, 2012

EAM_AlNbTi_1996 (Ti, Al, Nb)

D.Farkas and C. Jones, *Interatomic potentials for ternary Nb-Ti-Al alloys*, Modelling and Simulation in Materials Science and Engineering, 4, p. 23, 1996

EAM_AlNiH_1997 (Ni, Al, H)

H.Angelo, N. R. Moody, and M. I. Baskes, *Trapping of hydrogen to lattice defects in nickel*, Modelling and Simulation in Materials Science and Engineering, 3, p. 289, 1995

M.Baskes, X. Sha, J. Angelo, and N. Moody, *Trapping of hydrogen to lattice defects in nickel*, Modelling and Simulation in Materials Science and Engineering, 5, p. 651, 1997

EAM_AlNi_2002 (Al, Ni)

Y.Mishin, M. Mehl, and D. Papaconstantopoulos, *Embedded-atom potential for B2-NiAl*, Physical Review B, 65, p. 224114, 2002

EAM_AlNi_2004 (Al, Ni)

Y.Mishin, *Atomistic modeling of the and gamma prime-phases of the Ni–Al system*, Acta materialia, 52, pp. 1451–1467, 2004

EAM_AlNi_2009 (Al, Ni)

G.Purja Pun and Y. Mishin, *Development of an interatomic potential for the Ni-Al system*, Philosophical Magazine, 89, pp. 3245–3267, 2009

EAM_AlPb_2000 (Pb, Al)

A.Landa, P. Wynblatt, D. Siegel, J. Adams, O. Mryasov, and X. Liu, *Development of glue-type potentials for the Al–Pb system: phase diagram calculation*, Acta materialia, 48, pp. 1753–1761, 2000

EAM_AlTi_2003 (Ti, Al)

R.Zope and Y. Mishin, *Interatomic potentials for atomistic simulations of the Ti-Al system*, Physical Review B, 68, p. 024102, 2003

EAM_AlZr_Sheng_2011 (Zr, Al)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Al_1999 (Al)

Y.Mishin, D. Farkas, M. Mehl, and D. Papaconstantopoulos, *Interatomic potentials for monoatomic metals from experimental data and ab initio calculations*, Physical Review B, 59, p. 3393, 1999

EAM_Al_2003 (Al)

R.Zope and Y. Mishin, *Interatomic potentials for atomistic simulations of the Ti-Al system*, Physical Review B, 68, p. 024102, 2003

EAM_Al_2009 (Al)

J.Winey, A. Kubota, and Y. Gupta, *A thermodynamic approach to determine accurate potentials for molecular dynamics simulations: thermoelastic response of aluminum*, Modelling and Simulation in Materials Science and Engineering, 17, p. 055004, 2009

EAM_Al_2009b (Al)

V.Zhakhovskii, N. Inogamov, Y. V. Petrov, S. Ashitkov, and K. Nishihara, *Molecular dynamics simulation of femtosecond ablation and spallation with different interatomic potentials*, Applied Surface Science, 255, pp. 9592–9596, 2009

EAM_Al_Lui_2004 (Al)

X.Liu, F. Ercolessi, and J. B. Adams, *Aluminium interatomic potential from density functional theory calculations with improved stacking fault energy*, Modelling and Simulation in Materials Science and Engineering, 12, p. 665, 2004

EAM_Al_Sheng_2011 (Al)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Al_Zhou_2004 (Al)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Au_2004 (Au)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Au_2005 (Au)

G.Grochola, S. P. Russo, and I. K. Snook, *On fitting a gold embedded atom method potential using the force matching method*, The Journal of chemical physics, 123, p. 204719, 2005

EAM_Au_2009 (Au)

V.Zhakhovskii, N. Inogamov, Y. V. Petrov, S. Ashitkov, and K. Nishihara, *Molecular dynamics simulation of femtosecond ablation and spallation with different interatomic potentials*, Applied Surface Science, 255, pp. 9592–9596, 2009

EAM_Au_Olsson_2016 (Au)

A.Olsson, *Transverse resonant properties of strained gold nanowires*, Journal of Applied Physics, 108, p. 034318, 2010

EAM_Au_Sheng_2011 (Au)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Ca_Sheng_2011 (Ca)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Ce_Sheng_2011 (Ce)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Co_2004 (Co)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Co_2012 (Co)

C.Pun and Y. Mishin, *Embedded-atom potential for hcp and fcc cobalt*, Physical Review B, 86, p. 134116, 2012

EAM_CrFeNi_2013 (Ni, Cr, Fe)

G.Bonny, N. Castin, and D. Terentyev, *Interatomic potential for studying ageing under irradiation in stainless steels: the FeNiCr model alloy*, Modelling and Simulation in Materials Science and Engineering, 21, p. 085004, 2013

EAM_CuFeNi_2009 (Cu, Ni, Fe)

G.Bonny, R. C. Pasianot, N. Castin, and L. Malerba, *Ternary Fe–Cu–Ni many-body potential to model reactor pressure vessel steels: First validation by simulated thermal annealing*, Philosophical Magazine, 89, pp. 3531–3546, 2009

EAM_CuZr_Sheng_2011 (Cu, Zr)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Cu_2001b (Cu)

Y.Mishin, M. Mehl, D. Papaconstantopoulos, A. Voter, and J. Kress, *Structural stability and lattice defects in copper: Ab initio, tight-binding, and embedded-atom calculations*, Physical Review B, 63, p. 224106, 2001

EAM_Cu_2004 (Cu)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Cu_Sheng_2011 (Cu)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_FeNi_2009 (Ni, Fe)

G.Bonny, R. Pasianot, and L. Malerba, *Fe–Ni many-body potential for metallurgical applications*, Modelling and Simulation in Materials Science and Engineering, 17, p. 025010, 2009

EAM_Fe_2004 (Fe)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_HPd_2008 (H, Pd)

X.Zhou, J. Zimmerman, B. Wong, and J. Hoyt, *An embedded-atom method interatomic potential for Pd–H alloys*, Journal of Materials Research, 23, pp. 704–718, 2008

EAM_Ir_Sheng_2011 (Ir)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_MgCu_Sheng_2011 (Cu, Mg)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_MgTi_Sheng_2011 (Ti, Mg)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_MgY_Sheng_2011 (Mg, Y)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Mg_2004 (Mg)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_MoUXe_2013 (Xe, U, Mo)

D.Smirnova, A. Y. Kuksin, S. Starikov, V. Stegailov, Z. Insepov, J. Rest, and A. Yacout, *A ternary EAM interatomic potential for U–Mo alloys with xenon*, Modelling and Simulation in Materials Science and Engineering, 21, pp. 35011–35034, 2013

EAM_Mo_2004 (Mo)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Nb_2010 (Nb)

M.Fellinger, H. Park, and J. W. Wilkins, *Force-matched embedded-atom method potential for niobium*, Physical Review B, 81, p. 144119, 2010

EAM_NiP_Sheng_2011 (Ni, P)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_NiZr_Sheng_2011 (Zr, Ni)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Ni_1999 (Ni)

Y.Mishin, D. Farkas, M. Mehl, and D. Papaconstantopoulos, *Interatomic potentials for monoatomic metals from experimental data and ab initio calculations*, Physical Review B, 59, p. 3393, 1999

EAM_Ni_2004 (Ni)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Ni_Sheng_2011 (Ni)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Pb_2004 (Pb)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Pb_Sheng_2011 (Pb)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_PdSi_Sheng_2011 (Si, Pd)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Pd_Sheng_2011 (Pd)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Pd_Zhou_2004 (Pd)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Pt_2004 (Pt)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Pt_Sheng_2011 (Pt)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Rh_Sheng_2011 (Rh)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Sr_Sheng_2011 (Sr)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Ta1_2013 (Ta)

R.Ravelo, T. Germann, O. Guerrero, Q. An, and B. Holian, *Shock-induced plasticity in tantalum single crystals: Interatomic potentials and large-scale molecular-dynamics simulations*, Physical Review B, 88, p. 134101, 2013

EAM_Ta2_2013 (Ta)

R.Ravelo, T. Germann, O. Guerrero, Q. An, and B. Holian, *Shock-induced plasticity in tantalum single crystals: Interatomic potentials and large-scale molecular-dynamics simulations*, Physical Review B, 88, p. 134101, 2013

EAM_Ta_2003 (Ta)

Y.Li, D. J. Siegel, J. B. Adams, and X. Liu, *Embedded-atom-method tantalum potential developed by the force-matching method*, Physical Review B, 67, p. 125101, 2003

EAM_Ta_2004 (Ta)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Ta_Sheng_2011 (Ta)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Ti_2004 (Ti)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_U_2013 (U)

D.Smirnova, S. Starikov, and V. Stegailov, *Interatomic potential for uranium in a wide range of pressures and temperatures*, Journal of Physics: Condensed Matter, 24, p. 015702, 2012

EAM_WHHe_Bonny_1_2014 (W, H, He)

G.Bonny, P. Grigorev, and D. Terentyev, *On the binding of nanometric hydrogen–helium clusters in tungsten*, Journal of Physics: Condensed Matter, 26, p. 485001, 2014

EAM_WHHe_Bonny_2_2014 (W, H, He)

G.Bonny, P. Grigorev, and D. Terentyev, *On the binding of nanometric hydrogen–helium clusters in tungsten*, Journal of Physics: Condensed Matter, 26, p. 485001, 2014

EAM_WRe_2017 (W, Re)

G.Bonny, A. Bakaev, D. Terentyev, and Y. A. Mastrikov, *Interatomic potential to study plastic deformation in tungsten-rhenium alloys*, Journal of Applied Physics, 121, p. 165107, 2017

EAM_W_2004 (W)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Zhou_2004 (Cu, Al, Pb, Ta, Mg, Fe, Ti, Zr, Pd, Au, W, Co, Ag, Ni, Mo, Pt)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_ZrCuAg_Sheng_2011 (Cu, Zr, Ag)

T.Fujita, P. Guan, H. Sheng, A. Inoue, T. Sakurai, and M. Chen, *Coupling between chemical and dynamic heterogeneities in a multicomponent bulk metallic glass*, Physical Review B, 81, p. 140204, 2010

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_ZrCuAl_Sheng_2011 (Cu, Zr, Al)

Y.Cheng, E. Ma, and H. Sheng, *Atomic level structure in multicomponent bulk metallic glass*, Physical review letters, 102, p. 245501, 2009

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_ZrPt_Sheng_2011 (Zr, Pt)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EAM_Zr_2004 (Zr)

X.Zhou, R. Johnson, and H. Wadley, *Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers*, Physical Review B, 69, p. 144113, 2004

EAM_Zr_Sheng_2011 (Zr)

H.Sheng, M. Kramer, A. Cadien, T. Fujita, and M. Chen, *Highly optimized embedded-atom-method potentials for fourteen fcc metals*, Physical Review B, 83, p. 134118, 2011

EMT_AgCu_2000 (Cu, Ag)

T.Rasmussen, *Simulation of misfit dislocation loops at the A g/C u (111) interface*, Physical Review B, 62, p. 12664, 2000

EMT_CuAgAuNiPdPt_1996 (Cu, Pd, Au, Ag, Ni, Pt)

W.Jacobsen, P. Stoltze, and J. Nørskov, *A semi-empirical effective medium theory for metals and alloys*, Surface Science, 366, pp. 394–402, 1996

EMT_CuMg_2004 (Cu, Mg)

N.Bailey, J. Schiøtz, and K. W. Jacobsen, *Simulation of Cu-Mg metallic glass: Thermodynamics and structure*, Physical Review B, 69, p. 144205, 2004

EMT_CuZr_2007 (Cu, Zr)

A.Pduraru, A. Kenoufi, N. P. Bailey, and J. Schiøtz, *An interatomic potential for studying CuZr bulk metallic glasses*, Advanced Engineering Materials, 9, pp. 505–508, 2007

FeustonGarofalini_CaHOSi_2004 (Na, Ca, O, Si, H, Al)

B.Feuston and S. Garofalini, *Onset of polymerization in silica sols*, Chemical physics letters, 170, pp. 264–270, 1990

J.Dolado, M. Griebel, and J. Hamaekers, *A molecular dynamic study of cementitious calcium silicate hydrate (C–S–H) gels*, Journal of the American Ceramic Society, 90, pp. 3938–3942, 2007

A.Litton and S. H. Garofalini, *Modeling of hydrophilic wafer bonding by molecular dynamics simulations*, Journal of Applied Physics, 89, pp. 6013–6023, 2001

X.Su and S. H. Garofalini, *Role of nitrogen on the atomistic structure of the intergranular film in silicon nitride: A molecular dynamics study*, Journal of materials research, 19, pp. 3679–3687, 2004

GuillotSator_OSiTiAlFe2MgCaNaK_2006 (Mg, Fe, Ti, Na, K, Ca, O, Si, Al)

. Guillot and N. Sator, *A computer simulation study of natural silicate melts. Part I: Low pressure properties*, Geochimica et Cosmochimica Acta, 71, pp. 1249 - 1265, 2007 link

GuillotSator_OSiTiAlFe3MgCaNaK_2006 (Mg, Fe, Ti, Na, K, Ca, O, Si, Al)

. Guillot and N. Sator, *A computer simulation study of natural silicate melts. Part I: Low pressure properties*, Geochimica et Cosmochimica Acta, 71, pp. 1249 - 1265, 2007 link

Guillot_KNCMFAT_2007 (Mg, Fe, Ti, Na, K, Ca, O, Si, Al)

. Guillot and N. Sator, *A computer simulation study of natural silicate melts. Part I: Low pressure properties*, Geochimica et Cosmochimica Acta, 71, pp. 1249 - 1265, 2007 link

Huang_AlBKLiNaOSi_2020 (Al, B, Na, K, O, Si, Li)

S.Sundararaman, L. Huang, S. Ispas, and W. Kob, *New interaction potentials for borate glasses with mixed network formers*, The Journal of Chemical Physics, 152, p. 104501, 2020

Huang_AlCaKLiNaOSi_2019 (Al, Na, K, Ca, O, Si, Li)

S.Sundararaman, L. Huang, S. Ispas, and W. Kob, *New interaction potentials for alkali and alkaline-earth aluminosilicate glasses*, The Journal of Chemical Physics, 150, p. 154505, 2019

Huang_BCaMgOSi_2021 (B, Mg, Ca, O, Si)

Y.Shih, S. Sundararaman, S. Ispas, and L. Huang, *New interaction potentials for alkaline earth silicate and borate glasses*, Journal of Non-Crystalline Solids, 565, p. 120853, 2021

Iwasaki_AlCuNOSiTiW_2001 (Cu, Ti, N, W, O, Si, Al)

T.Iwasaki and H. Miura, *Molecular dynamics analysis of adhesion strength of interfaces between thin films*, Journal of Materials Research, 16, pp. 1789–1794, 2001 link

JacksonCatlow_AlOSi_1988 (O, Al, Si)

R.Jackson and C. R. A. Catlow, *Computer Simulation Studies of Zeolite Structure*, Molecular Simulation,, 1, pp. 207–224, 1988

Jackson_HfO_2015 (Hf, O)

M.A.Araujo, M. E. G. Valerio, and R. A. Jackson, *Computer modelling of hafnium doping in lithium niobate*, ArXiv e-prints, 2015

Jackson_LiNbO_2015 (O, Li, Nb)

M.A.Araujo, M. E. G. Valerio, and R. A. Jackson, *Computer modelling of hafnium doping in lithium niobate*, ArXiv e-prints, 2015

Jahn_AlCaMgOSi_2007 (Mg, Ca, O, Si, Al)

S.Jahn and P. A. Madden, *Modeling Earth materials from crustal to lower mantle conditions: A transferable set of interaction potentials for the CMASsystem*, Physics of the Earth and Planetary Interiors, 162, pp. 129 - 139, 2007 link

Kerisit_LiOTi3_2010 (Ti, O, Li)

S.Kerisit, K. M. Rosso, Z. Yang, and J. Liu, *Computer Simulation of the Phase Stabilities of Lithiated TiO2 Polymorphs*, The Journal of Physical Chemistry C, 114, pp. 19096-19107, 2010

Kerisit_LiOTi4_2010 (Ti, O, Li)

S.Kerisit, K. M. Rosso, Z. Yang, and J. Liu, *Computer Simulation of the Phase Stabilities of Lithiated TiO2 Polymorphs*, The Journal of Physical Chemistry C, 114, pp. 19096-19107, 2010

Leinenweber_MgSiO_1988 (O, Si, Mg)

K.Leinenweber and A. Navrotsky, *A transferable interatomic potential for crystalline phases in the system MgOSiO2*, Physics and Chemistry of Minerals, 15, pp. 588-596, 1988 link

Lusvardi_SiPNaCaOF_2008 (F, Na, Ca, O, Si, P)

G.Lusvardi, G. Malavasi, M. Cortada, L. Menabue, M. C. Menziani, A. Pedone, and U. Segre, *Elucidation of the structural role of fluorine in potentially bioactive glasses by experimental and computational investigation.*, J Phys Chem B, 112, pp. 12730-9, 2008 link

MEAM_AgAuPdPtAl_2003 (Al, Pd, Au, Ag, Pt)

B.Lee, J. Shim, and M. Baskes, *Semiempirical atomic potentials for the fcc metals Cu, Ag, Au, Ni, Pd, Pt, Al, and Pb based on first and second nearest-neighbor modified embedded atom method*, Physical Review B, 68, p. 144112, 2003

MEAM_AgAu_2022 (Ag, Au)

M.Alvi, A. Faiyad, M. A. M. Munshi, M. Motalab, M. M. Islam, and S. Saha, *Cyclic and tensile deformations of Gold–Silver core shell systems using newly parameterized MEAM potential*, Mechanics of Materials, 169, p. 104304, 2022

MEAM_AgTaO_2013 (Ta, O, Ag)

H.Gao, A. Otero-de-la-Roza, S. Aouadi, E. Johnson, and A. Martini, *An empirical model for silver tantalate*, Modelling and Simulation in Materials Science and Engineering, 21, p. 055002, 2013

MEAM_AlCu_2022 (Cu, Al)

A.Mahata, T. Mukhopadhyay, and M. A. Zaeem, *Modified embedded-atom method interatomic potentials for Al-Cu, Al-Fe and Al-Ni binary alloys: From room temperature to melting point*, Computational Materials Science, 201, p. 110902, 2022

MEAM_AlFe_2022 (Al, Fe)

A.Mahata, T. Mukhopadhyay, and M. A. Zaeem, *Modified embedded-atom method interatomic potentials for Al-Cu, Al-Fe and Al-Ni binary alloys: From room temperature to melting point*, Computational Materials Science, 201, p. 110902, 2022

MEAM_AlH_2011 (Al, H)

W.Ko, J. Shim, and B. Lee, *Atomistic modeling of the Al–H and Ni–H systems*, Journal of Materials Research, 26, pp. 1552–1560, 2011

MEAM_AlHf_2022 (Hf, Al)

R.Fereidonnejad, A. O. Moghaddam, and M. Moaddeli, *Modified embedded-atom method interatomic potentials for Al-Ti, Al-Ta, Al-Zr, Al-Nb and Al-Hf binary intermetallic systems*, Computational Materials Science, 213, p. 111685, 2022

MEAM_AlLi_2021 (Al, Li)

S.Roy, A. Dutta, and N. Chakraborti, *A novel method of determining interatomic potential for Al and Al-Li alloys and studying strength of Al-Al3Li interphase using evolutionary algorithms*, Computational Materials Science, 190, p. 110258, 2021

MEAM_AlNb_2022 (Al, Nb)

R.Fereidonnejad, A. O. Moghaddam, and M. Moaddeli, *Modified embedded-atom method interatomic potentials for Al-Ti, Al-Ta, Al-Zr, Al-Nb and Al-Hf binary intermetallic systems*, Computational Materials Science, 213, p. 111685, 2022

MEAM_AlNi_2007 (Al, Ni)

C.e Silva, J. Ågren, M. T. Clavaguera-Mora, D. Djurovic, T. Gomez-Acebo, B. Lee, Z. Liu, P. Miodownik, and H. J. Seifert, *Applications of computational thermodynamicsthe extension from phase equilibrium to phase transformations and other properties*, Calphad, 31, pp. 53–74, 2007

MEAM_AlNi_2022 (Al, Ni)

A.Mahata, T. Mukhopadhyay, and M. A. Zaeem, *Modified embedded-atom method interatomic potentials for Al-Cu, Al-Fe and Al-Ni binary alloys: From room temperature to melting point*, Computational Materials Science, 201, p. 110902, 2022

MEAM_AlSiMgCuFe_2012 (Cu, Mg, Fe, Si, Al)

B.Jelinek, S. Groh, M. F. Horstemeyer, J. Houze, S. Kim, G. J. Wagner, A. Moitra, and M. I. Baskes, *Modified embedded atom method potential for Al, Si, Mg, Cu, and Fe alloys*, Physical Review B, 85, p. 245102, 2012

MEAM_AlTa_2022 (Ta, Al)

R.Fereidonnejad, A. O. Moghaddam, and M. Moaddeli, *Modified embedded-atom method interatomic potentials for Al-Ti, Al-Ta, Al-Zr, Al-Nb and Al-Hf binary intermetallic systems*, Computational Materials Science, 213, p. 111685, 2022

MEAM_AlTi_2022 (Ti, Al)

R.Fereidonnejad, A. O. Moghaddam, and M. Moaddeli, *Modified embedded-atom method interatomic potentials for Al-Ti, Al-Ta, Al-Zr, Al-Nb and Al-Hf binary intermetallic systems*, Computational Materials Science, 213, p. 111685, 2022

MEAM_AlU_2015 (U, Al)

M.Pascuet and J. Fernández, *Atomic interaction of the MEAM type for the study of intermetallics in the Al–U alloy*, Journal of Nuclear Materials, 467, pp. 229–239, 2015

MEAM_AlVH_2013 (V, Al, H)

J.Shim, W. Ko, K. Kim, H. Lee, Y. Lee, J. Suh, Y. W. Cho, and B. Lee, *Prediction of hydrogen permeability in V–Al and V–Ni alloys*, Journal of membrane science, 430, pp. 234–241, 2013

MEAM_AlZr_2022 (Zr, Al)

R.Fereidonnejad, A. O. Moghaddam, and M. Moaddeli, *Modified embedded-atom method interatomic potentials for Al-Ti, Al-Ta, Al-Zr, Al-Nb and Al-Hf binary intermetallic systems*, Computational Materials Science, 213, p. 111685, 2022

MEAM_Al_2015 (Al)

E.Asadi, M. A. Zaeem, S. Nouranian, and M. I. Baskes, *Two-phase solid–liquid coexistence of Ni, Cu, and Al by molecular dynamics simulations using the modified embedded-atom method*, Acta Materialia, 86, pp. 169–181, 2015

MEAM_BeO_2019 (Be, O)

J.Wei, W. Zhou, S. Li, P. Shen, S. Ren, A. Hu, and W. Zhou, *Modified embedded atom method potential for modeling the thermodynamic properties of high thermal conductivity beryllium oxide*, ACS omega, 4, pp. 6339–6346, 2019

MEAM_Bi_2021 (Bi)

H.Zhou, D. E. Dickel, M. I. Baskes, S. Mun, and M. A. Zaeem, *A modified embedded-atom method interatomic potential for bismuth*, Modelling and Simulation in Materials Science and Engineering, 29, p. 065008, 2021

MEAM_CH_2014 (C, H)

S.Nouranian, M. A. Tschopp, S. R. Gwaltney, M. I. Baskes, and M. F. Horstemeyer, *An interatomic potential for saturated hydrocarbons based on the modified embedded-atom method*, Physical Chemistry Chemical Physics, 16, pp. 6233–6249, 2014

MEAM_C_2005 (C)

B.Lee and J. W. Lee, *A modified embedded atom method interatomic potential for carbon*, Calphad, 29, pp. 7–16, 2005

MEAM_CoAl_2012 (Co, Al)

W.Dong, H. Kim, W. Ko, B. Lee, and B. Lee, *Atomistic modeling of pure Co and Co–Al system*, Calphad, 38, pp. 7–16, 2012

MEAM_CoCrMnNiFe_2017 (Fe, Mn, Cr, Co, Ni)

W.Choi, Y. Kim, D. Seol, and B. Lee, *Modified embedded-atom method interatomic potentials for the Co-Cr, Co-Fe, Co-Mn, Cr-Mn and Mn-Ni binary systems*, Computational Materials Science, 130, pp. 121–129, 2017

MEAM_CoCrMn_2017 (Mn, Co, Cr)

W.Choi, Y. Kim, D. Seol, and B. Lee, *Modified embedded-atom method interatomic potentials for the Co-Cr, Co-Fe, Co-Mn, Cr-Mn and Mn-Ni binary systems*, Computational Materials Science, 130, pp. 121–129, 2017

MEAM_CoNiCrFeMn_2018 (Fe, Mn, Cr, Co, Ni)

W.Choi, Y. H. Jo, S. S. Sohn, S. Lee, and B. Lee, *Understanding the physical metallurgy of the CoCrFeMnNi high-entropy alloy: an atomistic simulation study*, npj Computational Materials, 4, pp. 1–9, 2018

MEAM_CoTiV_2020 (Ti, Co, V)

S.Oh, D. Seol, and B. Lee, *Second nearest-neighbor modified embedded-atom method interatomic potentials for the Co-M (M= Ti, V) binary systems*, Calphad, 70, p. 101791, 2020

MEAM_CuC_2021 (Cu, C)

A.Agrawal and R. Mirzaeifar, *Copper-graphene composites; developing the MEAM potential and investigating their mechanical properties*, Computational Materials Science, 188, p. 110204, 2021

MEAM_CuCoMo_2020 (Cu, Co, Mo)

J.Wang, S. Oh, and B. Lee, *Second-nearest-neighbor modified embedded-atom method interatomic potential for Cu-M (M= Co, Mo) binary systems*, Computational Materials Science, 178, p. 109627, 2020

MEAM_CuCo_2020 (Cu, Co)

J.Wang, S. Oh, and B. Lee, *Second-nearest-neighbor modified embedded-atom method interatomic potential for Cu-M (M= Co, Mo) binary systems*, Computational Materials Science, 178, p. 109627, 2020

MEAM_CuMo_2020 (Cu, Mo)

J.Wang, S. Oh, and B. Lee, *Second-nearest-neighbor modified embedded-atom method interatomic potential for Cu-M (M= Co, Mo) binary systems*, Computational Materials Science, 178, p. 109627, 2020

MEAM_CuNTi_2020 (Ti, Cu, N)

M.Miraz, N. Dhariwal, W. Meng, B. R. Ramachandran, and C. D. Wick, *Development and application of interatomic potentials to study the stability and shear strength of Ti/TiN and Cu/TiN interfaces*, Materials & Design, 196, p. 109123, 2020

MEAM_CuNi_2003 (Cu, Ni)

B.Lee, J. Shim, and M. Baskes, *Semiempirical atomic potentials for the fcc metals Cu, Ag, Au, Ni, Pd, Pt, Al, and Pb based on first and second nearest-neighbor modified embedded atom method*, Physical Review B, 68, p. 144112, 2003

MEAM_Cu_2015 (Cu)

E.Asadi, M. A. Zaeem, S. Nouranian, and M. I. Baskes, *Two-phase solid–liquid coexistence of Ni, Cu, and Al by molecular dynamics simulations using the modified embedded-atom method*, Acta Materialia, 86, pp. 169–181, 2015

MEAM_Cu_2018 (Cu)

S.Etesami and E. Asadi, *Molecular dynamics for near melting temperatures simulations of metals using modified embedded-atom method*, Journal of Physics and Chemistry of Solids, 112, pp. 61–72, 2018

MEAM_FeAl_2010 (Al, Fe)

E.Lee and B. Lee, *Modified embedded-atom method interatomic potential for the Fe–Al system*, Journal of Physics: Condensed Matter, 22, p. 175702, 2010

MEAM_FeC_2006 (C, Fe)

B.Lee, *A modified embedded-atom method interatomic potential for the Fe–C system*, Acta Materialia, 54, pp. 701 - 711, 2006 link

MEAM_FeC_2014 (C, Fe)

S.Liyanage, S. Kim, J. Houze, S. Kim, M. A. Tschopp, M. I. Baskes, and M. F. Horstemeyer, *Structural, elastic, and thermal properties of cementite (Fe 3 C) calculated using a modified embedded atom method*, Physical Review B, 89, p. 094102, 2014

MEAM_FeCrV_2001 (V, Cr, Fe)

B.Lee, M. Baskes, H. Kim, and Y. K. Cho, *Second nearest-neighbor modified embedded atom method potentials for bcc transition metals*, Physical Review B, 64, p. 184102, 2001

MEAM_FeCr_2001 (Cr, Fe)

B.Lee, J. Shim, and H. M. Park, *A semi-empirical atomic potential for the Fe-Cr binary system*, Calphad, 25, pp. 527–534, 2001

MEAM_FeCu_2005 (Cu, Fe)

B.Lee, B. D. Wirth, J. Shim, J. Kwon, S. C. Kwon, and J. Hong, *Modified embedded-atom method interatomic potential for the Fe- Cu alloy system and cascade simulations on pure Fe and Fe- Cu alloys*, Physical Review B, 71, p. 184205, 2005

MEAM_FeH_2007 (H, Fe)

B.Lee, T. Lee, and S. Kim, *A modified embedded-atom method interatomic potential for the Fe–N system: a comparative study with the Fe–C system*, Acta materialia, 54, pp. 4597–4607, 2006

MEAM_FeMnSiC_2019 (Mn, C, Si, Fe)

I.Aslam, M. Baskes, D. Dickel, S. Adibi, B. Li, H. Rhee, M. A. Zaeem, and M. Horstemeyer, *Thermodynamic and kinetic behavior of low-alloy steels: An atomic level study using an Fe-Mn-Si-C modified embedded atom method (MEAM) potential*, Materialia, 8, p. 100473, 2019

MEAM_FeMn_2009 (Mn, Fe)

Y.Kim, Y. Shin, and B. Lee, *Modified embedded-atom method interatomic potentials for pure Mn and the Fe–Mn system*, Acta Materialia, 57, pp. 474–482, 2009

MEAM_FeN_2006 (N, Fe)

B.Lee, T. Lee, and S. Kim, *A modified embedded-atom method interatomic potential for the Fe–N system: a comparative study with the Fe–C system*, Acta materialia, 54, pp. 4597–4607, 2006

MEAM_FeP_2012 (P, Fe)

W.Ko, N. J. Kim, and B. Lee, *Atomistic modeling of an impurity element and a metal–impurity system: pure P and Fe–P system*, Journal of Physics: Condensed Matter, 24, p. 225002, 2012

MEAM_FePt_2006 (Pt, Fe)

J.Kim, Y. Koo, and B. Lee, *Modified embedded-atom method interatomic potential for the Fe–Pt alloy system*, Journal of materials research, 21, pp. 199–208, 2006

MEAM_FeTiC_2009 (Ti, C, Fe)

H.Kim, W. Jung, and B. Lee, *Modified embedded-atom method interatomic potentials for the Fe–Ti–C and Fe–Ti–N ternary systems*, Acta Materialia, 57, pp. 3140 - 3147, 2009 link

MEAM_FeTi_2008 (Ti, Fe)

I.Sa and B. Lee, *Modified embedded-atom method interatomic potentials for the Fe–Nb and Fe–Ti binary systems*, Scripta Materialia, 59, pp. 595–598, 2008

MEAM_Fe_2015 (Fe)

E.Asadi, M. A. Zaeem, S. Nouranian, and M. I. Baskes, *Quantitative modeling of the equilibration of two-phase solid-liquid Fe by atomistic simulations on diffusive time scales*, Physical Review B, 91, p. 024105, 2015

MEAM_Fe_2018 (Fe)

S.Etesami and E. Asadi, *Molecular dynamics for near melting temperatures simulations of metals using modified embedded-atom method*, Journal of Physics and Chemistry of Solids, 112, pp. 61–72, 2018

MEAM_GaInN_2009 (Ga, In, N)

C.Do, Y. Shin, and B. Lee, *Atomistic modeling of III–V nitrides: modified embedded-atom method interatomic potentials for GaN, InN and Ga1- xInxN*, Journal of Physics: Condensed Matter, 21, p. 325801, 2009

MEAM_Ge_2008 (Ge)

H.Kim, Y. Shin, and B. Lee, *A modified embedded-atom method interatomic potential for Germanium*, Calphad, 32, pp. 34–42, 2008

MEAM_HfNbTaTiZr_2021 (Hf, Ta, Ti, Zr, Nb)

X.Huang, L. Liu, X. Duan, W. Liao, J. Huang, H. Sun, and C. Yu, *Atomistic simulation of chemical short-range order in HfNbTaZr high entropy alloy based on a newly-developed interatomic potential*, Materials & Design, 202, p. 109560, 2021

MEAM_In_2008 (In)

C.Do, Y. Shin, and B. Lee, *A modified embedded-atom method interatomic potential for indium*, Calphad, 32, pp. 82–88, 2008

MEAM_LiSi_2012 (Li, Si)

26.Cui, F. Gao, Z. Cui, and J. Qu, *A second nearest-neighbor embedded atom method interatomic potential for Li–Si alloys*, Journal of Power Sources, 207, pp. 150–159, 2012

MEAM_Li_2011 (Li)

26. Cui, F. Gao, Z. Cui, and J. Qu, *Developing a second nearest-neighbor modified embedded atom method interatomic potential for lithium*, Modelling and simulation in materials science and engineering, 20, p. 015014, 2011

MEAM_MgAlZn_2018 (Mg, Al, Zn)

B. Dickel, M. I. Baskes, I. Aslam, and C. D. Barrett, *New interatomic potential for Mg–Al–Zn alloys with specific application to dilute Mg-based alloys*, Modelling and simulation in materials science and engineering, 26, p. 045010, 2018

MEAM_MgAl_2009 (Mg, Al)

Y. Kim, N. J. Kim, and B. Lee, *Atomistic modeling of pure Mg and Mg–Al systems*, Calphad, 33, pp. 650–657, 2009

MEAM_MgCaZn_2019 (Ca, Zr, Mg)

H. Jang, D. Seol, and B. Lee, *Modified embedded-atom method interatomic potential for the Mg–Zn–Ca ternary system*, Calphad, 67, p. 101674, 2019

MEAM_MgCa_2015 (Ca, Mg)

K. Kim, J. B. Jeon, and B. Lee, *Modified embedded-atom method interatomic potentials for Mg–X (X= Y, Sn, Ca) binary systems*, Calphad, 48, pp. 27–34, 2015

MEAM_MgLi_2012 (Mg, Li)

Y. Kim, I. Jung, and B. Lee, *Atomistic modeling of pure Li and Mg–Li system*, Modelling and Simulation in Materials Science and Engineering, 20, p. 035005, 2012

MEAM_MgNdPb_2017 (Pb, Mg, Nd)

K. Kim and B. Lee, *Modified embedded-atom method interatomic potentials for Mg-Nd and Mg-Pb binary systems*, Calphad, 57, pp. 55–61, 2017

MEAM_MgYSn_2015 (Sn, Mg, Y)

K. Kim, J. B. Jeon, and B. Lee, *Modified embedded-atom method interatomic potentials for Mg–X (X= Y, Sn, Ca) binary systems*, Calphad, 48, pp. 27–34, 2015

MEAM_MgY_2018 (Mg, Y)

R.Ahmad, S. Groh, M. Ghazisaeidi, and W. A. Curtin, *Modified embedded-atom method interatomic potential for Mg–Y alloys*, Modelling and Simulation in Materials Science and Engineering, 26, p. 065010, 2018

MEAM_MoW_2001 (W, Mo)

B.Lee, M. Baskes, H. Kim, and Y. K. Cho, *Second nearest-neighbor modified embedded atom method potentials for bcc transition metals*, Physical Review B, 64, p. 184102, 2001

MEAM_NAlTi_2019 (Ti, Al, N)

G.Almyras, D. G. Sangiovanni, and K. Sarakinos, *Semi-empirical force-field model for the Ti1- xAlxN (0 x 1) system*, Materials, 12, p. 215, 2019

MEAM_NaSn_2020 (Sn, Na)

Y.Kim, W. Ko, and B. Lee, *Second nearest-neighbor modified embedded atom method interatomic potentials for the Na unary and Na-Sn binary systems*, Computational Materials Science, 185, p. 109953, 2020

MEAM_NbFe_2008 (Nb, Fe)

I.Sa and B. Lee, *Modified embedded-atom method interatomic potentials for the Fe–Nb and Fe–Ti binary systems*, Scripta Materialia, 59, pp. 595–598, 2008

MEAM_NbTa_2001 (Ta, Nb)

B.Lee, M. Baskes, H. Kim, and Y. K. Cho, *Second nearest-neighbor modified embedded atom method potentials for bcc transition metals*, Physical Review B, 64, p. 184102, 2001

MEAM_NiAlCo_2015 (Al, Co, Ni)

Y.Kim, W. Jung, and B. Lee, *Modified embedded-atom method interatomic potentials for the Ni–Co binary and the Ni–Al–Co ternary systems*, Modelling and Simulation in Materials Science and Engineering, 23, p. 055004, 2015

MEAM_NiAlTi_2017 (Ti, Al, Ni)

Y.Kim, H. Kim, W. Jung, and B. Lee, *Development and application of Ni-Ti and Ni-Al-Ti 2NN-MEAM interatomic potentials for Ni-base superalloys*, Computational Materials Science, 139, pp. 225–233, 2017

MEAM_NiH_2011 (Ni, H)

W.Ko, J. Shim, and B. Lee, *Atomistic modeling of the Al–H and Ni–H systems*, Journal of Materials Research, 26, pp. 1552–1560, 2011

MEAM_NiTi_2015 (Ti, Ni)

W.Ko, B. Grabowski, and J. Neugebauer, *Development and application of a Ni-Ti interatomic potential with high predictive accuracy of the martensitic phase transition*, Physical Review B, 92, p. 134107, 2015

MEAM_NiTi_2019 (Ti, Ni)

S.Kavousi, B. R. Novak, M. I. Baskes, M. A. Zaeem, and D. Moldovan, *Modified embedded-atom method potential for high-temperature crystal-melt properties of Ti–Ni alloys and its application to phase field simulation of solidification*, Modelling and Simulation in Materials Science and Engineering, 28, p. 015006, 2019

MEAM_NiVH_2013 (V, Ni, H)

J.Shim, W. Ko, K. Kim, H. Lee, Y. Lee, J. Suh, Y. W. Cho, and B. Lee, *Prediction of hydrogen permeability in V–Al and V–Ni alloys*, Journal of membrane science, 430, pp. 234–241, 2013

MEAM_NiW_2003 (W, Ni)

J.Shim, S. I. Park, Y. W. Cho, and B. Lee, *Modified embedded-atom method calculation for the Ni–W system*, Journal of materials research, 18, pp. 1863–1867, 2003

MEAM_Ni_2015 (Ni)

E.Asadi, M. A. Zaeem, S. Nouranian, and M. I. Baskes, *Two-phase solid–liquid coexistence of Ni, Cu, and Al by molecular dynamics simulations using the modified embedded-atom method*, Acta Materialia, 86, pp. 169–181, 2015

MEAM_Ni_2018 (Ni)

S.Etesami and E. Asadi, *Molecular dynamics for near melting temperatures simulations of metals using modified embedded-atom method*, Journal of Physics and Chemistry of Solids, 112, pp. 61–72, 2018

MEAM_PbSn_2018 (Sn, Pb)

S.Etesami, M. I. Baskes, M. Laradji, and E. Asadi, *Thermodynamics of solid Sn and PbSn liquid mixtures using molecular dynamics simulations*, Acta Materialia, 161, pp. 320–330, 2018

MEAM_Pb_2003 (Pb)

B.Lee, J. Shim, and M. Baskes, *Semiempirical atomic potentials for the fcc metals Cu, Ag, Au, Ni, Pd, Pt, Al, and Pb based on first and second nearest-neighbor modified embedded atom method*, Physical Review B, 68, p. 144112, 2003

MEAM_PdAlCoCuFeMoNiTi_2018 (Cu, Fe, Ti, Pd, Co, Ni, Mo, Al)

G.Jeong, C. S. Park, H. Do, S. Park, and B. Lee, *Second nearest-neighbor modified embedded-atom method interatomic potentials for the Pd-M (M= Al, Co, Cu, Fe, Mo, Ni, Ti) binary systems*, Calphad, 62, pp. 172–186, 2018

MEAM_PtAlCoCuMoNiTiV_2017 (V, Cu, Ti, Co, Ni, Mo, Pt, Al)

J.Kim, D. Seol, J. Ji, H. Jang, Y. Kim, and B. Lee, *Second nearest-neighbor modified embedded-atom method interatomic potentials for the Pt-M (M= Al, Co, Cu, Mo, Ni, Ti, V) binary systems*, Calphad, 59, pp. 131–141, 2017

MEAM_PtPdC_2020 (C, Pt, Pd)

G.Jeong and B. Lee, *Interatomic potentials for Pt-C and Pd-C systems and a study of structure-adsorption relationship in large Pt/graphene system*, Computational Materials Science, 185, p. 109946, 2020

MEAM_SiC_2014 (C, Si)

K.Kang, T. Eun, M. Jun, and B. Lee, *Governing factors for the formation of 4H or 6H-SiC polytype during SiC crystal growth: An atomistic computational approach*, Journal of Crystal Growth, 389, pp. 120–133, 2014

MEAM_SiOAu_2005 (O, Si, Au)

C.Kuo and P. Clancy, *Development of atomistic MEAM potentials for the silicon–oxygen–gold ternary system*, Modelling and Simulation in Materials Science and Engineering, 13, p. 1309, 2005

MEAM_Si_2007 (Si)

B.Lee, *A modified embedded atom method interatomic potential for silicon*, Calphad, 31, pp. 95–104, 2007

MEAM_Si_2018 (Si)

X.Huang, X. Dong, L. Liu, and P. Li, *An improved modified embedded-atom method potential to fit the properties of silicon at high temperature*, Computational Materials Science, 153, pp. 251–257, 2018

MEAM_Sn_2017 (Sn)

H.Vella, M. Chen, F. H. Stillinger, E. A. Carter, P. G. Debenedetti, and A. Z. Panagiotopoulos, *Structural and dynamic properties of liquid tin from a new modified embedded-atom method force field*, Physical Review B, 95, p. 064202, 2017

MEAM_Sn_2018 (Sn)

W.Ko, D. Kim, Y. Kwon, and M. H. Lee, *Atomistic simulations of pure tin based on a new modified embedded-atom method interatomic potential*, Metals, 8, p. 900, 2018

MEAM_TiAl_2018 (Ti, Al)

S.Sun, B. R. Ramachandran, and C. D. Wick, *Solid, liquid, and interfacial properties of TiAl alloys: parameterization of a new modified embedded atom method model*, Journal of Physics: Condensed Matter, 30, p. 075002, 2018

MEAM_TiCN_2008 (Ti, C, N)

Y.Kim and B. Lee, *Modified embedded-atom method interatomic potentials for the Ti–C and Ti–N binary systems*, Acta Materialia, 56, pp. 3481 - 3489, 2008 link

MEAM_Ti_2006 (Ti)

Y.Kim, B. Lee, and M. Baskes, *Modified embedded-atom method interatomic potentials for Ti and Zr*, Physical Review B, 74, p. 014101, 2006

MEAM_USi_2017 (U, Si)

B.Beeler, M. Baskes, D. Andersson, M. W. Cooper, and Y. Zhang, *A modified Embedded-Atom Method interatomic potential for uranium-silicide*, Journal of Nuclear Materials, 495, pp. 267–276, 2017

MEAM_UZr_2015 (Zr, U)

A.Moore, B. Beeler, C. Deo, M. Baskes, and M. Okuniewski, *Atomistic modeling of high temperature uranium–zirconium alloy structure and thermodynamics*, Journal of Nuclear Materials, 467, pp. 802–819, 2015

MEAM_U_2014 (U)

R.Fernandez and M. Pascuet, *On the accurate description of uranium metallic phases: a MEAM interatomic potential approach*, Modelling and Simulation in Materials Science and Engineering, 22, p. 055019, 2014

MEAM_VH_2011 (V, H)

J.Shim, Y. Lee, E. Fleury, Y. W. Cho, W. Ko, and B. Lee, *A modified embedded-atom method interatomic potential for the V–H system*, Calphad, 35, pp. 302–307, 2011

MEAM_VNiTi_2017 (V, Ni, Ti)

B.Maisel, W. Ko, J. Zhang, B. Grabowski, and J. Neugebauer, *Thermomechanical response of NiTi shape-memory nano-precipitates in TiV alloys*, Physical Review Materials, 1, p. 033610, 2017

MEAM_VPdY_2013 (V, Y, Pd)

W.Ko and B. Lee, *Modified embedded-atom method interatomic potentials for pure Y and the V–Pd–Y ternary system*, Modelling and Simulation in Materials Science and Engineering, 21, p. 085008, 2013

MEAM_W_2022 (W)

P.Hiremath, S. Melin, E. Bitzek, and P. A. Olsson, *Effects of interatomic potential on fracture behaviour in single-and bicrystalline tungsten*, Computational Materials Science, 207, p. 111283, 2022

MEAM_ZnMg_2018 (Zr, Mg)

H.Jang, K. Kim, and B. Lee, *Modified embedded-atom method interatomic potentials for pure Zn and Mg-Zn binary system*, Calphad, 60, pp. 200–207, 2018

MEAM_ZrAgCu_2009 (Cu, Zr, Ag)

K.Kang, I. Sa, J. Lee, E. Fleury, and B. Lee, *Atomistic modeling of the Cu–Zr–Ag bulk metallic glass system*, Scripta Materialia, 61, pp. 801–804, 2009

MEAM_ZrH_2014 (Zr, H)

B.Lee and B. Lee, *A comparative study on hydrogen diffusion in amorphous and crystalline metals using a molecular dynamics simulation*, Metallurgical and Materials Transactions A, 45, pp. 2906–2915, 2014

MEAM_Zr_2006 (Zr)

Y.Kim, B. Lee, and M. Baskes, *Modified embedded-atom method interatomic potentials for Ti and Zr*, Physical Review B, 74, p. 014101, 2006

MTP_Si_2019a (Si)

A.Shapeev, *Moment tensor potentials: A class of systematically improvable interatomic potentials*, Multiscale Modeling & Simulation, 14, pp. 1153–1173, 2016

A.Bartók, J. Kermode, N. Bernstein, and G. Csányi, *Machine learning a general-purpose interatomic potential for silicon*, Physical Review X, 8, p. 041048, 2018

MTP_Si_2019b (Si)

A.Shapeev, *Moment tensor potentials: A class of systematically improvable interatomic potentials*, Multiscale Modeling & Simulation, 14, pp. 1153–1173, 2016

A.Bartók, J. Kermode, N. Bernstein, and G. Csányi, *Machine learning a general-purpose interatomic potential for silicon*, Physical Review X, 8, p. 041048, 2018

MacKerell_HO_1998 (O, H)

A.MacKerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiórkiewicz-Kuczera, D. Yin, and M. Karplus, *All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins*, The Journal of Physical Chemistry B, 102, pp. 3586-3616, 1998

MarianGastreich_SiBNH_2000 (Si, B, H, N)

M.Marian and M. Gastreich, *A systematic theoretical study of molecular Si/N, B/N, and Si/B/N (H) compounds and parameterisation of a force-field for molecules and solids*, Journal of Molecular Structure: THEOCHEM, 506, pp. 107–129, 2000

MarianGastreich_SiBN_2003 (B, Si, N)

M.Gastreich, J. D. Gale, and C. M. Marian, *Charged-particle potential for boron nitrides, silicon nitrides, and borosilazane ceramics: Derivation of parameters and probing of capabilities*, Physical Review B, 68, p. 094110, 2003

Marrocchelli_GeO_2010 (O, Ge)

D.Marrocchelli, M. Salanne, P. Madden, C. Simon, and P. Turq, *The construction of a reliable potential for GeO2 from first principles*, Molecular Physics, 107, pp. 443-452, 2009 link

Matsui_AlCaMgOSi_1994 (Mg, Ca, O, Si, Al)

M.Matsui, *A transferable interatomic potential model for crystals and melts in the system CaO–MgO–Al2O3–SiO3*, MinMag, 58, pp. 571–572, 1994

Matsui_MgOSi_1987 (Mg, O, Si)

M.Matsui, M. Akaogi, and T. Matsumoto, *Computational model of the structural and elastic properties of the ilmenite and perovskite phases of MgSiO3*, Physics and Chemistry of Minerals, 14, pp. 101–106, 1987

Matsui_OTi_1991 (Ti, O)

M.Matsui and M. Akaogi, *Molecular Dynamics Simulation of the Structural and Physical Properties of the Four Polymorphs of TiO2*, Molecular Simulation, 6, pp. 239-244, 1991

MitchellFincham_CaF_1993 (Ca, F)

P.Mitchell and D. Fincham, *Shell model simulations by adiabatic dynamics*, Journal of Physics: Condensed Matter, 5, pp. 1031–1038, 1993

MitchellFincham_MgO_1993 (Mg, O)

P.Mitchell and D. Fincham, *Shell model simulations by adiabatic dynamics*, Journal of Physics: Condensed Matter, 5, pp. 1031–1038, 1993

MitchellFincham_NaCl_1993 (Na, Cl)

P.Mitchell and D. Fincham, *Shell model simulations by adiabatic dynamics*, Journal of Physics: Condensed Matter, 5, pp. 1031–1038, 1993

Moliere_CSiF_1999 (F, C, Si)

C.Abrams and D. B. Graves, *Molecular dynamics simulations of Si etching by energetic CF 3+*, Journal of applied physics, 86, pp. 5938–5948, 1999

Moliere_ClFSi_2003 (F, Si, Cl)

D.Humbird and D. B. Graves, *Improved interatomic potentials for silicon–fluorine and silicon–chlorine*, The Journal of Chemical Physics, 120, pp. 2405-2412, 2004 link

Murty_HSi_1995 (H, Si)

M.W.Murty and H. A. Atwater, *Empirical interaction potential for Si-H interactions*, Phys. Rev. B, 51, pp. 4889–4893, 1995

Oligschleger_Se_1996 (Se)

C.Oligschleger, R. O. Jones, S. M. Reimann, and H. R. Schober, *Model interatomic potential for simulations in selenium*, Phys. Rev. B, 53, pp. 6165–6173, 1996 link

Pedone_2006Fe2 (Cu, Be, Gd, Zr, Na, O, Nd, Ni, Si, Co, Li, Er, Sn, Ca, Ge, Al, Ba, Cr, K, Sc, P, Zn, Mg, Fe, Ti, Mn, Ag, Sr)

A.Pedone, G. Malavasi, M. Menziani, A. Cormack, and U. Segre, *A new self-consistent empirical interatomic potential model for oxides, silicates and silica-based glasses*, J. Phys. Chem. B, 110, pp. 11780–11795, 2006

Pedone_2006Fe3 (Cu, Be, Gd, Zr, Na, O, Nd, Ni, Si, Co, Li, Er, Sn, Ca, Ge, Al, Ba, Cr, K, Sc, P, Zn, Mg, Fe, Ti, Mn, Ag, Sr)

A.Pedone, G. Malavasi, M. Menziani, A. Cormack, and U. Segre, *A new self-consistent empirical interatomic potential model for oxides, silicates and silica-based glasses*, J. Phys. Chem. B, 110, pp. 11780–11795, 2006

Pedone_AlCaNaOSi_2012 (Na, Ca, O, Si, Al)

A.Pedone, E. Gambuzzi, and M. C. Menziani, *Unambiguous Description of the Oxygen Environment in Multicomponent Aluminosilicate Glasses from 17O Solid State NMR Computational Spectroscopy*, The Journal of Physical Chemistry C, 116, pp. 14599-14609, 2012

Pedone_LiNaKSiO_2007 (Na, K, O, Si, Li)

A.Pedone, G. Malavasi, A. N. Cormack, U. Segre, and M. C. Menziani, *Insight into elastic properties of binary alkali silicate glasses; prediction and interpretation through atomistic simulation techniques*, Chemistry of materials, 19, pp. 3144–3154, 2007

Pedone_NaLiCaMgBSiO_2022 (Mg, B, Na, Ca, O, Si, Li)

M.Bertani, A. Pallini, M. Cocchi, M. C. Menziani, and A. Pedone, *A new self-consistent empirical potential model for multicomponent borate and borosilicate glasses*, Journal of the American Ceramic Society, 2022

Pedone_SiAlPNaKLiCaO_2021 (Al, Na, K, Ca, O, Si, P, Li)

M.Bertani, M. C. Menziani, and A. Pedone, *Improved empirical force field for multicomponent oxide glasses and crystals*, Physical Review Materials, 5, p. 045602, 2021

Pinilla_HO_2012 (O, H)

C.Pinilla, A. H. Irani, N. Seriani, and S. Scandolo, *Ab initio parameterization of an all-atom polarizable and dissociable force field for water*, The Journal of Chemical Physics, 136, p., 2012 link

ReaxFF_CHArHeNeKr_2016 (Kr, Ar, C, H, He, Ne)

K.Yoon, A. Rahnamoun, J. L. Swett, V. Iberi, D. A. Cullen, I. V. Vlassiouk, A. Belianinov, S. Jesse, X. Sang, O. S. Ovchinnikova, and others, *Atomistic-scale simulations of defect formation in graphene under noble gas ion irradiation*, ACS nano, 10, pp. 8376–8384, 2016

ReaxFF_CHBN_2015 (C, B, H, N)

S.Pai, B. C. Yeo, and S. S. Han, *Development of the ReaxFF CBN reactive force field for the improved design of liquid CBN hydrogen storage materials*, Physical Chemistry Chemical Physics, 18, pp. 1818–1827, 2016

ReaxFF_CHF (F, C, H)

. Oliver Böhm, *CHF parameter set Version 4.6*, 2015

ReaxFF_CHFe_2016 (C, H, Fe)

M.Islam, C. Zou, A. C. van Duin, and S. Raman, *Interactions of hydrogen with the iron and iron carbide interfaces: a ReaxFF molecular dynamics study*, Physical Chemistry Chemical Physics, 18, pp. 761–771, 2016

ReaxFF_CHGa_2021 (Ga, C, H)

S.Rajabpour, Q. Mao, N. Nayir, J. A. Robinson, and A. C. Van Duin, *Development and Applications of ReaxFF Reactive Force Fields for Group-III Gas-Phase Precursors and Surface Reactions with Graphene in Metal–Organic Chemical Vapor Deposition Synthesis*, The Journal of Physical Chemistry C, 125, pp. 10747–10758, 2021

ReaxFF_CHIn_2021 (C, H, In)

S.Rajabpour, Q. Mao, N. Nayir, J. A. Robinson, and A. C. Van Duin, *Development and Applications of ReaxFF Reactive Force Fields for Group-III Gas-Phase Precursors and Surface Reactions with Graphene in Metal–Organic Chemical Vapor Deposition Synthesis*, The Journal of Physical Chemistry C, 125, pp. 10747–10758, 2021

ReaxFF_CHNa_2016 (C, Na, H)

E.Hjertenæs, A. Q. Nguyen, and H. Koch, *A ReaxFF force field for sodium intrusion in graphitic cathodes*, Physical Chemistry Chemical Physics, 18, pp. 31431–31440, 2016

ReaxFF_CHOAlGe_2017 (C, O, H, Ge, Al)

Y.Zheng, S. Hong, G. Psofogiannakis, G. B. Rayner Jr, S. Datta, A. C. van Duin, and R. Engel-Herbert, *Modeling and in situ probing of surface reactions in atomic layer deposition*, ACS applied materials & interfaces, 9, pp. 15848–15856, 2017

ReaxFF_CHOAl_2016 (C, O, Al, H)

S.Hong and A. C. van Duin, *Atomistic-scale analysis of carbon coating and its effect on the oxidation of aluminum nanoparticles by ReaxFF-molecular dynamics simulations*, The Journal of Physical Chemistry C, 120, pp. 9464–9474, 2016

ReaxFF_CHOCaSiAlS_2012 (S, C, Ca, O, Si, H, Al)

L.Liu, A. Jaramillo-Botero, W. A. Goddard III, and H. Sun, *Development of a ReaxFF reactive force field for ettringite and study of its mechanical failure modes from reactive dynamics simulations*, The Journal of Physical Chemistry A, 116, pp. 3918–3925, 2012

ReaxFF_CHOCsKNaClIFLi_2019 (F, I, C, Na, Cs, K, O, H, Li, Cl)

M.Fedkin, Y. K. Shin, N. Dasgupta, J. Yeon, W. Zhang, D. Van Duin, A. C. van Duin, K. Mori, A. Fujiwara, M. Machida, and others, *Development of the ReaxFF Methodology for Electrolyte–Water Systems*, The Journal of Physical Chemistry A, 123, pp. 2125–2141, 2019

ReaxFF_CHOFeSCr_2015 (S, Fe, C, Cr, O, H)

K.Shin, H. Kwak, A. V. Vasenkov, D. Sengupta, and A. C. van Duin, *Development of a ReaxFF reactive force field for Fe/Cr/O/S and application to oxidation of butane over a pyrite-covered Cr2O3 catalyst*, ACS Catalysis, 5, pp. 7226–7236, 2015

ReaxFF_CHOFe_2010 (C, O, H, Fe)

M.Aryanpour, A. C. T. van Duin, and J. D. Kubicki, *Development of a Reactive Force Field for IronOxyhydroxide Systems*, The Journal of Physical Chemistry A, 114, pp. 6298-6307, 2010

ReaxFF_CHOGe_2018 (C, O, H, Ge)

N.Nayir, A. C. Van Duin, and S. Erkoc, *Development of a ReaxFF reactive force field for interstitial oxygen in germanium and its application to GeO2/Ge interfaces*, The Journal of Physical Chemistry C, 123, pp. 1208–1218, 2018

ReaxFF_CHONBAl_2022 (B, C, N, O, H, Al)

A.Lele, P. Krstic, and A. C. van Duin, *ReaxFF Force Field Development for Gas-Phase hBN Nanostructure Synthesis*, The Journal of Physical Chemistry A, 2022

ReaxFF_CHONSFPtClNi_2010 (F, S, C, N, O, Ni, H, Pt, Cl)

H.Mueller, A. C. T. van Duin, and W. A. Goddard, *Development and Validation of ReaxFF Reactive Force Field for Hydrocarbon Chemistry Catalyzed by Nickel*, The Journal of Physical Chemistry C, 114, pp. 4939-4949, 2010

ReaxFF_CHONSFe_2021 (S, Fe, C, N, O, H)

E.Moerman, D. Furman, and D. J. Wales, *Development of ReaxFF Reactive Force Field for Aqueous Iron–Sulfur Clusters with Applications to Stability and Reactivity in Water*, Journal of chemical information and modeling, 61, pp. 1204–1214, 2021

ReaxFF_CHONSMgPNaCuCl_2013 (Cu, Mg, S, C, Na, N, O, H, P, Cl)

S.Monti, C. Li, and V. Carravetta, *Reactive dynamics simulation of monolayer and multilayer adsorption of glycine on Cu (110)*, The Journal of Physical Chemistry C, 117, pp. 5221–5228, 2013

ReaxFF_CHONSMgPNaCuCl_2013_2 (Cu, Mg, S, C, Na, N, O, H, P, Cl)

S.Monti, A. Corozzi, P. Fristrup, K. L. Joshi, Y. K. Shin, P. Oelschlaeger, A. C. Van Duin, and V. Barone, *Exploring the conformational and reactive dynamics of biomolecules in solution using an extended version of the glycine reactive force field*, Physical Chemistry Chemical Physics, 15, pp. 15062–15077, 2013

ReaxFF_CHONSMgPNaCuCl_2018 (Cu, Mg, S, C, Na, N, O, H, P, Cl)

W.Zhang and A. C. Van Duin, *Improvement of the ReaxFF description for functionalized hydrocarbon/water weak interactions in the condensed phase*, The Journal of Physical Chemistry B, 122, pp. 4083–4092, 2018

ReaxFF_CHONSMgPNaCuCl_2018_08 (Cu, Mg, S, C, Na, N, O, H, P, Cl)

A.Vashisth, C. Ashraf, W. Zhang, C. E. Bakis, and A. C. van Duin, *Accelerated ReaxFF simulations for describing the reactive cross-linking of polymers*, The Journal of Physical Chemistry A, 122, pp. 6633–6642, 2018

ReaxFF_CHONSMgPNaCu_2013 (Cu, Mg, S, C, Na, N, O, H, P)

L.Huang, T. Bandosz, K. L. Joshi, A. C. Van Duin, and K. E. Gubbins, *Reactive adsorption of ammonia and ammonia/water on CuBTC metal-organic framework: A ReaxFF molecular dynamics simulation*, The Journal of chemical physics, 138, p. 034102, 2013

ReaxFF_CHONSMgPNaTiClFAu_2016 (F, Mg, S, Ti, C, Na, N, Au, O, H, P, Cl)

S.Monti, V. Carravetta, and H. Ågren, *Simulation of gold functionalization with cysteine by reactive molecular dynamics*, The journal of physical chemistry letters, 7, pp. 272–276, 2016

ReaxFF_CHONSMgPNaTiClFKLi_2020 (F, Mg, S, Ti, C, Na, N, K, O, H, P, Li, Cl)

K.Ganeshan, Y. K. Shin, N. C. Osti, Y. Sun, K. Prenger, M. Naguib, M. Tyagi, E. Mamontov, D. Jiang, and A. C. Van Duin, *Structure and dynamics of aqueous electrolytes confined in 2D-TiO2/Ti3C2T2 MXene heterostructures*, ACS Applied Materials & Interfaces, 12, pp. 58378–58389, 2020

ReaxFF_CHONSMgPNaTiClF_2013 (F, Mg, S, Ti, C, Na, N, O, H, P, Cl)

S.Kim and A. C. Van Duin, *Simulation of titanium metal/titanium dioxide etching with chlorine and hydrogen chloride gases using the ReaxFF reactive force field*, The Journal of Physical Chemistry A, 117, pp. 5655–5663, 2013

ReaxFF_CHONSMgPNaTiClF_2013_2 (F, Mg, S, Ti, C, Na, N, O, H, P, Cl)

S.Kim, A. C. van Duin, and J. D. Kubicki, *Molecular dynamics simulations of the interactions between TiO2 nanoparticles and water with Na+ and Cl-, methanol, and formic acid using a reactive force field*, Journal of Materials Research, 28, p. 513, 2013

ReaxFF_CHONSSiCaCsKSrNaMgAlCu_2015 (Cu, Mg, S, C, Na, Cs, N, K, Ca, O, H, Si, Sr, Al)

M.Psofogiannakis, J. F. McCleerey, E. Jaramillo, and A. C. van Duin, *ReaxFF Reactive Molecular Dynamics Simulation of the Hydration of Cu-SSZ-13 Zeolite and the Formation of Cu Dimers*, The Journal of Physical Chemistry C, 119, pp. 6678–6686, 2015

ReaxFF_CHONSSiCaCsKSrNaMgAl_2014 (Mg, S, C, Na, Cs, N, K, Ca, O, H, Si, Sr, Al)

K.Joshi, G. Psofogiannakis, A. C. Van Duin, and S. Raman, *Reactive molecular simulations of protonation of water clusters and depletion of acidity in H-ZSM-5 zeolite*, Physical Chemistry Chemical Physics, 16, pp. 18433–18441, 2014

ReaxFF_CHONSSiGe_2016 (S, C, N, O, H, Si, Ge)

G.Psofogiannakis and A. C. van Duin, *Development of a ReaxFF reactive force field for Si/Ge/H systems and application to atomic hydrogen bombardment of Si, Ge, and SiGe (100) surfaces*, Surface Science, 646, pp. 253 - 260, 2016 link

ReaxFF_CHONSSiLi_2013 (S, C, N, O, Si, H, Li)

Q.A/S, *Atomistix ToolKit 2014 Reference Manual*, 2014

ReaxFF_CHONSSiNaAl_2012 (S, C, Na, N, O, H, Si, Al)

C.Bai, L. Liu, and H. Sun, *Molecular dynamics simulations of methanol to olefin reactions in HZSM-5 zeolite using a reaxff force field*, The Journal of Physical Chemistry C, 116, pp. 7029–7039, 2012

ReaxFF_CHONSSiNaP_2014 (S, C, Na, N, O, H, Si, P)

B.Zhang, A. C. van Duin, and J. K. Johnson, *Development of a ReaxFF reactive force field for tetrabutylphosphonium glycinate/CO2 mixtures*, The Journal of Physical Chemistry B, 118, pp. 12008–12016, 2014

ReaxFF_CHONSSiPtZrNiCuCoHeNeArKrXe_2010 (Cu, Kr, Ar, S, C, Zr, N, Xe, O, Ni, H, Si, Co, He, Pt, Ne)

M.Kamat, A. C. Van Duin, and A. Yakovlev, *Molecular dynamics simulations of laser-induced incandescence of soot using an extended ReaxFF reactive force field*, The Journal of Physical Chemistry A, 114, pp. 12561–12572, 2010

ReaxFF_CHONSSiPtZrNiCuCo_2005 (Cu, S, C, Zr, N, O, Ni, H, Si, Co, Pt)

K.Nielson, A. C. van Duin, J. Oxgaard, W. Deng, and W. A. Goddard, *Development of the ReaxFF reactive force field for describing transition metal catalyzed reactions, with application to the initial stages of the catalytic formation of carbon nanotubes*, The Journal of Physical Chemistry A, 109, pp. 493–499, 2005

ReaxFF_CHONSSiPt_2009 (S, C, N, O, H, Si, Pt)

L.Zhang, A. C. V. Duin, S. V. Zybin, and W. A. Goddard Iii, *Thermal decomposition of hydrazines from reactive dynamics using the ReaxFF reactive force field*, The Journal of Physical Chemistry B, 113, pp. 10770–10778, 2009

ReaxFF_CHONSSi_2009 (S, C, N, O, H, Si)

L.Zhang, S. V. Zybin, A. C. van Duin, S. Dasgupta, W. A. Goddard III, and E. M. Kober, *Carbon cluster formation during thermal decomposition of octahydro-1, 3, 5, 7-tetranitro-1, 3, 5, 7-tetrazocine and 1, 3, 5-triamino-2, 4, 6-trinitrobenzene high explosives from ReaxFF reactive molecular dynamics simulations*, The Journal of Physical Chemistry A, 113, pp. 10619–10640, 2009

ReaxFF_CHONSSi_2011 (S, C, N, O, Si, H)

L.Liu, Y. Liu, S. V. Zybin, H. Sun, and W. A. Goddard, *ReaxFF-lg: Correction of the ReaxFF Reactive Force Field for London Dispersion, with Applications to the Equations of State for Energetic Materials*, The Journal of Physical Chemistry A, 115, pp. 11016-11022, 2011

ReaxFF_CHONSSi_2012 (S, C, N, O, H, Si)

P.Kulkarni, D. G. Truhlar, S. Goverapet Srinivasan, A. C. van Duin, P. Norman, and T. E. Schwartzentruber, *Oxygen interactions with silica surfaces: Coupled cluster and density functional investigation and the development of a new ReaxFF potential*, The Journal of Physical Chemistry C, 117, pp. 258–269, 2012

ReaxFF_CHONSSi_2012_2 (S, C, N, O, H, Si)

D.Newsome, D. Sengupta, H. Foroutan, M. F. Russo, and A. C. van Duin, *Oxidation of Silicon Carbide by O2 and H2O: A ReaxFF Reactive Molecular Dynamics Study, Part I*, The Journal of Physical Chemistry C, 116, pp. 16111–16121, 2012

ReaxFF_CHONSSi_2018 (S, C, N, O, H, Si)

F.Soria, W. Zhang, P. A. Paredes-Olivera, A. C. Van Duin, and E. M. Patrito, *Si/C/H ReaxFF reactive potential for silicon surfaces grafted with organic molecules*, The Journal of Physical Chemistry C, 122, pp. 23515–23527, 2018

ReaxFF_CHONSZr_2020 (S, C, Zr, N, O, H)

S.Dwivedi, M. Kowalik, N. Rosenbach, D. S. Alqarni, Y. K. Shin, Y. Yang, J. C. Mauro, A. Tanksale, A. L. Chaffee, and A. C. Van Duin, *Atomistic Mechanisms of Thermal Transformation in a Zr-Metal Organic Framework, MIL-140C*, The Journal of Physical Chemistry Letters, 12, pp. 177–184, 2020

ReaxFF_CHONS_2010 (S, C, N, O, H)

T.Mattsson, J. M. D. Lane, K. R. Cochrane, M. P. Desjarlais, A. P. Thompson, F. Pierce, and G. S. Grest, *First-principles and classical molecular dynamics simulation of shocked polymers*, Phys. Rev. B, 81, p. 054103, 2010 link

ReaxFF_CHONSiCuAgZn_2015 (Cu, C, N, O, Ag, Si, H, Zn)

A.Lloyd, D. Cornil, A. Van Duin, D. van Duin, R. Smith, S. D. Kenny, J. Cornil, and D. Beljonne, *Development of a ReaxFF potential for Ag/Zn/O and application to Ag deposition on ZnO*, Surface Science, 645, pp. 67–73, 2016

ReaxFF_CHONSiPtZrYBaTi_2013 (Ba, Y, Ti, C, Zr, N, O, Si, H, Pt)

S.Naserifar, L. Liu, W. A. Goddard III, T. T. Tsotsis, and M. Sahimi, *Toward a Process-Based Molecular Model of SiC Membranes. 1. Development of a Reactive Force Field*, The Journal of Physical Chemistry C, 117, pp. 3308–3319, 2013

ReaxFF_CHONSi_2020 (C, N, O, Si, H)

Y.Wang, Y. Shi, Q. Sun, K. Lu, M. Kubo, and J. Xu, *Development of a Transferable ReaxFF Parameter Set for Carbon-and Silicon-Based Solid Systems*, The Journal of Physical Chemistry C, 124, pp. 10007–10015, 2020

ReaxFF_CHONTi_2012 (Ti, C, N, O, H)

A.Jaramillo-Botero, Q. An, M. Cheng, W. A. Goddard III, L. W. Beegle, and R. Hodyss, *Hypervelocity Impact Effect of Molecules from Enceladus Plume and Titans Upper Atmosphere on NASAs Cassini Spectrometer from Reactive Dynamics Simulation*, Physical review letters, 109, p. 213201, 2012

ReaxFF_CHONZrYBa_2008 (Ba, Y, C, Zr, N, O, H)

C.Van Duin, B. V. Merinov, S. S. Jang, and W. A. Goddard, *ReaxFF reactive force field for solid oxide fuel cell systems with application to oxygen ion transport in yttria-stabilized zirconia*, The Journal of Physical Chemistry A, 112, pp. 3133–3140, 2008

ReaxFF_CHON_2003 (C, O, H, N)

A.Strachan, A. C. van Duin, D. Chakraborty, S. Dasgupta, and W. A. Goddard III, *Shock waves in high-energy materials: The initial chemical events in nitramine RDX*, Physical Review Letters, 91, p. 098301, 2003

ReaxFF_CHON_2009 (C, O, H, N)

J.Budzien, A. P. Thompson, and S. V. Zybin, *Reactive molecular dynamics simulations of shock through a single crystal of pentaerythritol tetranitrate*, The Journal of Physical Chemistry B, 113, pp. 13142–13151, 2009

ReaxFF_CHON_2010 (C, O, H, N)

O.Rahaman, A. C. Van Duin, W. A. Goddard III, and D. J. Doren, *Development of a ReaxFF reactive force field for glycine and application to solvent effect and tautomerization*, The Journal of Physical Chemistry B, 115, pp. 249–261, 2010

ReaxFF_CHON_2019 (C, O, H, N)

M. Kowalik, C. Ashraf, B. Damirchi, D. Akbarian, S. Rajabpour, and A. C. Van Duin, *Atomistic scale analysis of the carbonization process for C/H/O/N-based polymers with the ReaxFF reactive force field*, The Journal of Physical Chemistry B, 123, pp. 5357–5367, 2019

ReaxFF_CHONaCaMg_2022 (Mg, C, Na, Ca, O, H)

N. Dasgupta, C. Chen, and A. C. Van Duin, *Development and application of ReaxFF methodology for understanding the chemical dynamics of metal carbonates in aqueous solutions*, Physical Chemistry Chemical Physics, 24, pp. 3322–3337, 2022

ReaxFF_CHONi_2015 (C, O, Ni, H)

F. Tavazza, T. Senftle, C. Zou, C. Becker, and A. T. van Duin, *Molecular Dynamics Investigation of the Effects of Tip–Substrate Interactions during Nanoindentation*, The Journal of Physical Chemistry C, 119, pp. 13580–13589, 2015

ReaxFF_CHOSCuCl_2018 (Cu, S, C, O, H, Cl)

J. Yeon, H. L. Adams, C. E. Junkermeier, A. C. van Duin, W. T. Tysoe, and A. Martini, *Development of a ReaxFF force field for Cu/S/C/H and reactive MD simulations of methyl thiolate decomposition on Cu (100)*, The Journal of Physical Chemistry B, 122, pp. 888–896, 2017

ReaxFF_CHOSFClN_2014 (F, S, C, N, O, H, Cl)

M. Wood, A. C. van Duin, and A. Strachan, *Coupled thermal and electromagnetic induced decomposition in the molecular explosive HMX; a reactive molecular dynamics study*, The Journal of Physical Chemistry A, 118, pp. 885–895, 2014

ReaxFF_CHOSMoNiAuTi_2022 (S, Ti, C, Au, O, Ni, H, Mo)

Q. Mao, Y. Zhang, M. Kowalik, N. Nayir, M. Chandross, and A. C. T. van Duin, *Oxidation and hydrogenation of monolayer MoS2 with compositing agent under environmental exposure: The ReaxFF Mo/Ti/Au/O/S/H force field development and applications*, Frontiers in Nanotechnology, 4, 2022 link

ReaxFF_CHOSMoNiLiBFPN_2021 (F, B, C, N, O, Ni, H, Mo, P, Li)

Y. Liu, Q. Sun, P. Yu, Y. Wu, L. Xu, H. Yang, M. Xie, T. Cheng, and W. A. Goddard III, *Effects of High and Low Salt Concentrations in Electrolytes at Lithium–Metal Anode Surfaces Using DFT-ReaxFF Hybrid Molecular Dynamics Method*, The Journal of Physical Chemistry Letters, 12, pp. 2922–2929, 2021

ReaxFF_CHOS_2016 (C, O, H, S)

J.Müller and B. Hartke, *ReaxFF reactive force field for disulfide mechanochemistry, fitted to multireference ab initio data*, Journal of chemical theory and computation, 12, pp. 3913–3925, 2016

ReaxFF_CHOSiLiF_2017 (F, C, O, Si, H, Li)

K.Yun, S. J. Pai, B. C. Yeo, K. Lee, S. Kim, and S. S. Han, *Simulation Protocol for Prediction of a Solid-Electrolyte Interphase on the Silicon-based Anodes of a Lithium-Ion Battery: ReaxFF Reactive Force Field*, The journal of physical chemistry letters, 8, pp. 2812–2818, 2017

ReaxFF_CHOSiNa_2018 (C, Na, O, Si, H)

S.Hahn, J. Rimsza, L. Criscenti, W. Sun, L. Deng, J. Du, T. Liang, S. B. Sinnott, and A. C. Van Duin, *Development of a ReaxFF reactive force field for NaSiO x/water systems and its application to sodium and proton self-diffusion*, The Journal of Physical Chemistry C, 122, pp. 19613–19624, 2018

ReaxFF_CHOSi_2005 (Si, C, O, H)

K.Chenoweth, S. Cheung, A. C. Van Duin, W. A. Goddard, and E. M. Kober, *Simulations on the thermal decomposition of a poly (dimethylsiloxane) polymer using the ReaxFF reactive force field*, Journal Of The American Chemical Society, 127, pp. 7192–7202, 2005

ReaxFF_CHOV_2008 (V, C, O, H)

K.Chenoweth, A. C. Van Duin, P. Persson, M. Cheng, J. Oxgaard, and W. A. Goddard Iii, *Development and application of a ReaxFF reactive force field for oxidative dehydrogenation on vanadium oxide catalysts*, The Journal of Physical Chemistry C, 112, pp. 14645–14654, 2008

ReaxFF_CHO_2008 (C, O, H)

K.Chenoweth, A. C. van Duin, and W. A. Goddard, *ReaxFF reactive force field for molecular dynamics simulations of hydrocarbon oxidation*, The Journal of Physical Chemistry A, 112, pp. 1040–1053, 2008

ReaxFF_CHO_2016 (C, O, H)

C.Ashraf and A. C. van Duin, *Extension of the ReaxFF combustion force field toward syngas combustion and initial oxidation kinetics*, The Journal of Physical Chemistry A, 121, pp. 1051–1068, 2017

ReaxFF_CHO_2017 (C, O, H)

R.Smith, K. Jolley, C. Latham, M. Heggie, A. van Duin, D. van Duin, and H. Wu, *A ReaXFF carbon potential for radiation damage studies*, Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms, 393, pp. 49–53, 2017

ReaxFF_CHO_2022 (C, O, H)

M.Kaski, S. Hrabar, A. C. van Duin, and Z. Postawa, *Development of a Charge-Implicit ReaxFF for C/H/O Systems*, The journal of physical chemistry letters, 13, pp. 628–633, 2022

ReaxFF_CHPt_2008 (C, Pt, H)

C.Sanz-Navarro, P. Åstrand, D. Chen, M. Rønning, A. C. Van Duin, T. Jacob, and W. A. Goddard, *Molecular dynamics simulations of the interactions between platinum clusters and carbon platelets*, The Journal of Physical Chemistry A, 112, pp. 1392–1402, 2008

ReaxFF_CHSAu_2011 (C, H, S, Au)

T.Järvi, A. C. Van Duin, K. Nordlund, and W. A. Goddard III, *Development of interatomic reaxff potentials for Au–S–C–H systems*, The Journal of Physical Chemistry A, 115, pp. 10315–10322, 2011

ReaxFF_CH_2017 (C, H)

Q.Mao, Y. Ren, K. Luo, and A. C. van Duin, *Dynamics and kinetics of reversible homo-molecular dimerization of polycyclic aromatic hydrocarbons*, The Journal of chemical physics, 147, p. 244305, 2017

ReaxFF_CH_2018 (C, H)

M.Kaski, D. Maciazek, Z. Postawa, C. M. Ashraf, A. C. Van Duin, and B. J. Garrison, *Development of a charge-implicit ReaxFF potential for hydrocarbon systems*, The Journal of Physical Chemistry Letters, 9, pp. 359–363, 2018

ReaxFF_C_2015 (C)

S.Srinivasan, A. C. van Duin, and P. Ganesh, *Development of a ReaxFF potential for carbon condensed phases and its application to the thermal fragmentation of a large fullerene*, The Journal of Physical Chemistry A, 119, pp. 571–580, 2015

ReaxFF_Co_2014 (Co)

X. Zhang, E. Iype, S. V. Nedea, A. P. Jansen, B. M. Szyja, E. J. Hensen, and R. A. van Santen, *Site stability on cobalt nanoparticles: a molecular dynamics ReaxFF reactive force field study*, The Journal of Physical Chemistry C, 118, pp. 6882–6886, 2014

ReaxFF_CuZr_2019 (Cu, Zr)

H. Huang, L. Ai, A. Van Duin, M. Chen, and Y. Lü, *ReaxFF reactive force field for molecular dynamics simulations of liquid Cu and Zr metals*, The Journal of Chemical Physics, 151, p. 094503, 2019

ReaxFF_GaN (Ga, N)

. Oliver Böhm, *GaN parameter set Version 4.6*, 2015

ReaxFF_HOAu_2010 (O, H, Au)

A. Keith, D. Fantauzzi, T. Jacob, and A. C. T. van Duin, *Reactive forcefield for simulating gold surfaces and nanoparticles*, Phys. Rev. B, 81, p. 235404, 2010 link

ReaxFF_HOCuCl_2010 (Cu, O, H, Cl)

C. Van Duin, V. S. Bryantsev, M. S. Diallo, W. A. Goddard, O. Rahaman, D. J. Doren, D. Raymand, and K. Hermansson, *Development and validation of a ReaxFF reactive force field for Cu cation/water interactions and copper metal/metal oxide/metal hydroxide condensed phases*, The Journal of Physical Chemistry A, 114, pp. 9507–9514, 2010

ReaxFF_HONB_2010 (B, O, H, N)

M. Weismiller, A. C. T. V. Duin, J. Lee, and R. A. Yetter, *ReaxFF Reactive Force Field Development and Applications for Molecular Dynamics Simulations of Ammonia Borane Dehydrogenation and Combustion*, The Journal of Physical Chemistry A, 114, pp. 5485-5492, 2010

ReaxFF_HONiYZr_2019 (Y, Zr, O, Ni, H)

S. Liu, L. C. Saha, A. Iskandarov, T. Ishimoto, T. Yamamoto, Y. Umeno, S. Matsumura, and M. Koyama, *Atomic structure observations and reaction dynamics simulations on triple phase boundaries in solid-oxide fuel cells*, Communications Chemistry, 2, pp. 1–9, 2019

ReaxFF_HOSiAlLi_2012 (Al, O, Si, H, Li)

B.Narayanan, A. C. van Duin, B. B. Kappes, I. E. Reimanis, and C. V. Ciobanu, *A reactive force field for lithium–aluminum silicates with applications to eucryptite phases*, Modelling and Simulation in Materials Science and Engineering, 20, p. 015002, 2012

ReaxFF_HOSiAlLi_2016 (O, Si, H, Li, Al)

A.Ostadhossein, S. Kim, E. D. Cubuk, Y. Qi, and A. C. van Duin, *Atomic insight into the lithium storage and diffusion mechanism of SiO2/Al2O3 electrodes of lithium ion batteries: ReaxFF reactive force field modeling*, The Journal of Physical Chemistry A, 120, pp. 2114–2127, 2016

ReaxFF_HOSi_2019 (Si, O, H)

N.Nayir, A. C. Van Duin, and S. Erkoc, *Development of the reaxff reactive force field for inherent point defects in the si/silica system*, The Journal of Physical Chemistry A, 123, pp. 4303–4313, 2019

ReaxFF_HOZn_2010 (O, H, Zn)

D.Raymand, A. C. van Duin, D. Spångberg, W. A. G. III, and K. Hermansson, *Water adsorption on stepped ZnO surfaces from MDsimulation*, Surface Science, 604, pp. 741 - 752, 2010 link

ReaxFF_HO_2017 (O, H)

W.Zhang and A. C. van Duin, *Second-generation ReaxFF water force field: improvements in the description of water density and OH-anion diffusion*, The Journal of Physical Chemistry B, 121, pp. 6021–6032, 2017

ReaxFF_HPd_2014 (H, Pd)

T.Senftle, M. J. Janik, and A. C. van Duin, *A ReaxFF investigation of hydride formation in palladium nanoclusters via Monte Carlo and molecular dynamics simulations*, The Journal of Physical Chemistry C, 118, pp. 4967–4981, 2014

ReaxFF_HSMo_2017 (Mo, H, S)

A.Ostadhossein, A. Rahnamoun, Y. Wang, P. Zhao, S. Zhang, V. H. Crespi, and A. C. van Duin, *ReaxFF reactive force-field study of molybdenum disulfide (MoS2)*, The journal of physical chemistry letters, 8, pp. 631–640, 2017

ReaxFF_LiS_2015 (Li, S)

M.Islam, A. Ostadhossein, O. Borodin, A. T. Yeates, W. W. Tipton, R. G. Hennig, N. Kumar, and A. C. van Duin, *ReaxFF molecular dynamics simulations on lithiated sulfur cathode materials*, Physical Chemistry Chemical Physics, 17, pp. 3383–3393, 2015

ReaxFF_MgO_2023 (Mg, O)

F.Fiesinger, D. Gaissmaier, M. van den Borg, J. BeSSner, A. C. T. van Duin, and T. Jacob, *Development of a Mg/O ReaxFF Potential to describe the Passivation Processes in Magnesium-Ion Batteries***, ChemSusChem, 16, p. e202201821, 2023 link

ReaxFF_OPd_2013 (O, Pd)

T.Senftle, R. J. Meyer, M. J. Janik, and A. C. Van Duin, *Development of a ReaxFF potential for Pd/O and application to palladium oxide formation*, The Journal of chemical physics, 139, p. 044109, 2013

ReaxFF_OPt_2014 (O, Pt)

D.Fantauzzi, J. Bandlow, L. Sabo, J. E. Mueller, A. C. van Duin, and T. Jacob, *Development of a ReaxFF potential for Pt–O systems describing the energetics and dynamics of Pt-oxide formation*, Physical Chemistry Chemical Physics, 16, pp. 23118–23133, 2014

ReaxFF_RuH_2022 (Ru, H)

C.Onwudinanti, M. Pols, G. Brocks, V. Koelman, A. C. T. van Duin, T. Morgan, and S. Tao, *A ReaxFF Molecular Dynamics Study of Hydrogen Diffusion in Ruthenium–The Role of Grain Boundaries*, The Journal of Physical Chemistry C, 126, pp. 5950-5959, 2022 link

ReaxFF_SiCNH (H, C, Si, N)

. Oliver Böhm, *SiCNH parameter set*, 2015

ReaxFF_SiCu_2021 (Cu, Si)

K.Akbari Roshan, M. Khajeh Talkhoncheh, J. E. Mueller, W. A. Goddard III, and A. C. van Duin, *Development of the ReaxFF Reactive Force Field for Cu/Si Systems with Application to Copper Cluster Formation during Cu Diffusion Inside Silicon*, The Journal of Physical Chemistry C, 125, pp. 19455–19466, 2021

ReaxFF_SiLiC_2023 (C, Li, Si)

B.Olouou Guifo, J. E. Mueller, D. van Duin, M. K. Talkhoncheh, A. C. T. van Duin, D. Henriques, and T. Markus, *Development and Validation of a ReaxFF Reactive Force Field for Modeling Silicon–Carbon Composite Anode Materials in Lithium-Ion Batteries*, The Journal of Physical Chemistry C, 127, pp. 2818-2834, 2023 link

ReaxFF_SiLi_2015 (Li, Si)

H.Jung, M. Lee, B. C. Yeo, K. Lee, and S. S. Han, *Atomistic observation of the lithiation and delithiation behaviors of silicon nanowires using reactive molecular dynamics simulations*, The Journal of Physical Chemistry C, 119, pp. 3447–3455, 2015

Rohl_OZn_1996 (O, Zn)

M.Nyberg, M. A. Nygren, L. G. Pettersson, D. H. Gay, and A. L. Rohl, *Hydrogen dissociation on reconstructed ZnO surfaces*, The Journal of Physical Chemistry, 100, pp. 9054–9063, 1996

Schelling_OYZr_2001 (Zr, O, Y)

R.Schelling, S. R. Phillpot, and D. Wolf, *Mechanism of the Cubic-to-Tetragonal Phase Transition in Zirconia and Yttria-Stabilized Zirconia by Molecular-Dynamics Simulation*, Journal of the American Ceramic Society, 84, pp. 1609–1619, 2001 link

Scherer_BO_2019 (B, O)

C.Scherer, F. Schmid, M. Letz, and J. Horbach, *Structure and dynamics of B2O3 melts and glasses: From ab initio to classical molecular dynamics simulations*, Computational Materials Science, 159, pp. 73 - 85, 2019 link

StillingerWeber_AlGaN_2013 (Ga, Al, N)

W.Zhou, R. E. Jones, C. J. Kimmer, J. C. Duda, and P. E. Hopkins, *Relationship of thermal boundary conductance to structure from an analytical model plus molecular dynamics simulations*, Physical Review B, 87, p. 094303, 2013

StillingerWeber_AlGaN_2013b (Ga, Al, N)

W.Zhou, R. E. Jones, J. C. Duda, and P. E. Hopkins, *Molecular dynamics studies of material property effects on thermal boundary conductance*, Physical Chemistry Chemical Physics, 15, pp. 11078–11087, 2013

StillingerWeber_BN_2005 (B, N)

W.Moon and H. J. Hwang, *A modified Stillinger–Weber empirical potential for boron nitride*, Applied surface science, 239, pp. 376–380, 2005

StillingerWeber_BN_2007 (B, N)

W.Moon, M. S. Son, and H. J. Hwang, *Theoretical study on structure of boron nitride fullerenes*, Applied surface science, 253, pp. 7078–7081, 2007

StillingerWeber_CdTeZnSeHgS_2013 (S, Hg, Te, Se, Zn, Cd)

X.Zhou, D. Ward, J. Martin, F. van Swol, J. Cruz-Campa, and D. Zubia, *Stillinger-Weber potential for the II-VI elements Zn-Cd-Hg-S-Se-Te*, Physical Review B, 88, p. 085309, 2013

StillingerWeber_InGaN_2011 (Ga, In, N)

26.Zhang, A. Chatterjee, C. Grein, A. J. Ciani, and P. W. Chung, *Atomic-scale modeling of Inx Ga1-x N quantum dot self-assembly*, Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena, 29, p. 03C133, 2011 link

StillingerWeber_InGaN_2015 (Ga, In, N)

X.Zhou and R. E. Jones, *Towards Molecular Dynamics Simulations of InGaN Nanostructures.*, 2015 link

StillingerWeber_MoS_2013 (S, Mo)

J.Jiang, H. S. Park, and T. Rabczuk, *Molecular dynamics simulations of single-layer molybdenum disulphide (MoS2): Stillinger-Weber parametrization, mechanical properties, and thermal conductivity*, Journal of Applied Physics, 114, p. 064307, 2013

StillingerWeber_MoS_2017 (S, Mo)

M.Wen, S. N. Shirodkar, P. Plechá, E. Kaxiras, R. S. Elliott, and E. B. Tadmor, *A force-matching Stillinger-Weber potential for MoS2: Parameterization and Fisher information theory based sensitivity analysis*, Journal of Applied Physics, 122, p. 244301, 2017

StillingerWeber_NiSi_2017 (Ni, Si)

S.Hashimoto, R. Yokogawa, S. Oba, S. Asada, T. Xu, M. Tomita, A. Ogura, T. Matsukawa, M. Masahara, and T. Watanabe, *Enhanced nickelidation rate in silicon nanowires with interfacial lattice disorder*, Journal of Applied Physics, 122, p. 144305, 2017

StillingerWeber_SiGe_1995 (Ge, Si)

M.Laradji, D. Landau, and B. Dünweg, *Structural properties of Si 1-x Ge x alloys: A Monte Carlo simulation with the Stillinger-Weber potential*, Physical Review B, 51, p. 4894, 1995

StillingerWeber_SiGe_2008 (Ge, Si)

A.Gabriel, *Atomistic simulation of solid-phase epitaxial regrowth of amorphous Germanium*, 2008

StillingerWeber_SiOCF_2005 (F, O, C, Si)

V.Smirnov, A. Stengach, K. Gaynullin, V. Pavlovsky, S. Rauf, P. Stout, and P. Ventzek, *Molecular-dynamics model of energetic fluorocarbon-ion bombardment on SiO 2 I. Basic model and CF 2+-ion etch characterization*, Journal of applied physics, 97, p. 093302, 2005

StillingerWeber_Si_1985 (Si)

F.Stillinger and T. A. Weber, *Computer simulation of local order in condensed phases of silicon*, Phys. Rev. B, 31, pp. 5262–5271, 1985

SuttonChen_Classical_1998 (Cu, Pd, Au, Ir, Ag, Ni, Rh, Pt)

Y.Kimura, Y. Qi, T. Cagin, and W. Goddard, *The quantum Sutton–Chen many-body potential for properties of fcc metals*, Phys. Rev., to be submitted, 1998

SuttonChen_Fe_2000 (Fe)

S.Belonoshko, R. Ahuja, and B. Johansson, *Quasi–Ab initio molecular dynamic study of Fe melting*, Physical Review Letters, 84, p. 3638, 2000

SuttonChen_NiAl_2008 (Al, Ni)

S.Kazanc and C. Tatar, *Investigation of the effect of pressure on some physical parameters and thermoelastic phase transformation of NiAl alloy*, International Journal of Solids and Structures, 45, pp. 3282–3289, 2008

SuttonChen_NiCuAgAuPtRh_1999 (Cu, Au, Ag, Ni, Rh, Pt)

T.Cagin, G. Dereli, M. Uludoan, and M. Tomak, *Thermal and mechanical properties of some fcc transition metals*, Physical Review B, 59, p. 3468, 1999

SuttonChen_Original_1991 (Pb, Al, Cu, Pd, Au, Ir, Ag, Ni, Rh, Pt)

H.Rafii-Tabar and A. Sulton, *Long-range Finnis-Sinclair potentials for fcc metallic alloys*, Philosophical Magazine Letters, 63, pp. 217–224, 1991

SuttonChen_Original_1998 (Cu, Pd, Au, Ir, Ag, Ni, Rh, Pt)

Y.Kimura, Y. Qi, T. Cagin, and W. Goddard, *The quantum Sutton–Chen many-body potential for properties of fcc metals*, Phys. Rev., to be submitted, 1998

SuttonChen_Quantum_1998 (Cu, Pd, Au, Ir, Ag, Ni, Rh, Pt)

Y.Kimura, Y. Qi, T. Cagin, and W. Goddard, *The quantum Sutton–Chen many-body potential for properties of fcc metals*, Phys. Rev., to be submitted, 1998

Tangney_AlO_2013 (O, Al)

J.Sarsam, M. W. Finnis, and P. Tangney, *Atomistic force field for alumina fit to density functional theory*, The Journal of Chemical Physics, 139, p. 204704, 2013

Tangney_OSi_2002 (O, Si)

P.Tangney and S. Scandolo, *An ab initio parametrized interatomic force field for silica*, The Journal of chemical physics, 117, pp. 8898–8904, 2002

Tangney_OTi_2010 (Ti, O)

X.Han, L. Bergqvist, P. H. Dederichs, H. Müller-Krumbhaar, J. K. Christie, S. Scandolo, and P. Tangney, *Polarizable interatomic force field for TiO2 parametrized using density functional theory*, Phys. Rev. B, 81, p. 134108, 2010 link

TersoffBrenner_CSiF_1999 (F, C, Si)

C.Abrams and D. B. Graves, *Molecular dynamics simulations of Si etching by energetic CF 3+*, Journal of applied physics, 86, pp. 5938–5948, 1999

TersoffBrenner_ClFSi_2003 (F, Si, Cl)

D.Humbird and D. B. Graves, *Improved interatomic potentials for silicon–fluorine and silicon–chlorine*, The Journal of Chemical Physics, 120, pp. 2405-2412, 2004 link

TersoffBrenner_SiF_1999 (F, Si)

C.Abrams and D. B. Graves, *Molecular dynamics simulations of Si etching by energetic CF 3+*, Journal of applied physics, 86, pp. 5938–5948, 1999

TersoffBrenner_SiF_1999_RepulsionCorrection (F, Si)

C.F.Abrams and D. B. Graves, *Molecular dynamics simulations of Si etching by energetic CF 3+*, Journal of applied physics, 86, pp. 5938–5948, 1999

Tersoff_AlGaAs_2000 (Ga, Al, As)

K.Nordlund, J. Nord, J. Frantz, and J. Keinonen, *Strain-induced Kirkendall mixing at semiconductor interfaces*, Computational materials science, 18, pp. 283–294, 2000

Tersoff_AlNO_2009 (O, Al, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_AlNO_2009b (O, Al, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_Au_2012 (Au)

M.Backman, N. Juslin, and K. Nordlund, *Bond order potential for gold*, The European Physical Journal B, 85, pp. 1–5, 2012

Tersoff_BNC_2000 (C, B, N)

K.Matsunaga, C. Fisher, and H. Matsubara, *Tersoff potential parameters for simulating cubic boron carbonitrides*, JAPANESE JOURNAL OF APPLIED PHYSICS PART 2 LETTERS, 39, pp. L48–L51, 2000

Tersoff_BNO_2009 (B, O, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_BNO_2009b (B, O, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_BN_2003 (B, N)

W.Moon, M. S. Son, and H. J. Hwang, *Molecular-dynamics simulation of structural properties of cubic boron nitride*, Physica B: Condensed Matter, 336, pp. 329–334, 2003

Tersoff_BeCH_2009 (C, Be, H)

C.Björkas, N. Juslin, H. Timko, K. Vörtler, K. Nordlund, K. Henriksson, and P. Erhart, *Interatomic potentials for the Be-C-H system*, Journal of Physics: Condensed Matter, 21, p. 445002, 2009

Tersoff_BeH_2009 (Be, H)

C.Björkas, N. Juslin, H. Timko, K. Vörtler, K. Nordlund, K. Henriksson, and P. Erhart, *Interatomic potentials for the Be-C-H system*, Journal of Physics: Condensed Matter, 21, p. 445002, 2009

Tersoff_BeW_2010 (W, Be)

C.Björkas, K. Henriksson, M. Probst, and K. Nordlund, *A Be–W interatomic potential*, Journal of Physics: Condensed Matter, 22, p. 352206, 2010

Tersoff_CH_2005 (C, H)

N.Juslin, P. Erhart, P. Traskelin, J. Nord, K. O. Henriksson, K. Nordlund, E. Salonen, and K. Albe, *Analytical interatomic potential for modeling nonequilibrium processes in the W–C–H system*, Journal of applied physics, 98, pp. 123520–123520, 2005

Tersoff_CH_2010 (C, H)

N.Juslin, P. Erhart, P. Traskelin, J. Nord, K. O. Henriksson, K. Nordlund, E. Salonen, and K. Albe, *Analytical interatomic potential for modeling nonequilibrium processes in the W–C–H system*, Journal of applied physics, 98, pp. 123520–123520, 2005

L.Lindsay and D. Broido, *Optimized Tersoff and Brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene*, Physical Review B, 81, p. 205441, 2010

Tersoff_C_1989 (C)

J.Tersoff, *Modeling solid-state chemistry: Interatomic potentials for multicomponent systems*, Phys. Rev. B, 39, pp. 5566–5568, 1989

Tersoff_C_1994 (C)

J.Tersoff, *Chemical order in amorphous silicon carbide*, Physical Review B, 49, p. 16349, 1994

Tersoff_C_2005 (C)

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Physical Review B, 71, p. 035211, 2005

Tersoff_C_2010 (C)

L.Lindsay and D. Broido, *Optimized Tersoff and Brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene*, Physical Review B, 81, p. 205441, 2010

Tersoff_C_2012 (C)

E.Bellido and J. M. Seminario, *Molecular Dynamics Simulations of Ion-Bombarded Graphene*, The Journal of Physical Chemistry C, 116, pp. 4044-4049, 2012

Tersoff_ErH_2011 (H, Er)

S.Peng, L. Yang, X. Long, H. Shen, Q. Sun, X. Zu, and F. Gao, *Bond-Order Potential for Erbium-Hydride System*, The Journal of Physical Chemistry C, 115, pp. 25097–25104, 2011

Tersoff_FeC_2009 (C, Fe)

K.Henriksson and K. Nordlund, *Simulations of cementite: An analytical potential for the Fe-C system*, Physical Review B, 79, p. 144107, 2009

Tersoff_FeCu_2012 (Cu, Fe)

Y.Hou, R. S. Wang, J. T. Wang, X. B. Liu, G. Chen, and P. Huang, *An analytic bond-order potential for the Fe–Cu system*, Modelling and Simulation in Materials Science and Engineering, 20, p. 045016, 2012

Tersoff_FePt_2007 (Pt, Fe)

M.Müller, P. E. K., and Albe, *Thermodynamics of L1_ 0 ordering in FePt nanoparticles studied by Monte Carlo simulations based on an analytic bond-order potential*, Physical Review B, 76, p. 155412, 2007

Tersoff_Fe_2007 (Fe)

M.Müller, P. Erhart, and K. Albe, *Analytic bond-order potential for bcc and fcc ironcomparison with established embedded-atom method potentials*, Journal of Physics: Condensed Matter, 19, p. 326220, 2007

Tersoff_GaAs_2002 (Ga, As)

K.Albe, K. Nordlund, J. Nord, and A. Kuronen, *Modeling of compound semiconductors: Analytical bond-order potential for Ga, As, and GaAs*, Physical Review B, 66, p. 035205, 2002

Tersoff_GaAs_2008 (Ga, As)

T.Hammerschmidt, P. Kratzer, and M. Scheffler, *Analytic many-body potential for InAs/GaAs surfaces and nanostructures: Formation energy of InAs quantum dots*, Physical Review B, 77, p. 235303, 2008

Tersoff_GaAs_2011 (Ga, As)

K.Fichthorn, Y. Tiwary, T. Hammerschmidt, P. Kratzer, and M. Scheffler, *Analytic many-body potential for GaAs (001) homoepitaxy: Bulk and surface properties*, Physical Review B, 83, p. 195328, 2011

Tersoff_GaNO_2009 (Ga, O, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_GaNO_2009b (Ga, O, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_GaN_2003 (Ga, N)

J.Nord, K. Albe, P. Erhart, and K. Nordlund, *Modelling of compound semiconductors: analytical bond-order potential for gallium, nitrogen and gallium nitride*, Journal of Physics: Condensed Matter, 15, p. 5649, 2003

Tersoff_InAs_2008 (In, As)

T.Hammerschmidt, P. Kratzer, and M. Scheffler, *Analytic many-body potential for InAs/GaAs surfaces and nanostructures: Formation energy of InAs quantum dots*, Physical Review B, 77, p. 235303, 2008

Tersoff_InGaAs_2000 (Ga, In, As)

K.Nordlund, J. Nord, J. Frantz, and J. Keinonen, *Strain-induced Kirkendall mixing at semiconductor interfaces*, Computational materials science, 18, pp. 283–294, 2000

Tersoff_InNO_2009 (O, In, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_InNO_2009b (O, In, N)

O.Okeke and J. Lowther, *Molecular dynamics of binary metal nitrides and ternary oxynitrides*, Physica B: Condensed Matter, 404, pp. 3577–3581, 2009

Tersoff_O_2006 (O)

P.Erhart, N. Juslin, O. Goy, K. Nordlund, R. Müller, and K. Albe, *Analytic bond-order potential for atomistic simulations of zinc oxide*, Journal of Physics: Condensed Matter, 18, p. 6585, 2006

Tersoff_Powell_2007 (In, N, P, Ga, Al, Sb, As)

D.Powell, M. Migliorato, and A. Cullis, *Optimized Tersoff potential parameters for tetrahedrally bonded III-V semiconductors*, Physical Review B, 75, p. 115202, 2007

Tersoff_PtC_2002 (C, Pt)

K.Albe, K. Nordlund, and R. S. Averback, *Modeling the metal-semiconductor interaction: Analytical bond-order potential for platinum-carbon*, Physical Review B, 65, p. 195124, 2002

Tersoff_Pt_2002 (Pt)

K.Albe, K. Nordlund, and R. S. Averback, *Modeling the metal-semiconductor interaction: Analytical bond-order potential for platinum-carbon*, Physical Review B, 65, p. 195124, 2002

Tersoff_SiBN_2001 (B, Si, N)

K.Matsunaga and Y. Iwamoto, *Molecular dynamics study of atomic structure and diffusion behavior in amorphous silicon nitride containing boron*, Journal of the American Ceramic Society, 84, pp. 2213–2219, 2001

Tersoff_SiC_1989 (C, Si)

J.Tersoff, *Modeling solid-state chemistry: Interatomic potentials for multicomponent systems*, Phys. Rev. B, 39, pp. 5566–5568, 1989

J.Tersoff, *Erratum: Modeling solid-state chemistry: Interatomic potentials for multicomponent systems*, Physical Review B, 41, pp. 3248–3248, 1990

Tersoff_SiC_1994 (C, Si)

J.Tersoff, *Chemical order in amorphous silicon carbide*, Physical Review B, 49, p. 16349, 1994

Tersoff_SiC_1998 (C, Si)

R.Devanathan, T. Diaz de la Rubia, and W. Weber, *Displacement threshold energies in -SiC*, Journal of nuclear materials, 253, pp. 47–52, 1998

Tersoff_SiC_2005 (C, Si)

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Physical Review B, 71, p. 035211, 2005

Tersoff_SiGeO_2013 (Ge, O, Si)

C.Chuang, Q. Li, D. Leonhardt, S. M. Han, and T. Sinno, *Atomistic analysis of Ge on amorphous SiO2 using an empirical interatomic potential*, Surface Science, 609, pp. 221–229, 2013

Tersoff_SiGeO_LT_2013 (Ge, O, Si)

C.Chuang, Q. Li, D. Leonhardt, S. M. Han, and T. Sinno, *Atomistic analysis of Ge on amorphous SiO2 using an empirical interatomic potential*, Surface Science, 609, pp. 221–229, 2013

Tersoff_SiGe_1989 (Ge, Si)

J.Tersoff, *Modeling solid-state chemistry: Interatomic potentials for multicomponent systems*, Phys. Rev. B, 39, pp. 5566–5568, 1989

J.Tersoff, *Erratum: Modeling solid-state chemistry: Interatomic potentials for multicomponent systems*, Physical Review B, 41, pp. 3248–3248, 1990

Tersoff_SiNH_1999 (Si, H, N)

F.de Brito Mota, J. Justo, and A. Fazzio, *Hydrogen role on the properties of amorphous silicon nitride*, Journal of applied physics, 86, pp. 1843–1847, 1999

Tersoff_SiO_2007 (O, Si)

S.Munetoh, T. Motooka, K. Moriguchi, and A. Shintani, *Interatomic potential for Si–O systems using Tersoff parameterization*, Computational materials science, 39, pp. 334–339, 2007

Tersoff_Si_1988 (Si)

J.Tersoff, *Empirical interatomic potential for silicon with improved elastic properties*, Physical Review B, 38, pp. 9902–9905, 1988

J.Tersoff, *New empirical approach for the structure and energy of covalent systems*, Physical Review B, 37, p. 6991, 1988

Tersoff_Si_1988b (Si)

J.Tersoff, *Empirical interatomic potential for silicon with improved elastic properties*, Physical Review B, 38, pp. 9902–9905, 1988

Tersoff_Si_2005 (Si)

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Physical Review B, 71, p. 035211, 2005

Tersoff_WCH_2005 (W, C, H)

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Physical Review B, 71, p. 035211, 2005

N.Juslin, P. Erhart, P. Traskelin, J. Nord, K. O. Henriksson, K. Nordlund, E. Salonen, and K. Albe, *Analytical interatomic potential for modeling nonequilibrium processes in the W–C–H system*, Journal of applied physics, 98, pp. 123520–123520, 2005

Tersoff_WCH_2005b (W, C, H)

P.Erhart and K. Albe, *Analytical potential for atomistic simulations of silicon, carbon, and silicon carbide*, Physical Review B, 71, p. 035211, 2005

N.Juslin, P. Erhart, P. Traskelin, J. Nord, K. O. Henriksson, K. Nordlund, E. Salonen, and K. Albe, *Analytical interatomic potential for modeling nonequilibrium processes in the W–C–H system*, Journal of applied physics, 98, pp. 123520–123520, 2005

Tersoff_WH_2011 (W, H)

X.Li, X. Shu, Y. Liu, F. Gao, and G. Lu, *Modified analytical interatomic potential for a W–H system with defects*, Journal of Nuclear Materials, 408, pp. 12–17, 2011

Tersoff_ZnO_2006 (O, Zn)

P.Erhart, N. Juslin, O. Goy, K. Nordlund, R. Müller, and K. Albe, *Analytic bond-order potential for atomistic simulations of zinc oxide*, Journal of Physics: Condensed Matter, 18, p. 6585, 2006

Tersoff_Zn_2006 (Zn)

P.Erhart, N. Juslin, O. Goy, K. Nordlund, R. Müller, and K. Albe, *Analytic bond-order potential for atomistic simulations of zinc oxide*, Journal of Physics: Condensed Matter, 18, p. 6585, 2006

Trinastic_HfOSiTaTi_2013 (Hf, Ta, Ti, O, Si)

J.Trinastic, R. Hamdan, Y. Wu, L. Zhang, and H. Cheng, *Unified interatomic potential and energy barrier distributions for amorphous oxides*, The Journal of chemical physics, 139, p. 154506, 2013

VFF_Keating_CGeSi_1966 (Si, C, Ge)

P.Keating, *Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure*, Phys. Rev., 145, pp. 637–645, 1966 link

VFF_Martin_CGeSi_1970 (Si, C, Ge)

R.Martin, *Elastic Properties of ZnS Structure Semiconductors*, Phys. Rev. B, 1, pp. 4005–4011, 1970 link

VanBeest_SiOAlP_1990 (O, Al, Si, P)

B.Van Beest, G. Kramer, and R. Van Santen, *Force fields for silicas and aluminophosphates based on ab initio calculations*, Physical Review Letters, 64, p. 1955, 1990

Voth_HO_2006 (O, H)

Y.Wu, H. L. Tepper, and G. A. Voth, *Flexible simple point-charge water model with improved liquid-state properties*, The Journal of Chemical Physics, 124, p. 024503, 2006

WalshRichardCatlow_InO_2009 (O, In)

A.Walsh, C. R. A. Catlow, A. A. Sokol, and S. M. Woodley, *Physical properties, intrinsic defects, and phase stability of indium sesquioxide*, Chemistry of Materials, 21, pp. 4962–4969, 2009

Wang_BCaNaOSi_2018 (B, Na, Ca, O, Si)

M.Wang, N. A. Krishnan, B. Wang, M. M. Smedskjaer, J. C. Mauro, and M. Bauchy, *A new transferable interatomic potential for molecular dynamics simulations of borosilicate glasses*, Journal of Non-Crystalline Solids, 498, pp. 294 - 304, 2018 link

Wang_HfOZr_2012 (Hf, Zr, O)

Y.Wang, F. Zahid, J. Wang, and H. Guo, *Structure and dielectric properties of amorphous high- oxides: HfO2, ZrO2, and their alloys*, Phys. Rev. B, 85, p. 224110, 2012 link

Yasukawa_HOSi_1996 (H, O, Si)

A.Yasukawa, *Using An Extended Tersoff Interatomic Potential to Analyze The Static-Fatigue Strength of SiO2 under Atmospheric Influence*, JSME international journal. Ser. A, Mechanics and material engineering, 39, pp. 313-320, 1996 link

Yasukawa_OSi_2003 (O, Si)

A.Yasukawa, *An Interatomic Potential for Strength Analysis under Atomospheric Influence*, Ibaraki district conference, 2003, pp. 71-72, 2003 link

Pretrained moment tensor potential (MTP) parameter sets

Listed below are the defined pre-trained MTP parameter sets developed by Synopsys QuantumATK. These parameter sets require the *QuantumATK-ML-Elite* license.

QuantumATK_MTP_Ag_SiO2_2022 (O, Ag, Si)

Synopsys pretrained MTP parameters for c-Ag / SiO2 interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 2000 MTP basis functions., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_CoSi_2022_12 (Co, Si)

Synopsys pretrained MTP parameters for cobalt silicide and Si / Co interfaces. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and 400 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_CuH_2025_06 (Cu, H)

Synopsys pretrained MTP parameters for Hydrogen diffusion in Copper. Trained using DFT-LCAO PBES and PseudoDojo-high as reference calculator and 300 MTP basis functions., Synopsys QuantumATK, X-2025.06

QuantumATK_MTP_CuMnTaNSiO_2025_06 (Cu, Ta, Mn, N, O, Si)

Synopsys pretrained MTP parameters for Cu-Mn-Ta-N-Si-O alloy materials and interfaces for back-end-of-line (BEOL) applications. Trained using DFT-LCAO PBES and PseudoDojo-medium as reference calculator and 800 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK, X-2025.06

QuantumATK_MTP_FeCo_Alloy_2022 (Co, Fe)

Synopsys pretrained MTP parameters for bulk FexCo(1-x) alloy configuration, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_FeCo_MgO_2022 (Co, O, Mg, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, MgO and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_FeCo_Ta_2022 (Ta, Co, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, Ta and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_FeCo_W_2022 (W, Co, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, W and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Fe_MgO_2022 (Co, O, Mg, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, MgO and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Fe_Ta_2022 (Ta, Co, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, Ta and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Fe_W_2022 (W, Co, Fe)

Synopsys pretrained MTP parameters for bulk Fe, FexCo(1-x) alloy, W and their interfaces, where x=0.5-1. Trained using DFT-LCAO PBE and spin polarization with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_GST_2022_12 (Sb, Te, Ge)

Synopsys pretrained MTP parameters for bulk crystal and amorphous Ge2Sb2Te5 systems. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfO2_2022 (Hf, O)

Synopsys pretrained MTP parameters for bulk crystal and amorphous HfO2. Trained to monoclinic, orthorhombic, cubic, and low- and high-temperature am. HfO2, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_HfO2_2022_12 (Hf, O)

Synopsys pretrained MTP parameters for bulk crystal and amorphous HfO2. Trained to monoclinic, orthorhombic, cubic, and low- and high-temperature am. HfO2, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfO2_HfCl4_2022 (Hf, O, Cl)

Synopsys pretrained MTP parameters for HfCl4 deposition on HfO2 surfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_HfO2_Ru_2022 (Hf, O, Ru)

Synopsys pretrained MTP parameters for HfO2 / Ru interfaces. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and 5000 MTP basis functions., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfO2_Sc_2022 (Hf, O, Sc)

Synopsys pretrained MTP parameters for HfO2 / Sc interfaces. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and 5000 MTP basis functions., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfO2_SiO2_2022 (Hf, O, Si)

Synopsys pretrained MTP parameters for SiO2 / HfO2 interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_HfO2_SiO2_2022_12 (Hf, O, Si)

Synopsys pretrained MTP parameters for SiO2 / HfO2 interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfO2_TiN_2022 (Ti, Hf, O, N)

Synopsys pretrained MTP parameters for HfO2 / TiN interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_HfO2_TiN_2022_12 (Ti, Hf, O, N)

Synopsys pretrained MTP parameters for HfO2 / TiN interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_HfTiNAlO_2023_09 (Hf, Ti, N, O, Al)

Synopsys pretrained MTP parameters for bulk crystal and amorphous HfTiNAlO systems and interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 800 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK., V-2023.09

QuantumATK_MTP_InGaZnOH_2023_09 (In, O, H, Ga, Zn)

Synopsys pretrained MTP parameters for crystal amorphous InGaZnO (IGZO) with Hydrogen. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Supports MTP error prediction., Synopsys QuantumATK, U-2023.09

QuantumATK_MTP_InGaZnO_2022_12 (Ga, O, In, Zn)

Synopsys pretrained MTP parameters for crystal amorphous InGaZnO (IGZO). Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_MgO_FeCo_W_Ta_2022_12 (Ta, Mg, Fe, W, Co, O)

Synopsys pretrained MTP parameters for MgO / FeCo / W or MgO / FeCo / Ta interfaces in an MRAM stack. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_MoS2_Cl_Ar_2025_06 (Cl, Ar, S, Mo)

Synopsys pre-trained MTP parameters for MoS2 layers Cl and Ar. Trained using DFT-LCAO PBE PseudoDojo-high as reference calculator and 750 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK., X-2025.06

QuantumATK_MTP_Ru_2022 (Ru)

Synopsys pretrained MTP parameters for bulk crystal and amorphous Ru. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Sc_2022 (Sc)

Synopsys pretrained MTP parameters for bulk crystal and amorphous Sc. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_SiC_Defects_2023 (C, Si)

Synopsys pre-trained MTP parameters for 4H and 6H bulk hexagonal SiC. Included defects are Si/C vacancies, Si/C anti-sites (native substitutionals), and various Si/C interstitials. DFT-LCAO PBE PseudoDojo-medium as reference calculator and 3000 MTP basis., Synopsys QuantumATK, V-2023.09

QuantumATK_MTP_SiO2_2022 (O, Si)

Synopsys pretrained MTP parameters for bulk crystal and amorphous SiO2. Trained to quartz, cristobalite, and low- and high-temperature am. SiO2, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_SiO2_2022_12 (O, Si)

Synopsys pretrained MTP parameters for bulk crystal and amorphous SiO2. Trained to quartz, cristobalite, and low- and high-temperature am. SiO2, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Si_SiO2_2022 (O, Si)

Synopsys pretrained MTP parameters for Si / SiO2 interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_Si_SiO2_2022_12 (O, Si)

Synopsys pretrained MTP parameters for Si / SiO2 interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_Si_SiO2_HfO2_TiN_2022_12 (Hf, Ti, N, O, Si)

Synopsys pretrained MTP parameters for Si / SiO2 / HfO2 / TiN interfaces in an HKMG stack. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_Si_SiO2_HfO2_TiN_2025_06 (Hf, Ti, N, O, Si)

Synopsys pretrained MTP parameters for Si / SiO2 / HfO2 / TiN interfaces in an HKMG stack. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and 800 MTP basis functions. Retrained with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, X-2025.06

QuantumATK_MTP_Si_SiO2_HfO2_TiN_Al_2025_06 (Hf, Ti, N, O, Si, Al)

Synopsys pretrained MTP parameters for Si / SiO2 / HfO2 / TiN interfaces in an HKMG stack with Al doping. Trained using DFT-LCAO PBE PseudoDojo-medium as reference calculator and 800 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK, X-2025.06

QuantumATK_MTP_Si_Ti_TiSi_2022 (Ti, Si)

Synopsys pretrained MTP parameters for Si / Ti / TiSi interfaces. Trained to TiSi, C49- and C54-TiSi2, low- and high-temperature am. TiSi, a well as interfaces with crystalline Si and Ti, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 300 MTP basis functions., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_Si_Ti_TiSi_2023_09 (Ti, Si)

Synopsys pretrained MTP parameters for Si / Ti / TiSi interfaces. Trained to TiSi, C49- and C54-TiSi2, low- and high-temperature am. TiSi, a well as interfaces with crystalline Si and Ti, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 300 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK, V-2023.09

QuantumATK_MTP_TiNAlO_2022 (Ti, O, Al, N)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiNAlO systems and interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 734 MTP basis functions., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_TiNAlO_2022_12 (Ti, O, Al, N)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiNAlO systems and interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 734 MTP basis functions., Synopsys QuantumATK. Retrained version with improved accuracy., U-2022.12

QuantumATK_MTP_TiNAlO_2023_09 (Ti, O, Al, N)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiNAlO systems and interfaces. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 800 MTP basis functions. Supports MTP error prediction., Synopsys QuantumATK. Retrained version with improved accuracy., V-2023.09

QuantumATK_MTP_TiN_2022 (Ti, N)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiN. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and PredefinedSmall MTP basis., Synopsys QuantumATK, T-2022.03

QuantumATK_MTP_TiN_2022_12 (Ti, N)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiN. Trained using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 734 MTP basis functions. Retrained version with improved accuracy. Supports MTP error prediction., Synopsys QuantumATK, U-2022.12

QuantumATK_MTP_TiSi_2022 (Ti, Si)

Synopsys pretrained MTP parameters for bulk crystal and amorphous TiSi. Trained to TiSi, C49- and C54-TiSi2, low- and high-temperature am. TiSi, using DFT-LCAO PBE with PseudoDojo-medium as reference calculator and 300 MTP basis functions., Synopsys QuantumATK, T-2022.03

4.2.5 NEGF: Device Calculators

Introduction

This section describes the most important aspects of the formalism in QuantumATK for simulating device systems using the non-equilibrium Greens function (NEGF) method. For a more detailed description of the theory and application examples you may consult QuantumATK reference paper by Smidstrup *et al.*[1] and some of the many research papers on the subject[2],[3],[4],[5].

In QuantumATK, the NEGF formalism is implemented in combination with *DFT: LCAO* and *Semi Empirical* in the *DeviceLCAOCalculator* and the *DeviceSemiEmpiricalCalculator*, respectively.

Device configuration

A typical device configuration is illustrated in Fig. 4.2.

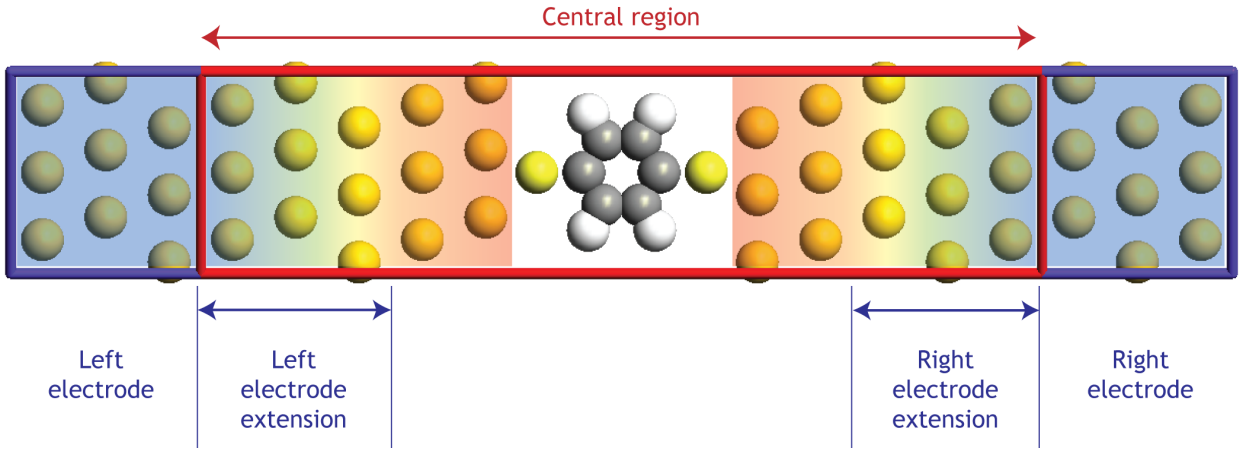


Fig. 4.2: Geometry of a device configuration with two electrodes (two-probe configuration). In the left and right electrode regions the system is periodic in the transport direction (the *C* direction). The central region seamlessly connects the left and right electrode regions, and the part of the central region where the atom positions follow the periodic arrangement of the electrodes is called the left and right electrode extension. The background color symbolizes the perturbation of the electrode effective potential relative to the electrode self-consistent value. A sufficient fraction of the electrodes must be included in the central region to screen out the perturbation of the scattering region, such that the outermost part of the central region attains an effective potential similar to the bulk potential of the electrodes. The matching between these potentials forms the boundary condition for the electrostatic problem.

The system can be divided into three regions: left, central, right. The implementation relies on the so-called screening approximation, which assumes that the properties of the left and right regions, the electrodes, can be described by solving a bulk problem for the fully periodic electrode cell. The screening approximation will formally be fulfilled

[1] S.ăSmidstrup, T.ăMarkussen, P.ăVancraeyveld, J.ăWellendorff, J.ăSchneider, T.ăGunst, B.ăVerstichel, D.ăStradi, P.ăA. Khomyakov, U.ăG. Vej-Hansen, M.-E. Lee, S.ăT. Chill, F.ăRasmussen, G.ăPenazzi, F.ăCorsetti, A.ăOjanperä, K.ăJensen, M.ăL.ăN. Palsgaard, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. QuantumATK: An integrated platform of electronic and atomic-scale modelling tools. *Journal of Physics: Condensed Matter*, 32(1):015901, Oct 2019. URL: https://doi.org/10.1088/1361-648X/ab4007 (visited on 2019-10-10), doi:10.1088/1361-648x/ab4007.

[2] M.ăBrandbyge, J.-L. Mozos, P.ăOrdejon, J.ăTaylor, and K.ăStokbro. Density-functional method for nonequilibrium electron transport. *Phys. Rev. B*, 65:165401, Mar 2002. doi:10.1103/PhysRevB.65.165401.

[3] K.ăStokbro, J.ăTaylor, M.ăBrandbyge, and H.ăGuo. *Ab-initio based Non-equilibrium Greens Function Formalism for Calculating Electron Transport in Molecular Devices*, pages 117–151. Springer, 2005.

[4] H.ăHaug and A.-P. Jauho. *Quantum Kinetics in Transport and Optics of Semiconductors*. Volume 123. Springer-Verlag Berlin Heidelberg, 2 edition, 2008. Springer Series in Solid-State Sciences. doi:10.1007/978-3-540-73564-9.

[5] S.ăDatta. *Electronic Transport in Mesoscopic Systems*. Cambridge University Press, 1997. ISBN 0521599431. Cambridge Studies in Semiconductor Physics and Microelectronic Engineering.

when the current through the system is sufficiently small that the electrode regions can be described by an equilibrium electron distribution.

Fig. 4.2 illustrates how the electrode regions must be extended into the central region, in order to screen out the perturbations from the scatterer (in this case the benzene molecule) inside the device. For metallic electrodes it is usually sufficient to extend the electrodes 5-10 Å into the central region.

Non-equilibrium electron distribution

The left and right regions are equilibrium systems with periodic boundary conditions, and the properties of these systems are obtained using a conventional electronic structure calculation. The challenge in calculating the properties of a device system lies in the calculation of the non-equilibrium electron distribution in the central region.

The assumption is that the system is in a steady state such that the electron density of the central region is constant in time. The electron density is given by the occupied eigenstates of the system. Since the chemical potential is different in the two electrodes, the contribution from each electrode to the total electron density in the central region must be calculated independently:

$$n(\mathbf{r}) = n^L(\mathbf{r}) + n^R(\mathbf{r}).$$

The contribution from the left (n^L) and right (n^R) electrodes can be obtained by calculating the scattering states, which are the eigenstates of the system when scattering boundary conditions are used. Fig. 4.3 illustrates a left moving scattering state with origin in the right electrode.

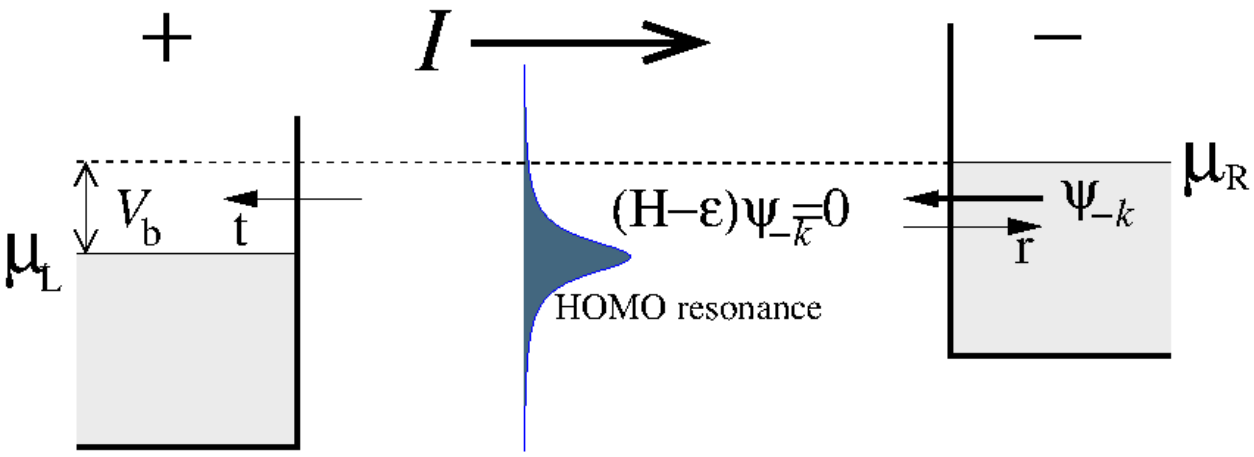


Fig. 4.3: The electron distribution in a device configuration. The left and right regions have an equilibrium electron distribution with chemical potentials μ_L and μ_R, related through the applied sample bias, $\mu_R - \mu_L = eV_b$. The electrons with energies in the bias window $\mu_L \leq \varepsilon \leq \mu_R$, give rise to a steady state electrical current. The figure illustrates a left moving scattering state with origin in the right electrode.

The left and right density are now calculated by summing up the occupied scattering states,

$$n^L(\mathbf{r}) = \sum_\alpha |\psi_\alpha(\mathbf{r})|^2 f\left(\frac{\varepsilon_\alpha - \mu_L}{kT}\right),$$

$$n^R(\mathbf{r}) = \sum_\alpha |\psi_\alpha(\mathbf{r})|^2 f\left(\frac{\varepsilon_\alpha - \mu_R}{kT}\right).$$

The scattering states of the system are calculated by first calculating the Bloch states in the electrodes and subsequently solving the Schrödinger equation of the central region using the Bloch states as matching boundary conditions.

The non-equilibrium Greens function (NEGF) method

Instead of using the scattering states to calculate the non-equilibrium electron density, QuantumATK uses the non-equilibrium Greens function (NEGF) method; the two approaches are formally equivalent and give identical results[Page 1935, 2].

The electron density is given in terms of the electron density matrix, as described in section *Electron density*. We divide the density matrix into left and right contributions,

$$D = D^L + D^R.$$

The left density matrix contribution is calculated using the NEGF method as[Page 1935, 2]

$$D^L = \int \rho^L(\varepsilon) f\left(\frac{\varepsilon - \mu_L}{k_B T_L}\right) d\varepsilon,$$

where

$$\rho^L(\varepsilon) \equiv \frac{1}{2\pi} G(\varepsilon)\Gamma^L(\varepsilon)G^\dagger(\varepsilon)$$

is the spectral density matrix. Note that while there is a non-equilibrium electron distribution in the central region, the electron distribution in the electrode is described by a Fermi function f with an electron temperature T_L .

In this equation, G is the retarded Greens function, and

$$\Gamma^L = \frac{1}{i}(\Sigma^L - (\Sigma^L)^\dagger)$$

is the broadening function of the left electrode, given in terms of the left electrode self energy, Σ^L . A similar equation exists for the right density matrix contribution.

The following section describes the calculation of G and Σ in more detail.

Retarded Greens function

The key quantity to calculate is the retarded Greens function matrix,

$$G(\varepsilon) = \frac{1}{(\varepsilon + i\,\delta_+)S - H},$$

where δ_+ is an infinitesimal positive number. S and H are the overlap and Hamiltonian matrices, respectively, of the entire system.

The Greens function is only required for the central region and can be calculated from the Hamiltonian of the central region by adding the electrode self energies:

$$G(\varepsilon) = [(\varepsilon + i\delta_+)S - H - \Sigma^L(\varepsilon) - \Sigma^R(\varepsilon)]^{-1}.$$

Therefore, the calculation of the Greens function of the central region at a specific energy requires the inversion of the Hamiltonian matrix of the central region. In QuantumATK this matrix is stored in a sparse format, and the inversion is done using an $\mathcal{O}(N)$ algorithm[6].

[6] D.ăE. Petersen, H.ăH.ăB. Sorensen, P.ăC. Hansen, S.ăSkelboe, and K.ăStokbro. Block Tridiagonal Matrix Inversion and Fast Transmission Calculations. *J. Comput. Phys.*, 227(6):3174–3190, 2008. doi:10.1016/j.jcp.2007.11.035.

Self energy

The self energy describe the effect of the electrode states on the electronic structure of the central region. The self energy can be calculated from the electrode Hamiltonian. QuantumATK provides 4 different methods for calculating the self energy:

- *DirectSelfEnergy* uses an exact diagonalization of the Hamiltonian for calculating the self energy[7].
- *RecursionSelfEnergy* uses an iterative scheme for calculating the self energy[8], which gives essentially the same results as the direct method but is faster for some systems.
- *SparseRecursionSelfEnergy* uses the same iterative scheme as *RecursionSelfEnergy*, but exploits inherent sparsity. This provides a smaller memory footprint, as well as increased performance for all but the smallest systems.
- *KrylovSelfEnergy* is an approximate method which is an order of magnitude faster that the other methods, but may in rare cases give inaccurate results[9],[10] .

Complex contour integration

The energy integral needed to obtain the density matrix within the NEGF framework is evaluated through a complex contour integration. It is divided into two parts; an integral over equilibrium states and an integral over non-equilibrium states.

The integral over the **equilibrium states** can be done using two different methods:

- *SemiCircleContour*: Defines a semi-circular complex contour that gives high computational efficiency[Page 1935, 2].
- *OzakiContour*: Evaluation of the complex contour integral is based on the residue theorem and a continued-fraction representation of the Fermi–Dirac districation[11][12]. This is a highly stable method, but is also less efficient than *SemiCircleContour*.

The integral over the **non-equilibrium states** must be performed along the real axis, and uses the *RealAxisContour* method. Note that for large biases, this is often the most demanding part of the calculation.

Lastly, one of two different **contour methods** must be chosen (usually just left at the default value):

- *SingleContour* uses a single contour for the calculation. This is appropriate for small biases, and is the fastest method.
- *DoubleContour* uses a double contour for the calculation. This gives best stability and must be used for high biases. The method can also handle if there are bound states inside the bias window.

See also *ContourParameters*.

[7] S.ăSanvito, C.ăJ. Lambert, J.ăH. Jefferson, and A.ăM. Bratkovsky. General Greens-function formalism for transport calculations with spd Hamiltonians and giant magnetoresistance in Co- and Ni-based magnetic multilayers. *Phys. Rev. B*, 59:11936–11948, May 1999. doi:10.1103/PhysRevB.59.11936.

[8] M.ăP. LopezăSancho, J.ăM. LopezăSancho, and J.ăRubio. Highly convergent schemes for the calculation of bulk and surface Green functions. *J. Phys. F: Metal Physics*, 15(4):851, 1985. URL: http://stacks.iop.org/0305-4608/15/i=4/a=009.

[9] H.ăH.ăB. Sorensen, P.ăC. Hansen, D.ăE. Petersen, S.ăSkelboe, and K.ăStokbro. Krylov subspace method for evaluating the self-energy matrices in electron transport calculations. *Phys. Rev. B*, 77:155301, Apr 2008. doi:10.1103/PhysRevB.77.155301.

[10] H.ăH.ăB. Sorensen, P.ăC. Hansen, D.ăE. Petersen, S.ăSkelboe, and K.ăStokbro. Efficient wave-function matching approach for quantum transport calculations. *Phys. Rev. B*, 79:205322, May 2009. doi:10.1103/PhysRevB.79.205322.

[11] Taisuke Ozaki. Continued fraction representation of the Fermi-Dirac function for large-scale electronic structure calculations. *Phys. Rev. B*, 75:035123, 2007. doi:10.1103/PhysRevB.75.035123.

[12] T.ăOzaki, K.ăNishio, and H.ăKino. Efficient implementation of the nonequilibrium green function method for electronic transport calculations. *Phys. Rev. B*, 81:035116, 2010. doi:10.1103/PhysRevB.81.035116.

Spill-in terms

In terms of the density matrix, D , the electron density of the central region is given by

$$n(\mathbf{r}) = \sum_{ij} D_{ij}\phi_i(\mathbf{r})\phi_j(\mathbf{r}).$$

The Greens function of the central region gives the density matrix of the central region, D_{CC} , however, to calculate the density correctly close to the central cell boundaries the terms involving, $D_{LL}, D_{LC}, D_{CR}, D_{RR}$ are also needed. These terms are denoted spill-in terms[13].

ATK includes all the spill-in terms, both for calculating the electron density $n(\mathbf{r})$ and the Hamiltonian integral. This gives additional stability and well-behaved convergence in the device algorithm[Page 1939, 13].

Effective potential

Once the non-equilibrium density is obtained the next step in the self-consistent calculation is the calculation of the effective potential as the sum of the exchange-correlation and electrostatic Hartree potentials. The calculation of the exchange-correlation potential is straightforward since it is a local or semi-local function of the density. However, the calculation of the electrostatic Hartree potential requires some additional consideration for a device system. The following describes the calculation of the Hartree potential in more detail.

The starting point is the calculation of the self-consistent Hartree potential in the left and right electrodes. The Hartree potential of a bulk system is defined up to an arbitrary constant. However, in a device setup the Hartree potentials of the two electrodes are aligned through their chemical potentials (i.e. their Fermi levels), since these are related by the applied bias:

$$\mu_R - \mu_L = eV_b.$$

The Hartree potential of the central region is obtained by solving the Poisson equation, using the bulk-like Hartree potentials of the electrodes as boundary conditions at the interfaces between the electrodes and the central region. An overview of the available Poisson solvers and how boundary conditions are applied for NEGF calculations is given in *Boundary conditions*.

Total energy and forces

A device system is an open system where charge can flow in and out of the central region from the left and right reservoirs. Since the particle number is not conserved it is necessary to use a grand canonical potential to describe the energetics of the system[14]:

$$\Omega[n] = \varepsilon[n] - e\,N_L\,\mu_L - e\,N_R\,\mu_R,$$

where $N_{L/R}$ is the number of electrons contributed to the central region from the left/right electrode.

Due to the screening approximation, the central region will be charge neutral, and therefore $N_L + N_R = N$, where N is the ionic charge in the central region. Thus, at no applied bias, we ahve $\mu_L = \mu_R$ and the particle terms in Ω will be constant when atoms are moved in the central region. However, at finite bias, $\mu_L \neq \mu_R$, and the particle terms in Ω will be important.

The forces are given by

$$\mathbf{F}_i = -\frac{\partial\Omega[n]}{\partial\mathbf{R_i}}.$$

[13] D.ăStradi, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. General atomistic approach for modeling metal-semiconductor interfaces using density functional theory and nonequilibrium Greens function. *Phys. Rev. B*, 93:155302, Apr 2016. doi:10.1103/PhysRevB.93.155302.

[14] T.ăN. Todorov. Local heating in ballistic atomic-scale contacts. *Philosophical Magazine Part B*, 77(4):965–973, 1998. doi:10.1080/13642819808206398.

It can be shown that the calculation of this force is identical to the calculation of the equilibrium force, as described in *Total energy and forces*. In the non-equilibrium case it is however required that the density and energy density matrix is calculated with the NEGF framework[Page 1935, 2], [Page 1939, 14], [15].

See also *TotalEnergy*.

Transmission coefficient

When the self-consistent non-equilibrium density matrix has been obtained, it is possible to calculate various transport properties of the system. One of the most notable is the *TransmissionSpectrum* from which you can obtain the current and differential conductance.

By the transmission amplitude t_k we define the fraction of a scattering state k which propagates through a device. The transmission coefficient at energy ε is obtained by summing up the transmission amplitude from all the states at this energy,

$$T(\varepsilon) = \sum_k t_k^\dagger t_k \delta(\varepsilon - \varepsilon_k).$$

The transmission coefficient may also be obtained from *the retarded Greens function* using

$$T(\varepsilon) = G(\varepsilon)\Gamma^L(\varepsilon)G^\dagger(\varepsilon)\Gamma^R(\varepsilon),$$

and this is how it is calculated in QuantumATK. The transmission amplitude of individual scattering states may be obtained through the *TransmissionEigenvalues*.

Electrical current

To calculate the current you must first calculate a *TransmissionSpectrum*, and then from this extract the electrical current using the `current()` method on the object. This approach has the advantage that once the *TransmissionSpectrum* is calculated, it is fast to calculate the current for different electrode temperatures[Page 1939, 13].

See *TransmissionSpectrum* for more details.

References

4.3 Python in QuantumATK

Table of Contents

•*ATK-Python*

•*Physical quantities and units*

•*Read and Write Support*

•*Metatext*

•*Spin*

•*Command Line Usage with QATK Shell Environments*

[15] R.ăZhang, I.ăRungger, S.ăSanvito, and S.ăHou. Current-induced energy barrier suppression for electromigration from first principles. *Phys. Rev. B*, 84:085445, Aug 2011. doi:10.1103/PhysRevB.84.085445.

All calculations in QuantumATK are controlled via *ATK-Python*, which is an extension to the well-known Python scripting language. All QuantumATK Python scripts are executed using the **atkpython executable**, which is installed with QuantumATK.

If you have no prior knowledge about the Python Programming Language you might want to check out our quick introduction in sect1.python.basics.

If you have an QuantumATK Python script named `script.py`, it is easy to run it from command line:

```
$ atkpython script.py
```

You can also simply execute `atkpython` without an argument to invoke an **interactive session** that allows you to execute ATK-Python commands one after another (hit Enter to execute a line):

```
$ atkpython
# ------------------------------------------------------------------ #
# QuantumATK license information.                                    #
# ------------------------------------------------------------------ #
QuantumATK 2016.0

In [1]: a=1.0

In [2]: b=2.0

In [3]: c=a+b

In [4]: print("a+b = c =", c)
a+b = c = 3.0

In [5]: bulk_configuration = BulkConfiguration(
    ...:     bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    ...:     elements=[Silicon, Silicon],
    ...:     fractional_coordinates=[[0.,0.,0.],[0.25,0.25,0.25]])

In [6]: nlprint(bulk_configuration)
+-------------------------------------------------------------+
| Bulk Bravais lattice                                        |
+-------------------------------------------------------------+
Type:
FaceCenteredCubic

Lattice constants:
a =     5.430600 Ang
b =     5.430600 Ang
c =     5.430600 Ang

Lattice angles:
alpha =    90.000000 deg
beta  =    90.000000 deg
gamma =    90.000000 deg

Primitive vectors:
u_1 =      0.000000      2.715300      2.715300 Ang
u_2 =      2.715300      0.000000      2.715300 Ang
u_3 =      2.715300      2.715300      0.000000 Ang
```

(continues on next page)

(continued from previous page)

```
+------------------------------------------------------------+
| Bulk: Cartesian (Angstrom) / fractional                    |
+------------------------------------------------------------+
2
Bulk
Si    0.000000e+00  0.000000e+00  0.000000e+00    0.00000  0.00000  0.00000
Si    1.357650e+00  1.357650e+00  1.357650e+00    0.25000  0.25000  0.25000
```

Input lines 1–4 in the example above are standard Python commands. However, input line 5 creates the primitive silicon bulk using the *BulkConfiguration* class from ATK-Python, while input line 6 uses the *nlprint* functionality to print the main contents of the parameters defining the silicon bulk.

All the standard functionality of Python is available when you invoke QuantumATK. However, the main purpose of this chapter is to introduce the ATK-Python module. If you have no prior experience with Python, we encourage you to first go through the section sect1.python.basics.

4.3.1 ATK-Python

ATK-Python extends the standard Python environment with concepts and objects relevant for computational nano-scale physics and chemistry. This enables a simple, flexible, and intuitive way to operate QuantumATK: Use Python scripting to define nano-structures, atomic-scale simulators, and post-SCF analyses to be performed. Simply write the QuantumATK Python script and execute it.

For example, ATK-Python contains a periodic table of the elements, units such as Rydberg and Angstrom, methods for calculating the one-electron spectrum of a molecule, band structure of solids, and transmission spectra of nano-scale devices, as well of constructors for creating molecules, Bravais lattices, and devices.

The *DFT: LCAO* calculator may be largely implemented in highly efficient C++ routines, but setting up and executing DFT calculations is done using Python and ATK-Python commands. The same applies to setting up and executing analysis tools, as well as reading and writing of computational data. Moreover, the graphical user interface NanoLab uses ATK-Python to read data produced by QuantumATK. ATK-Python is therefore the scripting language that binds all QuantumATK products together, and is a platform on which other developers and companies can build applications and extend the functionality of QuantumATK products.

Important: All QuantumATK functionality available through ATK-Python is documented in the *QuantumATK Reference Manual*.

Python packages in QuantumATK

The QuantumATK distribution comes with all the standard Python modules, plus some additional packages which are useful for scientific computing, such as NumPy and SciPy. The list is long and can change with every release, the best way to get the full list for your local installation of QuantumATK is to run this command on Linux:

```
sh$ /quantumatk_installation_directory/atkpython/bin/python -m pip list
```

or this command on Windows:

```
PS> C:\quantumatk_installation_directory\atkpython\Scripts\python -m pip list
```

Using NumPy with QuantumATK

NumPy is the fundamental package for **scientific computing** with Python, since it can be used to perform advanced mathematical operations much faster than using ordinary Python lists. The NumPy module is therefore used throughout ATK-Python to store and manipulate values from analysis functions. NumPy objects resemble ordinary lists, but contain a lot more functionality, and QuantumATK ships with built-in NumPy support to easily facilitate its usage.

A few major differences between ordinary lists and NumPy arrays can be seen from this short example:

```
>>> from numpy import array
>>> a = array([1,2]) # a NumPy array
>>> a = a+[3,4]
>>> print(a)
[4, 6]
>>> b = [1,2]  # an ordinary Python list
>>> b = b+[3,4]
>>> print(b)
[1, 2, 3, 4]
```

As seen in the example above, NumPy arrays can in many ways be regarded as matrices. The following example underlines this:

```
>>> a = array([1,2],[3,4]])
>>> a = a *[3,4]
>>> print(a)
[[ 3  8]
[ 9 16]]
>>> print(a.trace())
19
>>> a.transpose()
>>> print(a)
[[ 3  9]
 [ 8 16]]
>>> print(a.trace())
19
```

Note in the above that the values 8 and 9 changed place in the matrix after applying the `transpose()` operation. However, as expected, the *trace* of the matrix remains the same.

NumPy arrays may also be converted into lists:

```
>>> a = array([1,2],[3,4]])
>>> print(a.tolist())
[[1, 2], [3, 4]]
```

There are many more possibilities using arrays from the NumPy module, and it is usually faster than iterating through for-loops or using ordinary lists!

More information can be found at the NumPy website, or by using the `dir()` command on a NumPy `array` object. Details on how NumPy can be used for improved performance can be found at the online resource Python Performance Tips.

Cloning of QuantumATK Python objects

It is possible to get a **copy** of an QuantumATK object using a method called *cloning*. This is done by adding a closed parenthesis after the object:

```
parameter_object = IterationControlParameters(tolerance=1.e-5)
parameter_object_clone = parameter_object()
```

Very importantly, it is possible to **modify the parameters** of the clone by specifying the new parameters during cloning:

```
parameter_object = IterationControlParameters(tolerance=1.e-5)
parameter_object_clone = parameter_object(max_steps=50, damping_factor=0.2)
```

Plotting using pylab

The following script uses the NumPy and matplotlib modules for creating a 2D plot:

```
import pylab
x = numpy.linspace(-1,5,10)
y = numpy.exp(x)
pylab.figure()
pylab.plot(x,y)
pylab.show()
```

Note that the NumPy package is automatically loaded with the ATK-Python module, so the `import numpy` statement is not really needed when running **atkpython**.

Note that QuantumATK also has an advanced plotting module for plotting publication quality 2D plots. The Plot module is automatically included in the ATK-Python module, and plots can be created, modified and shown both from scripts and through the NanoLab GUI. Read more in the *NanoLab Plot Reference Manual*.

4.3.2 Physical quantities and units

Units are a key concept in QuantumATK. All parameters that correspond to physical quantities, such as lengths, energies, voltages, etc., should be specified with an explicit unit. Similarly, all physical results returned from QuantumATK calculations also contain an explicit unit. PhysicalQuantity objects are created by multiplying the scalar, list, or array, containing the quantitys value(s), with the desired unit:

```
>>> a = [[1.0, 2.0], [3.0, 4.0]]*Angstrom
>>> t = 0.5*femtoSecond**-1
```

See below sections for physical units available in *QuantumATK*.

All PhysicalQuantity objects have two query methods:

- `inUnitsOf(Unit)`: Returns the numerical value in the specified unit as a numpy-array, respectively numpy-float object for scalar values.
- `convertTo(Unit)`: Returns the value of the PhysicalQuantity as a new PhysicalQuantity object in the specified unit.

Moreover, since the PhysicalQuantity class derives from numpy.array, PhysicalQuantity objects can be used, in most respects, as a numpy array. This means that many class methods of numpy arrays, such as sum(), max(), or reshape() can be used with PhysicalQuantity objects.

Element-wise operations between two PhysicalQuantity objects work as in numpy, e.g.:

```
>>> a = [[1.0, 2.0], [3.0, 4.0]]*Ang
>>> b = [[2.0, 2.0], [4.0, 4.0]]*nanoMeter
>>> c = a + b
>>> print(c)
[[ 21.  22.]
 [ 43.  44.]] Ang
```

Note, that addition and multiplication require compatible units for all operands.

Most numpy universal functions, as well as the two numpy functions numpy.dot and numpy.cross, work for PhysicalQuantity objects, in the same way as for numpy arrays.

Note, however, that most other numpy and python functions, e.g. numpy.arange, are not supported for PhysicalQuantity. In order to use them, the units have to be removed, via `inUnitsOf()` before the function is invoked:

```
>>> a = 5.0*Ang
>>> b = 1.0*nanoMeter
>>> delta = 0.5*Ang
>>> distances = numpy.arange(a.inUnitsOf(Ang), b.inUnitsOf(Ang), delta.inUnitsOf(Ang))
```

If the result of a PhysicalQuantity-operation is unitless, e.g:

```
>>> a = [[1.0, 2.0], [3.0, 4.0]]*Ang
>>> b = [[2.0, 2.0], [4.0, 4.0]]*Ang**-1
>>> c = a*b
>>> print(c)
[[ 21.  22.]
 [ 43.  44.]]
```

the result is directly returned as a numpy array, respectively as numpy float for scalar values.

Usage Examples

Getting a float value:

```
>>> a = 5*Angstrom
>>> print(a.inUnitsOf(nanoMeter))
0.5
```

Getting a PhysicalQuantity object:

```
>>> print(a.convertTo(nanoMeter))
0.5 nm
```

Physical quantities can be transformed with an exponent:

```
>>> a = 2. * Meter * Second**-2
>>> v = (2 * a * (1*Meter))**0.5
>>> print(v)
2.0 m/s
```

Inverse units are specified by using the exponent operator **:

```
>>> f = 2.2/Second
>>> print(f.inUnitsOf(Second**-1))
2.2
```

Units are attached to values by multiplication. Thus, to specify a length of 5 Bohr:

```
>>> a = 5*Bohr
```

By printing the value of the variable `a`, the unit will automatically be displayed:

```
>>> print(a)
5.0 Bohr
```

Units can also be composite. The unit for force is Newton, which is Joule per Meter. This is a rather awkward unit for nano-scale calculations, where something like electron volt per nm makes more sense. Any energy divided by a length is, however, a valid force unit, so to specify a force, write:

```
>>> F = 5*eV/Bohr
```

Next, multiply this by a length again and the result will be an energy:

```
>>> b = F*5*Bohr
>>> print(b)
25*eV
```

Some unit abbreviations are only available with the `Units` prefix:

```
>>> b = 5.1*Units.Ry
>>> print(b)
5.1 Rydberg
```

Units that by default are specified without a prefix, can also be given with a prefix:

```
>>> b = 5.1*Rydberg
>>> c = 5.1*Units.Rydberg
```

Units available in QuantumATK

The following units are made available when importing *QuantumATK*:

Table 4.4: Units available in QuantumATK. More units are available using the `Units` prefix.

Unit type	Name
Length units	nm
	nanoMeter
	Ang
	Angstrom
	Bohr
	Meter
Energy units	Rydberg
	eV
	meV

continues on next page

Table 4.4 – continued from previous page

	electronVolt
	Hartree
	J
	Joule
	Calorie
	kiloCaloriePerMol
	kiloJoulePerMol
Force units	Newton
	nanoNewton
Mass unit	kiloGram
Temperature unit	Kelvin
Time units	fs
	femtoSecond
	femtosecond
	ps
	picoSecond
	picosecond
	ns
	nanoSecond
	nanosecond
	microSecond
	microsecond
	milliSecond
	millisecond
	Second
	Minute
	Hour
	Day
Conductivity related units	Ampere
	Volt
	Siemens
	G0
	Coulomb
Pressure units	bar
	Pa
	GPa
Spin unit	hbar
Number unit	Mol
	mol
Angle units	Radians
	Degrees
Physical constants	boltzmann_constant
	planck_constant
	avogadro_number
	speed_of_light
	atomic_mass_unit
	hbar
	electron_mass
	elementary_charge
	vacuum_permitivity

4.3.3 Read and Write Support

Read and write functionality in *QuantumATK* is provided by two functions: *nlread* and *nlsave*. Storage of several objects per file is supported. Each object in native ATK files is associated with a unique identifier – the `object_id`. If a new entry is saved without specifying an `object_id`, the entry is appended to the file with an auto-generated `object_id`. If an `object_id` is specified which already is present in the file, the old entry is automatically deleted.

ATK natively supports the HDF5 file format (from QuantumATK version $\geq$ 2017). It is platform independent, i.e. the files can, for instance, be written on a Linux platform and later be read on a Windows platform. The internal data structure is performance-optimized.

HDF5 (Default File Format)

specification	HDF5 format
`object_id` (default)	classname_x with x being an increasing integer

HDF5 is the default file format for QuantumATK version $\geq$ 2017. The file format supports *Metatext*, and deleting objects – see *nldelete*. Due to the performance-optimized storage, the file size is not automatically reduced if objects have been deleted / overwritten. The free space can be reclaimed with *nlrepack*. The stored data can easily be accessed by hdf-view or by any program based on libhdf5.

4.3.4 Metatext

Most of the objects available in *QuantumATK* have support for Metatext. This feature allows the user to store additional text on an object. All *Configuration* and *Analysis* objects support this feature. The information is automatically written to / read from *HDF5* files. Access to the Metatext of an object `obj` is provided by two functions:

obj.setMetatext(metatext)	Sets the `metatext` on `obj`. `metatext` has to be of type str.
obj.metatext()	Returns the metatext of `obj`.

Moreover, with the utility functions *readMetatext* one can access the Metatext of an object stored in a file.

4.3.5 Spin

Spin is a flag. As such it cannot be constructed; `Spin()` is an invalid command. Instead, Spin provides derived classes (flags) to represent spin components and projections:

Spin.Up	The up component of a spinor (up-up component of a spin matrix).
Spin.Down	The down component of a spinor (down-down component of a spin matrix).
Spin.RealUpDown	The real part of the up-down component of a spinor (spin matrix.)
Spin.ImagUpDown	The imaginary part of the up-down component of a spinor (spin matrix).
Spin.All	All spin components.
Spin.Sum	The sum Spin.Up + Spin.Down
Spin.X	The spin projection along x (Spin.X = 2*Spin.RealUpDown).
Spin.Y	The spin projection along y (Spin.Y = -2*Spin.ImagUpDown).
Spin.Z	The spin projection along z (Spin.Z = Spin.Up - Spin.Down).
Spin.Unknown	Unknown spin.

Usage Example

Calculate the electron density for all spin and evaluate some components:

```
# Calculate the electron density for a given configuration.
ed_up = ElectronDensity(configuration, spin=Spin.All)

# Take some spin projections.
x = ed.spinProjection(spin=Spin.X)
y = ed.spinProjection(spin=Spin.Y)
z = ed.spinProjection(spin=Spin.Z)
s = ed.spinProjection(spin=Spin.Sum)
r = ed.spinProjection(spin=Spin.RealUpDown)
i = ed.spinProjection(spin=Spin.ImagUpDown)
u = ed.spinProjection(spin=Spin.Up)
d = ed.spinProjection(spin=Spin.Down)

# Evaluate for Spin.X at the origin.
data = ed.evaluate(0.0*Bohr, 0.0*Bohr, 0.0*Bohr, spin=Spin.X)
```

Note about Spin.All

Precisely which spin components are returned when calling an objects query method with `spin = Spin.All` depends on the queried object. E.g. `ElectronDensity.evaluate(x, y, z, spin=Spin.All)` returns a list of four electron density values at the grid point (x, y, z) corresponding to `Spin.Sum`, `Spin.X`, `Spin.Y`, and `Spin.Z`. In other cases (e.g. ExchangeCorrelationPotential), the returned array contains the values corresponding to the spinor components `Spin.Up`, `Spin.Down`, `Spin.RealUpDown`, and `Spin.ImagUpDown`. Refer to the objects documentation for details.

Note on Spin in low level interface functions

In all low level interface functions such as calculateHamiltonianAndOverlap, calculateDensityMatrix, calculateSelfEnergy etc., the following rules for the spin parameter apply:

- **UNPOLARIZED**: Valid spin parameters are Spin.Up and Spin.All, which both yield the same result in this case, as there is no designated spin direction in UNPOLARIZED calculations.
- **POLARIZED**: Valid spin parameters are Spin.All, Spin.Up, and Spin.Down. The default is Spin.All, in which case the function returns a pair of matrices, one for the Spin.Up and one for the Spin.Down component. For Spin.Up or Spin.Down, only the respective spin component is returned.
- **NONCOLLINEAR / SPINORBIT**: In noncollinear calculations, only Spin.All is an accepted parameter. The returned matrix contains the spin components Spin.Up, Spin.Down, Spin.UpDown, and Spin.DownUp in an interleaved fashion, see below for an example.

Examples

```
# Calculate the density matrix for a polarized system.
D = calculateDensityMatrix(polarized_configuration, spin=Spin.All)
# Extract the Spin.Up component.
D_uu = D[0]
# Extract the Spin.Down component.
D_dd = D[1]

# Calculate the density matrix for a noncollinear system.
```

(continues on next page)

(continued from previous page)

```
D = calculateDensityMatrix(noncollinear_configuration, spin=Spin.All)
# Get all up-up entries:
D_ud = D[::2,::2]
# Get all down-down entries:
D_du = D[1::2,1::2]
# Get all up-down entries:
D_ud = D[1::2,::2]
# Get all down-up entries:
D_du = D[::2,1::2]
```

4.3.6 Command Line Usage with QATK Shell Environments

Introduction

One of the most notable features of QuantumATK is the powerful graphical user interface, which makes it easy to develop complex workflows and analyze their results. Nevertheless, the command line (CLI) usage is common and sometimes necessary, for example in high performance computing (HPC) systems or when you want to experiment with new Python packages.

We present here two useful tools for these purposes:

- **QATK Shell Environment**: a convenient way to have all QuantumATK utilities at your fingertips when working in a CLI environment.
- **Python Virtual Environments**: the best way to experiment with additional Python packages in combination with QuantumATK modules.

In the following section you will find several examples of commands, the commands come after the command prompt which is here shortened to `sh$` for Bash on Linux and `PS>` for PowerShell on Windows.

Use QuantumATK effectively from CLI: Shell Environments

Starting from the W-2024.09 release, you can find an additional file in your QuantumATK installation, `atkpython/bin/activate_env` on Linux and `atkpython/Scripts/activate_env.ps1` on Windows. The first one is a Bash script and the second one a PowerShell script. These scripts make it easier to use ATKPython and other QuantumATK tools from a command line prompt, and the next two section will explain how.

Bash on Linux

From an open Bash shell you activate the environment by dot sourcing the script:

```
sh$ source <quantumatk_installation>/atkpython/bin/activate_env
```

You will notice a change in the command prompt written in the terminal, for example the common prompt `username@hostname$` will become `(qatk) username@hostname$`, the `(qatk)` prefix in the prompt text will help you remember you are inside the QATK Shell Environment.

Note: This can remind you of Python virtual environments, but it is not the same, because there is no virtualized environment, you are directly using the QuantumATK installation. The implications of this are explained in this section: *Customize the environment: Python venvs*.

Once the QATK Shell Environment is active, you have all the main QuantumATK binaries available in the shell, for example you can execute `atkpython` or execute the built-in `python` executable with access to all QuantumATK modules:

```
(qatk) sh$ atkpython -c 'print((5*Angstrom).convertTo(Bohr))'
9.44863062283 Bohr
(qatk) sh$ python -c 'from QuantumATK import Units; print((5*Units.Angstrom).
↪convertTo(Units.Bohr))'
9.44863062283 Bohr
```

The environment also gives access to a large class of executables, including those included with the pre-installed Python modules and the default Intel MPI installation. This means you can easily run an ATKPython script in parallel from inside this environment:

```
(qatk) sh$ mpiexec -n 4 atkpython /path/to/script.py
```

If you prefer to use a system installation of MPI instead of the Intel MPI setup included with QuantumATK, you can just add a flag when sourcing the activation script:

```
sh$ source <quantumatk_installation>/atkpython/bin/activate_env --use-system-mpi
```

In order to deactivate the environment and go back to the previous state of the shell:

```
(qatk) sh$ deactivate_qatk
```

The `(qatk)` prefix will disappear from the prompt, which means that the QATK Shell Environment is no longer active. If you do not want this prefix to be shown in the command prompt, you can assign any value to the `QATK_ENV_DISABLE_PROMPT` environment variable before activating the environment.

Advanced options for `activate_env`

Sourcing the `activate_env` script without additional options will set up the `LD_LIBRARY_PATH` and `PATH` environment variables to point to all the necessary bundled libraries and executables used for QuantumATK. This ensures that the QuantumATK tools can work in most environments.

By default, the `activate_env` script will add the following to the environment:

1.Prepend to `LD_LIBRARY_PATH`:

1.The C++ standard library `libstdc++`

2.A software rendering implementation of OpenGL `libGL`

3.The Intel MPI libraries

4.Many other 3rd party libraries located in `<quantumatk_installation>/atkpython/lib` (e.g. `libQt5Core`).

5.The python library used for QuantumATK

2.Append to `LD_LIBRARY_PATH`:

1.The CUDA libraries

3.Prepend to `PATH`:

1.The Intel MPI executables

The `activate_env` script comes with several options to change this behavior. You can see the full list of options by running:

```
sh$ <quantumatk_installation>/atkpython/bin/activate_env --help
```

Note: All libraries bundled with QuantumATK are using `DT_RUNPATH` to find their dependencies, so that QuantumATK can also run with an empty `LD_LIBRARY_PATH`. Since `LD_LIBRARY_PATH` has higher priority than `DT_RUNPATH` and could potentially be set to a conflicting value, we set `LD_LIBRARY_PATH` to the QuantumATK libraries by default. However, options to change this behavior are available.

Common cases where you might want to use these options are:

1. Systems with hardware accelerated OpenGL

   Use the system OpenGL libraries instead of the bundled software rendering implementation to get the best QuantumATK Nanolab performance:

   ```
   sh$ source <quantumatk_installation>/atkpython/bin/activate_env --use-system-gl
   ```

2. Systems with well integrated MPI libraries and executables (e.g. on HPC clusters)

   Try to use the system MPI installation, in case you encounter issues with the bundled Intel MPI:

   ```
   sh$ source <quantumatk_installation>/atkpython/bin/activate_env --use-system-mpi
   ```

3. Systems with a `libstdc++` newer than `libstdc++.so.6.0.31`

   Use the system `libstdc++` instead of the bundled one, as it will allow for the use of system tools in the QuantumATK environment:

   ```
   sh$ source <quantumatk_installation>/atkpython/bin/activate_env --use-system-stdcpp
   ```

4. Conflicts between the 3rd party libraries in `<quantumatk_installation>/atkpython/lib` and system tools you want to use in the QuantumATK environment

   In these cases you can try omitting the `lib` folder from the `LD_LIRBARY_PATH`:

   ```
   sh$ source <quantumatk_installation>/atkpython/bin/activate_env --prefer-system-libs
   ```

PowerShell on Windows

From an open Windows PowerShell shell you activate the environment with:

```
PS> & "<quantumatk_installation>\atkpython\Scripts\activate_env.ps1"
```

Just as in the case of Bash on Linux, the prompt text will show the prefix `(qatk)` when the environment is active, and it gives access to the already named executables and modules. As in the Linux case, you can assign any value to `$QATK_ENV_DISABLE_PROMPT` if you do not want this prefix to be displayed.

Note: The above command might fail because of restrictions in executing scripts inside PowerShell, if this happens the solution might be to modify the execution policy:

```
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

The Intel MPI setup is included by default in the environment, but if you want to use a separate system installation of MPI you can add a flag as argument to the script:

```
PS> & "<quantumatk_installation>\atkpython\Scripts\activate_env.ps1" -UseSystemMPI
```

You can easily deactivate the environment and go back to the original shell state:

```
(qatk) PS> deactivate_qatk
```

Note: The above applies to all flavours of PowerShell, it is tested on Windows PowerShell which is shipped with Windows 10/11, but it is compatible with the newer PowerShell Core and PowerShell 7.

Customize the environment: Python venvs

In this section we will go through some of the reasons why it is a good idea to use virtual environments with Python, with a special focus on ATKPython.

The problems

With the scripts shown in the previous section you can now use all the powerful tools shipped with QuantumATK, including all the pre-installed Python modules. In fact ATKPython includes NumPy, SciPy, Pandas and many other Python packages which are useful for scientific computing.

But what if you need to install an additional package?

We suppose that you need a package named `final-destination`. You could find instructions online suggesting to run:

```
python -m pip install final-destination
```

But this is usually not a good idea because it installs the package in the *system site-packages directory* which, in the case of the Python executable shipped with QuantumATK, resides inside the QuantumATK installation directory. Therefore, if you run the above command and you have permissions to write to the QuantumATK installation directory, you are going to overwrite the pre-installed files. This can cause two issues. First, if the `final-destination` package depends on, for example, a very recent version of NumPy, this will be installed on top of the pre-installed NumPy package. But this new version of NumPy has not been reviewed by the QuantumATK team, and parts of QuantumATK or pre-installed Python packages might not be compatible with it. The result could be that you are no longer able to start QuantumATK and you would then need to reinstall the software from scratch. Second, if the package installation goes fine and `final-destination` is now installed and works, it will still *pollute* the QuantumATK environment, adding possible sources of problems that the QuantumATK team could not consider when preparing the QuantumATK release. If there is any issue after installing a package this way, the first suggestion would again be to reinstall the software from scratch.

A slightly better alternative is:

```
python -m pip install --user final-destination
```

This command will install the `final-destination` package and all its dependencies to the QuantumATK user folder (`$HOME/.quantumatk` on Linux, `%USERPROFILE%\.quantumatk` on Windows). Unfortunately the problem outlined above still apply, the only positive difference is that you can remove the QuantumATK user folder instead of removing the entire installation. But the QuantumATK user folder also contains the project folders, the license configuration, and many other files you would probably prefer to keep.

The solution

All of the above problems can be solved by using Python Virtual Environments (venv), which allow you to create isolated and self-contained environments you can safely play around with, without any risk for the QuantumATK installation or user folder.

Assuming you are already inside an active QATK Shell Environment and you want to name the virtual environment `final-destination-venv`, you can create a Python venv and activate it with:

```
(qatk) sh$ python -m venv --system-site-packages final-destination-venv
(qatk) sh$ source final-destination-venv/bin/activate
```

The prompt will now show the prefix `(final-destination-venv) (qatk)` to represent the two active environments. Note that the order matters in this case, since the first in the prompt is the last one that has been activated and is therefore the one that has priority. Of course this prefix is a bit long, so we suggest using shorter names for your venvs or specifying a shorter prompt with the `--prompt` flag.

The commands above are almost identical in Windows PowerShell:

```
(qatk) PS> python -m venv --system-site-packages final-destination-venv
(qatk) PS> & "final-destination-venv\Scripts\Activate.ps1"
```

Now that you are inside a virtual environment you can feel free to install your needed package:

```
python -m pip install final-destination
```

This will be installed inside the virtual environment and will not affect anything outside of it. After you are done working with the virtual environment you can go back to the original shell by deactivating the active environments, starting by the last one that has been activated:

```
(final-destination-venv) (qatk) sh$ deactivate
(qatk) sh$ deactivate_qatk
```

You will see the prefixes disappear which means that the environments have been deactivated and your `final-destination` package will not get in your way until you activate the `final-destination-venv` environment again.

If you do not need to have direct access to the additional executables contained in the QuantumATK installation folder, and you do not need to run anything under MPI, then you can skip the QATK Shell Environment activation above:

```
sh$ <quantumatk_installation>/atkpython/bin/python -m pip --system-site-packages final-
↪destination-venv
```

This will create a virtual environment which still has access to the Python modules shipped with QuantumATK, and will work in most cases. If you encounter any issue with this approach while using the Python modules shipped with QuantumATK, just revert to the two steps procedure explained above.

The virtual environment folder will always be available on the system until you manually delete it, and the venv can be activated again at any time by *dot sourcing* the `activate` (`Activate.ps1`) script on Linux (Windows) as explained above. There is no need to create the venv every time with the `python -m venv` command.

You can have several venvs, each in its dedicated folder, and activate them when they are needed. This allows for experimentation and can help organizing your work, when developing several projects in parallel.

Note: It is not guaranteed that QuantumATK modules can be imported and used in a virtual environment created with a different version of QuantumATK.

Step by step examples

We include some examples which summarize in a few commands what we explained above.

Bash on Linux, QATK Shell Environment + Python venv:

```
sh$ QATK_DIR="/path/to/your/QuantumATK/installation"
sh$ VENV_DIR="/path/to/your/custom/venv"
sh$ source "${QATK_DIR}/atkpython/bin/activate_env"
(qatk) sh$ python -m venv --system-site-packages --upgrade-deps "$VENV_DIR"
(qatk) sh$ source "${VENV_DIR}/bin/activate"
(venv) (qatk) sh$ python -m pip install emoji
[...]
(venv) (qatk) sh$ python -m pip show emoji
[...]
Location: /path/to/your/custom/venv/lib/python3.11/site-packages
[...]
(venv) (qatk) sh$ mpiexec -n 4 python -c "import emoji; print(emoji.emojize('This is an␣
↪ambulance: :ambulance:'))"
This is an ambulance:
This is an ambulance:
This is an ambulance:
This is an ambulance:
(venv) (qatk) sh$ atkpython -c "import emoji; length=3.5*Units.m; print(emoji.emojize(f
↪'The :ambulance: is {length.convertTo(Bohr)} long.'))"
The  is 66140414359.8 Bohr long.
(venv) (qatk) sh$ deactivate
(qatk) sh$ deactivate_qatk
sh$
```

In the above example you can see that we have access to `atkpython` and `mpiexec` because we are inside the QATK Shell Environment, but we can also install a new package (emoji) which ends up inside the Python venv folder (notice the Location line in the output) without disturbing our QuantumATK installation. We have also added the `--upgrade-deps` flag which makes sure that `pip` is updated to the latest version inside the virtual environment.

PowerShell on Windows, only Python venv:

```
PS> $QATK_DIR="C:\path\to\your\QuantumATK\installation"
PS> $VENV_DIR="C:\path\to\your\custom\venv"
PS> "$QATK_DIR\atkpython\python.exe" -m venv --system-site-packages --upgrade-deps "
↪$VENV_DIR"
PS> & "${VENV_DIR}\Scripts\Activate.ps1"
(venv) PS> python -m pip install pyjokes emoji
[...]
(venv) PS> pyjoke
What is Benoit B. Mandelbrot's middle name? Benoit B. Mandelbrot.
(venv) PS> python -c "from ATKVersion import Version; import emoji; print(emoji.emojize(f
↪'The QATK version is {Version.version()}. :atom_symbol:'))"
The QATK version is W-2024.09.
(venv) PS> deactivate
PS>
```

In this case we are using the `python` executable shipped with QuantumATK to create a Python venv, so we can access QuantumATK modules (if we import them) and we can peacefully install new packages inside the virtual environment.

4.4 QuantumATK Reference Manual

4.4.1 Geometry

- *MoleculeConfiguration*
- *BulkConfiguration*
- *SurfaceConfiguration*
- *DeviceConfiguration*
- *NanoSheet*
- *NanoTube*
- *NanoRibbon*
- *NudgedElasticBand*
- *AlloyConfiguration*
- *AlloySite*
- *BoxRegion*
- *SphereRegion*
- *TubeRegion*
- *AtomicShift*
- *AtomicCompensationCharge*

Lattices

- bravaislattices
- SimpleCubic
- BodyCenteredCubic
- FaceCenteredCubic
- Rhombohedral
- Hexagonal
- SimpleTetragonal
- BodyCenteredTetragonal
- SimpleOrthorhombic
- BodyCenteredOrthorhombic
- FaceCenteredOrthorhombic
- BaseCenteredOrthorhombic
- SimpleMonoclinic
- BaseCenteredMonoclinic
- Triclinic
- UnitCell

4.4.2 Calculators

Common Parameters

- *IterationControlParameters*
- *AndersonMixer*
- *PulayMixer*
- *NumericalAccuracyParameters*
- *SingleContourIntegralParameters*
- *DoubleContourIntegralParameters*
- *DynamicalMatrix*
- *SlaterOrbital*
- *InitialSpin*
- *RandomSpin*
- *CheckpointHandler*
- *ExchangeCorrelation*
- poissonsolvers
- SelfEnergyCalculators
- preconditioners
- *CosmoSolvationParameters*

DFT Calculators

- *LCAOCalculator*
- *PlaneWaveCalculator*
- *DeviceLCAOCalculator*
- *FragmentCalculator*
- *BasisSet*
- *optimizeBasisSet*
- *ConfinedOrbital*
- *AnalyticalSplit*
- *PolarizationOrbital*
- *NumericalOrbital*
- *NormConservingPseudoPotential*
- *GrimmeDFTD2*
- *GrimmeDFTD3*

MBPT Calculators

- *GWCalculator*

Semi-Empirical Calculator

Common parameters

- *SemiEmpiricalCalculator*
- *DeviceSemiEmpiricalCalculator*
- *PairPotential*
- *ATK_U*
- *ATK_W*

Slater-Koster parameters

- *SlaterKosterCalculator*
- *DeviceSlaterKosterCalculator*
- *SlaterKosterOnsiteParameters*
- *SlaterKosterTable*
- *HotbitDirectory*
- *DFTBDirectory*
- *SlaterKosterHamiltonianParametrization*
- *NRLHamiltonianParametrization*

Huckel parameters

- *HuckelCalculator*
- *DeviceHuckelCalculator*
- *HuckelBasisParameters*
- *HuckelHamiltonianParametrization*

Counterpoise Correction

- *counterpoiseCorrected*

Low level entities

- *calculateHamiltonianAndOverlap*
- *calculateElectrodeHamiltonianAndOverlap*
- *calculateSelfEnergy*
- *calculateDensityMatrix*
- *orbitalInfo*
- *calculateRetardedGreenFunction*
- *calculateGreenFunctionComponent*
- *calculateDynamicalMatrixAndOverlap*
- *calculatePhononSelfEnergy*
- *calculatePhononRetardedGreenFunction*
- *calculatePhononGreenFunctionComponent*
- *calculateVelocity*
- *calculateTB09C*

4.4.3 Analysis

Common Analysis

- *AllElectronDensity*
- *BaderCharges*
- *CosmoRS*
- *ElectronDensity*
- *ElectronLocalizationFunction*
- *ExchangeCorrelationPotential*
- *ElectronDifferenceDensity*
- *ElectrostaticDifferencePotential*
- *EffectivePotential*
- *ExternalPotential*
- *ChemicalPotential*
- *GridData*
- *GridValues*
- *gridValues*
- *HTSTEvent*
- *MolecularEnergySpectrum*
- *TotalEnergy*
- *Forces*

- *Stress*
- *Eigenstate*
- *Eigenvalues*
- *MullikenPopulation*
- *SolvationEnergy*
- *SolventSurface*
- *SpecialThermalDisplacement*
- *PartialCharges*
- *Projection*
- *ProjectionList*
- *MakeTrajectory*
- *VibrationalMode*
- *IdealGasThermoChemistry*

Bulk Analysis

- *Bandstructure*
- *BornEffectiveCharge*
- *EffectiveBandstructure*
- *ComplexBandstructure*
- *FatBandstructure*
- *PhononBandstructure*
- *PhononDensityOfStates*
- *BlochState*
- *DensityOfStates*
- *DielectricTensor*
- *ProjectedDensityOfStates*
- *EffectiveMass*
- *GilbertDamping*
- *LocalBandstructure*
- *LocalDensityOfStates*
- *Mobility*
- *DeformationPotential*
- *HeisenbergExchange*
- *ElectroOpticalTensor*
- *ElectronPhononCoupling*
- *OpticalSpectrum*

- *OrbitalMoment*
- *PartialElectronDensity*
- *Polarization*
- *PiezoelectricTensor*
- *ElasticConstants*
- *LocalStress*
- *LocalStructure*
- *RamanSpectrum*
- *SecondHarmonicsGenerationSusceptibility*
- *ElectricFieldGradients*
- *ShieldingTensors*
- *STM*
- *ElectronicInverseParticipationRatio*
- *VibrationalInverseParticipationRatio*
- *CrystalThermoChemistry*

Device Analysis

- *IVCurve*
- *TransmissionSpectrum*
- *InelasticTransmissionSpectrum*
- *PhononTransmissionSpectrum*
- *TransmissionPathways*
- *TransmissionEigenvalues*
- *TransmissionEigenstate*
- *DeviceDensityOfStates*
- *LocalDeviceDensityOfStates*
- *CurrentDensity*
- *SpinTransferTorque*
- *SurfaceBandstructure*
- *ProjectedLocalDensityOfStates*
- *Photocurrent*
- *OneShotSelfConsistentBornApproximation*

DFT + U

- *calculateLocalScreenedCoulombCorrectionHubbardU*
- *calculateFiniteDifferenceLinearResponseHubbardU*
- *createLocalDFTUBasisSet*

4.4.4 Study

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

- *DynamicalMatrix*
- *HamiltonianDerivatives*
- *IVCharacteristics*
- *InelasticIVCharacteristics*
- *MagneticAnisotropyEnergy*
- *OptimizeDeviceConfiguration*
- *SusceptibilityDerivatives*
- *SurfaceProcessSimulation*
- *MomentTensorPotentialTraining*
- *ShockleyReadHallRecombination*
- *PointDefectLuminescence*

4.4.5 Dynamics and Optimization

Optimization

- *OptimizeGeometry*
- *OptimizeNudgedElasticBand*
- *OptimizeBulkFreeEnergy*
- *CrystalStructurePrediction*
- *EvolutionarySQS*
- *Trajectory*
- Optimization Algorithms
  - *LBFGS*
  - *FIRE*

Molecular Dynamics

- *MolecularDynamics*
- *MaxwellBoltzmannDistribution*
- *ZeroVelocities*
- *ConfigurationVelocities*
- *PlumedMetadynamics*
- *MDTrajectory*
- Molecular Dynamics Methods
  - *Langevin*
  - *NVEVelocityVerlet*
  - *NVTBerendsen*
  - *NPTBerendsen*
  - *NVTBussiDonadioParrinello*
  - *NVTNoseHoover*
  - *NPTMartynaTobiasKlein*
  - *NonEquilibriumHeatExchange*
  - *NonEquilibriumMomentumExchange*

Surface Process Simulation

- *SurfaceProcessSimulation*
- Hook Functions
  - *CleanVacuumRegion*
  - *RemoveElementFromSubstrate*
  - *ThermalizeSubstrate*

Moment Tensor Potential

- *ActiveLearningSimulation*
- *ConfigurationDataContainer*
- *MomentTensorPotentialTraining*
  - *MolecularDynamicsSnapshotsParameters*
  - *MomentTensorPotentialFittingParameters*
  - *NonLinearCoefficientsParameters*
  - *scanOverNonLinearCoefficients*
  - *CrystalInterfaceTrainingParameters*
  - *AlloyTrainingParameters*

– *RandomDisplacementsParameters*

– *MolecularConfigurationsParameters*

– *TrainingSet*

– *crystalTrainingRandomDisplacements*

– *readTrainingData*

Monte Carlo

- *AdaptiveKineticMonteCarlo*
- *ForceBiasMonteCarlo*
- *ForceBiasMonteCarloNPTBerendsen*
- *TimeStampedForceBiasMonteCarlo*
- *HybridMonteCarlo*
- *SemiGrandCanonicalMonteCarloHook*
- *SemiGrandCanonicalTransitionMatrixHook*
- *StaticMonteCarlo*

Hyperdynamics

- *CollectiveVariableHyperdynamics*
- *StaticHyperdynamics*
- *HyperdynamicsBondDistortion*
- *HyperdynamicsPositionDistortion*
- *IrregularTimeMeanSquareDisplacement*

Constraints

- *FixAtomConstraints*
- *RigidBody*
- *FixCenterOfMass*
- *BravaisLatticeConstraint*
- *SpaceGroupConstraint*
- *FixStrain*
- *BondConstraint*
- *ElectricFieldConstraint*

MD Analysis

- *AngularDistribution*
- *ChemicalCompositionProfile*
- *CoordinationNumber*
- *DensityProfile*
- *InfraredSpectrum*
- *IonicConductivity*
- *KineticEnergyDistribution*
- *MeanSquareDisplacement*
- *NearestNeighbor*
- *NeutronScattering*
- *PartialStructureFactor*
- *QNumbers*
- *RadialDistribution*
- *SelfDiffusion*
- *SpecificHeatCapacity*
- *TemperatureProfile*
- *VelocityAutocorrelation*
- *VelocityDistribution*
- *VibrationalDensityOfStates*
- *VoidsizeDistribution*
- *XRayScattering*

Image Interpolation Algorithms (NEB)

- *LinearInterpolation*
- *HalgrenLipscomb*
- *ImageDependentPairPotential*

4.4.6 Defects

Characterization

- *NamedPointDefect*
- *BaseDefectGenerator*
- *VacancyGenerator*
- *SubstitutionalGenerator*
- *InterstitialGenerator*

- *SplitInterstitialGenerator*
- *DefectPairGenerator*
- *Defects*
- *generateDefectsConfigurations*
- *BaseChemicalPotential*
- *ReferenceChemicalPotential*
- *CalculatedChemicalPotential*
- *PristineConfiguration*
- *IsotropicFiniteSizeCorrectionParameters*
- *ChargedPointDefectConfiguration*
- *InterfaceDefectAnalysis*

Migration

- *DefectMigrationPaths*
- *DefectDiffusionRates*
- *DefectDiffusivity*
- *generateRingConfigurations*

4.4.7 COSMO-RS

Material Descriptions

- *CosmoRealSpecies*
- *CosmoRealSolvent*
- *CosmoRealGas*
- *CosmoRealSolid*
- *CosmoRSMixture*

Properties

- *CosmoRSParameters*
- *Acidity*
- *GasSolubility*
- *LiquidEquilibrium*
- *PartitionCoefficient*
- *SolidSolubility*
- *VaporPressure*

Stored Data

- *CosmoRSSpeciesDatabase*
- *CosmoRSParameterSets*

4.4.8 Polymers

Builders And Equilibration

- *PolymerSequence*
- *PolymerMonteCarloBuilder*
- *CrosslinkReaction*
- *CrosslinkConnector*
- *CrosslinkBuilder*
- *ReactionTemplate*
- *ForceCappedEquilibration*
- *PolymerEquilibration*
- *CoarseGrainMapper*

Potentials

- *OPLSPotentialBuilder*
- *OPLSMinPotentialBuilder*
- *DreidingPotentialBuilder*
- *UFFPotentialBuilder*
- *QEqAtomicCharges*
- *ReaxFFAtomicCharges*
- *RepeatESPPartialCharges*

Analysis

- *tagPolymerMolecules*
- *RadiusOfGyration*
- *EndToEndDistances*
- *FreeVolume*
- *PolymerSegmentAnalyzer*
- *CohesiveEnergyDensity*
- *MolecularOrderParameter*

4.4.9 Input and Output

- *nlread*
- *nlsave*
- *nlprint*
- *setVerbosity*
- *showVerbosityRegions*
- *saveVerbositySettings*

4.4.10 Periodic Table

- *PeriodicTableElement*
- *allElements*

4.4.11 Utilities

- *SplineInterpolation1D*
- *MonkhorstPackGrid*
- *RegularKpointGrid*
- *AdaptiveGrid*
- *processIsMaster*
- *ParallelMapConfigurations*
- *ParallelEnergyAndForces*
- *randomSeed*
- *explicitHydrogenConfiguration*
- *unitedAtomsConfiguration*
- *moleculeFromSmiles*

4.4.12 Plot

Main reference: *NanoLab Plot Reference Manual*.

NanoLab Plot Reference Manual

The Plot module is built with the aim of making it easy to create publication quality scientific plots.

All plots can be saved using *Plot.save* and loaded using *nlread*. Specific plot items can be found using the *Plot.Plot* find method. An interactive plot window can be launched with the *Plot.show* function.

The NanoLab Plot framework helps you to edit, show and save plots created by the Analyzers in NanoLab. The Plot framework also makes it straight-forward to construct simple or complex plots from scratch, and to mix and match those plots with the plots created in NanoLab.

Note: All parts of the Plot framework are contained within the Plot module, meaning that all relevant classes and functions must be called with the Plot prefix.

Modifying Plots

Any plot constructed through the NanoLab GUI can be saved to an HDF5 file in the save dialog. The plots will show up in the NanoLab and can be opened again by the PlotViewer Plugin.

In a script the saved plot can be loaded by using the standard *nlread* function:

```
>>> plot = nlread('saved_plot.hdf5')[0]
```

The plot can then be shown using the special *Plot.show* function:

```
>>> Plot.show(plot, size=(600, 400))
```

This will show the plot in an interactive plot window with the full editing capability of NanoLab. The modified plot can then again be saved in the plot window; or it can be saved through scripting by using the special *Plot.save* function.

The plot can also be modified in a script. The easiest way to find the relevant plot items is through the *Plot.Plot*.find method:

```
>>> oxygen_lines = plot.find(types=Plot.Line, label='Oxygen')
```

Building Plots

All plots are constructed from a *Plot.PlotModel*.

```
>>> model = Plot.PlotModel()
```

The plots are then built from plot items, such as *Plot.Line*, *Plot.Scatter*, *Plot.HorizontalFill*, *Plot.Arrow*, *Plot.Measure*, etc.:

```
>>> model.addItem(
>>>     Line([0, 1, 2, 3, 4], [1, 2.2, 3.4, 6.7, 8.9])
>>>)
```

Multiple PlotModels can then be embedded in a *Plot.GridLayout* or *Plot.OverlayLayout* in order to build complex plots.

When a plot has been constructed it can be shown using the *Plot.show* function and saved using the *Plot.save* function.

Let us try to sketch a complete example that shows how to plot a dashed red *Plot.Line* with star markers on a log-plot. We also add a *Plot.CurveFit* to the line data.

```
# PlotModel with custom units.
model = Plot.PlotModel(x_unit=Angstrom, y_unit=eV)

# Change title
model.framing().setTitle('My plot')

# Set axis label and scale
```

(continues on next page)

(continued from previous page)

```
model.xAxis().setLabel('Length')
model.yAxis().setLabel('Energy')
model.yAxis().setScale(Plot.SCALES.LOG)

# Show legend.
model.legend().setVisible(True)

#Line.
line = Plot.Line([1, 2, 3, 4, 4] * nm, [1, 2.2, 3, 2.2, 1] * eV)
line.setLabel('My red line')
line.setColor('red')
line.setLineStyle(Plot.LINE_STYLES.DASHED)
line.setMarkerStyle(Plot.MARKER_STYLES.STAR)
line.setMarkerSize(8)

# Add fit.
fit = Plot.CurveFit('a * (x - x0)**2 + b')
line.addItem(fit)

# Add line.
model.addItem(line)

# Auto-adjust axis limits.
model.setLimits()

# Show plot and save afterwards to file and as raster.
Plot.show(model)
Plot.save(model, 'my_plot.hdf5')
Plot.save(model, 'my_plot.png')
```

Later, the plot can be loaded, and the line color can be changed to e.g. blue before it is saved again.

Framework

These are the core parts of the framework.

- *Plot.PlotModel*
- *Plot.show*
- *Plot.save*
- *Plot.Plot*

Plot Items

These are all the different items that can be added to or edited in a PlotModel.

- *Plot.Arrow*
- *Plot.AttributesLabel*
- *Plot.Average*
- *Plot.Axis*
- *Plot.Contour*
- *Plot.CurveFit*
- *Plot.Density*
- *Plot.Framing*
- *Plot.Group*
- *Plot.HorizontalBar*
- *Plot.HorizontalFill*
- *Plot.HorizontalLine*
- *Plot.HorizontalStackedFill*
- *Plot.Label*
- *Plot.Line*
- *Plot.LineGroup*
- *Plot.Measure*
- *Plot.PolynomialFit*
- *Plot.RollingAverage*
- *Plot.Scatter*
- *Plot.TransformedLine*
- *Plot.VerticalBar*
- *Plot.VerticalFill*
- *Plot.VerticalLine*
- *Plot.VerticalStackedFill*
- *Plot.WeightedLine*

Plot Layouts

These are the classes used for laying out plots in complex layouts.

- *Plot.GridLayout*
- *Plot.OverlayLayout*
- *Plot.PlotFrame*

Plot Constants

Some of the plot options must be supplied as special constants. All constants enumerations are uppercase, and their options are accessed using the usual dot notation, e.g. `Plot.COLOR_MODES.COLOR` or `Plot.SCALES.LOG`.

Enum	Constants
Plot.ALIGNMENTS	•TOP •CENTER •BOTTOM •LEFT •RIGHT
Plot.ARROW_STYLES	•FILLED •UNFILLED •NONE
Plot.BOUNDARY_CONDITIONS	•EXCLUDE •PERIODIC •MIRROR •REFLECT
Plot.COLOR_MODES	•COLOR •COLORS •MANY_COLORS •COLOR_MAP
Plot.DETAIL_LEVELS	•BASIC •MEDIUM •FULL
Plot.EDGES	•LEFT •RIGHT •TOP •BOTTOM •CENTER
Plot.KERNELS	•AVERAGE •GAUSSIAN
Plot.LAYOUT_MODES	•NORMAL •SHARE_X •SHARE_Y •SHARE_BOTH
Plot.LEGEND_LOCATIONS	•BEST •UPPER_LEFT •UPPER_CENTER •UPPER_RIGHT •CENTER_LEFT •CENTER •CENTER_RIGHT •LOWER_LEFT •LOWER_CENTER •LOWER_RIGHT •RIGHT
Plot.LINE_STYLES	•SOLID •DASHED

Full Plot module

All the plot classes and functions in alphabetical order:

Plot

Classes

Plot.Arrow

class Arrow(*tail_x*, *tail_y*, *head_x*, *head_y*)
An arrow pointing from tail to head and with custom color, line width and arrow head size.

Parameters

- **tail_x** (*PhysicalQuantity*) – The tail x coordinate.
- **tail_y** (*PhysicalQuantity*) – The tail y coordinate.
- **head_x** (*PhysicalQuantity*) – The head x coordinate.
- **head_y** (*PhysicalQuantity*) – The head x coordinate.

addItem(*item*, *emit=True*)
Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)
Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)
Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

applyStyle(*style_scheme=None*, *style_properties=None*)
Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

arrowScale()

Returns
The arrow scale.

Return type
float

color()

Returns
The label color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

headStyle()

Returns
The arrow head style.

Return type
ARROW_STYLES eum

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

`isMovable`()

Returns
True, if the label is movable.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`lineStyle`()

Returns
The arrow line style.

Return type
LINE_STYLES enum

`lineWidth()`

Returns
The arrow line width.

Return type
float

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

`positions()`

Returns
The postions of the arrow ends in the format (x1, y1, x2, y2).

Return type
None | 4-tuple of PhysicalQuantity

`removeItem(item, emit=True)`
Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

`removeSubscriber(subscriber)`
Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setArrowScale(*arrow_scale*)

Set the arrow scale.

Parameters
arrow_scale (*float*) – The arrow scale.

setColor(*color*)

Set the color.

Parameters
color (*str*) – A matplotlib formatted color.

setHeadPosition(*x*, *y*, *emit=True*)

Set the position.

Parameters

- **x** (*PhysicalQuantity*) – The x position.
- **y** (*PhysicalQuantity*) – The y position.
- **emit** (*bool*) – True, if signal should be emitted.

setHeadStyle(*head_style*)

Set the style of the arrow head.

Parameters
head_style (*ARROW_STYLES enum*) – The head style.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*LINE_STYLES enum*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The arorw line width.

setMovable(*movable*)

Set the movable property.

Parameters
movable (*bool*) – True, if the label should be movable.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setTailPosition(*x*, *y*, *emit=True*)

The tail position.

Parameters

- **x** (*PhysicalQuantity*) – The tail x position.
- **y** (*PhysicalQuantity*) – The tail y position.
- **emit** (*bool*) – True, if signal should be emitted.

setTailStyle(*tail_style*)

Set the style of the arrow tail.

Parameters
tail_style (*ARROW_STYLES enum*) – The tail style.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

`tags()`

Returns
The set of tags.

Return type
set

`tailStyle()`

Returns
The arrow tail style.

Return type
ARROW_STYLES enum

Plot.AttributesLabel

`class AttributesLabel(x, y)`

A Label that displays the attributes of a parent object in the plot.

Parameters

- **x** (*PhysicalQuantity*) – The x-coordinate of the label.
- **y** (*PhysicalQuantity*) – The y-coordinate of the label.

`addItem(item, emit=True)`

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

`addSubscriber(subscriber)`

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

`addTag(*args)`

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

`alignment()`

Returns
The vertical and horizontal alignment.

Return type
2-tuple of str

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The label color in hex format.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

detailLevel()

Returns
The detail level of the attributes.

Return type
enum

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

fontSize()

Returns
The font size in points.

Return type
float

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isBoxOn()

Returns
True, if the label has a box.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isMovable()

Returns
True, if the label is movable.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

classmethod isValidParent(*item*)

Check if an item is a valid parent.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

`position()`

Returns
The label postion.

Return type
None | 2-tuple of *PhysicalQuantity*

`removeItem(item, emit=True)`

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

`removeSubscriber(subscriber)`

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

`selectionDetails()`

Returns
The current selection details.

Return type
dict | None

`setAlignment(vertical, horizontal)`

Set the label alignment with respect to the position.

Parameters

- **vertical** (*str*) – The vertical alignment (top, center, bottom).
- **horizontal** (*str*) – The horizontal alignment (left, center, right).

`setBoxOn(box_on)`

Set whether a box should be drawn around the label text.

Parameters
box_on (*bool*) – True, if the label should have a box.

`setColor(color)`

Set the color.

Parameters
color (*str*) – The label color in hex format.

`setDetailLevel(level)`

Set the detail level.

Parameters
level (*enum*) – A level from the DETAIL_LEVELS enum.

setFontSize(*fontsize*)

Set the font size.

Parameters
fontsize (*float*) – The font size in points.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setMovable(*movable*)

Set whether the label should be movable.

Parameters
movable (*bool*) – True, if the label should be movable.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setPosition(*x*, *y*)

Set the label position.

Parameters

- **x** (*PhysicalQuantity*) – The x-position.
- **y** (*PhysicalQuantity*) – The y-position.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setText(*text*)

Set the text.

Parameters
text (*str*) – The label text.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

text()

Returns
The label text.

Return type
str

update(*data=None*)

Internal method.

Plot.Average

class Average

An Average can be added to a Line or a Scatter, and will then calculate the average value of its parent data.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

attributes(*latex=False*, *detail_level='full'*)

The current line attributes.

Parameters

- **latex** (*bool*) – If True, return the attributes in Latex format.
- **detail_level** (*DETAIL_LEVELS enum*) – The level of detail of the attributes (unused).

Returns
The current line attributes.

Return type
dict

bounds()

Returns
The bounds.

Return type
tuple of physicalQuantity

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

evaluate(*x_data*)

Evaluate the average for the given x_data.

Parameters
x_data (*PhysicalQuantity*) – The x-data.

Returns
The evaluated x data and y data.

Return type
2-tuple of PhysicalQuantity

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isAxisSpanned()

Returns
True if the fit line should span the x-axis range of the plot.

Return type
bool

isBoundsVisible()

Returns
True, if fit bounds are shown in the plot.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isInteractive()

Returns
Whether the user can interact with this line.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

classmethod isValidParent(*item*)

Check if an item is derivable.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The line label.

Return type
str

`lineStyle()`

Returns
The line style.

Return type
str | None

`lineWidth()`

Returns
The line width.

Return type
float

`markerEdgeColor()`

Returns
The edge color of the marker in hex format.

Return type
str

`markerEvery()`

Returns
The number of points between every marker.

Return type
int

`markerFaceColor()`

Returns
The face color of the marker in hex format.

Return type
str

`markerSize()`

Returns
The marker size.

Return type
float

`markerStyle()`

. :returns: The marker style. :rtype: str | None

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setBounds(*lower*, *upper*, *update=True*)

Set the derived line bounds.

Parameters

- **lower** (*float | PhysicalQuantity*) – The lower bound.
- **upper** (*float | PhysicalQuantity*) – The upper bound.
- **update** (*bool*) – True, if we should call update.

setBoundsVisible(*visible*)

Set whether to indicate the fit bounds in the plot.

Parameters
visible (*bool*) – If True, fit bounds will be shown in the plot.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x=None*, *y=None*)

Overload since derived lines should not expose data setting.

Parameters

- **x** (*iterable*) – The x values.
- **y** (*iterable*) – They values.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setInteractive(*interactive*)

Set whether the user can interact with this line.

Parameters
interactive (*bool*) – The desired state.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setSpanAxis(*span_axis*)

Set whether the fit line should span the x-axis range of the plot.

Parameters
span_axis (*bool*) – If True, the fit line should span the x-axis range of the plot.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

sourceData()

Checks if the source data is valid, and if yes, return the valid source data on the form: (xs, ys, x_min, x_max, x_unit, y_unit) Else returns None

Returns
The validated and clipped source data.

Return type
tuple | None

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

update(*data=None*)

Perform the update.

Parameters
data (*tuple | NoneType*) – The source data to use.

validSourceCoordinates(*xs*, *ys*)

True if the provided data coordinates are valid.

Parameters

- **xs** (ndarray) – The source x coordinates.
- **ys** (ndarray) – The source y coordinates.

Returns
True, if the coordinates are valid.

Return type
bool

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.Axis

class Axis(*unit=<Unit at 140734618035408 >*)

An Axis defines the x- or y-axis in a plot. The axis specifies limits, scale, label, fonts, ticks and grids.

Parameters
unit (Unit) – The unit of the axis.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

autoscalePadding()

. :returns: The padding used when autoscaling. :rtype: 2-tuple of float

checkAutoScaleIncludeZero()

Utility to check if including zero is possible and if the option is selected.

Returns
True, if we can and should include zero.

Return type
bool

color()

The color of the axis and the axis label.

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isAutoScaleZeroIncluded()

Returns
True, if the value zero should be explicitly included/shown on this axis.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isLocked()

Returns
True if the axis scroll is locked.

Return type
bool

`isMajorGridOn`()

Returns
True if major grid is shown.

Return type
bool

`isMinorGridOn`()

Returns
True if the minor grid is shown.

Return type
bool

`isMirrored`()

Returns
True, if ticks and label are mirrored.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`isZeroLineOn`()

Returns
True if zero line is shown.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items

•None : All items.

Parameters

•**selected** (*None | bool*) – True, if only selected items should be returned.

•**tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The label of the axis.

Return type
str

labelFontSize()

Returns
The axis font size in points.

Return type
float

limits()

Returns
The limits of the axis.

Return type
2-tuple of float

majorDivisions()

Returns
The number of major ticks.

Return type
Automatic | int

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

scale()

Returns
Axis scale (linear | log).

Return type
str

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAutoScaleIncludeZero(*include_zero*)

Parameters
include_zero (*bool*) – True, if the value zero should be explicitly included/shown on this axis.

setAutoscalePadding(*padding_lower*, *padding_upper*)

Set autoscale padding. Do not emit changes.

Parameters

- **padding_lower** (*float*) – The lower padding.
- **padding_upper** (*float*) – The upper padding.

setColor(*color*)

Set the color of the axis and the axis label.

Parameters
color (*str*) – The color to set in hex format.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*, *font_size=None*)

Set the label.

Parameters

- **label** (*str*) – The label.
- **font_size** (*float*) – The absolute font size in points.

setLabelFontSize(*font_size*)

Parameters
font_size (*float*) – The label font size in points.

setLimits(*lower_limit=None*, *upper_limit=None*)

Set axis limits.

Parameters

- **lower_limit** (*float*) – The lower limit.
- **upper_limit** (*float*) – The upper limit.

setLocked(*state*)

Set the locked state of the axis.

Parameters
state (*bool*) – True if the axis should be locked.

setMajorDivisions(*major_divisions*)

Set the number of major divisions of the axis.

Parameters
major_divisions (*Automatic | int*) – The number of divisions.

setMajorGridOn(*grid_on*)

Parameters
grid_on (*bool*) – Set major grid lines on/off.

setMinorGridOn(*grid_on*)

Parameters
grid_on (*bool*) – Set minor grid lines on/off.

setMirrored(*mirrored*)

Set ticks and labels mirrored..

Parameters
mirrored (*bool*) – True, if ticks should show up on the opposite site.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setScale(*scale*)

Set scale. When setting to log scale, this may also change the limits.

Parameters
scale (*str*) – The scale (linear, log or symlog).

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setTickFontSize(*font_size*)

Set the tick font size.

Parameters
font_size (*float*) – The absolute font size in points.

setTicks(*positions=None*, *labels=None*, *font_size=None*, *major_divisions=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *length=None*, *direction=None*)

Set tick positions and labels for the axis.

Setting positions=None resets the ticks to use default locators and formatters.

Parameters

- **positions** (*list*) – List of tick float positions.
- **labels** (*list*) – List of tick label strings.
- **font_size** (*float*) – The absolute font size in points.
- **major_divisions** (*[type]*) – [description].
- **length** (*float*) – The length of ticks in points.
- **direction** (*enum*) – The direction of ticks.

setUnit(*unit*)

Set the axis unit.

Parameters
unit (Unit) – The unit.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setZeroLineOn(*line_on*)

Parameters
line_on (*bool*) – True, if zero-line should be shown.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

tickDirection()

Returns
The direction of ticks.

Return type
enum

tickFontSize()

Returns
The tick font size in points.

Return type
float

tickLabels()

Returns
The list of tick labels.

Return type
list

tickLength()

Returns
The length of ticks in points.

Return type
float

ticks()

Returns
The a list of the tick positions as floats.

Return type
list

unit()

Returns
The unit of the axis.

Return type
Unit

Plot.Colors

class Colors

classmethod applyToItems(*items*, *offset=0*)
Color each plot item in a list with a distinct color.

Parameters

- **items** (*list of PlotItems*) – The items to color.
- **offset** (*int*) – The offset to start from.

classmethod list(*number=None*, *offset=0*)
Get colors in a list.

Parameters

- **number** (*int*) – The number of colors to get.
- **offset** (*int*) – The (optional) offset to start at.

Returns
List of colors.

Return type
list of str | str

Plot.Contour

class Contour(*x_data*, *y_data*, *z_data*)

A Contour displays three-dimensional data in a number of discrete levels, colored accordingly, and with an additional color bar.

The x, y data are 1D arrays spanning the regular z mesh. The z data is a 2D array mesh with values of the (x, y) points.

Parameters

- **x_data** (*PhysicalQuantity* | ndarray) – The one-dimensional x data.
- **y_data** (*PhysicalQuantity* | ndarray) – The one-dimensional y data.
- **z_data** (*PhysicalQuantity* | ndarray) – The two-dimensional z data.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the contour.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The contour face color in HEX format.

Return type
str

colorMap()

Returns
The contour color map.

Return type
str | None

colorMode()

Returns
The current color mode.

Return type
str

colors()

Returns
The currnetly set list colors.

Return type
List of colors | None

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the x-data or y-data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
2-tuple of PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isColorBarVisible()`

Returns
True, if colorbar is visible.

Return type
bool

`isFilled()`

Returns
True, if contour should be filled.

Return type
bool

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isLabelsVisible()`

Returns
True, if contour labels should be visible.

Return type
bool

`isSelectable()`

Returns
True, if the item is selectable.

Return type
bool

`isSelected()`

Returns
True, if the item is selected.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The label of the z-axis.

Return type
str

`labelFontSize()`

Returns
The label font size.

Return type
int

`labelsFontSize()`

Returns
The contour labels font size.

Return type
int

`levels()`

Returns
The number of contour levels.

Return type
int

limits()

Returns
The limits of the z-axis.

Return type
List of *PhysicalQuantity*

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

scale()

Returns
The z-axis scale (linear | log).

Return type
str

`selectionDetails()`

Returns
The current selection details.

Return type
dict | None

`setAlpha(alpha)`

Set the contour opacity as a float between zero and one.

Parameters
alpha (*float*) – The opacity.

`setColor(color)`

Set the uniform color of entire contour.

Note that this color is only used if the color mode has been set to color.

Parameters
color (*Any matplotlib supported color definition*) – The contour face color.

`setColorBarVisible(visible)`

Set colobar visibility.

Parameters
visible (*bool.*) – True, if colorbar should be shown.

`setColorMap(color_map)`

Set the contour color map.

Note that this color map is only used if the color mode has been set to color_map.

Parameters
color_map (*str*) – The color map.

`setColorMode(mode)`

Set the color mode.

Parameters
mode (*COLOR_MODES enum | str*) – The mode to set (color, colors, color_map).

`setColors(colors)`

Set the color sof the different contour levels.

Note that these colors are only used if the color mode has been set to colors.

Parameters
colors (*List of colors | None*) – The colors to set.

`setData(x_data, y_data, z_data)`

Set the data used in the contour.

Parameters

- **x_data** (*PhysicalQuantity* | ndarray) – The one-dimensional x data.
- **y_data** (*PhysicalQuantity* | ndarray) – The one-dimensional y data.
- **z_data** (*PhysicalQuantity* | ndarray) – The two-dimensional z data.

setFilled(*filled*)

Set whether the contour should be filled with colors.

Parameters
filled (*bool*) – True, if contour should be filled.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the z-axis label.

Parameters
label (*str*) – Set z-axis label.

setLabelFontSize(*font_size*)

Set label font size.

Parameters
font_size (*int*) – The label font size.

setLabelsFontSize(*font_size*)

Set contour labels font size.

Parameters
font_size (*int*) – The font size.

setLabelsVisible(*visible*)

True, if contour labels should be visible.

Only applies to unfilled contours.

Parameters
visible (*bool*) – True, if visible.

setLevels(*levels*)

Set number of contour levels.

Parameters
levels (*int*) – The number of different contour levels.

setLimits(*lower_limit=None*, *upper_limit=None*)

Set the z-axis limits.

Parameters

- **lower_limit** (*float*) – The lower z-axis limit.
- **upper_limit** (*float*) – The upper z-axis limit.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setScale(*scale*)

Set the z-axis scale to either linear or log.

Parameters
scale (*str*) – The z-axis scale (linear | log).

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setTickFontSize(*font_size*)

Set the color bar tick font size.

Parameters
font_size (*int*) – The font size.

setUnit(*unit*)

Set the unit of the z-scale.

Parameters
unit (Unit) – The z-axis unit.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setZData(*z_data*)

Set z data.

Parameters
z_data (*PhysicalQuantity* | ndarray) – The two-dimensional z data.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

tickFontSize()

Returns
The color bar tick font size.

Return type
int

unit()

Returns
The unit of the z-axis.

Return type
Unit

xData()

Returns
The one-dimensional x data.

Return type
PhysicalQuantity

yData()

Returns
The one-dimensional y data.

Return type
PhysicalQuantity

zData()

Returns
The two-dimensional z data.

Return type
PhysicalQuantity

Plot.CurveFit

class CurveFit(*expression=None*)
A CurveFit can be added to a Line or Scatter, and will then fit the supplied expression to the parent data.

Parameters
expression (*str*) – The expression for the fit function.

addItem(*item*, *emit=True*)
Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.

- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

attributes(*latex=False*, *detail_level='full'*)

A description of the fit attributes.

Parameters

- **latex** (*bool*) – If True, return the attributes in Latex format.
- **detail_level** (*DETAIL_LEVELS enum*) – The level of detail of the attributes.

Returns
A description of the fit attributes.

Return type
collections.OrderedDict

bounds()

Returns
The bounds.

Return type
tuple of physicalQuantity

color()

Returns
The line color.

Return type
str

conditions()

Returns
The fit conditions.

Return type
collections.OrderedDict

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

determination()

Returns
The fit coefficient of determination.

Return type
float | NoneType

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

evaluate(*x_data*)

Evaluate the curve fit for the given x_data.

Parameters
x_data (*PhysicalQuantity*) – The x-data.

Returns
The evaluated x data and y data.

Return type
2-tuple of PhysicalQuantity

expression()

Returns
The fit expression.

Return type
str

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isAxisSpanned()

Returns
True if the fit line should span the x-axis range of the plot.

Return type
bool

isBoundsVisible()

Returns
True, if fit bounds are shown in the plot.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isInteractive()

Returns
Whether the user can interact with this line.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

classmethod isValidParent(*item*)

Check if an item is derivable.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The line label.

Return type
str

`lineStyle()`

Returns
The line style.

Return type
str | None

`lineWidth()`

Returns
The line width.

Return type
float

`markerEdgeColor()`

Returns
The edge color of the marker in hex format.

Return type
str

`markerEvery()`

Returns
The number of points between every marker.

Return type
int

`markerFaceColor()`

Returns
The face color of the marker in hex format.

Return type
str

`markerSize()`

Returns
The marker size.

Return type
float

`markerStyle()`

. :returns: The marker style. :rtype: str | None

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parameters()

Returns
The polynomial coefficients.

Return type
numpy.ndarray

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

safeSetExpression(*expression*, *update=True*)

Safe setting of expression. Fails silently without changing anything.

Parameters

- **expression** (*str*) – The expression.
- **update** (*bool*) – True if the fit should be updated.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setBounds(*lower*, *upper*, *update=True*)

Set the derived line bounds.

Parameters

- **lower** (*float | PhysicalQuantity*) – The lower bound.
- **upper** (*float | PhysicalQuantity*) – The upper bound.
- **update** (*bool*) – True, if we should call update.

setBoundsVisible(*visible*)

Set whether to indicate the fit bounds in the plot.

Parameters
visible (*bool*) – If True, fit bounds will be shown in the plot.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setConditions(*conditions*, *update=True*)

Set conditions.

Parameters

- **conditions** (*dict or list of floats*) – The conditions to set.
- **update** (*bool*) – True, if the fit should be updated.

setData(*x=None*, *y=None*)

Overload since derived lines should not expose data setting.

Parameters

- **x** (*iterable*) – The x values.
- **y** (*iterable*) – They values.

setExpression(*expression*, *update=True*)

Set the fit expression.

Parameters

- **expression** (*str*) – The fit expression to set.
- **update** (*bool*) – True, if the fit should be updated.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setInteractive(*interactive*)

Set whether the user can interact with this line.

Parameters
interactive (*bool*) – The desired state.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setSpanAxis(*span_axis*)

Set whether the fit line should span the x-axis range of the plot.

Parameters
span_axis (*bool*) – If True, the fit line should span the x-axis range of the plot.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

sourceData()

Checks if the source data is valid, and if yes, return the valid source data on the form: (xs, ys, x_min, x_max, x_unit, y_unit) Else returns None

Returns
The validated and clipped source data.

Return type
tuple | None

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

update(*data=None*)

Perform the update.

Parameters
data (*tuple | NoneType*) – The source data to use.

validSourceCoordinates(*xs*, *ys*)

Check valid source coordinates.

Parameters

- **xs** (*numpy.ndarray*) – The x data values.
- **ys** (*numpy.ndarray*) – The y data values.

Returns
True, if the source coordinates are valid input for the fit algorithm.

Return type
bool

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Usage Examples

Let us first construct a line object containing the data that we wish to fit.:

```
line = Plot.Line([0, 1, 2, 3, 4, 5], [1.3, 2.3, 4.2, 4.6, 2.6, 1.1])
```

Fitting to a parabola, can e.g. be done like this:

```
fit = Plot.CurveFit('curvature * (x - x0)**2 + offset')
line.addItem(fit)
print(fit.parameters())
print('R2=', fit.determination())
```

This will output the following values for `curvature`, `x0` and `offset`:

```
{
    'curvature': -0.5017857159048791,
    'x0': 2.50854092811564,
    'offset': 4.14691159949586
}
R2=0.8912692142744284
```

Of course something similar could have been achieved using a *Plot.PolynomialFit*. However, CurveFit can also fit to e.g. a gaussian curve:

```
fit = Plot.CurveFit('a * gaussian(x - b, c)')
line.addItem(fit)
print(fit.parameters())
print('R2=', fit.determination())
```

And in this case it produces a slightly better fit as evidenced by the `R2` value:

```
{
    'a': 16.505234582600096,
    'b': 2.5597234857461135,
    'c': 1.6884506777324249
}
R2= 0.9739999352271602
```

Notes

CurveFit can fit to any expression `y = f(x)`. The expression can be composed of all valid python algebraic components like numbers, operators `+, -, /, *`, the power operator `**`, the imaginary unit `1j` and many more.

In addition a number of special functions can be used.

Table 4.5: Special functions available in NanoLab expressions.

Function	Notes
`abs`	Absolute value
`real`	Real part
`imag`	Imaginary part
`sin`	Sine (radians)

continues on next page

Table 4.5 – continued from previous page

Function	Notes
cos	Cosine (radians)
tan	Tangent (radians)
asin	Inverse sine
acos	Inverse cosine
atan	Inverse tangent
atan2(a, b)	Inverse tangent of a/b in correct quadrant.
exp	Exponential
expm1	exp(a) - 1
sinh	Hyperbolic sine
cosh	Hyperbolic cosine
tanh	Hyperbolic tangent
asinh	Inverse sinh
acosh	Inverse cosh
atanh	Inverse tanh
log	Natural logarithm
log10	Logarithm base 10
log2	Logarithm base 2
sqrt	Square root
erf	Error function
erfc	1 - erf
gamma	Euler gamma function
ceil	Ceiling function
floor	Floor function
theta	Heaviside theta function, theta(a) = 0 for a < 0 and 1 for a > 0.
step	Heaviside theta function
sign	Sign function
sum	Sum
cumsum	Cumulative sum
cumtrapz	Cumulative trapezoidal sum |
gradient	1D gradient
wofz	Faddeeva function
lorentzian(x, g)	Lorentzian function, where g is the full width at half maximum (FWHM).
gaussian(x, a)	Gaussian function, wheere a is the full width at half maximum (FWHM).
voigt(x, a, g)	Voigt profile. Convolution of gaussian and lorentzian function. Here, a is the gaussian FWHM, and g is the lorentzian FWHM.
arange(a, b, step=1)	Range of values from a to b with custom step.

Plot.Density

class Density(*x=None*, *y=None*, *weights=None*)

A density plot item to effeciently show the density of (very many) points.

Parameters

- **x** (PhysicalQauntity | ndarray) – The x coordinates.
- **y** (PhysicalQauntity | ndarray) – The y coordinates.
- **weights** (PhysicalQauntity | ndarray) – The optional weights.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

broadening()

Returns
The broadening.

Return type
float

color()

Returns
The color in hex format.

Return type
str

colorMap()

The color map to use..

Returns
The color map.

Return type
str

colorMode()

The current color mode.

When set to COLOR_MODES.COLOR, the density is colored using the current color. When set to COLOR_MODES.COLOR_MAP colors the density using the current color_map.

Returns
The color mode (colormap or color)

Return type
COLOR_MODES enum

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

dpi()

Returns
The resolution in dots per inch.

Return type
int

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
2-tuple of PhysicalQauntity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The density label.

Return type
str

`limits()`

Returns
The limits of the color axis.

Return type
2-tuple of float

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

`removeItem(item, emit=True)`

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

resampler()

Returns
The resampler.

Return type
PlotStorable with resample method | NoneType

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity.

Parameters
alpha (*float*) – The opacity.

setBroadening(*broadening*)

Set the broadening.

Parameters
broadening (*float*) – The broadening.

setColor(*color*)

Set the color.

Parameters
color (str | QColor) – The color to set.

setColorMap(*color_map*)

Set the color map.

Parameters
color_map (*str*) – The color map.

setColorMode(*mode*)

Set the color mode.

Parameters
mode (*COLOR_MODES enum*) – The mode to set.

setData(*x_data*, *y_data*, *weights=None*)

Set data.

Parameters

- **x_data** (PhysicalQauntity | ndarray) – The one-dimensional x data.
- **y_data** (PhysicalQauntity | ndarray) – The one-dimensional y data.
- **weights** (ndarray | None) – The optional weights.

setDpi(*dpi*)

Set the resolution in dots per inch.

Parameters
dpi (*int*) – The resolution in dots per inch.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the density label.

Parameters
label (*str*) – The label.

setLimits(*lower_limit*, *upper_limit*)

Set the color axis limit.

Parameters

- **lower_limit** (*float | NoneType*) – The lower limit.
- **upper_limit** (*float | NoneType*) – The upper limit.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setResampler(*resampler*)

Set the resampler used to dynamically upsample or downsample the data.

Parameters
resampler (*Any PlotStorable with a resample method | None*) – The resampler.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setWeights(*weights*)

Set the weights.

Parameters
weights (PhysicalQauntity | ndarray) – The weights.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

weights()

Returns
The weights if any.

Return type
ndarray | None

xData()

The 1D x-data.

Returns
The x data.

Return type
PhysicalQauntity | ndarray

yData()

The 1D y-data.

Returns
The y data.

Return type
PhysicalQauntity | ndarray

Plot.Framing

class Framing(*title=''*, *font_size=16.0*, *show_title=True*)

The Framing specifies properties of the visual frame of the plot model (not to be confused with Plot.PlotFrame). This includes properties such as the plot title, the border line width and the background color.

Parameters

- **title** (*str*) – The title text.
- **font_size** (*float*) – The title font size.
- **show_title** (*bool*) – If True, the title is visible.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

backgroundColor()

Returns
The background color of the plot in hex format.

Return type
str

borderLineWidth()

Returns
The border line width in points.

Return type
float

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

fontSize()

Returns
The font size.

Return type
float

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isBottomBorderVisible()

Returns
Whether or not the border is visible.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

`isLeftBorderVisible`()

Returns
Whether or not the border is visible.

Return type
bool

`isRightBorderVisible`()

Returns
Whether or not the border is visible.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isTitleVisible`()

Returns
Whether or not the title is visible.

Return type
bool

`isTopBorderVisible`()

Returns
Whether or not the border is visible.

Return type
bool

`isVisible`()

Not implemented for Framing.

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setBackgroundColor(*color*)

Set the background color of the plot.

Parameters
color (*str*) – The color to set in hex format.

setBorders(*line_width=None*, *show_left_border=None*, *show_right_border=None*, *show_top_border=None*, *show_bottom_border=None*)

Set border properties.

Parameters

- **line_width** (*int*) – The border line width.
- **show_left_border** (*bool*) – If True, the left(polar) border is visible for rectilinear(polar) projections.
- **show_right_border** (*bool*) – If True, the right(inner) border is visible for rectilinear(polar) projections.
- **show_top_border** (*bool*) – If True, the top border(end) is visible for rectilinear(polar) projections.
- **show_bottom_border** (*bool*) – If True, the bottom(start) border is visible for rectilinear(polar) projections.

setFontSize(*font_size*)

Set the font size for the plot title.

Parameters
font_size (*float*) – The absolute font size in points.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setText(*title*)

Set the title text.

Parameters
text (*str*) – The title text.

setTitle(*title*)

Set the title text.

Parameters
text (*str*) – The title text.

setTitleVisible(*visible*)

Set the title visibility.

Parameters
visible (*bool*) – If True, the title is visible.

setVisible()

Not implemented for Framing.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

text()

Returns
The title text.

Return type
str

title()

Returns
The title text.

Return type
str

Plot.GridLayout

class GridLayout(*rows=1*, *columns=1*)

Grid layout that sets up a grid where frames can be placed.

Items are placed at a row, column position and has a size, that is a multiple of rows or columns.

The grid can optionally sync the x- or y-axis of all immediately contained frames.

Parameters

- **rows** (*int*) – The number of rows.
- **columns** (*int*) – The number of columns.

addItem(*item*, *row=None*, *column=None*, *height=1*, *width=1*)

Add item to grid layout.

Parameters

- **item** (*LayoutItem based*) – The item to add.
- **row** (*int*) – The row to insert at.
- **column** (*int*) – The column to insert at.
- **height** (*int*) – The height (row span).
- **width** (*int*) – The width (column span).

addModel(*model*, *row=None*, *column=None*)

Convenience method for easily adding a plot model to the layout.

Parameters

- **model** (PlotModel) – The plot model to add.
- **row** (*int*) – The row where to add it.
- **column** (*int*) – The column where to add it.

Returns
The added plot frame.

Return type
PlotFrame

copy()

Returns
A copy of the layout.

Return type
BaseLayout

dimensions()

Returns
The dimensions (rows, columns).

Return type
2-tuple of int

find(*, *types=None*, *function=None*, ***kwargs*)

Find plot items within this layout filtered on types and properties.

Find all lines with e.g. the label Result, by writing:

```
layout.find(types=Line, label='Result')
```

Or find all Scatter plots with an outline:

```
layout.find(types=Scatter, outline=True).
```

Or all lines which has a label that starts with the characters band,:

```
def labelStartWithBand(item):
    return item.label().lower().startswith('band')

layout.find(
    types=Line,
    function=labelStartsWithBand
).
```

Parameters

- **types** (*None | PlotItem type*) – An optional type or multiple types, e.g. Line or Scatter.
- **function** (*None | function*) – An optional filter function taking the plot item as the only input.
- ****kwargs** (*dict*) – Optional properties which should be matched exactly for the items.

Returns
A list of the items matching the query.

Return type
list of PlotItem

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

isCompatible(*item*)

Check compatibility of item.

Parameters
item (*LayoutItem based*) – The item.

Returns
True, if compatible.

Return type
bool

itemAt(*row*, *column*)

Return the frame at the provided position.

Parameters

- **row** (*int*) – The row where the frame is.
- **column** (*int*) – The column where the frame is.

items()

Returns
The items in the layout.

Return type
list of LayoutItems.

mode()

Returns
The layout mode.

Return type
enumerator | int

models(*plotter=None*)

Get all models in the layout for a given plotter.

Parameters
plotter (*Plotter based | NoneType*) – The plotter or None for all plotters.

Returns
The models.

Return type
list

plotters()

Returns
The plotters in the layout.

Return type
set of Plotters

removeItem(*item*)

Remove item from layout.

Parameters
item (PlotFrame | *BaseLayout*) – The item to remove from the layout.

Returns
True, if removal succeeded. Otherwise False.

Return type
bool

replaceItem(*old_item*, *new_item*)

Replace an existing item.

Parameters

- **old_item** (*LayoutItem based*) – The old item to replace.

•**new_item** (*LayoutItem based*) – The new item the replaces the old item.

setDimensions(*rows*, *columns*)

Set layout dimensions.

Parameters

•**rows** (*int*) – Number of rows.

•**columns** (*int*) – Number of columns.

setMode(*mode*)

Set the layouts mode, i.e. if axes should be shared.

Parameters
mode (*enumerator | int*) – The layout type.

setSpacing(*spacing*)

Set spacing between layout items. This is given as a fraction of the average axis width and height.

Parameters
spacing (*float*) – The spacing.

spacing()

Returns
The spacing between layout items as a fraction sof the average axis width or height.

Return type
float

Usage Examples

A GridLayout arranges *Plot.PlotFrame* in a grid. A GridLayout can be constructed in the NanoLab GUI by dropping one plot next to another. See *Merge plots* for further details.

Below we show how to achieve the same functionality in a script. As an example we construct two *PlotModels*, the first showing density as a function of distance, the other showing energy as a function of distance. We build a GridLayout showing one model above the other and with their x-axises shared.

```
# Model showing the density as a function of distance.
density_model = Plot.PlotModel(Angstrom, Angstrom**-3)
density_model.addItem(Plot.Line([0, 1,  2] * Angstrom, [0.1, 0.2, 0.3] * Angstrom**-3))

# Frame the model.
density_frame = PlotFrame()
density_frame.addModel(density_model)

# Model showing an energy potential as a function of distance.
potential_model = Plot.PlotModel(Angstrom, eV)
potential_model.addItem(Plot.Line([0, 1, 2] * Angstrom, [2, 3, 4] * eV))

# Frame the model. Position y-axis to the right.
potential_frame = PlotFrame()
potential_frame.addModel(density_model)
potential_frame.title().setVisible(False)
```

(continues on next page)

(continued from previous page)

```
# Construct the GridLayout and add the two models such that they share the x-axis.
# Here we make sure that the bottom frame is twice as high as the first.
layout = Plot.GridLayout(rows=3, columns=1)
layout.setMode(Plot.LAYOUT_MODES.SHARE_X)
layout.addItem(density_frame, row=0, column=0, width=1, height=1)
layout.addItem(potential_frame, row=1, column=0, width=1, height=2)

# Show the plot.
Plot.show(layout)
```

Plot.Group

class Group(*label=None*)

A Group can contain any number of plot items.

Parameters
label (*str*) – The group label.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0*)

Propagate the limits of all children.

Parameters

- **axis_name** (*str*) – The axis name (x or y).
- **padding** (*float*) – The padding.

Returns
The limits.

Return type
NoneType | tuple of *PhysicalQuantity*

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The group label.

Return type
str

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the group label.

Parameters
label (*str*) – The label.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

Plot.HorizontalBar

class HorizontalBar(*bins*, *data*)

A bar plot showing data as horizontal filled bins.

The shape of the bins is determined in the following way: - If the number of bins equals the number of data values, the bins parameter is interpreted as the bin centers and bin widths are uniformly set to the match minimum distance between the bin centers. - If the number of bins is one larger than the number of data values, the bins parameter is interpreted as the edges of the bins. - If the number of bins is double the number of data values, the bins are instead interpreted as the left and right edge of each of the bins.

Parameters

- **bins** (Sequence of *PhysicalQuantity*) – The bins.

•**data** (Sequence of *PhysicalQuantity*) – The data values.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

•**item** (*PlotItem based*) – The item.

•**emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the bar plot.

Return type
int

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

•**style_scheme** (*dict*) – The style scheme.

•**style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

binCenters()

Returns
The center of each bin.

Return type
Sequence of *PhysicalQuantity*

binEdges()

. :returns: The left edges of each bin. :rtype: *PhysicalQuantity*

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColor()

Returns
The edge color of the bar plot.

Return type
str

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

The limits around the bar plot.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Optional padding.

Returns
The min/max data values for the requested axis.

Return type
2-tuple of *PhysicalQuantity*

faceColor()

Returns
The face color of the bar plot.

Return type
str

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

histogram()

Returns
The bin counts for each bin.

Return type
Sequence of *PhysicalQuantity*

hoverDetails()

Returns
The current hover details.

Return type
dict | None

`isHoverable`()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)
Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **`selected`** (*None | bool*) – True, if only selected items should be returned.
- **`tag`** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The label.

Return type
str

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)
Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)
Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

. :param alpha: The opacity of the bar plot. :type alpha: float

setColor(*color*)

Set the same color for both the bar face and the bar edge.

Parameters
color (*str*) – The color to set in hex format.

setData(*bins*, *data*, *emit=True*)

Set bins and histogram data.

The shape of the bins is determined in the following way: - If the number of bins equals the number of data values, the bins parameter is interpreted as the bin centers and bin widths are uniformly set to the match minimum distance between the bin centers. - If the number of bins is one larger than the number of data values, the bins parameter is interpreted as the edges of the bins. - If the number of bins is double the number of data values, the bins are instead interpreted as the left and right edge of each of the bins.

Parameters

- **bins** (Sequence of *PhysicalQuantity*) – The bins.
- **data** (Sequence of *PhysicalQuantity*) – The data values.
- **emit** (*bool*) – True, if the data should be emitted.

setEdgeColor(*color*)

Set the bar plot edge color.

Parameters
color (*str*) – The edge color of the bar plot.

setFaceColor(*color*)

. :param color: The face color of the bar plot. :type color: str

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the bar plot.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

xData()

Returns
The bin counts for each bin.

Return type
Sequence of *PhysicalQuantity*

yData()

Returns
The center of each bin.

Return type
Sequence of *PhysicalQuantity*

Plot.HorizontalFill

class HorizontalFill(*x*, *y_lower*, *y_upper*)
A HorizontalFill for filling an area between two lines

Parameters

- **x** (Sequence of *PhysicalQuantity*) – The x-values of the data.
- **y_lower** (Sequence of *PhysicalQuantity*) – The lower y-values in the fill.
- **y_upper** (Sequence of *PhysicalQuantity*) – The upper y-values in the fill.

addItem(*item*, *emit=True*)
Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)
Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)
Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the fill.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)
Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

copy()
Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColor()

Returns
The edge color of the fill in hex format.

Return type
str

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

The data extent.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data.

Returns
The minimum and maximum data values for the requested axis.

Return type
2-tuple of *PhysicalQuantity*

faceColor()

Returns
The edge color of the fill in hex format.

Return type
str

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label`()

Returns
The current label.

Return type
str

lower()

Returns
The lower values in the fill.

Return type
float

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)
Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)
Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity.

Parameters
alpha (*float*) – The opacity of the fill.

setColor(*color*)

Set the same color for both the face and the edge.

Parameters
color (*str*) – Set the color of the fill in hex format.

setEdgeColor(*color*)

Set the edge color of the fill.

Parameters
color (*str*) – The edge color of the fill in hex-format.

setFaceColor(*color*)

Set the face color of the fill.

Parameters
color (*str*) – The face color of the fill.in hex format.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the line.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

upper()

Returns
The upper values in the fill.

Return type
float

values()

Returns
The values in the fill.

Return type
float

Plot.HorizontalLine

class HorizontalLine(*y*)

A HorizontalLine shows a straight horizontal line.

Parameters
y (*PhysicalQuantity | float*) – The y-value of the line.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isSelectable()`

Returns
True, if the item is selectable.

Return type
bool

`isSelected()`

Returns
True, if the item is selected.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The line label.

Return type
str

`lineStyle()`

Returns
The line style.

Return type
str | None

`lineWidth()`

Returns
The line width.

Return type
float

`markerEdgeColor()`

Returns
The edge color of the marker in hex format.

Return type
str

`markerEvery()`

Returns
The number of points between every marker.

Return type
int

`markerFaceColor()`

Returns
The face color of the marker in hex format.

Return type
str

`markerSize()`

Returns
The marker size.

Return type
float

`markerStyle()`

. :returns: The marker style. :rtype: str | None

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

`removeItem(item, emit=True)`

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

`removeSubscriber(subscriber)`

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

`selectionDetails()`

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*y*)

Set y-data.

Parameters
y (*PhysicalQuantity*) – The y-data.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.HorizontalStackedFill

class HorizontalStackedFill(*y_values*, *values*)

A HorizontalStackedFill shows multiple fills laid out next to each other.

Parameters

- **y_values** (Sequence of *PhysicalQuantity*) – The y-values of the data.
- **values** (2D *PhysicalQuantity*) – The values to plot in the stack.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alphas()

Returns
The opacities of the fills.

Return type
list

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColors()

Returns
The edge colors of the fills.

Return type
list

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

faceColors()

Returns
The face colors of the fills.

Return type
list

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isSelectable()`

Returns
True, if the item is selectable.

Return type
bool

`isSelected()`

Returns
True, if the item is selected.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

labels()

Returns
The labels of the fills.

Return type
list

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlphas(*alphas*)

Set the alphas for all fills in the stack.

Parameters
alphas (*list*) – The opacities of the fills.

setColors(*colors*)

Set the same color for both the face and the edge for all filss in the stack.

Parameters
colors (*list*) – The colors of the fills.

setEdgeColors(*colors*)

Set the edge colors for all fills in the stack.

Parameters
colors (*list*) – The edge colors of the fills.

setFaceColors(*colors*)

Set the face colors for all fills in the stack.

Parameters
colors (*list*) – The face colors of the fills.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabels(*labels*)

Set the labels for all fills in the stack.

Parameters
labels (*list*) – The labels of the fills.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

values()

Returns
The values of the data.

Return type
2D *PhysicalQuantity*

Plot.Label

class Label(*x*, *y*, *text=''*)

A Label displays text in a plot.

Parameters

- **x** (*PhysicalQuantity*) – The x-coordinate of the label.
- **y** (*PhysicalQuantity*) – The y-coordinate of the label.
- **text** (*str*) – The label text.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alignment()

Returns
The vertical and horizontal alignment.

Return type
2-tuple of str

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The label color in hex format.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

`fontSize()`

Returns
The font size in points.

Return type
float

`classmethod fromProperties(`*properties*`)`

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isBoxOn()`

Returns
True, if the label has a box.

Return type
bool

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isMovable()`

Returns
True, if the label is movable.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`model`()

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order`()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

position()

Returns
The label postion.

Return type
None | 2-tuple of *PhysicalQuantity*

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlignment(*vertical*, *horizontal*)

Set the label alignment with respect to the position.

Parameters

- **vertical** (*str*) – The vertical alignment (top, center, bottom).
- **horizontal** (*str*) – The horizontal alignment (left, center, right).

setBoxOn(*box_on*)

Set whether a box should be drawn around the label text.

Parameters
box_on (*bool*) – True, if the label should have a box.

setColor(*color*)

Set the color.

Parameters
color (*str*) – The label color in hex format.

setFontSize(*fontsize*)

Set the font size.

Parameters
fontsize (*float*) – The font size in points.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setMovable(*movable*)

Set whether the label should be movable.

Parameters
movable (*bool*) – True, if the label should be movable.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setPosition(*x*, *y*)

Set the label position.

Parameters

- **x** (*PhysicalQuantity*) – The x-position.
- **y** (*PhysicalQuantity*) – The y-position.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setText(*text*)

Set the text.

Parameters
text (*str*) – The label text.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

text()

Returns
The label text.

Return type
str

Plot.Line

class Line(*x*, *y*, *tooltips=None*, *selectable=True*)

A line display a line in a plot.

Parameters

- **x** (Sequence | *PhysicalQuantity* | ndarray) – The x-data of the line.
- **y** (Sequence | *PhysicalQuantity* | ndarray) – The y-data of the line.
- **tooltips** (*list | list of str*) – The line tooltips.
- **selectable** (*bool*) – True, if the line is selectable.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data extent.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label`()

Returns
The line label.

Return type
str

`lineStyle`()

Returns
The line style.

Return type
str | None

`lineWidth`()

Returns
The line width.

Return type
float

markerEdgeColor()

Returns
The edge color of the marker in hex format.

Return type
str

markerEvery()

Returns
The number of points between every marker.

Return type
int

markerFaceColor()

Returns
The face color of the marker in hex format.

Return type
str

markerSize()

Returns
The marker size.

Return type
float

markerStyle()

. :returns: The marker style. :rtype: str | None

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x=None*, *y=None*, *tooltips=None*)

Set data.

Parameters

- **x** (Sequence | *PhysicalQuantity* | ndarray) – The x values.
- **y** (Sequence | *PhysicalQuantity* | ndarray) – The y values.
- **tooltips** (*list of str | str*) – Tooltips for each entry.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

tooltips()

Returns
The tooltips.

Return type
list of str | NoneType

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.LineGroup

class LineGroup(*items=None*)

Group containing multiple lines.

The group makes it easy to handle many lines.

It also has better performance when drawing lines, but unfortunately only supports a subset of the normal line properties. This means that e.g. markers cannot be drawn, and that visibility is only controlled for all line in the line group.

The line group also shows up in the legend as a single item.

Parameters
items (*list*) – List of Line instances.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the group.

Return type
float | None

applyStyle(*style_scheme=None*, *style_properties=None*)

Override PlotItem.applyStyle.

Apply the style of the LineGroup to the lines.

Parameters

- **style_scheme** (*dict*) – The style scheme.

•**style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style.

colors()

Returns
The colors.

Return type
list

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

•**key** (*str*) – The key to emit.

•**value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Set extent.

Parameters

•**axis_name** (*str*) – The axss name (x or y).

•**padding** (*float*) – The extra padding.

Returns
The extent.

Return type
Sequence of PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

`isHoverable`()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)
Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The current label.

Return type
str

lineStyles()

Returns
A list of line style for each item.

Return type
list

lineWidths()

Returns
A list of line widths for each item.

Return type
list

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the alpha/opacity, for all items.

Parameters
alpha (*float*) – The opacity.

setColor(*color*)

Set the line color for all items.

Parameters
color (*Any matplotlib supported color definition*) – The color.

setColors(*colors*)

Set the line color for all items.

Parameters
color (*Any matplotlib supported color definition*) – The color.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setItems(*items*)

Set items.

Parameters
items (*items.*) – The items to set.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the linegroup.

setLineStyle(*line_style*)

Set the line style, for all items.

Parameters
line_style (*str | None*) – The line style.

setLineStyles(*line_styles*)

Set the line style, for all items.

Parameters
line_style (*list*) – The line style list.

setLineWidth(*line_width*)

Set the line width, for all items.

Parameters
line_width (*float*) – The line width.

setLineWidths(*line_widths*)

Set the line width, for all items. return list(self._properties[items])

Parameters
line_width (*list*) – The line width list.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *information=None*)

Set the selected property. Overloads the base class method.

Parameters

- **selected** (*bool*) – True, if item given by index should be selected.
- **information** (*dict*) – information of the pick event.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

Plot.Measure

class Measure(*tail_x*, *tail_y*, *head_x*, *head_y*)

A Measure shows the distance between its tail-point and head-point along the x- and y-axis.

A Measure can be interactively moved with the mouse and can snap to nearby points on e.g. a Line.

Parameters

- **tail_x** (scalar *PhysicalQuantity*) – The tail x coordinate.
- **tail_y** (scalar *PhysicalQuantity*) – The tail y coordinate.
- **head_x** (scalar *PhysicalQuantity*) – The head x coordinate.
- **head_y** (scalar *PhysicalQuantity*) – The head x coordinate.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.

•**style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The measure color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

•**key** (*str*) – The key to emit.

•**value** (*object*) – The value to emit.

fontSize()

Returns
The measure label font size.

Return type
float

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isMovable()

Returns
True, if the label is movable.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

isSnapping()

Returns
True, if snapping is active.

Return type
bool

isVerticalLabelFlipped()

Returns
True, if the label displaying measurement along the vertical axis is flipped.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

isXShown()

Returns
True, if the x-measurement is shown.

Return type
bool

isYShown()

[summary].

Returns
True, if the y-measurement should be shown.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

labelPosition()

Returns
The label position in terms of whether they are inside or outside of the arrows.

Return type
enum

lineWidth()

Returns
The measure line width.

Return type
float

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

positions()

Returns
The postions of the arrow ends in the format (x1, y1, x2, y2).

Return type
None | 4-tuple of PhysicalQuantity

precision()

Returns
The number of significant digits shown the in measurement labels.

Return type
int

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setColor(*color*)

Set the color of the measure.

Parameters
color (*str*) – A matplotlib compatible color.

setFontSize(*fontsize*)

Set the font size of the measure labels.

Parameters
fontsize (*float*) – The font size.

setHeadPosition(*x*, *y*, *emit=True*)

Set the position.

Parameters

- **x** (*PhysicalQuantity*) – The x position.
- **y** (*PhysicalQuantity*) – The y position.
- **emit** (*bool*) – True, if signal should be emitted.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabelPosition(*position*)

Set the label position to be either inside or outside of the arrows.

Parameters
position (*enum*) – The position choice.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The measure line width.

setMovable(*movable*)

Set the movable property.

Parameters
movable (*bool*) – True, if the label should be movable.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setPrecision(*digits*)

Set the number of significant digits to show in the measurement labels.

Parameters
digits (*int*) – The number of significant digits.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setSnapping(*snap*)

Set whether the measurement should snap to points on *Line*, *Scatter* or *LineGroup*.

Parameters
snap (*bool*) – True, if the measurement should snap.

setTailPosition(*x*, *y*, *emit=True*)

The tail position.

Parameters

- **x** (*PhysicalQuantity*) – The tail x position.
- **y** (*PhysicalQuantity*) – The tail y position.
- **emit** (*bool*) – True, if signal should be emitted.

setVerticalLabelFlipped(*flipped*)

Set whether the label displaying measurement along the vertical axis is rotated 90 degrees.

Parameters
flipped (*bool*) – True, if the label displaying measurement along the vertical axis is flipped.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setXShown(*x_measure_shown*)

Set whether to show the x-meaurement.

Parameters
x_measure_shown (*bool*) – True, if x-measurement should be shown.

setYShown(*y_measure_shown*)

Set whether to show the y-meaurement.

Parameters
y_measure_shown – if y-measurement should be shown.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

`tags()`

Returns
The set of tags.

Return type
set

Plot.MultiLineResampler

`class MultiLineResampler(sampling=None)`

A resampler for resampling data in a `Plot-Density` item. Works really well for data representing multiple intersecting lines.

This resampler is essential if the size of the saved plot should not explode.

Provide 1D x-data and an array of multiple line data e.g. 2D y-data, and perhaps some extra weights data in the same shape as the y-data.

The resampler then upsamples the data using sampling number of linear spaced points in the x-data.

Parameters
sampling (*int | NoneType*) – The number of sampling points to use.

`copy()`

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

`classmethod fromProperties(properties)`

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`resample(x_data, y_data, weights=None)`

Resample the data.

Parameters

- **x_data** (PhysicalQauntity | ndarray) – The 1D x-data.
- **y_data** (PhysicalQauntity | ndarray) – The 1D or 2D y-data.
- **weights** (ndarray) – Optional weirhts

Returns
The upsampled data.

Return type
2 or 3-tuple of ndarrays

sampling()

Returns
The sampling.

Return type
int

setSampling(*sampling*)

Set the sampling.

Parameters
sampling (*int*) – The sampling.

Plot.OverlayLayout

class OverlayLayout

An overlaying plot layout that can contain exactly two frames, shown as an overlay on the same plot.

The two frames can share either the x-axis or the y-axis. The actual mode can take values of either SHARE_X or SHARE_Y.

addItem(*item*)

Add a frame to the layout.

Raise an error if more than two frames are added.

Parameters
item (PlotFrame) – The frame.

copy()

Returns
A copy of the layout.

Return type
BaseLayout

find(*, *types=None*, *function=None*, ***kwargs*)

Find plot items within this layout filtered on types and properties.

Find all lines with e.g. the label Result, by writing:

```
layout.find(types=Line, label='Result')
```

Or find all Scatter plots with an outline:

```
layout.find(types=Scatter, outline=True).
```

Or all lines which has a label that starts with the characters band,:

```
def labelStartWithBand(item):
    return item.label().lower().startswith('band')

layout.find(
```

(continues on next page)

(continued from previous page)

```
    types=Line,
    function=labelStartsWithBand
).
```

Parameters

- **types** (*None | PlotItem type*) – An optional type or multiple types, e.g. Line or Scatter.
- **function** (*None | function*) – An optional filter function taking the plot item as the only input.
- ****kwargs** (*dict*) – Optional properties which should be matched exactly for the items.

Returns
A list of the items matching the query.

Return type
list of PlotItem

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

items()

Returns
The items in the layout.

Return type
list of LayoutItems.

mode()

Returns
The overlay mode.

Return type
enumerator | int

models(*plotter=None*)

Get all models in the layout for a given plotter.

Parameters
plotter (*Plotter based | NoneType*) – The plotter or None for all plotters.

Returns
The models.

Return type
list

plotters()

Returns
The plotters in the layout.

Return type
set of Plotters

removeItem(*item*)

Remove a frame from the layout.

Parameters
item (PlotFrame) – The frame to remove.

replaceItem(*old_item*, *new_item*)

Replace an existing item.

Parameters

- **old_item** (*LayoutItem based*) – The old item to replace.
- **new_item** (*LayoutItem based*) – The new item the replaces the old item.

setMode(*mode*)

Set the overlay mode.

Note that this layout does not have a LAYOUT_MODES.NORMAL layout mode, but only supports LAYOUT_MODES.SHARE_X or LAYOUT_MODES.SHARE_Y.

Parameters
mode (*enumerator | int*) – The overlay layout mode.

Usage Examples

An OverlayLayout lays two *Plot.PlotFrame* over one another. Each PlotFrame has its own axises. The most common usage is to show two PlotFrames together which only share a single axis.

An OverlayLayout can be constructed in the NanoLab GUI by dropping one plot on top of another. See the *Merge plots* part of the Plotting guide for further details.

Below we show how to achieve the same functionality in a script. As an example we construct two *PlotModels*, the first showing density as a function of distance, the other showing energy as a function of distance. They are overlaid on top of each other with a shared x-axis.

```
# Model showing the density as a function of distance.
density_model = Plot.PlotModel(Angstrom, Angstrom**-3)
density_model.addItem(Plot.Line([0, 1,  2] * Angstrom, [0.1, 0.2, 0.3] * Angstrom**-3))

# Frame the model.
density_frame = PlotFrame()
density_frame.addModel(density_model)

# Model showing an energy potential as a function of distance.
potential_model = Plot.PlotModel(Angstrom, eV)
potential_model.addItem(Plot.Line([0, 1, 2] * Angstrom, [2, 3, 4] * eV))

# Frame the model. Position y-axis to the right.
potential_frame = PlotFrame()
```

(continues on next page)

(continued from previous page)

```
potential_frame.addModel(density_model)
potential_frame.yAxis().setMirrored(True)
potential_frame.title().setVisible(False)

# Construct the OverlayLayout and add the two models such that they share the x-axis.
overlay = Plot.OverlayLayout()
overlay.setMode(Plot.LAYOUT_MODES.SHARE_X)
overlay.addItem(density_frame)
overlay.addItem(potential_frame)

# Show the plot.
Plot.show(overlay)
```

Plot.Plot

class Plot(*layout*, *size*)

A Plot wraps a plot layout such that it can be saved to and loaded from an HDF5 file.

Parameters

- **layout** (*BaseLayout based*) – The plot layout to wrap.
- **size** (*2-tuple of floats*) – The pixel size of the layout.

find(*, *types=None*, *function=None*, ***kwargs*)

Find plot items within this plot filtered on type and properties.

Find all lines with e.g. the label Result, by writing:

```
layout.find(types=Line, label='Result')
```

Or find all Scatter plots with an outline:

```
layout.find(types=Scatter, outline=True).
```

Or all lines which has a label that starts with the characters band,:

```
def labelStartWithBand(item):
    return item.label().lower().startswith('band')

layout.find(
    types=Line,
    function=labelStartsWithBand
).
```

Parameters

- **types** (*None | PlotItem type*) – An optional type or types, e.g. Line or Scatter.
- **function** (*None | function*) – An optional filter function taking the plot item as the only input.
- ****kwargs** (*dict*) – Optional properties which should be matched exactly for the items.

Returns

A list of the items matching the query.

Return type
list of PlotItem

`layout()`

Returns
The wrapped plot layout.

Return type
BaseLayout based

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`size()`

Returns
The layout pixel size.

Return type
2-tuple of floats

`uniqueString()`

Return a unique string representing the state of the object.

Plot.PlotFrame

`class PlotFrame(use_frame_model=True)`

A plot frame lives inside a layout, and contains one or more plot models.

In the matplotlib front end, a frame is plotted as exactly one main matplotlib Axes object. However, the frame may control multiple auxiliary axes like e.g. colorbars.

The plot frame plots all the items contained in the plot models. Special care must be taken with the title, legend, axiss and user annotations. Each model contains their own instances of these items, and it can be unclear how, for example, the x-axis of one plot model should be reconciled with another.

Instead, the frame uses only a single x-axis, y-axis etc. There are two implementations:

1. The frame uses a frame model, and ignores all the model titles, legends etc. This is the implementation for plots that are saved. This also means that plugin developers should not rely on changing axis, title or legends as part of their controls, since these features can easily be lost when the plot has been saved and loaded anew.

2. The frame instead uses the first model. This is the default solution for embedded plot views where frames are added using the GridLayout.addModel() method. It ensures maximum control for embedded PlotViews, but you should not rely too heavily on the freedom it allows, as all the saved plots use a private view model (see 1.).

Parameters
use_frame_model (*bool*) – True, if the frame should use a separate view model..

addModel(*model*, *flipped=False*)

Add a model to the plot frame.

Note that the frame will explicitly *not* listen to model changes.

Syncronization between frame and model must happen in the view implementation.

Parameters

- **model** (PlotModel) – The model to add.
- **flipped** (*bool*) – True, if the model axes are flipped.

autoscaleLimits()

Update the autoscaled axis limits.

axis(*name*)

The axis with a given name.

Parameters
name (*str*) – The name (x or y).

Returns
The axis.

Return type
class:*~.Plot.Axis*

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

flip()

Flip the x- and y-axis.

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

isFlipped()

Returns
The frame model.

Return type
Plot Model

isModelFlipped(*model*)

Check if a given model is flipped in this frame.

Parameters
model (PlotModel) – The model to check.

Returns
True, if the model is flipped.

Return type
bool

model()

Returns
The frame model.

Return type
PlotModel

models(*plotter=None*)

Return all models, perhaps filtered by a plotter.

Parameters
plotter (*None | Plotter based*) – The plotter to filter on.

Returns
The added models.

Return type
list of class:~.*Plot.PlotModel*

plotters()

Returns
The plotter referenced in the frame.

Return type
list of Plotter based

projection()

Returns
The projection.

Return type
str | NoneType

setFlipped(*flipped*)

Set flipped property.

Parameters
flipped (*bool*) – True, if the frame is flipped.

setProjection(*projection*)

Set the projection.

Parameters
projection (*str*) – The projection.

usesFrameModel()

. :returns: True, if the frame has a separate frame model. :rtype: bool

Usage Examples

A PlotFrame makes it possible to embed a *Plot.PlotModel* into a layout – just like a beautiful painting must be framed before it can be hung on the wall togehter with other paintings.

However, the PlotFrame is much more versatile than a standard picture frame. The PlotFrame can contain multiple PlotModels:

```
>>> frame = Plot.PlotFrame()
>>> frame.addModel(Plot.PlotModel())
```

And the frame can easily exchange the x- and y-axises:

```
>>> frame.flip()
```

Note: The PlotFrame must contain contain at least one PlotModel before it can be embedded in e.g. a *Plot.GridLayout* or a *Plot.OverlayLayout*.

Multiple PlotModels can also be embedded in a common PlotFrame, and complex plots can be built by combining multiple PlotFrames in a layout like the *Plot.GridLayout* or the *Plot.OverlayLayout*. A more complex usage example is given below.

```
# Build one plot model.
frame1.addModel(model1)
frame1.addModel(model2, flipped=True)

# Add the second model to its own frame.
frame2 = Plot.PlotFrame()
frame2.addmodel(model1)

# Add frames to grid layout.
layout = Plot.GridLayout(2, 1)
layout.addItem(frame1)
layout.addItem(frame2)

# Show the plot.
Plot.show(layout)
```

Plot.PlotModel

class PlotModel(*x_unit=<Unit at 140734618035408 >*, *y_unit=<Unit at 140734618035408 >*, *projection='rectilinear'*)

A plot model contains plot items.

The items contain data objects like lines, bars etc. and can be named for easy retrieval.:

```
>>> model = PlotModel(x_unit, y_unit)
>>> model.setItem('line', Line([0, 1], [1, 2]))

>>> line = model.item('line').
```

The plot model also contains the special items: xAxis, yAxis, legend and title. These items are special in the sense that they are only applied when the frame containing the model is reset. So for most purposes they only serve as initial values.

Parameters

- **x_unit** (Unit) – The unit of the x axis of the plot model data.
- **y_unit** (Unit) – The unit of the y axis of the plot model data.
- **projection** (*str*) – The projection.

addItem(*item*)

Add a plot item to the model.

Parameters
item (*PlotItem based*) – The item to add.

addItems(*items*)

Add multiple items to the model.

Parameters
items (*list of PlotItem*) – The items.

axis(*name*)

Parameters
name (*str*) – The name of the axis to get, x or y.

Returns
The x- or y-axis for the sub-plot.

Return type
Axis

clearItems(*plot_item_type=None*)

Clear all items from the model.

Parameters
plot_item_type (*tuple of class*) – Optional tuple of item types to clear.

copy(*items=True*)

Create a copy of the model (sometimes without items).

Parameters
items (*bool*) – True, if items should be copied.

Returns
The copied model.

Return type
PlotModel

deleteKey(*key*, *emit=True*)

Delete the item stored at specific key.

Parameters

- **key** (*str*) – The key to delete.
- **emit** (*bool*) – True, if the change should be emitted.

findItems(*types=None*, *function=None*, ***kwargs*)

Find all (possibly nested) items by querying the item properties. Mutliple arguments can be supplied and applies in an exclusive way,:

```
model.findItems(
    types=Line,
    function=lambda item: 'line' in item.label(),
    line_width=2
)
```

It is always faster and safer to find the item directly by traversing the object hierarchy, e.g.:

```
item = model.item('my_key')
```

or for nested items:

```
item = model.item('my_key').items()[1]
```

If searching for top level items only, the items method can be very useful.

Parameters

- **types** (*NoneType | PlotItem class*) – An optional type filter specifying the plot items to find e.g. Line or Scatter.
- **function** (*NoneType | function*) – An optional filter function taking the plot item as its only input.
- ****kwargs** (*dict*) – Optional query properties.

Returns
The items matching the query.

Return type
dict

framing()

Returns
The model framing.

Return type
Framing

classmethod fromProperties(*properties*)

Convert properties into plot model.

Parameters
properties (*dict*) – The properties.

Returns
The created PlotModel.

Return type
PlotModel

item(*key*)

Get the item stored on a specific key in the model. Simply returns None if if no item exist at the given key.

Parameters
key (*str*) – The key to get.

Returns
The item

Return type
object

`itemKey`(*item*)

Get the key for the given item.

Parameters
`item` (*PlotItem*) – The given item.

Returns
The key.

Return type
str

`itemKeys`()

All currently set item keys.

Returns
All item keys.

Return type
iterable of str

`items`(*tag=None*, *plot_item_type=None*)

Get plot items of specific type.

Parameters

- `tag` (*str*) – Filter items on an optional tag.
- `plot_item_type` (*class | tuple of class*) – Filter items on an optional item type.

Returns
Returns the items of either plot item type.

Return type
view | list

`legend`()

Returns
The model legend.

Return type
Legend

`limits`(*name*)

Get limits for one axis.

Parameters
`name` (*str*) – The axis name.

Returns
Limits.

Return type
tuple of float

`plotter`()

Returns
The plotter this model belongs to.

Return type
Plotter | None

projection()

Returns
The model projection (rectilinear or polar).

Return type
str

removeItem(*item*)

Remove an item from the model.

Parameters
item (*PlotItem based*) – The item to remove.

selectedItems()

Returns
The selected plot items.

Return type
list of PlotItems

setItem(*key*, *item*, *overwrite=True*, *emit=True*)

Add a plot item to the model and assign it to a specific.

Parameters

- **key** (*str*) – The key to set.
- **item** (*PlotItem based*) – The item to add.
- **overwrite** (*bool*) – True, if any existing item at that key should be silently overwritten.
- **emit** (*bool*) – True, if the addition should be emitted.

setLimits(*name=None*, *lower_limit=None*, *upper_limit=None*, *item_filter=None*)

Set model data limits. Passing no limits will autoscale the current plot limits.

Parameters

- **name** (*str*) – The name of the axis (x or y).
- **lower_limit** (*None | float*) – The lower limit.
- **upper_limit** (*None | float*) – The upper limit.
- **item_filter** (*function*) – An additional item filter used when automatically setting limits.

setProjection(*projection*)

Set the model projection.

Parameters
projection (*str*) – The projection (rectilinear or polar).

title()

Returns
The model framing, which contains the title.

Return type
Framing

static transferUniformProperties(*templates*, *items*, *exclude_data=True*)

Determine which properties that are the same for all of the provided templates, and transfer those properties to the provided items.

Parameters

- **templates** (*list of PlotItem*) – The list of plot items to use as templates.
- **items** (*list of PlotItem*) – The items to transfer properties to.
- **exclude_data** (*bool*) – True, if we do not wish to transfer the item data.

Returns
True, if the transfer succeeded.

Return type
bool

xAxis()

Returns
The x-axis for the-plot.

Return type
Axis

yAxis()

Returns
The y-axis for the-plot.

Return type
Axis

Usage Examples

The PlotModel is a container for PlotItems like `Line`, `Scatter`, `Label` etc. The most basic PlotModel is constructed without units on the axises:

```
>>> model = Plot.PlotModel()
>>> model.addItem(Plot.Line([0, 1, 2, 3, 4], [1, 2, 3, 4, 5]))
```

Set up a PlotModel with Angstrom on the x-axis and eV on the y-axis.

```
>>> model = PlotModel(x_unit=Angstrom, y_unit=eV)
>>> model.addItem(Line([0, 1, 2, 3, 4] * nm, [1, 2, 3, 4, 5] * eV))
```

PlotModels can be shown and saved using the *Plot.show* and *Plot.save* functions.

```
>>> Plot.show(model)
>>> Plot.save(model, 'export.hdf5')
```

Notes

Multiple PlotModels can also be embedded in a common *Plot.PlotFrame*, and complex plots can be built by combining multiple PlotFrames in a layout like the *Plot.GridLayout* or the *Plot.OverlayLayout*.

Plot.PolynomialFit

class PolynomialFit(*order=1*)

A PolynomialFit can be added to a `Line` or `Scatter`. A PolynomialFit then fits a polynomial of the given order to the parent data. The fit will update every time the parent data changes.

Parameters
order (*int*) – The polynomial order.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

attributes(*latex=False*, *detail_level='full'*)

A description of the fit properties.

Parameters

- **latex** (*bool*) – If True, return the attributes in Latex format.
- **detail_level** (*DETAIL_LEVELS enum*) – The level of detail of the attributes.

Returns
A description of the fit properties.

Return type
dict

bounds()

Returns
The bounds.

Return type
tuple of physicalQuantity

coefficients()

Returns
The polynomal coefficients.

Return type
numpy.ndarray

color()

Returns
The line color.

Return type
str

copy()
Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

determination()

Returns
The fit coefficient of determination.

Return type
float | NoneType

emit(*key*, *value=None*)
Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

evaluate(*x_data*)
Evaluate the polynomial between minimum and maximum bounds.

Parameters
x_data (*PhysicalQuantity*) – The x-data.

Returns
The evaluated x data and y data.

Return type
2-tuple of PhysicalQuantity

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isAxisSpanned()

Returns
True if the fit line should span the x-axis range of the plot.

Return type
bool

isBoundsVisible()

Returns
True, if fit bounds are shown in the plot.

Return type
bool

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isInteractive()

Returns
Whether the user can interact with this line.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

classmethod isValidParent(*item*)

Check if an item is derivable.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The line label.

Return type
str

lineStyle()

Returns
The line style.

Return type
str | None

lineWidth()

Returns
The line width.

Return type
float

markerEdgeColor()

Returns
The edge color of the marker in hex format.

Return type
str

markerEvery()

Returns
The number of points between every marker.

Return type
int

markerFaceColor()

Returns
The face color of the marker in hex format.

Return type
str

markerSize()

Returns
The marker size.

Return type
float

markerStyle()

. :returns: The marker style. :rtype: str | None

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
The order of the fitted polynomial.

Return type
int

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setBounds(*lower*, *upper*, *update=True*)

Set the derived line bounds.

Parameters

- **lower** (*float | PhysicalQuantity*) – The lower bound.
- **upper** (*float | PhysicalQuantity*) – The upper bound.
- **update** (*bool*) – True, if we should call update.

setBoundsVisible(*visible*)

Set whether to indicate the fit bounds in the plot.

Parameters
visible (*bool*) – If True, fit bounds will be shown in the plot.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x=None*, *y=None*)

Overload since derived lines should not expose data setting.

Parameters

- **x** (*iterable*) – The x values.
- **y** (*iterable*) – They values.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setInteractive(*interactive*)

Set whether the user can interact with this line.

Parameters
interactive (*bool*) – The desired state.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOrder(*order*, *update=True*)

Set the order of the fitted polynomial.

Parameters

- **order** (*int*) – The polynomial order.
- **update** (*bool*) – True if the fit should be updated.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setSpanAxis(*span_axis*)

Set whether the fit line should span the x-axis range of the plot.

Parameters
span_axis (*bool*) – If True, the fit line should span the x-axis range of the plot.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

sourceData()

Checks if the source data is valid, and if yes, return the valid source data on the form: (xs, ys, x_min, x_max, x_unit, y_unit) Else returns None

Returns
The validated and clipped source data.

Return type
tuple | None

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

update(*data=None*)

Perform the update.

Parameters
data (*tuple | NoneType*) – The source data to use.

validSourceCoordinates(*xs*, *ys*)

True if the data coordinates are valid.

Parameters

- **xs** (ndarray) – The source x-coordinates.
- **ys** (ndarray) – The source y-coordinates.

Returns
True, if the coordinates are valid.

Return type
bool

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.RollingAverage

class RollingAverage(*window=5*)

A RollingAverage can be added to a Line or Scatter, and will then calculate the rolling average using the supplied window size.

Parameters
window (*int*) – The number of close points to average over.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

attributes(*latex=False*, *detail_level=None*)

A description of the fit properties.

Parameters

- **latex** (*bool*) – If True, return the attributes in Latex format.
- **detail_level** (*DETAIL_LEVELS enum*) – The level of detail of the attributes (unused).

Returns
A description of the fit properties.

Return type
dict

boundaryCondition()

Returns
The current boundary condition.

Return type
BOUNDARY_CONDICTIONS enum

bounds()

Returns
The bounds.

Return type
tuple of physicalQuantity

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isInteractive`()

Returns
Whether the user can interact with this line.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`classmethod isValidParent`(*item*)

Check if an item is derivable.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items

•None : All items.

Parameters

•**selected** (*None | bool*) – True, if only selected items should be returned.

•**tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

kernel()

Returns
The current averaging kernelS.

Return type
KERNELS enum

label()

Returns
The line label.

Return type
str

lineStyle()

Returns
The line style.

Return type
str | None

lineWidth()

Returns
The line width.

Return type
float

markerEdgeColor()

Returns
The edge color of the marker in hex format.

Return type
str

markerEvery()

Returns
The number of points between every marker.

Return type
int

markerFaceColor()

Returns
The face color of the marker in hex format.

Return type
str

markerSize()

Returns
The marker size.

Return type
float

markerStyle()

. :returns: The marker style. :rtype: str | None

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setBoundaryCondition(*boundary_condition*, *update=True*)

Set the boundary condition.

Parameters

- **boundary_condition** (*BOUNDARY_CONDICTIONS enum*) – The boundary condition
- **update** (*bool*) – True, if the average should be updated.

setBounds(*lower*, *upper*, *update=True*)

Set the derived line bounds.

Parameters

- **lower** (*float | PhysicalQuantity*) – The lower bound.
- **upper** (*float | PhysicalQuantity*) – The upper bound.
- **update** (*bool*) – True, if we should call update.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x=None*, *y=None*)

Overload since derived lines should not expose data setting.

Parameters

- **x** (*iterable*) – The x values.
- **y** (*iterable*) – They values.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.

- **details** (*dict | NoneType*) – The hover details.

setInteractive(*interactive*)

Set whether the user can interact with this line.

Parameters
interactive (*bool*) – The desired state.

setKernel(*kernel*, *update=True*)

Set the rolling average kernel.

The KERNELS.AVERAGE kernel is simply flat across the window length. A KERNELS.GAUSSIAN kernel is a gaussian with a variance given by the window length.

Parameters

- **kernel** (*KERNELS enum*) – The kernel to use.
- **update** (*bool*) – True, if the average should be updated.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setWindow(*window*, *update=True*)

Set the window size for the rolling average.

Parameters

- **window** (*int*) – The window size to set.
- **update** (*bool*) – True, if the fit should be updated.

sourceData()

Checks if the source data is valid, and if yes, return the valid source data on the form: (xs, ys, x_min, x_max, x_unit, y_unit) Else returns None

Returns
The validated and clipped source data.

Return type
tuple | None

style()

Returns
All the current style properties.

Return type
dict

`styleProperties`(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

`tags`()

Returns
The set of tags.

Return type
set

`update`(*data=None*)

Perform the update.

Parameters
data (*tuple | NoneType*) – The source data to use.

`validSourceCoordinates`(*xs*, *ys*)

True if the data coordinates are valid.

Parameters

- **xs** (*numpy.ndarray*) – The source x coordinates.
- **ys** (*numpy.ndarray*) – The source y coordinates.

Returns
True, if the coordinates are valid.

Return type
bool

`window`()

Returns
The window size to use for the rolling average.

Return type
int

`xData`()

Returns
The x-data.

Return type
PhysicalQuantity

`yData`()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.Scatter

class Scatter(*x=None*, *y=None*, *values=None*)

A Scatter displays x- and y-data as a scatter plot.

Parameters

- **x** (Sequence | *PhysicalQuantity* | None) – The x-values of the data.
- **y** (Sequence | *PhysicalQuantity* | None) – The y-values of the data.
- **values** (*Sequence | None*) – The values of the data. Used for coloring or resizing the data points.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the points.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The face color in hex format.

Return type
str

colorMap()

. :returns: The color map. :rtype: str

colorMode()

Returns
The color mode.

Return type
COLOR_MODES enum

colors()

Returns
The colors.

Return type
None | list of str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data extent.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The current scatter label.

Return type
str

`model()`

Returns
The model that this item belongs to.

Return type
PlotModel | None

`order()`

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

`outline()`

Returns
True, if the scatter circles should be outlined.

Return type
bool

`owner()`

Returns
The owner type.

Return type
int | enum

`parent()`

Returns
The parent.

Return type
PlotItem based | None

`removeItem`(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

scaling()

Returns
The scaling parameters.

Return type
tuple

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Parameters
alpha (*float*) – The opacity of the points.

setColor(*color*)

Set the uniform color.

Note that this color is only used when color mode is set to COLOR.

Parameters
color (*str*) – The face color.

setColorMap(*color_map*)

The color map.

Note that the color map is only used when the color mode is set to COLOR_MAP.

Parameters
color_map (*None | str*) – The color map to use.

setColorMode(*mode*)

Set the color mode. One of either COLOR_MODES.COLOR, COLOR_MODES.COLORS, COLOR_MODES.COLOR_MAP.

Parameters
mode (*COLOR_MODES enum*) – The mode to set.

setColors(*colors*)

Set the colors.

Note that the colors are only used when the color mode is set to COLORS

Parameters
colors (*None | list of str*) – The colors in hex format.

setData(*x=None*, *y=None*, *tooltips=None*)

Set the scatter data.

Parameters

- **x** (list | ndarray | *PhysicalQuantity*) – The x-data.
- **y** (list | ndarray | *PhysicalQuantity*) – The y-data.
- **tooltips** (*list | str | NoneType*) – The corresponding tool tips.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the scatter label.

Parameters
label (*str*) – The label of the scatter.

setOutline(*outline*)

Set the outline property.

Parameters
outline (*bool*) – True, if the scatter circles should be outlined.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setScaling(*lower*, *upper*, *use_absolute_values=False*)

Scale points based on their values. Requires *values* to be specified to have any effect.

Parameters

- **lower** (*float*) – The size of the smallest value.
- **upper** (*float*) – The size of the largest value.
- **use_absolute_values** (*bool*) – Use absolute values when scaling the points.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setValues(*values*)

Set the weights of the data.

Parameters
values (*Sequence | None*) – The values of the data. Used for coloring or resizing the data points.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
The current scatter style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

tooltips()

Returns
The tool tips.

Return type
list of str | str | NoneType

values()

Returns
The weights of the data. Used for coloring or setting line widths.

Return type
Sequence | None

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.TransformedLine

class TransformedLine(*x_expression='x'*, *y_expression='y'*)

A TransformedLine can be added to a Line or Scatter, and will then calculate a transformed line using the supplied x- and y-expression.

Parameters

- **x_expression** (*str*) – The expression to apply to the source x-data.
- **y_expression** (*str*) – The expression to apply to the source y-data.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

attributes(*latex=False*, *detail_level='full'*)

The attributes of the Transformed line.

Parameters

- **latex** (*bool*) – If True, return the attributes in Latex format.
- **detail_level** (*DETAIL_LEVELS enum*) – The level of detail of the attributes (unused).

Returns
Either an OK message, or an error message if any of the expressions did not evaluate properly.

Return type
collections.OrderedDict

bounds()

Returns
The bounds.

Return type
tuple of physicalQuantity

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data limits.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | PhysicalQuantity

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isInteractive()

Returns
Whether the user can interact with this line.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

classmethod isValidParent(*item*)

Check if an item is derivable.

Parameters
item (*PlotItem based*) – The item to check.

Returns
True, if the item can be derived.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **`selected`** (*None | bool*) – True, if only selected items should be returned.
- **`tag`** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label`()

Returns
The line label.

Return type
str

`lineStyle`()

Returns
The line style.

Return type
str | None

`lineWidth`()

Returns
The line width.

Return type
float

markerEdgeColor()

Returns
The edge color of the marker in hex format.

Return type
str

markerEvery()

Returns
The number of points between every marker.

Return type
int

markerFaceColor()

Returns
The face color of the marker in hex format.

Return type
str

markerSize()

Returns
The marker size.

Return type
float

markerStyle()

. :returns: The marker style. :rtype: str | None

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

safeSetXExpression(*expression*, *update=True*)

Safe setting of expression. Fails silently without changing anything.

Parameters

- **expression** (*str*) – The expression.
- **update** (*bool*) – True if the fit should be updated.

safeSetYExpression(*expression*, *update=True*)

Safe setting of expression. Fails silently without changing anything.

Parameters

- **expression** (*str*) – The expression.
- **update** (*bool*) – True if the fit should be updated.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setBounds(*lower*, *upper*, *update=True*)

Set the derived line bounds.

Parameters

- **lower** (*float | PhysicalQuantity*) – The lower bound.
- **upper** (*float | PhysicalQuantity*) – The upper bound.
- **update** (*bool*) – True, if we should call update.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x=None*, *y=None*)
Overload since derived lines should not expose data setting.

Parameters

- **x** (*iterable*) – The x values.
- **y** (*iterable*) – They values.

setHoverable(*hoverable*)
Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)
Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setInteractive(*interactive*)
Set whether the user can interact with this line.

Parameters
interactive (*bool*) – The desired state.

setLabel(*label*)
Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)
Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)
Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)
Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)
Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setXExpression(*expression*, *update=True*)

Set the expression for x-data.

Parameters

- **expression** (*str*) – The x-expression to set.
- **update** (*bool*) – True, if the fit should be updated.

setYExpression(*expression*, *update=True*)

Set the expression for y-data.

Parameters

- **expression** (*str*) – The y-expression to set.
- **update** (*bool*) – True, if the fit should be updated.

sourceData()

Checks if the source data is valid, and if yes, return the valid source data on the form: (xs, ys, x_min, x_max, x_unit, y_unit) Else returns None

Returns
The validated and clipped source data.

Return type
tuple | None

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

tooltip()

Returns
The tool tip.

Return type
str

update(*data=None*)

Perform the update.

Parameters
data (*tuple | NoneType*) – The source data to use.

validSourceCoordinates(*xs*, *ys*)

Re-implement for custom validation of the source. See e.g. PolynomialFit.

Parameters

- **xs** (*PhysicalQuantity*) – The source x data.
- **ys** (*PhysicalQuantity*) – The source y data.

Returns
True, if valid.

Return type
bool

xData()

Returns
The x-data.

Return type
PhysicalQuantity

xExpression()

Return the expression for x-data.

Returns
The expression.

Return type
str

yData()

Returns
The y-data.

Return type
PhysicalQuantity

yExpression()

Return the expression for y-data.

Returns
The expression.

Return type
str

Plot.VerticalBar

class VerticalBar(*bins*, *data*)

A bar plot showing data as vertical filled bins.

The shape of the bins is determined in the following way: - If the number of bins equals the number of data values, the bins parameter is interpreted as the bin centers and bin widths are uniformly set to the match minimum distance between the bin centers. - If the number of bins is one larger than the number of data values, the bins parameter is interpreted as the edges of the bins. - If the number of bins is double the number of data values, the bins are instead interpreted as the left and right edge of each of the bins.

Parameters

- **bins** (Sequence of *PhysicalQuantity*) – The bins.
- **data** (Sequence of *PhysicalQuantity*) – The data values.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the bar plot.

Return type
int

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

binCenters()

Returns
The center of each bin.

Return type
Sequence of *PhysicalQuantity*

binEdges()

. :returns: The left edges of each bin. :rtype: *PhysicalQuantity*

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColor()

Returns
The edge color of the bar plot.

Return type
str

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

The limits around the bar plot.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Optional padding.

Returns
The min/max data values for the requested axis.

Return type
2-tuple of *PhysicalQuantity*

faceColor()

Returns
The face color of the bar plot.

Return type
str

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

histogram()

Returns
The bin counts for each bin.

Return type
Sequence of *PhysicalQuantity*

hoverDetails()

Returns
The current hover details.

Return type
dict | None

`isHoverable`()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)
Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **`selected`** (*None | bool*) – True, if only selected items should be returned.
- **`tag`** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The label.

Return type
str

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)
Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)
Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

. :param alpha: The opacity of the bar plot. :type alpha: float

setColor(*color*)

Set the same color for both the bar face and the bar edge.

Parameters
color (*str*) – The color to set in hex format.

setData(*bins*, *data*, *emit=True*)

Set bins and histogram data.

The shape of the bins is determined in the following way: - If the number of bins equals the number of data values, the bins parameter is interpreted as the bin centers and bin widths are uniformly set to the match minimum distance between the bin centers. - If the number of bins is one larger than the number of data values, the bins parameter is interpreted as the edges of the bins. - If the number of bins is double the number of data values, the bins are instead interpreted as the left and right edge of each of the bins.

Parameters

- **bins** (Sequence of *PhysicalQuantity*) – The bins.
- **data** (Sequence of *PhysicalQuantity*) – The data values.
- **emit** (*bool*) – True, if the data should be emitted.

setEdgeColor(*color*)

Set the bar plot edge color.

Parameters
color (*str*) – The edge color of the bar plot.

setFaceColor(*color*)

. :param color: The face color of the bar plot. :type color: str

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the bar plot.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

xData()

Returns
The center of each bin.

Return type
Sequence of *PhysicalQuantity*

yData()

Returns
The bin counts for each bin.

Return type
Sequence of *PhysicalQuantity*

Plot.VerticalFill

class VerticalFill(*y*, *x_lower*, *x_upper*)
A VerticalFill display a filled area between two lines.

Parameters

- **y** (Sequence of *PhysicalQuantity*) – The y-values of the data.
- **x_lower** (SSequence of *PhysicalQuantity*) – The lower x-values in the fill.
- **x_upper** (Sequence of *PhysicalQuantity*) – The upper x-values in the fill.

addItem(*item*, *emit=True*)
Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)
Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)
Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the fill.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)
Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

copy()
Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColor()

Returns
The edge color of the fill in hex format.

Return type
str

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis*, *padding=0.0*)

The data extent.

Parameters

- **axis** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
2-tuple *PhysicalQuantity*

faceColor()

Returns
The edge color of the fill in hex format.

Return type
str

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

`isHovered`()

Returns
True, if the item is hovered.

Return type
bool

`isSelectable`()

Returns
True, if the item is selectable.

Return type
bool

`isSelected`()

Returns
True, if the item is selected.

Return type
bool

`isVisible`()

Returns
True, if the item is visible.

Return type
bool

`items`(*selected=None*, *tag=None*)
Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label`()

Returns
The current label.

Return type
str

lower()

Returns
The lower values in the fill.

Return type
float

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)
Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)
Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity.

Parameters
alpha (*float*) – The opacity of the fill.

setColor(*color*)

Set the same color for both the face and the edge.

Parameters
color (*str*) – Set the color of the fill in hex format.

setEdgeColor(*color*)

Set the edge color of the fill.

Parameters
color (*str*) – The edge color of the fill in hex-format.

setFaceColor(*color*)

Set the face color of the fill.

Parameters
color (*str*) – The face color of the fill.in hex format.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- hovered (*bool*) – True, if the item was hovered.
- details (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the line.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

upper()

Returns
The upper values in the fill.

Return type
float

values()

Returns
The values in the fill.

Return type
float

Plot.VerticalLine

class VerticalLine(*x*)

A VerticalLine shows a straight vertical line.

Parameters
x (*PhysicalQuantity*) – The x-value of the line.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity for the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The line color.

Return type
str

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isSelectable()`

Returns
True, if the item is selectable.

Return type
bool

`isSelected()`

Returns
True, if the item is selected.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

`label()`

Returns
The line label.

Return type
str

`lineStyle()`

Returns
The line style.

Return type
str | None

`lineWidth()`

Returns
The line width.

Return type
float

`markerEdgeColor()`

Returns
The edge color of the marker in hex format.

Return type
str

`markerEvery()`

Returns
The number of points between every marker.

Return type
int

`markerFaceColor()`

Returns
The face color of the marker in hex format.

Return type
str

`markerSize()`

Returns
The marker size.

Return type
float

markerStyle()

. :returns: The marker style. :rtype: str | None

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

Set the opacity for the line.

Parameters
alpha (*float*) – The opacity.

setColor(*color*)

Set the line color.

Parameters
color (*str*) – The line color in hex format.

setData(*x*)

Set x data.

Parameters
x (*PhysicalQuantity*) – The x-data.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The line label.

setLineStyle(*line_style*)

Set the line style.

Parameters
line_style (*str | None*) – The line style.

setLineWidth(*line_width*)

Set the line width.

Parameters
line_width (*float*) – The line width.

setMarkerColor(*color*)

Set color of the marker. Remember to also set a valid marker style.

Parameters
color (*valid color*) – The color to set.

setMarkerEdgeColor(*color*)

Set the marker edge color. Remember to also set a valid marker style.

Parameters
color (*str*) – The edge color of the marker.

setMarkerEvery(*every*)

Set the number of points between each marker. Remember to also set a valid marker style.

Parameters
every (*int*) – The number of points between every marker.

setMarkerFaceColor(*color*)

Set the marker face-color. Remember to also set a valid marker style.

Parameters
color (*str*) – The face color of the marker.

setMarkerSize(*size*)

Set the marker size. Remember to also set a valid marker style.

Parameters
size (*float*) – The marker size.

setMarkerStyle(*marker*)

Set the marker style.

Parameters
marker (*str | None*) – The marker style.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

style()

Returns
All the current style properties.

Return type
dict

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Plot.VerticalStackedFill

class VerticalStackedFill(*x_values*, *values*)

A VerticalStackedFill shows multiple fills laid on top of each other.

Parameters

- **x_values** (Sequence of *PhysicalQuantity*) – The x-values of the data.
- **values** (2D *PhysicalQuantity*) – The values to plot in the stack.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alphas()

Returns
The opacities of the fills.

Return type
list

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

edgeColors()

Returns
The edge colors of the fills.

Return type
list

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

faceColors()

Returns
The face colors of the fills.

Return type
list

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

`hoverDetails()`

Returns
The current hover details.

Return type
dict | None

`isHoverable()`

Returns
True, if hoverable.

Return type
bool

`isHovered()`

Returns
True, if the item is hovered.

Return type
bool

`isSelectable()`

Returns
True, if the item is selectable.

Return type
bool

`isSelected()`

Returns
True, if the item is selected.

Return type
bool

`isVisible()`

Returns
True, if the item is visible.

Return type
bool

`items(`*selected=None*, *tag=None*`)`

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

labels()

Returns
The labels of the fills.

Return type
list

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlphas(*alphas*)

Set the alphas for all fills in the stack.

Parameters
alphas (*list*) – The opacities of the fills.

setColors(*colors*)

Set the same color for both the face and the edge for all filss in the stack.

Parameters
colors (*list*) – The colors of the fills.

setEdgeColors(*colors*)

Set the edge colors for all fills in the stack.

Parameters
colors (*list*) – The edge colors of the fills.

setFaceColors(*colors*)

Set the face colors for all fills in the stack.

Parameters
colors (*list*) – The face colors of the fills.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabels(*labels*)

Set the labels for all fills in the stack.

Parameters
labels (*list*) – The labels of the fills.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

`setSelectable`(*selectable*)

Set whether this item is selectable.

Parameters
`selectable` (*bool*) – True, if the item should be selectable.

`setSelected`(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- `selected` (*bool*) – True, if item should be selected.
- `details` (*dict | NoneType*) – The selection details.

`setVisible`(*visible*)

Set visible.

Parameters
`visible` (*bool*) – True, if the item should be visible.

`styleProperties`(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
`style_scheme` (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

`tags`()

Returns
The set of tags.

Return type
set

`values`()

Returns
The values of the data.

Return type
2D *PhysicalQuantity*

Plot.WeightedLine

`class WeightedLine`(*x=None*, *y=None*, *weights=None*)

A WeightedLine plots a line where the line width changes along the line.

Parameters

- `x` (Sequence | *PhysicalQuantity* | None) – The x-values of the data.
- `y` (Sequence | *PhysicalQuantity* | None) – The y-values of the data.
- `weights` (*Sequence | None*) – The weights of the data. Used for coloring or setting line widths.

addItem(*item*, *emit=True*)

Add item as child of this item.

Parameters

- **item** (*PlotItem based*) – The item.
- **emit** (*bool*) – True, if change should be emitted.

addSubscriber(*subscriber*)

Add a subscriber. Each subscriber is called once every time the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

addTag(**args*)

Add tag. Used for grouping plot items.

Parameters
***args** (*tuple of str*) – The tags to add.

alpha()

Returns
The opacity of the line.

Return type
float

applyStyle(*style_scheme=None*, *style_properties=None*)

Apply style properties from a style scheme.

Parameters

- **style_scheme** (*dict*) – The style scheme.
- **style_properties** (*dict | None*) – A specific set of style properties to apply. If None, the style properties are retrieved from the plot model using the given style scheme.

color()

Returns
The line color in hex format

Return type
str

colorMap()

Returns
The current color map.

Return type
str

colorMode()

The current color mode.

Returns
The color mode.

Return type
COLOR_MODES enum

copy()

Make a copy of the plot storable by using the serialize/deserialize functionality.

This creates a new instance with a new id, but all other properties copied.

emit(*key*, *value=None*)

Emit a single key and value change to all subscribers.

Parameters

- **key** (*str*) – The key to emit.
- **value** (*object*) – The value to emit.

extent(*axis_name*, *padding=0.0*)

Get the data extent.

Parameters

- **axis_name** (*str*) – x or y.
- **padding** (*float*) – Padding around the data. **Default:** 0.0.

Returns
The min/max data values for the requested axis.

Return type
Sequence | *PhysicalQuantity*

classmethod fromProperties(*properties*)

Create a plot storable from a dict of properties.

Used when deserializing a PlotStorable.

Parameters
properties (*dict*) – The properties which are used to populate the properties dict of the newly created plot storable.

Returns
The newly created PlotStorable.

Return type
PlotStorable

hoverDetails()

Returns
The current hover details.

Return type
dict | None

isHoverable()

Returns
True, if hoverable.

Return type
bool

isHovered()

Returns
True, if the item is hovered.

Return type
bool

isSelectable()

Returns
True, if the item is selectable.

Return type
bool

isSelected()

Returns
True, if the item is selected.

Return type
bool

isVisible()

Returns
True, if the item is visible.

Return type
bool

items(*selected=None*, *tag=None*)

Return the requested items.

The optional selected key word filter selected and deselected lines, with the options:

- True : Only selected items
- False : Only unselected items
- None : All items.

Parameters

- **selected** (*None | bool*) – True, if only selected items should be returned.
- **tag** (*str*) – The tag to filter on.

Returns
The child items with the selected filtering.

Return type
items

label()

Returns
The current label.

Return type
str

model()

Returns
The model that this item belongs to.

Return type
PlotModel | None

order()

Returns
A number describing the current order of the item within the PlotModel. Can be used for sorting.

Return type
int | None

owner()

Returns
The owner type.

Return type
int | enum

parent()

Returns
The parent.

Return type
PlotItem based | None

removeItem(*item*, *emit=True*)

Remove item from the child items.

Parameters

- **item** (*PlotItem based*) – The item to remove.
- **emit** (*bool*) – True, if removal should be emitted.

removeSubscriber(*subscriber*)

Remove a subscriber method from the plot item updates.

Parameters
subscriber (*bound method*) – The subscriber. Must be callable.

scaling()

Returns
The scaling of the weights.

Return type
float

selectionDetails()

Returns
The current selection details.

Return type
dict | None

setAlpha(*alpha*)

. :param alpha: The opacity of the line. :type alpha: float

setColor(*color*)

Set the line color.

Note that the color map is only used when the color mode is COLOR.

Parameters
color (*str*) – The color in hex format.

setColorMap(*color_map*)

Set a color map.

Note that the color map is only used when the color mode is COLOR_MAP.

Parameters
color_map (*str*) – A color map.

setColorMode(*mode*)

Set the color mode (whether to use color map or color for drawing).

Parameters
mode (*COLOR_MODES enum*) – The mode.

setData(*x=None*, *y=None*)

Set data.

Parameters

- **x** (list | ndarray | *PhysicalQuantity*) – The x-data.
- **y** (list | ndarray | *PhysicalQuantity*) – The y-data.

setHoverable(*hoverable*)

Enable or disable hover evetn. When enabled the line will be set to hovered, once the mouse hovers over the item.

Parameters
hoverable (*bool*) – True, if the item should be sensitive to hover events.

setHovered(*hovered*, *details=None*)

Set hovered with some details. This is called once the mouse hovers over the item. Should only be called once hoverable has been enabled.

Parameters

- **hovered** (*bool*) – True, if the item was hovered.
- **details** (*dict | NoneType*) – The hover details.

setLabel(*label*)

Set the label.

Parameters
label (*str*) – The label of the line.

setOwner(*owner*)

Set the owner of the item (either OWNERS.MODEL or OWNERS.VIEW).

Parameters
owner (*int | enum*) – The owner type.

setScaling(*scaling*)

Set the scaling.

Parameters
scaling (*float*) – The scaling of the weights. Requires *weights* to be specified to have any effect.

setSelectable(*selectable*)

Set whether this item is selectable.

Parameters
selectable (*bool*) – True, if the item should be selectable.

setSelected(*selected*, *details=None*)

Set the item as selected or unselected.

Parameters

- **selected** (*bool*) – True, if item should be selected.
- **details** (*dict | NoneType*) – The selection details.

setVisible(*visible*)

Set visible.

Parameters
visible (*bool*) – True, if the item should be visible.

setWeights(*weights*)

Set the weights of the data.

Parameters
weights (*Sequence | None*) – The weights of the data. Used for coloring and setting line widths.

styleProperties(*style_scheme*)

Check if the given style scheme has a style for this item.

Parameters
style_scheme (*dict*) – The style scheme.

Returns
The style, if any.

Return type
dict | None

tags()

Returns
The set of tags.

Return type
set

weights()

Returns
The weights of the data. Used for coloring and setting line widths.

Return type
Sequence | None

xData()

Returns
The x-data.

Return type
PhysicalQuantity

yData()

Returns
The y-data.

Return type
PhysicalQuantity

Functions

Plot.save

save(*plot*, *filename*, *size=None*, ***kwargs*)

Save a plot to a file.

Save can save the full, interactive plot. This is done by supplying a filename with the HDF5 file extension. The plot can then later be loaded by using the standard `nlread` function.

Save also supports rendering of the plot to SVG, PNG, JPEG or PDF. Simply pass in a filename with the appropriate file extension. Note that this requires full graphical support for the NanoLab GUI. This support can be lacking on clusters or computing nodes. When generating plots on a computing node, they must then be saved in HDF5 files. The plots can then be rendered to SVG, PNG, JPEG or PDF on a machine with the proper graphical support.

Parameters

- **plot** (*BaseLayout based | PlotModel*) – The plot to save.
- **filename** (*str*) – The filename.
- **size** (*NoneType | 2-tuple of ints*) – The plot size in pixels.
- ****kwargs** (*dict*) – Additional arguments passed on to matplotlib when saving a figure.

Plot.show

show(*plot*, *size=None*)

Show an interactive plot.

Note that this function requires full graphical support for the NanoLab GUI. This support can be lacking on clusters or computing nodes. When generating plots on a computing node, they must then be saved in HDF5 files. The plots can then be shown on a machine with the proper graphical support.

Parameters

- **plot** (*Plot | BaseLayout based | PlotModel*) – The plot to show.
- **size** (*NoneType | 2-tuple of ints*) – The plot size in pixels.

Returns
The final layout and size.

Return type
2-tuple of GridLayout and size tuple

The NanoLab Plot module makes it possible to create and manipulate publication quality plots. Read more in the dedicated *NanoLab Plot Reference Manual*.

4.4.13 Full QuantumATK package

QuantumATK

Modules

Classes

ARTnSaddleSearch

class ARTnSaddleSearch(*constraints=None*, *initial_direction_generator=None*, *index_selector=None*, *active_sphere=None*, *min_move_distance=None*, *adaptive_displacement=False*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *optimizer_method=None*, *lanczos_relative_error=None*, *lanczos_maximum_iterations=None*, *finite_difference=None*, *finite_difference_method=None*, *threshold_orthogonal_steps=None*, *inflection_threshold=None*, *reorthogonalize=None*)

Class for performing a saddle search using the ARTn algorithm.

Parameters

- **constraints** (*list of ints*) – The atoms that should be constrained during the process search. **Default:** []
- **initial_direction_generator** (*DirectionGenerator*) – The generator for the initial displacements. **Default:** RandomDirection
- **index_selector** (*callable*) – A callable to select the indices to displace. Takes as input the current configuration and returns the indices to displace.
- **active_sphere** – This argument is used to control the radius of an active sphere around the atoms selected by the index_selector argument. All atoms that are outside this sphere will be constrained.
- **min_move_distance** (*PhysicalQuantity* of type length) – The minimum distance that the selected indices must have moved between reactant and saddle. All saddle configurations that do not meet this criteria will be discarded.
- **adaptive_displacement** (*bool*) – When using adaptive displacement the initial atom displacement magnitude will be set dynamically. The chosen magnitude will be 0.8 multiplied with the average distance between reactant and saddle point. This will overewrite the value set in by the initial_direction_generator.
- **max_forces** (*PhysicalQuantity* of type force) – The maximum forces allowed in the orthogonal optimization. **Default:** 0.01*eV/Angstrom
- **max_steps** (*int*) – The maximum number of uphill steps and orthogonal steps. **Default:** 200
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length for the uphill steps and orthogonal steps. **Default:** 0.1*Angstrom
- **optimizer_method** (list of OptimizerMethod) – The optimizer method to use. **Default:** LBFGS
- **lanczos_relative_error** (*float*) – The relative error for the Lanczos algorithm. **Default:** 0.0001
- **lanczos_maximum_iterations** (*int*) – The maximum number of iterations for the Lanczos algorithm. **Default:** 20
- **finite_difference** (*PhysicalQuantity* of type length) – The step size for the finite difference estimation of the second derivatives. **Default:** 0.01*Angstrom
- **finite_difference_method** (*Forward | Central*) – The type of finite difference used to estimate the second derivatives. **Default:** Central

- **threshold_orthogonal_steps** (*int*) – The maximum number of orthogonal steps below the inflection point. **Default:** 5
- **inflection_threshold** (*PhysicalQuantity* of type energy per length squared=) – The threshold for the inflection point of the energy after which the algorithm changes from following the initial direction to the minimum mode. Uses the current estimate of the lowest eigenvalue. A value of 0.0*eV/Angstrom**2 means that the actual inflection point is used. Values smaller than 0.0*eV/Angstrom**2 will cause the switching to be delayed. **Default:** 0.0*eV/Angstrom**2

reset()

Reset the direction generator.

run(*reactant_configuration*)

Run the saddle search algorithm.

Parameters
reactant_configuration (*AtomicConfiguration*) – The configuration to start the search from.

Returns
The data from the saddle search.

Return type
SaddleSearchData

setRandomSeed(*seed*)

Set the random seed for the initial direction generator.

Parameters
seed (*int*) – The random seed.

Notes

ARTnSaddleSearch is one of the three saddle search methods available for use in *AdaptiveKineticMonteCarlo*. It uses the original Activation Relaxation Technique nouveau (ARTn) as described in[1] and[2].

ARTnSaddleSearch requires an initial direction which is used to initiate the saddle search, controlled by the initial_direction_generator argument. The direction generator should be a callable that returns a vector. This callable will be used to generate a configuration from which the saddle search is started. Therefore, care must be taken so that it returns a vector of the correct dimensionality. There are two pre-made direction generators available. The RandomDirectionGenerator generates a random direction, while the *HypersphereDirection* generates a radial vector on a hypersphere of given radius. Each time a new saddle search is initiated, a new initial direction is generated.

Note that the magnitude argument of the initial direction generator is ignored when using *ARTnSaddleSearch*.

In addition to the initial direction, one can specify which atom(s) to displace. This is done using the index_selector argument. This argument can be either a list of indices or a (picklable) callable.

[1] Antoine Jay, Miha Gunde, Nicolas Salles, Matic Pobernik, Layla Martin-Samos, Nicolas Richard, Stefano deăGironcoli, Normand Mousseau, and Anne Hémeryck. Activation–relaxation technique: an efficient way to find minima and saddle points of potential energy surfaces. *Computational Materials Science*, 209:111363, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0927025622001446, doi:https://doi.org/10.1016/j.commatsci.2022.111363.

[2] Jisu Jung, Hyungmin An, Jinhee Lee, and Seungwu Han. Modified activation-relaxation technique (artn) method tuned for efficient identification of transition states in surface reactions. *Journal of Chemical Theory and Computation*, 20(18):8024–8034, 2024. URL: https://doi.org/10.1021/acs.jctc.4c00767, doi:10.1021/acs.jctc.4c00767.

Acidity

class Acidity(*reactant_species*, *product_species*, *solvent_components*, *temperature=None*, *parameters=None*)

Calculate the pKa of a hydrogen transfer reaction taking into account solvation effects.

Parameters

- **reactant_species** (Sequence of *CosmoRealSpecies*) – The neutral reactants in the in the proton transfer reaction
- **product_species** (Sequence of *CosmoRealSpecies*) – The ionized products in the in the proton transfer reaction
- **solvent_components** (*CosmoRealSolvent* | *MoleculeConfiguration* | *CosmoRSMixture*) – The solvent in which the proton transfer occurs
- **temperature** (*PhysicalQuantity* of type temperature) – The solvent temperature. **Default:** `298 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

acidity()

Returns
The pKa of the reaction, defined as the negative base-10 logarithm of the equilibrium constant.

Return type
float

parameters()

Returns
The COSMO-RS parameters

Return type
CosmoRSParameters

productSpecies()

Returns
The ionized products in the proton transfer reaction.

Return type
Sequence of *CosmoRealSpecies*

reactantSpecies()

Returns
The neutral reactants in the in the proton transfer reaction.

Return type
Sequence of *CosmoRealSpecies*

solventComponents()

Returns
The components of the solvent.

Return type
CosmoRSMixture

temperature()

Returns
The solvent temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Calculate the pK_a of acetic acid in water.

```
# Create the species for acetate
cosmo_rs = CosmoRS(
    configuration=optimized_configuration,
    solvation_energy=solvation_energy
)
acetate = cosmo_rs.evaluate()

# Calculate the pKa of acetic acid
acetic_acid_pka = Acidity(
    reactant_species=[acetic_acid, water],
    product_species=[acetate, hydronium],
    solvent_components=water,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters()
)
pka = acetic_acid_pka.acidity()
nlprint(f'The calculated pKa of acetic acid is {pka}')
```

acetic_acid_example.py

Notes

The *Acidity* object calculates the pK_a of a compound. It does this by calculating the acid/base equilibrium of the form:

$$AH + B \rightleftarrows A^- + BH^+$$

Here A acts as an acid transferring its proton to molecule B which acts as a base, forming two charged species. The molecule B can either be a solvent molecule or some other. In cases where water is a component of the solvent, B will often be water, forming a hydronium ion on the right hand side of the equation.

In COSMO-RS, pK_a is calculated based on the solvated energies of each molecule as well as each molecules interaction with solvent. The pK_a can be simply given as:

$$pK_a = \frac{E_{COSMO}^{A^-+BH^+} - E_{COSMO}^{AH+B} + \mu_S^{A^-+BH^+} - \mu_S^{AH+B}}{\ln(10)RT}$$

Here E_{COSMO} represents the COSMO internal energy of each pair of molecules and μ_S represents the solvated chemical potential of each molecular species.

Due to systematic errors in calculating both the internal energy and the solvated chemical potential, it is possible to use a linear correction to estimate the observed pK_a[1]. Given the pK_a calculated by the COSMO-RS method as pK_a^{raw}, the

[1] Andreas Klamt, Frank Eckert, Michael Diedenhofen, and MichaelăE. Beck. First principles calculations of aqueous pKa values for organic and inorganic acids using COSMO-RS reveal an inconsistency in the slope of the pKa scale. *J. Phys. Chem. A.*, 107:9380–9386, 2003. doi:10.1021/jp034688o.

corrected pK_a can be given as:

$$pK_a = A \times pK_a^{raw} + B$$

Here A and B are simply the slope and the intercept of the linear correction. They are both parameters contained in the *CosmoRSParameters* object.

It should also be noted that, with the exception of hydronium, all molecules in the COSMO-RS molecule database are neutral molecules. For pK_a calculations it will then be necessary to perform a DFT calculation on the deprotonated molecule. In this case, since it is a charged molecule, *Multipole* boundary conditions should be used. While slower than used *Periodic FFT* boundary conditions, *Multipole* boundary conditions ensure that the molecule is treated as a single, charged molecule without periodicity.

AcousticDeformationPotentialSelfEnergy

class AcousticDeformationPotentialSelfEnergy(*coupling=None*)

A class which represents a deformation potential model for acoustic phonon modes.

Parameters
coupling (*PhysicalQuantity* of type energy) – The electron-phonon coupling strength of the acoustic modes. The coupling strength can be given as a single value, or a 1D array specifying the value for each atom (in units of energy).
Default: `0 * eV`

coupling()

Returns
The electron-phonon coupling strength of the acoustic modes.

Return type
PhysicalQuantity with unit `eV**2`

uniqueString()

Return a unique string representing the state of the object.

ActiveLearningSimulation

class ActiveLearningSimulation(*fitting_parameters*, *initial_training_data*, *mtp_study_filename*, *mtp_study_object_id*, *reference_calculator*, *correction_calculator=None*, *d3_dispersion_calculator=None*, *candidate_threshold=None*, *retrain_threshold=None*, *check_interval=None*, *limit_candidates=None*, *max_forces_check=None*, *use_stress=None*, *candidate_trajectory_filename=None*, *candidate_trajectory_object_id=None*, *max_iterations=None*, *processes_per_calculation=None*, *extrapolation_selection_parameters=None*, *use_linearized_coefficient_matrix=None*, *minimum_bond_length_percent=None*, *log_filename_prefix=None*, *restart_simulation=None*)

Set up an object that can be used to run active learning simulations using the Moment Tensor Potential.

Parameters

- **fitting_parameters** (*MomentTensorPotentialFittingParameters*) – The parameters for MTP fitting.

- **initial_training_data** (*TrainingSet* | *Trajectory* | *ConfigurationDataContainer* | *MDTrajectory* | *MomentTensorPotentialTraining* | sequence of [*TrainingSet* | *Trajectory* | *ConfigurationDataContainer* | *MDTrajectory* | *MomentTensorPotentialTraining*] | Table) – The initial training data. All configurations in the initial training data must have precalculated reference data (energy, forces, and possibly stress), otherwise the configuration is not included in the active learning training.
- **mtp_study_filename** (*str*) – The file name of file that contains the MTP study object used for training the moment tensor potential.
- **mtp_study_object_id** (*str*) – The object_id of the MTP study object used to train the moment tensor potential.
- **reference_calculator** (*Calculator*) – The reference calculator.
- **correction_calculator** (*Calculator*) – Calculator used to correct reference values. The MTP is fit to the reference calculator minus the contribution of the correction calculator.
- **d3_dispersion_calculator** (*TremoloXCalculator*) – A D3 dispersion calculator that, when added, is used in the MD simulation during the active learning cycle as a correction to the MTP output. **Default:** None
- **candidate_threshold** (*float*) – The threshold above which configurations will be added as candidates for the next training iteration. **Default:** 1
- **retrain_threshold** (*float*) – The extrapolation threshold at which the simulation is stopped and trained again. **Default:** 3
- **check_interval** (*int*) – The interval at which the extrapolation grade is calculated and checked in MD simulations. In active learning optimization the extrapolation grade is always checked at every step. **Default:** 10
- **limit_candidates** (*int*) – Upper limit for the number of candidates collected each iteration that trigger retraining. This can be used to limit the number of reference calculator calls when many configurations fall between the candidate and retrain thresholds. **Default:** No limit
- **max_forces_check** (*PhysicalQuantity* of type energy / length) – If the max. force on an atom exceeds this value the extrapolation grade is always checked. Only available with MD active learning; has no effect in other cases. **Default:** 10*eV / Angstrom
- **use_stress** (*bool*) – Whether or not stress is used in training the MTP potential. **Default:** True
- **candidate_trajectory_filename** (*str*) – The filename for the trajectory to store the candidate configurations that have been added to the training set.
- **candidate_trajectory_object_id** (*str*) – The object ID for the trajectory to store the candidate configurations that have been added to the training set.
- **max_iterations** (*int*) – The max number of retraining iterations. **Default:** 20
- **processes_per_calculation** (int | `None`) – The number of processes used for each calculation with the reference calculator. **Default:** All processes corresponding to `None`.
- **extrapolation_selection_parameters** (*ExtrapolationSelectionParameters*) – The parameters that specify the details of the extrapolation grade calculation and candidate selection.
- **use_linearized_coefficient_matrix** (*bool*) – **Deprecated:** Use `extrapolation_grad_algorithm=MaxvolLinearized` in *ExtrapolationSelectionParameters* instead.
- **minimum_bond_length_percent** (*float*) – The minimum percentage of bond length as fraction of the covalent radii that is allowed in a training configuration. If any bond is less than that value then the configuration will be discarded from the training data. **Default:** 0.35
- **log_filename_prefix** (*str | LogToStdOut | None*) – The prefix used in the log file names generated by the training data calculation and fitting parts of the simulation.
- **restart_simulation** (*bool*) – Restart the simulation with any configurations added during previous active learning. simulations **Default:** False

additionalTrainingData()

Returns
The training dataset that has been added during the active learning simulation.

Return type
Trajectory

additionalTrainingSet()

Returns
The training dataset that has been added during the active learning simulation or None if no training data has been added.

Return type
TrainingSet | None

candidateThreshold()

Returns
The candidate threshold.

Return type
float

candidateTrajectoryFilename()

Returns
The filename to which the candidate trajectory is written.

Return type
str

candidateTrajectoryObjectId()

Returns
The object ID to which the candidate trajectory is written.

Return type
str

checkInterval()

Returns
The check interval for the extrapolation grade.

Return type
int

committeeSize()

Returns
The number of committee members when the query-by-committee method is used for the extrapolation grade.

Return type
int

correctionCalculator()

Returns
The correction calculator, if defined.

Return type
Calculator

`extrapolationGradeAlgorithm()`

Returns
Which extrapolation grade algorithm should be used.

Return type
MaxvolStandard | MaxvolLinearized | MaxForce | QueryByCommitteeForces | QueryByCommitteeEnergy

`fittingParameters()`

Returns
The MTP fitting parameters.

Return type
MomentTensorPotentialFittingParameters

`forcesCap()`

Returns
The forces cap.

Return type
PhysicalQuantity of type energy / length.

`initialTrainingData()`

Returns
The initial training dataset.

Return type
Trajectory

`initialTrainingSet()`

Returns
The initial training dataset for this active learning simulation.

Return type
TrainingSet

`limitCandidates()`

Returns
Upper limit for the number of candidates that are collected each iteration before retraining.

Return type
int

`logFilenamePrefix()`

Returns
The prefix used in MTP training log files. The flag LogToStdOut causes log output to be written to standard out.

Return type
str

`maxForcesCheck()`

Returns
The max. forces value at which a check of the extrapolation grade is enforced.

Return type
PhysicalQuantity of type energy / length.

maxIterations()

Returns
The maximum number of retraining iterations.

Return type
int

minBondLengthPercent()

Returns
The minimum bond length as fraction of the covalent radii.

Return type
float

mtpParameters()

Returns
Gets the MTPParameters containing the Optimized MTP parameters

Return type
MTPParameters | None

processesPerCalculation()

Returns
The number of processes that is used for a configuration when calculating the reference data.

Return type
int

referenceCalculator()

Returns
The reference calculator

Return type
Calculator

restartSimulation()

Returns
Whether or not the simulation and training are restarted from a previous run.

Return type
bool

retrainThreshold()

Returns
The retrain threshold.

Return type
float

runMolecularDynamics(*configuration*, *constraints=None*, *trajectory_filename=None*, *steps=None*, *log_interval=None*, *method=None*, *xyz_filename=None*, *hook_functions=None*, *pre_step_hook=None*, *post_step_hook=None*, *write_velocities=None*, *write_forces=True*, *write_stresses=None*, *domain_decomposition_pattern=None*, *trajectory_interval=None*, *measurement_hook=None*, *trajectory_object_id=None*, *number_of_independent_runners=None*, *log_filename_prefix=None*)

Run an active learning MD simulation.

Parameters

- **configuration** (*BulkConfiguration* | sequence of type *BulkConfiguration*) – The initial configuration. When using multiple independent runners this can be given as a list with a different item for each runner.
- **constraints** (list of ints | list of BaseConstraint) – The list of atomic indices, denoting fixed atoms, or constraint objects, such as *RigidBody*. **Default:** [].
- **trajectory_filename** (*str | sequence of str | None*) – The filename of the file to be used for storing the trajectory, or None if no trajectory should be written. A trajectory filename should not be given if configuration is a trajectory. When using multiple independent runners this can be given as a list with a different item for each runner. **Default:** None.
- **steps** (*int*) – The number of time-steps to take in the simulation. **Default:** 50.
- **log_interval** (*int*) – The interval at which information, such as time, energy, temperature, etc. is written to the log output. **Default:** 1.
- **method** (BaseMDmethod) – The MD method used for the simulation. When using multiple independent runners this can be given as a list with a different item for each runner. **Default:** *NVEVelocityVerlet*.
- **xyz_filename** (*str*) – The name of the file to be used for storing the xyz trajectory, or None if no xyz-trajectory should be written. **Default:** None.
- **hook_functions** (*None*) – Currently not supported in active learning.
- **pre_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments (step, time, configuration). The return status is ignored. Unhandled exceptions will terminate the evaluation. If a list is given the functions will be called in the given order. **Default:** None.
- **post_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments (step, time, configuration). Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return status is ignored. Unhandled exceptions will terminate the evaluation. If a list is given the functions will be called in the given order. **Default:** None.
- **write_velocities** (*bool*) – Write the velocities to the trajectory file every log_interval steps. If configuration is a trajectory (i.e. this is a restart calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** True.
- **write_forces** (*bool*) – Write the forces to the trajectory file every log_interval steps. If configuration is a trajectory (i.e. this is a restart calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** True.
- **write_stresses** (*bool*) – Write the stress to the trajectory file every log_interval steps. A value of None means that write_stresses is True by default for NPT methods and False otherwise. This is to avoid the additional work of calculating the stress when it is not needed. If configuration is a trajectory (i.e. this is a restart calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** None.
- **domain_decomposition_pattern** (list of type int | Automatic | None) – The pattern how the domains should be arranged in a parallel simulations. E.g. [1, 2, 4] means 1 domain in A-, 2 in B-, and 4 in C-direction. If Automatic domain decomposition is used, then the simulation cell will be divided into domains whose edges are as close together in length as possible. If None is given then domain decomposition will be disabled. **Default:** Automatic.
- **trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If configuration is a trajectory (i.e. this is a restart simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** The same value as log_interval.

- **measurement_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called at the end of each step after all constraints have been applied. The signature of the function requires the arguments (step, time, configuration). Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and may be accessed using the measurement method. Unhandled exceptions will terminate the evaluation. If a list is given the functions will be called in the given order. **Default:** None.

- **trajectory_object_id** (*str | None*) – The object id of the trajectory written to `trajectory_filename`. If a value of `None` is given, then an object id will be chosen automatically. **Default:** None.

- **number_of_independent_runners** (*int*) – The number of independent simulations that should be run. If greater than 1, each runner will run in an independent process group and at the end of each iteration, the candidates of all runners will be gathered, selected, and added to the training data. The simulation completes if all runners complete without exceeding the retraining threshold. **Default:** Length of `configuration`.

- **log_filename_prefix** (*str | None*) – The prefix used in log files containing details of the molecular dynamics simulation.

Returns
The final trajectory or, in case of multiple runners, a list containing the trajectories of all runners. In case the active learning simulation does not complete within the max. number of iterations, None is returned.

Return type
MDTrajectory | list | None

runOptimizeGeometry(*configuration*, *max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *optimize_cell=None*, *disable_stress=None*, *optimizer_method=None*, *target_stress=None*, *constrain_bravais_lattice=None*, *trajectory_interval=None*, *remove_drift=None*, *enable_optimization_stop_file=None*, *restart_strategy=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.NoRestart'>*, *number_of_independent_runners=None*, *log_filename_prefix=None*, *continue_on_non_convergence=None*)

Run an active learning geometry optimization.

Parameters

- **configuration** (*BulkConfiguration* | sequence of type *BulkConfiguration*) – The configuration to be optimized. When using multiple independent runners this can be given as a list with a different item for each runner.

- **max_forces** (*PhysicalQuantity* of type force) – The convergence criterion for the atomic forces. **Default:** 0.05*eV/Angstrom.

- **max_stress** (*PhysicalQuantity* of type pressure) – The convergence criterion for the maximum difference between the internal stress and the target stress. **Default:** 0.1*GPa.

- **max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200.

- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.

- **constraints** (*list of integers and Constraints objects*) – A list of indices of the atom with fixed positions and Constraints objects. **Default:** [].

- **trajectory_filename** (*str | sequence of str | None*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. When using multiple independent runners this can be given as a list with a different item for each runner. **Default:** None.

•**trajectory_object_id** (*str | None*) – The object id of the trajectory written to `trajectory_filename`. If a value of `None` is given, then an object id will be chosen automatically. **Default:** None.

•**optimize_cell** (*bool*) – The lattice vectors for bulk configuration will change during the optimization. Enabling the stress calculation for bulk configurations. **Default:** True for BulkConfigurations otherwise False.

•**disable_stress** (*bool*) – **Deprecated:** from v2022.03, use `optimize_cell` parameter instead.

•**optimizer_method** (*FIRE* | *LBFGS*) – The optimization algorithm to use. **Default:** *LBFGS*.

•**target_stress** (*PhysicalQuantity* of type pressure|PHYSICALQUANTITY| of type pressure) – The target internal stress (tensor) of the system. Can be given as a single value in case of isotropic pressure, or as an internal stress vector in Voigt notation or as a 3x3-matrix. **Default:** 0*GPa.

•**constrain_bravais_lattice** (*bool*) – Enable preserving the Bravais lattice symmetry of the configuration. **Default:** True if the `target_stress` is commensurate with the lattice symmetries.

•**trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1.

•**remove_drift** (*bool*) – In ab-initio calculations, the sum of the forces along each Cartesian direction does not necessarily sum to zero due to numerical inaccuracies. This option controls if the drift in the forces should be removed by subtracting the average force along each Cartesian direction from all atoms. **Default:** True.

•**enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the geometry optimization. If `True`, creation of the stop file will stop the optimization at the next step. The name of the stop file will be shown in the log output; it will be `stop-geometry-optimization-uniqueID`, where `uniqueID` is a randomly generated identifier for this optimization. The file must be created in the current working directory. **Default:** True.

•**restart_strategy** (`NoRestart`) – The restart mechanism has to be set to `NoRestart` for ActiveLearningSimulation **Default:** `NoRestart`.

•**number_of_independent_runners** (*int*) – The number of independent geometry optimizations that should be run. If greater than 1, each runner will run in an independent process group and at the end of each iteration, the candidates of all runners will be gathered, selected, and added to the training data. The simulation completes if all runners complete without exceeding the retraining threshold. **Default:** Length of `configuration`.

•**log_filename_prefix** (*str | None*) – The prefix used in log files containing details of the geometry optimization.

•**continue_on_non_convergence** (*bool*) – Continue with the converged configurations when using multiple runners and only some optimizations converge in the given number of active learning cycles. If `False` is given the calculation will raise an exception if any optimization does not converge. **Default:** False

runOptimizeNudgedElasticBand(*neb*, *max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *spring_constant=None*, *climbing_image=None*, *preoptimization=None*, *optimizer_method=None*, *log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *optimize_cell=None*, *target_stress=None*, *trajectory_interval=None*, *restart_strategy=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.NoRestart'>*)

Run an active learning nudged elastic band optimization.

Parameters

•**neb** (*NudgedElasticBand*) – The nudged elastic band configuration to optimize.

•**max_forces** (*PhysicalQuantity* of type force) – The convergence criterion for the atomic forces. **Default:** 0.05*eV/Angstrom.

- **max_stress** (*PhysicalQuantity* of type pressure) – The convergence criterion for the maximum difference between the internal stress and the target stress. **Default:** 0.1*GPa.
- **max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200.
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.
- **constraints** (*list of ints*) – List of atom indices that are kept fixed during optimization. **Default:** [].
- **trajectory_filename** (*str | sequence of str | None*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. **Default:** None.
- **spring_constant** (*PhysicalQuantity* of type eV/Ang**2) – The spring constant used for the NEB relaxation. **Default:** 5.0*(eV/Ang**2).
- **climbing_image** (*bool*) – Flag indicating if the climbing image algorithm should be used to find a transition state. **Default:** False.
- **preoptimization** (*bool*) – Flag indicating if the endpoints should be optimized before the NEB optimization. **Default:** False.
- **optimizer_method** (*FIRE* | *LBFGS*) – The optimizer to use for optimizing the structure. **Default:** *LBFGS*.
- **log_filename_prefix** (`Automatic` | str | None) – The logging output from each image will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** `Automatic`.
- **optimize_cell** (*bool*) – If it is *None*, it will be set automatically according to the NEB configuration. If the NEB configuration is composed of configurations of the type *BulkConfiguration* and the lattice vectors of the configurations are different, it is set to True; otherwise it is False. **Default:** None.
- **target_stress** (*PhysicalQuantity* of type pressure) – The target internal stress (tensor) of the system. Can be given as a single value in case of isotropic pressure, or as an internal stress vector in Voigt notation or as a 3x3-matrix. **Default:** 0*GPa.
- **trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1.
- **restart_strategy** (`NoRestart`) – The restart mechanism has to be set to `NoRestart` for ActiveLearningSimulation **Default:** `NoRestart`.

Returns
The optimized NEB. In case the active learning MD simulation does not complete within the max. number of iterations, None is returned.

Return type
NudgedElasticBand | None

runTimeStampedForceBiasMonteCarlo(*configuration*, *constraints=None*, *trajectory_filename=None*, *steps=None*, *log_interval=None*, *method=None*, *hook_functions=None*, *pre_step_hook=None*, *post_step_hook=None*, *write_velocities=None*, *write_forces=None*, *write_stresses=None*, *trajectory_interval=None*, *trajectory_object_id=None*, *number_of_independent_runners=None*, *log_filename_prefix=None*)

Run an active learning TFMC simulation.

Parameters

- **configuration** (*BulkConfiguration* | sequence of type *BulkConfiguration*) – The initial configuration. When using multiple independent runners this can be given as a list with a different item for each runner.

•**constraints** (list of ints | list of BaseConstraint) – The list of atomic indices, denoting fixed atoms, or constraint objects, such as *RigidBody*. **Default:** [].

•**trajectory_filename** (*str | sequence of str | None*) – The filename of the file to be used for storing the trajectory, or None if no trajectory should be written. A trajectory filename should not be given if configuration is a trajectory. When using multiple independent runners this can be given as a list with a different item for each runner. **Default:** None.

•**steps** (*int*) – The number of time-steps to take in the simulation. **Default:** 50.

•**log_interval** (*int*) – The interval at which information, such as time, energy, temperature, etc. is written to the log output. **Default:** 1.

•**method** (ForceBiasMontCarlo | *ForceBiasMonteCarloNPTBerendsen*) – The Monte Carlo method used for the simulation. When using multiple independent runners this can be given as a list with a different item for each runner. **Default:** *ForceBiasMonteCarlo*

•**hook_functions** (*None*) – Currently not supported in active learning.

•**pre_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments (step, time, configuration). The return status is ignored. Unhandled exceptions will terminate the evaluation. If a list is given the functions will be called in the given order. **Default:** None.

•**post_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments (step, time, configuration, forces, stress). The return status is ignored. Unhandled exceptions will terminate the simulation. If a list is given the functions will be called in the given order. **Default:** None

•**write_velocities** (*bool*) – Write the velocities to the trajectory file every log_interval steps. Since, the time-stamped force-bias Monte Carlo algorithm does not use velocities explicitly, zero velocities will be written. If configuration is a trajectory (i.e. this is a restart calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** False

•**write_forces** (*bool*) – Write the forces to the trajectory file every log_interval steps. **Default:** True

•**write_stresses** (*bool*) – Write the stress to the trajectory file every log_interval steps. **Default:** False

•**trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If configuration is a trajectory (i.e. this is a restart MD simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** The same value as log_interval.

•**trajectory_object_id** (*str | None*) – The object id of the trajectory written to trajectory_filename. If a value of None is given, then an object id will be chosen automatically. **Default:** None.

•**number_of_independent_runners** (*int*) – The number of independent simulations that should be run. If greater than 1, each runner will run in an independent process group and at the end of each iteration, the candidates of all runners will be gathered, selected, and added to the training data. The simulation completes if all runners complete without exceeding the retraining threshold. **Default:** Length of configuration.

•**log_filename_prefix** (*str | None*) – The prefix used in log files containing details of the molecular dynamics simulation.

Returns
The final trajectory or, in case of multiple runners, a list containing the trajectories of all runners. In case the active learning simulation does not complete within the max. number of iterations, None is returned.

Return type
MDTrajectory | list | None

static supportedConfigurationTypes()

Returns
Supported configuration types for initial training data.

Return type
tupleColuz

trainingSetTable()

Returns
A table containing the initial and additional training data for this active learning simulation.

Return type
Table

uniqueString()

Return a unique string representing the state of the object.

useLinearizedCoefficientMatrix()

Returns
Whether the matrix used to calculate the extrapolation grade should include only the linear coefficients or additionally the linearized version of the non-linear coefficients.

Return type
bool

static validatorType()

Returns
The validator type.

Return type
Validator

writeAtomicErrorEstimates()

Returns
True if the per-atom error estimates should be written to the trajectory.

Return type
bool

Usage Examples

Run a molecular dynamics ActiveLearningSimulation for amorphous HfO2 using an initial training data set that is read from file. The method `runMolecularDynamics` takes the same arguments as the normal *MolecularDynamics* function. The extrapolation process is done by *query-by-committee* method.

```
# Load the training data. This can be one or several TrainingSet,␣
↪MomentTensorPotentialTraining or
# Trajectory objects, which contain energy, forces, stress calculated with the reference
# calculator.
initial_training_data = [nlread('HfO2_crystal_training.hdf5', TrainingSet)[0]]

# Use the predefined small MTP basis.
mtp_basis = PredefinedBasisSmall
```

(continues on next page)

(continued from previous page)

```
# Optimize the non-linear coefficients on the energy only.
nl_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=True,
)

fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=mtp_basis,
    outer_cutoff_radii=4.5*Ang,
    mtp_filename='HfO2_active_learning.mtp',
    non_linear_coefficients_parameters=nl_parameters,
    use_element_specific_coefficients=True,
)

active_learning = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='HfO2_mtp_study',
    mtp_study_object_id='HfO2',
    reference_calculator=reference_calculator,
    candidate_threshold=1.0,
    retrain_threshold=3.0,
    check_interval=20,
    max_forces_check=10.0*eV/Ang,
    use_stress=True,
    candidate_trajectory_filename='HfO2_am_active_learning_candidates.hdf5',
    restart_simulation=True,
    extrapolation_selection_parameters=ExtrapolationSelectionParameters(
        extrapolation_grade_algorithm=QueryByCommitteeForces,
        descriptor_cutoff=0.1,
    ),
)

# Set up a high-temperature MD at 3000 K.
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=3000.0*Kelvin,
    remove_center_of_mass_momentum=True,
    random_seed=None,
    enforce_temperature=True,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=3000*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
)

constraints = [FixCenterOfMass()]

# Run the MD simulation through the active learning object.
```

(continues on next page)

(continued from previous page)

```
md_trajectory = active_learning.runMolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='HfO2_am_active_learning_3000K.hdf5',
    steps=100000,
    log_interval=100,
    method=method,
    domain_decomposition_pattern=[1, 1, 1],
)

# Extract the additional training data that has been added during active learning as a␣
↪TrainingSet
# object, and save it to a file.
additional_training_data = active_learning.additionalTrainingSet()
nlsave('HfO2_active_learning_additional_training_data.hdf5', additional_training_data)
```

active_learning_md_query_by_committee.py

Run a molecular dynamics ActiveLearningSimulation for amorphous HfO2 using an initial training data set that is read from file. The method `runMolecularDynamics` takes the same arguments as the normal *MolecularDynamics* function. The extrapolation process is done with the *maxvol* method.

```
# Load the training data. This can be one or several TrainingSet,␣
↪MomentTensorPotentialTraining or
# Trajectory objects, which contain energy, forces, stress calculated with the reference
# calculator.
initial_training_data = [nlread('HfO2_crystal_training.hdf5', TrainingSet)[0]]

# Use 300 MTP basis functions.
n_basis = 300

# Optimize the non-linear coefficients on the energy only.
nl_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=True,
)

fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=n_basis,
    outer_cutoff_radii=4.5*Ang,
    mtp_filename='HfO2_active_learning.mtp',
    non_linear_coefficients_parameters=nl_parameters
)

active_learning = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='HfO2_mtp_study',
    mtp_study_object_id='HfO2',
    reference_calculator=reference_calculator,
    candidate_threshold=1.0,
    retrain_threshold=3.0,
    check_interval=20,
```

(continues on next page)

(continued from previous page)

```
    max_forces_check=10.0*eV/Ang,
    use_stress=True,
    candidate_trajectory_filename='HfO2_am_active_learning_candidates.hdf5',
    restart_simulation=True,
)

# Set up a high-temperature MD at 3000 K.
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=3000.0*Kelvin,
    remove_center_of_mass_momentum=True,
    random_seed=None,
    enforce_temperature=True,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=3000*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
)

constraints = [FixCenterOfMass()]

# Run the MD simulation through the active learning object.
md_trajectory = active_learning.runMolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='HfO2_am_active_learning_3000K.hdf5',
    steps=100000,
    log_interval=100,
    method=method,
    domain_decomposition_pattern=[1, 1, 1],
)
```

active_learning_md.py

The following example shows how to set up active learning simulations with multiple runners. In this type of simulation, the MD is run with multiple different initial configurations, which are trained to simultaneously.

```
active_learning = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='HfO2_mtp_study',
    mtp_study_object_id='HfO2',
    reference_calculator=reference_calculator,
    candidate_threshold=1.0,
    retrain_threshold=3.0,
    check_interval=20,
    max_forces_check=10.0*eV/Ang,
    use_stress=True,
    candidate_trajectory_filename='HfO2_am_active_learning_candidates.hdf5',
    restart_simulation=True,
    extrapolation_selection_parameters=ExtrapolationSelectionParameters(
```

(continues on next page)

(continued from previous page)

```
        extrapolation_grade_algorithm=QueryByCommitteeForces,
        descriptor_cutoff=0.1,
    ),
)

# Set up a high-temperature MD at 3000 K.
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=3000.0*Kelvin,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=3000*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
)

constraints = [FixCenterOfMass()]

# Use 8 different initial configurations which are loaded from file and collected in a␣
↪list. The
# trajectory filenames are also set up here, so that each MD trajectory is written to a␣
↪different
# file.
number_of_initial_configurations = 8
initial_configuration_list = []
trajectory_filename_list = []
for i in range(number_of_initial_configurations):
    configuration = nlread(f'initial_configuration_{i}.hdf5', BulkConfiguration)[0]
    initial_configuration_list.append(configuration)
    trajectory_filename_list.append(f'HfO2_am_active_learning_3000K_{i}.hdf5')

# Run the MD simulation through the active learning object.
md_trajectory = active_learning.runMolecularDynamics(
    initial_configuration_list,
    constraints=constraints,
    trajectory_filename=trajectory_filename_list,
    steps=100000,
    log_interval=1000,
    method=method,
    domain_decomposition_pattern=[1, 1, 1],
)

# Extract the additional training data that has been added during active learning, as␣
↪TrainingSet
# object.
additional_training_data = active_learning.additionalTrainingSet()
```

`active_learning_md_multiple_runners.py`

When starting an active learning simulation from scratch it is recommended to scan over randomly generated initial non-linear coefficients and select the best fit for further training.

```
# This function replaces the setup of NonLinearCoefficientsParameters and
# MomentTensorPotentialFittingParameters without the need of a loop.
fitting_parameters_list = scanOverNonLinearCoefficients(
    number_of_initial_guesses=30,
    basis_size=n_basis,
    outer_cutoff_radii=4.5*Ang,
    mtp_filename_suffix='MTP_fit.mtp',
    use_element_specific_coefficients=True,
    random_seed=42,
)

# Perform an initial MTP training using a list of fitting parameters.
mtp_training = MomentTensorPotentialTraining(
    filename='pre_training.hdf5',
    object_id='mtp_training',
    training_sets=initial_training_data,
    calculator=reference_calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters_list,
    train_test_split=0.9,
    random_seed=13345,
    log_filename_prefix='pre_training',
)
mtp_training.update()

# Determine the best fit.
best_fit_index = mtp_training.rankFits(
    data_tags=None,
    weights=[[1, 1, 1], [1, 1, 1]],
    statistical_measure=R2Score
)[0][0]

# Get the parameters of the best fit which can be passed to the fitting_parameters␣
↪keyword of
# ActiveLearningSimulation.
best_fitting_parameters = mtp_training.fittingParametersList()[best_fit_index]

# Set up the active ActiveLearningSimulation as usual.
active_learning = ActiveLearningSimulation(
    fitting_parameters=best_fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='HfO2_mtp_study',
    mtp_study_object_id='HfO2',
    reference_calculator=reference_calculator,
    candidate_threshold=1.0,
    retrain_threshold=3.0,
    check_interval=20,
    max_forces_check=10.0*eV/Ang,
    use_stress=True,
    candidate_trajectory_filename='HfO2_am_active_learning_candidates.hdf5',
    restart_simulation=True,
)
```

active_learning_md_with_scan.py

Run an ActiveLearningSimulation geometry optimization.

```
# Set up non-linear coefficients.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=True,
)

# Set up parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=300,
    outer_cutoff_radii=4.5*Angstrom,
    mtp_filename='active_learning_mtp.mtp',
    non_linear_coefficients_parameters=non_linear_coefficients_parameters,
)

active_learning = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='mtp_study',
    mtp_study_object_id='mtp',
    reference_calculator=reference_calculator,
    candidate_threshold=0.1,
    retrain_threshold=1.0,
    check_interval=1,
    use_stress=True,
    candidate_trajectory_filename='active_learning_candidates.hdf5',
    restart_simulation=True,
    extrapolation_selection_parameters=ExtrapolationSelectionParameters(
        extrapolation_grade_algorithm=QueryByCommitteeForces,
        descriptor_cutoff=0.1,
    ),
)

constraints = [FixStrain(x=True, y=True, z=True)]

# Run the geometry optimization through the active learning object.
optimzation_configuration = active_learning.runOptimizeGeometry(
    bulk_configuration,
    max_steps=1000,
    constraints=constraints,
    trajectory_filename='active_learning_optimization_trajectory.hdf5',
    optimize_cell=False,
    trajectory_interval=1,
)

# Extract the additional training data that has been added during active learning, as␣
↪TrainingSet
# object.
additional_training_data = active_learning.additionalTrainingSet()
```

active_learning_optimization.py

Run an ActiveLearningSimulation nudged elastic band optimization.

```
# Set up non-linear coefficients.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=True,
)

# Set up parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=300,
    outer_cutoff_radii=4.5*Angstrom,
    mtp_filename='active_learning_mtp.mtp',
    non_linear_coefficients_parameters=non_linear_coefficients_parameters,
    use_element_specific_coefficients=True,
)

active_learning = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='mtp_study',
    mtp_study_object_id='mtp',
    reference_calculator=reference_calculator,
    candidate_threshold=0.1,
    retrain_threshold=1.0,
    check_interval=1,
    use_stress=True,
    candidate_trajectory_filename='active_learning_candidates.hdf5',
    restart_simulation=True,
)

# Run the NEB optimization through the active learning object.
optimzation_configuration = active_learning.runOptimizeNudgedElasticBand(
    neb,
    max_steps=1000,
    trajectory_filename='active_learning_optimization_trajectory.hdf5',
    optimize_cell=False,
    trajectory_interval=1,
)
```

active_learning_optimization_neb.py

Run an active learning MD and extract the training set table for a final fit to test different MTP basis sizes.

```
# Run the MD simulation through the active learning object.
md_trajectory = active_learning.runMolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='HfO2_am_active_learning_3000K.hdf5',
    steps=100000,
    log_interval=100,
    method=method,
    domain_decomposition_pattern=[1, 1, 1],
)
```

(continues on next page)

(continued from previous page)

```
# Extract the additional training data that has been added during active learning as a␣
↪TrainingSet
# object, and save it to a file.
additional_training_data = active_learning.additionalTrainingSet()
nlsave('HfO2_active_learning_additional_training_data.hdf5', additional_training_data)

# Extract a table with the initial and additional training data that has been added
# during active learning. This table can be used as input to a final MTP fit.
training_set_table = active_learning.trainingSetTable()

# Test different basis sizes.
fitting_parameters_list = []
for mtp_basis in [PredefinedBasisSmall, 400, 800]:
    # Optimize the non-linear coefficients on the energy only.
    nl_parameters = NonLinearCoefficientsParameters(
        perform_optimization=True,
        energy_only=True,
    )

    fitting_parameters = MomentTensorPotentialFittingParameters(
        basis_size=mtp_basis,
        outer_cutoff_radii=4.5*Ang,
        mtp_filename=f'HfO2_active_learning_{mtp_basis}.mtp',  # RJL: Not technically␣
↪active learning. Maybe rename.
        non_linear_coefficients_parameters=nl_parameters,
        use_element_specific_coefficients=True,
    )

    fitting_parameters_list.append(fitting_parameters)

# Perform an MTP training using the list of fitting parameters.
mtp_training = MomentTensorPotentialTraining(
    filename='Final_MTP_training_basis_size_scan.hdf5',
    object_id='mtp_training',
    training_sets=training_set_table,
    calculator=reference_calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters_list,
    train_test_split=0.9,
    random_seed=13345,
    log_filename_prefix='mtp_basis_size_scan',
)
mtp_training.update()
nlprint(mtp_training)
```

active_learning_md_query_by_committee.py

Notes

MTP Active Learning

The *ActiveLearningSimulation* class can be used to continuously extend the training data for a machine-learned Moment Tensor Potential (MTP) during a molecular dynamics simulation, as described in[1] and[2].

A typical active learning simulation is started from an initial training data set, which can be generated using the *MomentTensorPotentialTraining* framework. By calling a method such as `runMolecularDynamics` or `runOptimizeGeometry`, an active learning simulation is initiated. From the initial training data set a starting MTP is trained which is used to run the MD simulation. During the simulation the extrapolation grade is calculated at regular intervals for the current configuration, as described in[Page 2202, 1] and[2]. A value above zero means that the current configuration extrapolates the potential, i.e. is outside the space of configurations covered by the training data. If the extrapolation grade exceeds the first threshold `candidate_threshold`, a copy of the configuration is stored as a candidate to add to the extended training data set. If the extrapolation grade exceeds the `retrain_threshold`, the simulation is stopped. The most relevant configurations are selected from the collected candidates using the *maxvol* criterion[1]. Energy, forces, and stress are calculated for these configurations using the reference calculator, and added to the training data set. A new MTP is trained on this extended data set and the simulation is started from the beginning with the new MTP.

This cycle is repeated until the simulation completes without extrapolating significantly, i.e. without exceeding the retrain threshold, or the maximum number of iterations as specified by `max_iterations` is reached.

All additional configurations, which are added to the original training data set during an active learning simulation, are stored with their calculated reference energy, forces, and stress to a *ConfigurationDataContainer* object which is saved to `candidate_trajectory_filename`. The additional training data can also be be accessed via the method `additionalTrainingData()`. The method `additionalTrainingSet()` returns the additional training data as *TrainingSet* object. Furthermore, *ActiveLearningSimulation* provides a method `trainingSetTable()`, which returns a Table that contains both the initial training set as well as the additional training set. This table can directly be used as input to a *MomentTensorPotentialTraining* object to perform a final MTP fitting run and optimize the MTP hyperparameters.

A D3 dispersion calculator can be added in order to account for dispersion during the MD simulations of the active learning cycle. This is achieved by creating the *ActiveLearningSimulation* object with a D3 dispersion calculator attached via the `d3_dispersion_calculator` keyword. This calculator is required to be a *TremoloXCalculator* calculator wrapped around a given D3 dispersion potential. For creating the calculator a D3 dispersion potential, e.g. `DispersionD3Z(xc='PBE')`, has to used for creating a *TremoloXCalculator* object - yielding the overall input `TremoloXCalculator(DispersionD3Z(xc='PBE'))`. The type of D3 dispersion potential and the xc functional included in the *TremoloXCalculator* object should be explicitly specified according to the specific modeling needs.

[1] EvgenyăV. Podryabinkin and AlexanderăV. Shapeev. Active learning of linearly parametrized interatomic potentials. *Comput. Mater. Sci.*, 140:171–180, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0927025617304536, doi:https://doi.org/10.1016/j.commatsci.2017.08.031.

[2] Konstantin Gubaev, EvgenyăV. Podryabinkin, GusăL.W. Hart, and AlexanderăV. Shapeev. Accelerating high-throughput searches for new alloys with active learning of interatomic potentials. *Comput. Mater. Sci.*, 156:148–156, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0927025618306372, doi:https://doi.org/10.1016/j.commatsci.2018.09.031.

Query-by-committee

Query-by-committee is an alternative to the *maxvol* algorithm to calculate the extrapolation grade for multi-element systems. In this case, an ensemble of different MTP models is trained on the same training data (default is 6 different models), each with different initial guesses. The ensemble standard deviation of the forces prediction is used as a measure of extrapolation, which can be interpreted as a prediction of the forces error in the configuration[3]. The query-by-committee algorithm typically provides a numerically more stable way to calculate the extrapolation grade and does not require reduced MTP accuracy settings. In particular it can be used with any MTP basis size and it is therefore the recommended algorithm.

The query-by-committee method can be activated by setting `extrapolation_grade_algorithm=QueryByCommitteeForces` in *ExtrapolationSelectionParameters*. Unlike maxvol, query-by-committee does not provide a native algorithm for selecting the most different candidate configurations out of all extrapolating candidates. Instead, a dedicated separate selection algorithms is used (MTP-structural-descriptor). The parameters for this method can be selected through the `extrapolation_selection_parameters`, which takes an *ExtrapolationSelectionParameters* object.

Multiple runners

Using multiple runners in active learning MD or optimization is an extension to the standard way of running active learning to increase the exploration of different new configurations. Here, the simulation is split up and run on multiple different initial configurations simultaneously, collecting candidates from all simulations. When all simulations have stopped due to exceeding the retraining threshold, all the candidates are gathered, selected, and collectively included in the training before the next iteration is started. This allows to include, e.g. various stoichiometric compositions, or interface representations in the same active learning simulation, instead of running multiple simulations sequentially.

Multiple runners can be set up by passing a list of initial configurations as `configuration` parameter in `runMolecularDynamics()` or `runOptimizeGeometry()`. For running this type of simulation efficiently, one should ideally allocate at least as many MPI processes as number of initial configurations.

Practical guide

Although it can also be used in production simulations, the *ActiveLearningSimulation* class is primarily designed to efficiently extend existing training data via MD simulations without having to run computationally expensive ab-initio MD simulations for amorphous or high-temperature systems. Apart from the automatically added training data in the candidate trajectory file, it can also be useful to take snapshots from the final MD trajectory, recalculate energy, forces, and stress with the reference calculators (e.g. by passing it as *TrainingSet* to *MomentTensorPotentialTraining*) to obtain additional training configurations that can be added to a larger training data set.

For consistency with the input of *MomentTensorPotentialTraining* it is recommended to use a list or Table of *TrainingSet* objects to pass the initial training data.

When using the default maxvol algorithm, it is recommended to reduce the MTP accuracy settings, e.g. MTP basis, outer cutoff radius, and to include only the most relevant training data in the initial training data set to be run in a robust and efficient way. One can occasionally encounter the error message No new candidates found in active learning, often accompanied by very large extrapolation grade values. This is typically caused by an MTP basis set being too large for the given training data, which leads to numerical inaccuracies when calculating the extrapolation grade and selecting the candidates to add to the next training iteration. In this case it often helps to reduce the MTP basis size in the *MomentTensorPotentialFittingParameters* until the problem disappears. Alternatively, one can try to include more diverse configurations to the initial training data, for example by running DFT-MD.

Choosing the query-by-committee algorithm instead, can often avoid this problem from the start and provide a more stable simulation.

[3] Christoph Schran, Krystof Brezina, and Ondrej Marsalek. Committee neural network potentials control generalization errors and enable active learning. *The Journal of Chemical Physics*, 153(10):104105, 2020.

When starting an active learning simulation from scratch it is recommended to scan over randomly generated initial non-linear coefficients and select the best fit for further training, as shown in the example section. If an active learning simulation is continued, then it is best to use the non-linear coefficients from the previous run to keep he training consistent. This can be done by passing the MTP filename from the previous run as `initial_coefficients` parameter in the *NonLinearCoefficientsParameters*.

In order to run an active learning MD simulation more efficiently one can increase the `check_interval` parameter, to reduce the frequency at which the extrapolation grade is calculated. As a fallback the extrapolation grade is always checked, whenever the largest force on an atom exceeds the value `max_forces_check`, which is typically a sign of extrapolation. This is only supported for MD, whereas in optimization simulations the extrapolation grade is always checked at every step.

Note, that the initial training data must have pre-calculated reference data (e.g. DFT energy, forces, stress). That means e.g. `TrainingSet` objects must be set up with `recalculate_training_data=False` ActiveLearningSimulation does not support calculating the reference data for the initial training data that is given.

RJL This sentence seems to be cut off. The training data generated during an active learning run can be accessed via

Active Learning MD

A typical workflow for training via active learning MD could be to use displaced bulk crystal or interface configurations generated via *RandomDisplacementsParameters* or *CrystalInterfaceTrainingParameters* respectively, as initial training data. Simulating the crystal at high temperature using active learning MD to melt it can then be used to include liquid and amorphous configurations to the training data. Typical values for candidate and retrain threshold are 1.0 and 3.0, respectively. The larger these values are, the more extrapolation is allowed.

MD with active learning can be combined with hook functions. This can be used to include non-equilibrium simulations in the training.

Active Learning geometry optimization

This type of optimization can supplement molecular dynamics simulations by bridging e.g. amorphous and crystalline phases. Since optimization trajectories tend to be shorter than MD trajectories, and configurations can change more significantly between optimization steps, it is recommended to set the `check_interval` parameter to 1 and lower the candidate- and retrain thresholds to add sufficiently many new structures to the training set. Note that `restart_strategy` and hook functions available in *OptimizeGeometry* are not supported in *ActiveLearningSimulation*. This method is used when crystal structure prediction is run with an *ActiveLearningSimulation* object (see also the *CrystalStructurePrediction* reference manual).

Active Learning nudged elastic band optimization

This type of active learning optimization can be used to train potentials for reaction paths and transition states. Similarly to *OptimizeGeometry*, hook functions and `restart_strategy` arguments are not available when running *OptimizeNudgedElasticBand* through active learning. Every image in the *NudgedElasticBand* is checked individually for its extrapolation grade, potentially adding multiple candidate structures each optimization iteration. However, it is still recommended to set the `check_interval` parameter to 1. A preoptimization of the endpoints can be performed. In that case the initial MTP should be able to perform the optimization without unphysical results, as they are not part of the learning process. Note that multiple independent runners are not supported.

AdaptiveGrid

class AdaptiveGrid(*kA_range=None*, *kB_range=None*, *tolerance=None*, *error_measure=None*, *number_of_initial_levels=None*, *maximum_number_of_levels=None*, *kpoints=None*, *intervals=None*, *triangles=None*, *function_values=None*, *integral_value=None*)

Parameters

- **kA_range** (*list(2) of float | float*) – Interval for k-points in the kA-direction given in fractional coordinates. If a single number is given there will be no k-points in this direction. **Default:** `[-0.5, 0.5]`
- **kB_range** (*list(2) of float | float*) – Interval for k-points in the kA-direction given in fractional coordinates. If a single number is given there will be no k-points in this direction. **Default:** `[-0.5, 0.5]`
- **tolerance** (*Positive float*) – Tolerance for integration. **Default:** `1e-3`
- **error_measure** (Absolute | `Relative`) – Method to calculate error. Either `Absolute` (absolute error) or `Relative` (relative error). **Default:** `Absolute`
- **number_of_initial_levels** (*Non-negative int*) – Minimum number of refinement steps in the adaptive grid. **Default:** `3`
- **maximum_number_of_levels** (*Non-negative int*) – Maximum number of refinement steps in the adaptive grid. **Default:** `20`
- **kpoints** – List of initial k-point, e.g. `[[0.0, 0.0], [0.1, 0.0]]`. Mutually exclusive with kA_range and kB_range. Can be used for restarting from a previous calculation. **Default:** `None`
- **intervals** (*list(n_intervals) of list(2) of int*) – List of intervals represented two integers, e.g. `[(0, 1), (1, 2), (2, 3)]`. The integers correspond to the indices of the k-points that are the end points of each interval. Mutually exclusive with kA_range and kB_range. Can be used for restarting from a previous calculation. **Default:** `None`
- **triangles** (*list(n_triangles) of list(3) of int*) – List of triangles represented three integers, e.g. `[(0, 1, 2), (0, 2, 3)]`. The integers correspond to the indices of the k-points that are the corner points of each triangle. Mutually exclusive with kA_range and kB_range. Can be used for restarting from a previous calculation. **Default:** `None`
- **function_values** (*list(n_kpoints) of float*) – List of function values corresponding to each k-point. Mutually exclusive with kA_range and kB_range. Can be used for restarting from a previous calculation. **Default:** `None`

dimension()

Returns
The dimension of the k-point integration. Can be either 1 or 2.

Return type
int

errorMeasure()

Returns
The `error_measure` used for this AdaptiveGrid.

Return type
`Absolute` or `Relative`

functionValues()

Returns
The values of the integrated function in all the k-points.

Return type
ndarray(n) where n is the number of k-points.

`initialIntervals()`

Returns
A list of the initial intervals.

Return type
list(n) of `Intervals` where n is the number of intervals.

`initialKpoints()`

Returns
The list of initial k-points.

Return type
ndarray(n, m) of floats where n is the number of k-points and m is the dimension.

`initialTriangles()`

Returns
A list of the initial triangles.

Return type
list(n) of `Triangles` where n is the number of triangles.

`integralValue()`

Returns
The value of the integrated function.

Return type
float

`intervals()`

Returns
A list of the intervals once the integration is finished.

Return type
list(n) of `Intervals` where n is the number of intervals.

`kARange()`

Returns
The range of k-points in the kA-direction.

Return type
list(2) of float.

`kBRange()`

Returns
The range of k-points in the kB-direction.

Return type
list(2) of float.

`kpoints()`

Returns
All of the k-points used in the integration.

Return type
list(n) of k-points where n is the number of k-points.

`kpointsWeights()`

Returns
The weight of all the k-points used in the integration.

Return type
list(n) of float where n is the number of k-points.

`maximumNumberOfLevels()`

Returns
The number of maximum levels allowed in the adaptive integration.

Return type
int

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`numberOfInitialLevels()`

Returns
The number of initial levels used in the adaptive integration.

Return type
int

`tolerance()`

Returns
The tolerance at which the adaptive integration should stop.

Return type
float

`triangles()`

Returns
A list of the triangles once the integration is finished.

Return type
list(n) of `Triangles` where n is the number of triangles.

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Define an adaptive grid:

```
# Setup adaptive grid object.
adaptive_grid = AdaptiveGrid(
        kA_range=[-0.5, 0.5],
        kB_range=[-0.5, 0.5],
        tolerance=1e-2,
        error_measure=Absolute,
        number_of_initial_levels=2,
        maximum_number_of_levels=7)

# Use adaptive grid instead of kpoints in transmission spectrum.
transmission_spectrum = TransmissionSpectrum(device_configuration,
                                             kpoints=adaptive_grid)


# Use adaptive grid instead of kpoints in spin transfer torque.
spin_transfer_torque = SpinTransferTorque(device_configuration,
                                          contributions=Left,
                                          kpoints=adaptive_grid)
```

adaptive_grid.py

Notes

When using adaptive grids for integration, the integration domain is divided into triangles, as illustrated in Fig. 4.4. The integral over each triangle area is estimated as

$$I_0 = A_0(F_1 + F_2 + F_3)/3,$$

where A_0 is the area of the triangle, and F_1, F_2, F_3 are the function values at the triangle corner points. In order to estimate whether the integral, I_0, (left inset in Fig. 4.4) is converged or not, each triangle is divided into four smaller triangles with integral values I_1, I_2, I_3, I_4 (right inset in Fig. 4.4). The integral error, E_{I_0}, for triangle integral I_0 is computed as

$$E^{abs}_{I_0} = \frac{|I_0 - (I_1 + I_2 + I_3 + I_4)|}{A_0},$$

if the `error_measure` is `Absolute` and as

$$E^{rel}_{I_0} = \frac{|I_0 - (I_1 + I_2 + I_3 + I_4)|}{I_{tot}},$$

if the `error_measure` is `Relative`, where I_{tot} is the total integral over the entire integration domain at the present level of refinement.

It depends on the actual problem, if one should use the `error_measure=Absolute` or `error_measure=Relative` option. If one has some knowledge about the function values, e.g., the transmission spectrum, the use of `Absolute` may be more advantageous. On the other hand, if there is no information available on the function values, using `Relative` is a better option. This is also the case if `spin.Up` and `spin.Down` have rather different transmission spectra.

Fig. 4.5 illustrates schematically the data flow in the adaptive algorithm.

The AdaptiveGrid can be used as the kpoint keyword in the following analysis objects:

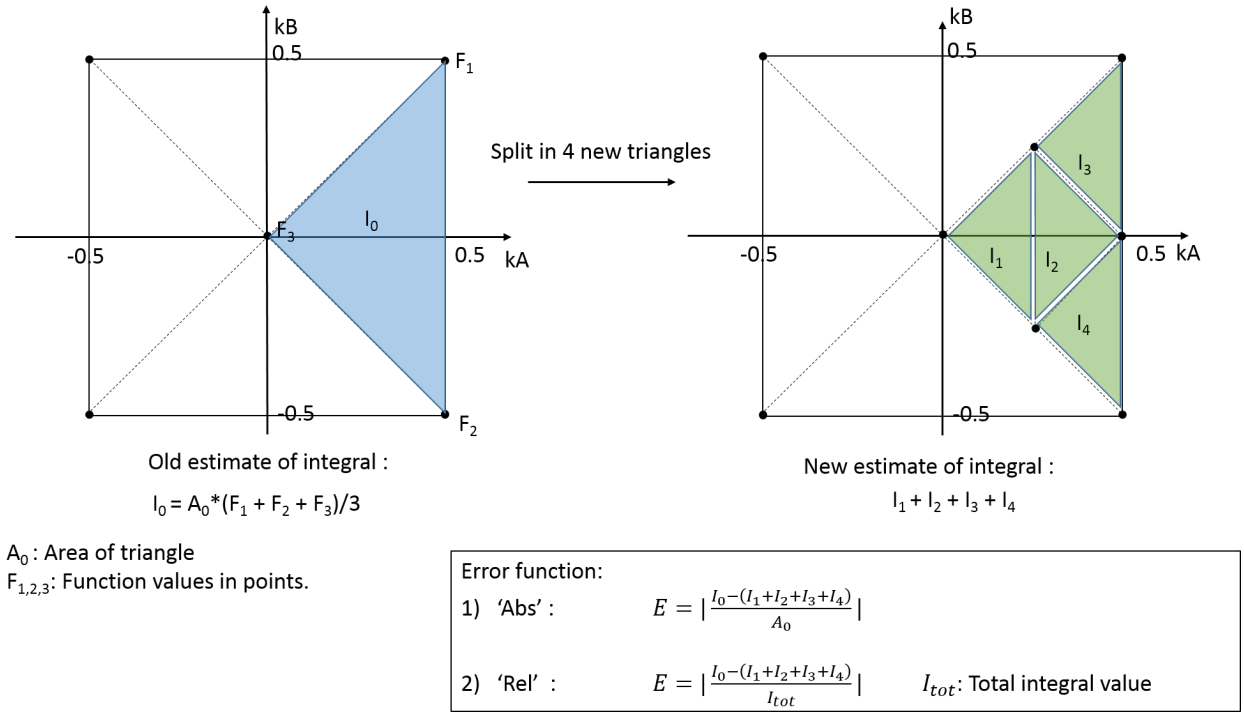


Fig. 4.4: The integration domain is divided into triangle areas. In order to estimate whether the integral, I_0, (left inset) is converged or not, each triangle is divided into four smaller triangles with integral values I_1, I_2, I_3, I_4 (right inset).

- *TransmissionSpectrum*
- *SpinTransferTorque*

AdaptiveHistoryRestart

class AdaptiveHistoryRestart(*effective_rank_fraction=None*, *rank_tolerance=None*)

Class for constructing an adaptive history restart option for the *PulayMixer*. A restart of the mixer history is triggered when the rank of the Gram matrix of the residual vectors is lower than a given fraction of the history size.

A restart can be beneficial to mitigate error stagnation, which can be sometimes been observed in the SCF convergence. The rank of the Gram matrix is used as decision criteria, since it can be seen as a measure of quality of the previous SCF history steps in determining the next guess.

Parameters

- **effective_rank_fraction** (*float of range [0; 1]*) – A tolerance level which determines when an SCF history should be reset/restarted. In the default case, if the effective rank is 50% less than the number of history steps then restart. A value of 0 implies higher chance of restart, and a value of 1 is equivalent to selecting no restart. **Default:** `0.5`
- **rank_tolerance** (*float*) – The tolerance used to determine whether two vectors are linearly dependent. It must be non-negative. **Default:** `1e-6`

effectiveRankFraction()

Returns
The effective rank fraction.

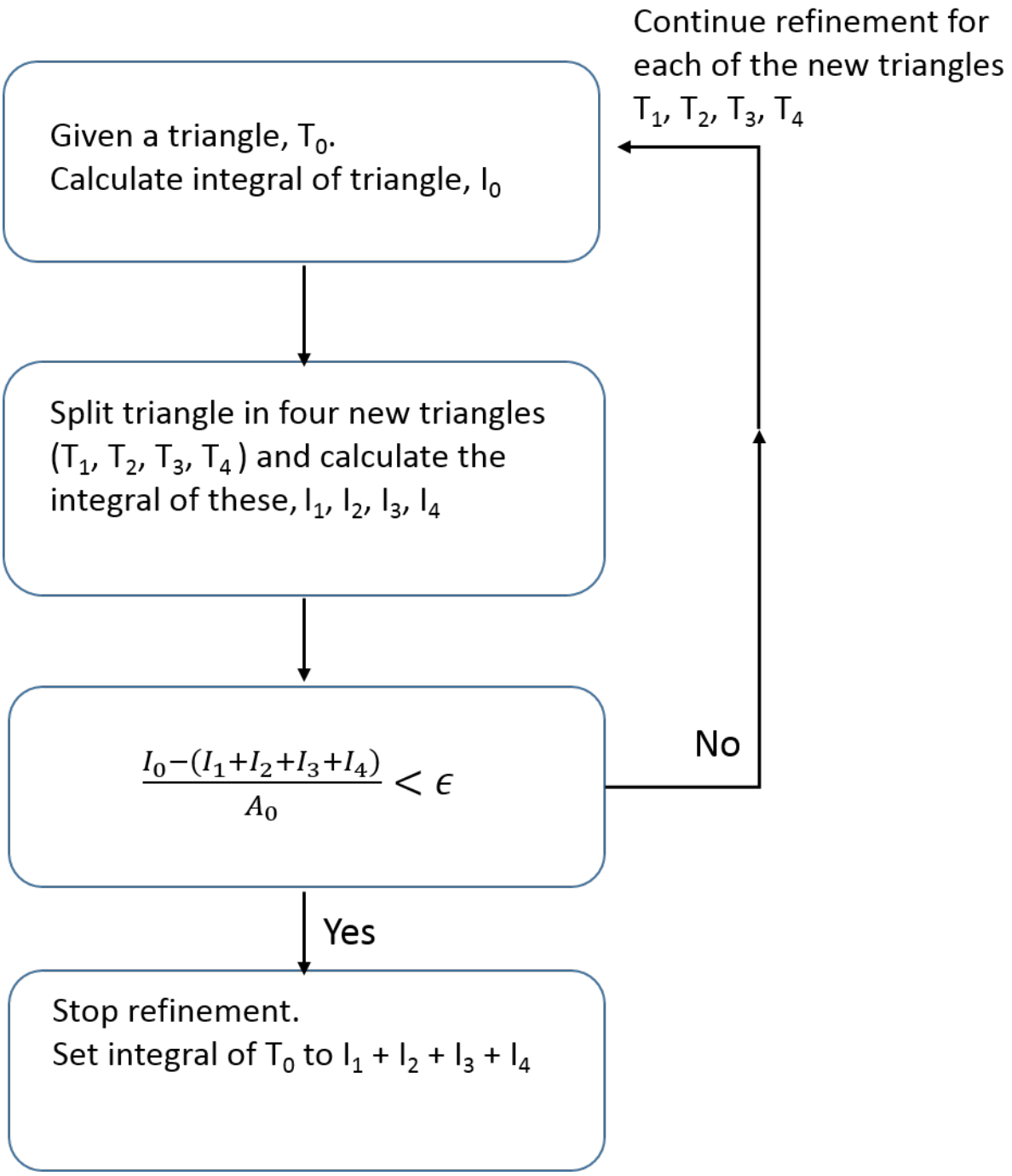


Fig. 4.5: Schematic drawing of the adaptive algorithm. If the error is larger than the tolerance, ϵ, the triangle is divided into four smaller triangles.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

rankTolerance()

Returns
The rank tolerance.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

AdaptiveKineticMonteCarlo

class AdaptiveKineticMonteCarlo(*markov_chain*, *kmc_temperature*, *calculator*, *kmc=None*, *superbasin_threshold=None*, *saddle_offset_magnitude=None*, *saddle_search_method=None*, *optimize_geometry_parameters=None*, *htst_parameters=None*, *confidence=None*, *filename_prefix=None*, *write_log=None*, *write_searches=None*, *write_markov_chain=None*, *write_kmc=None*)

Class for representing AdaptiveKineticMonteCarlo simulations.

Parameters

•**markov_chain** (*MarkovChain*) – The Markov chain.

•**kmc_temperature** (*PhysicalQuantity* of type temperature) – The temperature of the kinetic Monte Carlo simulation.

•**calculator** (*Calculator*) – The calculator object that should be attached to the configuration.

•**kmc** (*KineticMonteCarlo*) – Optional, existing kinetic Monte Carlo simulation, useful for resuming a calculation. **Default:** None.

•**superbasin_threshold** (*int*) – The number of steps between two states before they are merged together and treated with the coarse graining algorithm. A value of *None* disables coarse graining. **Default:** None.

•**saddle_offset_magnitude** (*SaddleSearchParameters*) – The parameters for the saddle search.

•**saddle_search_method** (*LanczosSaddleSearch* | ARTnSaddleSearch | MDSaddleSearch) –

•**optimize_geometry_parameters** (*OptimizeGeometryParameters*) –

•**htst_parameters** (*HTSTParameters*) – The parameters used when calculating the reaction rate based on harmonic transition-state-theory (HTST). By default, the prefactors will be approximated by only including atoms that move more than 0.05 Angstrom and the force constants will be calculated using a central finite difference scheme. A fixed prefactor can instead be assumed for all reactions: HTSTParameters(assumed_prefactor=1e12/Second) **Default:** None.

•**confidence** (*float*) – The confidence in the kinetic Monte Carlo rate table for each state in the simulation. This confidence is the probability that the correct kinetic Monte Carlo step was taken. The confidence should be in the range (0, 1). **Default:** 0.99.

- **filename_prefix** (*str*) – The filename prefix for saved files. **Default:** akmc.
- **write_log** (*bool*) – Whether the log that records the outcome for each search should be written. **Default:** True.
- **write_searches** (*bool*) – Whether the objects for each search (e.g. saddle point, NEB, or MD) should be written. **Default:** True.
- **write_markov_chain** (*bool*) – Whether the *MarkovChain* object should be written. **Default:** True.
- **write_kmc** (*bool*) – Whether the *KineticMonteCarlo* object should be written. **Default:** True.

kmc()

Returns
The KineticMonteCarlo object.

Return type
KineticMonteCarlo

log()

Returns
The AKMC log object.

Return type
AKMCLog

markovChain()

Returns
The Markov chain object

Return type
MarkovChain

run(*max_searches*, *max_kmc_steps=None*)

Run the adaptive kinetic Monte Carlo simulation.

Parameters

- **max_searches** (*int*) – The AKMC simulation will terminate after this many saddle searches.
- **max_kmc_steps** (*int*) – The AKMC simulation will terminate after this many kinetic Monte Carlo steps. **Default:** 10000

state()

Returns
The current state id.

Return type
int

Usage Examples

Below are two examples of modelling diffusion using AKMC. The first example models the diffusion of Hydrogen in a Copper lattice using the Lanczos saddle search method to locate new states. In the second example, two platinum atoms are diffusing on a platinum surface This example uses the MD saddle search method to locate new states.

```
# Set up the direction generator for the saddle search.
direction_generator = HypersphereDirection(
    order=Furthest,
    magnitude=(0.5*Angstrom, 1.0*Angstrom),
)

# Set up the LanczosSaddleSearch object. We are interested in the hydrogen atom, so we␣
↪set the
# index_selector to the index of the hydrogen atom.
hydrogen_index = copper.indicesFromTags(['hydrogen'])
saddle_search = LanczosSaddleSearch(
    initial_direction_generator=direction_generator,
    constraints=optimized_configuration.indicesFromTags(['constrain']),
    index_selector=hydrogen_index,
)

akmc = AdaptiveKineticMonteCarlo(
    markov_chain=markov_chain,
    kmc_temperature=300.0 * Kelvin,
    calculator=optimized_configuration.calculator(),
    kmc=kmc,
    saddle_search_method=saddle_search
)

akmc.run(max_searches=10, max_kmc_steps=1000)
```

hydrogen_diffusion_example.py

```
saddle_search = MDSaddleSearch(
    temperature=1000.0*Kelvin,
    constraints=constraints,
)

akmc = AdaptiveKineticMonteCarlo(
    markov_chain=markov_chain,
    kmc_temperature=300.0*Kelvin,
    calculator=bulk_configuration.calculator(),
    kmc=kmc,
    confidence=0.99,
    htst_parameters=htst_parameters,
    filename_prefix='akmc',
    saddle_search_method=saddle_search
)

akmc.run(max_searches=50, max_kmc_steps=10000)
```

akmc.py

Notes

In solid state systems, atoms reside near their equilibrium positions and reactions are rare events. MD simulations are typically limited to nanosecond timescale while room temperature solid-state reactions of interest may occur on the millisecond timescale or longer.

Adaptive kinetic Monte Carlo (AKMC) is a tool for simulating the dynamics of solid-state systems.[1][2] The goal to is automatically construct a kinetic Monte Carlo (KMC) simulation without prior knowledge of the reaction rates. Unlike lattice KMC, the atomic positions are not limited to lattice sites. Instead the geometries are the stable minima of the potential energy surface obtained from geometry optimization.

The reaction rates are calculated using harmonic transition state theory (HTST). See the *HTSTEvent notes* for more details. HTST calculates the reaction rate between stable states (minima) if the saddle point between them is known. Thus in AKMC the main goal (and computational expense) is to locate new stable states and the saddle points between then.

Algorithm Details

There are three different saddle search methods available. The `LanczosSaddleSearch` and `ARTnSaddleSearch` methods are based on the Lanczos and ARTn algorithms respectively. The `MDSaddleSearch` method is based on molecular dynamics simulations. The saddle search methods are used to locate saddle points on the potential energy surface (PES) between two given geometries. If no saddle point is found, a number of different exceptions can be raised, depending on why the search failed.

We recommend using the `LanczosSaddleSearch` method for most systems (which is also the default). As an alternative, the `ARTnSaddleSearch` method can be used.

Once a saddle point has been found a check is made to ensure that the saddle point is *connected* to the initial state. It is possible that the saddle point that was found does not describe a reaction for the initial minima. Two geometry optimization starting from the saddle point (displaced in the position and negative directions of the unstable direction) are performed. If one of the minima is the initial state then the saddle is *connected*. If neither minima is the starting minima then the saddle is *disconnected* and the saddle search ends.

Once a connected saddle point is located for the first time, the HTST prefactor must be calculated for the forward and reverse reactions. The prefactor can either be set to a fixed value for all reactions or it can be estimated. The prefactor calculation is controlled by the `htst_parameters` argument.

The prefactor involves calculating the dynamical matrix at both minima and the saddle point. This is a computationally expensive calculation and for many systems it is unnecessary, because there is a typical value for the prefactor that does not vary much between reactions. This value can be determined by calculating the prefactor for some reactions in the system.

When numerically calculating the prefactor, not all atoms need to be included in the dynamical matrix to get a good result. Instead, only the atoms that move significantly along the reaction pathway (and their nearest neighbors) need to be considered. The `minimium_displacement` parameter stored in the `HTSTParameters` object allows you to specify how much an atom needs to move by to be included in the dynamical matrix.

Note that there is a slight difference in the meaning of the `max_searches` argument depending on which saddle search method used. When using `MDSaddleSearch`, saddle searches are performed each KMC step until either the specified confidence has been reached, or the maximum number of searches has been performed. When using the other two saddle search methods the saddle searches are terminated when `max_searches` have been performed, since in this case there is no concept of confidence.

[1] Graeme Henkelman and Hannes Jónsson. Long time scale kinetic Monte Carlo simulations without lattice approximation and predefined event table. *The Journal of Chemical Physics*, 115(21):9657–9666, 2001. doi:10.1063/1.1415500.

[2] SamuelăT. Chill and Graeme Henkelman. Molecular dynamics saddle search adaptive kinetic Monte Carlo. *The Journal of Chemical Physics*, 140:214110, 2014. doi:10.1063/1.4880721.

Parallel Saddle Searches

AKMC simulations are highly parallelizable since each saddle search is an independent calculation. Thus, many saddle searches can be run in parallel. When multiple MPI processes are used to run an AKMC calculation, QuantumATK automatically determines how many saddle searches can be run in parallel. It is important to note that one process is always used as a master that coordinates the work of the parallel saddle searches. This makes calculating the total number of MPI processes to use a little trickier than normal.

The number of saddle searches that will be run simultaneously is given by:

$$\frac{N_\mathrm{p} - 1}{N_\mathrm{s}}$$

where N_p is the total number of MPI Processes and N_s is the number of processes to use per saddle search (this is defined by the *ParallelParameters* object attached to the calculator that is being used).

Resuming Previous Calculations

The Python script that is generated by the QuantumATK Scripter supports resuming the AKMC simulation. The typical way to run the calculation would be to set the `max_searches` parameter to a value small enough that the script will finishing running in a reasonable period of time (perhaps a day or so for DFT calculators) and then re-running the script as needed to model the dynamics of interest.

When each saddle search is completed all of the files on disk are updated impenitently. This means that it is possible to monitor the progress of the simulation as it progresses.

AKMC With MDSaddleSearch and Active Learning

When running AKMC calculation with the MDSaddleSearch method, one can also utilize active learning (see *ActiveLearningSimulation*) to simulate the dynamics of solid-state systems while training MTPs. While AKMC with active learning can be utilized for different purposes, the main purpose is to populate a training set for an *ActiveLearningSimulation* object with diverse configurations encountered during AKMC molecular dynamics or optimization steps. AKMC with active learning can also be used for production runs but, for this use case, the quality of the results depends on the system at hand as well as the variance and degree of coverage of the initial active learning training set in respect to the actual system (as described by the reference calculator).

As with normal AKMC usage, the initial configuration needs to be preoptimized and have its *TotalEnergy* evaluated before starting the AKMC state searches. When passing an *ActiveLearningSimulation* object to the AKMC object, the AKMC simulation will automatically use the active learning objects methods for MD and optimization. Hence, the *calculator* input is **ignored** and will not be used during the AKMC simulation. Configurations and their associated energies are therefore evaluated via the MTP calculator as defined from the current active learning object/training set state. Here it should be noted that the AKMC simulation does **not** restart in the event that new configurations are added to the training data as is the default for other active learning applications. Instead, the AKMC simulation continues from the latest transition state search but with an updated MTP calculator.

Since the fitted MTP can change over the course of the AKMC simulation, certain pitfalls should be noted - especially if used for production runs. If the approach is mainly used for diversifying MTP training data, the implications on the quality of the AKMC output are not as severe. States found at previous cycles in the AKMC simulation and their corresponding energies might no longer be accurate according to the newer MTP version. Assuming a simulation is started with an *ActiveLearningSimulation* object that was created from a small *TrainingSet* or a training set that does not represent the initially encountered solid state dynamics well, the credibility of the early AKMC results for energies and states is limited. Nevertheless, as the simulation carries on, more and more structures will be added to the training set from other phase-space regions of the solid assuming that the given reference calculator can describe the structures with sufficient precision and level of detail. If the conditions for a such scenario is met in a production run,

it should be considered to either use a larger and more diverse initial training set, or starting a new, separate AKMC simulation using the updated MTP from the beginning. Another significant caveat is that the saddle searches cannot utilize the parallelization scheme described above. In order to ensure consistency between subsequent state transition searches, the AKMC simulation is only allowed to run saddle searches in serial mode when using active learning. In spite of these caveats, AKMC with active learning is a promising approach for speeding up simulations of solid state dynamics in complex systems.

A usage example showing how the method could be utilized is sketched in the following code snippet, where a silicon carbide supercell with an interstitial magnesium atom is modelled. A training set for the active learning object has been created beforehand. The code snippet shows how the setup deviates from a normal AKMC simulation in order to have the initial configuration have a calculator and total energy based on the MTP created from the *ActiveLearningSimulation* object.

```
# First AKMC simulation: Initial configuration setup for an AKMC simulation
# with active learning that should be started from scratch

# Set up configuration
configuration_0 = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Add tags - constraints needed for AKMC
configuration_0.addTags('fixed', [36, 37])

configuration_name_0 = "optimized_configuration_sic_mginterstitial"

configuration_table = Table()
configuration_table.addInstanceColumn(key='configuration', types=BulkConfiguration)
configuration_table.append(configuration_0)

configuration_names_table = Table()
configuration_names_table.addStringColumn(key='configuration_name')
configuration_names_table.append(configuration_name_0)


# %% ActiveLearningOptimizeGeometry

optimized_configuration_table = runActiveLearningOptimizeGeometry(
    active_learning_simulation=active_learning_simulation,
    configurations=configuration_table,
    max_forces=0.01 * eV / Angstrom,
    constraints=[BravaisLatticeConstraint()],
    trajectory_filename='AKMC-AL_SiC_Mg-interstitial.hdf5',
    trajectory_object_id='alopt',
    log_filename_prefix='AKMC-AL_pre-optimize_geometry',
)

nlsave(
    'AKMC-AL_SiC_Mg-interstitial.hdf5', optimized_configuration_table, object_id='alopt_1
↪'
)
```

(continues on next page)

(continued from previous page)

```
training_set_table = active_learning_simulation.trainingSetTable()


# Custom block for extraction the configuration from the table


def custom(table):
    # This custom script supports atkpython syntax
    # and can perform almost any procedure.

    configuration = table[0, 0]
    configuration.update()
    return configuration


configuration = custom(optimized_configuration_table)

nlsave('AKMC-AL_SiC_Mg-interstitial.hdf5', configuration)


# %% TotalEnergy

total_energy = TotalEnergy(configuration=configuration)
nlsave('AKMC-AL_SiC_Mg-interstitial.hdf5', total_energy)
```

After the initial setup, the AKMC simulation is scripted as usual but with an extra parameter for the active learning simulation object.

```
saddle_search_method = MDSaddleSearch(
    temperature=2900.0 * Kelvin,
    saddle_search_parameters=saddle_search_parameters,
    active_learning=active_learning_simulation,
    constraints=configuration.indicesFromTags(['fixed']),
)

akmc = AdaptiveKineticMonteCarlo(
    markov_chain=markov_chain,
    kmc_temperature=300.0 * Kelvin,
    calculator=configuration.calculator(),
    kmc=kmc,
    filename_prefix='akmc-al',
    saddle_search_method=saddle_search_method,
)

akmc.run(max_searches=50, max_kmc_steps=50)
```

The full script is available for download: `AKMC-AL_example.py`

The restarting with AKMC-AL needs to be handled with more care than for regular AKMC where a static calculator is used. Due to the (potential) updates of the MTP, there is no guarantee that one AKMC simulation script run is going to produce results that are consistent with the next run. For production runs, if many new candidates are added during a script run, it may be favorable to use the most recent *ActiveLearningSimulation* object to create a new AKMC simulation (with new Markov Chain and Kinetic Monte Carlo filenames). If the MTP on the other hand is already so

mature that it is rarely updated or if training set updates do not alter predictions significantly, the AKMC simulation can be resumed by loading the saved initial configuration and total energy objects and use them as AKMC inputs instead of preoptimizing the initial configuration and evaluating its total energy again. The restarting syntax is also outlined in the script, but is commented out and should remain so until the first submission of the script has finished running.

AdsorbedSurfaceGenerator

class AdsorbedSurfaceGenerator(*substrate=None*, *search_depth=None*, *distance=None*, *grouping_precision=None*)

A data structure for keeping track of the input needed to create an adsorbed surface.

Parameters

- **substrate** (*BulkConfiguration* | *SurfaceConfiguration*) – Optionally provide a substrate.
- **search_depth** (*PhysicalQuantity* of type length) – The depth below the surface (defined from the highest atom in the configuration) at which atoms are included in defining adsorption sites.
- **distance** (*PhysicalQuantity* of type length) – The distance above the surface at which to place the adsorption sites.
- **grouping_precision** (*float*) – The precision of grouping unique adsorption sites. Higher values result in more adsorption sites. The allowed range is [0, 2].

addAnchorAtom(*molecule_key*, *atom_index*, *site_type=None*, *site_index=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Random'>*)

Add an anchor atom to the molecule type referenced by molecule_key.

Parameters

- **molecule_key** (*str*) – A unique key specifying a molecule type.
- **atom_index** (*int*) – The atom index of the anchor atom.
- **site_type** (*str*) – The site type that the anchor atom should occupy.
- **site_index** (*Random | int*) – A site_index from the given site type that this anchor atom should occupy.

addMolecule(*molecule_key*, *molecule*, *site_type=None*, *distribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Random'>*, *count=0*, *occupancy=0.0*, *height=PhysicalQuantity(2.0, Ang)*, *orientation=0*, *anchor_atom_index=0*, *gridcell_repetitions=(1, 1)*, *gridcell_center=False*)

Add a molecule type to the surface.

Parameters

- **molecule_key** (*str*) – A unique key specifying a molecule type.
- **molecule** (*MoleculeConfiguration*) – A molecule to be added to the surface.
- **site_type** (*str*) – The site type that the molecules main anchor atom should occupy.
- **distribution** (*int | Random | Quasirandom | SurfaceGrid*) – The distribution for the molecule, which is also the distribution of the main anchor atom. Should be a site index for the given site_type or a flag.
- **count** (*int*) – The number of molecules of this type.
- **occupancy** (*float*) – The occupancy of this molecule type on the main_anchor_atom_site_type. The allowed range is from 0.0 (a clean surface) to 1.0 (maximum site occupation).
- **height** (*PhysicalQuantity* of type length) – The height of the molecules of this type.
- **orientation** (*ORIENTATION | ORIENTATIONS*) – The orientation for this molecule type.

•**anchor_atom_index** (*int*) – The atom index for the main anchor atom.

•**gridcell_repetitions** (*tuple of two integers*) – If distributing to SurfaceGrid, the repetitions of the grid.

•**gridcell_center** (*bool*) – If distributing to SurfaceGrid, whether to use the center of grid cells.

adsorptionSiteParameters()

Returns
The parameters used to find the current adsorption sites as a tuple of the search_depth, distance and grouping_precision, in that order.

Return type
tuple

adsorptionSiteTypes()

Returns
The names of all available adsorption site types.

Return type
tuple

changeAnchorAtom(*molecule_key*, *index*, *atom_index=None*, *site_type=None*, *site_index=None*)

Change an anchor atom in the molecule type referenced by molecule_key.

Parameters

•**molecule_key** (*str*) – A unique key specifying a molecule type.

•**index** (*int*) – The index of the anchor atom to change.

•**atom_index** (*int*) – The atom index of the anchor atom.

•**site_type** (*str*) – The site type that the anchor atom should occupy.

•**site_index** (*Random | SurfaceGrid | Quasirandom | int*) – A distribution flag or a site_index from the given site type that this anchor atom should occupy.

changeMolecule(*molecule_key*, *new_molecule_key=None*, *molecule=None*, *site_type=None*, *distribution=None*, *count=None*, *occupancy=None*, *height=None*, *orientation=None*, *anchor_atom_index=None*, *gridcell_repetitions=None*, *gridcell_center=None*)

Change a molecule type.

Parameters
molecule_key (*str*) – A unique key specifying a molecule type.

:param new_molecule_key:A new molecule key. :type new_molecule_key:str

Parameters

•**molecule** (*MoleculeConfiguration*) – The molecule being changed.

•**site_type** (*str*) – The site type that the molecules main anchor atom should occupy.

•**distribution** (*int | Random | Quasirandom | SurfaceGrid*) – The distribution for the molecule, which is also the distribution of the main anchor atom. This should be a site index for the given site_type or a flag.

•**count** (*int*) – The number of molecules of this type.

•**occupancy** (*float*) – The occupancy of this molecule type on the main_anchor_atom_site_type. Range: [0.0, 1.0].

•**height** (*PhysicalQuantity* of type length) – The height of the molecules of this type.

•**orientation** (*ORIENTATION | ORIENTATIONS*) – The orientation for this molecule type.

•**anchor_atom_index** (*int*) – The atom index for the main anchor atom.

•**gridcell_repetitions** (*tuple of two integers*) – If distributing to SurfaceGrid, the repetitions of the grid.

•**gridcell_center** (*bool*) – If distributing to SurfaceGrid, whether to use the center of grid cells.

copyMoleculeTypes(*other*, *molecule_key=None*)

Copy molecule types from another instance of AdsorbedSurfaceGenerator into this instance. If molecule_key is given, a copy of a single molecule type corresponding to the molecule_key is made, else all molecule types are copied. Note that this does not copy the site_type and distribution for any anchor atoms, as they may not match the adsorption sites in this instance, and should be set explicitly after copying.

Parameters

•**other** (AdsorbedSurfaceGenerator) – The AdsorbedSurfaceGenerator to copy from.

•**molecule_key** (*str*) – A specific molecule_key.

findAdsorptionSites(*search_depth=PhysicalQuantity(0.9, Ang)*, *distance=PhysicalQuantity(2.0, Ang)*, *grouping_precision=1.0*)

Find adsorption sites by defining a volume with the dimensions cell vector A, cell vector B and search_depth. All atoms within this volume are assumed to constitute the substrate surface and will be used to generate adsorption sites. These sites are then placed at the distance from the substrate surface specified by the distance argument.

Parameters

•**search_depth** (*PhysicalQuantity* of type length) – The depth below the surface (defined from the highest atom in the configuration) at which atoms are included in defining adsorption sites.

•**distance** (*PhysicalQuantity* of type length) – The distance above the surface at which to place the adsorption sites.

•**grouping_precision** (*float*) – The precision of grouping unique adsorption sites. Higher values result in more adsorption sites. The allowed range is [0, 2].

Returns
The adsorption sites.

Return type
dict

generateAdsorbedSurface()

Create a configuration consisting of a substrate with adsorbed molecules.

Returns
A configuration consisting of the substrate with adsorbed molecules.

Return type
BulkConfiguration

moleculeKeys()

Returns
The keys of all molecule types added.

Return type
tuple

numberOfAnchorAtoms(*molecule_key*)

Get the number of anchor atoms on the molecule type referenced by molecule_key.

Parameters
molecule_key (*str*) – The key of the molecule of interest.

Returns
The number of anchor atoms.

Return type
int

`static overlappingMolecules`(*configuration*, *tolerance=PhysicalQuantity(1.0, Ang)*)

Find overlapping molecules. The premise of this method is as follows:

1.Find all pairs of overlapping atoms. These pairs are unique and do not contain mirrored copies, i.e., only one of [0, 1] and [1, 0] will be in the list.

2.Collect molecule tags for relevant overlaps by extracting the tags for each pair of overlapping atoms.

a.Ignore any that involve the substrate.

b.Ignore tags that are were not added by AdsorbedSurfaceGenerator by using the prefix adsmol.

c.Intramolecular overlaps are false positives, so we ignore these as well. These will have the same molecular tag on both atoms.

3.By default, only take the second molecule tag in a pair of overlapping molecules, since only one of them needs to be moved to fix the overlap. If take_both=True, then both tags are taken (This is useful if we need to check if a particular molecule is overlapping any other).

Parameters

•**configuration** (BulkConfiguration) – The adsorbed surface configuration.

•**tolerance** (PhysicalQuantity of type distance) – The intermolecule distance tolerance. **Default:** 1 * Angstrom.

Returns
The set of unique tags for overlapping molecules.

Return type
list

`raiseDistributionError`(*message*)

Reset the site distribution and raise NLValueError.

Parameters
message (*str*) – The error message.

`redistributeOverlappingMolecules`(*configuration*, *tolerance=PhysicalQuantity(1.0, Ang)*)

Try to redistribute overlapping molecules to remove the overlaps.

Note! This only applies to molecules distributed with the Random distribution.

The premise of this method is as follows:

1. Find molecules that need to be moved due to atomic overlaps with other molecules. The substrate is ignored since there are better ways to avoid molecules cutting into the surface.

2. If any, then loop over the found molecules and for each of them: a. Loop over all free sites for the molecules site type b. Place the molecule and check if it overlaps anything. c. If yes, then check the next free site. d. If not, then accept the position and move on to the next molecule.

3. If all free sites were checked and none of them satisfied the distance criteria, the molecule remains at its original adsorption site. .

Parameters

•**configuration** (BulkConfiguration) – The adsorbed surface configuration.

•**tolerance** (PhysicalQuantity of type distance) – The intermolecule distance tolerance. **Default:** 1 * Angstrom.

Returns
The adjusted configuration.

Return type
BulkConfiguration

removeAnchorAtom(*molecule_key*, *index*)

Remove an anchor atom in the molecule given by molecule_key.

Parameters

•**molecule_key** (*str*) – A unique key specifying a molecule type.

•**index** (*int*) – The index of the anchor atom to remove.

removeMolecule(*molecule_key*, *redistribute=False*)

Remove a molecule type.

Parameters
molecule_key (*str*) – A unique key specifying a molecule type.

setSubstrate(*substrate*)

Set a substrate if the current substrate is None. SurfaceConfigurations will be converted into BulkConfiguration. If the given configuration has insufficient vacuum at its surface, vacuum will be added such that the total amount of vacuum becomes 10 Angstrom.

Parameters
substrate (*BulkConfiguration* | *SurfaceConfiguration*) – The substrate.

substrate()

Returns
The configuration used as the substrate, which is a surface slab with vacuum.

Return type
BulkConfiguration

Usage Examples

Example 1. Adsorb water molecules onto the ontop sites of a Pt(100) surface at 50% coverage, anchored by their oxygen atoms and oriented vertically.

```
from example_configurations import substrate, water

# ------------------------------------------------------------
# Create surface with adsorbed molecules.
# ------------------------------------------------------------
# Instantiate the AdsorbedSurfaceGenerator with the substrate and find adsorption sites.
generator = AdsorbedSurfaceGenerator(
    substrate=substrate,
    search_depth=0.9 * Angstrom,
    distance=1.0 * Angstrom,
    grouping_precision=1.0
)

# Add molecules.
```

(continues on next page)

(continued from previous page)

```
generator.addMolecule(
    molecule_key='Water',
    molecule=water,
    count=9,
    orientation=ORIENTATIONS.VERTICAL,
    anchor_atom_index=1,
    site_type='ontop_Platinum',
    distribution=Random
)

# Generate a configuration with the molecules adsorbed to the substrate surface.
configuration = generator.generateAdsorbedSurface()
```

adsorbed_surface_example1.py example_configurations.py

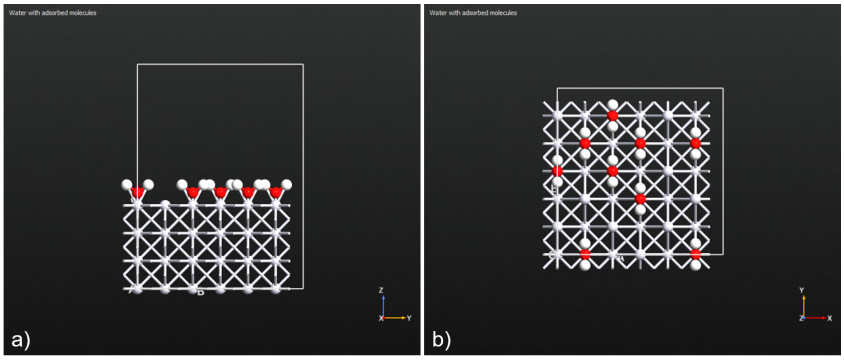


Fig. 4.6: Resulting configuration from Example 1. a) Side view. b) Top view.

Example 2. Adsorb water molecules onto the ontop sites of a Pt(100) surface at 50% coverage, anchored by both of their hydrogen atoms and oriented with the second anchor atom on bridge sites.

```
from␣
↪example_configurations␣
↪import substrate, water

# ----------
↪-------------------------
↪-------------------------
# Create surface with␣
↪adsorbed molecules.
# -------------------------
↪-------------------------
↪----------
# Instantiate the␣
↪AdsorbedSurfaceGenerator␣
↪with the substrate and␣
↪find adsorption sites.
generator =␣
↪AdsorbedSurfaceGenerator(
    substrate=substrate,
    search_depth=0.9 *␣
↪Angstrom,
    distance=1.0 *␣
↪Angstrom,
    grouping_precision=1.0
)

# Add molecules.
generator.addMolecule(
    molecule_key='Water',
    molecule=water,
```

(continues on next page)

(continued from previous page)

```
    count=9,
    ␣
↪orientation=ORIENTATIONS.
↪ANCHORED,
    anchor_atom_index=0,
    site_type='ontop_
↪Platinum',
    distribution=Random
)
generator.addAnchorAtom(
    molecule_key='Water',
    atom_index=2,
    site_type='bridge',
    distribution=Random
)

# Generate a configuration␣
↪with the molecules␣
↪adsorbed to the␣
↪substrate surface.
configuration = generator.
↪generateAdsorbedSurface()
```

`adsorbed_surface_example2.py` `example_configurations.py`

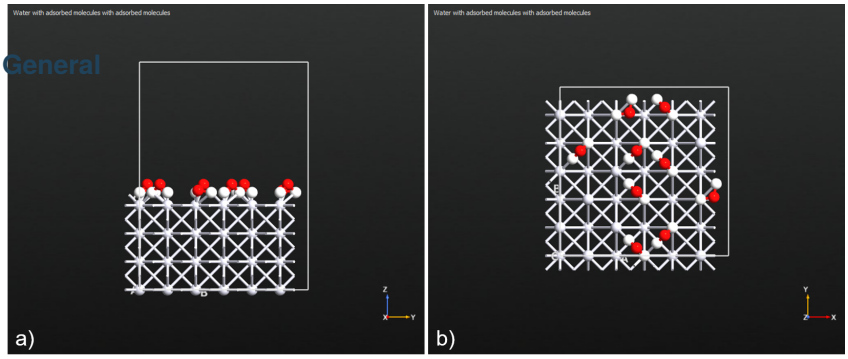


Fig. 4.7: Resulting configuration from Example 2. a) Side view. b) Top view.

The *AdsorbedSurfaceGenerator* class is designed to collect the necessary components for creating a surface with adsorbed molecules, as well as to modify these components and generate the final configuration. The basic steps of using the class are:

1. Add a substrate.
2. Calculate adsorption sites for the substrate.
3. Add molecule types and define how they should be distributed onto the substrate.
4. Optionally add additional anchor atoms.
5. Call `generateAdsorbedSurface` to create the final configuration.

The class can be instantiated with a substrate configuration and parameters for finding adsorption sites as optional input arguments. This allows for a quick start-up by setting the substrate and calculating the adsorption sites in one go.

Substrate and adsorption sites

The substrate can be set using the `setSubstrate` method. Note that each instance of *AdsorbedSurfaceGenerator* can only set a substrate once. This is to avoid a mismatch between the substrate and adsorption sites, as well as the substrate and anchor atom site types and distributions. If a new substrate should be used, a new instance of *AdsorbedSurfaceGenerator* needs to be created. If the same molecules are to be used on the new substrate, use the `copyMoleculeTypes` method to copy any molecule types from the old instance to the new instance, i.e.

```
new_instance.copyMoleculeTypes(old_instance)
```

to copy all molecule types or

```
new_instance.copyMoleculeTypes(old_instance, 'water')
```

to copy only the molecule type water.

Note that this method does not copy the site_type or distribution from any anchor atoms, so these must be set in the new instance after copying to ensure that they match the new substrate and its adsorption sites. Similarly, if the parameters used to calculate adsorption sites in the old instance can be used again in the new instance, these can be retrieved using the `adsorptionSiteParameters` method, i.e.

```
new_instance.findAdsorptionSites(*old_instance.adsorptionSiteParameters())
```

Adsorption sites can be found using the `findAdsorptionSites` method. A list of basic adsorption site types can be found in **Table 1**.

The volume defined by the search_depth parameter should not contain atoms that are repeated in multiple layers, i.e. atoms should not have the same in-plane coordinates in multiple layers. Otherwise, undesired adsorption sites may be generated inside the substrate. See the figure below for clarification. The option to set the search_depth to an arbitrary value within the substrate thickness is useful for more complex crystal structures where adsorption sites are not all at the same height above the surface. This is especially useful for amorphous substrates, in which the surface volume is arbitrarily defined.

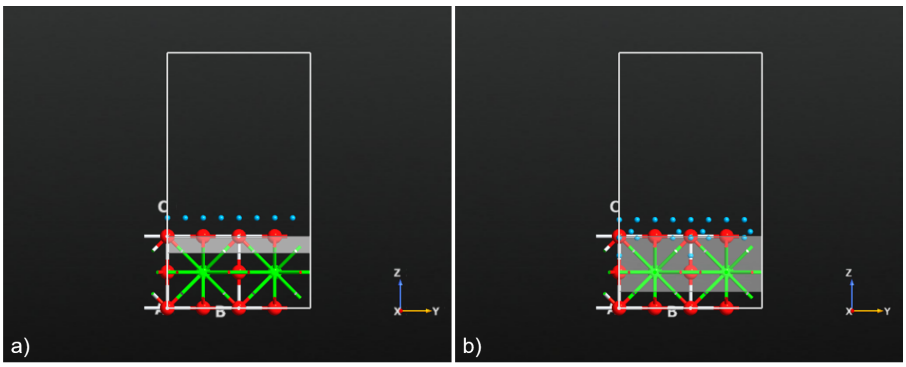


Fig. 4.8: Examples of the volume defined by surface_depth, a) with and b) without repeating atoms in multiple layers. In b), some of the red atoms are repeated in all three layers. Since the volume includes both the first and second layers, adsorption sites can be seen between these layers, inside the substrate.

The grouping_precision parameter can be used to further sort the basic adsorption site types by unique local environments. For instance, the short and long bridge sites on an elemental crystal FCC(110) surface can be sorted into bridge_0 and bridge_1. Similarly, for multi-element substrates, bridge sites between unique pairs of elements can also be sorted into bridge_0, bridge_1, etc.

Table 1. Basic adsorption site types. Additional sorting among each basic site type may be indicated by a trailing underscore and integer.

Site type	Description
on-top_el-ement	A surface site that is 1-fold coordinated. *element* is the name of the element that the site is on top of, e.g., ontop_Oxygen.
bridge	A surface site that is 2-fold coordinated.
hol-low_3fol	A surface site that is 3-fold coordinated.
hol-low_4fol	A surface site that is 4-fold coordinated. Requires at least two layers of atoms in the substrate, otherwise these are labeled as bridge sites.
hcp	A 3-fold coordinated site that corresponds to hexagonal-close-packed stacking. Requires at least three layers of atoms in the substrate to be identified, otherwise these are labeled as hollow_3fold sites.
fcc	A 3-fold coordinated site that corresponds to face-centered-cubic stacking. Requires at least three layers of atoms in the substrate to be identified, otherwise these are labeled as hollow_3fold sites.

Molecules and alignment

Molecule types can be added, modified and removed using the `addMolecule`, `changeMolecule` and `removeMolecule` methods, respectively.

Each molecule type will have one anchor atom by default, which cannot be removed. It is used to define how a molecule type should be distributed onto the substrate surface. Each anchor atom specifies the index of an atom in the molecule, the site type that that atom (equivalently, the molecule type) should occupy and how the molecule should be distributed on these sites.

- For a single molecule (i.e. `count=1`), the distribution may be `Random` or a specific site.
- For multiple molecules, the distribution options are `Random` or `Quasirandom`.
- The `Quasirandom` distribution will attempt to space out the molecules among the sites, and approaches a uniform distribution for sufficiently large numbers of molecules.

The number of molecules of each molecule type can be set either as a coverage or a count. The coverage is calculated in terms of the number of sites that should be occupied for the chosen site type. If both count and coverage are given, the coverage will be used.

The molecule orientation can be given as an option from `ORIENTATIONS`; the available options are summarize in **Table 2**. The orientation can also be given as a set of custom rotation angles, by using the named tuple for which descriptions of the different angles are given in **Table 3**. Example:

```
ORIENTATION(surface_rotation_angle, surface_tilt_angle, molecule_rotation_angle)
```

If `ORIENTATIONS.ANCHORED` is used, it is expected that more than one anchor atom has been added. Additional anchor atoms can be added, modified and removed using the `addAnchorAtom`, `changeAnchorAtom` and `removeAnchorAtom` methods, respectively. If, however, additional anchor atoms have not been given but the orientation is set to `ORIENTATIONS.ANCHORED`, the generator will use the `ORIENTATIONS.VERTICAL` orientation. For additional anchor atoms beyond the first, the site type and distribution of each anchor atom should be set.

- Similar to the first anchor atom, the distribution of additional anchor atoms can be set to a specific site of the chosen site type for a single molecule.
- For both single and multiple molecules, the distribution of additional anchor atoms can also be set to `Random`. In this case, this means the generator will find the site of the chosen site type that comes closest to the that anchor atom.

The same type of molecule can be added more than once. This can be useful when more precise control of the placement of each molecule of a single type is needed. To achieve this, add each molecule with a count of one and distribute it to a specific site type and site.

Table 2. Orientation options.

Orientation	Description
ORIENTATI(VERTICAL	Orient the molecule such that it appears to stand vertically on the substrate. This is the default orientation and corresponds to when all rotation angles are equal to zero. Often, this orientation also coincides with the ORIENTAITONS.ALIGN_I3 orientation.
ORIENTATI(FLAT	Orient the molecule such that it appears to lay flat on the substrate. This corresponds to setting the tilt angle to 90 degrees.
ORIENTATI(ALIGN_I1	Orient the molecule such that the first (smallest) principle moment axis of inertia is aligned to the substrate surface normal.
ORIENTATI(ALIGN_I2	Orient the molecule such that the second principle moment axis of inertia is aligned to the substrate surface normal.
ORIENTATI(ALIGN_I3	Orient the molecule such that the third (largest) principle moment axis of inertia is aligned to the substrate surface normal.
ORIENTATI(ANCHORED	Orient the molecule by designating multiple anchor atoms.

Table 3. Custom orientation angles. All angles are given in degrees.

Angle	Range	Description
surface_rotation_	0 - 360	The angle of the molecule around the substrate surface normal.
surface_tilt_angl	-90 - +90	The angle of the molecule towards the substrate surface, in the direction or the surface_rotation_angle.
molecule_rotation_	0 - 360	The angle of the molecule around the axis defined by the surface_rotation_angle and surface_tilt_angle.

AlgorithmParameters

class AlgorithmParameters(*density_predictor=None*, *density_matrix_method=None*, *store_grids=None*, *store_basis_on_grid=None*, *store_energy_density_matrix=None*, *scf_restart_step_length=None*, *use_symmetries=None*, *reciprocal_space_paw_quantities=None*)

Class for representing the parameters for setting solver parameters.

Some parameter defaults are specific for each calculator, see *HuckelCalculator*, *SlaterKosterCalculator*, *LCAOCalculator*, or *PlaneWaveCalculator*.

Parameters

- **density_predictor** (DensityPredictor | False) – The parameters for the density predictor to be used for the initial guess for density in a DFT calculation. This option is currently only used for LCAOCalculator. **Default:** False
- **density_matrix_method** (*DiagonalizationSolver* | *FeastSolver* | *ChebyshevExpansionSolver* | *PEXSISolver* | PlaneWaveResidualMinimizer | *GeneralizedDavidsonSolver* | *PPCGSolver*) – The method used for obtaining the density matrix. **Default:** *PPCGSolver* for *PlaneWaveCalculator*. *DiagonalizationSolver* for the other calculators.

- **store_grids** (*bool*) – Enable or disable the storage of grids that can be reused in the SCF process. True will increase the memory usage, but reduce the computation time. **Default:** `True`
- **store_basis_on_grid** (`Automatic` | `True` | `False`) – Control the storage of the basis functions on a real space grid. When enabled, increases the memory usage but reduces the computation time. **Default:** `Automatic`
- **store_energy_density_matrix** (bool | `Automatic`) – Enable or disable the computation and storage of the energy density matrix while the density matrix is calculated. When enabled, can slow down the SCF loop but speeds up subsequent evaluations of forces and/or stress. **Default:** Disabled unless forces and/or stress will be evaluated immediately after the self-consistent calculation (`Automatic`).
- **scf_restart_step_length** (*PhysicalQuantity* of type length) – The maximum distance by which an atom can move so that the calculation can be restarted from the previous initial state. **Default:** `0.1 * Ang`
- **use_symmetries** (`Automatic` | `True` | `False`) – Whether the symmetries of the configuration should be used to reduce the number of k-points. If set to `Automatic` the symmetries are disabled for a single k-point and enabled for multiple k-points. **Default:** `True`
- **reciprocal_space_paw_quantities** (*bool*) – If this flag is set to true, the paw quantities will be calculated in reciprocal space, this can only be done when we use periodic boundary conditions. **Default:** `True`

densityMatrixMethod()

Returns
The advanced solver.

Return type
DiagonalizationSolver | *FeastSolver* | *ChebyshevExpansionSolver* | *PEXSISolver* | PlaneWaveResidualMinimizer | *GeneralizedDavidsonSolver* | *PPCGSolver*

densityPredictor()

Returns
The density predictor.

Return type
DensityPredictor | False

nlinfo()

Returns
The nlinfo.

Return type
dict

static noSCFDensityMatrixMethod()

Returns
The solver to use for LCAO when no SCF iterations are done.

Return type
DiagonalizationSolver

scfRestartStepLength()

Returns
The SCF restart step length of this calculator.

Return type
PhysicalQuantity of type length

storeBasisOnGrid()

Returns
True if the storage of the basis functions on a real space grids is enabled

Return type
bool

storeEnergyDensityMatrix()

Returns
True if the simultaneous computation and storage of the energy density matrix during the calculation of the density matrix is enabled.

Return type
bool | Automatic

storeGrids()

Returns
True if the storage of grids is enabled

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

useSymmetries()

Returns
Whether the symmetries of the configuration should be used to reduce the number of k-points.

Return type
bool

Usage Examples

Setup a calculation that uses the Diagonalization Solver.

```
density_matrix_method = DiagonalizationSolver()
algorithm_parameters = AlgorithmParameters(density_matrix_method)
calculator = LCAOCalculator(algorithm_parameters=algorithm_parameters)
```

Setup a calculation that uses the Feast Solver.

```
density_matrix_method = FeastSolver(lambda_min=-10*Hartree, lambda_max=10*Hartree,␣
↪number_of_eigenvalues=10)
algorithm_parameters = AlgorithmParameters(density_matrix_method)
calculator = LCAOCalculator(algorithm_parameters=algorithm_parameters)
```

AllElectronDensity

class AllElectronDensity(*configuration*, *grid_sampling=None*)

A class for calculating the all electron density for a configuration.

Parameters

- **configuration** (*BulkConfiguration*.) – The configuration for which the density should be calculated.
- **grid_sampling** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The density grid sampling. The sampling must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

classmethod defaultDensityMeshCutoff(*max_atomic_number*)

The default density mesh cutoff.

These cutoffs have been obtained by performing calculations on bulk systems like He, Li, Be, C, O, N, O, F, Ne, .. and integrating the all eletron density to give the integrated number of electrons.

The cutoff is the cutoff that gives an error in the integrated number of electron below 0.01

Parameters
max_atomic_number (*int*) – The maximum atomic number.

Returns
The default density mesh cutoff.

Return type
PhysicalQuantity of type energy

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-4}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-3}

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`primitiveVectors`()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

`scale`(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`shape`()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electron density is calculated for, always `Spin.All`.

Return type
`Spin.All`

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

classmethod supportedAtomicNumber(*atomic_number*)

Return true, if the atomic number is supported.

Atomic numbers larger than 20 is not supported because the accumulation of electron density near the atomic position can not easily be described with a cartesian grid.

Parameters
atomic_number (*int*) – The maximum atomic number.

Returns
True, then atomic number supported.

Return type
bool

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the all-electron density and save it to a file:

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(4.348*Angstrom)

# Define elements
elements = [Silicon, Carbon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# PAW Data Set
#---------------------------------------
basis_set = [
    PAWPBESuggested.Carbon,
    PAWPBESuggested.Silicon,
    ]

k_point_sampling = KpointDensity(
    density_a=4.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )
```

(continues on next page)

(continued from previous page)

```
iteration_control_parameters = IterationControlParameters(
    damping_factor=0.2,
    number_of_history_steps=5,
    )

calculator = PlaneWaveCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('SiC_all_electron_density.hdf5', bulk_configuration)

# ------------------------------------------------------------
# All Electron Density
# ------------------------------------------------------------
all_electron_density = AllElectronDensity(
    configuration=bulk_configuration,
    grid_sampling=5000*Hartree,
    )
nlsave('SiC_all_electron_density.hdf5', all_electron_density)
```

SiC_all_electron_density.py

Notes

•This class inherits from the *GridValues* class.

•Returns the all-electron density $n_{ae}(\mathbf{r})$

AlloyConfiguration

class AlloyConfiguration(*bravais_lattice*, *sites*, *cartesian_coordinates=None*, *fractional_coordinates=None*)

The Cartesian coordinates and fractional coordinates cannot be given at the same time. The list arguments containing the sites and coordinates must have the same length.

Parameters

•**bravais_lattice** (BravaisLattice) – A lattice of the bulk configuration.

•**sites** (list of type *AlloySite*) – A sequence containing the site information of the alloy.

•**cartesian_coordinates** (*PhysicalQuantity* of type length | None) – A sequence containing a sequence of atomic coordinates for each site in the configuration. Has the dimensionality nx3. **Default:** None

•**fractional_coordinates** (*array(n, 3) | None*) – A sequence containing sequences of fractional coordinates for each site in the configuration. **Default:** None

bravaisLattice()

Returns
The Bravais lattice

Return type
BravaisLattice

cartesianCoordinates()

Returns
The Cartesian coordinates of the sites.

Return type
PhysicalQuantity of type length

fractionalCoordinates()

Returns
The fractional coordinates of the bulk configuration as a nx3 array.

Return type
array of floats

repeat(*na=1*, *nb=1*, *nc=1*)
Repeat the AlloyConfiguration with the integer values na, nb, and nc along the three primitive unit cell vectors.

Parameters

- **na** (*int*) – The repetition along the a-axis. **Default:** 1
- **nb** (*int*) – The repetition along the b-axis. **Default:** 1
- **nc** (*int*) – The repetition along the c-axis. **Default:** 1

Returns
The repeated bulk system.

Return type
AlloyConfiguration

sample(*rng=None*)
Returns a random realization of the alloy as a bulk configuration.

Parameters
rng (*numpy.random.RandomState | None*) – The random number generator to use. If None then the global rng is used.

Returns
A BulkConfiguration with the elements randomly sampled from the AlloySite specifications.

Return type
BulkConfiguration

sites()

Returns
The list of all sites.

Return type
list of type AlloySite

uniqueSites()

Returns
A list of the unique sites.

Return type
list of type AlloySite

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define an alloy configuration for a 64 atom InGaAs:sub:2 alloy. The sites containing As will not be optimized.

```
# Set up lattice
vector_a = [11.3074, 0.0, 0.0]*Angstrom
vector_b = [0.0, 11.3074, 0.0]*Angstrom
vector_c = [0.0, 0.0, 11.3074]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define site types
# Site type with fixed occupancy, sites of these type will NOT be optimized.
site_a = AlloySite(Arsenic=1.0)

# Site type with fractional occupancy, site of these type will be optimized.
site_b = AlloySite(Gallium=0.5, Indium=0.5)

# Define sites
sites = [
    site_b, site_b, site_b, site_b, site_b, site_b, site_b, site_b,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_b, site_b, site_b, site_b, site_b, site_b, site_b, site_b,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_b, site_b, site_b, site_b, site_b, site_b, site_b, site_b,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_b, site_b, site_b, site_b, site_b, site_b, site_b, site_b,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
]

# Define coordinates
cartesian_coordinates = [[ 0.      ,  0.      ,  0.      ],
                         [ 0.      ,  0.      ,  5.6537  ],
                         [ 0.      ,  5.6537  ,  0.      ],
                         [ 0.      ,  5.6537  ,  5.6537  ],
                         [ 5.6537  ,  0.      ,  0.      ],
                         [ 5.6537  ,  0.      ,  5.6537  ],
                         [ 5.6537  ,  5.6537  ,  0.      ],
                         [ 5.6537  ,  5.6537  ,  5.6537  ],
                         [ 1.413425,  1.413425,  1.413425],
                         [ 1.413425,  1.413425,  7.067125],
                         [ 1.413425,  7.067125,  1.413425],
                         [ 1.413425,  7.067125,  7.067125],
```

(continues on next page)

(continued from previous page)

```
                        [ 7.067125,  1.413425,  1.413425],
                        [ 7.067125,  1.413425,  7.067125],
                        [ 7.067125,  7.067125,  1.413425],
                        [ 7.067125,  7.067125,  7.067125],
                        [ 2.82685 ,  2.82685 ,  0.      ],
                        [ 2.82685 ,  2.82685 ,  5.6537  ],
                        [ 2.82685 ,  8.48055 ,  0.      ],
                        [ 2.82685 ,  8.48055 ,  5.6537  ],
                        [ 8.48055 ,  2.82685 ,  0.      ],
                        [ 8.48055 ,  2.82685 ,  5.6537  ],
                        [ 8.48055 ,  8.48055 ,  0.      ],
                        [ 8.48055 ,  8.48055 ,  5.6537  ],
                        [ 4.240275,  4.240275,  1.413425],
                        [ 4.240275,  4.240275,  7.067125],
                        [ 4.240275,  9.893975,  1.413425],
                        [ 4.240275,  9.893975,  7.067125],
                        [ 9.893975,  4.240275,  1.413425],
                        [ 9.893975,  4.240275,  7.067125],
                        [ 9.893975,  9.893975,  1.413425],
                        [ 9.893975,  9.893975,  7.067125],
                        [ 2.82685 ,  0.      ,  2.82685 ],
                        [ 2.82685 ,  0.      ,  8.48055 ],
                        [ 2.82685 ,  5.6537  ,  2.82685 ],
                        [ 2.82685 ,  5.6537  ,  8.48055 ],
                        [ 8.48055 ,  0.      ,  2.82685 ],
                        [ 8.48055 ,  0.      ,  8.48055 ],
                        [ 8.48055 ,  5.6537  ,  2.82685 ],
                        [ 8.48055 ,  5.6537  ,  8.48055 ],
                        [ 4.240275,  1.413425,  4.240275],
                        [ 4.240275,  1.413425,  9.893975],
                        [ 4.240275,  7.067125,  4.240275],
                        [ 4.240275,  7.067125,  9.893975],
                        [ 9.893975,  1.413425,  4.240275],
                        [ 9.893975,  1.413425,  9.893975],
                        [ 9.893975,  7.067125,  4.240275],
                        [ 9.893975,  7.067125,  9.893975],
                        [ 0.      ,  2.82685 ,  2.82685 ],
                        [ 0.      ,  2.82685 ,  8.48055 ],
                        [ 0.      ,  8.48055 ,  2.82685 ],
                        [ 0.      ,  8.48055 ,  8.48055 ],
                        [ 5.6537  ,  2.82685 ,  2.82685 ],
                        [ 5.6537  ,  2.82685 ,  8.48055 ],
                        [ 5.6537  ,  8.48055 ,  2.82685 ],
                        [ 5.6537  ,  8.48055 ,  8.48055 ],
                        [ 1.413425,  4.240275,  4.240275],
                        [ 1.413425,  4.240275,  9.893975],
                        [ 1.413425,  9.893975,  4.240275],
                        [ 1.413425,  9.893975,  9.893975],
                        [ 7.067125,  4.240275,  4.240275],
                        [ 7.067125,  4.240275,  9.893975],
                        [ 7.067125,  9.893975,  4.240275],
                        [ 7.067125,  9.893975,  9.893975]]*Angstrom
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
alloy_configuration = AlloyConfiguration(
    bravais_lattice=lattice,
    sites=sites,
    cartesian_coordinates=cartesian_coordinates,
)
```

ingaas_alloy.py

Notes

This configuration type can be used with *EvolutionarySQS* to generate an optimized random alloy, based on the special quasi-random structure (SQS) method[1]. To set up an alloy configuration, you need to specify the Bravais lattice and either the fractional or Cartesian coordinates. This is similar to constructing a *BulkConfiguration*, however, instead of the list of elements, you need to give a list of *AlloySite* objects.

If a lattice site has an *AlloySite* containing only one element with an occupancy of 1, the element will remain fixed during the SQS optimization, whereas for *AlloySite* objects with fractional occupancies (i.e. more than one element and occupancies < 1) the occupation of this site will be optimized according the element ratio specified in the *AlloySite* object. Currently only a single type of alloy site with fractional occupancies is supported and an alloy site can only have binary composition, i.e. at maximum be composed of two elements, or one element and one vacancy.

Note that *AlloyConfiguration* cannot be used in actual calculations. Instead, the *BulkConfiguration* produced by *EvolutionarySQS* should be used.

AlloySite

class AlloySite(***site_info*)

Represents a type of site in an alloy. It is an equivalent to the role PeriodicTableElement plays in configurations.

Parameters
site_info (*dict*) – A dictionary that maps element symbols or name strings to fractional occupancies.

elements()

Returns
The elements that belongs to this site.

Return type
list of type PeriodicTableElement | *Vacancy*

fractionalOccupancy(*element*)

Parameters
element (PeriodicTableElement | Vacancy) – The element to find the fractional occupancy of.

Returns
The fractional occupation for a given element.

Return type
float

[1] A.ăvanăde Walle, P.ăTiwary, M.ădeăJong, D.L. Olmsted, M.ăAsta, A.ăDick, D.ăShin, Y.ăWang, L.-Q. Chen, and Z.-K. Liu. Efficient stochastic generation of special quasirandom structures. *Calphad*, 42:13–18, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0364591613000540.

rgbColor()

Returns
The average color of the elements (equal weighting).

Type
numpy array

sample(*num_atoms*, *rng=None*)

For a given total number of atoms return a random permutation of the elements in this site.

Parameters

- **num_atoms** (*int*) – The total number of elements that will be returned.
- **rng** (*numpy.random.RandomState | None*) – The random number generator to use. If None then the global rng is used.

Returns
A randomly permuted list of elements of length num_atoms.

Return type
list of type PeriodicTableElement | *Vacancy*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define an alloy sites with a single element, two elements with fractional occupancies, and an element and a vacancy.

```
# Site type with fixed occupancy, sites of these type will NOT be optimized
# in EvolutionarySQS.
site_a = AlloySite(Arsenic=1.0)

# Site type with fractional occupancy, site of these type will be optimized
# in EvolutionarySQS.
site_b = AlloySite(Gallium=0.5, Indium=0.5)

# Site type with an element and a vacancy.
site_c = AlloySite(Silicon=0.8, Vacancy=0.2)
```

alloy_site.py

Notes

AlloySite types can be used instead of elements to specify sites with fractional occupancies in alloys. These sites can be passed to an *AlloyConfiguration* which can be used in *EvolutionarySQS* to generate an optimized random alloy, based on the special quasi-random structure (SQS) method[1].

The occupancies of an alloy site are specified by passing the element names as an argument names and the fractional occupancies as argument values to the constructor of the *AlloySite* object. In addition to elements, vacancies can be given to the constructor, as well, using `Vacancy` as argument name.

[1] A.ăvanăde Walle, P.ăTiwary, M.ădeăJong, D.L. Olmsted, M.ăAsta, A.ăDick, D.ăShin, Y.ăWang, L.-Q. Chen, and Z.-K. Liu. Efficient stochastic generation of special quasirandom structures. *Calphad*, 42:13–18, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0364591613000540.

If a lattice site has an *AlloySite* containing only one element with an occupancy of 1, the element will remain fixed during the SQS optimization, whereas for *AlloySite* objects with fractional occupancies (i.e. more than one elements and occupancies < 1) this occupation of this site will be optimized according the element ratio specified in the *AlloySite* object. Currently only a single type of alloy site with fractional is supported and an alloy site can only have binary composition, i.e. at maximum be composed of two elements, or one element and one vacancy.

AlloyTrainingParameters

class AlloyTrainingParameters(*reference_configurations*, *percentages=None*, *new_element=<class 'NL.CommonConcepts.PeriodicTable.Hydrogen'>*, *algorithm=None*, *supercell_repetition=None*, *rattle=None*, *atomic_rattling_amplitudes=None*, *sample_size=None*, *random_seed=None*, *optimize=None*, *optimize_geometry_parameters=None*, *log_filename_prefix=None*, *data_tag=None*)

Class for storing parameters for generating a set of alloy training configurations.

Parameters

- **reference_configurations** (*BulkConfiguration* | sequence of [*BulkConfiguration*]) – One or more bulk reference configurations to be used to generate the training set. This is the unit cell from which a supercell can be defined and additional strain can be applied.
- **percentages** (*float or sequence of floats*) – The percentages the atoms that should be substituted. **Default:** 25.
- **new_element** (*PeriodicTableElement* or None (vacancy)) – The element to change to. **Default:** *Hydrogen*.
- **algorithm** (*FixedFraction | NormalDistribution | None*) – The algorithm to randomly change a percentage of the atoms. NormalDistribution means you assign a number [0,1] from a normal distribution across all atoms and only pick those that have a value below the given percentage. FixedFraction means if you set 25%, you get 25% of the total number of atoms changed. The size of the random sample are the percentage of atoms provided, rounded to nearest integer. **Default:** FixedFraction.
- **supercell_repetition** (*sequence (size 3) of int*) – The supercell to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **rattle** (*bool*) – Switch to turn on rattling of the cell to generate more configurations. **Default:** False.
- **atomic_rattling_amplitudes** (sequence (size 3) of *PhysicalQuantity* of type length) – List of three random displacements (small, medium, large) of the atomic positions. **Default:** `[0.15, 0.275, 0.4] * Angstrom`.
- **sample_size** (*int*) – The number of training configurations to generate. The actual number of returned samples can differ slightly due to rounding. When rattle is not set, the returned number of samples will likely be smaller since duplicates are removed. **Default:** 200.
- **random_seed** (*int*) – The random seed used for generating the displacements. **Default:** Generated automatically.
- **optimize** (bool | *Calculator*) – Switch to turn on optimization of reference configurations using a fast LCAO calculator with SingleZeta basis, KpointDensity(density_a=1.0*Angstrom) and NumericalAccuracyParameters(density_mesh_cutoff=60.0*Hartree). Alternatively a calculator object might be supplied to replace the default one. **Default:** False.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters* or None) – Parameters to be passed to a OptimizeGeometry object. **Default:** None.
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting.

algorithm()

Returns
The algorithm to randomly change a percentage of the atoms.

Return type
FixedFraction | NormalDistribution

atomicRattlingAmplitudes()

Returns
Maximum rattling intensities.

Return type
PhysicalQuantity of type length | None

configurations()

Returns
The list of configurations to be used to generate the training set.

Return type
configuration: *BulkConfiguration*

dataTag()

Returns
The selection tag added to the data in the training set.

Return type
str

logFilenameIdentifier()

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

logFilenamePrefix()

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | LogToStdOut | None

newElement()

Returns
The element to change to.

Return type
PeriodicTable.Element or None (vacancy)

optimize()

Returns
Switch to turn on optimization of interface configurations using the LCAO calculator.

Return type
bool

optimizeGeometryParameters()

Returns
The optimize geometry parameters.

Return type
OptimizeGeometryParameters

percentages()

Returns
The percentages of the atoms that should be changed.

Return type
float or sequence of floats

randomSeed()

Returns
The random seed used for generating the displacements, or None if it should be generated automatically.

Return type
int | None

rattle()

Returns
Switch to turn on rattling of the interfaces to generate more configurations.

Return type
bool | None

referenceConfigurations()

Returns
The list of reference configurations containing the new element.

Return type
list of [*BulkConfiguration*]

sampleSize()

Returns
The number of training configurations for each combination of list parameters.

Return type
int

supercellRepetition()

Returns
The supercell to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
tuple (size 3) of int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Setup of a training set for a copper/silver alloy using *AlloyTrainingParameters*.

Note: The particular force-field used for optimization in below script is only for demonstration purposes and should be replaced by a higher quality method in actual training.

```
# Set up lattice
lattice = FaceCenteredCubic(3.61496*Angstrom)

# Define elements
elements = [Copper]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration_copper = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define the substituting element
substitute = Silver

# Define calculator for pre-optimization calculations.
potentialSet = EAM_AgCu_2009()
calculator_ff = TremoloXCalculator(parameters=potentialSet)

optimize_geometry_parameters = OptimizeGeometryParameters(
    max_forces=0.1*eV/Ang,
    max_step_length=0.2*Ang,
    trajectory_interval=1,
    optimize_cell=True,
    optimizer_method=FIRE(),
    enable_optimization_stop_file=True,
    restart_strategy=NoRestart,
)

training_set = AlloyTrainingParameters(
    reference_configurations=bulk_configuration_copper,
    percentages=[10, 20, 30],
    new_element=substitute,
    algorithm=FixedFraction,
    supercell_repetition=(3, 3, 3),
    rattle=True,
    atomic_rattling_amplitudes=[0.275, 0.4, 0.15] * Angstrom,
    sample_size=60,
    random_seed=667,
    optimize=calculator_ff,
    optimize_geometry_parameters=optimize_geometry_parameters,
```

(continues on next page)

(continued from previous page)

```
)

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = KpointDensity(
    density_a=5.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=100.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=MethfesselPaxton(0.2*eV, 1),
    )

iteration_control_parameters = IterationControlParameters(
    tolerance=5e-05,
    damping_factor=0.3,
    number_of_history_steps=12,
    max_steps=1000,
    non_convergence_behavior=StopCalculation(),
    )

checkpoint_handler = CheckpointHandler(
    time_interval=1000000.0*Hour,
    )
algorithm_parameters = AlgorithmParameters(
    scf_restart_step_length=0.3*Angstrom,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    checkpoint_handler=checkpoint_handler,
    algorithm_parameters=algorithm_parameters,
    )

# Set up non-linear coefficients with optimization.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
   perform_optimization=True,
   energy_only=False,
)

# Set up parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
   basis_size=1000,
   outer_cutoff_radii=4.5*Angstrom,
   mtp_filename='mtp_Cu-Ge_alloy.mtp',
   non_linear_coefficients_parameters=non_linear_coefficients_parameters,
)

# Set up MTP training.
moment_tensor_potential_training = MomentTensorPotentialTraining(
```

(continues on next page)

(continued from previous page)

```
    filename='mtp_study',
    object_id='training',
    training_sets=training_set,
    calculator=calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters,
    train_test_split=0.8,
    random_seed=13345,
    number_of_processes_per_task=8,
    log_filename_prefix='fit_mtp_Cu-Ge_alloy',
)
moment_tensor_potential_training.update()
```

`alloy-training-sets.py`

Notes

The *AlloyTrainingParameters* class can be used to generate training configurations by generating random alloys from a crystalline host materials and a replacement element using the *substitutionalAlloy* API.

The sample_size parameter (approximately) determines the total number of generated configurations.

It is recommended to pre-optimize the automatically generated alloy configurations by specifying a fast calculator. The optimization parameters can be adjusted by passing an *OptimizeGeometryParameters* object. By default the LCAO method with a single zeta basis set is used.

To achieve a more broadly applicable training set, the generated alloy configurations can be randomly rattled. A set of three displacement amplitudes has to be provided which will be ordered smallest to largest. In addition to the positions, also the cell shape and volume are rattled simultaneously using the smallest atomic displacement. The unrattled configurations are always part of the final set of configurations.

Alternatively the *crystalTrainingRandomDisplacements* protocol can be used with the trajectory of optimized alloy configurations.

For the actual training data generation, a *AlloyTrainingParameters* object needs to be passed into a *MomentTensorPotentialTraining* object.

AnalyticalSplit

class AnalyticalSplit(*base_orbital=None*, *split_norm=None*, *base_orbitial=None*)

Class for representing the analytical split of a confined or polarization orbital,

Parameters

- **base_orbital** (*ConfinedOrbital* | *PolarizationOrbital*) – The basis orbital from which this orbital should be split.
- **split_norm** (*float between 0 and 1*) – The input orbital will be split at the radius where the input orbital has this given norm (`1 - split_norm`).
- **base_orbitial** –

angularMomentum()

Returns
The angular momentum.

Return type
int

baseOrbital()

Returns
The base orbital used for generating this split orbital.

Return type
ConfinedOrbital | *PolarizationOrbital*

baseOrbitial()
Deprecated: Use baseOrbital() instead.

Returns
The base orbital used for generating this split orbital.

Return type
ConfinedOrbital | *PolarizationOrbital*

nlinfo()

Returns
The nlinfo.

Return type
dict

splitNorm()

Returns
The split norm.

Return type
float

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Define a *BasisSet* for Hydrogen:

```
hydrogen_1s = ConfinedOrbital(
    principal_quantum_number=1,
    angular_momentum=0,
    radial_cutoff_radius=5.28603678847*Bohr,
    confinement_start_radius=0.8 * 5.28603678847*Bohr,
    additional_charge=0.0,
    confinement_strength=20.000*Hartree*Bohr,
    radial_step_size=0.001*Bohr,
    )

hydrogen_1s_split = AnalyticalSplit(hydrogen_1s, split_norm=0.40)

my_hydrogen_basis = BasisSet(
    element=Hydrogen,
    orbitals=[Hydrogen_1s, hydrogen_1s_split],
```

(continues on next page)

(continued from previous page)

```
    occupations=[0.7 , 0.3],
    pseudopotential=NormConservingPseudoPotential('normconserving/H.LDAPZ.zip'),
    )
```

Notes

The *AnalyticalSplit* orbital (ϕ_l^{split}) is obtained by constructing an analytical orbital that matches the `base_orbital` (ϕ_l^{base}) smoothly at the radius r^{split}. The functional form used for the *AnalyticalSplit* orbital is

$$\phi_l^{\text{split}}(r) = \begin{cases} r^l(a_l - b_l r^2) & \text{if } r < r^{\text{split}} \\ \phi_l^{\text{base}}(r) & \text{if } r \geq r^{\text{split}} \end{cases}$$

The radius r^{split} is determined by specifying the `split_norm` (ΔN) of the `base_orbital`, which is defined by

$$\Delta N = \int_{r^{\text{split}}}^{r^c} r^2 |\phi_l^{\text{base}}(r)|^2 dr$$

Further information about the basis functions can be found in *LCAO basis set*.

AndersonMixer

class AndersonMixer(*mixing_fraction=None*, *regularization_parameter=None*)

Class representing a mixer following the Anderson mixing scheme[1].

Parameters

- **mixing_fraction** (*float*) – The mixing fraction. **Default:** Automatically determined in the first step.
- **regularization_parameter** (*float*) – Regularization parameter for numerical stability. **Default:** 0.01

mixingFraction()

Returns
The fraction of the output to be mixed in with the next input.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

regularizationParameter()

Returns
The stabilization factor to be added to the diagonal of the linear system.

Return type
float

[1] V.ăEyert. A Comparative Study on Methods for Convergence Acceleration of Iterative Vector Sequences. *Journal of Computational Physics*, 124(2):271–285, 1996. doi:10.1006/jcph.1996.0059.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up an Anderson mixer which automatically sets the mixing fraction optimally:

```
# Iteration control parameters with the default Anderson mixer.
mixer=AndersonMixer()
iteration_control_parameters=IterationControlParameters(algorithm=mixer)
```

Set up an Anderson mixer with a user defined mixing fraction.

```
# Iteration control parameters with a custom Anderson mixer.
mixer=AndersonMixer(mixing_fraction=0.2)
iteration_control_parameters=IterationControlParameters(algorithm=mixer)
```

Notes

AndersonMixer implements the Anderson method[Page 2248, 1,2].

In the Anderson mixing scheme the best guess for the next input is given by

$$h^{\mathrm{in}}_{n+1} = \bar{h}^{\mathrm{in}}_{n} + \eta\Delta\left(\bar{h}^{\mathrm{in}}_{n}\right),$$

where η is the mixing fraction, h^{in}_{i} is the input value of the `mixing_variable` at iteration i, and $\Delta(h^{i}) = h^{\mathrm{out}}_{i} - h^{\mathrm{in}}_{i}$ is the residual vector.

$\bar{h}^{\mathrm{in}}_{n}$ is determined using a linear combination of solutions from the previous `number_of_history_steps`:

$$\bar{h}^{\mathrm{in}}_{n} = h^{\mathrm{in}}_{n} + \sum_{j}^{n_{\mathrm{hist}}} \theta^{n}_{j}\left(h^{\mathrm{in}}_{n-j} - h^{\mathrm{in}}_{n}\right)$$

in which the variables θ^{i}_{n} are determined by minimizing the residual of $\bar{h}^{\mathrm{in}}_{n}$.

Angle1Potential

class Angle1Potential(*particleType1*, *particleType2*, *particleType3*, *k*, *theta0*, *rho1*, *rho2*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **k** (*PhysicalQuantity of type energy / angle**2*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter.

[2] DonaldăG. Anderson. Iterative Procedures for Nonlinear Integral Equations. *J. ACM*, 12(4):547–560, October 1965. doi:10.1145/321296.321305.

•**rho1** (*PhysicalQuantity of type length*) – Potential parameter.

•**rho2** (*PhysicalQuantity of type length*) – Potential parameter.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

This potential specifies a three-body potential:

$$E_{ijk}(\theta_{ijk}, r_{ij}, r_{jk}) = \frac{1}{2}k(\theta_{ijk} - \theta_0)^2 \exp[-(r_{ij}/\rho_1 + r_{jk}/\rho_2)]\,,$$

which is often used in core-shell potentials.

AngleCorrection6Potential

class AngleCorrection6Potential(*particleType1*, *particleType2*, *particleType3*, *k0*, *k1*, *k2*, *k3*, *k4*, *k5*, *k6*, *costheta0*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type (central particle when calculating angles)

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**k0** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k1** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k2** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k3** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k4** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k5** (*PhysicalQuantity of type energy*) – Potential parameter.

•**k6** (*PhysicalQuantity of type energy*) – Potential parameter.

•**costheta0** (*float*) – Potential parameter.

•**r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius (distance where the spline-smoothing starts).

•**r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

This potential calculates angle-correction terms of the following kind:

$$E_{ijk}(\theta_{ijk}, r_{ij}, r_{jk}) = \frac{1}{2} f_C(r_{ij}, r_{inner}, r_{cut}) f_C(r_{ik}, r_{inner}, r_{cut}) \sum_{d=0}^{6} k_d (\theta_{ijk} - \theta_0)^d,$$

where f_C is a cosine-like cutoff function, which smoothly switches the potential to zero over a range between r_{inner} and r_{cut}.

AngularDistribution

class AngularDistribution(*md_trajectory*, *cutoff_radius=None*, *start_time=None*, *end_time=None*, *triplet_selection=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the AngularDistribution object.

Parameters

•**md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration the angular-distribution should be calculated for.

•**cutoff_radius** (*PhysicalQuantity* of type length) – Cutoff radius to define the maximum considered bond distance between the central atom and each of its two neighbors. Either a single value for both neighbors or one value for each pair. **Default:** `2.5 * Angstrom`

•**start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`

- **end_time** (*PhysicalQuantity* of type time.) – The end time. **Default:** the last frame time
- **triplet_selection** (list of *PeriodicTableElement* | list of ints | list of str) – Only consider angles between these atoms. The order must be (central_atom, neighbor_1, neighbor_2). The selection can be given by a list of 3 elements i.e. *PeriodicTableElement* or a list of 3 atom indices i.e. ints or by a list of tags i.e. str. **Default:** first element
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

angles()

Return the angles associated with the histogram.

data()

Return the angular distribution histogram.

Usage Examples

Load an *MDTrajectory* containing ethane molecules and calculate the angular distribution for C-C-H angles:

```
md_trajectory = nlread('ethane_md.nc')[-1]

angular_distribution = AngularDistribution(md_trajectory,
                                           cutoff_radius=[2.3*Angstrom, 1.2*Angstrom],
                                           triplet_selection=[Carbon, Carbon, Hydrogen])

# Get the histogram of the angle distribution and the associated angles.
histogram = angular_distribution.data()
angles    = angular_distribution.angles().inUnitsOf(Degrees)

# Plot the data using pylab.
import pylab

pylab.bar(angles, histogram, label='C-C-H angles')
pylab.xlabel('angle (degrees)')
pylab.ylabel('Histogram')
pylab.legend()

pylab.show()
```

angular_distribution.py

Notes

Set the `cutoff_radius` parameter to the maximum bond length that should be considered. A good choice is typically the end of the first peak in the *RadialDistribution* function of the respective elements. If only one value is given, this value is used for both neighbor pairs. As shown in the usage example, you can also specify different cutoff radii for the two neighbor pairs.

Instead of each of the three elements in the `triplet_selection` parameter, you can alternatively specify a list or a tag to select the atoms between which the angular distribution is calculated.

AtomicChemicalPotential

class AtomicChemicalPotential(*element*, *reference_configuration=None*, *internal_energy=None*, *entropy=None*)

A class defining the chemical potential for an atomic species. Used as part of a *ChargedPointDefect* study.

Parameters

- **element** (*PeriodicTableElement*) – The element for which the atomic chemical potential is defined.
- **reference_configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The reference configuration from which the components of the atomic chemical potential (internal energy and entropy) will be calculated if they havent been specified. The configuration is only allowed to consist of one type of atomic species corresponding to `element`. If a *MoleculeConfiguration* is specified, it will be internally stored as a `BulkConfiguration`. **Default:** Determined by the *ChargedPointDefect* study.
- **internal_energy** (*PhysicalQuantity* of type energy) – The internal energy component of the atomic chemical potential. **Default:** To be calculated by the *ChargedPointDefect* study using the `reference_configuration`.
- **entropy** (*PhysicalQuantity* of type entropy) – The entropy component of the atomic chemical potential. **Default:** To be calculated by the *ChargedPointDefect* study using the `reference_configuration`.

element()

Returns
The element for which the atomic chemical potential is defined.

Return type
PeriodicTableElement

entropy()

Returns
The entropy component of the atomic chemical potential, or None if it will be calculated automatically.

Return type
PhysicalQuantity of type entropy | None

internalEnergy()

Returns
The internal energy component of the atomic chemical potential, or None if it will be calculated automatically.

Return type
PhysicalQuantity of type energy | None

referenceConfiguration()

Returns
The reference configuration from which the components of the atomic chemical potential (internal energy and entropy) will be calculated. If None, either the configuration is not needed because the components have been explicitly specified, or it has not yet been determined by the *ChargedPointDefect* study. MoleculeConfiguration are automatically returned as BulkConfiguration

Return type
BulkConfiguration | None

uniqueString()

Return a unique string representing the state of the object.

Notes

See *Atomic Chemical Potentials* for a description of how to use this object as part of a *ChargedPointDefect* study.

AtomicChemicalPotentialList

class AtomicChemicalPotentialList(*material_specifications*, *atomic_chemical_potential_list*, *dynamical_matrix_processes_per_task=None*)

Constructor for the AtomicChemicalPotentialList object. An AtomicChemicalPotentialList ensures that either heat of formation conditions (when all elements present in the material specification pristine configuration are specified) or rich conditions (when only some are specified) are satisfied. If non native elements are included its atomic chemical potentials will be used directly without assuming either rich or heat of formation conditions for non native elements.

Parameters

- **material_specifications** (*MaterialSpecifications*) – Material specifications involved in the calculations. The atomic chemical potentials from this material specifications are ignored. Only the ones explicitly specified in the atomic_chemical_potentials parameter of this constructor are used.
- **atomic_chemical_potentials** – A list of atomic chemical potential objects to work with.

atomicChemicalPotentials()

Returns
The user provided list of chemical potentials

Return type
list of *AtomicChemicalPotential*

calculatedAtomicChemicalPotentials(*temperature=None*)

This function can only be called on updated objects.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature used for calculating entropies **Default:** `300.0 * Kelvin`

Returns
List of atomic chemical potentials with their properties properly calculated. This list can be directly assigned to a MaterialSpecification for further calculations.

Return type
list of *AtomicChemicalPotential*

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

dynamicalMatrixProcessesPerTask()

Returns
The number of processes that will be used to execute each task for DynamicalMatrix study objects.

Return type
int

filename()

Returns
The filename where the study object is stored.

Return type
str

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

materialSpecifications()

Returns
The material specifications associated with the chemical potentials

Return type
MaterialSpecifications

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

tasksFinished()

Returns
how many tasks are finished out of how many

Return type
tuple of int, int

totalEnergyReference()

Function to be called on updated objects only.

Returns
The total energy for the unit cell in the material specifications.

Return type
PhysicalQuantity of type energy.

totalEntropyReference(*temperature=None*)

Function to be called on updated objects only

Parameters
temperature (*PhysicalQuantity* of type temperature) – Temperature for entropy calculation. **Default:** 300K

Returns
The total entropy for the unit cell in the material specifications at the specified temperature. None if not needed (assumed entropies specified).

Return type
PhysicalQuantity of type energy / boltmann_constant | None

uniqueString()

Return a unique string representing the state of the object.

update()

Performs the calculations for the AtomicChemicalPotentialList,

Example

A simple example for a TiN system where Ti-rich conditions are desired is:

```
Ti_rich   = AtomicChemicalPotentialList(TiN,[AtomicChemicalPotential(Titanium)])
Ti_rich.update()
nlprint(Ti_rich)

# update reference material with updated chemical potential - Ti rich condition
TiN_rich = TiN(atomic_chemical_potentials=Ti_rich.calculatedAtomicChemicalPotentials())
```

AtomicCompensationCharge

class AtomicCompensationCharge(*atomic_shifts=None*)

Represents a change in the ion charge of individual atoms, resulting in a different ion electrostatic potential.

Parameters

atomic_shifts (*array of tuples*) – The charge shifts that should be applied to any given atom and/or element. Each charge shift entry should be a tuple with tags and corresponding charge shifts, or a tuple with element and charge shift value. Mixing tuple types is allowed. **Default:** no charge shift for any atom

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Simulate a p-doped silicon crystal and add a compensation charge to the silicon atoms:

```
# Set up configuration.
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[ 0.  ,  0.  ,  0.  ],
                            [ 0.25,  0.25,  0.25]]
    )

# Set a negative compensation charge on each atom to compensate for p-doping.
compensation_charge = AtomicCompensationCharge([[Silicon, -0.0005]])
bulk_configuration.setExternalPotential(compensation_charge)

# Calculator
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 5),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)

bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'X', 'W', 'L', 'G', 'K', 'X', 'U', 'W', 'K', 'L'],
    points_per_segment=20,
    bands_above_fermi_level=All
    )

nlsave('si_pdoped.nc', bandstructure)
```

si_pdoped.py

Notes

The atomic compensation charge controls the charge of the neutral atomic densities used to construct the initial guess for the total density. It is typically used to apply a doping to the system. The total charge of the system is set in the calculator, see for instance the keyword `charge` of the *LCAOCalculator*.

For *SurfaceConfiguration* and *DeviceConfiguration*, the atomic compensation charge of the central region atoms equivalent to the electrode atoms should be similar to the compensation charge of the electrode atoms. For the rest of the atoms, there are no requirements. The atomic compensation charge could for instance be used to model a certain doping distribution of the device.

An atomic compensation charge can be applied to a *MoleculeConfiguration*, *BulkConfiguration*, *SurfaceConfiguration*, and *DeviceConfiguration* via the `setExternalPotential` method.

AtomicShift

class AtomicShift(*atomic_shifts=None*)

Class to represent a potential shift of the orbitals on individual atoms.

Parameters

atomic_shifts (list of tuple (size 2) with elements: int | *PeriodicTableElement*, *PhysicalQuantity* of type energy) – A list of potential shifts to be applied to different atoms/elements. Each entry should be a tuple giving the atoms for which the shift is applied (either by index or by element) and the value of the potential shift. Mixing tuple types is allowed. **Default:** no potential shift for any atom

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Add a periodic external potential to silicon and calculate the band structure:

```
# Define a periodic potential along z
def potentialShift(f, shift):
    """
    :param  f:      Position in fractional coordinates
    :type  f:       array of floats

    :param shift:   Amplitude of the shift with unit energy
    :type shift:    float

    :returns:   The external potential at fractional coordinate position.
    :rtype:     float
    """
    return shift*numpy.cos(2.*numpy.pi*f[2])


# Set up a silicon lattice
configuration = BulkConfiguration(
    SimpleCubic(5.4306*Angstrom),
    [Silicon,]*8,
    fractional_coordinates = [[0.0, 0.0, 0.0], [0.25, 0.25, 0.25],
                              [0.5, 0.5, 0.0], [0.75, 0.75, 0.25],
```

(continues on next page)

(continued from previous page)

```
                                [0.5, 0.0, 0.5], [0.75, 0.25, 0.75],
                                [0.0, 0.5, 0.5], [0.25, 0.75, 0.75]])

# Repeat the structure along z
configuration = configuration.repeat(1,1,3)

# Define a selfconsistent Huckel calculator
calculator = HuckelCalculator(
    basis_set=CerdaHuckelParameters.Silicon_GW_diamond_Basis,
    numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(4, 4, 2)),
    iteration_control_parameters = IterationControlParameters()
    )

# Set a calculator on the configuration
configuration.setCalculator(calculator())

# Perform a loop with 4 different shifts
for shift in [0.0, 1.0, 5.0, 10.0]*eV:

    fractional = configuration.fractionalCoordinates()
    atom_potentials = [(i,potentialShift(f, shift)) for i,f in enumerate(fractional)]

    configuration.setExternalPotential(AtomicShift(atom_potentials))


    # Calculate the bandstructure
    bandstructure = Bandstructure(
        configuration=configuration,
        route=['G', 'Z'],
        )

    nlsave('atomic_shift.nc', bandstructure)
```

atomic_shift.py

Notes

The atomic shift adds a term to the tight-binding Hamiltonian of the form

$$\Delta H_{ij} = \frac{1}{2}\left(V_i + V_j\right) S_{ij},$$

where S_{ij} is the overlap matrix and V_i is the atomic shift of orbital i.

An atomic shift can be applied to a *MoleculeConfiguration*, *BulkConfiguration*, *SurfaceConfiguration*, and *DeviceConfiguration* through the `setExternalPotential` method.

AutomaticNEB

class AutomaticNEB(*neb*, *number_of_intermediate_images*, *simultaneous_images=None*, *optimization_parameters=None*, *spring_constant=None*, *climbing_image=None*, *log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *resolution_ratio=None*, *suppress_warnings=None*)

The automatic NEB optimization algorithm adds images one by one in the largest gap between current images. The NEB segment around the added image is then optimized using the attached calculator. The spring constant of the new segments is calculated to conserve the original spring constant. Thus, the automatic NEB is able to locate the transition state with fewer image optimizations than the traditional NEB.

The name of log and trajectory files is passed through the optimization_parameters keyword. An additional prefix is added to the filenames. The first part of the prefix determines the step in the automatic NEB. This is either the current number of intermediate images or *CI* for the climbing image optimization. The second and third parts of the prefix are the indices of the endpoints of the segment being optimized. For example: *3_1_4_* corresponds to the second step with images 2 and 3 being optimized.

Parameters

- **neb** (*NudgedElasticBand*) – The NEB to optimize. The NEB must have a calculator set. The interpolation algorithm set on the NEB is also used in the automatic optimization. The *class: `~.ImageDependentPairPotential* is recommended for the interpolation. If the NEB has only two images, the initial set of simultaneous images are set all at once. The NEB can not have more intermediate images than simultaneous images at the start of the optimization.
- **number_of_intermediate_images** (*int*) – The total number of intermediate images the NEB should have after the automatic optimization is complete.
- **simultaneous_images** (*int*) – The number of images to optimize simultaneously.
- **optimization_parameters** (*OptimizeGeometryParameters*) – The optimization parameters.
- **spring_constant** (*class: `~.PhysicalQuantity* of type force per length squared) – The spring constant for the new segments. If a single value is provided, it will be used for all new segments. Note that in this case the original distances between images are not conserved. If a list is provided, it must have the same length as the number of segments in the NEB.
- **climbing_image** (*bool*) – Whether to perform a climbing-image optimization around the highest energy image after all intermediate images have been added. **Default:** `False`.
- **log_filename_prefix** (`Automatic` | str | None) – The logging output from each segment will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** `Automatic`.
- **resolution_ratio** (*float*) – The resolution ratio used to switch between geometry and energy gaps. Energy gaps are furthermore weighted to prefer higher absolute energies closer to the transition state. A value of 0.0 means that a new image is always added in the geometric gap. A very large value means that a new image is always added in the energy gap. The value 1.0 is free of any bias. **Default:** `0.5`.
- **suppress_warnings** (*bool*) – Whether to suppress warnings printed during the optimization of the NEB segments. Warnings that will be ignored include: Detection of additional minima, unconverged images, and drift in the forces. In the majority of cases, these warnings are not relevant for the automatic NEB and can be ignored. **Default:** `True`.

climbingImage()

Returns
Whether to perform a climbing-image optimization around the highest energy image after all intermediate images have been added.

Return type
bool

finalNEB()

Returns
The final nudged elastic band configuration.

Return type
NudgedElasticBand

interpolationAlgorithm()

Returns
The interpolation algorithm used in the NEB.

Return type
InterpolationAlgorithm

logFilenamePrefix()

Returns
The prefix for the log and trajectory filenames.

Return type
str

neb()

Returns
The input nudged elastic band configuration.

Return type
NudgedElasticBand

numberOfIntermediateImages()

Returns
The total number of intermediate images the NEB should have after the automatic optimization is complete.

Return type
int

optimizationParameters()

Returns
The optimization parameters.

Return type
OptimizeGeometryParameters

resolutionRatio()

Returns
The resolution ratio used to switch between the geometric and energy gaps.

Return type
float

run()

Run the main loop of the automatic NEB optimization. In each step of the main loop a subsection of the NEB containing the newest image is optimized. Then another image is added to the largest gap in the NEB. The process is repeated until the desired number of intermediate images have been added.

Returns
The optimized NEB trajectory.

Return type
NudgedElasticBand

simultaneousImages()

Returns
The number of images to optimize simultaneously.

Return type
int

springConstant()

Returns
The input spring constant.

Return type
PhysicalQuantity of type force per length squared

springConstants()

Returns
Return the spring constants of the NEB segments.

Return type
PhysicalQuantity

suppressWarnings()

Returns
Whether to suppress warnings printed during the optimization of the NEB segments.

Return type
bool

Notes

The automatic *NudgedElasticBand* (NEB) method[1] is an alternative to a regular NEB optimization *OptimizeNudgedElasticBand*. It aims to combine both a reduction in computational resources and improved stability in the process of finding the transition state. It achieves this by converging the path segment by segment, rather than all at once. Finally, the region around the transition state is refined to ensure the highest accuracy. If one is interested in converging the the full minimum energy path, the *OptimizeNudgedElasticBand()* function should be used.

Usage Examples

The *AutomaticNEB* has a similar interface as the *OptimizeNudgedElasticBand()* function. However, one key difference is that *AutomaticNEB* is a class and and the simulation is run by calling the *run()* method.

```
optimization_parameters = OptimizeGeometryParameters(
    max_forces=0.02 * eV/Angstrom,
)
auto_neb = AutomaticNEB(
    neb=initial_neb,
    number_of_intermediate_images=5,
```

(continues on next page)

[1] EsbenăL. Kolsbjerg, MichaelăN. Groves, and Bjørk Hammer. An automated nudged elastic band method. *The Journal of Chemical Physics*, 145(9):094107, 09 2016. URL: https://doi.org/10.1063/1.4961868, doi:10.1063/1.4961868.

(continued from previous page)

```
    simultaneous_images=2,
    optimization_parameters=optimization_parameters,
    spring_constant=0.5 * eV/Angstrom**2,
    climbing_image=True
)
final_neb = auto_neb.run()
```

BaderCharges

class BaderCharges(*configuration*, *electron_density*, *electron_density_reference=None*)

Class for performing a Bader charge analysis.

Parameters

- **configuration** (AtomicConfiguration) – The configuration to perform the charge analysis on.
- **electron_density** (*AllElectronDensity*) – The electron density of the configuration.
- **electron_density_reference** (*ElectronDensity*) – If electron_density is actually a valence electron density, then the all electron density can optionally be provided.

class BaderChargesMaximum(*coordinates*, *charge*, *atom*, *distance*)

Create new instance of BaderChargesMaximum(coordinates, charge, atom, distance)

atom

Alias for field number 2

charge

Alias for field number 1

coordinates

Alias for field number 0

count(*value*, */*)

Return number of occurrences of value.

distance

Alias for field number 3

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

atomicCharges()

Return the calculated atomic charges.

Returns
The number of valence electrons for each atom.

Return type
numpy.array

maxima()

Return the calculated bader charge maxima.

Returns
An list of tuples that each contain the coordinates of a charge density maximum as a *PhysicalQuantity* of type length, the charge contained in the Bader volume as a float, the index of atom that the maxima was mapped to, and the distance to that atom as *PhysicalQuantity* of type length.

Return type
list of type tuple

`maximaCoordinates()`

Return the coordinates of the calculated bader charge maxima.

Returns
The coordinates of each maxima in an array of shape (*n*, 3) where *n* is the number of maxima.

Return type
PhysicalQuantity of type length

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

This module requies an all-electron density, which can be obtain from the PlaneWave PAW all-electron density object *AllElectronDensity*

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(4.348*Angstrom)

# Define elements
elements = [Silicon, Carbon]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#--------------------------------------
# PAW Data Set
#--------------------------------------
basis_set = [
    PAWPBESuggested.Carbon,
    PAWPBESuggested.Silicon,
    ]

k_point_sampling = KpointDensity(
    density_a=4.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

iteration_control_parameters = IterationControlParameters(
    damping_factor=0.2,
    number_of_history_steps=5,
    )

calculator = PlaneWaveCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('SiC_bader.hdf5', bulk_configuration)

# ------------------------------------------------------------
# All Electron Density
# ------------------------------------------------------------
all_electron_density = AllElectronDensity(
    configuration=bulk_configuration,
    grid_sampling=5000*Hartree,
    )
```

(continues on next page)

(continued from previous page)

```
nlsave('SiC_bader.hdf5', all_electron_density)

# ------------------------------------------------------------
# Bader Charges
# ------------------------------------------------------------
bader = BaderCharges(
    configuration=bulk_configuration,
    electron_density=all_electron_density,
    )
nlsave('SiC_bader.hdf5', bader)

# Print Bader object
nlprint(bader)
```

`SiC_bader.py`

A similar calculation can be performed using the LCAO-version of the PAW calculator: `SiC_bader_pawlcao.py`

In order for the Bader analysis to be accurate, the all-electron density needs to be sampled on a fine grid due to the rapid variations close to the nuclei. Therefore the all-electron density is constructed with a `grid_sampling` argument. Heavy elements with high atomic numbers require, in general, a finer grid sampling than lighter elements.

A Bader charge object can also be constructed though the **Bader Analysis** plugin on the **LabFloor**. In order to enable the plugin, select a configuration and an all-electron density. Optionally, when used to analyze VASP output files, the electron density files named `AECCAR0` and `AECCAR2` should also be selected. See the notes below for more information.

Notes

This class currently implements the on-grid Bader analysis algorithm developed by Henkelman, et al.[1].

Bader charge analysis can only work properly when an all-electron density is available. This is because the analysis assumes that the charge density maxima correspond to the coordinates of the atomic nuclei, however, pseudo-potential methods remove charge near the nuclei. This means that this type of analysis is currently only useful with the PAW (PlaneWave or LCAO) all-electron density object and VASP (because it supports writing an all-electron density) calculators in QuantumATK.

If you want to perform a Bader charge analysis using VASP, it is important to be aware that the `CHGCAR` files only contain the valence electron density. In order to write out the extra information that is needed, you must add `LAECHG=.TRUE.` to the `INCAR` file. This will cause the atomic core electron density to be written to the file `AECCAR0` and the non-pseudoized valence electron density (i.e. the density that can be added to the core electron density) is written to the file `AECCAR2`.

Bader calculations should be converged with respect the the electron density grid spacing. This means that, in VASP, you should increase the values of `NEGXF`, `NEGYF`, and `NEGYZ` in order to increase the number of grid points in the `CHGCAR` and `AECCAR` files.

[1] G.ăHenkelman, A.ăArnaldsson, and H.ăJonsson. A fast and robust algorithm for Bader decomposition of charge density. *Comput. Mat. Sci.*, 36(3):354–360, 2006. doi:10.1016/j.commatsci.2005.04.010.

Bandstructure

class Bandstructure(*configuration*, *route=None*, *points_per_segment=None*, *kpoints=None*, *bands_above_fermi_level=None*, *projection_list=None*, *processes_per_kpoint=None*, *method=None*, *interpolation_method=None*, *diagonalization_method=None*)

Analysis class for calculating the bandstructure for a bulk configuration.

Parameters

•**configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the bandstructure.

•**route** (*list of str*) – The route to take through the Brillouin-zone as a list of symmetry points of the unit cell, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `kpoints`. **Default:** Unit-cell dependent route.

•**points_per_segment** (*Positive int*) – The number of points per segment of the route. This option is mutually exclusive to `kpoints`. **Default:** `20`.

•**kpoints** (*list of lists of floats*) – A list of 3-dimensional fractional k-points at which to calculate the energies of the bands e.g. `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. The shape is $(K, 3)$ where K is the number of k-points. This option is mutually exclusive to `route`, and `points_per_segment`. **Default:** Unit-cell dependent route.

•**bands_above_fermi_level** – **Deprecated:** from v2023.09, see `diagonalization_method` and *FullDiagonalizationSolver* instead.

•**projection_list** (*ProjectionList*) – A projection list object defining a projection. **Default:** If no projection list is specified all orbitals will be used.

•**processes_per_kpoint** (int | `None`) – The number of processes to use per kpoint. **Default:** For *PlaneWaveCalculator*, the default is the same number of processes per k-point as used in the SCF calculation. Otherwise, the default is `1`.

•**method** – **Deprecated:** from v2023.09, use `interpolation_method` parameter instead.

•**interpolation_method** (`None` | `Full` | *KDotPExpansion1D* | *KDotPExpansion3D*) – The method used for the bandstructure calculation.

Default is to perform an exact eigenvalue solution for each k-point.

Alternatively, the k.p expansion method can be use to interpolate the eigenvalues at the requested k-points using the exact eigensolution computed for a small subset of the k-points. There are two modes for the k.p method. 1) Using *KDotPExpansion1D*, for each bandstructure path segment, a few exact eigensolution are calculated and the eigenvalues of the remaining k-points are interpolated using a 1D correction term. 2) Using *KDotPExpansion3D*, the eigenvalues for the requested k-points are interpolated from exact solution on a 3D grid. The bands, grid and wave functions of a previous ground state computation can be used in this approach.

Default: `Full` meaning that all k-points will be calculated exactly.

•**diagonalization_method** – Method used for diagonalizing the hamiltonian.

This parameter allows to choose between a full diagonalization solver and an iterative subspace solver. The full diagonalization solver evaluates all bands from the lowest energy one to a given number of bands above fermi level. The iterative subspace solver allows to evaluate a given number of bands around fermi level, or around an energy of choice.

The full diagonalization solver is more robust, but can be proibitively expensive for very large systems.

The iterative solver can deal with very large systems (tens of thousands atoms and beyond) and greatly outperforms when calculating a small number of eigenvalues, but it is also inherently less robust.

Note: the exact method used when selecting *FullDiagonalizationSolver* is defined by the calculator (see *AlgorithmParameters*). *IterativeDiagonalizationSolver* is not supported for *PlaneWaveCalculator*

Default: `FullDiagonalizationSolver`

Type
FullDiagonalizationSolver | *IterativeDiagonalizationSolver*

conductionBandEdge()

Calculate the conduction band edge.

Returns
The conduction band edge.

Return type
PhysicalQuantity of type energy

directBandGap()

Calculate the direct band gap.

Returns
The direct band gap.

Return type
PhysicalQuantity of type energy

energyZero()

The energy zero is equal to the Fermi level. For fixed spin moment calculations it is the average of the Fermi level for spin up and spin down.

Returns
The energy zero.

Return type
`PhysicalQuantity` of type energy

evaluate(*spin=None*)

Return the bandstructure for a given spin.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the bandstructure should be returned for. Must be either `Spin.Up` or `Spin.Down`, or for noncollinear calculations `Spin.All`. **Default:** `Spin.All`

Returns
The eigenvalues for the specified `spin`. The shape is (K, B) or (S, K, B) where S is the number of spins, K is the number of k-points, and B is the number of bands.

Return type
`PhysicalQuantity` of type energy | list of `PhysicalQuantity` of type energy

fermiLevel(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. Must be either `Spin.Up`, `Spin.Down`, or `Spin.All`. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** `Spin.All`

Returns
The Fermi level in absolute energy.

Return type
`PhysicalQuantity` of type energy

fermiTemperature()

Returns
The Fermi temperature used in this bandstructure.

Return type
PhysicalQuantity of type temperature

indirectBandGap()

Calculate the indirect band gap.

Returns
The indirect band gap.

Return type
PhysicalQuantity of type energy

interpolationMethod()

Returns
The method used for the bandstructure calculation.

Return type
None | Full | *KDotPExpansion1D* | *KDotPExpansion3D*

kpoints()

Returns
The list of 3-dimensional fractional k-points at which the energies of the bands are calculated. The shape is (K, 3) where K is the number of k-points.

Return type
list of lists of floats

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Deprecated: Deprecated. Use interpolationMethod instead.

Returns
The method used for the bandstructure calculation.

Return type
None | Full | *KDotPExpansion1D* | *KDotPExpansion3D*

Deprecated: Use `interpolationMethod()` instead.

nlinfo()

Returns
Structured information about the Bandstructure.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

processesPerKPoint()

Deprecated: from v2019

Returns
The number of processes per k-point used in the calculation.

Return type
int > 0

route()

Returns
The route through the Brillouin-zone as a list of symmetry points of the unit cell.

Return type
list of str

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

valenceBandEdge()

Calculate the valence band edge.

Returns
The valence band edge.

Return type
PhysicalQuantity of type energy

Usage Examples

Calculate the bandstructure of silicon and save it to a file:

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------
lattice = FaceCenteredCubic(5.4306*Angstrom)
elements = [Silicon, Silicon]
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
```

(continues on next page)

(continued from previous page)

```
    )

# -----------------------------------------------------------------
# Calculator
# -----------------------------------------------------------------
basis_set = [CerdaHuckelParameters.Silicon_GW_diamond_Basis]
hamiltonian_parametrization = HuckelHamiltonianParametrization(
    basis_set=basis_set)
k_point_sampling = MonkhorstPackGrid(9,9,9)
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    density_mesh_cutoff=10.0*Hartree,
    )
calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# -----------------------------------------------------------------
# Bandstructure
# -----------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'X'],
    points_per_segment=40,
    bands_above_fermi_level=4
    )

# Read the valence and conduction band edges wrt. the Fermi level
edge_valence = bandstructure.valenceBandEdge()
edge_conduction = bandstructure.conductionBandEdge()
print(edge_valence)
# Read the direct and indirect band gaps
gap_direct   = bandstructure.directBandGap()
gap_indirect = bandstructure.indirectBandGap()
print(gap_indirect)
# Print out results
print('Valence band maximum    = %.2f eV' % edge_valence.inUnitsOf(eV))
print('Conduction band minimum = %.2f eV' % edge_conduction.inUnitsOf(eV))
print('Direct band gap   = %.2f eV' % gap_direct.inUnitsOf(eV))
print('Indirect band gap = %.2f eV' % gap_indirect.inUnitsOf(eV))
```

si_band.py

Notes

To export the data of a *Bandstructure*, use the method *nlprint*.

Note that the k-points are given in units of the reciprocal vectors $\mathbf{k}_A$, $\mathbf{k}_B$, and $\mathbf{k}_C$. E.g. the symmetry point X have the fractional k-point coordinates $(\frac{1}{2}, 0, \frac{1}{2})$ since it is described by $X = \frac{1}{2}\mathbf{k}_A + \frac{1}{2}\mathbf{k}_C$.

A query method for the symmetry points of the Brillouin zone can be found in BravaisLattices_common methods of the BravaisLattices class.

For `bands_above_fermi_level` set to a valid integer N_{ba}, the total number of bands above the Fermi level will be N_{ba} for the spin type *unpolarized* and $2N_{\text{ba}}$ for *polarized* and *noncollinear*. This is because there is one spin channel in the former case and two spin channels in the latter two cases. *Spin-orbit* calculations are categorized as non-collinear calculations and therefore follow the same band selection as *noncollinear*. The case where `bands_above_fermi_level` is 1 is illustrated in Fig. 4.9. Note, that all bands below the Fermi level are always selected.

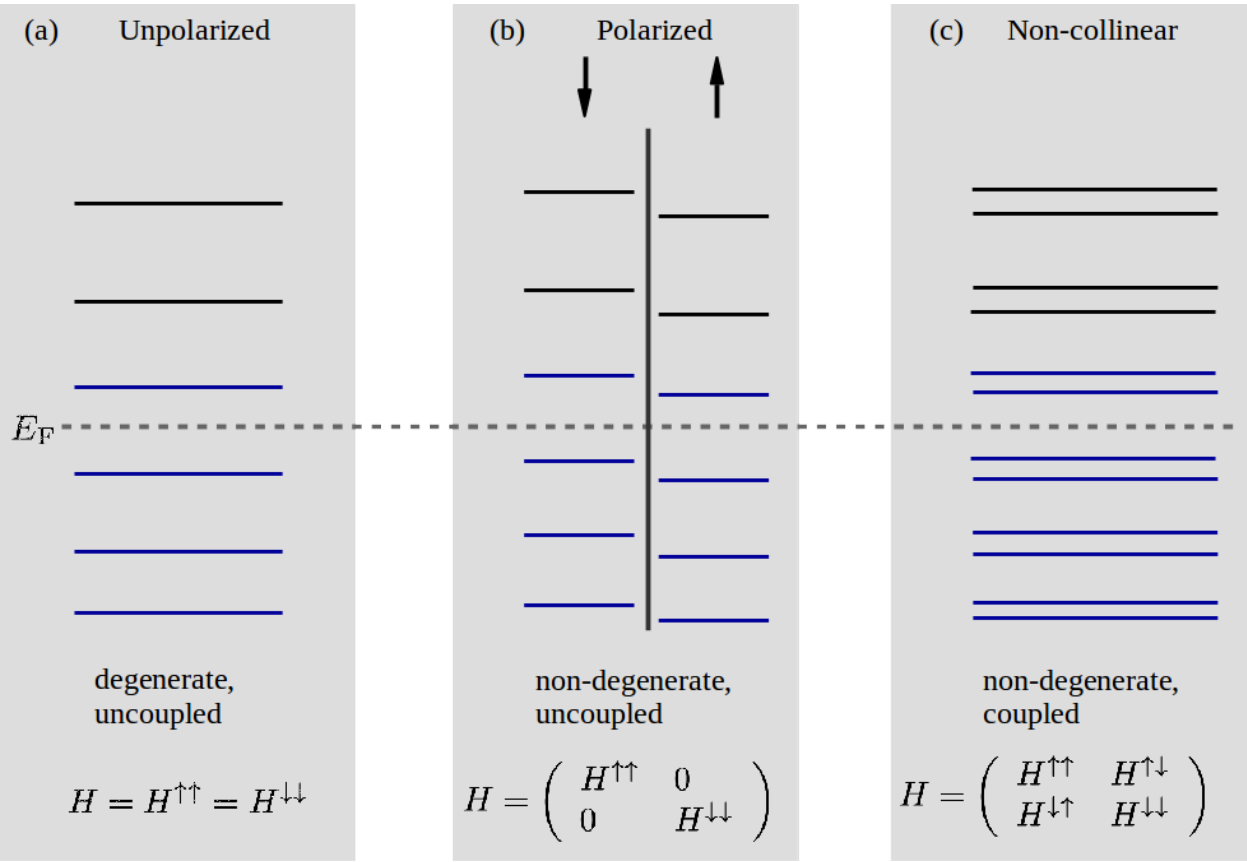


Fig. 4.9: The total number of bands selected (blue lines) by specifying 1 band above the Fermi level for **(a)** an unpolarized, **(b)** a polarized, and **(c)** a non-collinear calculation. The Fermi level E_{F} is shown as a horizontal dashed gray line. The vertical separation line in **(b)** signifies that the bands can be categorized according to spin up or down since there is no coupling. The structure of the Hamiltonian for each of the 3 spin types is shown at the bottom.

BaseChemicalPotential

class BaseChemicalPotential(*element=None*, *reference_temperature=None*)

Create the base class for the chemical potential.

Parameters

- **element** (*PeriodicTableElement*) – The element for which the base class is calculated.
- **reference_temperature** (*PhysicalQuantity* of type temperature) – The temperature for the given reference data.

chemicalPotential(*temperature=None*)

Calculate the chemical potential.

Returns
The chemical potential of the atom calculated from the internal energy and entropy per atom of the specified element in some reference configuration.

Return type
PhysicalQuantity of type energy | None

element()

Returns
The element for which the atomic chemical potential is defined.

Return type
PeriodicTableElement

referenceTemperature()

Returns
The reference temperature for the chemical potential.

Return type
PhysicalQuantity of type temperature.

uniqueString()

Return a unique string representing the state of the object.

Notes

The *BaseChemicalPotential* class is the base class for all of the chemical potential classes. These are classes that define the atomic reference states for elements, and enable the calculation of formation energies and concentrations of defects. The chemical potential definition should reflect the source of the elements in the defect being modeled. This can take into account the surrounding environment of the defect, and whether that environment has appropriate reservoirs for the element. The chemical potential also includes the internal energy of the atom according to the reference calculation method. In density functional theory calculations, this includes the energy required to remove the electrons from the atom.

There are two specific objects that can be used to create a chemical potential. These are:

- *CalculatedChemicalPotential*. In this object the atomic chemical potential is directly calculated from an reference atomic configuration.
- *ReferenceChemicalPotential*. In this object the atomic chemical potential is directly given as input.

The *BaseChemicalPotential* object should not be instantiated. This class is used to refer to any type of atomic chemical potential. As an example, in a Table, the *BaseChemicalPotential* type can be specified for a column type, so that the column can contain both kinds of atomic chemical potentials.

BaseDefectGenerator

class BaseDefectGenerator(*host_configuration*, *symmetry_tolerance=None*)

The base class for an object which describes the possible defects for a given host material.

Parameters

- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters
top_n_configurations (*int*) – Number of most distinct configurations.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

- **point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.
- **keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`

•**select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)

Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)

Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

symmetryTolerance()

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

uniqueDefects()

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

uniqueString()

Return a unique string representing the state of the object.

Notes

The *BaseDefectGenerator* class is the base class for all of the defect generators. These are the classes that generate the symmetrically unique defects of different kinds for a given pristine material. Currently the kinds of defects that can be specified are:

- Vacancy defects, where an atom is removed from the lattice. These can be generated with the *VacancyGenerator* object.
- Substitutional defects, where an atom is exchanged for another. These can be generated with the *Substitutional* object.
- Interstitial defects, where an atom is inserted into a space in the lattice. Interstitial defects at likely lattice positions can be generated using the *InterstitialGenerator* object. This uses Voronoi analysis to find likely positions for interstitial defects.
- Split interstitial defects can be generated using the *SplitInterstitial* object. This defect can be understood as an interstitial defect combined with a displaced lattice atom. The center of the atom pair is positioned at the original lattice cite with the atoms moved in opposite directions along a given axis.
- Pair defects combine a site defect with another point defect. Using pair defects it is possible to create vacancy-interstitial, vacancy-substitution or double substitution defects. These can be generated using the a *DefectPairGenerator*.

The *BaseDefectGenerator* object should not be instantiated. This class is used to refer to any type of defect generator. As an example, in a Table, the *BaseDefectGenerator* type can be specified for a column type, so that the column can contain different types of defect generators.

BasisSet

class BasisSet(*element*, *orbitals*, *occupations*, *hubbard_u=None*, *projector_shift=None*, *filling_method=None*, *pseudopotential=None*, *onsite_spin_orbit_split=None*, *filter_mesh_cutoff=None*, *dft_half_parameters=None*, *gram_schmidt_orthonormalization=True*)

Class for representing the basis set for given element in DFT.

Parameters

- **element** (*PeriodicTableElement*) – The element associated with this basis set.
- **orbitals** (*list*) – The set of orbitals that forms this basis set. Each element in the list should be an instance of *ConfinedOrbital*, *AnalyticalSplit*, *PolarizationOrbital*, *NumericalOrbital*, or *HydrogenOrbital*.
- **occupations** (*list of non-negative float*) – The initial occupation of each basis orbital.
- **hubbard_u** (*PhysicalQuantity* type energy) – The Hubbard U energy for each orbital shell
- **projector_shift** (*PseudoPotentialProjectorShift*) – Pseudopotential projector shift for each angular momentum.
- **filling_method** (`SphericalSymmetric` | `Anisotropic`) – The method used for setting up the initial occupation. **Default:** `SphericalSymmetric`.
- **pseudopotential** (*NormConservingPseudoPotential*) – The pseudopotential to be used for generating this basis set. **Default:** `None`.
- **onsite_spin_orbit_split** (*PhysicalQuantity* of type energy) – Spin-orbit splits for each orbital. **Default:** `numpy.zeros(n_orbitals) * eV` where `n_orbitals` is the number of orbitals.
- **filter_mesh_cutoff** (*PhysicalQuantity* of type energy) – A cutoff for filtering the basis functions. Must not be negative. **Default:** `0.0 * eV`.
- **dft_half_parameters** (*DFTHalfParameters* | `Automatic` | `Disabled`) – The DTF-1/2 parameters used for this basis set in the LCAO calculation. When set to `Automatic` the DFT-1/2 parameters will be populated from a table of optimized parameters. By setting it to `Disabled` DFT-1/2 will not be active for this basis set. This is used to disable

DFT-1/2 for some elements or atomic sites while DFT-1/2 may be active for other atoms in the configuration. **Default:** `Automatic`

- **gram_schmidt_orthonormalization** (*bool*) – Whether or not the basis orbitals should be transformed to an orthonormal basis using the Gram-Schmidt process. **Default:** `True`

angularMomenta()

Return the angular momenta of the basis set.

Returns
The angular momentum (azimuthal quantum number) for each orbital.

Return type
list

dftHalfParameters()

The DFT-1/2 parameters.

Returns
The DFT-1/2 parameters.

Return type
DFTHalfParameters | `Automatic` | `Disabled`

element()

The element for this basis set.

Returns
The atomic element.

Return type
PeriodicTableElement

fillingMethod()

The method using for filling the orbitals.

Returns
The method for filling the orbitals.

Return type
`SphericalSymmetric` | `Anisotropic`

filterMeshCutoff()

The cutoff for the filter function.

Returns
The cutoff.

Return type
PhysicalQuantity of type energy

gramSchmidtOrthonormalization()

Whether the basis set has been orthonormalized.

Returns
True if the basis set is orthonormal.

Return type
bool

hubbardU()

The Hubbard U associated with the orbitals.

Returns
The Hubbard U energies either as a *PhysicalQuantity* with one element for each orbital.

Return type
PhysicalQuantity of type energy.

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfValenceElectrons()

Function to get the number of electrons in total.

Returns
The total number of valence electrons.

Return type
int

occupations()

The occupations associated with the orbitals.

Returns
The occupations of the orbitals given as a list of non-negative floats.

Return type
list

onsiteSpinOrbitSplit()

List of spin-orbit splits per orbital.

Returns
Spin-orbit splittings with one element for each orbital.

Return type
PhysicalQuantity of type energy

orbitalPermutationList()

Get the orbital permutation list.

Returns
The orbital permutation list.

Return type
list

orbitals()

The orbitals that forms this basis set.

Returns
A list of orbital objects.

Return type
list

projectorShift()

A query method for the pseudopotential projector shift.

Returns
The projector shift energies specified for each angular momentum.

Return type
PseudoPotentialProjectorShift

pseudopotential()

The pseudopotential used in generating the basis set.

Returns
The pseudopotential.

Return type
NormConservingPseudoPotential

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Use a **SingleZeta** basis set for Hydrogen and the (default) **DoubleZetaPolarized** basis set for Oxygen:

```
basis_set = [
    LDABasis.Hydrogen_SingleZeta,
    LDABasis.Oxygen_DoubleZetaPolarized
]
calculator = LCAOCalculator(basis_set=basis_set)
```

Define a *BasisSet* for Hydrogen:

```
hydrogen_1s = ConfinedOrbital(
    principal_quantum_number=1,
    angular_momentum=0,
    radial_cutoff_radius=5.28603678847*Bohr,
    confinement_start_radius=0.8 * 5.28603678847*Bohr,
    additional_charge=0.0,
    confinement_strength=20.000*Hartree*Bohr,
    radial_step_size=0.001*Bohr,
    )

hydrogen_1s_split = AnalyticalSplit(hydrogen_1s, split_norm = 0.40)

my_hydrogen_basis = BasisSet(
    element=Hydrogen,
    orbitals=[hydrogen_1s, hydrogen_1s_split],
    occupations=[0.7 , 0.3],
    pseudopotential=NormConservingPseudoPotential('normconserving/H.LDAPZ.zip'),
    )

calculator = LCAOCalculator(basis_set=my_hydrogen_basis)
```

Define a *BasisSet* for Hafnium with a hubbard U term for the 5d orbital, and using anisotropic filling:

```
basis_set = [
    GGABasis.Hafnium_SingleZetaPolarized(
        hubbard_u=[5.8,0.0,0.0]*eV,
        filling_method = Anisotropic)
]
```

Notes

•For further info on the different types of basis orbitals, see *LCAO basis set*.

•With the **SphericalSymmetric** filling method, the orbitals in a shell have the same occupation, while with the **Anisotropic** filling method, the low-m values are fully occupied and the high-m values are empty. See also *XC+U mean-field Hubbard term*.

•To see the default parameters for a basis set it is convenient to use the Script Generator tool in QuantumATK and set the script detail to `Show default`.

•For an example of defining a *BasisSet* with a set of custom *DFTHalfParameters*, see *Usage Examples*. More information on the `Automatic` DFT-1/2 parameters and the DFT-1/2 method in general is given in *DFT-1/2 method*.

BasisSetInitialization

class BasisSetInitialization(*basis_set=None*)

The plane-wave basis set initialization method class.

Parameters
basis_set (*BasisSet*) – The basis set to use for the initialization method. Note that this feature is not enabled for PAW calculations, in which case the argument should be left default. **Default:** An automatically generated minimal basis set.

basisSet()

Returns
The basis set.

Return type
list

BasisSetOptimizer

class BasisSetOptimizer(*bulk_configuration*, *kpoint_sampling=None*, *orbitals_per_shell_map=None*, *radial_cutoff_radius_map=None*, *confinement_start_radius_map=None*, *elements=None*, *energy_cutoff=None*, *confinement_strength=None*, *tolerance=None*, *max_number_of_iterations=None*)

Class that know how to optimize a subset of a complete basis set w.r.t. a PlaneWave calculation.

Parameters

•**bulk_configuration** (*BulkConfiguration*) – A bulk configuration with an updated PlaneWave calculator attached.

•**kpoint_sampling** (*KpointDensity* | *MonkhorstPackGrid*) – The k-point sampling to use for the optimization.

- **orbitals_per_shell_map** (*dict | list*) – A map of type {element : [n0, n1, n2, n3]} where n0, n1, are the number of orbitals with angumar momentum 0, 1, 2, to be used for the complete basis set. If a list of numbers is provided, that will be applied to all elements in the optimization. **Default:** [5, 5, 4, 3] applied to all elements.
- **radial_cutoff_radius_map** (*dict of floats*) – For every element, this defines the radius at which the radial function grid stops.
- **confinement_start_radius_map** (*dict of floats*) – For every element, this defines the radius at which the confining potential starts.
- **elements** (*list of strings*) – The elements for which to optimize.
- **energy_cutoff** (*PhysicalQuantity of type energy.*) – The energy cutoff decides how many plane-wave bands will be included when optimizating the basis set.
- **confinement_strength** (*Physical Quantitity of type energy.*) – The strenght of the confining potential.
- **tolerance** (*float*) – The tolerance determines the value of the spilling for each band at which the basis set is deemed converged.
- **max_number_of_iterations** (*int*) – In the conjugate gradient optimizer for the basis set, the max number of iterations can be set. Beyond this it stops even if it isnt converged.

optimize(*spilling_criterion*, *size_criterion_map*, *print_spilling_per_band=False*)

Run the basis set optimization.

Parameters

- **spilling_criterion** (*float*) – Tolerance for when to stop the optimization. When the maximum spilling is below the spilling criterion, the optimization will stop and return the smallest basis set fulfilling this criterion.
- **size_criterion_map** (*dict of type {element: int}*) – A map of elements and maximum allowed basis set size.
- **print_spilling_per_band** (*bool*) – Boolean controlling if band resolved spilling should be printed. **Default:** False

Returns
List of optimized basis set for each element.

Return type
list of *BasisSet*

optimizeForOrbitalConfiguration(*subspace_sizes_map*)

Optimize the basis set for a given orbital configuration, which is the number of orbitals you want in each shell for each element type.

Parameters
subspace_sizes_map (*dict of list of floats*) – For every element type, a list containing the number of functions you want in the basis set for each angular momemntum channel

printCompleteSpilling()

Outputs the spilling of the complete basis set compared to the plane-wave calculation for the energy cutoff specified.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

The *BasisSetOptimizer* class is used to optimize the basis set for a given configuration. The basis set is optimized by by minimizing the spilling of the LCAO Basis set with respect to a reference plane-wave calculation performed with the same pseudo potential. It was developed to generate high quality basis sets to be used in GW calculations using the *GWCalculator* class.

The following example demonstrates how to optimize the basis set. First we define the configuration:

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates)
```

Then we define a plane wave calculator to perform the reference calculation:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=MonkhorstPackGrid(9,9,9)
)
pw_calculator = PlaneWaveCalculator(
    wave_function_cutoff=50*Hartree,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    checkpoint_handler=NoCheckpointHandler)

bulk_configuration.setCalculator(pw_calculator)
bulk_configuration.update()
```

Then we define the basis set optimizer and optimize the basis set:

```
basis_set_optimizer = BasisSetOptimizer(
    bulk_configuration,
    orbitals_per_shell_map={Silicon: [6,6,5,4]},
    energy_cutoff=10*eV,
    tolerance=1.0e-5)
```

The energy cutoff decides up to what energy above the Fermi level the bands will be included for the spilling optimization. The orbitals per shell map is a dictionary that maps the elements to the number of orbitals per shell included in the complete basis set, out of which the optimal subset will be selected. This input decides how many angular momentum shells will be included in the basis set optimization. In this example the complete basis set will have 6 s-orbitals, 6 p-orbitals, 5 d-orbitals, and 4 f-orbitals.

Finally we optimize the basis set and save the optimized basis set:

```
opt_basis_set = basis_set_optimizer.optimize()
nlsave("opt_basis_set.hdf5", opt_basis_set)
```

doing this will choose the smallest optimal combination of orbitals that meets the tolerance criterium.

To find the optimal basis set with a given size, you can do the following:

```
subspace_sizes_map = {Silicon: [3,3,2,1]}
optimized_parameter_set = optimizer.optimizeForOrbitalConfiguration(
    subspace_sizes_map=subspace_sizes_map
)
```

This will find an optimal basis set with 3 s-orbitals, 3 p-orbitals, 2 d-orbitals, and 1 f-orbitals. To get a basis set object from the output of the optimization, you can do the following:

```
optimized_basis_set = optimizer._constructBasisSetFromParametersMap(
    parameters_map=optimized_parameter_set.parameters_map,
    subsize_sizes_map=subspace_sizes_map)
nlsave("opt_basis_set.hdf5", opt_basis_set)
```

BlochState

class BlochState(*configuration=None*, *quantum_number=None*, *spin=None*, *k_point=None*, *density_mesh_cutoff=None*)

A class for calculating the wave function of a Bloch state.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration for which the Bloch state should be calculated.
- **quantum_number** (*int*) – The quantum number of the desired Bloch state. **Default:** `0`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin to calculate the Bloch state for. **Default:** `Spin.All`
- **k_point** (*list(3) of floats*) – The k-point in fractional coordinates that the Bloch state should be calculated for. **Default:** `[0.0, 0.0, 0.0]`
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative of the wave function in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* of type length) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* of type length) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* of type length) – The Cartesian z coordinate.

•**spin** (Spin.Up | Spin.Down | Spin.All) – The spin component to project on. **Default:** The spin of this Bloch state object.

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Up, Spin.Down) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-3/2}$ Œ length^{-1}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

•**downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.

•**downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.

•**downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`eigenvalue`()

Returns the eigenvalue associated with the calculated eigenstate (as absolute energy).

Returns
The eigenvalue associated with the calculated eigenstate (as absolute energy). For a polarized calculation with `Spin.All`, both `Spin.Up` and `Spin.Down` values are returned

Return type
PhysicalQuantity of type energy | list of *PhysicalQuantity* of type energy.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate the wave function in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* of type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* of type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* of type length) – The Cartesian z coordinate.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component to project on. **Default:** The spin of this Bloch state object.

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Up`, `Spin.Down`) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-3/2}$

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`kPoint`()

Returns
The k-point to calculate the Bloch state for.

Return type
list(3) of floats

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

quantumNumber()

Returns
The quantum number of the desired eigenstate.

Return type
int

quantumNumberReference()

Returns
Whether the quantum number is counted starting from the lowest available state (Absolute), or from the first occupied state (Lumo).

Return type
Absolute``|``Lumo

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the Bloch state is calculated for.

Return type
Spin.Up | Spin.Down | Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin the object was created with. If the spin was Spin.All, Spin.Sum will be used for the projection.

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate and save a Bloch state for a gold FCC crystal:

```
# Define lattice
lattice = FaceCenteredCubic(4.08*Angstrom)

# Define elements
elements = [Gold]

# Define coordinates
coordinates = [[2.0, 2.0, 2.0]]*Angstrom

# Set up configuration
bulk_configuration = BulkConfiguration(
    lattice,
    elements,
    coordinates
    )

# Define a a calculator
bulk_configuration.setCalculator(LCAOCalculator())

# Calculate and save the Bloch state with quantum number 5
bloch_state = BlochState(bulk_configuration, quantum_number=5,
                         k_point=[0.0, 0.5, 0.5])
nlsave('bloch_state.nc', bloch_state)
```

blochstate.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Note: For more advanced options and calculating multiple *BlochState* objects *calculateBlochStates()* is advisable.

BondConstraint

class BondConstraint(*fixed_bonds=None*, *fixed_angles=None*, *inversion_order=4*, *rotation_order=1*)

This constraint fixes the bond lengths of the system in a *MolecularDynamics()* simulation or *OptimizeGeometry()* calculation based on the LINCS method.

Parameters

- **fixed_bonds** (*list*) – The bonds to constrain. These are specified as a list of pairs. If the pair contains two integers, these are taken as the indices of the bond. If the pair contains two strings, these are taken as tags describing two sets of atoms between which the bonds are to be constrained. If the pair contains two elements, these are taken as the elements between which all bonds are to be constrained.
- **fixed_angles** (*list*) – The angles to constrain. These are specified as a list of triples. The angle is constrained by constraining the two bonds and the distance between the end atoms in the angle. In this way small molecules such as water can be constrained to be rigid. If the triple contains to integers, these are taken as the indices of the angle. If the triple contains two strings, these are taken as tags describing three sets of atoms between which the angles are to be

constrained. If the triple contains three elements, these are taken as the elements between which all angles are to be constrained.

- **inversion_order** (*int*) – The number of terms used to approximate the matrix inversion in the LINCS equations. **Default:** 4
- **rotation_order** (*int*) – The number of cycles used in the correction for rotational bond lengthening. **Default:** 1

bondIndices()

Returns
List of indices of the bonds that are constrained

Return type
ndarray

bondLengths()

Returns
List of the constrained bond lengths

Return type
PhysicalQuantity of type length

frozenDegreesOfFreedom(*local_atoms=None*)

The number of constrained degrees of freedom.

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run a molecular dynamics calculation on a PVC polymer with constrained bonds for all hydrogen atoms. This allows using a longer time-step of 2 femtoseconds.

```
method = NVTNoseHoover(
    time_step=2*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=initial_velocity,
)

# Set bond constraint to constrain all C-H bonds
bond_constraint = BondConstraint([[Carbon, Hydrogen]])
com_constraint = FixCenterOfMass()
```

(continues on next page)

(continued from previous page)

```
# With 2fs time steps 5,000 MD steps corresponds to 10ps of simulation time.
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[bond_constraint, com_constraint],
    trajectory_filename='PVC_Bond_Constraint.hdf5',
    steps=5000,
    log_interval=500,
    method=method
)
```

PVC_Bond_Constraint.py PVC_Input.hdf5

Notes

The *BondConstraint* class implements the LINCS bond constraint[1] for both molecular dynamics and geometry optimization simulations. In the LINCS algorithm bonds are held rigid by introducing bond distance constraints g:

$$g_i(\mathbf{r}_n) = |\mathbf{r}_{i_1} - \mathbf{r}_{i_2}| - d_i = 0 \quad i = 1, ...K$$

These are then included in the equations of motion using Lagrange multipliers $\lambda_i(t)$ such that:

$$-\mathbf{M}\frac{d^2\mathbf{r}}{dt^2} = \frac{\partial}{\partial \mathbf{r}}\left(\mathbf{V} - \lambda \cdot \mathbf{g}\right)$$

These equations of motion are solved at each step of the simulation, allowing the changes in the selected bond lengths to be projected out of the atomic coordinates. Bond length changes are also projected out of the velocities, forces and stresses calculated during the simulation.

These bond constraints can be used with atoms at relative distances to each other during a simulation. This can aid in locating transition states or other intermediate configurations. The bond constraints can also be used to remove high frequency motion during molecular dynamics. In these kinds of simulations, the necessary time step of the simulation is often determined by the high frequency bond vibrations. This is especially true for bonds containing light elements such as hydrogen. By constraining these bonds it can be possible to increase the time step used for the molecular dynamics simulation, as those high frequency modes no longer have to be represented in the dynamics.

In *BondConstraint*, the bonds to be constrained are specified in the argument `fixed_bonds`. This takes a list of pairs of either elements, tags or atom indices as integers. For elements and tags, all static bonds between those elements or pairs will be constrained. If atom indices are given, the distance between these atoms will be constrained whether or not they are bonded. The constrained bond length will be taken from the starting configuration in the simulation.

It is also possible to constrain bond angles using the argument `fixed_angles`. This argument takes a list of triples of elements, tags or bond indices. The atoms specified in this triple are then constrained in a rigid triangle, approximating a rigid angle. Regardless of the type in input used, a distance constraint is added between the two end atoms whether or not they are bonded. This type of constraint can be used to create rigid small molecules, such as rigid water.

Warning: The solution to the constraint equations is performed using an approximate matrix inverse. Including angle constraints makes this approximation less valid, and increases the number of terms that need to be included to get a good solution. If angle constraints are included the parameter `inversion_order` may need to be significantly increased to account for this.

[1] Berk Hess, Henk Bekker, Herman J.ăC. Berendsen, and Johannes G. E.ăM. Fraaije. LINCS: A Linear Constraint Solver for Molecular Simulations. *J. Comput. Chem.*, 18:1463, Feb 1997. URL: https://onlinelibrary.wiley.com/doi/10.1002/%28SICI%291096-987X%28199709%2918%3A12%3C1463%3A%3AAID-JCC4%3E3.0.CO%3B2-H, doi:10.1002/(SICI)1096-987X(199709)18:12<1463::AID-JCC4>3.0.CO;2-H.

In the LINCS algorithm there are two iterative parts to the solution of the constraint equation. The first is the solution of a matrix inverse, which is approximated as a series. The number of terms in this series is modified using the argument `inversion_order`. Secondly the LINCS algorithm corrects changes in bond length due to rotation using an iterative solution. The number of iterations is modified using the argument `rotation_order`. By default these have values of 4 and 1 respectively. This is suitable for most calculations with just constrained bonds. Including constrained angles or large number of bonds may require increasing these values to obtain stable constraints.

BornEffectiveCharge

class BornEffectiveCharge(*configuration*, *kpoints_a=None*, *kpoints_b=None*, *kpoints_c=None*, *atomic_displacement=None*, *use_symmetry=None*, *non_selfconsistent_update=None*, *polarization_cartesian_directions=None*, *atom_indices=None*, *processes_per_displacement=None*)

Class for calculating the Born effective charge of a configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration with attached calculator that supports Berry-phase polarization calculations.
- **kpoints_a** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the first direction. **Default:** `MonkhorstPackGrid(n_aa, n_ab, n_ac)`, where *n_aa*, *n_ab*, and *n_ac* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_aa = 2 * na + 1, n_ab = nb, n_ac = nc`.
- **kpoints_b** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the second direction. **Default:** `MonkhorstPackGrid(n_ba, n_bb, n_bc)`, where *n_ba*, *n_bb*, and *n_bc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ba = na, n_bb = 2 * nb + 1, n_bc = nc`.
- **kpoints_c** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the third direction. **Default:** `MonkhorstPackGrid(n_ca, n_cb, n_cc)`, where *n_ca*, *n_cb*, and *n_cc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ca = na, n_cb = nb, n_cc = 2 * nc + 1`.
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01*Angstrom`
- **use_symmetry** (*bool*) – If enabled, only the symmetrically unique atoms are displaced and the remaining born effective charges are calculated using symmetry. **Default:** `True`
- **non_selfconsistent_update** (*bool*) – If True, the displaced configurations are updated non-self-consistently. This leads to a computational speed-up, but the results should be carefully checked against self-consistent calculations. **Default:** `False`
- **polarization_cartesian_directions** (list of `CartesianDirection.{X,Y,Z}`) – Restrict the evaluation of the Polarization to specific Cartesian directions. For example, if `[CartesianDirection.X]` is specified, only the `xx, xy, xz` components of the born effective charge tensor are calculated. This option can be used to speed up the calculation if only such components are required. **Default:** `[CartesianDirection.X, CartesianDirection.Y, CartesianDirection.Z]`
- **atom_indices** (`All` | list of ints.) – The atom indices for which to calculate the Born effective charges. If not `All`, the use of symmetries will be disabled. **Default:** `All` i.e. all atoms in the configuration.
- **processes_per_displacement** (`Automatic` | int) – The number of processes to use per atomic displacement. When set to `Automatic` the number will be determined automatically maximizing the number of displacements calculated in parallel and minimizing the number of idle processes. One may set this number manually in order to reduce the memory requirements for each process. **Default:** `Automatic`

atomIndices()

Returns
The atom indices.

Return type
list of int.

atomicDisplacement()

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity of type length

evaluate(*charge_sum_rule_method=None*)

The Born effective charge $Z_{\alpha,ij}$ for the ion α displaced in direction *j* leading to the Polarization in direction *i*.

Parameters
charge_sum_rule_method (None | DistributeEvenly | DistributeRelative.) – Keyword specifying if a charge sum-rule should be applied to the Born effective charges. Can be None for no sum-rule application. Alternatively the charge error is either distributed evenly (DistributeEvenly) or relative (DistributeRelative) to the Born effective charges on the atoms. **Default:** None

Returns
The Born effective charge as an array of shape (*n*, 3, 3), where *n* is the number of atoms. The second last entry is the Cartesian direction of polarization and the last is the direction in which the ion is displaced.

Return type
PhysicalQuantity of type charge

kpointsA()

Returns
The k-point sampling used for integrating along the first direction.

Return type
MonkhorstPackGrid

kpointsB()

Returns
The k-point sampling used for integrating along the second direction.

Return type
MonkhorstPackGrid

kpointsC()

Returns
The k-point sampling used for integrating along the third direction.

Return type
MonkhorstPackGrid

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

nonSelfconsistentUpdate()

Returns
True if the displaced configurations are updated non-selfconsistently.

Return type
bool

polarizationCartesianDirections()

Returns
The Cartesian components for which the polarization is evaluated.

Return type
list of `CartesianDirection.{X,Y,Z}`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

useSymmetry()

Returns
True if the use of crystal symmetry to reduce the number of displacements is enabled.

Return type
bool

Usage Examples

Calculate the born effective charge of GaAs:

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.6537*Angstrom)

# Define elements
elements = [Gallium, Arsenic]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = LDA.PZ

k_point_sampling = MonkhorstPackGrid(
    na=14,
    nb=14,
    nc=14,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=30.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=FermiDirac(100.0*Kelvin*boltzmann_constant),
    )

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('GaAs.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Born Effective Charge
# ------------------------------------------------------------
born_effective_charge = BornEffectiveCharge(
    configuration=bulk_configuration,
    kpoints_a=MonkhorstPackGrid(21, 5, 5),
    kpoints_b=MonkhorstPackGrid(5, 21, 5),
    kpoints_c=MonkhorstPackGrid(5, 5, 21),
    atomic_displacement=0.01*Angstrom,
    )
nlsave('GaAs.hdf5', born_effective_charge)
```

born_effective_charge.py

The output from the calculation is:

```
+-------------------------------------------------------------------------------+
| Born Effective Charge Report                                                  |
+-------------------------------------------------------------------------------+
|                                                                               |
| 0 Ga:                                                                         |
|                                                                               |
| [[ 2.08384046e+00  2.01655410e-16 -2.01655418e-16]                            |
|  [ 2.01655410e-16  2.08384046e+00 -2.01655418e-16]                            |
|  [ 0.00000000e+00  0.00000000e+00  2.08384046e+00]] e                         |
|                                                                               |
+-------------------------------------------------------------------------------+
|                                                                               |
| 1 As:                                                                         |
|                                                                               |
| [[-2.08597738e+00 -2.01862227e-16  2.01862227e-16]                            |
|  [-2.01862228e-16 -2.08597738e+00  2.01862228e-16]                            |
|  [ 0.00000000e+00  0.00000000e+00 -2.08597738e+00]] e                         |
|                                                                               |
+-------------------------------------------------------------------------------+
```

The calculated Born effective charge tensors are diagonal with a diagonal value of around ± 2.1 for Ga and As, respectively, in agreement with experiments and previous calculations[1].

Notes

The Born effective charge tensor is defined as the derivative of the polarization with respect to the atomic coordinate

$$Z^*_{\mu,ij} = \Omega_0 \frac{\partial \mathbf{P}_i}{\partial \mathbf{r}_{\mu,j}},$$

where Ω_0 is the unit cell volume, $\mathbf{P}_i$ is the ith Cartesian component of the Polarization and $\mathbf{r}_{\mu,j}$ is the jth Cartesian component of the position of atom μ.

When calculating the Born effective charge notice the following

- Perform the calculation with a low electronic broadening (e.g. 100 K). The default of 1000 K can lead to wrong results for small band gap materials.
- Carefully check the results agaist the parameter *atomic_displacement*. Ususally the default value of *0.01*Angstrom* is a good choice but you might need to either increase or decrease it.

BornEffectiveChargeParameters

class BornEffectiveChargeParameters(*kpoints_a=None*, *kpoints_b=None*, *kpoints_c=None*, *atomic_displacement=None*, *use_symmetry=None*, *non_selfconsistent_update=None*, *polarization_cartesian_directions=None*, *atom_indices=None*, *processes_per_displacement=None*)

Class for parameters used to evaluate `BornEffectiveCharge` analysis.

Parameters

[1] R.ăD. King-Smith and D.ăVanderbilt. Theory of polarization of crystalline solids. *Phys. Rev. B*, 47:1651–1654, Jan 1993. doi:10.1103/PhysRevB.47.1651.

- **kpoints_a** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the first direction. **Default:** `MonkhorstPackGrid(n_aa, n_ab, n_ac)`, where *n_aa*, *n_ab*, and *n_ac* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_aa = 2 * na + 1, n_ab = nb, n_ac = nc`.
- **kpoints_b** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the second direction. **Default:** `MonkhorstPackGrid(n_ba, n_bb, n_bc)`, where *n_ba*, *n_bb*, and *n_bc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ba = na, n_bb = 2 * nb + 1, n_bc = nc`.
- **kpoints_c** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the third direction. **Default:** `MonkhorstPackGrid(n_ca, n_cb, n_cc)`, where *n_ca*, *n_cb*, and *n_cc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ca = na, n_cb = nb, n_cc = 2 * nc + 1`.
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01*Angstrom`
- **use_symmetry** (*bool*) – If enabled, only the symmetrically unique atoms are displaced and the remaining born effective charges are calculated using symmetry. **Default:** `True`
- **non_selfconsistent_update** (*bool*) – If True, the displaced configurations are updated non-self-consistently. This leads to a computational speed-up, but the results should be carefully checked against self-consistent calculations. **Default:** `False`
- **polarization_cartesian_directions** (List of list of `CartesianDirection.{X,Y,Z}`) – Restrict the evaluation of the Polarization to specific cartesian directions. For example, if `[CartesianDirection.X]` is specified, only the `xx, xy, xz` components of the born effective charge tensor are calculated. This option can be used to speed up the calculation if only such components are required. **Default:** `[CartesianDirection.X, CartesianDirection.Y, CartesianDirection.Z]`
- **processes_per_displacement** (`Automatic` | int) – The number of processes to use per atomic displacement. When set to `Automatic` the number will be determined automatically maximizing the number of displacements calculated in parallel and minimizing the number of idle processes. One may set this number manually in order to reduce the memory requirements for each process. **Default:** `Automatic`

asDict()

Returns
The input parameters as keyword, value dictionary.

Return type
dict

atomIndices()

Returns
The atom indices.

Return type
list of int | `All`

atomicDisplacement()

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity of type length

kpointsA()

Returns
The k-point sampling used for integrating along the first direction.

Return type
MonkhorstPackGrid

kpointsB()

Returns
The k-point sampling used for integrating along the second direction.

Return type
MonkhorstPackGrid

kpointsC()

Returns
The k-point sampling used for integrating along the third direction.

Return type
MonkhorstPackGrid

nlinfo()

Returns
The nlinfo.

Return type
dict

nonSelfconsistentUpdate()

Returns
True if the displaced configurations are updated non-selfconsistently.

Return type
bool

polarizationCartesianDirections()

Returns
The Cartesian components for which the tensor is evaluated.

Return type
List of list of `CartesianDirection.{X,Y,Z}`

processesPerDisplacement()

Returns
The number of processes per displacement.

Return type
`Automatic` | int

uniqueString()

Return a unique string representing the state of the object.

useSymmetry()

Returns
True if the use of crystal symmetry to reduce the number of displacements is enabled.

Return type
bool

BoxRegion

class BoxRegion(*value*, *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*)

Class for representing an orthorhombic region, i.e. a rectangular box of metallic or dielectric material.

Parameters

- **value** (float | *PhysicalQuantity* compatible with Volt) – The value that should be assigned to the box. Either the voltage or the dielectric constant of the region.
- **xmin** (*PhysicalQuantity* of type length) – The lower boundary of the box along the x-axis.
- **xmax** (*PhysicalQuantity* of type length) – The upper boundary of the box along the x-axis.
- **ymin** (*PhysicalQuantity* of type length) – The lower boundary of the box along the y-axis.
- **ymax** (*PhysicalQuantity* of type length) – The upper boundary of the box along the y-axis.
- **zmin** (*PhysicalQuantity* of type length) – The lower boundary of the box along the z-axis.
- **zmax** (*PhysicalQuantity* of type length) – The upper boundary of the box along the z-axis.

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *header=True*)

Print a string containing an ASCII description of the SpatialRegion.

Parameters

- **stream** (*Stream based object*) – The io to write to.
- **header** (*bool*) – Option to emphasize the description as a header or not.

toGLObject(*color*)

Create the necessary object(s) to pass back into the GLConfiguration.

Parameters
color (*GLColor*) – The region type-specific color.

Returns
The box.

Return type
GLBox

uniqueString()

Return a unique string representing the state of the object.

value()

Function for asking for the value of the box region.

Returns
The value of the box region.

Return type
float | *PhysicalQuantity* compatible with Volt

xmax()

Returns
The maximum x-value of the box.

Return type
PhysicalQuantity of type length

xmin()

Returns
The minimum x-value of the box.

Return type
PhysicalQuantity of type length

ymax()

Returns
The maximum y-value of the box.

Return type
PhysicalQuantity of type length

ymin()

Returns
The minimum y-value of the box.

Return type
PhysicalQuantity of type length

zmax()

Returns
The maximum z-value of the box.

Return type
PhysicalQuantity of type length

zmin()

Returns
The minimum z-value of the box.

Return type
PhysicalQuantity of type length

Usage Examples

Define a dielectric region with dielectric constant $4\varepsilon_0$:

```
dielectric_region = BoxRegion(
    4.0,
    xmin=0*Angstrom, xmax=41.439*Angstrom,
    ymin=0.5*Angstrom, ymax=3.57914*Angstrom,
    zmin=5*Angstrom, zmax=50.4135*Angstrom
    )
```

dielectric_boxregion.py

Define two metallic regions where the second metallic region is constructed by cloning the first:

```
metallic_region1 = BoxRegion(
    1*Volt,
    xmin=0*Angstrom, xmax=41.439*Angstrom,
    ymin=0*Angstrom, ymax=0.5*Angstrom,
    zmin=5*Angstrom, zmax=15.*Angstrom
    )

metallic_region2 = metallic_region1(
    value=-1*Volt,
    ymin=10*Angstrom, ymax=12*Angstrom
    )
```

metal_boxregion.py

Notes

The following spatial regions are available:

- *BoxRegion*
- *SphereRegion*
- *TubeRegion*

BravaisLatticeConstraint

class BravaisLatticeConstraint

This constraint preserves the bravais lattice in constant stress simulations or optimization.

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run a geometry optimization of a rutile crystal with the Bravais lattice type fixed.

```
constraints = [BravaisLatticeConstraint()]

bulk_configuration = OptimizeGeometry(
    bulk_configuration,
    max_forces=0.05*eV/Ang,
    max_stress=0.1*GPa,
    max_steps=200,
    max_step_length=0.2*Ang,
    constraints=constraints,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
)
```

bravais_lattice_constraint.py

Notes

- *BravaisLatticeConstraint* can be used in *OptimizeGeometry* to constrain the Bravais lattice type. This means that the stress is symmetrized according to the lattice type. The lattice vectors will still be optimized, but the lattice type will not change.

BrennerCalculator

class BrennerCalculator(*dynamical_matrix_parameters=None*)

Class for representing a Brenner Potential 2002 calculator.

Parameters
dynamical_matrix_parameters (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.

dynamicalMatrixParameters()

This method is deprecated.

Returns
None always.

Return type
None

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

supportedElements()

This function queries the calculator for the elements it supports.

Returns
A list of chemical symbols for each element supported.

Return type
list of str

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Method to upgrade the current calculator based on the given configuration.

Parameters
configuration (AtomicConfiguration sub-class) – The configuration the calculator will be used for.

Usage Examples

Define a Brenner calculator:

```
calculator = BrennerCalculator()
```

Notes

The calculator uses the classical Brenner potential version 2002[1] and is limited to the elements: Hydrogen, Carbon, Silicon, and Germanium.

The implementation uses the ASE-ASAP code.

BuckinghamPotential

class BuckinghamPotential(*particleType1*, *particleType2*, *A*, *rho*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.

[1] D.ăW. Brenner, O.ăShenderova, J.ăHarrison, S.ăJ. Stuart, B.ăNi, and S.ăB Sinnott. A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons. *J. Phys.: Condensed Matter*, 14(4):783–802, 2002. arXiv:S0953-8984(02)31186-X, doi:10.1088/0953-8984/14/4/312.

•**rho** (*PhysicalQuantity of type length*) – Potential parameter.

•**r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
```

(continues on next page)

(continued from previous page)

```
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ----------------------------------------------------------------
# Calculator
# ----------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='VanBeest_SiOAlP_1990')

# Add the paricle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the Buckingham potentials to the potential set
potentialSet.addPotential(BuckinghamPotential('O', 'O',
                                              A=1388.773*eV,
                                              rho=0.36231884*Angstrom,
                                              r_i=5.0*Angstrom,
                                              r_cut=5.5*Angstrom))
potentialSet.addPotential(BuckinghamPotential('Si', 'O',
                                              A=18003.7572*eV,
                                              rho=0.20520481*Angstrom,
                                              r_i=5.0*Angstrom,
                                              r_cut=5.5*Angstrom))

# Add the 6th-power dispersion term to the potential set
potentialSet.addPotential(LennardJonesMNPotential('O', 'O',
                                                  r_cut=9.0*Angstrom,
                                                  A=0.0*eV*Angstrom,
                                                  B=175.0*eV*Angstrom**6.0,
                                                  m=1.000000,
                                                  n=6.000000))
potentialSet.addPotential(LennardJonesMNPotential('Si', 'O',
                                                  r_cut=9.0*Angstrom,
                                                  A=0.0*eV*Angstrom,
                                                  B=133.5381*eV*Angstrom**6.0,
                                                  m=1.000000,
                                                  n=6.000000))

# Add the Coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom))
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

(continues on next page)

(continued from previous page)

```
bulk_configuration.update()
```

Notes

The *BuckinghamPotential* defines a pair potential of the form

$$V_{ij}(r) = Ae^{-r/\rho}\,.$$

Note that this potential form uses a repulsive term only. In general, Buckingham potentials often also include an attractive dispersion term of the form $-C/r^6$.

We have not included such a term in the **TremoloX** *BuckinghamPotential* implementation, as this can easily be achieved by adding a *LennardJonesMNPotential*, for example, as shown in the example above.

This permits combinations of the *BuckinghamPotential* with all kinds of attractive terms avaliable in the *TremoloXPotentialSet*, providing extended flexibility.

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

BulkConfiguration

class BulkConfiguration(*bravais_lattice*, *elements*, *cartesian_coordinates=None*, *fractional_coordinates=None*, *ghost_atoms=None*, *velocities=None*, *tag_data=None*, *fast_init=False*, *bonds=None*, *dielectric_regions=None*, *metallic_regions=None*, *partial_charges=None*, *improper_dihedral_indices=None*)

Class for representing a periodic structure, i.e bulk configuration.

Construction of a bulk configuration from a `BravaisLattice` object. The additional arguments are the elements, the Cartesian coordinates or fractional coordinates. The Cartesian coordinates and fractional coordinates cannot be given at the same time. The list arguments containing the elements and coordinates must have the same length.

Parameters

- **bravais_lattice** (`BravaisLattice`) – A lattice of the bulk configuration.
- **elements** (*PeriodicTableElement*) – A sequence containing the elements of the configuration.
- **cartesian_coordinates** (*PhysicalQuantity* of type length | None) – A sequence containing a sequence of atomic coordinates for each element in the configuration. Has the dimensionality nx3. **Default:** None
- **fractional_coordinates** (*array(n, 3) | None*) – A sequence containing sequences of fractional coordinates for each element in the configuration. **Default:** None

- **ghost_atoms** (*int(n)*) – A list of atom indices to treat as ghost atoms. **Default:** None
- **velocities** (*PhysicalQuantity* of type length/time | None) – The velocities of the atoms. Has the dimensionality nx3. **Default:** None
- **tag_data** (*dict*) – A dict of tag data, linking tags to a list of atom indices. **Default:** None
- **fast_init** (*bool*) – Skip checking types and do not copy the arguments (i.e. only store references to them. This is an advanced option that should only be used internally. **Default:** False
- **bonds** (*array with shape (n,2) or (n,5) | None*) – An array with two atom indices for each bond along with the vector which periodic images this bond connects to.
- **dielectric_regions** (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.
- **metallic_regions** (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.
- **partial_charges** (*PhysicalQuantity of type charge | None*) – A PhysicalQuantity array of the atomic partial charge for each atom.
- **improper_dihedral_indices** (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

addBonds(*bond_list*)

Add bonds.

Parameters
bond_list (*numpy.ndarray*) – The list of bonds to add.

addTags(*tags*, *indices=None*)

Add a set of tags to atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms.
- **indices** (*list | int | None*) – The list of indices to match atoms against. **Default:** All indices.

atomicMasses()

Returns
The masses of the atoms in the configuration.

Return type
PhysicalQuantity of type mass

atomicNumbers()

Returns
The list of atomic numbers associated with the elements.

Return type
list of ints

bonds()

Get the list of bond connections that can be used for bonded potentials in ATK-ForceField.

The connectivity indices refer to the wrapped configuration.

Returns
An array with the the two atom indices for each bond along with the vector which periodic images this bond connects.

Return type
array

bravaisLattice()

Returns
The bravais lattice of the configuration.

Return type
BravaisLattice

calculator()

Returns
The calculator attached to the configuration, i.e. the calculator that will be used for both simulation and analysis.

Return type
Calculator

cartesianCoordinates()

Returns
The Cartesian coordinates of the atoms as an nx3 array.

Return type
PhysicalQuantity of type length

center(*x_axis=True*, *y_axis=True*, *z_axis=True*)

Create a new *BulkConfiguration* by centering the atoms and the vacuum around the system.

The centered system is constructed with a cell of the type UnitCell.

Parameters

- **x_axis** (*bool*) – Boolean indicating if the system should be centered along the x-axis. **Default:** True.
- **y_axis** (*bool*) – Boolean indicating if the system should be centered along the y-axis. **Default:** True.
- **z_axis** (*bool*) – Boolean indicating if the system should be centered along the z-axis. **Default:** True.

Returns
The centered bulk system.

Return type
BulkConfiguration

cleave(*h=1*, *k=0*, *l=0*, *max_perpendicular_repeats=100*, *outer_atom_index=0*, *surface_transformation=None*, *layers=None*, *top_vacuum=None*, *bottom_vacuum=None*, *outer_atom_placement=None*)

Method to cleave the bulk system along the plane given by the Miller indices h, k, and l. Per the usual convention, the Miller indices defined a plane in the conventional unit cell.

Parameters

- **h** (*int*) – The first Miller index. **Default:** 1.
- **k** (*int*) – The second Miller index. **Default:** 0.
- **l** (*int*) – The third Miller index. **Default:** 0.
- **max_perpendicular_repeats** (*int*) – The maximum number of times the crystal lattice should be repeated in order to become perpendicular to the surfaces plane. Will throw an exception if it is not possible. These repetitions are only used when *layers* is *None*. **Default:** 100
- **outer_atom_index** (*int*) – The index of the configuration which should be the outer atom in the surface. This is the atom which intersects the cleave plane. **Default:** 0

- **surface_transformation** (*array of type float*) – A 2x2 array that transforms the lattice vectors perpendicular to the surface normal (i.e. the surface vectors). This defaults to the identity matrix, which leaves them unchanged. If the values are non integral, this will result in a strained surface. **Default:** *[[1, 0], [0, 1]]*
- **layers** (*float*) – The number of atomic layers in the repeated structure. By default the smallest number of layers that yields perpendicular unit cell (c-axis purely along the z-axis) is used. **Default:** None
- **top_vacuum** (*PhysicalQuantity* of type length) – The vacuum padding to add to the top of the cell. **Default:** 0.0*Angstrom
- **bottom_vacuum** (*PhysicalQuantity* of type length) – The vacuum padding to add to the bottom of the cell. **Default:** 0.0*Angstrom
- **outer_atom_placement** (*float*) – The fractional c-coordinate of the outer atom. By default the location of this atom will be at c=1.0 if no top or bottom vacuum is used. If vacuum is used, then the atom will be located at the correct place in the unit cell to accommodate the vacuum settings. **Default:** None

Returns
The cleaved bulk configuration.

Return type
BulkConfiguration

coarseGrainDescriptors()

Returns
The list of either elements or coarse grain particles for each element. Isotopes are returned as just their base element, UnitedAtoms and Particle are given as their specific type.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

copy()

Returns
A copy of the current configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndDeleteAtoms(*indices*)

Create a new configuration by deleting some atoms from this configuration.

Parameters
indices (*list of int*) – The indices of the atoms to delete.

Returns
The configuration with some atoms deleted.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndMerge(*other*)

Create a new configuration by merging this configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The other configuration.

Returns
The merged configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndShiftAtoms(*displacement*, *indices=None*)

Create a new configuration with some atoms translated.

Parameters

- **displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.
- **indices** (*list of int*) – The indices to shift. **Default:** All.

Returns
The configuration with the translation applied.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

deleteAtoms(*indices*)

Delete atoms from the configuration by specifying their indices.

Parameters
indices (*array of int*) – The indices to delete.

deleteBonds(*bond_list=None*, *pair_selection=None*)

Delete bonds connected to atomic indices.

Parameters

- **bond_list** (*A two-dimensional sequence*) – The pairs of bonds indices.
- **pair_selection** (*list | None*) – Specifies two groups between which bonds are delete. Selectable groups are elements, index lists, tag names, or None (all atoms).

dielectricRegions()

Returns
The dielectric regions in the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

elements()

Returns
The elements in configuration.

Return type
list of *PeriodicTableElement*

externalPotential()

Returns
The external potential present on the configuration.

Return type
AtomicShift | *AtomicCompensationCharge* | None

findBonds(*fuzz_factor=1.1*, *pair_selection=None*, *periodic_boundaries=None*)

Find bonds in the configuration according to the combined covalent radii of the element pairs, multiplied with a fuzz factor. Optionally, find bonds only between two specified sub-groups of atoms. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.
- **pair_selection** (list(2) of type *PeriodicTableElement*, list of int, or str) – Specifies two groups between which bonds are detected. Selectable groups are elements, index lists, tag names, or None (all atoms). By default bonds between all atoms in the configuration are taken into account.
- **periodic_boundaries** (*3-tuple of bool*) – The periodic boundaries **Default:** The periodic boundaries for the configuration.

fixedSpinDirections()

Returns
The fixed spin directions for the configuration.

Return type
FixedSpin | None

fractionalCoordinates()

Returns
The fractional coordinates of the bulk configuration as a nx3 array.

Return type
array of floats

generateShifts()

Method for generating a list of origin shifts along all 3 lattice directions.

It will create a list of 27 shifts of the supercell origin (-1 to 1) in the three periodic directions.

Returns
The array of shifts

Return type
PhysicalQuantity of type length

ghostAtoms()

Returns
The list of ghost atoms.

Return type
list of ints

improperDihedralIndices()

Returns
The list of atom indices for each improper dihedral or None if no improper dihedrals are defined. Improper dihedrals are mainly used in bonded force fields.

Return type
numpy array | None

indices()

Returns
The indices of the atoms.

Return type
list of int

indicesFromIsotopes(*isotopes*)

Parameters
isotopes (*list of type PeriodicTableElement or Isotope*) – The isotopes to select.

Returns
The indices of the selected isotopes.

Return type
list of type int

indicesFromTags(*tags=None*)

List the indices associated with a given collection of tags.

Parameters
tags (*list | str*) – A list of tags for which all matching indices should be extracted.

Returns
The list of indices corresponding to the specified tag name(s).

Return type
list of ints

magneticMoments()

Returns
The magnetic moments of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type Bohr magneton.

massDensity()

Returns
The mass density.

Return type
PhysicalQuantity of type mass per volume

merge(*other*)

Merge configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – A different configuration.

metallicRegions()

Returns
The metallic regions for the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The configuration information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AtomicConfiguration object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger().

numberOfAtoms()

Returns
The total number of atoms in the configuration.

Return type
int

partialCharges(*indices=None*)

Get the list of partial atomic charges that can be used for representing electrostatic interactions in ATK-ForceField.

Parameters
indices (*list | int | None*) – The indices for which to return the partial charges. **Default:** All indices.

Returns
A PhysicalQuantity array of the atomic partial charge for each atom.

Return type
PhysicalQuantity of type charge | None

particleDescriptors()

Returns
The list of particle descriptors for each atom. This should return the exact same list as was given in the constructor argument elements.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

static periodicBoundaries()

Returns
The periodic boundary conditions of the configuration.

Return type
list

primitiveVectors()

Returns
The primitive lattice vectors.

Return type
PhysicalQuantity of type length

reduce(*na=1*, *nb=1*, *nc=1*, *stack_systems=True*)

Reduce the *BulkConfiguration* with the integer values na, nb, and nc along the three primitive unit cell vectors. The reduced system is constructed with a cell of the type UnitCell.

Parameters

- **na** (*int*) – The repetition along the a-axis. **Default:** 1.
- **nb** (*int*) – The repetition along the b-axis. **Default:** 1.
- **nc** (*int*) – The repetition along the c-axis. **Default:** 1.
- **stack_systems** (*bool*) – If True the basis atoms are reduced as a unit, i.e. a0 and b0 are reduced as: a0,b0,a1,b1, If False the basis atom are reduced individually, i.e. a0 and b0 are reduced as: a0,a1,, b0,b1, **Default:** True.

Returns
The reduced bulk system.

Return type
BulkConfiguration

removeTags(*tags=None*, *indices=None*, *purge=False*)

Remove a set of tags from atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms. **Default:** All tags.
- **indices** (*list | int*) – The list of indices to match atoms against. **Default:** All indices.
- **purge** (*bool*) – When removing tags from the configuration, delete the tag completely when not associated with any atoms anymore. **Default:** False.

repeat(*na=1*, *nb=1*, *nc=1*, *stack_systems=True*, *transfer_density_matrix=True*)

Repeat the *BulkConfiguration* with the integer values na, nb, and nc along the three primitive unit cell vectors. The repeated system is constructed with a cell of the type UnitCell.

Parameters

- **na** (*int*) – The repetition along the a-axis. **Default:** 1.
- **nb** (*int*) – The repetition along the b-axis. **Default:** 1.
- **nc** (*int*) – The repetition along the c-axis. **Default:** 1.
- **stack_systems** (*bool*) – If True the basis atoms are repeated as a unit, i.e. a0 and b0 are repeated as: a0,b0,a1,b1, If False the basis atom are repeated individually, i.e. a0 and b0 are repeated as: a0,a1,, b0,b1, **Default:** True.
- **transfer_density_matrix** (*bool*) – If True, the density matrix (or rather the schroedinger container) of the non-repeated system is used to set up the density matrix of the repeated system. **Default:** True.

Returns
The repeated bulk system.

Return type
BulkConfiguration

scalePartialCharges(*scale_factor*, *indices=None*)

Scale the partial charges with a given scale factor. These partial charges are used with Forcefield calculators.

Parameters

- **scale_factor** (*float*) – The factor for scaling charges.
- **indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

scalePrimitiveVectors(*scale_a=None*, *scale_b=None*, *scale_c=None*)

Scale the length of each lattice vector. The fractional coordinates of the atoms and the Bravais lattice type will be remain unchanged if possible.

Parameters

- **scale_a** (*float*) – The scaling factor for the a-axis to apply. **Default:** 1.0.
- **scale_b** (*float*) – The scaling factor for the b-axis to apply. **Default:** 1.0.
- **scale_c** (*float*) – The scaling factor for the c-axis to apply. **Default:** 1.0.

scaleVolume(*scaling_factor*)

Scale the volume of the crystal by an isotropic expansion of the lattice vectors. The fractional coordinates of the atoms and the Bravais lattice type will be unchanged.

Parameters

scaling_factor (*float*) – The scaling factor to apply. A value of `1.01` leads to a 1% increase in the volume.

setBonds(*bond_list*, *skip_checks=False*)

Set the bonds on the configuration. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

- **bond_list** (*list(n, 2) | list(n, 5) | None*) – A list which contains for each bond the indices of the two connected atoms. Optionally, three more integers can be specified for each bond, which must be between -1 and 1, and which denote to which neighboring image cell the bond is connected. Without these additional indices, the minimum image convention is obeyed.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setBravaisLattice(*bravais_lattice*, *conserve_coordinates=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Cartesian'>*, *skip_checks=False*)

Change the bravais lattice.

Parameters

- **bravais_lattice** (`BravaisLattice`) – The new lattice.
- **conserve_coordinates** (`Cartesian` | `Fractional`) – The type of coordinates to conserve.. **Default:** `Cartesian`.
- **skip_checks** (*bool*) – If True, ignores bond connectivity and input argument checks. **Default:** False.

setCalculator(*calculator*, *initial_state=None*, *initial_spin=None*)

Attach a *Calculator* to the configuration which will be used in calculations involving the configuration.

Parameters

- **calculator** (*Calculator* | None) – The calculator object that should be attached to the configuration.
- **initial_state** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* with a calculator | None) – The initial state to be used for this configuration. **Default:** No initial state.
- **initial_spin** (*InitialSpin* | None) – The initial InitialSpin object to be used for this configuration. **Default:** `InitialSpin()`, except when used with a calculator of type *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*. In this case `InitialSpin([0., 0., ...])` is used, i.e. the scaled spins for each atom are set to 0.

setCartesianCoordinates(*cartesian_coordinates*, *indices=None*, *skip_checks=False*)

Set the Cartesian coordinates of the atoms.

Parameters

- **cartesian_coordinates** (*PhysicalQuantity* of type length) – The new coordinates of the atoms in each image.

- **indices** (*list*) – The indices of the atoms to set the positions of. **Default:** All indices.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setDielectricRegions(*dielectric_regions*)

Set the dielectric regions for the configuration.

Parameters
dielectric_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

setExternalPotential(*external_potential*)

Set an external potential on the configuration that will be used in calculations involving the configuration.

Parameters
external_potential (*AtomicShift* | *AtomicCompensationCharge*) – The external potential to apply.

setFractionalCoordinates(*fractional_coordinates*, *indices=None*, *skip_checks=False*)

Set the Cartesian coordinates of the atoms.

Parameters

- **fractional_coordinates** (*array of floats*) – The new coordinates of the atoms as a nx3 array.
- **indices** (*array of ints | None*) – The indices of the atoms to set the positions of. **Default:** All indices.
- **skip_checks** (*bool*) – If True, ignores bond connectivity and input argument checks. **Default:** False.

setImproperDihedralIndices(*improper_dihedral_indices*)

Set the list of atom indices for each improper dihedral in bonded force fields.

Parameters
improper_dihedral_indices (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

setMagneticField(*magnetic_field*)

Set local magnetic field. The spins will be forced to point in the directions given by the magnetic_field object. The magnetic field can be defined for each atom. This only has an effect for Noncollinear or Spinorbit calculations.

Parameters
magnetic_field (*FixedSpin*) – The magnetic field to be applied.

setMagneticMoments(*magnetic_moments=None*, *skip_checks=False*)

Function to set magnetic moments on the configuration.

Parameters

- **magnetic_moments** (*PhysicalQuantity* of type Bohr magneton.) – The magnetic_moments to set on the configuration. Has the dimensionality nx3. **Default:** None.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setMetallicRegions(*metallic_regions*)

Set the metallic regions for the configuration.

Parameters
metallic_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setPartialCharges(*charge_list*, *indices=None*, *skip_checks=False*, *update_calculator=True*)

Set the partial charges on the configuration. The partial charges are used primarily to model electrostatic interactions in the TremoloX-calculator.

Parameters

•**charge_list** (*PhysicalQuantity of type charge | None*) – A list of atomic partial charges which contains a charge for each atom.

•**indices** (*list | int | None*) – The indices for which to set the partial charges. **Default:** All indices.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

•**update_calculator** (*bool*) – Whether to update and attached Forcefield calculator with the new charges. **Default:** True.

setPrimitiveVectors(*a*, *b*, *c*, *conserve_coordinates=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Cartesian'>*, *skip_checks=False*)

Set the primitive lattice vectors for the configuration. This will change the `BravaisLattice` to a `UnitCell`. By default the Cartesian coordinates are not changed, which means that the relative positions of the atoms will change when the lattice vectors change.

Parameters

•**a** (*PhysicalQuantity* of type length) – The first lattice vector.

•**b** (*PhysicalQuantity* of type length) – The second lattice vector.

•**c** (*PhysicalQuantity* of type length) – The third lattice vector.

•**conserve_coordinates** (`Cartesian` | `Fractional`) – The type of coordinates to conserve. **Default:** `Cartesian`.

•**skip_checks** (*bool*) – If True, ignores bond connectivity and input argument checks. **Default:** True.

setVelocities(*velocities=None*, *skip_checks=False*)

Function to set velocities on the configuration.

Parameters

•**velocities** (*PhysicalQuantity* of type velocity | None) – The velocities to set on the configuration. Has the dimensionality nx3. **Default:** None.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

shiftAtoms(*displacement*, *indices=None*, *skip_checks=False*)

Shift atoms by indices.

Parameters

•**displacement** (*PhysicalQuantity* of type length | array) – The displacement that should be applied to the atom positions. Can be given as cartesian or fractional coordinates.

•**indices** (*NoneType | list of ints*) – The indices to shift.

•**skip_checks** (*bool*) – True, if all consistency checks should be skipped.

shiftPartialCharges(*total_charge*, *indices=None*)

Shift the partial charges so that their sum is the given total charge value. These partial charges are used with Forcefield calculators.

Parameters

•**total_charge** (*PhysicalQuantity of type charge*) – The new total charge.

•**indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

sortAlongC()

Method for sorting the atoms along C.

symbols()

Returns
The element symbols of the configuration.

Return type
list of str

tags(*indices=None*)

List the tags associated with a given collection of indices. The list returned is the set union of tags associated with the given indices. If no collection of indices is provided, then all tags on the configuration are returned.

Parameters
indices (*list | int*) – The indices to check. **Default:** All indices.

Returns
The set union of tags present on the provided indices.

Return type
set

toMoleculeConfiguration()

Method for converting the BulkConfiguration to a MoleculeConfiguration.

Returns
The molecule configuration corresponding to the original bulk with the unitcell removed.

Return type
BulkConfiguration

uniqueElements(*ordered=False*)

Parameters
ordered (*bool*) – If the elements should be returned in ascending order by atomic number.

Returns
The unique elements contained in the configuration.

Return type
list of *PeriodicTableElement*

uniqueString()

Return a unique string representing the state of the object.

update(*force_restart=False*)

A self-consistent solution is generated, using the currently set calculator.

Parameters
force_restart (*bool*) – Force the self-consistent calculation to restart. **Default:** False.

velocities()

Returns
The velocities of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type velocity

Usage Examples

Define the geometry of a lithium BCC crystal:

```
lattice = BodyCenteredCubic( 3.509 * Ang)
li_bcc_bulk = BulkConfiguration(
    lattice,
    [Lithium],
    [ ( 0.0, 0.0, 0.0 ) * Ang ]
    )
```

li_bcc.py

Specify the geometry of a diamond crystal:

```
elements = [ Carbon ] * 2
coordinates = [
    ( 0.00, 0.00, 0.00 ),
    ( 0.25, 0.25, 0.25 )
    ]
diamond_lattice = FaceCenteredCubic( 3.567 * Ang )
diamond = BulkConfiguration(
    diamond_lattice,
    elements,
    fractional_coordinates = coordinates
    )
```

si_diamond.py

Notes

ATK recognizes four types of atomic geometries:

•Molecules (*MoleculeConfiguration*)

•Bulk (*BulkConfiguration*)

•One Probe (*SurfaceConfiguration*)

•Two Probe (*DeviceConfiguration*)

1D or 2D periodic geometries, such as nanotubes, slabs, and atomic chains must be represented as a *BulkConfiguration* with sufficient vacuum to isolate the system in the non-periodic directions.

It is possible to specify vacuum basis sets (ghost atoms) by adding an additional atom and include the index of that atom in the `ghost_atom` list. In this case the valence basis set of the atom is included in the orbital list, but there will be no atomic potential at the site.

COMB3FieldCorrection

class COMB3FieldCorrection(*particleType1*, *particleType2*, *Pchi*, *PJ*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **Pchi** (*PhysicalQuantity of length**2 * charge*) – Potential parameter.
- **PJ** (*PhysicalQuantity of length**4*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – Inner cutoff radius (distance where the smoothing starts)
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB3 potential for a TiN rocksalt crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# Set up a Titanium-Nitride cell

vector_a = [4.235, 0.0, 0.0]*Angstrom
vector_b = [0.0, 4.235, 0.0]*Angstrom
vector_c = [0.0, 0.0, 4.235]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Titanium, Nitrogen, Titanium, Nitrogen, Titanium, Nitrogen,
            Titanium, Nitrogen]

# Define coordinates
```

(continues on next page)

(continued from previous page)

```
fractional_coordinates = [[ 0. ,  0. ,  0. ],
                          [ 0.5,  0.5,  0.5],
                          [ 0.5,  0.5,  0. ],
                          [ 0. ,  0. ,  0.5],
                          [ 0.5,  0. ,  0.5],
                          [ 0. ,  0.5,  0. ],
                          [ 0. ,  0.5,  0.5],
                          [ 0.5,  0. ,  0. ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name = 'COMB_NTi_2014')
potentialSet.addParticleType(ParticleType(
    symbol='N',
    mass=14.0067 * atomic_mass_unit,
    atomicNumber=7
))
potentialSet.addParticleType(ParticleType(
    symbol='Ti',
    mass=47.867 * atomic_mass_unit,
    atomicNumber=22
))

option = COMBCoulombOption(alpha = 0.2*1/Angstrom, r_cut = 11.0*Angstrom)
potentialSet.addOption(option)
potential = COMB3Particle(
    particleType = 'Ti',
    l = 2.136011*1/Angstrom,
    mu = 1.178831*1/Angstrom,
    m = 1,
    n = 0.566048,
    QL = -4.0*elementary_charge,
    QU = 4.0*elementary_charge,
    DL = 0.005*Angstrom,
    DU = -0.5*Angstrom,
    nB = 10,
    J1 = 3.095768*eV/elementary_charge,
    J2 = 4.23028*eV/elementary_charge**2,
    J3 = -1.039759*eV/elementary_charge**3,
    J4 = 0.357428*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 0.724352*1/Angstrom,
    Z = 3.022932*elementary_charge,
```

(continues on next page)

(continued from previous page)

```
    P = 0.335*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3Particle(
    particleType = 'N',
    l = 5.218037*1/Angstrom,
    mu = 3.738549*1/Angstrom,
    m = 1,
    n = 1.0,
    QL = -2.0*elementary_charge,
    QU = 6.0*elementary_charge,
    DL = 0.007664*Angstrom,
    DU = -1.213951*Angstrom,
    nB = 10,
    J1 = 6.59963*eV/elementary_charge,
    J2 = 5.955097*eV/elementary_charge**2,
    J3 = 0.760433*eV/elementary_charge**3,
    J4 = 0.009388*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 1.371794*1/Angstrom,
    Z = -1.53917*elementary_charge,
    P = 0.5167863369*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    A = 516.5873248*eV,
    B0 = 117.0421345*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
    l = 2.136011*1/Angstrom,
    mu0 = 1.178831*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 0.545629*1/Angstrom,
    b0 = 0.077183,
    b1 = 0.146606,
    b2 = 0.226674,
    b3 = 0.093054,
    b4 = -0.130433,
    b5 = -0.065549,
    b6 = 0.5504,
    c0 = -0.012318,
    c1 = 0.175079,
    c2 = 0.066085,
    c3 = -0.076109,
    N = 1.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
```

(continues on next page)

(continued from previous page)

```
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'N',
    A = 7654.972656*eV,
    B0 = 2102.295654*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
    l = 5.218037*1/Angstrom,
    mu0 = 3.738549*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 3.738549*1/Angstrom,
    b0 = 1.691325,
    b1 = 0.283832,
    b2 = 0.698596,
    b3 = 1.724015,
    b4 = 0.812665,
    b5 = 0.0,
    b6 = 0.0,
    c0 = 0.0,
    c1 = 0.0,
    c2 = 0.0,
    c3 = 0.0,
    N = 1.0,
    r_i = 2.0*Angstrom,
    r_cut = 2.3*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'N',
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 2.297286*1/Angstrom,
    b0 = 0.225186,
    b1 = 0.269113,
    b2 = 0.261613,
    b3 = 0.033596,
    b4 = -0.113159,
    b5 = 2.2e-05,
    b6 = 3e-06,
    c0 = -0.032039,
    c1 = -0.082354,
    c2 = 0.024479,
```

(continues on next page)

(continued from previous page)

```
    c3 = -0.073187,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'Ti',
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 5.066036*1/Angstrom,
    b0 = 0.024877,
    b1 = -0.103268,
    b2 = 0.263335,
    b3 = 0.412664,
    b4 = 0.017287,
    b5 = 2.3e-05,
    b6 = 0.0,
    c0 = -0.056517,
    c1 = -0.036331,
    c2 = 0.007137,
    c3 = 0.06462,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'N',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'Ti',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'N',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = AngleCorrection6Potential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    particleType3 = 'Ti',
    k0 = 0.0*eV,
    k1 = 0.043561*eV,
    k2 = 0.0*eV,
    k3 = 0.049195*eV,
    k4 = 0.0*eV,
    k5 = 0.0*eV,
    k6 = 0.0*eV,
    costheta0 = 0.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")

bulk_configuration.setCalculator(calculator)
```

See also ref.comb3common.notes.

COMB3PairPotential

class COMB3PairPotential(*particleType1*, *particleType2*, *A*, *B0*, *B1*, *B2*, *l*, *mu0*, *mu1*, *mu2*, *beta*, *b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *c0*, *c1*, *c2*, *c3*, *N*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Note: COMB3Particle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**A** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B0** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B1** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B2** (*PhysicalQuantity of type energy*) – Potential parameter.

•**l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).

•**mu0** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**mu1** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**mu2** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**beta** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**b0** (*float*) – Potential parameter.

•**b1** (*float*) – Potential parameter.

•**b2** (*float*) – Potential parameter.

•**b3** (*float*) – Potential parameter.

•**b4** (*float*) – Potential parameter.

•**b5** (*float*) – Potential parameter.

•**b6** (*float*) – Potential parameter.

•**c0** (*float*) – Potential parameter.

•**c1** (*float*) – Potential parameter.

•**c2** (*float*) – Potential parameter.

•**c3** (*float*) – Potential parameter.

•**N** (*float*) – Potential parameter.

•**r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius (distance where the smoothing starts).

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB3 potential for a TiN rocksalt crystal by adding particle types and interaction functions to the TremoloxPotentialSet.

```
# Set up a Titanium-Nitride cell

vector_a = [4.235, 0.0, 0.0]*Angstrom
vector_b = [0.0, 4.235, 0.0]*Angstrom
vector_c = [0.0, 0.0, 4.235]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Titanium, Nitrogen, Titanium, Nitrogen, Titanium, Nitrogen,
            Titanium, Nitrogen]

# Define coordinates
fractional_coordinates = [[ 0. ,  0. ,  0. ],
                          [ 0.5,  0.5,  0.5],
                          [ 0.5,  0.5,  0. ],
                          [ 0. ,  0. ,  0.5],
                          [ 0.5,  0. ,  0.5],
                          [ 0. ,  0.5,  0. ],
                          [ 0. ,  0.5,  0.5],
                          [ 0.5,  0. ,  0. ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name = 'COMB_NTi_2014')
potentialSet.addParticleType(ParticleType(
    symbol='N',
    mass=14.0067 * atomic_mass_unit,
    atomicNumber=7
))
potentialSet.addParticleType(ParticleType(
    symbol='Ti',
    mass=47.867 * atomic_mass_unit,
    atomicNumber=22
))

option = COMBCoulombOption(alpha = 0.2*1/Angstrom, r_cut = 11.0*Angstrom)
```

(continues on next page)

(continued from previous page)

```
potentialSet.addOption(option)
potential = COMB3Particle(
    particleType = 'Ti',
    l = 2.136011*1/Angstrom,
    mu = 1.178831*1/Angstrom,
    m = 1,
    n = 0.566048,
    QL = -4.0*elementary_charge,
    QU = 4.0*elementary_charge,
    DL = 0.005*Angstrom,
    DU = -0.5*Angstrom,
    nB = 10,
    J1 = 3.095768*eV/elementary_charge,
    J2 = 4.23028*eV/elementary_charge**2,
    J3 = -1.039759*eV/elementary_charge**3,
    J4 = 0.357428*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 0.724352*1/Angstrom,
    Z = 3.022932*elementary_charge,
    P = 0.335*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3Particle(
    particleType = 'N',
    l = 5.218037*1/Angstrom,
    mu = 3.738549*1/Angstrom,
    m = 1,
    n = 1.0,
    QL = -2.0*elementary_charge,
    QU = 6.0*elementary_charge,
    DL = 0.007664*Angstrom,
    DU = -1.213951*Angstrom,
    nB = 10,
    J1 = 6.59963*eV/elementary_charge,
    J2 = 5.955097*eV/elementary_charge**2,
    J3 = 0.760433*eV/elementary_charge**3,
    J4 = 0.009388*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 1.371794*1/Angstrom,
    Z = -1.53917*elementary_charge,
    P = 0.5167863369*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    A = 516.5873248*eV,
    B0 = 117.0421345*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
```

(continues on next page)

(continued from previous page)

```
    l = 2.136011*1/Angstrom,
    mu0 = 1.178831*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 0.545629*1/Angstrom,
    b0 = 0.077183,
    b1 = 0.146606,
    b2 = 0.226674,
    b3 = 0.093054,
    b4 = -0.130433,
    b5 = -0.065549,
    b6 = 0.5504,
    c0 = -0.012318,
    c1 = 0.175079,
    c2 = 0.066085,
    c3 = -0.076109,
    N = 1.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'N',
    A = 7654.972656*eV,
    B0 = 2102.295654*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
    l = 5.218037*1/Angstrom,
    mu0 = 3.738549*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 3.738549*1/Angstrom,
    b0 = 1.691325,
    b1 = 0.283832,
    b2 = 0.698596,
    b3 = 1.724015,
    b4 = 0.812665,
    b5 = 0.0,
    b6 = 0.0,
    c0 = 0.0,
    c1 = 0.0,
    c2 = 0.0,
    c3 = 0.0,
    N = 1.0,
    r_i = 2.0*Angstrom,
    r_cut = 2.3*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'N',
```

(continues on next page)

(continued from previous page)

```
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 2.297286*1/Angstrom,
    b0 = 0.225186,
    b1 = 0.269113,
    b2 = 0.261613,
    b3 = 0.033596,
    b4 = -0.113159,
    b5 = 2.2e-05,
    b6 = 3e-06,
    c0 = -0.032039,
    c1 = -0.082354,
    c2 = 0.024479,
    c3 = -0.073187,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'Ti',
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 5.066036*1/Angstrom,
    b0 = 0.024877,
    b1 = -0.103268,
    b2 = 0.263335,
    b3 = 0.412664,
    b4 = 0.017287,
    b5 = 2.3e-05,
    b6 = 0.0,
    c0 = -0.056517,
    c1 = -0.036331,
    c2 = 0.007137,
    c3 = 0.06462,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
```

(continues on next page)

(continued from previous page)

```
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'N',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'Ti',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'N',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = AngleCorrection6Potential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    particleType3 = 'Ti',
    k0 = 0.0*eV,
    k1 = 0.043561*eV,
    k2 = 0.0*eV,
    k3 = 0.049195*eV,
    k4 = 0.0*eV,
    k5 = 0.0*eV,
    k6 = 0.0*eV,
    costheta0 = 0.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
)
```

(continues on next page)

(continued from previous page)

```
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")

bulk_configuration.setCalculator(calculator)
```

See also ref.comb3common.notes.

COMB3Particle

class COMB3Particle(*particleType*, *l*, *mu*, *m*, *n*, *QL*, *QU*, *DL*, *DU*, *nB*, *J1*, *J2*, *J3*, *J4*, *q0*, *xi*, *Z*, *P*, *uniformCharge=False*)

Constructor of the potential.

Parameters

- **particleType** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).
- **mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **m** (*int*) – Potential parameter.
- **n** (*float*) – Potential parameter.
- **QL** (*PhysicalQuantity of type charge*) – Potential parameter.
- **QU** (*PhysicalQuantity of type charge*) – Potential parameter.
- **DL** (*PhysicalQuantity of type length*) – Potential parameter.
- **DU** (*PhysicalQuantity of type length*) – Potential parameter.
- **nB** (*even int*) – Potential parameter.
- **J1** (*PhysicalQuantity of type energy / charge*) – Potential parameter.
- **J2** (*PhysicalQuantity of type energy / charge**2*) – Potential parameter.
- **J3** (*PhysicalQuantity of type energy / charge**3*) – Potential parameter.
- **J4** (*PhysicalQuantity of type energy / charge**4*) – Potential parameter.
- **q0** (*PhysicalQuantity of type charge*) – Potential parameter.
- **xi** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **Z** (*PhysicalQuantity of type charge*) – Potential parameter.
- **P** (*PhysicalQuantity of type length**3*) – Potential parameter.
- **uniformCharge** (*bool*) – Specifies whether all particles of this type must have the same charge.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB3 potential for a TiN rocksalt crystal by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up a Titanium-Nitride cell

vector_a = [4.235, 0.0, 0.0]*Angstrom
vector_b = [0.0, 4.235, 0.0]*Angstrom
vector_c = [0.0, 0.0, 4.235]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Titanium, Nitrogen, Titanium, Nitrogen, Titanium, Nitrogen,
            Titanium, Nitrogen]

# Define coordinates
fractional_coordinates = [[ 0. ,  0. ,  0. ],
                          [ 0.5,  0.5,  0.5],
                          [ 0.5,  0.5,  0. ],
                          [ 0. ,  0. ,  0.5],
                          [ 0.5,  0. ,  0.5],
                          [ 0. ,  0.5,  0. ],
                          [ 0. ,  0.5,  0.5],
                          [ 0.5,  0. ,  0. ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name = 'COMB_NTi_2014')
```

(continues on next page)

(continued from previous page)

```
potentialSet.addParticleType(ParticleType(
    symbol='N',
    mass=14.0067 * atomic_mass_unit,
    atomicNumber=7
))
potentialSet.addParticleType(ParticleType(
    symbol='Ti',
    mass=47.867 * atomic_mass_unit,
    atomicNumber=22
))

option = COMBCoulombOption(alpha = 0.2*1/Angstrom, r_cut = 11.0*Angstrom)
potentialSet.addOption(option)
potential = COMB3Particle(
    particleType = 'Ti',
    l = 2.136011*1/Angstrom,
    mu = 1.178831*1/Angstrom,
    m = 1,
    n = 0.566048,
    QL = -4.0*elementary_charge,
    QU = 4.0*elementary_charge,
    DL = 0.005*Angstrom,
    DU = -0.5*Angstrom,
    nB = 10,
    J1 = 3.095768*eV/elementary_charge,
    J2 = 4.23028*eV/elementary_charge**2,
    J3 = -1.039759*eV/elementary_charge**3,
    J4 = 0.357428*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 0.724352*1/Angstrom,
    Z = 3.022932*elementary_charge,
    P = 0.335*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3Particle(
    particleType = 'N',
    l = 5.218037*1/Angstrom,
    mu = 3.738549*1/Angstrom,
    m = 1,
    n = 1.0,
    QL = -2.0*elementary_charge,
    QU = 6.0*elementary_charge,
    DL = 0.007664*Angstrom,
    DU = -1.213951*Angstrom,
    nB = 10,
    J1 = 6.59963*eV/elementary_charge,
    J2 = 5.955097*eV/elementary_charge**2,
    J3 = 0.760433*eV/elementary_charge**3,
    J4 = 0.009388*eV/elementary_charge**4,
    q0 = 0.0*elementary_charge,
    xi = 1.371794*1/Angstrom,
```

(continues on next page)

(continued from previous page)

```
    Z = -1.53917*elementary_charge,
    P = 0.5167863369*Angstrom**3,
    uniformCharge = False
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    A = 516.5873248*eV,
    B0 = 117.0421345*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
    l = 2.136011*1/Angstrom,
    mu0 = 1.178831*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 0.545629*1/Angstrom,
    b0 = 0.077183,
    b1 = 0.146606,
    b2 = 0.226674,
    b3 = 0.093054,
    b4 = -0.130433,
    b5 = -0.065549,
    b6 = 0.5504,
    c0 = -0.012318,
    c1 = 0.175079,
    c2 = 0.066085,
    c3 = -0.076109,
    N = 1.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'N',
    A = 7654.972656*eV,
    B0 = 2102.295654*eV,
    B1 = 0.0*eV,
    B2 = 0.0*eV,
    l = 5.218037*1/Angstrom,
    mu0 = 3.738549*1/Angstrom,
    mu1 = 0.0*1/Angstrom,
    mu2 = 0.0*1/Angstrom,
    beta = 3.738549*1/Angstrom,
    b0 = 1.691325,
    b1 = 0.283832,
    b2 = 0.698596,
    b3 = 1.724015,
    b4 = 0.812665,
    b5 = 0.0,
    b6 = 0.0,
```

(continues on next page)

(continued from previous page)

```
    c0 = 0.0,
    c1 = 0.0,
    c2 = 0.0,
    c3 = 0.0,
    N = 1.0,
    r_i = 2.0*Angstrom,
    r_cut = 2.3*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'Ti',
    particleType2 = 'N',
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 2.297286*1/Angstrom,
    b0 = 0.225186,
    b1 = 0.269113,
    b2 = 0.261613,
    b3 = 0.033596,
    b4 = -0.113159,
    b5 = 2.2e-05,
    b6 = 3e-06,
    c0 = -0.032039,
    c1 = -0.082354,
    c2 = 0.024479,
    c3 = -0.073187,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3PairPotential(
    particleType1 = 'N',
    particleType2 = 'Ti',
    A = 1661.765935*eV,
    B0 = 104.805547*eV,
    B1 = 135.992262*eV,
    B2 = 30.112007*eV,
    l = 3.17815*1/Angstrom,
    mu0 = 2.453223*1/Angstrom,
    mu1 = 1.766886*1/Angstrom,
    mu2 = 2.439896*1/Angstrom,
    beta = 5.066036*1/Angstrom,
    b0 = 0.024877,
    b1 = -0.103268,
    b2 = 0.263335,
```

(continues on next page)

(continued from previous page)

```
    b3 = 0.412664,
    b4 = 0.017287,
    b5 = 2.3e-05,
    b6 = 0.0,
    c0 = -0.056517,
    c1 = -0.036331,
    c2 = 0.007137,
    c3 = 0.06462,
    N = 1.0,
    r_i = 2.6*Angstrom,
    r_cut = 3.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'Ti',
    particleType2 = 'N',
    Pchi = 0.0253919125995*Angstrom**2*elementary_charge,
    PJ = 0.0531158828648*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'Ti',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = COMB3FieldCorrection(
    particleType1 = 'N',
    particleType2 = 'N',
    Pchi = 0.0728019460398*Angstrom**2*elementary_charge,
    PJ = 0.271659870233*Angstrom**4,
    r_i = 10.0*Angstrom,
    r_cut = 11.0*Angstrom
)
potentialSet.addPotential(potential)
potential = AngleCorrection6Potential(
    particleType1 = 'Ti',
    particleType2 = 'Ti',
```

(continues on next page)

(continued from previous page)

```
    particleType3 = 'Ti',
    k0 = 0.0*eV,
    k1 = 0.043561*eV,
    k2 = 0.0*eV,
    k3 = 0.049195*eV,
    k4 = 0.0*eV,
    k5 = 0.0*eV,
    k6 = 0.0*eV,
    costheta0 = 0.0,
    r_i = 3.9*Angstrom,
    r_cut = 4.1*Angstrom
)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")

bulk_configuration.setCalculator(calculator)
```

See also ref.comb3common.notes.

COMBCoulombOption

class COMBCoulombOption(*alpha*, *r_cut*)

Constructor of the option.

Parameters

- **alpha** (*PhysicalQuantity of type length**-1*) – Splitting parameter for the Wolf summation that is used internally to deal with long-range interactions.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff for all electrostatic interactions for which no explicit cutoff is given.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

COMBMixitPotential

class COMBMixitPotential(*particleType1*, *particleType2*, *K=None*, *r0=None*)
Constructor of the potential.

Note: COMBParticle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **K** (*float*) – Potential parameter.
- **r0** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()
Return the names of all used parameters as a list.

getAllParameters()
Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()
Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)
Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)
Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
```

(continues on next page)

(continued from previous page)

```
                            [ 0.5303,  0.5303,  0.33333333],
                            [ 0.4135,  0.2669,  0.1191    ],
                            [ 0.2669,  0.4135,  0.547567  ],
                            [ 0.7331,  0.1466,  0.785767  ],
                            [ 0.5865,  0.8534,  0.214233  ],
                            [ 0.8534,  0.5865,  0.452433  ],
                            [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
```

(continues on next page)

(continued from previous page)

```
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
```

(continues on next page)

(continued from previous page)

```
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBOptimizationOption

class COMBOptimizationOption(*maxIterations=None*, *relFTol=None*, *gradTol=None*, *linEqTol=None*, *disableOptimization=None*)

Constructor of the option.

Parameters

- **maxIterations** (*int*) – The maximal number of iterations that are allowed during the optimization of the COMB potential.
- **relFTol** (*float*) – Stop the optimization of the COMB potential if the relative change in the objective function value is smaller than this value. If set to < 0, this stopping criterion is disabled.
- **gradTol** (*float*) – Stop the optimization of the COMB potential if the 2-norm of the gradient of the objective function is smaller than this value. If set to < 0, this stopping criterion is disabled.

- **linEqTol** (*float*) – The tolerance that is passed to the linear equation solver that is called when dipoles are calculated.
- **disableOptimization** (*bool*) – If set to True, the calculation of the dynamical charges in the COMB potential is disabled. Instead, the charges of the corresponding particle types are used.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a COMB potential and add an optimization option to stop the internal optimization of the dynamical charges after 20 iterations.

```
# Set up a new COMB potential.
comb_potential = COMB_OSi_2007()

# Set up an optimization option to stop the optimization after 20 iterations.
comb_optimization_option = COMBOptimizationOption(
    maxIterations=20,
)

# Add the option to the potential.
comb_potential.addOption(comb_optimization_option)
```

Notes

This option can be used to specify the details of the internal optimization of the dynamic charges in a COMB potential. It can be added to a *TremoloXPotentialSet* via the `addOption()` method.

COMBOverCoordinationCorrection

class COMBOverCoordinationCorrection

Currently unsupported.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))
```

(continues on next page)

(continued from previous page)

```
# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
```

(continues on next page)

(continued from previous page)

```
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBPairPotential

class COMBPairPotential(*particleType1*, *particleType2*, *A*, *B*, *R*, *S*, *l*, *mu*, *K=None*, *r0=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type energy*) – Potential parameter.
- **R** (*PhysicalQuantity of type length*) – Potential parameter.
- **S** (*PhysicalQuantity of type length*) – Potential parameter.
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).
- **mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **K** (*float*) – Potential parameter.
- **r0** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBParticle(particleType='Si',
                         A = 1830.81*eV,
                         B = 471.17*eV,
                         R = 2.6*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                          S = 3.0*Angstrom,
                          l = 2.4799*1/Angstrom,
                          mu = 1.7322*1/Angstrom,
                          beta = 1.0999e-06,
                          m = 3,
                          n = 0.78734,
                          c = 100390.0,
                          d = 16.218,
                          h = -0.59826,
                          QL = -4.0,
                          QU = 4.0,
                          DL = 1.7982*Angstrom,
                          DU = -1.7094*Angstrom,
                          nB = 10,
                          J1 = -0.3*eV/elementary_charge,
                          J2 = -2.2*eV/elementary_charge**2,
                          J3 = -1.25*eV/elementary_charge**3,
                          J4 = 2.53*eV/elementary_charge**4,
                          q0 = 0.6112*elementary_charge,
                          uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBParticle(particleType = 'O',
                          A = 3326.7*eV,
                          B = 260.89*eV,
                          R = 2.6*Angstrom,
                          S = 3.0*Angstrom,
                          l = 5.36*1/Angstrom,
                          mu = 2.68*1/Angstrom,
                          beta = 2.0,
                          m = 1,
                          n = 1.0,
                          c = 6.6,
                          d = 1,
                          h = -0.229,
                          QL = -1.8349,
                          QU = 5.5046,
                          DL = 0.00148*Angstrom,
                          DU = -0.00112*Angstrom,
                          nB = 10,
                          J1 = 12.006*eV/elementary_charge,
                          J2 = 10.5205*eV/elementary_charge**2,
                          J3 = 0.0*eV/elementary_charge**3,
                          J4 = 0.0*eV/elementary_charge**4,
                          q0 = 0.0*elementary_charge,
                          uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
```

(continues on next page)

(continued from previous page)

```
                                K = 20.0,
                                r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBPointWiseCoulomb(particleType1 = 'Si',
                                 particleType2 = 'Si',
                                 R = 1.1*Angstrom,
                                 S = 5.45*Angstrom,
                                 eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBPointWiseCoulomb(particleType1 = 'Si',
                                 particleType2 = 'O',
                                 R = 1.1*Angstrom,
                                 S = 5.45*Angstrom,
                                 eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBPointWiseCoulomb(particleType1 = 'O',
                                 particleType2 = 'O',
                                 R = 1.1*Angstrom,
                                 S = 5.45*Angstrom,
                                 eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBParticle

class COMBParticle(*particleType*, *A*, *B*, *R*, *S*, *l*, *mu*, *beta*, *m*, *n*, *c*, *d*, *h*, *QL*, *QU*, *DL*, *DU*, *nB*, *J1*, *J2*, *J3*, *J4*, *q0*, *uniformCharge=False*)

Constructor of the potential.

Parameters

•**particleType** (*ParticleType or ParticleIdentifier*) – Identifier of the particle type

•**A** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B** (*PhysicalQuantity of type energy*) – Potential parameter.

•**R** (*PhysicalQuantity of type length*) – Potential parameter.

•**S** (*PhysicalQuantity of type length*) – Potential parameter.

•**l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).

•**mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**beta** (*float*) – Potential parameter.

•**m** (*int*) – Potential parameter.

•**n** (*float*) – Potential parameter.

•**c** (*float*) – Potential parameter.

•**d** (*float*) – Potential parameter.

•**h** (*float*) – Potential parameter.

•**QL** (*PhysicalQuantity of type charge*) – Potential parameter.

•**QU** (*PhysicalQuantity of type charge*) – Potential parameter.

•**DL** (*PhysicalQuantity of type length*) – Potential parameter.

•**DU** (*PhysicalQuantity of type length*) – Potential parameter.

•**nB** (*even int*) – Potential parameter.

•**J1** (*PhysicalQuantity of type energy / charge*) – Potential parameter.

•**J2** (*PhysicalQuantity of type energy / charge**2*) – Potential parameter.

•**J3** (*PhysicalQuantity of type energy / charge**3*) – Potential parameter.

•**J4** (*PhysicalQuantity of type energy / charge**4*) – Potential parameter.

•**q0** (*PhysicalQuantity of type charge*) – Potential parameter.

•**uniformCharge** (*bool*) – Specifies whether all particles of this type must have the same charge.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
```

(continues on next page)

(continued from previous page)

```
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBPointWiseCoulomb

class COMBPointWiseCoulomb(*particleType1*, *particleType2*, *R*, *S*, *eta*)

Constructor of the potential.

Note: COMBParticle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **R** (*PhysicalQuantity of type length*) – Potential parameter.
- **S** (*PhysicalQuantity of type length*) – Potential parameter.
- **eta** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
```

(continues on next page)

(continued from previous page)

```
                                K = 20.0,
                                r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBSMCoulomb

class COMBSMCoulomb(*particleType1*, *particleType2*, *xii*, *xij*)

Constructor of the potential.

Note: COMBParticle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **xii** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **xij** (*PhysicalQuantity of type length**-1*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
```

(continues on next page)

(continued from previous page)

```
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
```

(continues on next page)

(continued from previous page)

```
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBSelfEnergyCorrection

class COMBSelfEnergyCorrection(*particleType1*, *particleType2*, *a*, *b*, *r_cut=None*)

Constructor of the potential.

Note: COMBParticle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of length**2 / charge*) – Potential parameter.

- **b** (*PhysicalQuantity of type length**4 / charge**2*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
```

(continues on next page)

(continued from previous page)

```
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# -----------------------------------------------------------------
# Calculator
# -----------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
```

(continues on next page)

(continued from previous page)

```
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBSelfEnergyCorrectionShan

class COMBSelfEnergyCorrectionShan(*particleType1*, *particleType2*, *a*, *b*, *r_cut=None*)

Constructor of the potential.

Note: COMBParticle must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of length**4 * charge*) – Potential parameter.
- **b** (*PhysicalQuantity of type length**4*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
```

(continues on next page)

(continued from previous page)

```
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
```

(continues on next page)

(continued from previous page)

```
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

COMBTriplePotential

class COMBTriplePotential(*particleType1*, *particleType2*, *particleType3*, *cosTheta*, *K0=None*, *K1=None*, *K2=None*, *K3=None*, *K4=None*, *K5=None*, *K6=None*)

Constructor of the potential.

Note: COMBParticle must be added for ALL THREE particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type. This particle type corresponds to the central particle in the calculation of the angle (the vertex).
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **cosTheta** (*float*) – Potential parameter (cosine of the equilibrium angle).
- **K0** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K1** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K2** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K3** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K4** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K5** (*PhysicalQuantity of type energy*) – Potential parameter.
- **K6** (*PhysicalQuantity of type energy*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a COMB potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='COMB_SiO')

# Add particle types for Si and O.
potentialSet.addParticleType(ParticleType(symbol='Si',
```

(continues on next page)

(continued from previous page)

```
                                          mass=28.0855*atomic_mass_unit,
                                          atomicNumber=14))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          atomicNumber=8))

# Add COMB potential for Si.
potential = COMBSingleTypePotential(particleType='Si',
                                    A = 1830.81*eV,
                                    B = 471.17*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 2.4799*1/Angstrom,
                                    mu = 1.7322*1/Angstrom,
                                    beta = 1.0999e-06,
                                    m = 3,
                                    n = 0.78734,
                                    c = 100390.0,
                                    d = 16.218,
                                    h = -0.59826,
                                    QL = -4.0,
                                    QU = 4.0,
                                    DL = 1.7982*Angstrom,
                                    DU = -1.7094*Angstrom,
                                    nB = 10,
                                    J1 = -0.3*eV/elementary_charge,
                                    J2 = -2.2*eV/elementary_charge**2,
                                    J3 = -1.25*eV/elementary_charge**3,
                                    J4 = 2.53*eV/elementary_charge**4,
                                    q0 = 0.6112*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Add COMB potential for O.
potential = COMBSingleTypePotential(particleType = 'O',
                                    A = 3326.7*eV,
                                    B = 260.89*eV,
                                    R = 2.6*Angstrom,
                                    S = 3.0*Angstrom,
                                    l = 5.36*1/Angstrom,
                                    mu = 2.68*1/Angstrom,
                                    beta = 2.0,
                                    m = 1,
                                    n = 1.0,
                                    c = 6.6,
                                    d = 1,
                                    h = -0.229,
                                    QL = -1.8349,
                                    QU = 5.5046,
                                    DL = 0.00148*Angstrom,
                                    DU = -0.00112*Angstrom,
                                    nB = 10,
```

(continues on next page)

(continued from previous page)

```
                                    J1 = 12.006*eV/elementary_charge,
                                    J2 = 10.5205*eV/elementary_charge**2,
                                    J3 = 0.0*eV/elementary_charge**3,
                                    J4 = 0.0*eV/elementary_charge**4,
                                    q0 = 0.0*elementary_charge,
                                    uniformCharge = False)
potentialSet.addPotential(potential)

# Combination rules between Si and O.
potential = COMBMixitPotential(particleType1 = 'O',
                               particleType2 = 'Si',
                               K = 20.0,
                               r0 = 1.65*Angstrom)
potentialSet.addPotential(potential)

# Three-body-terms.
potential = COMBTriplePotential(particleType1 = 'Si',
                                particleType2 = 'O',
                                particleType3 = 'O',
                                cosTheta = -0.333313248,
                                K0 = 3.16666666667*eV,
                                K1 = 0.0*eV,
                                K2 = 6.33333333333*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)
potential = COMBTriplePotential(particleType1 = 'O',
                                particleType2 = 'Si',
                                particleType3 = 'Si',
                                cosTheta = -0.806238149,
                                K0 = 1.5*eV,
                                K1 = 0.0*eV,
                                K2 = 3.0*eV,
                                K3 = 0.0*eV,
                                K4 = 0.0*eV,
                                K5 = 0.0*eV,
                                K6 = 0.0*eV)
potentialSet.addPotential(potential)

# Coulomb interactions.
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'Si',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.380689)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'Si',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                        eta = 0.2817839)
potentialSet.addPotential(potential)
potential = COMBCoulomb(particleType1 = 'O',
                        particleType2 = 'O',
                        R = 1.1*Angstrom,
                        S = 5.45*Angstrom,
                        eta = 0.20857489)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.combcommon.notes.

CalculatedChemicalPotential

class CalculatedChemicalPotential(*element*, *calculator*, *phonon_calculator=None*, *configuration=None*, *k_points=None*, *include_vibrations=None*, *repetitions=None*, *reference_temperature=None*)

A class defining the chemical potential for an atomic species. Used as part of a *ChargedPointDefect* study.

Parameters

- **element** (*PeriodicTableElement*) – The element for which the atomic chemical potential is defined.
- **calculator** (`All calculators`) – The calculator used to calculate the reference thermodynamics.
- **phonon_calculator** (`All calculators` | None) – The calculator used to calculate the vibrational energies. If not specified the reference calculator is used. The argument `include_vibrations` must also be `True` to include vibrational energies.
- **configuration** (*BulkConfiguration*) – The thermodynamic reference configuration for the element. **Default:** The reference configuration is taken from an elemental reference material. In that case an optimization is automatically performed.
- **k_points** (*KpointDensity* | *MonkhorstPackGrid* | None) – k-points used in the calculator for calculator. This may need to be set in cases where the reference material is a conductor, and therefore requires a higher number of k-points.
- **optimization_parameters** (*OptimizeGeometryParameters*) – The parameters to use to optimize the reference configuration.
- **include_vibrations** (*bool*) – Whether to include vibrational energies in the atomic free energy. The configuration will be optimized with the phonon calculator before calculating vibrations. **Default:** False.
- **repetitions** (*list of ints | None*) – The number of repetitions of the system in the A, B, and C-directions used in determining the vibrational frequencies. If not given repetitions are selected so that the cell is large enough to calculate the vibrations using a finite difference method.
- **reference_temperature** (*PhysicalQuantity* of type temperature) – The temperature used to calculate the free energy. **Default:** 300 Kelvin.

calculator()

Returns
The calculator used to calculate the chemical potential.

Return type
`All calculators`

chemicalPotential(*temperature=None*)

Calculate the chemical potential.

Returns
The chemical potential of the atom calculated from the internal energy and entropy per atom of the specified element in some reference configuration.

Return type
PhysicalQuantity of type energy | None

electronicInternalEnergy()

Returns the electronic internal energy.

Returns
The electronic energy per atom of the specified element in some reference configuration, or None if it has not been evaluated.

Return type
PhysicalQuantity of type energy | None

element()

Returns
The element for which the atomic chemical potential is defined.

Return type
PeriodicTableElement

entropy(*temperature=None*)

Returns the entropy from vibrational corrections, if included. If not included then the entropy is zero. If the object has not been updated then None is returned.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature at which the vibrational corrections are calculated.

Returns
The entropy component of the atomic chemical potential or None if it has not been evaluated.

Return type
PhysicalQuantity of type energy per temperature | None

includeVibrations()

Returns
Whether vibrational terms are included in the chemical potential.

Return type
bool

phononCalculator()

Returns
The calculator used to calculate the chemical potential.

Return type
`All calculators`

referenceConfiguration()

Returns
The reference configuration from which the components of the atomic chemical potential are calculated.

Return type
BulkConfiguration | None

referenceTemperature()

Returns
The reference temperature for the chemical potential.

Return type
PhysicalQuantity of type temperature.

repetitions()

Returns
The number of repetitions of the system in the A, B, and C-directions.

Return type
list of ints | None

uniqueString()

Return a unique string representing the state of the object.

update()

Perform the calculation to add data to the object.

vibrationalInternalEnergy(*temperature=None*)

Returns the internal energy from vibrations.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature at which the vibrational corrections are calculated.

Returns
The energy per atom of the specified element in some reference configuration, or None if it has not been evaluated.

Return type
PhysicalQuantity of type energy | None

Notes

The *CalculatedChemicalPotential* object defines an atomic chemical potential used in a *ChargedPointDefectConfiguration* calculation. The *CalculatedChemicalPotential* provides the reservoir energy for the element being added or removed from the pristine material to form the defect. This energy is used in calculating the formation energy and concentration of the defect. The value of the reservoir energy can depend on the nature of the environment around the defect. The atomic chemical potential also depends on the reference calculator used in the calculation, as it also includes the energy of the isolated atom according to the specific calculator.

The *CalculatedChemicalPotential* allows calculating the atomic chemical potential from an appropriate elemental reference configuration. The *CalculatedChemicalPotential* is created by supplying an element and the calculator to be used for calculating the elemental energy. A reference configuration can also be given, using the argument `configuration`. If a configuration is not given a standard reference structure is used. The k-points used in the energy calculation can be expressly set using the `k_points` argument. This is useful in cases where a different k-point sampling is required for the elemental reference material e.g. metals. Vibrational energies can be included by setting the argument `include_vibrations` to `True`. A specific calculator for the vibrations can be given with the argument `phonon_calculator`, otherwise the energy calculator is also used for the vibrations. The repetitions used in the vibration calculation can also be set with the `repetitions` argument.

ChargedPointDefect

class ChargedPointDefect(*bulk_configuration*, *filename*, *object_id*, *formation_energy_calculator=None*, *relaxation_calculator=None*, *band_gap_calculator=None*, *point_defect=None*, *charge_states=None*, *atomic_chemical_potentials=None*, *supercell_repetitions_list=None*, *relax_atomic_coordinates=None*, *dielectric_constant=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*, *elastic_correction_method=None*, *optimize_geometry_parameters=None*, *model_charge_correction=None*, *random_seed=None*, *resume=None*, *use_ghost=None*, *calculate_defect_symmetry=None*)

Constructor for the ChargedPointDefect object.

Parameters

- **bulk_configuration** (*BulkConfiguration*) – The reference bulk unit cell configuration to be used to generate the supercell configurations.
- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **formation_energy_calculator** (*Calculator*) – The calculator used for calculating the formation energy of the defect charge states. All total energies, band positions and finite-size corrections are calculated using this calculator. The calculator must contain a basis set for the elements in both the defect and pristine configurations. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `relaxation_calculator`. At least one of the two calculators must be provided.
- **relaxation_calculator** (*Calculator*) – The calculator used for relaxing the atomic coordinates of the defect supercells. The calculator must contain a basis set for the elements in the defect configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `formation_energy_calculator`. At least one of the two calculators must be provided.
- **band_gap_calculator** – The calculator used for calculating the band gap in which the transition levels will be placed. The calculator is only used for the pristine configuration, so only needs to contain a basis set for the elements in this configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `formation_energy_calculator`.
- **point_defect** (*Vacancy* | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*) – The point defect to embed in the host supercell. **Default:** `Vacancy()`
- **charge_states** – The list of charge states of the defect, as a discrete multiple of elementary charge. A charge of `-1` corresponds to one extra electron. **Default:** `0`

- **atomic_chemical_potentials** (sequence of *AtomicChemicalPotential*) – The chemical potentials for the atomic species involved in the creation of the defect (i.e., any species for which the number of atoms changes between the pristine and defect supercell). **Default:** All necessary atomic chemical potentials will be calculated using the pristine configuration (when possible) or a default elemental reference configuration.
- **supercell_repetitions_list** (*sequence (size 3) of int | sequence of sequence (size 3) of int*) – The list of supercells of the bulk unit cell to calculate, each given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. **Default:** `[(1, 1, 1)]`
- **relax_atomic_coordinates** (*bool*) – Whether to relax the atomic coordinates of the defect supercells. **Default:** `True`
- **dielectric_constant** (*float*) – The dielectric constant of the host material. Only needed to calculate the finite-size correction terms. **Default:** `1.0`
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of charged point defect calculations. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'chargedpointdefect_'`
- **number_of_processes_per_task** (int | *ProcessesPerNode* | `AllProcesses`) – The number of processes that will be used to execute each task. If this value is greater than or equal to the total number of available processes, each single task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. **Default:** `AllProcesses`, all available processes execute each task collaboratively.
- **elastic_correction_method** (*ImageStressCorrection*) – Specify a finite size elastic energy correction. If not specified, no finite size elastic energy correction will be applied. **Default:** None
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters to use for optimizing the geometry. Note that `pre_step_hook` and `post_step_hook` must be left unset. Also note that `max_stress`, `target_stress`, `restart_strategy` and `enable_optimization_stop_file` do not have any effect since stress is not optimized, and the restart mechanism within *OptimizeGeometry()* is disabled. Only constraints of type *FixAtomConstraints* and *RigidBody* will be maintained, others will be discarded. **Default:** `OptimizeGeometryParameters()`
- **model_charge_correction** (*ModelChargeCorrection*) – Specify a finite size model charge correction. **Default:** ModelChargeCorrection()
- **random_seed** (*int | None*) – The random seed to rattle atomic coordinates. `None` for the system to pick different ones for you. **Default:** None
- **resume** (*bool*) – Whether rerunning the simulation will resume non converged results. **Default:** False
- **use_ghost** (*bool*) – Whether ghost atoms will be used (for LCAO calculators only). **Default:** True
- **calculate_defect_symmetry** (*bool*) – Whether to calculate the symmetry of the defect. **Default:** `False`

atomNumberDifferences()

Returns
The atom number differences for each element.

Return type
dict of type {*PeriodicTableElement* : int}

atomicChemicalPotentials()

Returns
The atomic chemical potential for all the required elements. If an element for which the atomic chemical potential needs to be calculated is not present, it is calculated from the pristine cell at each supercell size.

Return type
list of *AtomicChemicalPotential*

bandShift(*supercell_repetitions*, *charge_state*)

Retrieve the band shift for a given defect supercell with respect to the pristine supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The band shift for the supercell. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

calculateDefectSymmetry()

Returns
Whether the defect symmetry is calculated.

Return type
bool

calculatedChargeStates(*supercell_repetitions=None*)

Retrieve the list of calculated charge states, either for the entire study or for a given supercell.

Parameters
supercell_repetitions (sequence (size 3) of int | *ExtrapolationScheme*) – The supercell given as the number of repetitions of the bulk unit cell along the (a, b, c) directions for which to return the list of calculated charge states. For an extrapolation scheme, the charge states for all supercells are returned. **Default:** All supercells.

Returns
The list of calculated charge states. The charge states are ordered from the most positive to the most negative.

Return type
list of int

calculatedSupercells(*charge_state=None*)

Retrieve the list of calculated supercells, either for the entire study or for a given charge state.

Parameters
charge_state (*int*) – The charge state for which to return the list of calculated supercells. **Default:** All charge states.

Returns
The list of calculated supercells, each given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. The supercells are ordered from the smallest to the largest in terms of the number of atoms; if multiple supercells have the same number of atoms they are ordered according to the repetitions in the (a, b, c) directions, in that order.

Return type
list of tuple (size 3) of int

chargeStates()

Returns
The list of all calculated and not calculated charge states. Any charge state which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of int

conductionBandMinimum(*supercell_repetitions*)

Retrieve the conduction band minimum of the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The conduction band minimum of the pristine unit cell configuration. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

defectConfiguration(*supercell_repetitions*, *charge_state*, *symmetry_index=None*, *unit_cell_translation=None*)

Retrieve the defect supercell configuration for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.
- **symmetry_index** (*int*) – The index of the symmetry operation for the pristine unit cell to apply to the defect supercell configuration. Note that this can only be used if the calculate_defect_symmetry option has been enabled for the calculation. **Default:** No transformation (i.e., the identity operation)
- **unit_cell_translation** (*sequence (size 3) of int*) – An additional translation to apply in terms of the unit cell vectors. **Default:** No translation

Returns
The defect supercell configuration. If not available, returns None.

Return type
BulkConfiguration | None

defectConfigurationCalculator(*supercell_repetitions*, *charge_state*)

Retrieve the calculator of the defect supercell configuration for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The calculator of the defect supercell configuration. If not available, returns None.

Return type
Calculator | None

defectSymmetry(*supercell_repetitions*, *charge_state*)

Calculate the symmetry information for the defect supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The symmetry information for the defect supercell. If not available, returns None.

Return type
dict | None

defectTotalEnergy(*supercell_repetitions*, *charge_state*)

Retrieve the total energy of the defect supercell configuration for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The total energy of the defect supercell configuration. If not available, returns None.

Return type
TotalEnergy | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

dielectricConstant()

Returns
The dielectric constant of the host material is updated.

Return type
float

elasticCorrectionEnergy(*supercell_repetitions*, *charge_state*)

Retrieve the value of the elastic energy correction for a given supercell. This result will only be available after the calculation ran successfully. If no correction was applied, None is returned.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The image stress correction. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

elasticCorrectionMethod()

Returns
The method used to evaluate the finite size elastic energy correction.

Return type
None | *ImageStressCorrection*

filename()

Returns
The filename where the study object is stored.

Return type
str

formationEnergy(*charge_state*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *supercell_repetitions=None*, *enable_finite_size_corrections=None*)

The formation energy for a given charge state q of the defect, defined as

$E_f^q = E^{\mathrm{def},q} - E^{\mathrm{bulk},q} - \sum_i \Delta n_i \mu_i + q\left(\mu_e^{\mathrm{ref}} + \Delta\mu_e\right)$.

The first two terms are the total energy of the supercell with and without the defect, respectively. Δn_i is the difference in the number of atoms between the two supercells for species i, and μ_i is the atomic chemical potential for that species. μ_e^{ref} is the absolute position of the reference level for the electronic chemical potential (either the valence band maximum or the conduction band minimum), and $\Delta\mu_e$ is the electronic chemical potential relative to it. Note that correction terms have not been included here (see Notes below for details).

Parameters

- **charge_state** (*int*) – The charge state of the defect.
- **electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to electronic_chemical_potential_reference. **Default:** 0.0 * eV
- **electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge
- **defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator; see Notes below for details. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect
- **supercell_repetitions** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.
- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True

Returns
The calculated formation energy.

Return type
PhysicalQuantity of type energy

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

modelChargeCorrection(*supercell_repetitions*, *charge_state*)

Retrieve the model charge correction parameters for the FNV correction for a given supercell and charge state.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **charge_state** (*int*) – The charge state of the defect.

Returns
The parameters used to define the model charge correction. If not available, returns None.

Return type
ModelChargeCorrection | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometryParameters()

Returns
Parameters required to optimize the geometry.

Return type
OptimizeGeometryParameters

periodicChargeCorrection(*supercell_repetitions*, *charge_state*)

Retrieve the periodic charge correction for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The periodic charge correction for the supercell. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

pointDefect()

Returns
The point defect associated with the charged point defect study.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

pristineConfiguration(*supercell_repetitions*)

Retrieve the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The pristine unit cell configuration. If not available, returns None.

Return type
BulkConfiguration | None

pristineConfigurationCalculator(*supercell_repetitions*)

Retrieve the calculator of the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The calculator of the pristine unit cell configuration. If not available, returns None.

Return type
Calculator | None

pristineFermiLevel(*supercell_repetitions*)

Retrieve the Fermi level of the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The Fermi level of the pristine unit cell configuration. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

pristineSymmetryData()

Retrieve the symmetry information for the pristine unit cell.

This result will only be available after the calculation ran successfully.

Returns
The symmetry information for the pristine unit cell. If not available, returns None.

Return type
dict | None

pristineTotalEnergy(*supercell_repetitions*)

Retrieve the total energy of the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The total energy of the pristine unit cell configuration. If not available, returns None.

Return type
TotalEnergy | None

randomSeed()

Returns
The random seed used for rattling coordinates

Return type
None | int

relaxAtomicCoordinates()

Returns
Whether the atomic coordinates of the defect supercells are relaxed.

Return type
bool

resume()

Returns
whether the resume option was specified

Return type
bool

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

setModelChargeCorrection(*supercell_repetitions*, *charge_state*, *model_charge_correction=None*)

Set the model charge correction parameters for the FNV correction for a given supercell and charge state.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **charge_state** (*int*) – The charge state of the defect.
- **model_charge_correction** (*ModelChargeCorrection*) – The parameters used to define the model charge correction. **Default:** `ModelChargeCorrection()`

stableChargeState(*electronic_chemical_potential*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *supercell_repetitions=None*, *enable_finite_size_corrections=None*)

The stable charge state at a given electronic chemical potential.

Parameters

- **electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to `electronic_chemical_potential_reference`.
- **electronic_chemical_potential_reference** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`
- **defect_type** (`DeepLevelDefect | ShallowAcceptor | ShallowDonor`) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with `band_gap_calculator`; see Notes below for details. Note that this parameter only has an effect if `band_gap_calculator` is different from `formation_energy_calculator`. **Default:** `DeepLevelDefect`
- **supercell_repetitions** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.
- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** `True`

Returns
The calculated stable charge state.

Return type
int

stableTransitions(*electronic_chemical_potential_reference=None*, *defect_type=None*, *supercell_repetitions=None*, *enable_finite_size_corrections=None*)

The list of transitions between stable charge states across the whole range of the electronic chemical potential. The transition levels are given relative to `electronic_chemical_potential_reference`.

Parameters

- **electronic_chemical_potential_reference** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`
- **defect_type** (`DeepLevelDefect | ShallowAcceptor | ShallowDonor`) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with `band_gap_calculator`; see Notes below for details. Note that this parameter only has an effect if `band_gap_calculator` is different from `formation_energy_calculator`. **Default:** `DeepLevelDefect`
- **supercell_repetitions** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.

- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** `True`

Returns
The list of transitions between stable charge states in order as the electronic chemical potential is increased.

Return type
list of list (size 2) of int

supercellRepetitionsList()

Returns
The list of all calculated and not calculated supercells of the bulk unit cell, each given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. Any supercell which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of tuple (size 3) of int

supercellSymmetryData(*repetitions*)

Retrieve the symmetry information for the pristine unit cell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int | None*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If None returns the symmetry of the pristine unit cell.

Returns
The symmetry information for the pristine unit cell. If not available, returns None.

Return type
dict | None

symmetricDefect(*supercell_repetitions*, *charge_state*, *symmetry_index=None*, *unit_cell_translation=None*)

Retrieve the defect supercell configuration for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.
- **symmetry_index** (*int*) – The index of the symmetry operation for the pristine unit cell to apply to the defect supercell configuration. Note that this can only be used if the calculate_defect_symmetry option has been enabled for the calculation. **Default:** No transformation (i.e., the identity operation)
- **unit_cell_translation** (*sequence (size 3) of int*) – An additional translation to apply in terms of the unit cell vectors. **Default:** No translation

Returns
The defect supercell configuration. If not available, returns None.

Return type
BulkConfiguration | None

tasksFinished()

Returns
finished task, unfinished tasks

Return type
tuple of list of str, list of str

transitionLevel(*charge_states*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *supercell_repetitions=None*, *enable_finite_size_corrections=None*)

The transition level between two calculated charge states m and n, defined as

$E\left(m/n\right) = \frac{E_f^n(\Delta\mu_e=0) - E_f^m(\Delta\mu_e=0)}{m-n}$.

The transition level is given relative to `electronic_chemical_potential_reference`.

Parameters

- **charge_states** (*sequence (size 2) of int*) – The two charge states to calculate the transition level for.
- **electronic_chemical_potential_reference** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`
- **defect_type** (`DeepLevelDefect | ShallowAcceptor | ShallowDonor`) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with `band_gap_calculator`; see Notes below for details. Note that this parameter only has an effect if `band_gap_calculator` is different from `formation_energy_calculator`. **Default:** `DeepLevelDefect`
- **supercell_repetitions** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.
- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** `True`

Returns
The calculated transition level.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

useGhost()

Returns
Whether ghost atoms are used (for LCAO calculations involving vacancies)

Return type
bool

valenceBandMaximum(*supercell_repetitions*)

Retrieve the valence band maximum of the pristine unit cell configuration associated with a given supercell.

This result will only be available after the calculation ran successfully.

Parameters
supercell_repetitions (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

Returns
The valence band maximum of the pristine unit cell configuration. If not available, returns None.

Return type
PhysicalQuantity of type energy | None

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *ChargedPointDefect* object can be used to perform a detailed study of a single point defect in bulk material in its various charge states. The defect can be a simple vacancy, substitutional, or interstitial, or a more complex defect cluster made from multiple simple defects.

The main results are the formation energy of the defect for different charge states as a function of the electronic chemical potential, and the corresponding stable charge transition levels (sometimes referred to as the thermodynamic transition levels, occupancy levels, or trap levels).

Performing accurate simulations of point defects, especially for non-neutral charge states, can be challenging. In particular, it is generally very important to ensure that the physical quantities of interest are properly converged with respect to the size of the supercell of bulk crystal in which the point defect is placed (as periodic boundary conditions are used, there will be spurious interactions between periodic replicas of the defect site). *ChargedPointDefect* aims to assist the user in automatically simulating a range of supercell sizes and analyzing the convergence with size for each charge state. It will also apply the correction by Freysoldt, Neugebauer and Van De Walle (FNV)[1] to mitigate the effects of these finite size interactions.

The object is initialized by specifying the primitive unit cell of the host crystal (without the defect) as a *BulkConfiguration*, the list of supercells of this primitive cell to generate for the study, and the defect to study as an instance of the desired defect class (`Vacancy` | `Substitutional` | `Interstitial` | `DefectCluster` | `SplitInterstitial`). The defect will be inserted into each supercell of the bulk material. The list of charge states of the defect to simulate must also be given.

The calculator which is used to calculate the formation energy is given by `formation_energy_calculator`. The atomic coordinates of the defect supercells are relaxed by default (this is controlled with `relax_atomic_coordinates`); it is possible to specify a separate calculator to perform the relaxations with `relaxation_calculator`, as these usually represent the most computationally intensive part of the study. Both calculators are automatically adjusted for each supercell to ensure a consistent level of accuracy.

One can optionally calculate an elastic strain correction to the formation energy by using a *ImageStressCorrection* object as the `elastic_correction_method`. The correction is described is more detail in Section *Finite-size scaling and correction terms*.

Finally, it is possible to specify a separate calculator to calculate the band gap of the host material with `band_gap_calculator`, in order to overcome the underestimation of the band gap from standard DFT functionals.

Note: The calculators must contain a basis/pseudopotential for all species in both the pristine and defect supercells.

[1] Christoph Freysoldt, Jörg Neugebauer, and ChrisăG. Van De Walle. Fully Ab Initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, 2009. doi:10.1103/PhysRevLett.102.016402.

Defining the point defect

This is specified by `point_defect`. The defect is defined in an abstract way, independently of the material it is embedded in or the supercell size. The study will combine it with the reference unit cell of the pristine material (`bulk_configuration`) and the various supercell sizes (`supercell_repetitions_list`) to create the defect supercells.

There are three simple point defect types: *Vacancy*, *Substitutional*, and *Interstitial*. The first two require a `site_index` at which to create the defect; the index refers to the atom index of the list of atoms of the reference unit cell. The third, instead, is not tied to a site but is defined by specifying a position in space (either in absolute coordinates or as fractional coordinates of the reference unit cell).

More complex defects can also be specified by using a *DefectCluster*; this is defined by providing a list of simple defects which will be combined together to form a composite defect. Examples of this include a divacancy (2 vacancies) or a Frenkel pair (vacancy + interstitial). Larger clusters can also be defined.

Although a *DefectCluster* can still be considered a point defect (it is not periodic in any direction), it has a size depending on the largest distance between any of the constituent defects. It is therefore possible for a cluster to be larger than the reference unit cell. In order to define such clusters, one can make use of the `unit_cell_index` argument for *Vacancy* and *Substitutional*, which allows for the constituent defects to be shifted to different unit cells. Similarly, *Interstitial* defects can be defined by giving a position outside of the unit cell. When the study generates the actual defect supercell configuration, the constituent defects are wrapped within the supercell. Therefore, all supercells used in the study must be large enough to accommodate the cluster.

Atomic Chemical Potentials

The chemical potentials for the atomic species involved in the creation of the defect (i.e., any species for which the number of atoms changes between the pristine and defect supercell) need to be determined in order to calculate the formation energy. Note that the atomic chemical potentials are only used for the final post-processing calculation of the formation energy; although the absolute value of the formation energy is affected, the differences between charge states (and, hence, the transition levels) are not.

The study will automatically try to make a reasonable choice for the values of the atomic chemical potentials. The default choice taken by the study for each element is as follows:

- If possible, the atomic chemical potential is calculated as the energy per atom of the pristine configuration at each supercell size. This will be the case if (a) the element is present in the pristine configuration, and (b) all the elements of the pristine configuration are either added or removed in their stoichiometric ratio in the creation of the defect.
- Otherwise, the atomic chemical potential is calculated as the energy per atom of a default elemental reference configuration. This default configuration is taken from the Δ-test[2] suite of elemental crystals. Note that the calculation is performed with the same calculator parameters as specified in `formation_energy_calculator`.

These automatic values can, however, be changed for one or more elements by providing an *AtomicChemicalPotential* object for each element in `atomic_chemical_potentials`. This allows the user to provide either an explicit hard-coded value of the energy per atom, or an alternative elemental reference configuration for the study to use in calculating the atomic chemical potential. In compound materials the class *AtomicChemicalPotentialList* can be used to compute heat of formation or rich conditions.

[2] Kurt Lejaeghere, Gustav Bihlmayer, Torbjorn Bjorkman, Peter Blaha, Stefan Blugel, Volker Blum, Damien Caliste, IvanoăE. Castelli, StewartăJ. Clark, Andrea DalăCorso, Stefano deăGironcoli, Thierry Deutsch, JohnăKay Dewhurst, Igor DiăMarco, Claudia Draxl, Marcin Dulak, Olle Eriksson, JoseăA. Flores-Livas, KevinăF. Garrity, Luigi Genovese, Paolo Giannozzi, Matteo Giantomassi, Stefan Goedecker, Xavier Gonze, Oscar Granas, E.ăK.ăU. Gross, Andris Gulans, Francois Gygi, D.ăR. Hamann, PhilăJ. Hasnip, N.ăA.ăW. Holzwarth, Diana Iusan, DominikăB. Jochym, Francois Jollet, Daniel Jones, Georg Kresse, Klaus Koepernik, Emine Kucukbenli, YaroslavăO. Kvashnin, Inka L.ăM. Locht, Sven Lubeck, Martijn Marsman, Nicola Marzari, Ulrike Nitzsche, Lars Nordstrom, Taisuke Ozaki, Lorenzo Paulatto, ChrisăJ. Pickard, Ward Poelmans, Matt I.ăJ. Probert, Keith Refson, Manuel Richter, Gian-Marco Rignanese, Santanu Saha, Matthias Scheffler, Martin Schlipf, Karlheinz Schwarz, Sangeeta Sharma, Francesca Tavazza, Patrik Thunstrom, Alexandre Tkatchenko, Marc Torrent, David Vanderbilt, MichielăJ. van Setten, Veronique VanăSpeybroeck, JohnăM. Wills, JonathanăR. Yates, Guo-Xu Zhang, and Stefaan Cottenier. Reproducibility in density functional theory calculations of solids. *Science*, 2016. doi:10.1126/science.aad3000.

For a detailed physical discussion on the choice of atomic chemical potentials, see Van de Walle and Neugebauer[3].

Results

The study provides a number of methods for processing the available results and returning the physical quantities of interest:

- `formationEnergy`: the formation energy for a given charge state as a function of the electronic chemical potential. The mathematical definition is given in the method description above.
- `stableChargeState`: the stable charge state at a given electronic chemical potential.
- `transitionLevel`: the transition level between two charge states. The mathematical definition is given in the method description above.
- `stableTransitions`: the list of transitions between stable charge states across the whole range of the electronic chemical potential.

Finite-size scaling and correction terms

All the post-processing methods given above take two optional arguments, `supercell_repetitions` and `enable_finite_size_corrections`. By default, the largest supercell is used and finite-size corrections are automatically included.

As previously mentioned, the FNV correction scheme[Page 2389, 1] is used. This scheme introduces two additional terms in the definition of the formation energy: a correction for the electrostatic interaction between periodic replicas of the charged defect center, and a band shift correction to the defect supercell to align the bands to a common reference with the neutral bulk. Moreover, we add an optional strain correction, which accounts for the elastic distortion of the surrounding lattice due to the defect.

The first correction term is added to the total energy of the defect supercell $E^{\mathrm{def},q}$, while the second is substracted from the absolute position of the reference level for the electronic chemical potential μ_e^{ref}. The complete definition of the formation energy therefore becomes

$$E_f^q = E^{\mathrm{def},q} + E_1^{\mathrm{FNV},q} + E^{\mathrm{strain},q} - E^{\mathrm{bulk},q} - \sum_i \Delta n_i \mu_i + q\left(\mu_e^{\mathrm{ref}} - E_2^{\mathrm{FNV},q} + \Delta\mu_e\right),$$

where $E_{1,2}^{\mathrm{FNV},q}$ are the two FNV correction terms, which depend both on the charge state and the supercell. The term $E^{\mathrm{strain},q}$ is the optional strain energy correction, which is given by

$$E^{\mathrm{strain},q} = -\frac{V}{2K}(P^{\mathrm{def}} - P^{\mathrm{bulk}})^2,$$

where V is the volume of the supercell, K is the bulk modulus of the host material, P^{def} is the external hydrostatic pressure in the supercell with the defect, and P^{bulk} is the external hydrostatic pressure for the pristine system. See equation 57 and related discussion in[4] for a more detailed description of the elastic correction and the underlying approximations.

Note: The elastic strain correction is only calculated if the parameter `elastic_correction_method` is explicitly set. If the parameter is not specified, $E^{\mathrm{strain},q}$ is set to zero in the formation energy calculation.

---

[3] ChrisăG. Van de Walle and Jörg Neugebauer. First-principles calculations for defects and impurities: Applications to III-nitrides. *J. Appl. Phys.*, 95:3851–3879, 2004. doi:10.1063/1.1682673.

[4] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and ChrisăG. Van De Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86(1):253–305, 2014. doi:10.1103/RevModPhys.86.253.

Note: The FNV correction terms only apply to non-neutral charge states. The neutral charge state results will be unaffected by the use of the correction.

The FNV correction works by approximating the additional charge on the defect site with a Gaussian distribution in a dielectric background. The width of this distribution can be set manually with the `setModelChargeCorrection` method; by default, it is fitted for each charge state and supercell size so as to best reproduce the long-range electrostatic potential profile of the defect supercell far away from the defect.

Note: In some cases it is not possible to reproduce the calculated potential profile with a Gaussian charge; the fitting procedure will then set an infinite width, effectively switching off the first correction term.

Extrapolating to infinite size

In addition to the finite-size corrections, it is possible to attempt an extrapolation of the formation energy to the limit of infinite system size. This can be specified by providing an instance of *ExtrapolationScheme* as `supercell_repetitions` in the post-processing methods. The extrapolation scheme will use the available results at different supercell sizes to fit a polynomial.

The form of the extrapolating function is somewhat empirical; the rationale behind it and its reliability for some practical test cases are discussed by Lany and Zunger[5]. Care should be taken when making use of the extrapolation results.

Band gap correction

Standard DFT functionals are known to severely underestimate band gaps; this can be a problem for interpreting defect transition levels, which are typically measured relative to the band edges. One solution is to use the HSE functional, which can predict band gaps very accurately; however, it has a much higher computational cost with respect to the standard functionals, and this will be particularly noticeable when performing defect calculations in large supercells.

Alternatively, one can employ a band-gap correction scheme for *ChargedPointDefect* by specifying a separate `band_gap_calculator`; this will perform an additional calculation of the band gap of the host material with this calculator, and position the defect transition levels within it for all post-processing methods. The calculation of the host band gap is computationally inexpensive as it is done only once per study and uses the primitive unit cell, not the supercell needed for the defect. Therefore, we suggest using either HSE or other methods available in QuantumATK for accurately calculating band gaps (*DFT-1/2*, *PPS*, *TB09-MGGA* or *DFT+U*).

When using `band_gap_calculator` there are three possibilities for how the transition levels will be aligned to the corrected band gap; this is specified in all post-processing methods with the `defect_type` argument. For deep defects (the default) the levels will be aligned to the Fermi level of the pristine host material, while for shallow defects they will be aligned either to the valence band edge (for acceptors) or the conduction band edge (for donors). In particular, the default option for deep defects has been shown to work remarkably well over a large range of defects and host materials[6][7][8], reproducing the transition levels calculated with HSE at a small fraction of the computational cost.

[5] Stephan Lany and Alex Zunger. Assessment of correction methods for the band-gap problem and for finite-size effects in supercell defect calculations: Case studies for ZnO and GaAs. *Phys. Rev. B*, 78:235104, 2008. doi:10.1103/PhysRevB.78.235104.

[6] Audrius Alkauskas, Peter Broqvist, and Alfredo Pasquarello. Defect Energy Levels in Density Functional Calculations: Alignment and Band Gap Problem. *Phys. Rev. Lett.*, 101:046405, Jul 2008. URL: https://link.aps.org/doi/10.1103/PhysRevLett.101.046405, doi:10.1103/PhysRevLett.101.046405.

[7] J.ăL. Lyons and C.ăG. Vanăde Walle. Computationally predicted energies and properties of defects in GaN. *npj Comput. Mater.*, 3:12, 2017. URL: https://doi.org/10.1038/s41524-017-0014-2, doi:10.1038/s41524-017-0014-2.

[8] Christopher Linderälv, Anders Lindman, and Paul Erhart. A Unifying Perspective on Oxygen Vacancies in Wide Band Gap Oxides. *J. Phys. Chem. Lett.*, 9(1):222–228, 2018. URL: https://doi.org/10.1021/acs.jpclett.7b03028, doi:10.1021/acs.jpclett.7b03028.

Usage Example

This example script shows how to perform a simple study of the vacancy in bulk silicon and display a summary of results:

```
from SMW import *

# Setup the reference bulk unit cell configuration.
lattice = FaceCenteredCubic(5.4306*Angstrom)

elements = [Silicon, Silicon]

fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates,
)

# Setup the calculator.
basis_set = getattr(LDABasis, 'Silicon_SingleZeta')
calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=LDA.PZ,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
        k_point_sampling=MonkhorstPackGrid(1, 1, 1))
)

# Initialize the ChargedPointDefect object.
charged_point_defect = ChargedPointDefect(
    bulk_configuration=bulk_configuration,
    filename='chargedpointdefect.hdf5',
    formation_energy_calculator=calculator,
    object_id='Si_vacancy',
    point_defect=Vacancy(0),
    charge_states=[2, 1, 0, -1, -2],
    supercell_repetitions_list=[(1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4)],
    relax_atomic_coordinates=False,
    dielectric_constant=11.68,
)

# Run the calculation.
charged_point_defect.update()

# Print a summary of results.
nlprint(charged_point_defect)
```

charged_point_defect.py

No stress correction is used in this example. Here is the resulting output from the `nlprint` command:

```
+-------------------------------------------------------------------------------+
| Charged Point Defect Report                                                   |
+-------------------------------------------------------------------------------+
| Finite size scaling:                                                          |
| (Formation energies are given at zero electronic chemical potential and       |
| including the correction terms. The band shift correction is given by the     |
| band shift multiplied by the charge times minus one.)                         |
+-------------------------------------------------------------------------------+
| Charge: +2                                                                    |
+-------------------------------------------------------------------------------+
|   Number   Inverse length        Formation      Finite size                   |
| of atoms     (1/Angstrom)      energy (eV)  correction (eV)                   |
|        2      2.92307e-01     -3.56904e+00      2.86730e+00                   |
|       16      1.46153e-01     -1.52429e+00      7.57940e-02                   |
|       54      9.74356e-02      1.94982e+00      5.62643e-01                   |
|      128      7.30767e-02      3.16854e+00      4.35393e-01                   |
|      Inf      0.00000e+00      5.70826e+00  (corrected)                       |
|      Inf      0.00000e+00      6.45299e+00  (uncorrected)                     |
|                                                                               |
| Finite size correction contributions:                                         |
|                                                                               |
|   Number             Band  Periodic charge     Image stress                   |
| of atoms       shift (eV)  correction (eV)  correction (eV)                   |
|        2     -4.01568e-01      2.06417e+00                -                 |
|       16     -3.78970e-02      0.00000e+00                -                 |
|       54      5.72778e-02      6.77198e-01                -                 |
|      128      5.13730e-03      4.45668e-01                -                 |
+-------------------------------------------------------------------------------+
| Charge: +1                                                                    |
+-------------------------------------------------------------------------------+
|   Number   Inverse length        Formation      Finite size                   |
| of atoms     (1/Angstrom)      energy (eV)  correction (eV)                   |
|        2      2.92307e-01     -3.55135e+00      1.04229e+00                   |
|       16      1.46153e-01     -6.72596e-01     -5.61343e-02                   |
|       54      9.74356e-02      2.07451e+00      1.19878e-01                   |
|      128      7.30767e-02      3.23208e+00      1.03283e-01                   |
|      Inf      0.00000e+00      6.23957e+00  (corrected)                       |
|      Inf      0.00000e+00      6.41137e+00  (uncorrected)                     |
|                                                                               |
| Finite size correction contributions:                                         |
|                                                                               |
|   Number             Band  Periodic charge     Image stress                   |
| of atoms       shift (eV)  correction (eV)  correction (eV)                   |
|        2     -5.27329e-01      5.14962e-01                -                 |
|       16      5.61343e-02      0.00000e+00                -                 |
|       54      5.00438e-02      1.69922e-01                -                 |
|      128      2.67978e-02      1.30081e-01                -                 |
+-------------------------------------------------------------------------------+
| Charge: 0                                                                     |
+-------------------------------------------------------------------------------+
|   Number   Inverse length        Formation      Finite size                   |
| of atoms     (1/Angstrom)      energy (eV)  correction (eV)                   |
|        2      2.92307e-01     -2.13008e+00      0.00000e+00                   |
```

(continues on next page)

(continued from previous page)

```
|       16      1.46153e-01      7.62129e-01      0.00000e+00                  |
|       54      9.74356e-02      2.87491e+00      0.00000e+00                  |
|      128      7.30767e-02      3.87204e+00      0.00000e+00                  |
|      Inf      0.00000e+00      6.73582e+00  (corrected)                      |
|      Inf      0.00000e+00      6.73582e+00  (uncorrected)                    |
|                                                                              |
| Finite size correction contributions:                                        |
|                                                                              |
|   Number             Band  Periodic charge     Image stress                  |
| of atoms       shift (eV)  correction (eV)  correction (eV)                  |
|        2     -6.41716e-01      0.00000e+00                -                 |
|       16      1.28916e-01      0.00000e+00                -                 |
|       54      4.77929e-02      0.00000e+00                -                 |
|      128      2.66264e-02      0.00000e+00                -                 |
+------------------------------------------------------------------------------+
| Charge: -1                                                                   |
+------------------------------------------------------------------------------+
|   Number   Inverse length        Formation      Finite size                  |
| of atoms     (1/Angstrom)      energy (eV)  correction (eV)                  |
|        2      2.92307e-01      4.17083e-01     -6.04335e-01                 |
|       16      1.46153e-01      2.65180e+00      3.14123e-01                 |
|       54      9.74356e-02      4.14399e+00      7.96451e-02                 |
|      128      7.30767e-02      5.13477e+00      1.57463e-01                 |
|      Inf      0.00000e+00      8.63413e+00  (corrected)                      |
|      Inf      0.00000e+00      7.82309e+00  (uncorrected)                    |
|                                                                              |
| Finite size correction contributions:                                        |
|                                                                              |
|   Number             Band  Periodic charge     Image stress                  |
| of atoms       shift (eV)  correction (eV)  correction (eV)                  |
|        2     -6.04335e-01      0.00000e+00                -                 |
|       16      6.02071e-02      2.53916e-01                -                 |
|       54      7.96451e-02      0.00000e+00                -                 |
|      128      2.98200e-02      1.27643e-01                -                 |
+------------------------------------------------------------------------------+
| Charge: -2                                                                   |
+------------------------------------------------------------------------------+
|   Number   Inverse length        Formation      Finite size                  |
| of atoms     (1/Angstrom)      energy (eV)  correction (eV)                  |
|        2      2.92307e-01      3.81898e+00     -1.11445e+00                 |
|       16      1.46153e-01      4.95059e+00      9.49127e-01                 |
|       54      9.74356e-02      6.01975e+00      6.15873e-01                 |
|      128      7.30767e-02      6.92347e+00      5.41905e-01                 |
|      Inf      0.00000e+00      1.04297e+01  (corrected)                      |
|      Inf      0.00000e+00      9.91014e+00  (uncorrected)                    |
|                                                                              |
| Finite size correction contributions:                                        |
|                                                                              |
|   Number             Band  Periodic charge     Image stress                  |
| of atoms       shift (eV)  correction (eV)  correction (eV)                  |
|        2     -5.57226e-01      0.00000e+00                -                 |
|       16     -3.31693e-02      1.01547e+00                -                 |
```

(continues on next page)

(continued from previous page)

```
|       54      7.41950e-02      4.67483e-01                -                    |
|      128      3.81477e-02      4.65610e-01                -                    |
+-------------------------------------------------------------------------------+
| Transition levels between stable charge states:                               |
| (All values are calculated using the corrected formation energies.)           |
+-------------------------------------------------------------------------------+
| Supercell: 2 atoms (1x1x1 unit cells)                                         |
+-------------------------------------------------------------------------------+
| Transition    Level (eV)                                                      |
|      +2/+1   1.76875e-02                                                      |
|       +1/0   1.42127e+00                                                      |
|       0/-1   2.54716e+00                                                      |
|      -1/-2   3.40190e+00                                                      |
+-------------------------------------------------------------------------------+
| Supercell: 16 atoms (2x2x2 unit cells)                                        |
+-------------------------------------------------------------------------------+
| Transition    Level (eV)                                                      |
|      +2/+1   8.51697e-01                                                      |
|       +1/0   1.43472e+00                                                      |
|       0/-1   1.88967e+00                                                      |
|      -1/-2   2.29879e+00                                                      |
+-------------------------------------------------------------------------------+
| Supercell: 54 atoms (3x3x3 unit cells)                                        |
+-------------------------------------------------------------------------------+
| Transition    Level (eV)                                                      |
|      +2/+1   1.24691e-01                                                      |
|       +1/0   8.00392e-01                                                      |
|       0/-1   1.26908e+00                                                      |
|      -1/-2   1.87577e+00                                                      |
+-------------------------------------------------------------------------------+
| Supercell: 128 atoms (4x4x4 unit cells)                                       |
+-------------------------------------------------------------------------------+
| Transition    Level (eV)                                                      |
|      +2/+1   6.35398e-02                                                      |
|       +1/0   6.39961e-01                                                      |
|       0/-1   1.26272e+00                                                      |
|      -1/-2   1.78871e+00                                                      |
+-------------------------------------------------------------------------------+
| Extrapolation to infinite limit                                               |
+-------------------------------------------------------------------------------+
| Transition    Level (eV)                                                      |
|       +2/0   5.13780e-01                                                      |
|       0/-2   1.84694e+00                                                      |
+-------------------------------------------------------------------------------+
```

It is important to note that this example is only intended to give an overview of the capabilities of *ChargedPointDefect*; it does not relax the atomic coordinates of the defect supercell, and makes use of a fast calculator.

ChargedPointDefectConfiguration

class ChargedPointDefectConfiguration(*pristine_configuration*, *point_defect*, *atomic_chemical_potentials*, *charge_states=None*, *pre_relaxation_calculator=None*, *pre_relaxation_parameters=None*, *relax_atomic_coordinates=None*, *relaxation_parameters=None*, *rattle_atomic_coordinates=None*, *random_seed=None*, *use_ghost_atoms=None*, *fix_atom=None*)

Create a charge point defect based on a specific configuration.

Parameters

- **pristine_configuration** (*PristineConfiguration*) – The original material without defects.
- **point_defect** (*NamedPointDefect*) – The named point defect used to generate the defected structure.
- **atomic_chemical_potentials** (Sequence or table of *CalculatedChemicalPotential* | *ReferenceChemicalPotential*) – List of chemical potentials for the elements in the defect configuration.
- **charge_states** (*Sequence of int*) – The integer charge states to calculate.
- **relax_atomic_coordinates** (*bool*) – Whether to relax the atomic coordinates. **Default:** True.
- **pre_relaxation_calculator** (`All calculators` | None) – The calculator used for relaxing the atomic coordinates. If not specified and relaxation is enabled the reference calculator is used.
- **pre_relaxation_parameters** (*OptimizeGeometryParameters* | None) – The parameters for the atomic pre-relaxation, if enabled. Only constraints of type *FixAtomConstraints* and *RigidBody* will be maintained, others will be discarded.
- **relaxation_parameters** (*OptimizeGeometryParameters* | None) – The parameters for the atomic relaxation, if enabled. The parameters will also be used when calculating the vibrations if a phonon calculator has been supplied. Only constraints of type *FixAtomConstraints* and *RigidBody* will be maintained, others will be discarded.
- **rattle_atomic_coordinates** (*bool*) – Whether to rattle the atomic coordinates before relaxation to destroy symmetry. **Default:** False
- **random_seed** (*int | None*) – The random seed for the atomic rattling.
- **use_ghost_atoms** (*bool*) – Whether to use ghost atoms when generating single vacancy defects. For other defect types ghost atoms are not supported. **Default:** False
- **fix_atom** – **Deprecated:** from v2024.09, has no effect and should no longer be used.

allConverged()

Returns
Whether all calculated SCF loops and geometry optimizations are converged.

Return type
bool

atomicChemicalPotentials(*element=None*)

Return one or all of the atomic chemical potentials used to define the zero of energy in the defect system.

Parameters
element (*PeriodicTableElement* | None) – The element for which the chemical potential is sought. If not given a list of all chemical potentials is returned.

Returns
The atomic chemical potentials used to define the zero energy in the configuration.

Return type
list | ReferenceChemcialPotential | *CalculatedChemicalPotential*

chargeStates()

Returns
The charge states, in integer electron charges, calculated for the defect.

Return type
tuple of int

concentration(*charge_state=0*, *element_in_pristine=None*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *temperature=PhysicalQuantity(300.0, K)*, *equivalent_defects=None*, *internal_degrees_of_freedom=1*)

Calculates the concentration of the defect in the given charge state and at the given temperature.

Parameters

- **charge_state** (*int*) – The charge state of the defect.
- **element_in_pristine** (None | All | *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference), All (thermodynamic reference), or an element in the pristine material (element poor reference).
- **electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to electronic_chemical_potential_reference. **Default:** 0.0 * eV
- **electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the concentration should be calculated. **Default:** 300K.
- **equivalent_defects** (*int*) – The number of equivalent sites for this defect in the primitive host cell. **Default:** Automatically determined using symmetry.
- **internal_degrees_of_freedom** (*int*) – The internal degrees of freedom of the defect. E.g. spin states or equivalent orientations (self-interstitials). **Default:** 1.

Returns
The concentration of the defect.

Return type
PhysicalQuantity of type inverse volume

defectCenterConfiguration(*charge_state=None*, *symmetry_group_index=0*, *symmetry_operation_index=0*, *unit_cell_translation=None*, *symmetry_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.DefectSuperCell'>*)

Retrieve the calculated position of the defect center for a given supercell, in the form of a bulk configuration (the configuration has a single hydrogen atom at the defect center).

This result will only be available after the calculation ran successfully.

Parameters

- **charge_state** (*int*) – The charge state of the defect. **Default:** 0
- **symmetry_group_index** (*int*) – The index of the group of symmetry operations that give the same configuration.
- **symmetry_operation_index** (*int*) – The index of the operation within the group to apply.
- **unit_cell_translation** (*sequence (size 3) of int*) – An additional translation to apply in terms of the unit cell vectors. **Default:** No translation.

- **symmetry_type** (*DefectUnitCell | DefectSuperCell*) – Whether the supercell or unit cell symmetry is used to transform the defect position. **Default:** DefectSuperCell.

Returns
The defect center configuration. If not available, returns None.

Return type
BulkConfiguration | None

defectConfiguration(*charge_state=None*)

The defect configuration associated with a given change state.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
The configuration of the defect.

Return type
BulkConfiguration

defectSymmetryData(*charge_state=None*)

Return information about the symmetry of the optimized defect.

Parameters
charge_state (*int*) – The charge state of the defect. **Default:** 0

Returns
The symmetry information for the defect supercell. If not available, returns None.

Return type
dict | None

finiteSizeCorrection(*charge_state=None*)

The finite size correction for the defect.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
The band shift correction.

Return type
IsotropicFiniteSizeCorrection | InterfaceFiniteSizeCorrection

fitConcentrationArrheniusParameters(*charge_state=None*, *element_in_pristine=None*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *temperatures=None*, *equivalent_defects=None*, *internal_degrees_of_freedom=1*)

Fit Arrhenius parameters describing concentration at a particular charge state or the intrinsic concentration. The Arrhenius equation is given as:

$C = A \exp(B/T)$

Parameters

- **charge_state** (*int | None*) – The charge state at which the parameters are calculated. If None is given parameters are calculated for the intrinsic concentration.
- **element_in_pristine** (None | All | *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference), All (thermodynamic reference), or an element in the pristine material (element poor reference).

•**electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to `electronic_chemical_potential_reference`. **Default:** `0.0 * eV`

•**electronic_chemical_potential_reference** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`

•**temperatures** (*PhysicalQuantity* of type temperature) – A list of temperatures at which the concentration is calculated for fitting. **Default:** Temperature range of 300K to 1500K in 100K increments.

•**equivalent_defects** (*int*) – The number of equivalent sites for this defect in the primitive host cell. **Default:** Automatically determined using symmetry.

•**internal_degrees_of_freedom** (*int*) – The internal degrees of freedom of the defect. E.g. spin states or equivalent orientations (self-interstitials).

Returns
Tuple of the Arrhenius pre-factor and barrier.

Return type
tuple of *PhysicalQuantity* of type inverse volume and *PhysicalQuantity* of type temperature.

fixAtom()

Deprecated: Drift automatically removed in the structure relaxation without fixing an atom.

Deprecated: Drift automatically removed in the structure relaxation without fixing an atom.

formationEnergyCalculator()

Returns
The calculator used to calculate the formation energy.

Return type
Calculator

formationEntropy(*charge_state=None*, *temperature=None*, *element_in_pristine=None*)

Determines the formation entropy for a given defect, defined as

$S_f^{\mathrm{harmonic}} = S_{\mathrm{vib}}^{\mathrm{def}} - S_{\mathrm{vib}}^{\mathrm{pr}} - \sum_i \Delta n_i s_i$

where the first and second terms correspond to the defect and pristine entropies, respectively, and the third term is the sum of the atomic partial entropies.

Parameters

•**charge_state** (*int*) – The charge of the defect.

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature used for calculating the formation entropy. **Default:** `300.0 * Kelvin`

•**element_in_pristine** (`None | All |` *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be `None` (elemental reference), `All` (thermodynamic reference), or an element in the pristine material (element poor reference).

Returns
The calculated formation entropy.

Return type
PhysicalQuantity of type energy per temperature.

formationFreeEnergy(*charge_state=None*, *element_in_pristine=None*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *enable_finite_size_corrections=None*, *temperature=None*, *include_vibrations=None*)

Return the Helmholtz formation free energy. This is calculated as:

$F_f = U_f - TS_f$

If vibrations are not included, the entropy S_f is zero and the formation internal energy and free energy is simply the electronic formation energy of the defect. If vibrations are included, vibrational corrections are included with both U_f and S_f. These are calculated based on the change in vibrational modes between the defect, pristine and reference materials.

Parameters

•**charge_state** (*int*) – The charge of the defect.

•**element_in_pristine** (None | All | *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference), All (thermodynamic reference), or an element in the pristine material (element poor reference).

•**electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to electronic_chemical_potential_reference. **Default:** 0.0 * eV

•**electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator; see Notes below for details. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True

•**temperature** (*bool*) – The temperature used for calculating vibrational energy **Default:** 300K

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. **Default:** True

Returns
The calculated formation energy.

Return type
PhysicalQuantity of type energy

formationInternalEnergy(*charge_state*, *element_in_pristine=None*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *enable_finite_size_corrections=None*, *temperature=None*, *include_vibrations=None*)

The internal formation energy for a given charge state q of the defect, defined as

$U_f^q = U^{\mathrm{def},q} - U^{\mathrm{bulk},q} - \sum_i \Delta n_i \, \mu_i + q \left(\mu_e^{\mathrm{ref}} + \Delta\mu_e\right)$.

The first two terms are the total internal energy of the supercell with and without the defect, respectively. Δn_i is the difference in the number of atoms between the two supercells for species i, and μ_i is the atomic chemical potential for that species. μ_e^{ref} is the absolute position of the reference level for the electronic chemical potential (either the valence band maximum or the conduction band minimum), and $\Delta\mu_e$ is the electronic chemical potential relative to it.

Parameters

•**charge_state** (*int*) – The charge state of the defect.

•**element_in_pristine** (None | All | *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference), All (thermodynamic reference), or an element in the pristine material (element poor reference).

•**electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to electronic_chemical_potential_reference. **Default:** 0.0 * eV

•**electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True

•**temperature** (*bool*) – The temperature used for calculating vibrational energy **Default:** 300K

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. **Default:** True

Returns
The calculated formation energy.

Return type
PhysicalQuantity of type energy

geometryConverged(*charge_state*)

Return whether a specific geometry optimization converged.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
Whether the geometry optimization converged for this defect.

Return type
bool

intrinsicConcentration(*element_in_pristine=None*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *temperature=PhysicalQuantity(300.0, K)*, *equivalent_defects=None*, *internal_degrees_of_freedom=1*)

The sum of concentrations of all available charge states.

Parameters

•**element_in_pristine** (None | All | *PeriodicTableElement*) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference), All (thermodynamic reference), or an element in the pristine material (element poor reference).

•**electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to electronic_chemical_potential_reference. **Default:** 0.0 * eV

•**electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the concentration should be calculated.

•**equivalent_defects** (*int*) – The number of equivalent sites for this defect in the primitive host cell. **Default:** Automatically determined using symmetry.

•**internal_degrees_of_freedom** (*int*) – The internal degrees of freedom of the defect. E.g. spin states or equivalent orientations (self-interstitials).

isConverged(*charge_state*)

Return whether a specific SCF loop and geometry optimization are converged.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
Whether the SCF loop converged for this defect.

Return type
bool

namedPointDefect()

Returns
The named point defect used to generate the defected configuration.

Return type
NamedPointDefect

nlinfo()

Returns
Information summarizing the defect calculation.

Return type
dict

nlprint(*stream*)

Print a string containing an ASCII table summarizing the available results for the defect.

Parameters
stream (*file-like*) – The stream to write to. This should be an object that supports strings being written to it using a `write` method.

numberOfSymmetricDefects(*charge_state=None*, *symmetry_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.DefectSuperCell'>*)

Return the number of distinct but symmetrically equivalent defect configurations.

Parameters

- **charge_state** (*int*) – The charge state of the defect.
- **symmetry_type** (*DefectSuperCell | DefectUnitCell.*) – Whether the nunber of defects is according to the unit cell or supercell symmetry. **Default:** DefectSuperCell

Returns
The total number of symmetrically equivalent defects.

Return type
int

numberOfSymmetryOperations(*charge_state=None*, *symmetry_group_index=0*, *symmetry_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.DefectSuperCell'>*)

Return the number of symmetry operations that produce identical defects.

Parameters

- **charge_state** (*int*) – The charge state of the defect.
- **symmetry_group_index** (*int*) – The index of the unique defect.
- **symmetry_type** (*DefectSuperCell | DefectUnitCell*) – Whether the number of defects is according to the unit cell or supercell symmetry. **Default:** DefectSuperCell.

Returns
The total number of symmetry operations that produce the same defect.

Return type
int

phononDensityOfStates(*charge_state=None*)

Phonon density of states of the defect.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
The phonon DOS if vibrational corrections are enabled. None if not.

Return type
PhononDensityOfStates | None

pointDefect()

Returns
The point defect used to generate the defected configuration.

Return type
BasePointDefect

possibleTransition(*other*)

Check to see if there is a possible transition path between the current and another defect.

Parameters
other (*ChargedPointDefectConfiguration*) – The other defect to test.

Returns
Whether a transition is possible.

Return type
bool

preRelaxationCalculator()

Returns
The calculator used to relax the atomic coordinates in the pre-optimization step.

Return type
Calculator

preRelaxationParameters()

Returns
The parameters of the optimization used to pre-relax the defect configuration.

Return type
OptimizeGeometryParameters

pristineConfiguration()

Returns
The reference pristine configuration from which the defect is formed.

Return type
PristineConfiguration

randomSeed()

Returns
Random seed used for generating random numbers.

Return type
int

rattleAtomicCoordinates()

Returns
Whether the atomic coordinates are rattled before optimization to break symmetry.

Return type
bool

relaxAtomicCoordinates()

Returns
Whether the atomic coordinates are relaxed using the formation energy calculator.

Return type
bool

relaxationParameters()

Returns
The parameters of the optimization used to relax the atomic coordinates.

Return type
OptimizeGeometryParameters

sameFormationEnergyCalculators(*other*)

Check that the same formation energy calculator was used in two different defects.

Parameters
other (*ChargedPointDefectConfiguration*) – The other defect to test for possible transitions to.

Returns
Whether the formation energy calculators are the same.

Return type
bool

scfConverged(*charge_state*)

Return whether a specific SCF loop converged.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
Whether the SCF loop converged for this defect.

Return type
bool

stableChargeState(*electronic_chemical_potential*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *enable_finite_size_corrections=None*, *temperature=None*, *include_vibrations=None*)

The stable charge state at a given electronic chemical potential.

Parameters

- **electronic_chemical_potential** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to `electronic_chemical_potential_reference`.
- **electronic_chemical_potential_reference** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator; see Notes below for details. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True

•**temperature** (*bool*) – The temperature used for calculating vibrational energy **Default:** 300K

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. **Default:** True

Returns
The calculated stable charge state.

Return type
int

stableTransitions(*electronic_chemical_potential_reference=None*, *defect_type=None*, *enable_finite_size_corrections=None*, *temperature=None*, *include_vibrations=None*)

The list of transitions between stable charge states across the whole range of the electronic chemical potential. The transition levels are given relative to electronic_chemical_potential_reference.

Parameters

•**electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator; see Notes below for details. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect

•**supercell_repetitions** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True

•**temperature** (*bool*) – The temperature used for calculating vibrational energy **Default:** 300K

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. **Default:** True

Returns
The list of transitions between stable charge states in order as the electronic chemical potential is increased, and the values of the corresponding transition levels.

Return type
tuple of (list of list (size 2) of int, list of float)

symmetricDefect(*charge_state=None*, *symmetry_group_index=0*, *symmetry_operation_index=0*, *unit_cell_translation=None*, *symmetry_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.DefectSuperCell'>*)

Generate a symmetrically equivalent defect configuration using the symmetry operations of the pristine material.

Parameters

•**charge_state** (*int*) – The charge state of the defect. **Default:** 0

•**symmetry_group_index** (*int*) – The index of the group of symmetry operations that give the same configuration.

•**symmetry_operation_index** (*int*) – The index of the operation within the group to apply.

•**unit_cell_translation** (*Sequence of int of size 3*) – An additional translation to apply in terms of the unit cell vectors. This can be used move the defect within the supercell.

•**symmetry_type** (*DefectUnitCell | DefectSuperCell*) – Whether the supercell or unit cell symmetry is used to transform the defect position. **Default:** DefectSuperCell.

Returns
The symmetrically equivalent defect.

Return type
BulkConfiguration

symmetricDefectPosition(*charge_state=0*, *symmetry_group_index=0*, *symmetry_operation_index=0*, *unit_cell_translation=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*, *symmetry_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.DefectSuperCell'>*)

Get the symmetrically equivalent defect position using the symmetry operations of the pristine cell.

Parameters

•**charge_state** (*int*) – The charge state of the defect. **Default:** 0

•**symmetry_group_index** (*int*) – The index of the group of symmetry operations that give the same configuration.

•**symmetry_operation_index** (*int*) – The index of the operation within the group to apply.

•**unit_cell_translation** (*Sequence of int of size 3*) – An additional translation to apply in terms of the unit cell vectors. This can be used move the defect within the supercell.

•**coordinate_type** (*Fractional | Cartesian*) – Whether the defect coordinates are given in Cartesian or fractional coordinates. Fractional coordinates are given in reference to the supercell translation vectors.

•**symmetry_type** (*DefectUnitCell | DefectSuperCell*) – Whether the supercell or unit cell symmetry is used to transform the defect position. **Default:** DefectSuperCell.

Returns
The symmetrically equivalent defect position.

Return type
PhysicalQuantity of type length | numpy.ndarray

totalEnergy(*charge_state=None*)

The defect total energy associated with a given charge state.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
The total energy analysis of the defect.

Return type
TotalEnergy

transitionLevel(*charge_states*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *enable_finite_size_corrections=None*, *temperature=None*, *include_vibrations=None*)

The transition level between two calculated charge states m and n, defined as

$E\left(m/n\right) = \frac{E_f^n(\Delta\mu_e=0) - E_f^m(\Delta\mu_e=0)}{m-n}$.

The transition level is given relative to `electronic_chemical_potential_reference`.

Parameters

•**charge_states** (*sequence (size 2) of int*) – The two charge states to calculate the transition level for.

- **electronic_chemical_potential_reference** (ValenceBandEdge | ConductionBandEdge) – The reference level for the electronic chemical potential. **Default:** ValenceBandEdge
- **defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator; see Notes below for details. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect
- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** True
- **temperature** (*bool*) – The temperature used for calculating vibrational energy **Default:** 300K
- **include_vibrations** (*bool*) – Whether vibrational terms are included, if available. **Default:** True

Returns
The calculated transition level.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

update()

Perform the calculation.

useGhostAtoms()

Returns
Whether ghost atoms replace lattice atoms in vacancy defects.

Return type
bool

vibrations(*charge_state=None*)

Vibrational frequencies of the defect.

Parameters
charge_state (*int*) – The charge of the defect.

Returns
The vibrations if vibrational corrections are enabled. None if not.

Return type
PhysicalQuantity of type inverse time | None

Usage Examples

In this example the vacancy and anti-site defects in silicon carbide are calculated using the *ChargedPointDefectConfiguration* object. Note that the DFT calculations can take several hours.

A workflow for the calculation can be downloaded here with a corresponding python script SiC_Defects_Example.py.

Notes

The *ChargedPointDefectConfiguration* object calculates the properties of a defect at one or more charge states. The defect configuration can be optimized for each charge state.

The *ChargedPointDefectConfiguration* is created by supplying a *PristineConfiguration*, a *NamedPointDefect* that defines the defect and a *BaseChemicalPotential* for each element in the defect. Additional arguments set the optimization procedure for the defects and also whether or not ghost atoms are used in vacancy defects. The calculation of the defects is started by calling the `update` method.

Once the calculation is completed the *ChargedPointDefectConfiguration* object can be analyzed in the *Charged Point Defect Analyzer*. Derived properties of the defects such as +formation energy, defect concentration and trap levels can be queried individually using the +corresponding methods.

ChebyshevExpansionSolver

class ChebyshevExpansionSolver(*maximum_order*, *tolerance*, *temperature=None*)

Calculate the density matrix using the Chebyshev expansion method.

Parameters

- **maximum_order** (*int > 0*) – The highest order used in the Chebyshev expansion.
- **tolerance** (*float*) – The threshold to treat Chebyshev coefficients as zero.
- **temperature** (*PhysicalQuantity* of type temperature.) – The smearing of the Fermi distribution. **Default:** 300*Kelvin

maximumOrder()

Returns
The highest order used in the Chebyshev expansion.

Return type
int

temperature()

Returns
The smearing temperature of the Fermi distribution.

Return type
PhysicalQuantity of type temperature

tolerance()

Returns
The threshold where the Chebyshev coefficients are treated as zero.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Notes

The implemented Chebyshev expansion method is based on the work of Baer et al., J.Chem.Phys.107(23),1997. The method has a $O(N^2)$ complexity.

The method decomposes the density matrix as a sum of Chebyshev polynomials

$$D = \sum_{n=0}^{P-1} a_n(\beta, \varepsilon_F) T_n(H)$$

with a_n the Chebyshev coefficients as a function of the inverse temperature β and Fermi energy ε_F. The Chebyshev polynomials T_n are evaluated as a function of the systems Hamiltonian H for each k-point.

In contrast to the method of Baer et al., the order of the Chebyshev expansion is dynamically changed to obtain density matrix results within the requested accuracy. On construction the caller needs to be specify the maximum number of terms in the Chebyshev expansion and the tolerance at which trailing Chebyshev coefficients are considered zero.

CheckpointHandler

class CheckpointHandler(*file_name=None*, *time_interval=None*)

Class for representing the parameters used for saving the configuration and calculator at regular intervals during self-consistent iterations.

Parameters

- **file_name** (*str*) – The name of the checkpoint file. The file can be given as an absolute path or a relative path interpreted relative to the current working directory. The path of the checkpoint file is written in the output log of the calculator. The extension can be either nc for a NetCDF file or hdf5 for an HDF5 file. **Default:** File of the form: `'TEMP/checkpointXXXXXXXX.hdf5'`, where `TEMP` is the temporary folder on the computer and `X` is a random digit.
- **time_interval** (*PhysicalQuantity* of type time interval | `Disabled`) – The approximate time between saving. The program will check if the time since last save exceeds the time interval after each self-consistent iteration. **Default:** `0.5 * Hour`.

fileName()

Returns
The name of the checkpoint file.

Return type
str

nlinfo()

Returns
The nlinfo.

Return type
dict

timeInterval()

Returns
The length of the time interval or `Disabled` if theres no checkpoint handling.

Return type
PhysicalQuantity of type time interval | `Disabled`

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set the checkpoint file to */home/checkpoint.nc* and tell the calculator to store itself every 20 minutes.

```
checkpoint_handler = CheckpointHandler('/home/checkpoint.nc', 20*Minute)
calculator = LCAOCalculator(checkpoint_handler=checkpoint_handler)
```

Turn off checkpoint handling for a calculator.

```
calculator = LCAOCalculator(checkpoint_handler=NoCheckpointHandler)
```

ChemicalCompositionProfile

class ChemicalCompositionProfile(*md_trajectory*, *start_time=None*, *end_time=None*, *resolution=None*, *direction_index=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the ChemicalCompositionProfile object.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the density profile for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** the last time frame.
- **resolution** (*PhysicalQuantity* of type length) – The bin size, which determines the resolution of the profile. **Default:** `2.0 * Angstrom`
- **direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Can only be an element of [0, 1, 2]. **Default:** `2`
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Returns
A dictionary containing the composition profile of each element as a percentage of the total composition

Return type
dict

zValues()

Returns
The positions of the bins associated with the density profile.

Return type
PhysicalQuantity of type length

Usage Examples

Load an *BulkConfiguration* and calculate the chemical composition profile along the C-vector of the cell:

```
# Load the configuration for analysis
configuration = nlread('Si-Si3N4_interface.hdf5')[-1]

# Calculate the mass density profile along the C-vector of the cell.
composition_profile = ChemicalCompositionProfile(
    configuration,
    direction_index=2,
    resolution=2.0 * Angstrom,
    start_time=0.0 * fs
)

# Get the chemical composition and the positions of the bin centers.
chemical_composition = composition_profile.data()
bin_centers = composition_profile.zValues()

# Create the plot model
model = Plot.PlotModel(x_unit=Angstrom)
model.title().setText('Chemical composition')
model.xAxis().setLabel('C axis')
model.yAxis().setLabel('Element Percentage')
model.legend().setVisible(True)

# Add line for each element
for key in chemical_composition.keys():
    line = Plot.Line(bin_centers, chemical_composition[key])
    line.setLabel(f'Percentage of {key}')
    model.addItem(line)

# Set different colors for each line
Plot.Colors.applyToItems(model.items(plot_item_type=Plot.Line))

# Set the ranges and show the plot
model.setLimits()
Plot.show(model)
```

`composition_profile.py Si-Si3N4_interface.hdf5`

Notes

The fraction of each element within a bin is given as a percentage value with respect to the total number of atoms.

ChemicalPotential

class ChemicalPotential(*configuration=None*)

A class for representing the ChemicalPotential.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the chemical potential should be calculated.

evaluate(*spin=None*)

Get the chemical potential(s).

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin channel the chemical potential should be returned for. Must be either Spin.Up, Spin.Down or Spin.All. The chemical potential will only depend on spin when the calculation has been done with a fixed spin moment. In that case if Spin.All is chosen a list of both chemical potentials is returned. **Default:** Spin.All

Returns
The chemical potential. For a device the chemical potentials of the electrodes are returned.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The chemical potential information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the chemical potential of a bulk system and print the result:

```
chemical_potential_object = ChemicalPotential(bulk_configuration)
fermi_level = chemical_potential_object.evaluate()[0]
print 'Fermi level = ', fermi_level.inUnitsOf(eV), ' eV'
```

Calculate the chemical potentials of a device system and print the results:

```
chemical_potential_object = ChemicalPotential(device_configuration)
left_chemical_potential = chemical_potential_object.evaluate()[0]
right_chemical_potential = chemical_potential_object.evaluate()[1]
print 'left chemical potential = ', left_chemical_potential.inUnitsOf(eV), ' eV'
print 'right chemical potential = ', right_chemical_potential.inUnitsOf(eV), ' eV'
```

CleanVacuumRegion

class CleanVacuumRegion(*fuzz_factor=1.5*, *substrate_search_radius=PhysicalQuantity(2.0, Ang)*)

Removes any atoms that are not in the same bonding graph as the fixed atoms. Atoms whose distance is less than the fuzz factor times the sum of covalent radii are considered bonded. So a larger fuzz factor is more conservative when determining which atoms are removed.

Parameters

- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond. **Default:** 1.5.
- **substrate_search_radius** (PhysicalQuantity of type distance | None) – Search radius for finding substrate atoms that have become unbonded from the substrate structure. Unbound atoms that have substrate atoms above them within a cylinder of the given radius are considered to be part of the substrate structure and therefore not removed. Setting this argument to None disables this additional check. **Default:** 2 Angstrom.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run an etching simulation of nitrogen on a silica surface using a ReaxFF potential, with an additional CleanVacuumRegion hook function.

```
# Attach ReaxFF calculator.
potentialSet = ReaxFF_CHONSSi_2012(strict_bondpairs = None)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
bulk_configuration.setCalculator(calculator)

surface_process_simulation = SurfaceProcessSimulation(
    substrate=bulk_configuration,
    filename='sio2_n_etching.hdf5',
    object_id='sps_0',
    random_seed=7,
```

(continues on next page)

(continued from previous page)

```
    temperature=300.0*Kelvin,
    fixed_thickness=4.0*Angstrom,
    thermostat_thickness=8*Angstrom,
    log_interval=10,
)

# Run 5 deposition events, each 20 ps long.
surface_process_simulation.addSequence(
    molecule=Nitrogen,
    number_of_events=5,
    md_time=20*ps,
    time_step=1.0*fs,
    mean_kinetic_energy=25*eV,
    std_kinetic_energy=0.3*eV,
    mean_incident_angle=0*Degrees,
    std_incident_angle=2*Degrees,
    post_md_hooks=[CleanVacuumRegion()],
)

surface_process_simulation.update()
```

sio2_n_etching.py

The results from the simulation can be visualized using the Movie Tool on the QATK LabFloor.

See also ref.spshookscommon.notes.

CoarseGrainMapper

class CoarseGrainMapper

Define a mapping object for converting an all-atom configuration to a coarse grain one. Rules need to be added to define the mapping, and then configuration can be converted.

addMappingRule(*template*, *bead_size*, *bead_indices*, *context_indices=None*, *martini_label=None*)

Add a definition of a coarse grain bead.

Parameters

- **template** (*MoleculeConfiguration* | *BulkConfiguration*) – A configuration that contains an example of the bead.
- **bead_size** (str | *PhysicalQuantity* of type length) – The size of the resulting bead. This can be specified as a Martini bead size (R, S or T) or given explicitly as a bead radius.
- **bead_indices** (*list of int*) – Indices of the atoms in the configuration that are united in the bead.
- **context_indices** (*list of int*) – Additional atoms in the configuration that helps define an instance of the bead.
- **martini_label** (*str*) – A label describing polarity of the bead according to the Martini typing scheme. This requires that the bead size is also specified with a Martini label.

beadIndices(*configuration*, *rule*, *use_precedence=True*)

Return the indices of the atoms that are included in each bead.

Parameters

•**configuration** (*MoleculeConfiguration* | *BulkConfiguration*) – Configuration for which the beads are being sought.

•**rule** (*int*) – The index of the rule for which the beads are sought.

•**use_precedence** (*bool*) – Whether or not precedence is used to exclude already mapped atoms.

Returns
List of sets of the atoms that form each bead.

Return type
list of sets

coarseGrainConfiguration(*atomic_configuration*, *bead_center=None*, *maintain_mass=False*)

Use the mapping rules to create a coarse-grain configuration from a all-atom configuration.

Parameters

•**atomic_configuration** (*MoleculeConfiguration* | *BulkConfiguration*) – The configuration to be converted.

•**bead_center** (`CenterOfGeometry` | `CenterOfMass` | `None`) – Whether the center of geometry or the center of mass of the mapped atoms are used to place each bead. The default None corresponds to using the center of geometry. **Default:** `CenterOfGeometry`

•**maintain_mass** (*bool*) – Whether or not exact bead masses are maintained for beads of the same type. The default is to use the most frequent mass for all beads of a particular type.

Returns
The configuration with mapped atoms converted to coarse grain beads.

Return type
MoleculeConfiguration | *BulkConfiguration*

coarseGrainForces(*atomic_configuration*, *forces*)

Use the mapping rules to create a coarse-grain forces from a all-atom forces.

Parameters

•**atomic_configuration** (*MoleculeConfiguration* | *BulkConfiguration*) – The configuration whose forces are being converted.

•**forces** (`PhysicalQuantity` array of type energy per distance) – The all atom forces being converted

Returns
The configuration with mapped atoms converted to coarse grain beads.

Return type
MoleculeConfiguration | *BulkConfiguration*

coarseGrainTrajectory(*trajectory*, *bead_center=None*, *maintain_mass=False*)

Use the mapping rules to create a coarse-grain trajectory from an all-atom trajectory. This method converts both Trajectory and MDTrajectory objects.

Parameters

•**atomic_configuration** (*Trajectory* | *MDTrajectory*) – The trajectory to be converted.

•**bead_center** (*CenterOfGeometry | CenterOfMass | None*) – Whether the center of geometry or the center of mass of the mapped atoms are used to place each bead. The default None corresponds to using the center of geometry **Default:** CenterOfGeometry

•**maintain_mass** (*bool*) – Whether or not exact bead masses are maintained for beads of the same type. The default is to use the most frequent mass for all beads of a particular type.

Returns
The configuration with mapped atoms converted to coarse grain beads.

Return type
MoleculeConfiguration | *BulkConfiguration*

numberOfRules()

Returns
The number of rules added to the mapping object.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Create a coarse-grain model of polystyrene from an existing all-atom configuration.

```
# Load the bulk all-atom configuration
aa_configuration = nlread('bulk_polystyrene.hdf5')[0]

# Load a template that helps to define the mapping
template = nlread('template.hdf5')[0]

# Create a mapping object and define mapping rules for the three different beads
# Each bead is set to the Martini Small (S) size.
mapper = CoarseGrainMapper()
mapper.addMappingRule(template, 'S', [0, 2, 3, 4, 15, 16, 17, 18])
mapper.addMappingRule(template, 'S', [1, 5, 7, 13, 14], context_indices=[0])
mapper.addMappingRule(template, 'S', [6, 8, 9, 10, 11, 12])

# Create the coarse-grain structure using the defined mapping rules
cg_configuration = mapper.coarseGrainConfiguration(aa_configuration)

# Save the configuration
nlsave('cg_polystyrene.hdf5', cg_configuration)
```

polystyrene_coarse_grain.py

The given script converts an all-atom configuration of bulk polystyrene into a coarse-grain one. The mapping follows the scheme in the figure below. Each bead corresponds to one full heavy atom and two halves, which are shared with neighboring beads. The backbone beads are centered on the carbons connected to the benzene ring. A second bead is then centered on the carbon in the benzene ring that connects to the backbone. Finally, two of the same bead types are used to map the remaining four carbon atoms in the benzene ring. All hydrogen atoms are included with their attached carbon.

Once the mapping object is created, the mapping scheme is created by adding rules that define each bead. This is done by giving a configuration, the size of the bead and the indices of the atoms in the configuration that define the bead. When the all-atom configuration is mapped to a coarse-grain one, graph matching is used to convert all groups of atoms whose bond graph matches the atoms given in the rule. Additional atoms can also be given that help to identify the atoms being converted, without being included in the bead itself. This is done in the example above in defining the

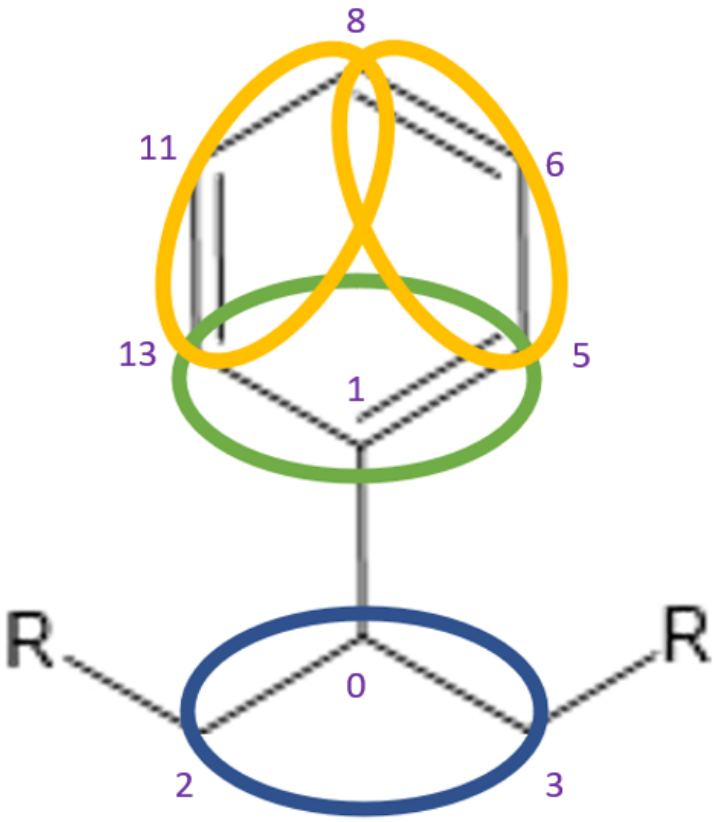


Fig. 4.10: Coarse-grain mapping of polystyrene. The numbers correspond to the indices of the carbon atoms in the polystyrene template.

second bead using the argument `context_indices`. Here the connecting carbon chain atom is also given so that the bead is centered on the connection between the benzene ring and the carbon chain.

Once the mapping is defined the coarse-grain representation is simply created by giving a configuration and calling the `coarseGrainConfiguration` method.

Notes

The *CoarseGrainMapper* class can be used to define a mapping between all-atom and coarse-grain structures that can be then used to convert different configurations or trajectories. The mapping is constructed by adding rules that define each bead. A bead is defined by giving an example of one or more atoms in a configuration that map to the bead. The mapping algorithm can then use graph matching to find all the occurrences of this bond pattern and convert it into a bead. For each rule a template configuration, a bead size and the indices in the configuration that define the bead must be given. The size of the bead can be specified using the Martini formalism of `R` (regular) `S` (small) or `T` (tiny) or by simply giving a particle radius. In cases where the same or similar bonding patterns are used for different beads, additional atoms can be specified using the `context_indices` argument. These atoms are added to the graph pattern but are not included in the bead definition. An optional label for each bead can be also added with the `martini_label` argument.

Once the rules are defined a coarse-grain configuration or trajectory can be created from an all-atom one using the `coarseGrainConfiguration` and `coarseGrainTrajectory` methods. In addition to the configuration these also optionally take a definition of the bead center. New beads can either be placed at the center of mass of the mapped atom, or simply at the average Cartesian coordinates ignoring mass weighting. These options are specified with the flags `CenterOfMass` and `CenterOfGeometry`, respectively. Whether or not the the exact mass of the beads is maintained can also be specified. When converting polymers or other chain structures, beads on the end will often have different numbers of hydrogens or not have the same atom sharing as beads in the center. By default the most frequent mass of each bead type is used. In the case of a simple polyethylene polymer, this results in the whole polymer being represented by C_2H_4 beads. This ignores the additional hydrogens on each end. Setting the argument `maintain_mass` to `True` creates additional beads types for the terminal beads, that map the correct C_2H_5 end group.

When the configuration is converted the rules are applied in order to convert the structure. The graph matching algorithm is able to share atoms between different beads, to enable the creation of symmetric mappings. In the given example, a symmetric mapping of polystyrene is generated by sharing atoms between beads, allowing beads be be centered on the atoms that connect the carbon chain and the benzene ring.

Similar to configurations, all-atom forces can be mapped into coarse-grain forces. This is done assuming that the atoms of the bead form a rigid unit, and so the total force on the bead is the vector sum of the forces acting on the atoms.

CohesiveEnergyDensity

class CohesiveEnergyDensity(*md_trajectory*, *polymer_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *nonbond_cutoff=None*, *electrostatic_cutoff=None*, *electrostatic_accuracy=None*, *use_coulomb_dsf=False*, *memory_cache=True*, *calculate_interaction=False*, *gui_worker=None*)

Create an analyzer to calculate the cohesive energy density of polymer chains.

Parameters

- **md_trajectory** (*MDTrajectory* | *BulkConfiguration*) – The MDTrajectory or AtomicConfiguration that contains the polymers.
- **polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used.
- **start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.

•**end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.

•**time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**nonbond_cutoff** (*PhysicalQuantity* of type length) – The cutoff used to calculate Lennard-Jones interactions. **Default:** The existing Lennard-Jones cutoff

•**electrostatic_cutoff** (*PhysicalQuantity* of type length) – The cutoff used to calculate, either DSF or SPME. **Default:** The existing electrostatic cutoff

•**electrostatic_accuracy** (*float*) – The accuracy of the electrostatic summation when using SPME. **Default:** The existing SPME accuracy

•**use_coulomb_dsf** (*bool*) – Whether to use DSF to evaluate electrostatic interactions. **Default:** False

•**memory_cache** (*bool*) – Whether to cache the entire trajectory in memory. **Default:** True

•**calculate_interaction** (*bool*) – Whether to calculate the individual interaction energy of each molecule. This can take significant additional time to calculate. **Default:** False

•**gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

averageCohesiveEnergy()

Returns
The average configuration cohesive energy in the selected images.

Return type
PhysicalQuantity of type energy

averageCohesiveEnergyDensity()

Returns
The average configuration cohesive energy density in the selected images.

Return type
PhysicalQuantity of type pressure

averageInteractionEnergy(*molecule_tag*)

Parameters
molecule_tag (*str*) – The tag of the molecule in the system used to identify it

Returns
The average interaction energy in the selected images of the selected molecule.

Return type
PhysicalQuantity of type energy

averageMoleculeEnergy(*molecule_tag*)

Parameters
molecule_tag (*str*) – The tag of the molecule in the system used to identify it

Returns
The average molecular energy in the selected images of the selected molecule.

Return type
PhysicalQuantity of type energy

averageNonMolecularEnergy()

Returns
The average energy of the non-molecular components in the selected images.

Return type
PhysicalQuantity of type energy

averageSolubility()

Returns
The average Hildebrand solubility in the selected images.

Return type
PhysicalQuantity of type pressure to the power of 1/2.

averageTotalEnergy()

Returns
The average configuration total energy in the selected images.

Return type
PhysicalQuantity of type energy

averageVolume()

Returns
The average unit cell volume in the selected images.

Return type
PhysicalQuantity of type volume

cohesiveEnergies()

Returns
A 1-D array of the cohesive energies of each image in the trajectory.

Return type
PhysicalQuantity of type energy

cohesiveEnergyDensities()

Returns
A 1-D array of the cohesive energy density of each image in the trajectory.

Return type
PhysicalQuantity of type energy per volume

interactionEnergies(*molecule_tag*)

Return the interaction energy of a specific molecule with other molecules. Requires that the option calculate_interaction is set to True

Parameters
molecule_tag – The tag of the molecule in the system used to identify it

Returns
A 1-D array of the interaction energy of the molecular in each image of the trajectory.

Return type
PhysicalQuantity of type energy

`moleculeEnergies`(*molecule_tag*)

Return the internal energy of a specific molecule.

Parameters
molecule_tag (*str*) – The tag of the molecule in the system used to identify it

Returns
A 1-D array of the energy of the molecule in each image of the trajectory.

Return type
PhysicalQuantity of type energy

`nonMolecularEnergies`()

Return the energy of the non-polymer component of the system.

Returns
A 1-D array of the surface energy in each image of the trajectory.

Return type
PhysicalQuantity of type energy

`polymerTags`()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

`solubilities`()

Returns
A 1-D array of the Hildebrand solubility parameters of each image.

Return type
PhysicalQuantity of type pressure to the power of 1/2.

`times`()

Returns
Times for each image used in the analysis.

Return type
`PhysicalQuantity` of type time

`totalEnergies`()

Returns
A 1-D array of the total energy of each image in the trajectory.

Return type
PhysicalQuantity of type pressure to the power of 1/2.

`volumes`()

Returns
A 1-D array of the unit cell volumes used in the calculation of energy density.

Return type
PhysicalQuantity of type volume

Usage Examples

Calculate the Hildebrand solubility from the cohesive energy density of a PVC melt during a molecular dynamics simulation starting from an equilibrated structure.

```
# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('PolyVinylChloride.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='PVC_Trajectory.hdf5',
    steps=10000,
    log_interval=1000,
    method=method
)

bulk_configuration = md_trajectory.lastImage()

# ------------------------------------------------------------
# Calculate the Hildebrand Solubility
# ------------------------------------------------------------
analyzer = CohesiveEnergyDensity(md_trajectory)
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
model = Plot.PlotModel(ps, Joule**0.5/cm**1.5)
model.axis('x').setLabel('Time')
model.axis('y').setLabel('Hildebrand Solubility')
model.legend().setVisible(False)

line = Plot.Line(analyzer.times(), analyzer.solubilities())
line.setColor('red')
model.addItem(line)

model.setLimits('x', 0*ps, 0.1*ps)
model.setLimits('y', 10*Joule**0.5/cm**1.5, 20*Joule**0.5/cm**1.5)

Plot.save(model, 'CohesiveEnergyDensity_Plot.png')
```

`CohesiveEnergyDensity_Example.py` `PolyVinylChloride.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate the Hildebrand solubility for the polymer system. This is shown in the two plot below.

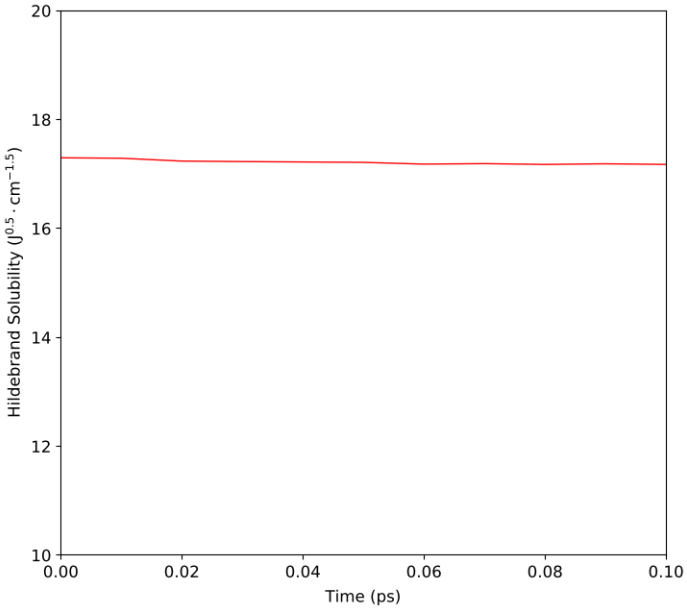


Fig. 4.11: Calculated Hildebrand solubility of PVC.

Notes

This class enables the calculation of the cohesive energy density of a polymer system from either a *MDTrajectory* or *BulkConfiguration*.

The cohesive energy of a polymer system is the energy required to completely separate all of the polymer chains from each other. It therefore consists of the non-bonding interactions between polymer chains, and gives a measure of their attractiveness to each other. In bulk systems this energy is normally expressed per unit of volume of the material, giving the cohesive energy density. The relative differences in cohesive energy densities of different polymers indicates their miscibility or immiscibility with each other. One important measure of the solubility of a polymer is the Hildebrand solubility parameter, δ_h. This is simply given as:

$$\delta_h = \sqrt{\frac{E_{coh}}{V}}$$

Here $E_{coh}$is the cohesive energy density of the unit cell and V is the cell volume. Polymers with a difference in solubility of less than 2MPa$^{0.5}$ are generally expected to be miscible[1].

A more precise measure of miscibility is given through the Flory-Huggins parameter χ[2]. For a binary mixture composed of polymers A and B, the volume fraction of each polymer ϕ can be given as:

$$\phi_A = \frac{n_A r_A}{n_A r_A + n_B r_B}$$

Here n is the number of polymer chains and r is the number of monomers per polymer chain. Given the volume fraction of each polymer, the free energy of mixing ΔG can be approximated as:

$$\Delta G = RT\left(\frac{\phi_A}{r_A}\ln\phi_A + \frac{\phi_B}{r_B}\ln\phi_B + +\phi_A\phi_B\chi_{AB}\right)$$

Here χ_{AB} is the Flory-Huggins parameter for the polymer mixture. In this equation the first two terms specify the entropy of mixing, while the last term approximates the enthalpy of mixing. The Flory-Huggins parameter is therefore a measure of the enthalpy of mixing.

Using a mean-field approximation the Flory-Huggins parameter can be given as:

$$\chi_{AB} = \frac{V_{mono}}{RT}\left[\phi_A\left(\frac{E_{coh}}{V}\right)_A + \phi_B\left(\frac{E_{coh}}{V}\right)_B - \left(\frac{E_{coh}}{V}\right)_{mix}\right]$$

Here V_{mono} is the average volume per monomer. In this approximation it is assumed that both monomers have similar sizes, and so size difference does not need to be taken into account when calculating thermodynamics of mixing. This definition also assumes that the two systems are in contact both with themselves and each other. This is not the case in dilute solutions where one polymer chain may be completely surrounded by others. To calculate the Flory-Huggins parameter the cohesive energy densities of both the pure polymers and the polymer blend is required. In this way, specific interactions between polymers, such as hydrogen bonding, is taken into account in the estimation of miscibility. It should also be noted that, unlike the Hildebrand solubility, the Flory-Huggins parameter is also explicitly a function of temperature. Temperature changes in the system can also indirectly alter the amount of cohesive energy in the system, as thermal motions push the configuration away from equilibrium. Polymer blends with a Flory-Huggins parameter below the critical value are expected to be miscible[Page 2425, 1]. The critical Flory-Huggins value χ_c can be given as:

$$\chi_c = \frac{1}{2}\left(\frac{1}{\sqrt{r_A}} + \frac{1}{\sqrt{r_B}}\right)^2$$

To calculate the cohesive energy density of a polymer system a *CohesiveEnergyDensity* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration* representing the polymer system. The specific images in

[1] Zhonglin Luo and Jianwen Jiang. Molecular dynamics and dissipative particle dynamics simulations for the miscibility of poly(ethylene oxide)/poly(vinyl chloride) blends. *Polymer*, 51:291, 2010. URL: https://www.sciencedirect.com/science/article/abs/pii/S0032386109010040, doi:10.1016/j.polymer.2009.11.024.

[2] PaulăJ. Flory. *Statistical thermodynamics of polymer solutions*, pages 495–540. Cornell University Press, 1953.

the *MDTrajectory* to analyzed can be selected using the start_time, end_time and time_resolution arguments. The polymer chains analyzed in the configuration can be specified with the polymer_tags argument. This takes a list of tags of polymer molecules in the configuration. By default polymers tagged with POLYMER_MOLECULE_# are analyzed, where # is a numerical index. The flag Automatic can also be used to automatically identify polymer chains using the bond graph in the configuration.

To calculate the interaction energy of the polymers, the existing forcefield calculator on the trajectory is used. The accuracy of this calculator can be modified with some additional arguments. The argument nonbond_cutoff controls the cutoff used to evaluate the non-bonding Lennard-Jones potentials. Likewise it is possible to change the electrostatic summation method. By default the summation method defined in the calculator is used. If no electrostatic summation method is set and no electrostatic summation arguments are given, then electrostatic interactions are not included in the calculation of cohesive energy. SPME summation can be used by setting the accuracy of this summation with the argument electrostatic_accuracy. Damped shifted-force electrostatic summation can also be used by setting the argument use_coulomb_dsf to True and giving a cutoff using the electrostatic_cutoff argument. By default the same methods in the attached calculator are used to calculate the cohesive energy. This is also generally the most appropriate use as this is the potential that was used to construct the system.

In addition to the cohesive energy density of the whole system, it is also possible to calculate the interaction energy of each individual polymer chain with the remainder of the system. This is specified by setting the argument calculate_interaction to True. Finally it is possible to set whether the MDTrajectory used to calculate the cohesive energy is cached in memory or read from disk during the calculation. Caching in memory is faster, but for very large trajectories may exhaust available memory. By default trajectories are cached in memory. To force reading from disk, set the argument memory_cache to False.

Once the object is created the properties of the polymer system can be returned using query methods. The unit cell volumes of each selected image can be returned with the volumes methods. The cohesive energy and cohesive energy density can be returned with the methods cohesiveEnergies and cohesiveEnergyDensities respectively. The Hildebrand solubility can be returned with the method solubilities. The method totalEnergies returns the total non-bonding energy of each image. The method nonMolecularEnergies returns the internal non-bonding energy of the atoms in each image that are not associated with a polymer tag. For example, in the case where the system contains a metal surface, nonMolecularEnergies returns the internal non-bonding energy of that surface. Finally the methods moleculeEnergies and interactionEnergies return the internal non-bonding energy and the interaction energy of each polymer molecule respectively. In these methods the polymer molecules are indexed using their tags. The method interactionEnergies is also only available if the calculation of interaction energies was specified when the *CohesiveEnergyDensity* object was created by setting the argument calculate_interaction to True.

ColdSmearing

class ColdSmearing(*broadening*)

Parameters
broadening (*PhysicalQuantity* of type energy or temperature) – The broadening of the distribution.

broadening()

Returns
The broadening of the distribution.

Return type
PhysicalQuantity of type energy

Usage Examples

Use the cold smearing occupation function with a broadening of 0.1 eV on an LCAOCalculator:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    occupation_method=ColdSmearing(0.1*eV))

calculator = LCAOCalculator(numerical_accuracy_parameters=numerical_accuracy_parameters)
```

Notes

Note: For comparison of different occupation methods and suggestions for which one to choose, see *Occupation Methods*.

In the cold smearing scheme[1] one replaces the delta function in the density of states by the function:

$$\tilde{\delta}(x) = \frac{1}{\sqrt{\pi}} e^{-(x-1/\sqrt{2})^2} (2 - \sqrt{2}x),$$

The integer occupation numbers are then replaced by fractional occupations determined by the distribution

$$f(x_i) = \frac{1}{2}\left[\sqrt{\frac{2}{\pi}} e^{-x_i^2 - \sqrt{2}x_i - 1/2} + 1 - \mathrm{erf}\left(x_i + \frac{1}{\sqrt{2}}\right)\right],$$

where $x_i = \frac{\epsilon_i - \mu}{\sigma}$ with σ the broadening. In this context the generalized entropy becomes

$$S = \frac{1}{\sqrt{\pi}} \sum_i e^{-(x_i + 1/\sqrt{2})^2} (1 + \sqrt{2}x_i)$$

with $x_i = \frac{\epsilon_i - \mu}{\sigma}$.

Like in the Methfessel-Paxton scheme (see `MethfesselPaxton`) the benefit of the cold smearing scheme is that contributions to the free energy from the broadening, lower than second order, are eliminated. Therefore, the total energy and forces are quite close to the real (zero temperature) value even for large values of the broadening parameter. Compared to the Methfessel-Paxton scheme, cold smearing has the additional benefit that negative occupations can not occur.

It is possible to extrapolate the total energy to zero broadening by including the term

$$\Delta E_{\sigma \to 0}(\sigma) = \frac{5}{6}\sigma S.$$

CollectiveVariableHyperdynamics

class CollectiveVariableHyperdynamics(*distortions*, *temperature=None*, *global_exponent=None*, *global_cut=None*, *gaussian_width=None*, *gaussian_height=None*, *gaussian_frequency=None*, *bias_damping_temperature=None*, *gaussian_limit=None*, *reaction_steps=None*, *optimize_new_state=None*, *measurement_frequency=None*, *tracked_atoms_tag=None*, *allow_no_variables=None*)

Hook function implementing collective variable hyperdynamics.

[1] Nicola Marzari, David Vanderbilt, Alessandro DeăVita, and M.ăC. Payne. Thermal Contraction and Disordering of the Al(110) Surface. *Phys. Rev. Lett.*, 82(16):3296–3299, April 1999. doi:10.1103/PhysRevLett.82.3296.

Parameters

•**distortions** (Sequence of HyperdynamicsBaseDistortion) – The geometric distortions used to create the collective variable.

•**temperature** (*PhysicalQuantity* of type temperature | None) – The temperature of the simulation, used for calculating the hypertime. **Default:** 300 Kelvin

•**global_exponent** (*float*) – The exponent used to create the global distortion. **Default:** 6

•**global_cut** – The value of the global distortion at the transition state. This can be used to set the position of the transition state where individual distortions do not range from 0 to 1. One example of this is when using a bond distortion without a maximum bond length. **Default:** 1

•**gaussian_width** (*float*) – The width of the added Gaussian bias potential. **Default:** 0.025

•**gaussian_height** (*PhysicalQuantity* of type energy) – The height of the added Gaussian bias potential. **Default:** 0.01 eV

•**gaussian_frequency** (*int*) – The number of steps between adding new Gaussian bias potentials. **Default:** 1000

•**bias_damping_temperature** (*PhysicalQuantity* of type temperature | None) – The temperature used to scale the bias potential using well-temperated metadynamics. None does not damp the added potential, resulting the in same size potential being added at each step. **Default:** None

•**gaussian_limit** (*float*) – The upper limit in the collective variable where Gaussians are added to the hyperdynamic bias potential. Gaussians are not added at values above this limit in order to avoid biasing the transition state. **Default:** 1.0

•**reaction_steps** (*int*) – The number of steps at the maximum collective variable before a reaction takes place. **Default:** 5000

•**optimize_new_state** (*bool*) – Use an optimized configuration to reset the distortions after a reaction is detected. **Default:** True

•**measurement_frequency** (*int*) – The step frequency with which measurements are added to the trajectory. **Default:** 10

•**tracked_atoms_tag** (*str*) – Tag giving the indices of atoms for which the position is measured during the simulation. None results in no atoms being tracked. **Default:** None

•**allow_no_variables** (*bool*) – Continue in cases where no variables are found for acceleration. The simulation continues the number of reaction steps without bias before trying again to find variables to accelerate. **Default:** True

allowNoVariables()

Returns
True if the simulation continues in cases where no variables are found for acceleration.

Return type
bool

biasDampingTemperature()

Returns
The temperature used to scale the bias potential.

Return type
PhysicalQuantity of type temperature

callInterval()

Returns
The call interval of this hook function.

Return type
int

`distortions()`

Returns
The geometric distortions used to create the collective variable.

Return type
Sequence of `HyperdynamicsBaseDistortion`

`gaussianCenters()`

Returns
The center of the Gaussian potentials added to form the bias potential.

Return type
ndarray

`gaussianFrequency()`

Returns
The frequency at which new Gaussian bias potentials are added.

Return type
int

`gaussianHeight()`

Returns
The height of the Gaussian bias potential.

Return type
PhysicalQuantity of type energy

`gaussianHeights()`

Returns
The height of the Gaussian potentials added to form the bias potential.

Return type
PhysicalQuantity of type energy

`gaussianLimit()`

Returns
The limit in the collective variable above which Gaussians are not added.

Return type
float

`gaussianWidth()`

Returns
The width of the Gaussian bias potential.

Return type
float

`gaussianWidths()`

Returns
The widths of the Gaussian potentials added to form the bias potential.

Return type
ndarray

globalCut()

Returns
The value of the global distortion at the transition state.

Return type
float

globalExponent()

Returns
The global exponent used in the collective variable.

Return type
float

measurementFrequency()

Returns
The step frequency with which measurements are saved to the trajectory.

Return type
int

numberOfGaussians()

Returns
The number of Gaussian potentials added to form the hyperdynamic bias potential.

Return type
int | None

numberOfVariables()

Returns
The total number of geometric coordinates being accelerated. If the geometric coordinates are not set with a configuration None is returned.

Return type
int | None

optimizeNewState()

Returns
True if the configuration is optimized before setting new distortions.

Return type
bool

reactionSteps()

Returns
The number of steps at the maximum collective variable before a reaction takes place.

Return type
int

temperature()

Returns
The simulation temperature.

Return type
PhysicalQuantity of type temperature

`trackedAtomsTag()`

Returns
The tag for the atoms that are tracked during the simulation.

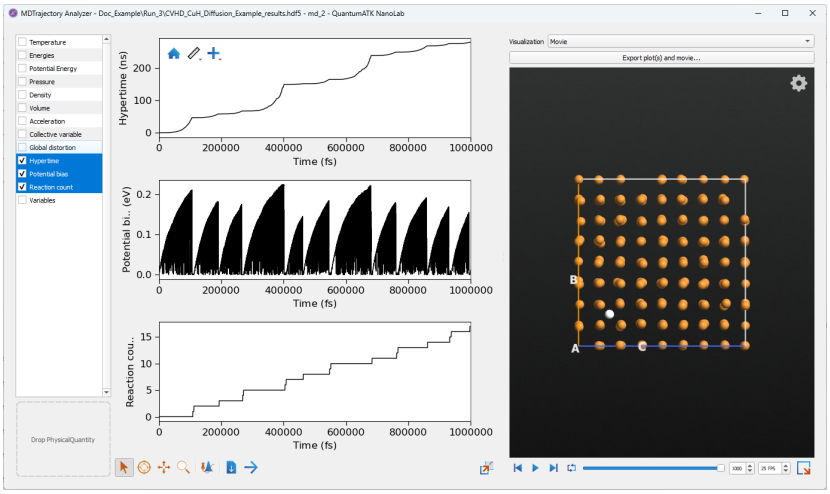
Return type
str

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

In this example collective variable hyperdynamics is used to study the diffusion of a single hydrogen atom in a copper crystal lattice. A position distortion is used to add a bias potential to push the atom through the lattice. The results of the simulation are recorded in the resulting *MDTrajectory* and can be viewed in the **Movie Tool**. The resulting diffusivity can also be calculated using the *IrregularTimeMeanSquareDisplacement* object.



This simulation may take several hours to run. Note that the simulation performs a limited amount of sampling, using only a million steps or 1ns of simulation time. The hypertime of the simulation however is over 250ns, meaning that the simulation was able to effectively sample this time for the cost of only 1ns of simulation. At 300 Kelvin the diffusion of hydrogen in copper is around $8.5 \times 10^{-14} m^2/s$ so very long trajectories are needed to measure the slow diffusion[1]. For accurate results it is necessary to use either more steps or to run multiple simulations. Running multiple simulations can enable a faster turn-around time, as the simulations can be run parametrically in parallel. The workflow for this example can be `downloaded here` with the corresponding Python script `CVHD_CuH_Diffusion_Example.py`.

[1] Hans Magnusson and Karin Frisk. Diffusion, Permeation and Solubility of Hydrogen in Copper. *Journal of Phase Equilibria and Diffusion*, 38:65, Jan 2017. URL: https://link.springer.com/article/10.1007/s11669-017-0518-y, doi:https://doi.org/110.1007/s11669-017-0518-y.

Notes

Hyperdynamics is a method for sampling rare events using molecular dynamics. In hyperdynamics a bias potential is added to degrees of freedom separating different states. The bias potential aids in lowering the energy barrier between these states, allowing the system to sample different states more freely. In this way the biased degrees of freedom move at elevated temperature, while the rest of the system moves at the simulation temperature. The acceleration factor given to the system by the bias potential can be calculated as the ensemble average of the bias potential. When this boost factor is multiplied by the simulation time it gives the effective sampling time, known as the hypertime.

In using the hypertime to estimate reaction rates, it is important that the bias potential is not applied at the transition state. If bias is applied both at the reactant and transition state it obscures the relative free energy barrier between two states. This can lead to errors in estimations of the rate constant. This also applies to kinetic based processes such as diffusion.

In collective variable hyperdynamics the bias potential is built up during the simulation by adding Gaussian functions to the potential at the value of the collective variable during the simulation[2]. By building up the potential it avoids requiring knowledge of the the appropriate shape or height of the bias potential. This is distinct from *StaticHyperdynamics*, which requires some knowledge of the transition state barrier height. In this way collective variable hyperdynamics is similar to metadynamics. The main difference is that in metadynamics a single bias potential is built up during the simulation. This is converged to create an estimate of the total free energy of a process in the system. In hyperdynamics the bias potential is simply used to accelerate a defined reaction. Once the reaction occurs, the bias potential is reset to zero and is built up again at the new reactant. Rather than the free energy profile, the important results from hyperdynamics simulations are the trajectory and the effective hypertime of the simulation.

In order to create the bias potential, one or more important geometric features, known as distortions, are first identified in the configuration. These can be things such as the position of atoms or the bond lengths between specific atoms. These distortions are generally constructed so that they range from 0 to 1 when moving from reactant to transition state. The different distortions are then combined into a global distortion through the equation:

$$\chi_t = \left(\sum_{i=1}^{N} \chi_i^p \right)^{1/p}$$

Here χ_t is the global distortion and χ_i are the individual distortions. The power p increases the contribution of the larger distortions to the global distortion. In the *CollectiveVariableHyperdynamics* class the distortions are given as a list using the `distortions` argument. These can be either *HyperdynamicsPositionDistortion* objects or *HyperdynamicsBondDistortion* objects. The exponent p is set using the `global_exponent` keyword argument.

To ensure that the gradients at the boundaries are zero, the collective variable is then defined using a cosine function such that:

$$\eta = \begin{cases} \frac{1}{2}\left[1 - \cos\left(\pi \left(\frac{\chi_t}{\chi_c}\right)^2\right)\right] & \chi_t \leq 1 \\ 1 & \chi_t > 1 \end{cases}$$

Here η is the collective variable and χ_c is a cutoff value of the global distortion. This is the value of the global distortion at the transition state. In the simplest case in which one distortion that ranges from 0 to 1 is used, this is simply one. If a combination of distortions are used, or if the distortions do not range from 0 to 1 the χ_c is used to make the collective variable range from 0 to 1. This cutoff value is set using the `global_cut` keyword argument. The collective variable is also used to determine when a reaction has occurred in the configuration. When the collective variable is greater than 1 for a set number of steps the system is considered to have reacted, and the references for distortions and the bias potential is reset to zero. The number of steps used to determine if a reaction has occurred is set with the `reaction_steps` keyword argument. By default this is set to 5000 steps. In setting the new distortions the reference configuration can first be optimized using the current calculator. This ensures that the distortions are set from a local energy minimum. In the case of bonds it ensures that the correct bonds are selected for boosting. Likewise in the case

[2] KristofăM. Bal and ErikăC. Neyts. Merging Metadynamics into Hyperdynamics: Accelerated Molecular Simulations Reaching Time Scales from Microseconds to Seconds. *Journal of Chemical Theory and Computation*, 11(10):4545–4554, 2015.

of positions that the reference positions are at a potential minimum. This optimized reference configuration is not used in the dynamics simulation, as to do so would alter the sampling of different states. It is in general recommended to optimize the reference configuration. Enabling optimization after reactions is done with the `optimize_new_state` keyword argument, which is `True` by default.

After each reaction the distortion definitions are reset to the new reference configuration. In some cases, such as bonds being boosted, it means that a different number of distortions may be selected. This can also include cases where no available distortions are found that can be accelerated. In this case the simulation can simply continue the number of steps between reactions and try again to find some applicable distortions. Alternatively the simulation can simply end if no distortions indicates that something has gone wrong with the simulation. Stopping on no distortions being found is controlled with the `allow_no_variables` keyword argument. By default this is set to `True`, meaning that the simulation will continue.

The total bias potential $\Delta V(\eta)$ is calculated by adding Gaussians at time intervals τ_G. This gives the expression for the total bias potential:

$$\Delta V(\eta) = \sum_{k=1}^{n_G} w_k \exp\left[-\frac{(\eta - \eta(k\tau_G))^2}{2\delta^2}\right]$$

Here w_k is the height of the Gaussian and δ is the Gaussian width. These can be set with the keyword arguments `gaussian_height` and `gaussian_width` respectively. The number of steps between adding Gaussians is also set with the `gaussian_frequency` keyword argument. Together with the Gaussian height and width this sets the rate of energy deposition into the bias during the simulation. The optimal energy deposition rate depends on the depth of the local potential well and the temperature of the simulation. The energy deposition rate should be low enough to allow for effective sampling of the reactant potential energy surface so that a reliable estimate of the hypertime is obtained.

The height of each added Gaussian can also be scaled down depending on the amount of bias already added at the collective variable.This is similar to the well-tempered metadynamics method, where the bias potential converges to a value based on a temperature used to set the rate of reducing the size of the added Gaussians. Using this method the temperature is proportional to the maximum effective temperature of the motion of the collective variable. Reducing the temperature reduces the energy deposition rate into the bias potential. The temperature used to set the height of the Gaussians is given with the `bias_damping_temperature` keyword argument. Here `None` means that no damping is done and that all added Gaussians have the same height. In order to avoid adding bias at the transition state, Gaussians can be added up to a limit in the collective variable. This limit depends on the width of the Gaussians and how close to the true transition state the collective variable extends to. This limit is set with the `gaussian_limit` keyword argument.

For each reaction the hypertime is calculated as the simulation time multiplied by the ensemble average of the bias potential. This is given as:

$$t_h = t_s \int_0^{t_s} \exp(\Delta V(\eta, t)/k_B T) dt$$

Here t_h is the hypertime, t_s is the simulation time and T is the simulation temperature. This is set with the `temperature` keyword argument. Note that as the bias potential is built up over the simulation, it is dependent on the time. When a new reaction is found the bias potential starts off as zero. At this point both the simulation time and hypertime move at the same rate. As the bias potential is built up the hypertime moves increasingly faster until a new reaction is found and the bias potential is reset. To get an accurate measure of the hypertime it is important to have sufficient sampling of areas of high bias. This can limit the bias energy deposition rate. If is is too high sufficient sampling of the bias potential may not occur.

When running a collective variable hyperdynamics simulation in *MolecularDynamics*, information about the simulation, such as the hypertime, current bias potential and collective variable value are added to the trajectory as measurements. These are shown in the **Movie Tool** when opening the trajectory. The number of steps between adding measurements is set using the `measurement_frequency` keyword argument. Additionally the positions of select atoms can be added as a measurement. This is useful in cases where the diffusion of single atom impurities is being studied. These are added with the measurement name `tracked_atoms`. As this is an array of atomic coordinates it

cannot be displayed in the **Movie Tool**. The tag of the atoms to track is given with the `tracked_atoms_tag` keyword argument. If no tag is given atom tracking is not added to the trajectory.

To calculate diffusion from a hyperdynamics simulation the *IrregularTimeMeanSquareDisplacement* class can be used. This class can access both the hypertime and tracked atomic positions from the trajectory and calculated the resulting mean square displacement of the tracked atoms. As diffusion is a stochastic process, it is often useful to run several simulations to fully sample the motion of the atoms. This is especially true for amorphous systems or systems lacking long-range order.

CombinedCalculator

class CombinedCalculator(*calculators=None*, *weights=None*)

Class for representing combinations of calculators for configurations of the type MoleculeConfiguration and BulkConfiguration.

Parameters

- **calculators** (sequence of `Calculator`) – The input calculators.
- **weights** (*sequence of float and/or int (same sequence length as the number of input calculators).*) – Weight for the calculators. **Default:** `[1.0, .. ,1.0]`

calculators()

Returns
The input calculators.

Return type
Sequence of `Calculator`

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

setCheckpointHandler(*checkpoint_handler*)

Set the checkpoint handler. Not applicable and will be passed on.

Parameters
checkpoint_handler (*CheckpointHandler*) – The checkpoint handler to use.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setWeights(*weights*)

Parameters
weights – The weights for calculators.

Rtype weights
list of float.

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Method for upgrading the configuration. The upgrade method is used when applying setCalculator for each encapsulated calculator.

Private method for updating the calculator from the configuration, if it is possible. @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

weights()

Returns
The weights for calculators

Return type
list of float

Usage Examples

The *CombinedCalculator* takes in as many calculators needed.

```
# Two arbitrary calculators are defined for the purpose of this example.
Icalculator = LCAOCalculator()

IIpotential_set = Tersoff_Si_1988()
IIcalculator = TremoloXCalculator(potential_set=IIpotential_set)

list_of_calculators = [Icalculator, IIcalculator]

# In case of specific weights, define them as well.
weights = [1, -0.5]

# Define CombinedCalculator with the two calculators and respective weights.
combined_calculator = CombinedCalculator(
```

(continues on next page)

(continued from previous page)

```
    calculators=list_of_calculators,
    weights=weights,
    )
```

Now the CombinedCalculator may be set on a configuration as usual and used in MD analysis, geometry optimization etc.

Notes

The *CombinedCalculator* adds energy, forces, and stress of the given list of calculators with the given weights. It can be used in simulations like MolecularDynamics, OptimizeGeometry, and can be used with analysis and study objects that require only energy, forces, or stress, such as Forces, TotalEnergy, Stress, ElasticConstants, PhononBandStructre, or PhononDensityOfStates. The CombinedCalculator does not support analysis objects that require the electronic structure.

ComplexBandstructure

class ComplexBandstructure(*configuration*, *energies=None*, *k_point=None*, *energy_zero_parameter=None*)

Analysis class for calculating the complex bandstructure for a bulk configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the complex bandstructure.
- **energies** (PhysicalQuantity of type energy) – The energies for which to calculate the complex bandstructure. **Note**: energies are relative to energy_zero_parameter. **Default:** numpy.linspace(-2, 2, 101) * eV
- **k_point** (*tuple of floats*) – The 2-dimensional fractional k-point (in the kA-kB plane) for which to calculate the complex bandstructure. The k-point is given as a tuple of length 2. **Default:** (0, 0)
- **energy_zero_parameter** (FermiLevel | AbsoluteEnergy) – The choice for the energy zero. **Note**: This also affects the energies parameter. **Default:** FermiLevel

energies()

Returns
The energies for which the complex bandstructure is calculated.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy

evaluate(*spin=None*)

Method for obtaining the complex bandstructure for a specific spin. The complex bandstructure has two components, propagating modes and decaying modes. Both types of modes can be characterized by a value k. For propagating modes k is real, giving the Bloch phase factor in the reciprocal C-direction, whereas decaying modes have a complex value of k.

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin to calculate for. **Default:** Spin.All

Returns
A tuple with two lists, the first corresponding to the propagating modes and the second corresponding to the decaying modes. The lists are ordered according to energy and each entry is a PhysicalQuantity with a (1/Angstrom) unit. The lists have different lengths (and can have zero length) corresponding to the number of modes at this energy.

Return type
tuple of lists

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.All

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

fermiTemperature()

Returns
The Fermi temperature.

Return type
PhysicalQuantity of type temperature

kPoint()

Returns
The 2-dimensional fractional k-koint for which the complex bandstructure is calculated.

Return type
tuple of floats

layerSeparation()

Returns
The distance between layers of the crystal in the transport direction. The transport direction is perpendicular to the AB-plane. This value is called L in the GUI.

Return type
PhysicalQuantity of type length

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

repetition()

Deprecated

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spins()

Returns
The available spin components.

Return type
tuple of Spin.Up | Spin.Down | Spin.All

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the complex bandstructure of silicon in the (100) direction, using a fast semi-empirical method, and save it to a file:

```
# Set up a Si (100) minimal configuration
vector_a = [3.84001408591, 0.0, 0.0]*Angstrom
vector_b = [0.0, 3.84001408591, 0.0]*Angstrom
vector_c = [1.92000704296, 1.92000704296, 2.7153]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

elements = [Silicon, Silicon]

fractional_coordinates = [[  2.50000000e-01,   7.50000000e-01,   5.00000000e-01],
                          [  3.33066907e-16,   2.22044605e-16,   1.00000000e+00]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define calculator - fast Slater-Koster method
calculator = SlaterKosterCalculator(
    basis_set=Vogl.Si_Basis,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
                k_point_sampling=(4, 4, 2))
     )
bulk_configuration.setCalculator(calculator)

# Calculate, save, and print the complex band structure
complex_bandstructure = ComplexBandstructure(
    configuration=bulk_configuration,
```

(continues on next page)

(continued from previous page)

```
    energies=numpy.linspace(-2,5,501)*eV,
    k_point=(0,0),
    )
nlsave('si_100_cbs.hdf5', complex_bandstructure)
nlprint(complex_bandstructure)
```

si_complex_band.py

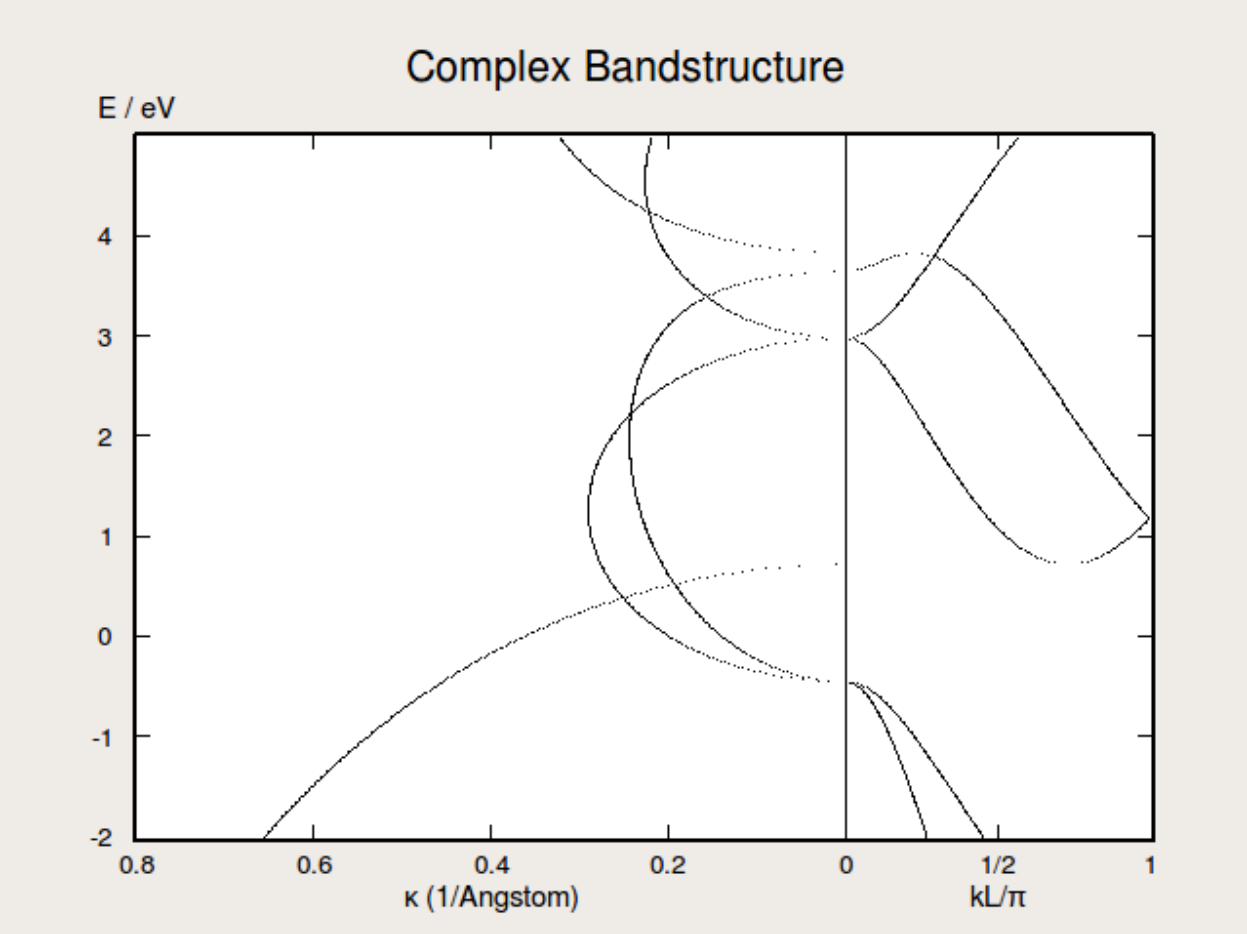


Fig. 4.12: Complex bandstructure for silicon along the (100) direction. The right-hand side of the plots illustrates the real bands while the left-hand side of the plot shows the imaginary part of k_C for the complex bands.

Notes

•The *ComplexBandstructure* is calculated for a specified energy, E, and k-point, k_A, k_B, in the plane spanned by the reciprocal A and B vectors. These are provided as input parameters for the calculation. The solutions are the eigenstates, $\psi_{n\mathbf{k}}$, of the Schrödinger equation

$$H\psi_{n\mathbf{k}} = E\psi_{n\mathbf{k}}$$

with no boundary conditions imposed in the direction perpendicular to the plane spanned by the unit cell vectors $\mathbf{A}$ and $\mathbf{B}$. We may write these eigenstates as

$$\psi_{n\mathbf{k}}(\mathbf{r}) = u_{\mathbf{k}}(\mathbf{r}) e^{-i\mathbf{k}_{AB}\mathbf{r}_{AB}} e^{-ik_C r_C},$$

where $u_{\mathbf{k}}$ is a periodic function and k_C, in general, is a complex number. The solutions with a purely real k_C are the usual Bloch states and we thus obtain the normal band structure. The solutions with a complex k_C form the complex band structure.

- The solutions, k_C, can be obtained after the calculation by using the method `evaluate()`. To print the data, use the method *nlprint*. In the figure above, which is obtained by plotting the complex band structure in QuantumATK, the k-values for the real band structure are normalized by L , the layer separation perpendicular to the AB-plane. The value of L can be extracted by using the method `layerSeparation()`.
- The *ComplexBandstructure* (and the real bands) is computed along the reciprocal C vector, which is perpendicular to the $\mathbf{A}$ and $\mathbf{B}$ unit cell vectors. Thus, if the *ComplexBandstructure* is desired in a particular direction, like (100) as in the example above, the unit cell should be set up such that this direction is perpendicular to the AB-plane. This is most easily achieved by using the **Surface** tool in the **Builder** in QuantumATK and choosing a *Periodic (bulk-like)* type of cell with a minimal number of layers.
- The implementation of the computation of the complex band structure in QuantumATK follows the method described in Ref.[1].

ConfigurationDataContainer

class ConfigurationDataContainer(*path=None*, *group_name=None*)

Constructor for the ConfigurationDataContainer that contains the different configurations.

Parameters

- **path** (*str | None*) – The path to the HDF5 file. If the path is None then all data will be stored in memory and no file will be written. **Default:** None
- **group_name** (*str | None*) – The name of the HDF5 group to store the data in. If the value is None then a new group will be automatically made. **Default:** None

allCoordinates()

Returns
The Cartesian coordinates of all images. A MxNx3 array is returned where M is the number of images and N is the number of atoms per image.

Return type
list of *PhysicalQuantity* of type length

calculator()

Returns
The calculator if one is available, otherwise None.

Return type
Calculator | None

cell(*idx*)

Parameters
idx (*int*) – The index to return.

[1] Y.-C. Chang and J.ăN. Schulman. Complex band structures of crystalline solids: An eigenvalue method. *Phys. Rev. B*, 25:3975–3986, Mar 1982. doi:10.1103/PhysRevB.25.3975.

Returns
The cell vectors of images with index idx or None if the trajectory length is zero or the configuration is a MoleculeConfiguration.

Return type
PhysicalQuantity of type length | None

coordinates(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The Cartesian coordinates of the images or None if the trajectory length is zero.

Return type
PhysicalQuantity of type length | None

elements(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The elements of the images as a list of length n or None if the trajectory length is zero.

Return type
list of type *PeriodicTableElement*

forces(*idx=None*)

Parameters
idx (*int | None*) – The index to return.

Returns
The forces of the image. If the trajectory length is zero or the forces were not written then None is returned.

Return type
PhysicalQuantity of type energy/length | list of *PhysicalQuantity* of type energy/length | None

image(*image_index*, *set_calculator=True*, *set_velocities=True*, *ignore_cache=False*)

Parameters

- **image_index** (*int*) – The index of the configuration that should be extracted.
- **set_calculator** (*bool*) – If True, the calculator is set on the returned image. This can be disabled to increase the performance when iterating over images, because the overhead of copying and setting up a new calculator can be large. **Default:** True
- **set_velocities** (*bool*) – DEPRECATED, only kept for compatibility. The velocities are always set on the configuration.
- **ignore_cache** – If True, the image is not read from the cache.

Returns
A copy of the configuration with the image_index.

Return type
BulkConfiguration | *MoleculeConfiguration*

imageEnergy(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The potential energies of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type energy | None

imageForces(*idx=None*)

Parameters
idx (*int | None*) – The index to return.

Returns
The forces of the image. If the trajectory length is zero or the forces were not written then None is returned.

Return type
PhysicalQuantity of type energy/length | list of *PhysicalQuantity* of type energy/length | None

imageStress(*idx=None*)

Parameters
idx (*int | None*) – The index to return.

Returns
The stresses of the image. If the trajectory length is zero or the stresses were not written then None is returned. None is also returned if the configuration is a MoleculeConfiguration.

Return type
PhysicalQuantity of type energy/length | list of *PhysicalQuantity* of type energy/length | None

images(*set_calculator=True*, *set_velocities=True*, *ignore_cache=False*)

Parameters

- **set_calculator** (*bool*) – If True, the calculator is set on the returned image. This can be disabled to increase the performance when iterating over images, because the overhead of copying and setting up a new calculator can be large. **Default:** True.
- **set_velocities** (*bool*) – If True, the velocities are set on the returned image. This can be disabled to increase performance. **Default:** True.

Returns
All the configurations in the trajectory.

Return type
list of *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

kineticEnergies(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The kinetic energies of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type energy | None

lastImage()

Returns
The last image in the trajectory with the calculator attached.

Return type
BulkConfiguration | *MoleculeConfiguration*

lastStep()

Returns
The step number of the last step.

Return type
int

lastTime()

Returns
The time associated with the last step.

Return type
PhysicalQuantity of type time

length()

Returns
The number of images in the trajectory.

Return type
int

masses(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The masses of the atoms of image with index idx with shape (num_atoms,) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type mass

measurement(*name*, *idx=Ellipsis*)

Parameters

- **name** (*str*) – The name of the measurement.
- **idx** (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
A 2-tuple of the time and the measurement.

Return type
tuple

measurementNames()

Returns
The names of the measurements.

Return type
list of type str

metatext()

Returns
The metatext of the object or None if no metatext is set.

Return type
str | None

nlinfo()

Returns
The trajectory information.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Write a text description of the trajectory to the stream.

Parameters
stream (*file-like object*) – The stream to write to. **Default:** sys.stdout

numberOfAtoms(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The number of atoms in the system.

Return type
int

partialCharges(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The partial charges of the atoms with shape (num_atoms,) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type mass

potentialEnergies(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The potential energies of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type energy | None

pressure(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The pressure value of the image with index idx or None if the trajectory length is zero.

Return type
PhysicalQuantity of type pressure | None

`pressureTensor(idx)`

Parameters
idx (*int*) – The index to return.

Returns
The pressure tensor of the image as a vector with shape (1, 3, 3) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type pressure | None

`pressures()`

Returns
The pressure values of all images as a vector with shape (m) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type pressure | None

`reservoirPressures(idx=Ellipsis)`

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The reservoir pressure of the MD barostat. If the trajectory length is zero or this trajectory was not generated from a NPT simulation then None is returned. NaN is returned for indices for which no temperature data is available.

Return type
PhysicalQuantity of type temperature

`reservoirTemperatures(idx=Ellipsis)`

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The reservoir temperature of the MD thermostat. If the trajectory length is zero or this trajectory was generated from a NVE simulation then None is returned. NaN is returned for indices for which no temperature data is available.

Return type
PhysicalQuantity of type temperature

`setMetatext(text)`
Set a given metatext string on the object.

Parameters
text (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`steps(idx=Ellipsis)`

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The step numbers of the images or None if the trajectory length is zero.

Return type
array of type int | None

`stresses`(*idx=None*)

Parameters
idx (*int | None*) – The index to return.

Returns
The stresses of the image. If the trajectory length is zero or the stresses were not written then None is returned. None is also returned if the configuration is a MoleculeConfiguration.

Return type
PhysicalQuantity of type energy/length | list of *PhysicalQuantity* of type energy/length | None

`tagData`(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The tag data of the image or None if the trajectory length is zero or the tags are not stored in the trajectory.

Return type
ndarray of type int | None

`temperatures`(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index to return. **Default:** All indices

Returns
The instantaneous kinetic temperatures of the images. If the trajectory length is zero or the temperatures were not written during the MolecularDynamics run then None is returned. NaN is returned for indices for which no temperature data is available.

Return type
PhysicalQuantity of type temperature

`timeInterval`()

Returns
The time interval between two images.

Return type
PhysicalQuantity of type time

`timeStep`()

Returns
The time step of the simulation.

Return type
PhysicalQuantity of type time or None if times are not available.

`times`(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The simulation times of the images or None if the trajectory length is zero.

Return type
PhysicalQuantity of type time | None

trajectoryInterval()

Returns
The interval at which snapshots are saved.

Return type
int

uniqueElements(*idx=None*)

Parameters
idx (*int | None*) – The index to return. If None, all images are considered.

Returns
The unique elements of the images, sorted after their atomic numbers or None if the trajectory length is zero.

Return type
list of type *PeriodicTableElement*

velocities(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The velocities of the images. If the trajectory length is zero or the velocities were not written during the MolecularDynamics run then None is returned.

Return type
PhysicalQuantity of type length/time

volume(*idx*)

Parameters
idx (*int*) – The index to return.

Returns
The volume of the unit cell of index idx. If this was a simulation of molecules, then None will be returned.

Return type
PhysicalQuantity of type length**3 | None

volumes()

Returns
The volume of all unit cells. If this was a simulation of molecules, then None will be returned.

Return type
PhysicalQuantity of type length**3 | None

Notes

The *MDTrajectory* format is very efficient but does not support storage of heterogeneous data such as configurations with different numbers and types of atoms. The *ConfigurationDataContainer* format is designed to overcome this limitation. However, it does so in a more efficient way than *Trajectory* which stores every configuration as an individual image.

ConfigurationVelocities

class ConfigurationVelocities(*remove_center_of_mass_momentum=None*)

The ConfigurationVelocities object uses the velocities saved in the configuration as initial velocities in a *MolecularDynamics()* simulation.

Parameters

remove_center_of_mass_momentum (*bool*) – If True, the center-of-mass momentum should be removed, otherwise kept.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Perform a molecular dynamics run of a silicon crystal, using the velocities defined in the bulk configuration as initial velocities in the MD simulation.

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[-0.009580233862,  0.002314591627,  0.010466101485],
                          [ 0.259580233862,  0.247685408373,  0.239533898515]]

# Define velocities
velocities = [[ 0.00221094,  0.00024205, -0.00126377],
              [-0.00221094, -0.00024205,  0.00126377]]*Angstrom/fs

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates,
    velocities=velocities
    )

potentialSet = Tersoff_Si_1988b()
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# Use the exisiting velocities of the configuration as initial velocities.
initial_velocity = ConfigurationVelocities(remove_center_of_mass_momentum=True)

method = NVEVelocityVerlet(initial_velocity=initial_velocity)

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
```

(continues on next page)

(continued from previous page)

```
    steps=500,
    log_interval=100,
    method=method
)
```

configuration_velocities.py

Notes

Pass this object as `initial_velocity` parameter in the MD method (e.g. *NVEVelocityVerlet*), if you want to use to velocities stored in the configuration (e.g from a previous MD run, or explicitly specified in the constructor of the configuration, cf. *BulkConfiguration*) as initial velocities in an MD simulation.

If the configuration does not have any velocities, the initial velocities will be set to zero.

ConfinedOrbital

class ConfinedOrbital(*principal_quantum_number=None*, *angular_momentum=None*, *radial_cutoff_radius=None*, *confinement_start_radius=None*, *additional_charge=None*, *confinement_strength=None*, *confinement_power=None*, *radial_step_size=None*)

Class for representing a confined atomic orbital.

Parameters

- **principal_quantum_number** (*int*) – The all electron principal quantum number (n) of the orbital.
- **angular_momentum** (*int*) – The angular momentum quantum Number (l) of the orbital.
- **radial_cutoff_radius** (*PhysicalQuantity* of type length) – The distance from the core where the basis orbital is zero (compact support radius).
- **confinement_start_radius** (*PhysicalQuantity* of type length) – Radial distance to where the confinement potential starts. The `confinement_start_radius` must be less than or equal to the `radial_cutoff_radius`.
- **additional_charge** (*float*) – Charge of the atom when generating the basis function. **Default:** `0.0`
- **confinement_strength** (*PhysicalQuantity* of type energy) – The confinement strength of the potential confining the atomic orbital. **Default:** `20 * Hartree`
- **confinement_power** (*int*) – The confinement power for the potential. **Default:** `1`
- **radial_step_size** (*PhysicalQuantity* of type length) – The radial step size determining the distance between grid points on the linear radial grid. **Default:** `0.01 * Bohr`

additionalCharge()

Returns
The additional charge used for generating this *ConfinedOrbital* instance.

Return type
float

angularMomentum()

Returns
The angular momentum.

Return type
int

confinementPower()

Returns
The power of the confinement potential.

Return type
int

confinementStartRadius()

Returns
The radius where the confinement potential starts.

Return type
PhysicalQuantity of type length

confinementStrength()

Returns
The strength of the confinement potential.

Return type
PhysicalQuantity of type energy

nlinfo()

Returns
The nlinfo.

Return type
dict

principalQuantumNumber()

Returns
The principal quantum number.

Return type
int

radialCutoffRadius()

Returns
The radial cutoff radius.

Return type
PhysicalQuantity of type length

radialStepSize()

Returns
The radial grid spacing.

Return type
PhysicalQuantity of type length

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Define a *BasisSet* for Hydrogen:

```
hydrogen_1s = ConfinedOrbital(
    principal_quantum_number = 1,
    angular_momentum = 0,
    radial_cutoff_radius = 5.28603678847*Bohr,
    confinement_start_radius = 0.8 * 5.28603678847*Bohr,
    additional_charge = 0.0,
    confinement_strength = 20.000*Hartree,
    confinement_power = 1,
    radial_step_size = 0.001*Bohr,
    )

my_hydrogen_basis = BasisSet(
    element = Hydrogen,
    orbitals = [hydrogen_1s],
    occupations = [1.0],
    pseudopotential = NormConservingPseudoPotential('normconserving/H.LDAPZ.zip'),
    )
```

Notes

The basis functions are found by solving the radial Schrödinger equation for the atom with a confinement potential $V_c(r)$. The confinement potential is defined by the parameters `confinement_strength` (V_0), `confinement_power` (n), `confinement_start_radius` (r_i), and `radial_cutoff_radius` (r_c) through the equation

$$V_c(r) = \begin{cases} 0 & \text{if } r < r_i \\ V_0 \left(\frac{r_c - r_i}{r_c - r}\right)^n \exp[-\frac{r_c - r_i}{r - r_i}] & \text{if } r_i < r < r_c \\ \infty & \text{if } r_c < r \end{cases}$$

Fig. 4.13 shows the confinement potential and the corresponding basis functions used for constructing the LDA standard basis set for hydrogen.

For backwards compatibility, if V_0 is given in units of energy×length (as was the convention in QuantumATK 12.2 and earlier, where a slightly different expression was used for the confinement potential), the rescaled value $V_0/(r_c - r_i)$ is used for the confinement strength, to conform to the new formula given above. Further information about the basis functions can be found in *LCAO basis set*.

ConstantPotential

class ConstantPotential(*particleType*, *c*)

Constructor of the potential.

Parameters

- **particleType** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **c** (*PhysicalQuantity of type energy*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

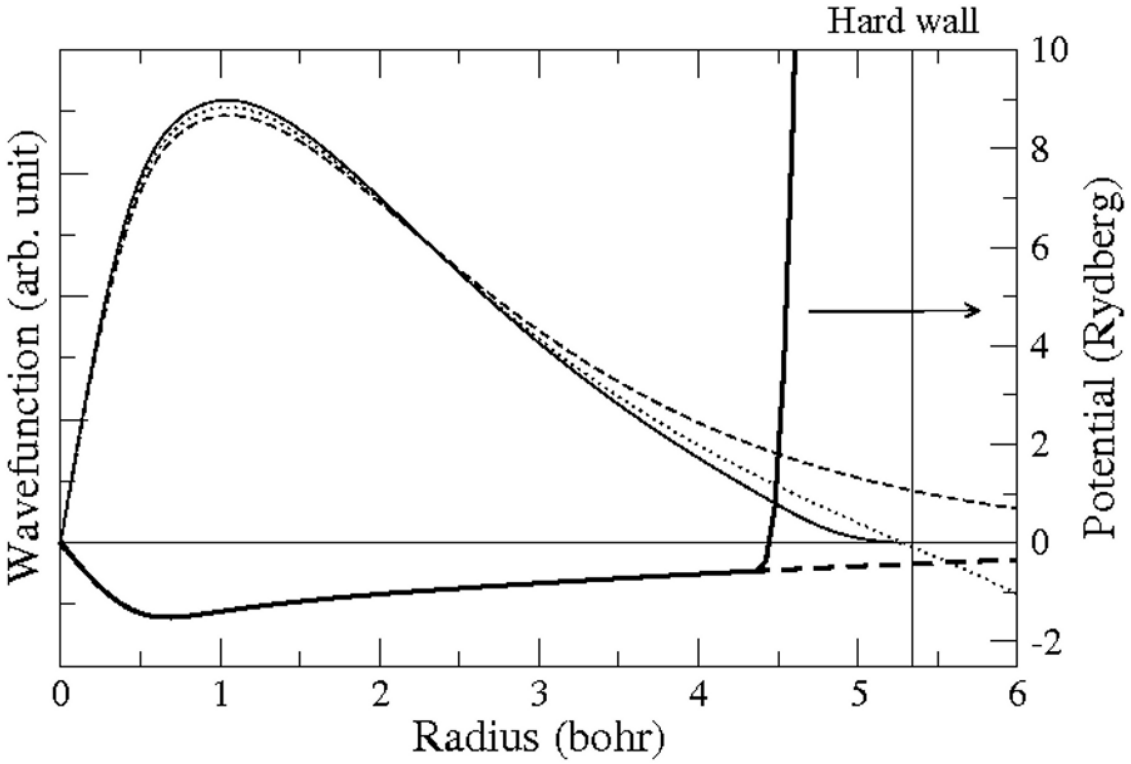


Fig. 4.13: The lower part of the plot shows the $\ell = 0$ effective potential for hydrogen (dashed) with the soft confinement potential (solid). The upper part shows the lowest occupied eigenstate of the confined potential (solid line), and the atomic s-wave function is indicated by the dashed curve (note that $r \cdot \psi(r)$ are plotted).The dotted curve shows the radial wave function with energy 0.01 Ry above the atomic eigenenergy. The position of the first node of this solution defines the position of r_c .

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

Add a constant offset to the potential energy. This potential has no effect on the forces or stress.

ContinuousRandomNetwork

class ContinuousRandomNetwork(*temperature=None*, *heating_rate=None*, *chain_length=None*, *find_bonds=None*, *optimize_geometry_parameters=None*, *random_seed=None*)

Define a continuous random network (CRN) Monte Carlo method.

Parameters

•**temperature** (*PhysicalQuantity* of type temperature) – The initial temperature of the Monte Carlo simulation. **Default:** `300.0 * Kelvin`

•**heating_rate** (*PhysicalQuantity* of type temperature) – The change in temperature per step. A negative temperature results in the simulation cooling at each step. **Default:** `0.0 * Kelvin`

•**chain_length** (*int*) – The length of the chain between the two bonds to switch. **Default:** `1`

•**find_bonds** (*bool*) – Whether the bonding network will be automatically determined using the initial configuration; otherwise, the bonds to use must already be set on the configuration. **Default:** `True`

•**optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters to use for relaxing configurations. Note that *pre_step_hook* and *post_step_hook* must be left unset. Also note that only *max_forces*, *max_steps*, *max_step_length*, *constraints* and *optimizer_method* will be used; all other parameters will be ignored. **Default:** `OptimizeGeometryParameters()`

•**random_seed** (*int*) – The random seed used for generating trial Monte Carlo steps. **Default:** Generated automatically.

chainLength()

Returns
The length of the chain between the two bonds to switch.

Return type
int

findBonds()

Returns
Whether the bonding network will be automatically determined using the initial configuration.

Return type
bool

heatingRate()

Returns
The change in temperature per step, or None for no heating.

Return type
PhysicalQuantity of type temperature | None

monteCarloStep(*configuration*, *energy*)

Perform a bond switch Monte Carlo step and relax the configuration with the new bonding network.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The current configuration.
- **energy** (*PhysicalQuantity* of type energy) – The current energy.

Returns
The new configuration and energy resulting from the trial step.

Return type
tuple (size 2) of (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*, *PhysicalQuantity* of type energy)

optimizeGeometryParameters()

Returns
Parameters required to optimize the geometry.

Return type
OptimizeGeometryParameters

randomSeed()

Returns
The random seed used for performing Monte Carlo steps.

Return type
int

temperature()

Returns
The initial temperature of the Monte Carlo simulation.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Perform a Metropolis Monte Carlo simulation of 100 steps using the ContinuousRandomNetwork model.

```
# Define the Monte Carlo method. Note here the temperature changes from the initial␣
↪50500 to
# 500K
monte_carlo_method = ContinuousRandomNetwork(
    temperature=50500.0*Kelvin,
    heating_rate=-500.0*Kelvin,
    chain_length=1
)

# Perform the Monte Carlo simulation
monte_carlo_trajectory = MetropolisMonteCarlo(
    configuration=alpha_silicon,
    trajectory_filename='Amorphous_Silicon.hdf5',
    steps=100,
    max_trial_moves=5000,
    log_interval=1,
    method=monte_carlo_method,
    trajectory_object_id=None
)
last_image = monte_carlo_trajectory.lastImage()
nlsave('Amorphous_Silicon.hdf5', last_image)
```

continuous_random_network_example.py

In the above example the simulation starts with crystalline alpha-silicon. During the simulation changing the bonding between silicon atoms generates an amorphous structure while still preserving the number of bonds at each silicon. Note that in this simulation a high initial temperature is used so that the configuration is able to get out of the global energy minimum. As the simulation progress the temperature is lowered as less energy is required to perform subsequent bond swap moves. At low temperatures the selected can also lower the overall energy of the amorphous structure.

Notes

The *ContinuousRandomNetwork* class defines a Monte Carlo model that can be used in the *MetropolisMonteCarlo()* method. This Monte Carlo model defines bond swap moves that can be used to generate amorphous configurations. In each trial move, a pair of bonds that are separated by a set number of bonds are exchanged. A relaxation is then performed with an added bond and angle potential on the swapped bonds to return the new bonds back to equilibrium bond lengths and angles. The main advantage of generating trial configurations in this way is that it helps to preserve the bonding geometry around each atom while also moving atoms away from their lattice positions.

The number of bonds between swapped bonds is set by the argument `chain_length`. This can be set so that only like bonds in the material are swapped. For single element materials, such as silicon, the default value of 1 is appropriate. For binary materials, such as silica, a value of 2 should be used to ensure that only silicon-oxygen bonds are exchanged.

ContourParameters

class ContourParameters(*equilibrium_contour=None*, *non_equilibrium_contour=None*, *method=None*)

Generic object representing the contour segments to use in a device calculation.

Parameters

- **equilibrium_contour** (*OzakiContour* | *SemiCircleContour*) – The equilibrium contour to use.
- **non_equilibrium_contour** (*RealAxisContour*) – The non-equilibrium contour to use.
- **method** (*SingleContour* | *DoubleContour*) – The contour method to use, either a single contour or a double contour. **Default:** *DoubleContour* for finite bias, *SingleContour* for zero bias.

equilibriumContour()

Returns
The equilibrium contour.

Return type
OzakiContour | *SemiCircleContour*

method()

Returns
The contour method used.

Return type
SingleContour | *DoubleContour*

nlinfo()

Returns
The nlinfo.

Return type
dict

nonEquilibriumContour()

Returns
The non-equilibrium contour.

Return type
RealAxisContour

uniqueString()

Return a unique string representing the state of the object.

Usage Example

The ContourParameters class sets the contours used to calculate the density matrix. Both the equilibrium and non-equilibrium contour can be specified, e.g. to combine an *OzakiContour* with a *RealAxisContour*:

```
equilibrium_contour = OzakiContour(number_of_poles=128)

non_equilibrium_contour = RealAxisContour(
    real_axis_point_density=0.002*Hartree,
    real_axis_infinitesimal=0.003*Hartree,
```

(continues on next page)

(continued from previous page)

```
    real_axis_kbt_padding_factor=10.0)

contour_parameters = ContourParameters(
    equilibrium_contour=equilibrium_contour,
    non_equilibrium_contour=non_equilibrium_contour)
```

The keyword `method` is used to specify how to deal with the numerical inaccuracy which causes a discrepancy between the left and right approach to obtain the density matrix, e.g.:

```
contour_parameters = ContourParameters(
    method=SingleContour(direction=Left))
```

will only calculate the Left part (see *SingleContour*). If the DoubleContour option is specified:

```
contour_parameters = ContourParameters(method=DoubleContour())
```

the left and right parts are both calculated and added up with a special weighting to obtain the best result (See *DoubleContour*).

Notes

The ContourParameters object specifies the contour points for integrating the density matrix in a DeviceConfiguration with or without bias.

The equilibrium part of the density matrix is obtained by integrating up to the left (right) Fermi level $\mu_{L(R)}$:

$$D_L = \Im\left[-\frac{2}{\pi}\int_{-\infty}^{+\infty} dE\; f(E-\mu_L)\; G(E+i0^+)\right] ,$$
$$D_R = \Im\left[-\frac{2}{\pi}\int_{-\infty}^{+\infty} dE\; f(E-\mu_R)\; G(E+i0^+)\right] ,$$

in which $f(E-\mu)$ is the Fermi-Dirac distribution. The equilibrium density matrix is calculated with an *EquilibriumContour* method. At zero bias D^L and D^R yield identical results.

The non-equilibrium contribution to the density matrix is calculated using a *NonEquilibriumContour* method, which integrates the spectral density matrix from states in the left (right) electrode between the two Fermi levels:

$$\Delta^R_{\mu\nu} = \int_{-\infty}^{+\infty} d\epsilon\; \rho^L_{\mu\nu}(\epsilon)\left[f(\epsilon-\mu_R) - f(\epsilon-\mu_L)\right] ,$$
$$\Delta^L_{\mu\nu} = \int_{-\infty}^{+\infty} d\epsilon\; \rho^R_{\mu\nu}(\epsilon)\left[f(\epsilon-\mu_L) - f(\epsilon-\mu_R)\right] ,$$

in which $\rho^{L(R)}$ is the left (right) spectral density matrix.

It follows there are two equivalent ways to calculate the total density matrix D:

$$D = D^L + \Delta^R ,$$
$$D = D^R + \Delta^L ,$$

When there is a finite bias in the system both expressions yield different estimates for the density matrix due to the finite accuracy of the integration. The `method` parameter determines how the density matrix is obtained from both expressions.

CoordinationNumber

class CoordinationNumber(*md_trajectory*, *cutoff_radius=None*, *start_time=None*, *end_time=None*, *pair_selection=None*, *time_resolution=None*, *info_panel=None*, *analysis_type=None*, *calculate_distribution=None*)

Class for calculating the coordination number for an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the coordination number for.
- **cutoff_radius** (*PhysicalQuantity* of type length) – Cut off distance for nearest neighbors (must be positive). **Default:** 2.5 * Angstrom.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** 0.0 * fs.
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **pair_selection** (*sequence*) – Only include contributions between this selection of atoms. A sequence has to contain two of the following types: Element, tag name, list of indices, or None. If None is given instead of a list then all atom pairs will be considered. **Default:** None.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis. If time_resoultion is None then it will be the same as the MD time step. **Default:** None.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.
- **analysis_type** (*Distribution | Profile | TimeEvolution | None*) – The type of coordination number analysis to perform. **Default:** None.
- **calculate_distribution** (*bool*) – A flag to determine whether the distribution of coordination numbers or the time-dependent evolution of coordination numbers along the trajectory should be calculated. DEPRECATED, use analysis_type instead. **Default:** None.

coordinationNumbers()

Returns
The coordination numbers associated with the histogram.

Return type
numpy.array | *PhysicalQuantity*

data()

Returns
The coordination number distribution if calculate_distribution has been selected, The coordination number profile if calculate_profile has been selected, otherwise an array with the time-dependent evolution of the average coordination number.

Return type
numpy array

distances()

Returns
The distances associated with coordination number profile.

Return type
numpy.array | *PhysicalQuantity*

times()

Returns
The MD-times in case the time-evolution has been calculated, None otherwise.

Return type
PhysicalQuantity of type time | None

Usage Examples

Load an *MDTrajectory* and calculate the coordination number distribution of the first atom with all surrounding hydrogen atoms:

```
md_trajectory = nlread('ethane_md.nc')[-1]

coordination_number = CoordinationNumber(md_trajectory,
                                         cutoff_radius=1.2*Angstrom,
                                         start_time=10000.0*fs,
                                         end_time=50000.0*fs,
                                         pair_selection=[[0], Hydrogen])

# Get the histogram and the associated the coordination numbers.
bin_edges = coordination_number.coordinationNumbers() - 0.4
histogram = coordination_number.data()

# Plot the data using pylab.
import pylab

pylab.bar(bin_edges, histogram, label='Coordination of the first atom', width=0.8)
pylab.xlabel('Coordination number')
pylab.ylabel('Histogram')
pylab.legend()

pylab.show()
```

coordination_number.py

Notes

Set the `cutoff_radius` parameter to the maximum bond length that should be considered. A good choice is typically the end of the first peak in the *RadialDistribution* function of the respective elements.

Use the `pair_selection` parameter to select the atom interactions that are included in the calculation of the coordination number distribution. The first entry specifies the group of central atoms, while the second entry specifies which of the surrounding atoms are considered. You can use elements, lists of indices, tags, or `None` (selecting all atoms) for each entry.

This analysis has two modes, distribution and time evolution, which can be switched via the `calculate distribution` argument. In the distribution analysis the histogram of the coordination numbers is plotted. In the time evolution analysis, the change of the number of atoms associated with each coordination number is analyzed as a function the simulation time. This analysis is performed each coordination number detected in the snapshots of the simulation, as shown in the second example, this yields $CN_{max}+1$ data sets, where CN_{max} is the largest coordination number found in the simulation.

CoreShellHarmonicPotential

class CoreShellHarmonicPotential(*particleType1*, *particleType2*, *K*, *r0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the core particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the shell particle type.
- **K** (*PhysicalQuantity of type energy * length**-2*) – Potential parameter.
- **r0** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a core-shell potential for TiO2-rutile by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
lattice = SimpleTetragonal(4.593*Angstrom, 2.959*Angstrom)

# Define elements
elements = [Titanium, Titanium, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[-0.002453289082, -0.007907909466,  0.008991998629],
                          [ 0.486821568998,  0.505349605489,  0.498303421225],
                          [ 0.313234905089,  0.31385912058 , -0.002386305869],
                          [ 0.708385881142,  0.705329563593,  0.004823488804],
                          [ 0.801244040623,  0.185105024712,  0.48201997633 ],
                          [ 0.183768385074,  0.808268364353,  0.489566273637]]

# Set up configuration
```

(continues on next page)

(continued from previous page)

```
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Adding the core-shell potential.
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name = 'TiO2_CoreShell')

# Add a particle for the oxygen-shell.
# The shell must have zero mass.
potentialSet.addParticleType(ParticleType(symbol='O_shell',
                                          mass=0.0*atomic_mass_unit,
                                          charge=-1.598,
                                          atomicNumber=None))

# Add a particle for the oxygen-core.
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=0.5,
                                          atomicNumber=8))

# Add a particle for titanium.
# For Ti we do not the core-shell approach to model polarization.
potentialSet.addParticleType(ParticleType(symbol='Ti',
                                          mass=47.867*atomic_mass_unit,
                                          charge=2.196,
                                          atomicNumber=22))

# Add the harmonic core-shell coupling for oxygen.
potential = CoreShellHarmonicPotential(particleType1 = 'O',
                                       particleType2 = 'O_shell',
                                       K = 44.3*eV/Angstrom**2,
                                       r0 = 0.0*Angstrom)
potentialSet.addPotential(potential)

# Add the normal short-ranged potentials between titanium atoms.
potential = TosiFumiPotential(particleType1 = 'Ti',
                              particleType2 = 'Ti',
                              A = 31120.528*eV,
                              B = 6.49350649351*1/Angstrom,
                              C = 5.25*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
                              sigma = 0.0*Angstrom,
                              r_i = 5.0*Angstrom,
                              r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Add short-ranged potentials between Ti and O-shells.
```

(continues on next page)

(continued from previous page)

```
potential = TosiFumiPotential(particleType1 = 'Ti',
                              particleType2 = 'O_shell',
                              A = 16957.71*eV,
                              B = 5.15463917526*1/Angstrom,
                              C = 12.59*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
                              sigma = 0.0*Angstrom,
                              r_i = 5.0*Angstrom,
                              r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Add short-ranged potentials between O-shells.
potential = TosiFumiPotential(particleType1 = 'O_shell',
                              particleType2 = 'O_shell',
                              A = 11782.884*eV,
                              B = 4.2735042735*1/Angstrom,
                              C = 30.22*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
                              sigma = 0.0*Angstrom,
                              r_i = 5.0*Angstrom,
                              r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Set a Coulomb-solver for the electrostatic interactions.
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha = 0.2))
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.coreshellcommon.notes.

CoreShellMorsePotential

class CoreShellMorsePotential(*particleType1*, *particleType2*, *E0*, *k*, *r0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the core particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the shell particle type.
- **E0** (*PhysicalQuantity of type energy*) – Potential parameter.
- **k** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **r0** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a core-shell potential for TiO2-rutile by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
lattice = SimpleTetragonal(4.593*Angstrom, 2.959*Angstrom)

# Define elements
elements = [Titanium, Titanium, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[-0.002453289082, -0.007907909466,  0.008991998629],
                          [ 0.486821568998,  0.505349605489,  0.498303421225],
                          [ 0.313234905089,  0.31385912058 , -0.002386305869],
                          [ 0.708385881142,  0.705329563593,  0.004823488804],
                          [ 0.801244040623,  0.185105024712,  0.48201997633 ],
                          [ 0.183768385074,  0.808268364353,  0.489566273637]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Adding the core-shell potential.
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name = 'TiO2_CoreShell')

# Add a particle for the oxygen-shell.
# The shell must have zero mass.
potentialSet.addParticleType(ParticleType(symbol='O_shell',
                                          mass=0.0*atomic_mass_unit,
                                          charge=-1.598,
                                          atomicNumber=None))
```

(continues on next page)

(continued from previous page)

```
# Add a particle for the oxygen-core.
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=0.5,
                                          atomicNumber=8))

# Add a particle for titanium.
# For Ti we do not the core-shell approach to model polarization.
potentialSet.addParticleType(ParticleType(symbol='Ti',
                                          mass=47.867*atomic_mass_unit,
                                          charge=2.196,
                                          atomicNumber=22))

# Add the harmonic core-shell coupling for oxygen.
potential = CoreShellHarmonicPotential(particleType1 = 'O',
                                       particleType2 = 'O_shell',
                                       K = 44.3*eV/Angstrom**2,
                                       r0 = 0.0*Angstrom)
potentialSet.addPotential(potential)

# Add the normal short-ranged potentials between titanium atoms.
potential = TosiFumiPotential(particleType1 = 'Ti',
                              particleType2 = 'Ti',
                              A = 31120.528*eV,
                              B = 6.49350649351*1/Angstrom,
                              C = 5.25*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
                              sigma = 0.0*Angstrom,
                              r_i = 5.0*Angstrom,
                              r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Add short-ranged potentials between Ti and O-shells.
potential = TosiFumiPotential(particleType1 = 'Ti',
                              particleType2 = 'O_shell',
                              A = 16957.71*eV,
                              B = 5.15463917526*1/Angstrom,
                              C = 12.59*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
                              sigma = 0.0*Angstrom,
                              r_i = 5.0*Angstrom,
                              r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Add short-ranged potentials between O-shells.
potential = TosiFumiPotential(particleType1 = 'O_shell',
                              particleType2 = 'O_shell',
                              A = 11782.884*eV,
                              B = 4.2735042735*1/Angstrom,
                              C = 30.22*eV*Angstrom**6,
                              D = 0.0*eV*Angstrom**8,
```

(continues on next page)

(continued from previous page)

```
                                sigma = 0.0*Angstrom,
                                r_i = 5.0*Angstrom,
                                r_cut = 6.0*Angstrom)
potentialSet.addPotential(potential)

# Set a Coulomb-solver for the electrostatic interactions.
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha = 0.2))
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.coreshellcommon.notes.

CoreShellOptimizationOption

class CoreShellOptimizationOption(*maxIterations=None*, *relFTol=None*, *gradTol=None*)

Constructor of the option.

Parameters

- **maxIterations** (*int*) – The maximal number of iterations that are allowed during the optimization of the core-shell potential.
- **relFTol** (*float*) – Stop the optimization of the core-shell potential if the relative change in the objective function value is smaller than this value. If set to < 0, this stopping criterion is disabled.
- **gradTol** (*float*) – Stop the optimization of the core-shell potential if the 2-norm of the gradient of the objective function is smaller than this value. If set to < 0, this stopping criterion is disabled.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a core-shell potential and add an optimization option to stop the internal optimization of the shell positions charges after 20 iterations.

```
# Set up a new core-shell potential.
coreshell_potential = Pedone_AlCaNaOSi_2012()

# Set up an optimization option to stop the optimization of the shell positions
# after 20 iterations.
optimization_option = CoreShellOptimizationOption(
    maxIterations=20,
)

# Add the option to the potential.
coreshell_potential.addOption(optimization_option)
```

Notes

This option can be used to specify the details of the internal optimization of the shell-positions in a Core-Shell potential. It can be added to a *TremoloXPotentialSet* via the `addOption()` method.

CosineAnglePotential

class CosineAnglePotential(*particleType1*, *particleType2*, *particleType3*, *k*, *n*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **k** (*PhysicalQuantity of type energy*) – Potential parameter.
- **n** (*int*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- An angle potential is calculated for a triple *(a,b,c)* of atoms, which are connected by two adjacent bonds *(a,b)* and *(b,c)*. The potential acts on the angle θ formed by these two bonds:

$$U_{ijk} = k_{ijk} \cos(n_{ijk}\theta_{ijk})$$

CosinePowerTorsionPotential

class CosinePowerTorsionPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *c*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.
- **c** (*Sequence of PhysicalQuantity of type energy or PhysicalQuantity array of type energy.*) – Potential parameter. The length of this list must not exceed 10.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

CosineTorsionPotential

class CosineTorsionPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *k*, *n*, *delta*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.

•**k** (*Sequence of PhysicalQuantity of type energy or PhysicalQuantity array of type energy.*) – Potential parameter. The length of this list must not exceed 10.

•**n** (*Sequence of int*) – Potential parameter. Must be of the same length as k.

•**delta** (*Sequence of PhysicalQuantity of type angle or PhysicalQuantity array of type angle.*) – Potential parameter. Must be of the same length as k.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a CosineTorsionPotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethane_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
```

(continues on next page)

(continued from previous page)

```
    ParticleType.fromElement(Carbon, charge=-0.3*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)
# Set up a new torsion potential for H-C-C-H groups and add it to the potential set.
# This potential has a multiplicity of 1.
torsion_potential = CosineTorsionPotential(
    particleType1='H',
    particleType2='C',
    particleType3='C',
    particleType4='H',
    k=[0.0067455*eV],
    n=[3],
    delta=[0.0]
)
potential_set.addPotential(torsion_potential)
# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the CosineTorsionPotential block of the script is shown. The full script can be found found in the file `ethane_bonded_potential.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- Each torsion potential is calculated for a quadruple *(a,b,c,d)* of atoms, which are connected by three bonds *(a,b)* , *(b,c)*, and *(c,d)*. The potential acts on the torsion angle ϕ around the central bond:

$$\phi(\mathbf{r}_{a,b}, \mathbf{r}_{b,c}, \mathbf{r}_{c,d}) = \arccos\left(\frac{(\mathbf{r}_{a,b} \times \mathbf{r}_{b,c}) \cdot (\mathbf{r}_{b,c} \times \mathbf{r}_{c,d})}{|\mathbf{r}_{a,b} \times \mathbf{r}_{b,c}||\mathbf{r}_{b,c} \times \mathbf{r}_{c,d}|}\right)$$

where $\mathbf{r}_{i,j}$, $\{i,j\} \subset \{a,b,c,d\}$ is the distance vector from atom *i* to atom *j*.

The torsion potential is calculated as

$$V(\phi) = \sum_{i=1}^{\text{mult}} k_i \left(1 + \cos(n_i\phi - \delta_i)\right)$$

where $\text{mult} < 6$ is the multiplicity of the torsion potential for the given quadruple of atom types. Each of the parameters *k*, *n*, and δ is given as a list of length mult.

CosmoRS

class CosmoRS(*configuration*, *solvation_energy*)

Class for creating a COSMO-RS species that contains the configuration information needed for the COSMO-RS method. This can either be returned or added directly to the COSMO-RS species database.

Parameters
configuration (*MoleculeConfiguration* | *SurfaceConfiguration* | *BulkConfiguration*) – Configuration with a calculator that supports COSMO-RS calculations.

evaluate()

Returns
The COSMO Real Species that represents the configuration.

Return type
CosmoRealSpecies

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Information about the calculated COSMO real species object.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Create a *CosmoRS* object for carbon dioxide. When opened in the QuantumATK **Data Tool**, the CosmoRS analysis object can be added to the COSMO-RS molecule database. Simply drag and drop the analysis object onto the *Import* tab of the COSMO-RS database.

```
# Calculate the solvation energy
solvation_energy = SolvationEnergy(
    configuration=optimized_configuration
)
nlsave('co2_cosmors_example.hdf5', solvation_energy)

# Create the CosmoRS analysis object
cosmo_rs = CosmoRS(
    configuration=optimized_configuration,
    solvation_energy=solvation_energy
)
nlsave('co2_cosmors_example.hdf5', cosmo_rs)
```

co2_cosmors_example.py

Notes

The *CosmoRS* analysis class is designed to package the calculations necessary to create a *CosmoRealSpecies* object. This is the representation of molecules used in all COSMO-RS calculations. Saving this information as a single analysis object makes it simple to add molecules to the COSMO-RS molecule database. To add molecules to the database in QuantumATK, open the molecule database in the **COSMO-RS Analyzer**. This is one of the *Supplementary Tools* available in the **Data Tool**. In the COSMO-RS molecule database, select the *Import* tab. This has an area where you can add molecules by dragging the *CosmoRS* analysis object from the **Data Tool** onto the *Import* tab.

To create a *CosmoRS* analysis object, both a configuration and a *SolvationEnergy* object are required. The configuration must also contain an updated COSMO calculation. Once the object is created, the `evaluate` method can be used to return the stored *CosmoRealSpecies* object. If working entirely in scripting the *CosmoRealSpecies* object can also be created directly without requiring a *CosmoRS* analysis.

CosmoRSMixture

class CosmoRSMixture(*mixture_components=None*)

Create a mixture to be used as input in COSMO-RS methods. This object takes a list of tuples of input. Each tuple specifies first mixture component as either a *CosmoRealSpecies* or a *MoleculeConfiguration*. Secondly the mole fraction of the component is given, as a float ranging from 0 to 1. The mole fractions must add to 1.

Parameters
mxiture_components (*Sequence of tuples.*) – The list of mixture components.

addMixtures(*ratio*, *other*)

Add two mixtures with a given ratio into a new mixture. Duplicates are added together into the one component.

Parameters

- **ratio** (*float*) – The fraction of the current mixture in the final one. The ratios of the two components should add to 1.
- **other** (*[type]*) – The other component to be added to the new mixture.

Returns
The new mixture created by adding the given two together. If both mixtures share a common component, these will be added together in the new mixture.

Return type
CosmoRSMixture

componentMoleFraction(*index*)

Return a specific component mole fraction.

Parameters
index (*int*) – The index of the mole fraction to return.

Returns
The requested mole fraction.

Return type
float

componentSpecies(*index*)

Return a specific component species.

Parameters
index (*int*) – The index of the component to return.

Returns
The requested species.

Return type
CosmoRealSpecies

components()

Returns
The components in the mixture, given as tuples of species and mole fraction.

Return type
Sequence of tuples

density()

Returns
The molar mass weighted density of the mixture.

Return type
PhysicalQuantity of type mass per volume

molarMass()

The mole fraction weighted molar mass of the mixture.

Returns
The mole fraction weighted molar mass of the mixture.

Return type
PhysicalQuantity of type mass per mole.

molarVolume()

The mole fraction weighted molar volume of the mixture.

Returns
The mole fraction weighted molar mass of the mixture.

Return type
PhysicalQuantity of type mass per mole.

moleFractions()

Returns
The mole fractions of each component.

Return type
Sequence of float

numberOfComponents()

Returns
The number of components in the mixture.

Return type
int

species()

Returns
The species included in the mixture.

Return type
Sequence of *CosmoRealSpecies*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Create a *CosmoRSMixture* object.

```
# Create a mixture.
mixture = CosmoRSMixture([
    (water, 0.6), (methanol, 0.4)
])

# Get the mixture density.
density = mixture.density()
nlprint(f'The density of the mixture is {density}')

# Get the molar volume.
molar_volume = mixture.molarVolume()
nlprint(f'The molar volume of the mixture is {molar_volume}')

# Get the molar mass.
molar_mass = mixture.molarMass()
nlprint(f'The molar mass of the mixture is {molar_mass}')

# Create another mixture.
another_mixture = CosmoRSMixture([
    (methanol, 0.4), (acetonitrile, 0.6)
])
```

(continues on next page)

(continued from previous page)

```
# Combine them to create a new mixture.
new_mixture = mixture.addMixtures(0.5, another_mixture)
for i, x in enumerate(new_mixture.moleFractions()):
    nlprint(f'The mole fraction of component {i} is {x}')
```

mixture_example.py

Notes

The *CosmoRSMixture* object contains information about a mixture composition. The object can be passed on to any other solvent related COSMO-RS object such as *LiquidEquilibrium*. The molar mass of the mixture is defined as the weighted molar mass of the mixture, $\overline{M}$.

$$\overline{M} = \sum_i x_i M_i$$

Here M is the molar mass and x the mole fraction of component species i.

It is also possible to calculate the weighted molar volume of a solvent mixture $\overline{V_m}$. V_m is defined as the volume occupied by 1 mole of species. This property can be determined based on the individual component densities their respective mole fractions.

$$\overline{V_m} = \sum_i x_i V_{m,i}$$

If all of the component species are provided with densities, the molar volume is calculated as

$$V_{m,i} = \frac{M_i}{\rho_i}$$

Here ρ is the density. If the density is not given for a species the molar volume is approximated from the COSMO cavity volume.

$$V_{m,i} = N_A V_i^{COSMO}$$

N_A is Avogadro constant.

The mixture density is then calculated based on the previous properties.

$$\rho = \frac{\overline{M}}{\overline{V_m}}$$

Finally, *CosmoRSMixture* objects may be combined to form a new mixture. This is done using `addMixtures` method where a mixture ratio is also needed. If the mixtures have a component species in common this will be added together to form a single component of the new mixture.

CosmoRSParameterSets

class CosmoRSParameterSets

deleteParameterSet(*name*)

Delete the the given named entry from the available parameter sets. This can only be done for user added entries.

Parameters
name (*str*) – The name of the parameter set to delete.

loadParameterSet(*name*)

Return the parameter set for the given named entry.

Parameters
name (*str*) – The name of the parameter set to return.

Returns
The parameter set.

Return type
CosmoRSParameters

parameterSets()

Returns
A list of the names of the available parameter sets.

Return type
list of str

saveParameterSet(*name*, *parameter_set*, *reference=None*, *description=None*)

Save a parameter set into the set of available parameter sets.

Parameters

- **name** (*str*) – The name to be given to the parameter set.
- **parameter_set** (*CosmoRSParameters*) – The data for the parameter set.
- **reference** (*str*) – An optional reference for where the parameters were obtained.
- **description** (*str*) – An optional description of the parameter set.

Usage Examples

List the available parameter sets and load a parameter set to calculate the chemical potential of water.

```
# Load the parameters database
param_database = CosmoRSParameterSets()

# Print the names of the available parameter sets
parameter_names = param_database.parameterSets()
nlprint('Available parameter set names:')
for name in parameter_names:
    nlprint(name)
nlprint('')

# Load a specific parameter set
parameters = param_database.loadParameterSet('klamt_1998')

# Calculate the chemical potential of water with the new parameter set
molecule_database = CosmoRSSpeciesDatabase()
water = molecule_database.exportSpecies('water')
solvent = CosmoRealSolvent(298*Kelvin, water, parameters)
chemical_potential = solvent.chemicalPotential(water)
nlprint(f'The estimated chemical potential of water is {chemical_potential}')
```

load_parameters_example.py

Save a parameter set into the database and then delete it again.

```
# Create the custom CosmoRSParameters object
parameters = CosmoRSParameters(
    hydrogen_bond_donors=Hydrogen,
    hydrogen_bond_acceptors=(Nitrogen, Oxygen, Fluorine),
    r_average=0.334*Ang,
    alpha_prime=1428.3*kiloCaloriePerMol*Ang**2/elementary_charge**2,
    f_correlation=2.784,
    c_hydrogen_bonding=8800*kiloCaloriePerMol*Ang**2/elementary_charge**2,
    sigma_hydrogen_bonding=0.869*elementary_charge/nm**2,
    hydrogen_bond_temperature=False,
    a_effective=7.79*Ang**2,
    lambda_comb=0.13,
    omega_ring=-0.216*kiloCaloriePerMol,
    eta_gas=-10.05,
    lambda_reg=0.611,
    lambda_ortho=0.820,
    pka_scale=1.0,
    pka_offset=0.0,
    dispersion={
        Hydrogen: -3.59*kiloCaloriePerMol/nm**2,
        Carbon: -3.47*kiloCaloriePerMol/nm**2,
        Nitrogen: -2.26*kiloCaloriePerMol/nm**2,
        Oxygen: -3.17*kiloCaloriePerMol/nm**2,
        Chlorine: -5.00*kiloCaloriePerMol/nm**2,
    },
)

# Save the parameters to the database
database = CosmoRSParameterSets()
database.saveParameterSet(
    'custom_parameter_set',
    parameters,
    reference='DOI: 10.1139/V09-008',
    description=(
        'Taken from "C. C. Pye, T. Zeigler, E. van Lenthe and J. L. Louwen. '
        'Can. J. Chem., 87, 790-797 (2009).'
    )
)

# Print the names of the available parameter sets
parameter_names = database.parameterSets()
nlprint('Available parameter set names:')
for name in parameter_names:
    nlprint(name)
nlprint('')

# Remove the entry from the database again
database.deleteParameterSet('custom_parameter_set')
```

save_delete_parameters_example.py

Notes

The *CosmoRSParameterSets* object functions as an interface to a local database of COSMO-RS parameter sets. This database can also be accessed by the *COSMO-RS Parameters* dialog in the **COSMO-RS Analyzer**. Parameter sets added to the *CosmoRSParameterSets* object also become available in the **COSMO-RS Analyzer**.

Parameter sets can be stored or retrieved with a simple save and load interface. The method `parameterSets` lists the available parameter sets. A parameter set is obtained from the database using the `loadParameterSet` method. This returns a *CosmoRSParameters* object which can be used directly in COSMO-RS calculations. Likewise, a parameter set can be saved using the `saveParameterSet` method, which takes along with it a a *CosmoRSParameters* object that defines the parameter set. Optional text information about the parameter set can also be provided. Once saved, a parameter set can be removed with the `deleteParameterSet` method. Only parameter sets added by the user can be deleted.

CosmoRSParameters

class CosmoRSParameters(*alpha_prime=None*, *a_effective=None*, *f_correlation=None*, *lambda_ortho=None*, *c_hydrogen_bonding=None*, *sigma_hydrogen_bonding=None*, *hydrogen_bond_temperature=None*, *hydrogen_bond_donors=None*, *hydrogen_bond_acceptors=None*, *r_average=None*, *lambda_comb=None*, *lambda_reg=None*, *omega_ring=None*, *eta_gas=None*, *dispersion=None*, *pka_scale=None*, *pka_offset=None*, *sigma_potential_tolerance=None*, *sigma_potential_max_steps=None*, *bin_size=None*)

Constructor for COSMO-RS parameters class. These parameters are used for calculating chemical potentials of different states. The default parameters are the generally recommended ones.

Parameters

- **alpha_prime** (*PhysicalQuantity* of type energy per area) – Weak/misfit interaction coefficient.
- **a_effective** (*PhysicalQuantity* of type area) – The thermodynamically independent contact area.
- **f_correlation** (*float*) – Amount of correlation charge density to include in the misfit energy.
- **lambda_ortho** (*float*) – Amount of local charge density that contributes to the orthogonal charge density.
- **c_hydrogen_bonding** (*PhysicalQuantity* of type energy per area) – Hydrogen bonding coefficient for strong electrostatic interactions.
- **sigma_hydrogen_bonding** (`PhysicalQuantity` of type charge per area) – The charge density threshold above which the interaction is considered as including hydrogen bonding.
- **hydrogen_bond_temperature** (*bool*) – Whether or not the strength of hydrogen bonding is scaled according to the temperature.
- **hydrogen_bond_donors** (Sequence of *PeriodicTableElement*) – The elements that can act as donors for hydrogen bonding.
- **hydrogen_bond_acceptors** (Sequence of *PeriodicTableElement*) – The elements that can act as acceptors for hydrogen bonding.
- **r_average** (*PhysicalQuantity* of type length) – The sphere radius in which the surface charges are averaged.
- **lambda_comb** (*float*) – Factor used in calculating the combinatoral chemical potential.
- **lambda_reg** (*float*) – Factor for the energy difference between the exact and sigma-averaged screening charges.
- **omega_ring** (*PhysicalQuantity* of type energy) – Gas phase energy contribution for atoms in rings.
- **eta_gas** (*float*) – Scaling of the gas phase chemical potential with temperature.

•**dispersion** (*dict*) – The dispersion energy densities for elements.

•**pka_scale** (*float*) – The gradient of the pKa linear correction.

•**pka_offset** (*float*) – The offset of the pKa linear correction.

•**sigma_potential_tolerance** (*PhysicalQuantity* of type charge per area) – The convergence threshold for the sigma potential.

•**sigma_potential_max_steps** (*int*) – The maximum number of steps before the sigma potential is evaluated as non-converged.

•**bin_size** (Sequence of *PhysicalQuantity* of type charge per area) – The bin size for the sigma profile histogram.

aEffective()

Returns
The thermodynamically independent contact area.

Return type
PhysicalQuantity of type area

alphaPrime()

Returns
Weak/misfit interaction coefficient.

Return type
PhysicalQuantity of type energy per area

binSize()

Returns
The histogram bin size of the charge densities.

Return type
PhysicalQuantity of type charge per area

cHydrogenBonding()

Returns
Hydrogen bonding coefficient for strong electrostatic interactions.

Return type
PhysicalQuantity of type energy per area

createDatabaseEntry(*file_path*, *name*, *reference=None*, *description=None*)
Create a parameter database file.

Parameters

•**file_path** (*str*) – The location of where the database entry file is to be saved.

•**name** (*str*) – The name of the parameter database entry.

•**reference** (str | None) – The optional reference for the source of the data.

•**description** (str | None) – Optional description of the database entry.

defaultDispersion()

Returns
The average of the elemental dispersion parameters, used when the dispersion for the given element is undefined.

Return type
PhysicalQuantity of type energy per area

dictionary()

Returns
A dictionary of the parameters that defines the model.

Return type
dict

difference(*other*, *rel_tol=0.0001*)

Return a dictionary of all of the parameters in the object that differ by the given factor from the parameters in another object.

dispersion()

Returns
The dispersion energy densities for elements.

Return type
dict

etaGas()

Returns
Scaling of the gas phase chemical potential with temperature.

Return type
float

fCorrelation()

Returns
Amount of correlation charge density to include in the misfit energy.

Return type
float

hydrogenBondAcceptors()

Returns
The elements that can act as acceptors of hydrogen bonding

Return type
Sequence of *PeriodicTableElement*

hydrogenBondDonors()

Returns
The elements that can act as donors of hydrogen bonding

Return type
Sequence of *PeriodicTableElement*

hydrogenBondTemperature()

Returns
Whether or not hydrogen bonding is scaled with temperature.

Return type
bool

lambdaComb()

Returns
Factor used in calculating the combinatoral chemical potential.

Return type
float

lambdaOrtho()

Returns
Amount of local charge density that contributes to the orthogonal charge density.

Return type
float

lambdaReg()

Returns
Factor for the energy difference between the exact and averaged screening charges.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

omegaRing()

Returns
Gas phase energy contribution for atoms in rings.

Return type
PhysicalQuantity of type energy

pkaOffset()

Returns
The intercept of the linear correction for the pKa.

Return type
float

pkaScale()

Returns
The gradient of the linear correction for the pKa

Return type
float

rAverage()

Returns
The sphere radius for which the surface charges are averaged.

Return type
PhysicalQuantity of type length

sigmaHydrogenBonding()

Returns
The charge density hydrogen bonding threshold.

Return type
PhysicalQuantity of type charge per area

sigmaPotentialMaxSteps()

Returns
The number of steps before the sigma potential is evaluated as non-converged.

Return type
int

sigmaPotentialTolerance()

Returns
The threshold under which the sigma potential is evaluated as converged.

Return type
PhysicalQuantity of type charge per area

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set some custom parameters for the COSMO-RS model and use them to calculate the vapor pressure of water at 300K.

```
# Load the water COSMO-RS species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')

# Create the parameter set with custom parameters for the gas chemical potential
parameters = CosmoRSParameters(
    r_average=0.5*Ang,
    omega_ring=-0.21*kiloCaloriePerMol,
    eta_gas=-9.15,
    lambda_reg=0.8,
    lambda_ortho=0.816,
    dispersion={
        Hydrogen: -4.10*kiloCaloriePerMol/nm**2,
        Oxygen: -4.20*kiloCaloriePerMol/nm**2,
    }
)

# Calculate the vapor pressure using the different parameters
liquid = CosmoRealSolvent(300*Kelvin, water, parameters)
gas = CosmoRealGas(300*Kelvin, parameters)
mu_liquid = liquid.chemicalPotential(water)
mu_gas = gas.chemicalPotential(water)
vapor_pressure = numpy.exp((mu_liquid - mu_gas) / (300 * Kelvin * boltzmann_constant)) *␣
↪bar
```

(continues on next page)

(continued from previous page)

```
nlprint(f'The calculated vapor pressure is {vapor_pressure}')
```

cosmors_parameters_example.py

Notes

The *CosmoRSParameters* class is a container class for the parameters used in the COSMO-RS method. These parameters are used in the calculation of the chemical potential of molecules in different states. They are mostly empirical parameters that are fitted to large sets of experimental data. The equations used in COSMO-RS are published in a number of places[1234]. Care must be taken when reading these sources however, as there are some mistakes and oversights in the original literature.

A number of the parameters are used for calculating either the liquid or solvent chemical potentials. When taking surface charges from a COSMO-DFT calculation the surface charge densities σ^* may fluctuate in an nonphysical manner. This is due to the linear fitting algorithm used to calculate the charge densities. The charge densities used in the COSMO-RS method are smoothed using a sigma averaging method. Here the smoothed charge densities σ are calculated as:

$$\sigma_i = \frac{\sum_j \sigma_j^* I_{ij}}{\sum_j I_{ij}}$$

$$I_{ij} = \frac{r_j^2}{r_j^2 + r_{av}^2} \exp\left(-\frac{d_{ij}^2}{r_{av}^2}\right)$$

Here d_{ij} is the distance between two surface segments, r_j is the average radius of the surface segment and r_{av} is the averaging radius. The averaging radius is one of the parameters in the COSMO-RS model, and can be set using the argument `r_average`.

In addition to the sigma averaged charge densities σ, COSMO-RS also uses orthogonal charges $\sigma^\perp$. These are created by first creating a set of charge densities σ° that averaged from the σ^* charge densities using double the averaging radius. The orthogonal charges are then defined as:

$$\sigma^\perp = \sigma^\circ - \lambda_{ortho}\sigma$$

Here λ_{ortho} is another COSMO-RS parameter. This can be set in the *CosmoRSParameters* object using the argument `lambda_ortho`.

With these two charge densities, the σ-potential $\bar{\mu}(\sigma, \sigma^\perp)$ is then calculated by iteratively summing over all of the surface segments of each component of the solvent. The equation for this can be given as:

$$\bar{\mu}(\sigma, \sigma^\perp) = -\ln\left[\frac{1}{\sum_i x_i A_i}\sum_i x_i \sum_{\nu \in X_i} s_\upsilon \exp\left(\bar{\mu}(\sigma_\nu, \sigma_\nu^\perp) - \frac{a_{eff}}{RT} E(\sigma, \sigma^\perp, \sigma_\nu, \sigma_\nu^\perp)\right)\right]$$

Here x_i is the mole fraction of each solvent component, A_i is the total molecule surface area, $E(\sigma, \sigma^\perp, \sigma_\nu, \sigma_\nu^\perp)$ is the surface segment interaction energy. The effective contact area a_{eff} is another free parameter in the COSMO-RS method. This can be set using the `a_effective` argument. Note also that since the σ-potential $\bar{\mu}$ appears on both the right and left side of the equation, the sigma potential for each surface segment is calculated iteratively.

[1] Andreas Klamt, Volker Jonas, Thorsten Bürger, and John C.ăW. Lohrenz. Refinement and Parametrization of COSMO-RS. *J. Phys. Chem. A.*, 102:5074–5085, 1998. doi:10.1021/jp980017s.

[2] Andreas Klamt. *The basic COSMO-RS*, pages 83–107. Elsevier, 2005.

[3] Andreas Klamt. *Refinements, parameterization, and the complete COSMO-RS*, pages 109–125. Elsevier, 2005.

[4] CoryăC. Pye, Tom Ziegler, Erik van Lenthe, and JaapăN. Louwen. An implementation of the conductor-like screening model of solvation withing the Amsterdam density functional package - Part II. COSMO for real solvents. *Can. J. Chem.*, 87:790–797, 2009. doi:10.1139/V09-008.

The surface segment interaction energy is the sum of two additional terms, the misfit energy and the hydrogen bonding energy. The misfit energy, E_{misfit} which measures the energy of interaction between two surface segments can be given as:

$$E_{misfit}(\sigma_i, \sigma_j, \sigma_i^{\perp}, \sigma_j^{\perp}) = \frac{\alpha'}{2}(\sigma_i + \sigma_j)[(\sigma_i + \sigma_j) + f_{corr}(\sigma_i^{\perp} + \sigma_j^{\perp})]$$

Here the charge densities are taken from two different surface elements i and j. This equation also contains two additional free parameters. α' represents the strength of the misfit energy interaction, and can be set with the input argument `alpha_prime`. Likewise f_{corr} represents the strength of the correlation charge density interaction, and can be set with the argument `f_correlation`.

The hydrogen bonding energy E_{hb} utilizes only the local charge densities σ. This energy can be given as:

$$E_{hb}(\sigma_i, \sigma_j) = c_{hb} f_{hb}(T) \max(0, \max[\sigma_i, \sigma_j] - \sigma_{hb}) \min(0, \min[\sigma_i, \sigma_j] + \sigma_{hb})$$

In this equation c_{hb} represents the strength of the hydrogen bonding interaction, while σ_{hb} represents the charge density limit above which interactions are deemed to include hydrogen bonding. These are both free parameters, and can be set with the `c_hydrogen_bonding` and `sigma_hydrogen_bonding` respectively. Scaling of the hydrogen bonding interaction can also be included through the function $f_{hb}(T)$. This can be given as:

$$f_{hb}(T) = \frac{T \ln(1 + \exp(\bar{E}/RT)/200)}{\bar{T} \ln(1 + \exp(\bar{E}/R\bar{T})/200)}$$

Here $\bar{E}$ is a reference energy, set to 20kJ/mol and $\bar{T}$ is a reference temperature, set to 298.15K. This temperature scaling lowers the hydrogen bonding energy at temperatures above 298.15K. As this scaling has no free parameters, it can only be either enabled or disabled. This is done with the argument `hydrogen_bond_temperature`.

Since the σ-potential is calculated by summing over specific surface elements, it is possible to also use the attached atom of each surface segment to determine whether or not to include hydrogen bonding. This can be useful in cases, such as when modeling metal cations, where molecules may have large charge densities from atoms that do not have any hydrogen bonding character. The elements to include as hydrogen bonding donors or acceptors can be specified with the arguments `hydrogen_bond_donors` and `hydrogen_bond_acceptors`, respectively. These both take a list of elements or the flag `All`. The flag `All` indicates that hydrogen bonding is to be included for all surface segments withing the hydrogen bonding range of charge density regardless of attached atom.

Once the σ-potential is calculated, the residual chemical potential μ_X^{res} of a molecule X can simply be calculated by summing the interactions of the solute surface segments with those of the solvent. This is done through the equation:

$$\mu_X^{res} = \frac{RT}{a_{eff}} \sum_{i \in X} s_\nu \bar{\mu}(\sigma_i, \sigma_i^{\perp})$$

The total chemical potential of a solvated molecule is the sum of both the residual and combinatorial chemical potential. This latter chemical potential includes cavitation and entropic terms. The combinatatorial chemical potential μ_X^{comb} can be given as:

$$\mu_X^{comb} = \lambda_{comb} RT \left[\ln(V^X) + \left(1 - \frac{V^X}{V_{av}} - \ln(V^{av})\right) + \left(1 - \frac{A^X}{A_{av}} - \ln(A^{av})\right)\right]$$

Here A^X and V^X are the surface area and volume of the solute species. Likewise A^{av} and V^{av} are the average surface area and volume of the components of the solvent. All areas and volumes are also divided by the reference of 1^2 and 1^3, respectively, to render them dimensionless. The free parameter λ_{comb} sets the relative size of the combinatorial chemical potential. This can be specified with the argument `lambda_comb`.

A number of free parameters are also used in the estimation of the gas phase chemical potential. In COSMO-RS the gas chemical potential of a substance is defined so that the vapor pressure p can be given as:

$$p = p_0 \exp([\mu^{liq} - \mu^{gas}]/RT)$$

Here p_0 is a reference pressure, defined to be 1 bar. The overall equation of the gas phase chemical potential of a molecule μ_X^{gas} can be given as:

$$\mu_X^{gas} = \Delta^X - \sum_k \gamma_k A_k^X - \omega n_{ring}^X + \eta RT$$

This expression contains a number of terms. The first, Δ^X represents the difference in solvation energy between the gas and solvent states. This can be given as:

$$\Delta^X = E^{gas} - E^{COSMO} + \frac{\lambda_{reg}}{2} \sum_i \Phi_i s_\nu (\sigma_i - \sigma_i^*)$$

Here E^{gas} and E^{COSMO} are the total energy of the molecule in the gas and COSMO solvent model, respectively. The COSMO energy includes both the dielectric energy and the polarization of the molecule resulting from the charge of the solvent cavity. The final sum is a consequence of the COSMO dielectric energy being calculated with the raw charges. In COSMO-RS smoothed charges are always used, and so the final sum accounts for the energy associated with smoothing the charge density distribution. Here Φ is the molecular electrostatic potential at each surface element. The free parameter λ_{reg} estimates how much the total COSMO energy is raised by the molecule being polarized by the solvent cavity surface. This can be set with the argument `lambda_reg`.

Going back to the equation for the total gas chemical potential, the second term accounts for the loss of dispersion interactions from the solvent in transferring the molecule to the gas phase. The dispersion energies are simply calculated by scaling the total surface area associated with each element, A_k^X with a scaling factor γ_k. These scaling factors are different for each element. Since the dispersion energy depends only on the solvent cavity, it is the same for any solvent. For that reason it is not included in the solvent chemical potential, but rather in the gas chemical potential. The scaling factors for each element can be set with the argument `dispersion`. This takes a dictionary, where the keys are each element to be set and the value is the scaling factor.

The third term in the gas chemical potential is a special correction for molecules with rings. Here n_{ring} represents the number of atoms in cyclic structures and η is an energy scaling factor. This can be set with the argument `omega_ring`. The final term in the gas chemical potential accounts for the temperature of the gas. Here η is a dimensionless scaling factor. This can be set with the argument `eta_gas`.

When calculating pK_a values, a linear correction can be applied to the value calculated by COSMO-RS to give an estimate of the actual pK_a value[5]. This corrects for both systematic deficiencies in the underlying DFT calculation and the COSMO-RS methodology. Using this linear correction the pK_a can be given as:

$$pK_a = A \times pK_a^{raw} + B$$

Here pK_a^{raw} is the pK_a value directly calculated by COSMO-RS. The factors A and B can be set with the arguments `pka_scale` and `pka_offset`, respectively.

The final parameters in the *CosmoRSParameters* object relate to numerical details in how the COSMO-RS calculation is performed. Since the σ-potential is calculated iteratively, this is done until either the σ-potential reaches a set convergence limit or a maximum number of steps has been reached. The convergence limit can be set with the argument `sigma_potential_tolerance`. Likewise the maximum number of steps can be set with the argument `sigma_potential_max_steps`. In some cases, such as at low temperature, the σ-potential may not converge using the default settings. In these cases either the convergence limit can be raised or the number of steps increased until convergence is reached. When calculating plots of either the σ-profile or the σ-potential the charge density is divided into a number of steps or bins. The argument `bin_size` sets the spacing of these steps. Since the COSMO-RS calculations are done by summing over surface segments, this argument is not used in calculating chemical potentials.

When creating a *CosmoRSParameters* object, any number or combination of parameters may be specified. The default values for all parameters is taken from the *quantumatk_u2022-12* parameter set. The values contained in this parameter set can be shown using the *COSMO-RS Parameters* dialog in the **COSMO-RS Analyzer**.

[5] Andreas Klamt, Frank Eckert, Michael Diedenhofen, and MichaelăE. Beck. First principles calculations of aqueous pKa values for organic and inorganic acids using COSMO-RS reveal an inconsistency in the slope of the pKa scale. *J. Phys. Chem. A.*, 107:9380–9386, 2003. doi:10.1021/jp034688o.

Once a parameter set is created, it can be saved for further use in QuantumATK. Parameter sets can be added to QuantumATK through the *CosmoRSParameterSets* class, which interfaces to a local database of parameter sets. Parameter sets can also be added manually. The method `createDatabaseEntry` creates a representation of the parameter set in a file. The parameter set is then added to the database by placing it in the `.quantumatk/databases/cosmors_parameters/` in your home directory.

CosmoRSSpeciesDatabase

class CosmoRSSpeciesDatabase

categories()

Returns
A list of the chemical categories used in the database.

Return type
list of str

deleteSpecies(*name*)

Delete the the given named entry from the database. This can only be done for user-added entries.

Parameters
name (*str*) – The name of the species to delete.

exportSpecies(*name*)

Return the CosmoRealSpecies for the given named entry.

Parameters
name (*str*) – The name of the species to return.

findSpecies(*category=None*, *formula=None*, *smiles=None*)

Return the list of the names of the available species. Different selection criteria can be used to return subsets of species names.

Parameters

- **category** (*str*) – Return only species that are in the given category.
- **formula** (*str*) – Return only species that have the given chemical formula.
- **smiles** (*str*) – Return only species whose bond graph matches the bond graph given by the SMILES string. Note that this does not distinguish between cis/trans isomers or stereoisomers.

Returns
A list names of species that match the given selection criteria.

Return type
list of str

importSpecies(*name*, *cosmo_data*, *category=None*, *description=None*, *melting_point=None*, *enthalpy_of_fusion=None*, *specific_heat_capacity_change=None*, *antoine_a=None*, *antoine_b=None*, *antoine_c=None*, *boiling_point=None*, *flash_point=None*, *density=None*)

Add a new configuration to the COSMO-RS database.

Parameters

- **name** (*str*) – The name of the data entry.
- **cosmo_data** (*CosmoRS* | *CosmoRealSpecies*) – The COSMO data including solvent and gas phase DFT calculations.
- **category** (*str*) – The classification category of the configuration.

- **description** (*str*) – A text description of the database entry.
- **melting_point** (PhysicalQuantity of type temperature) – The melting point of the pure substance.
- **enthalpy_of_fusion** (PhysicalQuantity of type energy) – The enthalpy or heat of fusion of the pure substance.
- **specific_heat_capacity_change** (PhysicalQuantity of type energy per temperature) – The difference in the molar isobaric specific heat capacity between the liquid and solid phase.
- **antoine_a** (PhysicalQuantity of type pressure) – The A parameter of the Antoine equation. The units indicate the pressure units in the equation.
- **antoine_b** (PhysicalQuantity of type temperature) – The B parameter of the Antoine equation.
- **antoine_c** (PhysicalQuantity of type temperature) – The C parameter of the Antoine equation.
- **boiling_point** (PhysicalQuantity of type temperature) – The boiling point of the pure substance.
- **flash_point** (PhysicalQuantity of type temperature) – Minimum temperature required for the vaporized pure substance to be combustible in air.
- **density** (PhysicalQuantity of type density) – The liquid density of the pure substance.

Usage Examples

Find names of molecules in the COSMO-RS database by category, chemical formula and bond graph.

```
# Load the COSMO database
database = CosmoRSSpeciesDatabase()

# Find all species with the chemical formula C4H8O
species_names = database.findSpecies(formula='C4H8O')
nlprint(species_names)

# Find specifically the molecule tetrahydrofuran
species_names = database.findSpecies(formula='C4H8O', smiles='O1CCCC1')
nlprint(species_names)
```

find_species_example.py

Load water and SO_2 molecules from the database and calculate the solubility of SO_2 in water.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
sulfur_dioxide = database.exportSpecies('sulfur dioxide')

# Determine the gas solubility.
gas_solubility = GasSolubility(
    sulfur_dioxide,
    water,
    solvent_density=1*kiloGram/liter,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Retrieve the Henry constant.
h_cp = gas_solubility.henryConstantCP()[0].convertTo(molar/bar)
```

(continues on next page)

(continued from previous page)

```
nlprint(f'The Henry constant of SO2 in water is {h_cp}')

# Retrieve the Henry volatility.
h_px = gas_solubility.henryVolatilityPX()[0].convertTo(bar)
nlprint(f'The Henry volatility of SO2 in water is {h_px}')
```

water_so2_example.py

Add and then remove an azide (ammonia) molecule.

```
# Create the COSMO-RS object
solvation_energy = SolvationEnergy(
    configuration=azane
)
azane_cosmo_rs = CosmoRS(
    configuration=azane,
    solvation_energy=solvation_energy
)

# Load the database
database = CosmoRSSpeciesDatabase()

# Add the molecule to the database
database.importSpecies(
    'azane',
    azane_cosmo_rs,
    category='Inorganic bases',
    description='Added as example of COSMO-RS molecular database',
    boiling_point=239.81*Kelvin,
    melting_point=195.42*Kelvin,
    density=0.682*kiloGram/liter,
    antoine_a=4.86886*bar,
    antoine_b=1113.928*Kelvin,
    antoine_c=-10.409*Kelvin,
)

# Remove the species from the database
database.deleteSpecies('azane')
```

add_remove_example.py

Notes

The *CosmoRSSpeciesDatabase* object provides a script interface to the molecule database in the **Cosmo-RS Analyzer**. Through this class *CosmoRealSpecies* objects can be exported from the database using the `exportSpecies` method. These *CosmoRealSpecies* objects can then be used directly in a COSMO-RS calculation. *CosmoRealSpecies* or *CosmoRS* objects can also be saved into the database using the `importSpecies` method. As well as the data object, the `importSpecies` method can optionally take a category label, a description or a range of experimental data that may be used in COSMO-RS calculations. These data can also be set when running the COSMO-RS calculation. Configurations that have been added to the database can also be removed by the `deleteSpecies` method. Configuration entries that are shipped with QuantumATK cannot be deleted.

In the database species have a name as a primary key. This is often the common name for the molecule, rather than the

systematic IUPAC name. Specific molecules can be searched for using a combination of classification class, chemical formula and SMILES string in the `findSpecies` method. Classification class and chemical formula match all records with the same data. SMILES strings use graph matching to match all records that have the same bond graph. Note that this does not distinguish between cis and trans isomers. The `findSpecies` method returns the name of all records that match the given criteria.

CosmoRealGas

class CosmoRealGas(*temperature*, *parameters=None*)

Constructor for COSMO-RS gas phase. This can be used to calculate gas interactions in the COSMO-RS model.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The gas temperature.
- **parameters** (*CosmoRSParameters*) – The COSMO real solvent parameters

chemicalPotential(*species*)

Calculate the gas phase chemical potential of the given species.

Parameters
species (*CosmoRealSpecies*) – The species for which the chemical potential is calculated.

Returns
The chemical potential.

Return type
PhysicalQuantity of type energy

parameters()

Returns
The parameters used for the COSMO-RS model.

Return type
CosmoRSParameters

temperature()

Returns
The temperature of the gas.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the gas-phase pseudochemical potential of water with respect to the ideal screened state.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')

# Create the gas.
```

(continues on next page)

(continued from previous page)

```
gas = CosmoRealGas(
    298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Calculate chemical potential of a methanol molecule in solvent water.
chemical_potential = gas.chemicalPotential(water)
nlprint(f'The chemical potential of gas-phase water with respect to ideal screened state␣
↪is {chemical_potential}')
```

water_gas_example.py

Notes

The *CosmoRealGas* object calculates the pseudochemical potential of a gaseous species with respect to the ideal screened state surrounded by a perfect conductor. In COSMO-RS the gas phase chemical potenial is defined so that the vapor pressure p of a substance can be given as

$$p = p_0 \exp([\mu^{liq} - \mu^{gas}]/RT)$$

Here μ is the chemical potential of the species and p_0 is a reference pressure, defined to be 1 bar. For more information on the COSMO-RS formalism, including how the gas phase chemical potential is approximated, please refer to *CosmoRSParameters*. Note that the *CosmoRealGas* is instantiated only with a temperature and a parameter set. The pseudochemical potential of a *CosmoRealSpecies* is then determined by calling the `chemicalPotential` method which takes in the *CosmoRealSpecies* as an argument.

CosmoRealSolid

class CosmoRealSolid(*temperature*, *parameters=None*)

Constructor for COSMO-RS solid phase. This can be used to calculate solid interactions in the COSMO-RS model.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The gas temperature.
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

chemicalPotential(*species*)

Calculate the pseudo-chemical potential of the solid state, using the pure liquid state as reference.

Parameters
species (*:class`~.CosmoRealSpecies`*) – The species for which the pseudo-chemical potential is calculated.

Returns
The pseudo-chemical potential.

Return type
PhysicalQuantity of type temperature

freeEnergyOfFusion(*species*)

Calculate the free energy of fusion of the given species.

Parameters
species (*CosmoRealSpecies*) – The species the free energy is calculated for.

Returns
The Gibbs free energy of fusion.

Return type
PhysicalQuantity of type energy

parameters()

Returns
The parameters used for the COSMO-RS model.

Return type
CosmoRSParameters

temperature()

Returns
The temperature of the solid.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up a water *CosmoRealSolid*. Example data available at Data link.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
water.setMeltingPoint(273.15 * Kelvin)
water.setEnthalpyOfFusion(6 * kiloJoulePerMol)
specific_heat_capacity_change = (4.187 - 2.108) * Joule / (gram*Kelvin)
specific_heat_capacity_change *= water.molarMass()
water.setSpecificHeatCapacityChange(specific_heat_capacity_change)

# Create the solid at 10K below the melting point.
solid = CosmoRealSolid(
    263*Kelvin,
    parameters=CosmoRSParameters(),
)

# Calculate Gibbs free energy of fusion of ice.
dG_fus = solid.freeEnergyOfFusion(water).convertTo(kiloJoulePerMol)
nlprint(f'The free energy of fusion of ice is {dG_fus}')

# Calculate chemical potential of ice.
chemical_potential = solid.chemicalPotential(water).convertTo(kiloJoulePerMol)
nlprint(f'The chemical potential of ice is {chemical_potential}')
```

water_solid_example.py

Notes

The *CosmoRealSolid* object allows calculation of the pseudochemical potential of a solid substance. The solid pseudochemical potential is calculated with reference to the liquid pseudochemical potential such that:

$$\mu^{sol} = \mu^{liq} - \Delta G_{fus}$$

Here μ_{sol} is the solid pseudochemical potential, μ_{liq} is the corresponding liquid chemical potential and ΔG_{fus} is the free energy of fusion of the species. The free energy of fusion cannot be calculated directly by COSMO-RS, and therefore must be approximated. This can be done in a number of ways. In general, the most accurate approximation can be given as:

$$\Delta G_{fus} = \Delta H_{fus}\left(1 - \frac{T}{T_m}\right) - \Delta Cp_{fus}(T_m - T) + \Delta Cp_{fus} T \ln\frac{T_m}{T}$$

Note several pure compound data are required. T_m is the melting pont, ΔH_{fus} the heat of fusion. Likewise, ΔCp_{fus} is the change in molar heat capacity upon melting which corresponds to the difference in molar heat capacity of a compound in its pure liquid and solid state[1]. If ΔCp_{fus} is not available the two last terms may be neglected which gives

$$\Delta G_{fus} = \Delta H_{fus}\left(1 - \frac{T}{T_m}\right)$$

If none of these data is provided the Gibbs free energy of fusion is predicted from a simple linear model fitted at 298K.

$$\Delta G^{i}_{fus}(298K) = 12.2V^{COSMO} - 0.76N_{ring\ atoms} + 0.54\mu_i^{water}$$

The linear coefficients are purely empirical and determined by fitting to experimental data[2]. Note that here μ_i^{water} represents the chemical potential of the given molecule i dissolved in water at 298K. This gives a measure of the polarity of the molecule.

As with *CosmoRealSolvent* and *CosmoRealGas* the temperature of the substance is given as input when the object is created. For meaningful results using *CosmoRealSolid* the given temperature should be below the melting point.

CosmoRealSolvent

class CosmoRealSolvent(*temperature*, *solvent_components*, *parameters=None*)

Constructor for COSMO-RS solvent. This can be used to calculate solvent interactions in the COSMO-RS model.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The solvent temperature.
- **solvent_components** (*MoleculeConfiguration* | *CosmoRealSpecies* | *CosmoRSMixture*) – The solvents chemical components. This can either be a pure compound or a mixture.
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

chemicalPotential(*species*)

Calculate the total chemical potential of the species in the solvent.

Parameters
species (*CosmoRealSpecies*) – The species for which the chemical potential is calculated.

[1] Zhen Song, Jingwen Wang, and Kai Sundmacher. Evaluation of COSMO-RS for solid–liquid equilibria prediction of binary eutectic solvent systems. *Green Energy & Environment*, 6(3):371–379, 2020. doi:10.1016/j.gee.2020.11.020.

[2] Andreas Klamt, Frank Eckert, Martin Hornig, MichaelăE. Beck, and Thorsten Bürger. Prediction of aqueous solubility of drugs and pesticides with COSMO-RS. *Journal of Computational Chemistry*, 23(2):275–281, 2002. doi:10.1002/jcc.1168.

Returns
The chemical potential.

Return type
PhysicalQuantity of type energy

`combinatorialChemicalPotential`(*species*)

Calculate the combinatorial chemical potential of the species in the solvent.

Parameters
species (*CosmoRealSpecies*) – The species for which the chemical potential is calculated.

Returns
The chemical potential.

Return type
PhysicalQuantity of type energy

`moleFractions`()

Returns
The mole fraction of each solvent component.

Return type
Sequence of floats

`parameters`()

Returns
The COSMO-RS parameters

Return type
CosmoRSParameters

`residualChemicalPotential`(*species*)

Calculate the residual chemical potential of the species in the solvent.

Parameters
species (*CosmoRealSpecies*) – The species for which the chemical potential is calculated.

Returns
The chemical potential.

Return type
PhysicalQuantity of type energy

`sigmaPotential`(*min_charge_density=None*, *max_charge_density=None*)

Calculate the sigma potential for the solvent based on the sigma profile. This assumes globally active hydrogen bonding and no orthogonal charges.

Parameters

- **min_charge_density** (*PhysicalQuantity* of type energy per area) – The minimum charge density considered.
- **max_charge_density** (*PhysicalQuantity* of type energy per area) – The maximum charge density considered.

Returns
A tuple of the charge densities and the corresponding sigma potentials respectively.

Return type
tuple

sigmaPotentialPlot(*min_charge_density=None*, *max_charge_density=None*)

Method to calculate and plot the sigma potential.

Parameters

- **min_charge_density** (*PhysicalQuantity* of type energy per area) – The minimum charge density considered.
- **max_charge_density** (*PhysicalQuantity* of type energy per area) – The maximum charge density considered.

Returns
The PlotModel of the sigma potential.

Return type
PlotModel

sigmaPotentialProjection(*charge_densities*)

Project the surface sigma potential onto the given charge density assuming hydrogen bonding is globally active and there are no orthogonal charges.

Parameters
charge_densities (*PhysicalQuantity* of type charge per area) – List of charge densities to project the sigma potential onto.

Returns
The projected sigma potential.

Return type
PhysicalQuantity of type energy per area

solventComponents()

The molecular composition of the solvent.

Returns
The molecular composition of the solvent.

Return type
CosmoRSMixture

solventSigmaProfile(*min_charge_density=None*, *max_charge_density=None*)

Calculate the combined solvent sigma profile.

Parameters

- **min_charge_density** (*PhysicalQuantity* of type energy per area) – The minimum charge density considered.
- **max_charge_density** (*PhysicalQuantity* of type energy per area) – The maximum charge density considered.

Returns
A tuple of the charge densities and the corresponding sigma profiles respectively.

Return type
tuple

solventSpecies()

Returns
The CosmoRealSpecies object for each solvent component.

Return type
Sequence of *CosmoRealSpecies*

surfaceSigmaPotential()

Returns the sigma potential for each of the surface points, including all hydrogen bond and orthogonal charge effects. The corresponding charge densities are also returned. Note that these depend on the points on the solvent surface, and are not evenly spaced

Returns
A tuple of the charge densities and the corresponding sigma potentials respectively.

Return type
tuple

temperature()

Returns
The solvent temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the chemical potential of methanol in water and the σ-potential of solvent water.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
methanol = database.exportSpecies('methanol')

# Create the solvent.
solvent = CosmoRealSolvent(
    298*Kelvin,
    water,
    parameters=CosmoRSParameters(),
)

# Calculate chemical potential of a methanol molecule in solvent water.
chemical_potential = solvent.chemicalPotential(methanol)
nlprint(f'The chemical potential of dissolving methanol in water is {chemical_potential}
↪')

# Get the sigma potential of water.
charge_densities, sigma_potential = solvent.sigmaPotential()

# Get the sigma potential plot of water
plot = solvent.sigmaPotentialPlot()
Plot.show(plot)
```

methanol_water_example.py

Notes

The *CosmoRealSolvent* object allows to calculate a fundamental construct of the COSMO-RS theory, namely the σ-potential of a pure liquid substance or mixture. For more information on the COSMO-RS formalism please refer to *CosmoRSParameters*. The main properties one can obtain using the object are the combinatorial and residual pseudochemical potential (μ_X^{comb} and μ_X^{res}, respectively) which makes up the total pseudochemical potential of a species X in solvent. It is also possible to calculate the solvent σ-profile. This can be given as:

$$p_S(\sigma) = \frac{\sum_i x_i p_i(\sigma)}{\sum_i x_i A_i}$$

Here i represents a component of the solvent S ensemble. A_i is the total surface area, x_i the given mole fraction and p the σ profile[1]. Hence, p_S represents a statistical distribution of surface charge densities of the ensemble.

The σ-potential is also calculated by the *CosmoRealSolvent*. When the object is created the σ-potential of each surface segment is calculated. This is done using both the normal and orthogonal charge densities σ and $\sigma^{\perp}$. Hydrogen bonding can also take into account the associated atom of each surface element, only adding hydrogen bonding when surface elements are connected to appropriate donors and acceptors. These calculated surface sigma potentials are then used to calculate the residual chemical potential of solutes in the solvent. While increasing the accuracy of the calculation, the downside to this approach is that the σ-potential is no longer simply a function of the charge density. This makes displaying the σ-potential difficult, due to its multivariate nature.

As a result the *CosmoRealSolvent* has three different methods for estimating the σ-potential. The first method is to simply calculate the σ-potential assuming no orthogonal charges and every surface element is hydrogen bond active. This is done in the `sigmaPotential` method. A second method is to calculate the σ-potential at a range of charge densities using the calculated surface σ-potential. This takes into account the orthogonal charges and different hydrogen bonding abilities of each surface element, and then projects that onto a set of charge densities. This is done in the `sigmaPotentialProjection` method. Lastly, it is possible to simply return the calculated σ-potential at each surface element along with the associated charge density. This takes into account the orthogonal charges and hydrogen bonding, but may not result in different potentials being given for the same charge density. This is done in the `surfaceSigmaPotential` method.

CosmoRealSpecies

class CosmoRealSpecies(*configuration*, *solvation_energy=None*, *melting_point=None*, *enthalpy_of_fusion=None*, *specific_heat_capacity_change=None*, *antoine_a=None*, *antoine_b=None*, *antoine_c=None*, *boiling_point=None*, *flash_point=None*, *density=None*)

The chemical species to be used in COSMO-RS calculations.

Parameters

- **configuration** (*MoleculeConfiguration* | *SurfaceConfiguration*) – The solute with an attached COSMO DFT calculator.
- **solvation_energy** (*SolvationEnergy* | SolvationEnergyDatabaseContainer) – The solvation energy analysis for the molecule.
- **melting_point** (PhysicalQuantity of type temperature) – The melting point of the pure substance.
- **enthalpy_of_fusion** (PhysicalQuantity of type energy) – The enthalpy or heat of fusion of the pure substance.
- **specific_heat_capacity_change** (PhysicalQuantity of type energy per temperature) – The difference the molar isobaric specific heat capacity between the liquid and solid phase.

[1] Andreas Klamt. The COSMO and COSMO-RS solvation models. *WIREs Comput Mol Sci*, 1(7):699, sep 2011. doi:10.1002/wcms.56.

- **antoine_a** (PhysicalQuantity of type pressure) – The A parameter of the Antoine equation. Note that here the units indicate the units of pressure in the equation.
- **antoine_b** (PhysicalQuantity of type temperature) – The B parameter of the Antoine equation.
- **antoine_c** (PhysicalQuantity of type temperature) – The C parameter of the Antoine equation.
- **boiling_point** (PhysicalQuantity of type temperature) – The boiling point of the pure substance.
- **flash_point** – Minimum temperature required for the vaporized pure substance to be combustible in air.
- **density** (PhysicalQuantity of type density) – The liquid density of the pure compound.

antoineA()

Returns
The A parameter of the Antoine equation. The units indicate the resulting unit of pressure.

Return type
PhysicalQuantity of type pressure

antoineB()

Returns
The B parameter of the Antoine equation.

Return type
PhysicalQuantity of type temperature

antoineC()

Returns
The C parameter of the Antoine equation.

Return type
PhysicalQuantity of type temperature

antoinePressure(*temperature*)

Estimate the partial pressure based on the Antoine equation. If one or more parameters are not set None is returned.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature at which the partial pressure is evaluated.

Returns
The partial pressure.

Return type
PhysicalQuantity of type pressure

boilingPoint()

Returns
The boiling point of the pure substance, if defined, None if not.

Return type
PhysicalQuantity of type temperature

cavitySurfaceArea()

Returns
The cavity surface area.

Return type
PhysicalQuantity of type area

cavityVolume()

Returns
The cavity volume.

Return type
PhysicalQuantity of type volume

configuration()

Returns
The updated COSMO configuration.

Return type
MoleculeConfiguration | *SurfaceConfiguration* | *BulkConfiguration*

density()

Returns
Liquid phase density.

Return type
PhysicalQuantity of type temperature

enthalpyOfFusion()

Returns
The enthalpy of fusion of the molecule, if defined, None if not

Return type
PhysicalQuantity of type energy.

flashPoint()

Returns
Minimum temperature required for the vaporized pure substance to be combustible in air, if defined, None if not.

Return type
PhysicalQuantity of type temperature

gasEnergy()

Returns
The gas phase energy of the molecule

Return type
PhysicalQuantity of type energy

meltingPoint()

Returns
The melting point of the molecule, if defined, None if not

Return type
PhysicalQuantity of type temperature.

molarMass()

Returns
The molar mass of the species.

Return type
PhysicalQuantity of type mass.

setAntoineA(*antoine_a*)

Set the stored A parameter of the Antoine equation. The units indicate the vapor pressure units.

Parameters
enthalpy_of_vaporization (PhysicalQuantity of type pressure) – The Antoine A parameter.

setAntoineB(*antoine_b*)

Set the stored B parameter of the Antoine equation.

Parameters
enthalpy_of_vaporization (PhysicalQuantity of type temperature) – The Antoine B parameter.

setAntoineC(*antoine_c*)

Set the stored C parameter of the Antoine equation.

Parameters
enthalpy_of_vaporization (PhysicalQuantity of type temperature) – The Antoine C parameter.

setBoilingPoint(*boiling_point*)

Set the stored boiling point.

Parameters
boiling_point (PhysicalQuantity of type temperature) – The melting point.

setDensity(*density*)

Set the stored density.

Parameters
density (PhysicalQuantity of type mass per volume) – The density.

setEnthalpyOfFusion(*enthalpy_of_fusion*)

Set the stored enthalpy of fusion.

Parameters
enthalpy_of_fusion (PhysicalQuantity of type energy) – The enthalpy of fusion.

setFlashPoint(*flash_point*)

Set the stored flash point.

Parameters
flash_point (PhysicalQuantity of type temperature) – The flash point.

setMeltingPoint(*melting_point*)

Set the stored melting point.

Parameters
melting_point (PhysicalQuantity of type energy) – The melting point.

setSpecificHeatCapacityChange(*specific_heat_capacity_change*)

Set the stored difference in the liquid and solid isobaric molar specific heat capacity.

Parameters
specific_heat_capacity_change (PhysicalQuantity of type energy per temperature) – The specific heat capacity change.

sigmaProfile(*r_average=None*, *bin_size=None*, *min_charge_density=None*, *max_charge_density=None*, *normalize=False*)

Method to compute the sigma profile.

Parameters

- **r_average** (*PhysicalQuantity* of type length) – The radius used for sigma averaging.

•**bin_size** (*PhysicalQuantity* of type charge per surface area) – The spacing in the sigma profile.

•**min_charge_density** (*PhysicalQuantity* of type charge per surface area) – The minimum charge density range. If not given or greater than the minimum species surface charge density, the minimum species charge density is used. **Default:** Determined by species surface charge density

•**max_charge_density** (*PhysicalQuantity* of type charge per surface area) – The maximum charge density range. If not given or less than the maximum species surface charge density, the maximum species charge density is used. **Default:** Determined by species surface charge density

•**normalize** (*bool*) – Whether or not the sigma profile is normalized. **Default:** `False`

Returns
Tuple of the charge densities and sigma profile for the species.

Return type
tuple

sigmaProfilePlot(*r_average=None*, *bin_size=None*, *normalize=False*)

Method to plot the smoothed sigma profile of the species.

Parameters

•**r_average** (*PhysicalQuantity* of type distance) – The averaging radius used to smooth the charge densities.

•**bin_size** (*PhysicalQuantity* of type charge per surface area.) – The bin size for the sigma profile histogram.

•**normalize** (*bool*) – Whether to normalize the profile or not. **Default:** `False`

Returns
The PlotModel of the sigma profile.

Return type
PlotModel

solventEnergy()

Returns
The solution phase energy of the molecule

Return type
`PhysicalQuantity` of type energy

specificHeatCapacityChange()

Returns
The difference the molar isobaric specific heat capacity between the liquid and solid phase, if defined, `None` if not.

Return type
`PhysicalQuantity` of type energy per temperature

totalSolvationEnergy()

Returns
The difference between the gas and solution phases of the molecule

Return type
`PhysicalQuantity` of type energy

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up a water *CosmoRealSpecies*.

```
# Create the species.
water_species = CosmoRealSpecies(
    water,
    solvation_energy=solvation_energy,
    melting_point=273.15*Kelvin,
    density=1*gram*cm**-3,
)

# Add boiling point data.
water_species.setBoilingPoint(373.15*Kelvin)

# Retrieve sigma profile.
charge_densities, sigma_profile = water_species.sigmaProfile()

# Retrieve the sigma profile plot.
plot = water_species.sigmaProfilePlot()
Plot.show(plot)
```

water_species_example.py

Notes

The *CosmoRealSpecies* object constitutes one of the fundamental objects within the present COSMO-RS implementation. It holds all the DFT information to describe interactions with a solvent, as well as any additional experimental data that may be used. Once the object is instantiated the additional experimental data can subsequently be modified with further information by using the `set`-prefixed methods. With the stored information the *CosmoRealSpecies* object is able to calculate the σ profile of the molecule. The σ profile, along with the corresponding charge densities, can be returned using the `sigmaProfile` method. If the profile is normalized, the profile represents the probability of finding the given charge density. Without normalization the profile represents the total surface area with the given charge density.

Due to the linear equations used to fit charges to the COSMO surface, some unreasonably large charge densities might occur on the COSMO cavity surface. This can be corrected by a charge smoothing process called σ-averaging. This is performed within the *CosmoRealSpecies* object according to the following equation[1]

$$\sigma_i = \frac{\sum_j \frac{q_j}{s_j + s_{av}} \exp\left(\frac{d_{ij}^2}{r_{av}^2}\right)}{\sum_j \frac{s_j}{s_j + s_{av}} \exp\left(\frac{d_{ij}^2}{r_{av}^2}\right)}$$

Here r_{av} is the averaging radius which is one of the parameters in *CosmoRSParameters*. s_{av} is directly related to r_{av} as $s_{av} = \pi r_{av}^2$. s_j and q_j are the COSMO area and surface charge of the $j`thsegment, respectively. : math : `d_{ij}$ is the Cartesian distance between the segments i and j where σ_i is the charge density being smoothed. This smoothing is done whenever the sigma profile is calculated.

[1] Andreas Klamt. *Refinements, parameterization, and the complete COSMO-RS*, pages 109–125. Elsevier, 2005.

CosmoSolvationParameters

class CosmoSolvationParameters(*solvent_dielectric_constant=None*, *solvent_surface_tension=None*, *grid_spacing=None*, *atomic_radii=None*, *atom_grid_density=None*, *hydrogen_grid_density=None*, *minimum_grid_point_area=None*, *shifted_force_cutoff=None*, *shifted_force_damping=None*)

Container class for the parameters used in COSMO solvent model calculations.

Parameters

- **solvent_dielectric_constant** (*float | Conductor*) – The dielectric constant of the solvent. The flag `Conductor` assumes the solvent is a perfect conductor and gives an infinite dielectric constant **Default:** Conductor.
- **solvent_surface_tension** (`PhysicalQuantity` of type force per distance) – The surface tension of the surrounding solvent. Used to calculate the cavitation energy.
- **grid_spacing** (`PhysicalQuantity` of type length) – The spacing of the grid used to map the curvature of the cavity. **Default:** `0.3 Angstrom`
- **atomic_radii** (*dict*) – A dictionary with custom atomic radii to use for different elements. Keys and values are `PeriodicTableElements` and `PhysicalQuantity` respectively.
- **atom_grid_density** (*int*) – The density of points used to generate the COSMO surface segments for non-hydrogen atoms. The total number of grid points is given as $10n^2 + 2$. **Default:** 4, which corresponds to 162 grid points.
- **hydrogen_grid_density** (*int*) – The density of points used to generate the COSMO surface segments for hydrogen atoms. The total number of grid points is given as $10n^2 + 2$. **Default:** 3, which corresponds to 92 grid points.
- **minimum_grid_point_area** (*PhysicalQuantity* of type area.) – The minimum allowed area associated with each grid point. Grid points with areas less than this limit are merged into surrounding grid points. **Default:** `0.1 Angstrom**2`
- **shifted_force_cutoff** (*PhysicalQuantity* of type distance) – The cutoff used for estimating electrostatic interactions with the shifted-force potential. This is only used in surfaces in which shifted-force potentials are used. **Default:** `10 Angstrom`.
- **shifted_force_damping** (*PhysicalQuantity* of type inverse distance) – The damping used for estimating electrostatic interactions with the shifted-force potential. This is only used in surfaces in which shifted-force potentials are used. **Default:** `0.1 Angstrom**-1`.

atomGridDensity()

Returns
The density of points used to generate the COSMO surface segments for non-hydrogen atoms.

Return type
int

atomicRadii()

Returns
A dictionary with custom atomic radii to use for different elements.

Return type
dict | None

gridSpacing()

Returns
The grid spacing of the tesserae grid.

Return type
PhysicalQuantity of type length

hydrogenGridDensity()

Returns
The density of points used to generate the COSMO surface segments for hydrogen atoms.

Return type
int

minimumGridPointArea()

Returns
The minimum allowed grid point area.

Return type
PhysicalQuantity of type area

nlinfo()

Returns
Calculator containing the info.

Return type
dict

shiftedForceCutoff()

Returns
The cutoff used in shifted-force surface electrostatic interactions.

Return type
PhysicalQuantity of type length

shiftedForceDamping()

Returns
The damping used in shifted-force surface electrostatic interactions.

Return type
PhysicalQuantity of type length

solventDielectricConstant()

Returns
The dielectric constant of the solvent.

Return type
float

solventSurfaceTension()

Returns
The surface tension of the solvent.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the optimized geometry and energy of a glycine zwitter ion in water.

```
# Create LCAOCalculator
solvation_parameters = CosmoSolvationParameters(
    solvent_dielectric_constant=78.3553,
    solvent_surface_tension=72.8*dyne/cm
)
calculator = LCAOCalculator(
    solvation_parameters=solvation_parameters
)

# Set Calculator
configuration.setCalculator(calculator)
configuration.update()
nlsave('glycine_zwitter_ion.hdf5', configuration)

# Optimize the geometry
optimized_configuration = OptimizeGeometry(
    configuration=configuration,
    trajectory_filename='glycine_zwitter_ion.hdf5'
)
nlsave('glycine_zwitter_ion.hdf5', optimized_configuration)
```

glycine_zwitter_ion.py

Notes

This object contains the parameters used in a **CO**nductor-like **S**creening **MO**del (COSMO) calculation[1][2]. COSMO is a continuum solvation method that can be added to an LCAO-DFT calculation to model the effect of a solvent surrounding the molecule or interacting with a surface. This is done by creating a discretized surface mesh around the molecule or surface. This mesh is then polarized according to the electrostatic potential from the molecule and the dielectric constant of the solvent. During the SCF optimization of the electron density, the potential from the cavity mesh is included. Similarly, in each SCF step the new electron density is also used to update the mesh charges. In this way the molecule and surrounding solvent are mutually polarized to a self consistent solution.

Mathematically, the total COSMO dielectric energy is given as:

$$E = \frac{1}{2}\mathbf{q}A\mathbf{q} + f(\epsilon)\left(\mathbf{z}B\mathbf{q} + \mathbf{cq}\right)$$

Here the vector $\mathbf{q}$ represents the charges on the cavity mesh. The matrix A is the Coulomb matrix giving the interaction energy between the mesh charges. Likewise B is a Coulomb matrix giving the interaction energy between the atomic nuclear charges and the mesh charges. The vector $\mathbf{c}$ completes the electrostatic interactions giving the interaction between the electron density within the cavity and the mesh charges. This is given as:

$$c_i = \int_V \frac{\rho(\mathbf{r})}{|\mathbf{r}_i - \mathbf{r}|} dr$$

Finally $f(\epsilon)$ is a scaling function of the dielectric constant ϵ. This is given as:

$$f(\epsilon) = \frac{\epsilon - 1}{\epsilon + 0.5}$$

[1] A.ăKlamt and G.ăSchüürmann. COSMO: A new approach to dielectric screening in solvents with explicit expressions for the screening energy and its gradient. *J. Chem. Soc., Perkin Trans. 2*, 99(7):799–805, feb 1993. doi:10.1039/P29930000799.

[2] Andreas Klamt. The COSMO and COSMO-RS solvation models. *WIREs Comput Mol Sci*, 1(7):699, sep 2011. doi:10.1002/wcms.56.

This allows the mesh charges can be calculated by minimizing the dielectric energy. This gives the equation for the charges

$$\mathbf{q} = f(\epsilon)A^{-1}\left(\mathbf{z}B + \mathbf{c}\right)$$

An additional correction is added to the charges to account for the electron density outside the solvent cavity. The correction assumes that the total charge of the cavity surface is the negative of the system charge multiplied by the scaling function. This is implemented by adding a Lagrange multiplier to solving the surface charges that constrains the total charge.

Analytic atomic forces for COSMO are also available. These are calculated by taking the derivatives of the dielectric energy. This enables COSMO to be included in geometry optimizations which can have a significant effect in highly charged configurations, such as molecular zwitter ions. Here the inclusion of the COSMO can stabilize the zwitter ion form, making this more energetically favorable then the neutral molecule.

In the *CosmoSolvationParameters* the dielectric constant ϵ is given using the `solvent_dielectric_constant` argument. This controls the strength of the polarization of the solvent around the molecule, and can be used to approximate different solvents. In the COSMO-RS model, the surrounding solvent is assumed to be a perfect conductor. This can be specified with the flag `Conductor`, which gives an infinite dielectric constant for the solvent corresponding to $f(\epsilon) = 1$.

In addition to the dielectric energy, an estimate of the non-electrostatic energy of solvation, including the cavitation and dispersion-repulsion energies can be included. This is done with a linear regression model that includes the solvent cavity surface area and surface tension as parameters. The solvent surface tension is specified with the argument `solvent_surface_tension`. If a solvent surface tension is not given, the non-electrostatic energy is not calculated, and simply given as zero. This is appropriate when performing calculations the the COSMO-RS model.

One of the important parts of the COSMO method is the solvent cavity construction. In QuantumATK the cavity is constructed by first calculating an pseudo-density isosurface around the solvent facing atoms[3]. This gives a smooth surface that approximates the solvent-accessible surface. Grid points from each atom are then projected out to this pseudo-density isosurface to form the mesh used to solve the COSMO surface charges. Surface area is also assigned to each grid point based on their position on the isosurface.

The construction of COSMO solvent cavity can be modified in a number of ways. The spacing of the grid used to calculate the initial pseudo-density is given with the argument `grid_spacing`. Lowering the distance between grid points places the calculated isodensity surface closer to the true isosurface, at the cost of more computation time and memory. The default value is sufficient for most COSMO calculations. The atomic radii, which determines the size of the COSMO cavity, can also be specified with the `atomic_radii` argument. This takes a dictionary of elements and corresponding atomic radii for the elements whose radii are to be modified. Here again the default radii are optimized for the COSMO method and are suitable for most calculations[4].

Once the pseudo-density surface is created, the COSMO grid points are created by projecting points from each atom out to the surface. The number of these projected points can be set, altering their density on the surface. Different densities are allowed for hydrogen and non-hydrogen atoms, as hydrogen is significantly smaller than other atoms. The arguments to set the number of points are `hydrogen_grid_density` and `atom_grid_density`. These each take an integer that specifies how many times the atomic grid is divided. For a given number n, the number of grid points is $10n^2 + 2$. For hydrogen and non-hydrogen atoms the defaults are 3 and 4 respectively, which corresponds to 92 and 162 points. Increasing the grid density increases the accuracy of the surface charge distribution at the cost of more memory and computational time. As with the other parameters, the defaults here should be sufficient for most calculations. When the points are projected from the atoms, they are assigned a surface area based on the segments of the pseudo-density surface that are closest to the point. Points with a surface area below a set limit are removed, and their surface area is assigned to surrounding grid points. This limit can be set with the argument `minimum_grid_point_araea`. By default this is set to 0.1^2 . Lowering this limit can cause numerical problems with large surface charges resulting in the COSMO calculation.

[3] Andreas Klamt and Michael Diedenhofen. A refined cavity construction algorithm for the conductor-like screening model. *J. Comp. Chem.*, 39(21):1648–1655, 2018. doi:10.1002/jcc.25342.

[4] Andreas Klamt. *Refinements, parameterization, and the complete COSMO-RS*, pages 109–125. Elsevier, 2005.

For surfaces, to account for the 2D periodicity of the structure a damped shifted-force potential is used in place of the normal Coulomb potential[5]. The equation for this potential is:

$$erfcV(r) = q_i q_j \left[\frac{(\alpha r)}{r} - \frac{(\alpha R_c)}{R_c} + \left(\frac{(\alpha R_c)}{R_c^2} + \frac{2\alpha \exp(-\alpha^2 R_c^2)}{\sqrt{\pi} R_c} \right) (r - R_c) \right]$$

Here r is the distance between the points, R_c is a cutoff, beyond which the potential is zero and α is a damping factor. The cutoff and damping factor can be set using the arguments `shifted_force_cutoff` and `shifted_force_damping`, respectively.

When performing COSMO calculations the boundary conditions of the simulation can be important. For neutral molecules it is generally accurate enough to use 3D-FFT boundary conditions. For charged molecules it is recommended to use Multipole boundary conditions with the conjugate gradient Poisson solver *ParallelConjugateGradientSolver*. These boundary conditions correctly model the overall charge of the molecule.

In bulk slab systems, the surface needs to be orientated in the C direction of the unit cell. It is then assumed that the system is only periodic in the A and B direcations. A COSMO surface is added only to the top layer of the surface, as it is assumed that the lower face continues into the bulk material. It is recommended that the bulk slabs 2D-FFT boundary conditions are used for A and B, and Dirichlet and Neumann are used for the bottom and top, respectively, for the C direction. With these boundary conditions the slab should be in the middle of the cell, so that the potential value and gradient have space to reach zero at each end of the cell. The same boundary conditions should also be used with COSMO calculations using a *SurfaceConfiguration*.

CosmoSolventSurface

class CosmoSolventSurface(*configuration*, *grid_spacing=None*, *atomic_radii=None*, *atom_grid_density=None*, *hydrogen_grid_density=None*, *refine_cosmo_grid_points=None*, *minimum_grid_point_area=None*)

Create a surface suitable for a COSMO calculation.

Parameters

- **configuration** (*MoleculeConfiguration* | *SurfaceConfiguration*) – The configuration the grid is being created for.
- **grid_spacing** (`PhysicalQuantity` of type length) – The spacing of the grid used to map the curvature of the cavity. **Default:** `0.3 Angstrom`
- **atomic_radii** (*dict*) – A dictionary with custom atomic radii to use for different elements. Keys and values are `PeriodicTableElements` and `PhysicalQuantity` respectively.
- **atom_grid_density** (*int*) – The density of points used to generate the COSMO surface segments for non-hydrogen atoms. The total number of grid points is given as $10n^2 + 2$. **Default:** 4, which corresponds to 162 grid points.
- **hydrogen_grid_density** (*int*) – The density of points used to generate the COSMO surface segments for hydrogen atoms. The total number of grid points is given as $10n^2 + 2$. **Default:** 3, which corresponds to 92 grid points.
- **refine_cosmo_grid_points** (*bool*) – Whether or not grid points are placed exactly on the pseudo-isodensity surface. **Default:** `False`
- **minimum_grid_point_area** (`PhysicalQuantity` of type area) – The minimum allowed area for a single cavity grid point. Points below this area are combined with other points. **Default:** `0.001 Angstrom**2`

[5] C.ăJ. Fennell and J.ăD. Gezelter. Is the Ewald summation still necessary? Pairwise alternatives to the accepted standard for long-range electrostatics. *J. Chem. Phys.*, 2006. doi:http://dx.doi.org/10.1063/1.2206581.

cavitySurfaceArea()

Returns
The cavity surface area.

Return type
PhysicalQuantity of type area

cavityVolume()

Returns
The cavity volume.

Return type
PhysicalQuantity of type volume

constructSurface(*fast_calc=True*)

Construct the tessellated and COSMO surface grids based on the internal parameters.

Parameters
fast_calc (*bool*) – Whether or not fast approximate pseudo-density calculation is used.

cosmoGridAreas()

Returns
The surface area of each COSMO surface segment.

Return type
PhysicalQuantity of type square distance.

cosmoGridAtomMap()

Returns
The index of the atom that each COSMO grid point is associated with.

Return type
ndarray

cosmoGridCenters()

Returns
The points that define the center of each COSMO surface segment.

Return type
PhysicalQuantity of type distance.

cosmoGridChargeDensities()

Returns
The COSMO charge densities at each point on the surface.

Return type
PhysicalQuantity of type charge per area

cosmoGridCharges()

Returns
The COSMO charges at each point on the surface.

Return type
PhysicalQuantity of type charge

cosmoGridNormals()

Returns
The normal vector pointing out of the cavity.

Return type
ndarray

cosmoGridPotentials()

Returns
The electrostatic potential from the molecule at the surface.

Return type
PhysicalQuantity of type energy per charge

elementalSurfaceArea()

Returns
Return a dictionary listing the surface area assigned to each atom

Return type
dict

minimumGridPointArea()

Returns
The minimum grid point area allowed in constructing the grid.

Return type
PhysicalQuantity of type area

numberOfCosmoPoints()

Returns
The number of COSMO grid points that maps the surface.

Return type
int

numberOfTesserae()

Returns
The number of tesserae that maps the surface.

Return type
int

tesserae()

Returns
The tesserae that define the surface as an N,3,3 array. N is the number of tesserae. The array is indexed by tesserae, then point in the triangle, then Cartesian coordinate.

Return type
PhysicalQuantity of type distance.

tesseraeAreas()

Returns
The surface area of each tessera.

Return type
PhysicalQuantity of type square distance.

tesseraeCenters()

Returns
The points that define the center of each tessera.

Return type
PhysicalQuantity of type distance.

tesseraeChargeDensities(*corners=False*)

The charge density assigned to the tessellated surface.

Parameters
corners (*bool*) – Whether to work with the corners of the tesserae or the center.

Returns
The charge densities of tesserae, based on the density at the closest grid point.

Return type
PhysicalQuantity of type charge per area

tesseraeNormals()

Returns
The unit normal of each tesserae in the cavity

Return type
ndarray

uniqueString()

Return a unique string representing the state of the object.

Notes

The *CosmoSolventSurface* object creates the grid of surface points used in the COSMO solvation model. The method used to construct the surface follows the method outlined by Klamt and Diedenhofen[1]. In this method a pseudo-density isosurface around the molecule is first created. The pseudo-density includes terms that smooths the surface between atoms and avoids the cusps that are typically formed when just spheres are used. The isosurface is created by calculating the pseudo-density on a grid around the molecule which is then used to create tessellations that define the surface. To define the actual COSMO grid, a spherical grid of points centered on each atom is projected out to the atomic radius. The tessellations of the pseudo-density grid are then used to assign surface area to each of these points. Each tesserae is assigned to the closest COSMO grid point. Points that have no associated area, often because they are inside the cavity, are removed from the grid. Each projected point is also moved to the center of the tesserae that are assigned to it.

Solvent surfaces can also be created for bulk slabs in a *BulkConfiguration* or a surface in a *SurfaceConfiguration*. In both cases the surface is assumed to be aligned with the C direction of the unit cell. The same general algorithm can be used for surfaces, with the 2D periodicity of the structure also being taken into account. In bulk slabs a solvent surface is only created on the top side of the surface. The bottom side is assumed to be modelling the effect of the bulk material, and is therefore not solvated.

The construction of COSMO solvent cavity can be modified in a number of ways. The spacing of the grid used to calculate the initial pseudo-density is given with the argument `grid_spacing`. Lowering the distance between grid points places the calculated isodensity surface closer to the true isosurface, at the cost of more computation time and memory. The default value is sufficient for most COSMO calculations. The atomic radii, which determines the size of the COSMO cavity, can also be specified with the `atomic_radii` argument. This takes a dictionary of elements and

[1] Andreas Klamt and Michael Diedenhofen. A refined cavity construction algorithm for the conductor-like screening model. *J. Comp. Chem.*, 39(21):1648–1655, 2018. doi:10.1002/jcc.25342.

corresponding atomic radii for the elements whose radii are to be modified. Here again the default radii are optimized for the COSMO method and are suitable for most calculations[2].

The number of these projected points can be set, altering their density on the surface. Different densities are allowed for hydrogen and non-hydrogen atoms, as hydrogen is significantly smaller than other atoms. The arguments to set the number of points are `hydrogen_grid_density` and `atom_grid_density`. These each take an integer that specifies how many times the atomic grid is divided. For a given number n, the number of grid points is $10n^2+2$. For hydrogen and non-hydrogen atoms the defaults are 3 and 4 respectively, which corresponds to 92 and 162 points. Increasing the grid density increases the accuracy of the surface charge distribution at the cost of more memory and computational time. As with the other parameters, the defaults here should be sufficient for most calculations. When the points are projected from the atoms, they are assigned a surface area based on the segments of the pseudo-density surface that are closest to the point. Points with a surface area below a set limit are removed, and their surface area is assigned to surrounding grid points. This limit can be set with the argument `minimum_grid_point_araea`.

To create a *CosmoSolventSurface* a molecule or surface must be included which defines the configuration that the surface is created for. Building the surface geometry is done by calling the `constructSurface` method on the object. This creates both the tessellated and COSMO grids, but does not add any information about surface charges. These are added during a DFT calculation with the COSMO model. Once the surface is defined and DFT surface charge information is added, the details of the surface can be queried using a number reader methods. These include details such as grid positions, charges, surface areas, surface normals and atom assoiciated with each grid point.

The *CosmoSolventSurface* used in a COSMO DFT calculation can be obtained from an updated configuration using the *SolventSurface* analysis object. This extracts and saves the surface used in the calculation. This surface includes the DFT based information such as surface charges. The *CosmoSolventSurface* object can also be added to the Viewer, allowing the visualization of the surface.

CoulombDSF

class CoulombDSF(*r_cut=None*, *alpha=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **alpha** (*PhysicalQuantity of type length**-1*) – Damping parameter in the DSF method. It determines how fast the Coulomb interactions decay with the particle distance.
- **bonded_mode** – Either CoulombDSF.evaluateAll or CoulombDSF.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equalization method (EEM) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

[2] Andreas Klamt. *Refinements, parameterization, and the complete COSMO-RS*, pages 109–125. Elsevier, 2005.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components
```

(continues on next page)

(continued from previous page)

```
potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O',
                                         r_0=2.1*Angstrom,
                                         k=2.0067*1/Ang,
                                         E_0=0.340554*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=1.0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O',
                                         r_0=3.618701*Angstrom,
                                         k=1.379316*1/Ang,
                                         E_0=0.042395*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=22.0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The **damped shifted force (DSF)** solver[1] represents an efficient method to calculate the electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The Coulomb interactions between two particles i and j are approximated by the following expression for the potential energy:

$$V_{ij}(r) = q_i q_j \left[\frac{\mathrm{erfc}(\alpha r)}{r} - \frac{\mathrm{erfc}(\alpha R_c)}{R_c} + \left(\frac{\mathrm{erfc}(\alpha R_c)}{R_c^2} + \frac{2\alpha}{\pi^{1/2}} \frac{\exp(-\alpha^2 R_c^2)}{R_c} \right) (r - R_c) \right] .$$

[1] C.ăJ. Fennell and J.ăD. Gezelter. Is the Ewald summation still necessary? Pairwise alternatives to the accepted standard for long-range electrostatics. *J. Chem. Phys.*, 2006. doi:http://dx.doi.org/10.1063/1.2206581.

When using *bonded force fields* the bonded_mode parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If CoulombSolver.evaluateAll (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If CoulombSolver.evaluate4 (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombDebye

class CoulombDebye(*r_cut=None*, *kappa=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **kappa** (*PhysicalQuantity of type length**-1*) – Damping parameter in the Debye method. It determines how fast the Coulomb interactions decay with the particle distance.
- **bonded_mode** – Either CoulombDebye.evaluateAll or CoulombDebye.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equalization method (EEM) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit,␣
↪charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',  mass=15.9994*atomic_mass_unit,␣
↪charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O', r_0=2.1*Angstrom, k=2.0067*1/Ang, E_
↪0=0.340554*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O', r_cut=7.5*Angstrom, c=1.
↪0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O', r_0=3.618701*Angstrom, k=1.379316*1/
```

(continues on next page)

(continued from previous page)

```
↪Ang, E_0=0.042395*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O', r_cut=7.5*Angstrom, c=22.
↪0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDebye(r_cut=9.0*Angstrom, kappa=0.5))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *CoulombDebye* solver calculates the electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The electrostatic interactions between two particles i and j are calculated as a screened Coulomb sum:

$$V_{ij}(r) = \frac{q_i q_j}{r} \exp(-\kappa r) \, .$$

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If `CoulombSolver.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `CoulombSolver.evaluate4` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombERFCPotential

class CoulombERFCPotential(*particleType1*, *particleType2*, *beta*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **beta** (*PhysicalQuantity of type length*) – Potential Parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_CoulombERFC')

# Add the particle types to the potential set
```

(continues on next page)

(continued from previous page)

```
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit,␣
↪charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',  mass=15.9994*atomic_mass_unit,␣
↪charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O', r_0=2.1*Angstrom, k=2.0067*1/Ang, E_
↪0=0.340554*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O', r_cut=7.5*Angstrom, c=1.
↪0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O', r_0=3.618701*Angstrom, k=1.379316*1/
↪Ang, E_0=0.042395*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O', r_cut=7.5*Angstrom, c=22.
↪0*Ang**12*eV))

# Add the coulomb pair potentials to the potential set
potentialSet.addPotential(CoulombERFCPotential('Si', 'Si', beta=2.3*Angstrom, r_cut=9.
↪0*Angstrom))
potentialSet.addPotential(CoulombERFCPotential('Si',  'O', beta=2.3*Angstrom, r_cut=9.
↪0*Angstrom))
potentialSet.addPotential(CoulombERFCPotential( 'O',  'O', beta=2.3*Angstrom, r_cut=9.
↪0*Angstrom))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *CoulombERFCPotential* potential can be used to approximately calculate pairwise electrostatic interactions between particles with partial charges, as defined in the *ParticleType*.

The Coulomb interactions between two particles i and j are screened by a *complementary error function*, similar to the real-space part of the *CoulombSPME* method:

$$V_{ij}(r) = \frac{q_i q_j}{r} \operatorname{erfc}(r/\beta_{ij}) \, .$$

In contrast to the **Coulomb solver** classes, which calculate the interactions between *all* charged particles, this method calculates only the interactions between the *specified particle type pairs*.

CoulombErf

class CoulombErf(*r_cut=None*, *a=None*, *alpha=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **bonded_mode** – Either CoulombErf.evaluateAll or CoulombErf.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equilibration (EEQ) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

CoulombErfDSF

class CoulombErfDSF(*alpha=None*, *r_cut=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **alpha** (*PhysicalQuantity of type length**-1*) – Damping parameter in the DSF method. It determines how fast the Coulomb interactions decay with the particle distance.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **bonded_mode** – Either CoulombErfDSF.evaluateAll or CoulombErfDSF.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.

•**doEEM** (*bool*) – Whether or not electronegativity equilibration (EEQ) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

CoulombEwald

class CoulombEwald(*alpha=None*, *epsilon=None*, *bonded_mode=None*)

Constructor of the Coulomb solver.

Parameters

•**alpha** (*PhysicalQuantity of type length**-1*) – Splitting parameter of the Ewald summation.

•**epsilon** (*float*) – The accuracy of the Ewald-summation. It is used to determine the cutoff radius for the short-range interactions such that erfc(alpha * r_cut) <= epsilon.

•**bonded_mode** – Either CoulombEwald.evaluateAll or CoulombEwald.evaluate4.

If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)
Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components
potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=2.4
```

(continues on next page)

(continued from previous page)

```
))
potentialSet.addParticleType(ParticleType(
    symbol='O',
    mass=15.9994*atomic_mass_unit,
    charge=-1.2
))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential(
    'Si',
    'O',
    r_0=2.1*Angstrom,
    k=2.0067*1/Ang,
    E_0=0.340554*eV,
    r_i=6.0*Angstrom,
    r_cut=7.5*Angstrom
))
potentialSet.addPotential(Repulsive12Potential(
    'Si',
    'O',
    r_cut=7.5*Angstrom,
    c=1.0*Ang**12*eV
))
potentialSet.addPotential(MorsePotential(
    'O',
    'O',
    r_0=3.618701*Angstrom,
    k=1.379316*1/Ang,
    E_0=0.042395*eV,
    r_i=6.0*Angstrom,
    r_cut=7.5*Angstrom
))
potentialSet.addPotential(Repulsive12Potential(
    'O',
    'O',
    r_cut=7.5*Angstrom,
    c=22.0*Ang**12*eV
))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombEwald(alpha=0.1*Angstrom**-1, epsilon=1.0e-10))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This Coulomb solver provides an efficient method to calculate the long-range electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The *CoulombEwald* solver works only for periodic systems, i.e. *BulkConfiguration*.

Similar to *CoulombSPME*, the method evaluates a screened, short-range part of the electrostatic interactions as a pair-wise sum in real space, whereas the remaining long-range contributions are calculated in reciprocal space[1].

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If `CoulombSolver.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `CoulombSolver.evaluate4` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombN2

class CoulombN2(*r_cut=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **bonded_mode** – Either CoulombN2.evaluateAll or CoulombN2.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equalization method (EEM) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

[1] P.ăP. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Annalen der Physik*, 369(3):253–287, 1921. doi:10.1002/andp.19213690304.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit,␣
↪charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',  mass=15.9994*atomic_mass_unit,␣
↪charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O', r_0=2.1*Angstrom, k=2.0067*1/Ang, E_
↪0=0.340554*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O', r_cut=7.5*Angstrom, c=1.
↪0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O', r_0=3.618701*Angstrom, k=1.379316*1/
```

(continues on next page)

(continued from previous page)

```
↪Ang, E_0=0.042395*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O', r_cut=7.5*Angstrom, c=22.
↪0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombN2(r_cut=9.0*Angstrom))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This solver calculates the electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The *CoulombN2* and the *CoulombN2Spline* solvers calculate the Coulomb sum between all particle pairs i and j directly:

$$V_{ij}(r) = \frac{q_i q_j}{r}.$$

The *CoulombN2* solver uses a hard cutoff without any smoothing, whereas the *CoulombN2Spline* solver applies a smoothing spline between the inner and the outer cutoff.

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If `CoulombSolver.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `CoulombSolver.evaluate4` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombN2Spline

class CoulombN2Spline(*r_cut=None*, *r_i=None*, *i_degree=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the spline smoothing starts.
- **i_degree** (*int*) – Deprecated. This parameter is not used anymore.
- **bonded_mode** – Either CoulombN2Spline.evaluateAll or CoulombN2Spline.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equalization method (EEM) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
# Calculator
# -----------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit,␣
↪charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',  mass=15.9994*atomic_mass_unit,␣
↪charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O', r_0=2.1*Angstrom, k=2.0067*1/Ang, E_
↪0=0.340554*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O', r_cut=7.5*Angstrom, c=1.
↪0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O', r_0=3.618701*Angstrom, k=1.379316*1/
↪Ang, E_0=0.042395*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O', r_cut=7.5*Angstrom, c=22.
↪0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombN2Spline(r_cut=9.0*Angstrom, r_i=7.0*Angstrom, i_
↪degree=3))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

When using *TremoloXCalculator*, this solver calculates the electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The *CoulombN2* and the *CoulombN2Spline* solvers calculate the Coulomb sum between all particle pairs i and j directly:

$$V_{ij}(r) = \frac{q_i q_j}{r}.$$

The *CoulombN2* solver uses a hard cutoff without any smoothing, whereas the *CoulombN2Spline* solver applies a smoothing spline between the inner and the outer cutoff.

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If `CoulombSolver.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `CoulombSolver.evaluate4` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombOption

class CoulombOption(*permittivity*)

Constructor of the option.

Parameters

permittivity (*PhysicalQuantity of energy * length / charge**2*) – The Coulomb constant 1 / (4 * pi * epsilon0).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

CoulombQTaperPotential

class CoulombQTaperPotential(*particleType1*, *particleType2*, *C*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **C** (*PhysicalQuantity of type energy*) – Potential Parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *CoulombQTaperPotential* potential can be used to calculate approximate pairwise electrostatic interactions between particles with partial charges, as defined in the *ParticleType*.

The Coulomb interactions between two particles i and j are smoothed by a tapering function:

$$V_{ij}(r) = \frac{q_i q_j}{r} f(r) + C(1 + f(r)) \,,$$

with

$$f(r) = 6 \left(\frac{r}{r_c} \right)^5 - 15 \left(\frac{r}{r_c} \right)^4 + 10 \left(\frac{r}{r_c} \right)^3 .$$

In contrast to the **Coulomb solver** classes, which calculate the interactions between *all* charged particles, this method calculates only the interactions between the *specified particle type pairs*.

CoulombReaxFF

class CoulombReaxFF(*r_cut=None*, *r_i=None*, *bonded_mode=None*, *doEEM=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this Coulomb solver.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius of the tapere function.
- **bonded_mode** – Either CoulombReaxFF.evaluateAll or CoulombReaxFF.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.
- **doEEM** (*bool*) – Whether or not electronegativity equilibration (EEQ) is used to determine the charges. It is switched off by default.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

CoulombSPME

class CoulombSPME(*r_cut=None*, *accuracy=None*, *g=None*, *cellratio=None*, *i_degree=None*, *interlaced=None*, *bonded_mode=None*)

Constructor of the Coulomb solver.

Parameters

- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius for the real-space interactions.
- **accuracy** (*float*) – The desired accuracy with respect to the energy. If this value is set and is positive, the values for g, cellratio, i_degree and interlaced are set automatically.
- **g** (*float*) – The splitting coefficient G in the SPME method.
- **cellratio** (*int*) – The number of discretization points in each direction. Must be a power of two.
- **i_degree** (*int*) – The order of the interpolation.
- **interlaced** (*bool*) – Flag to switch interlaced calculation on/off.
- **bonded_mode** – Either CoulombSPME.evaluateAll or CoulombSPME.evaluate4.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate4 is chosen, interactions between particles that are connected by a path of bonds of length less than four are omitted.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)
Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit,␣
↪charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',  mass=15.9994*atomic_mass_unit,␣
```

(continues on next page)

(continued from previous page)

```
↪charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O', r_0=2.1*Angstrom, k=2.0067*1/Ang, E_
↪0=0.340554*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O', r_cut=7.5*Angstrom, c=1.
↪0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O', r_0=3.618701*Angstrom, k=1.379316*1/
↪Ang, E_0=0.042395*eV, r_i=6.0*Angstrom, r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O', r_cut=7.5*Angstrom, c=22.
↪0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombSPME(r_cut=9.0*Angstrom, accuracy=1.0e-06))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This **smooth-particle-mesh-ewald (SPME)** solver[1] provides an efficient method to calculate the long-range electrostatic interactions between particles with partial charges, as defined in *ParticleType*.

The *CoulombSPME* solver works only for periodic systems, i.e. *BulkConfiguration*.

Essentially, the method evaluates a screened, short-range part of the electrostatic interactions as a pair-wise sum in real space, whereas the remaining long-range contributions are calculated in reciprocal space.

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected by less than 4 bonds. If `CoulombSolver.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `CoulombSolver.evaluate4` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one, two, or three consecutive bonds.

CoulombScale14Option

class CoulombScale14Option(*scale_14*)

Constructor of the option.

Parameters
scale_14 (*float*) – The Coulomb scaling factor for bonded 1-4 interactions

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

[1] U.ăEssmann, L.ăPerera, M.ăL. Berkowitz, T.ăDarden, H.ăLee, and L.ăG. Pedersen. A smooth particle mesh Ewald method. *J. Chem. Phys.*, 1995. doi:10.1063/1.470117.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define an OPLS-like potential for a hydrocarbon by adding particle types and interaction functions to the *TremoloX-PotentialSet*. This potential uses a 0.5 1-4 Coulomb scale factor.

```
_option = CoulombScale14Option(
    scale_14=0.5,
)
potentialSet.addOption(_option)
```

Notes

In most bonded forcefields such as OPLS, non-bonding interactions between atoms separated by 3 bonds are treated differently to other interactions. Some of the energy of these interactions may be captured by the torsion potential included in the forcefield, and so a scaling from 0 to 1 may need to be applied so that the combination of non-bonding interactions and torsion terms work together to best represent the internal energy of the molecule.

The **CoulombScale14Option** class sets the scaling for the electrostatic 1-4 terms used in the calculator. The above script shows how to set the scaling to 0.5. By default Coulomb solvers use a scaling factor of 1, meaning that the entire 1-4 Coulomb term is included. In the OPLS forcefield a 1-4 Coulomb scaling of 0.5 is used for both electrostatic and Lennard-Jones interactions. As well as setting the 1-4 scaling, the 1-2 and 1-3 terms are also commonly excluded. In the Coulomb solvers this is done using the `bonded_mode` parameter. By setting it to `CoulombSolver.evaluate4` or mode_14 the 1-2 and 1-3 Coulomb terms are excluded in the calculation.

CoulombSolver

class CoulombSolver

Initialize a Tremolo-X Coulomb solver.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

CrosslinkBuilder

class CrosslinkBuilder(*crosslink_connector*, *restart_trajectory=None*, *skip_initial_crosslink=None*, *nvt_steps_per_crosslink=None*, *npt_steps_per_crosslink=None*, *temperature=None*, *pressure=None*, *isotropic_pressure=None*, *cycles_per_md_step=None*, *cycles_to_terminate=None*, *reaction_percentage=None*, *bond_search_increment=None*, *max_bond_search_radii=None*, *use_fbmc_relax=None*, *fbmc_steps_per_crosslink=None*, *h_passivate_types=None*, *potential_builder=None*, *opls_type_list=None*, ***kwargs*)

Creates a polymer crosslink builder object.

Parameters

- **crosslink_connector** (*CrosslinkConnector*) – The polymer cross-linker that defines the possible reactions.
- **restart_trajectory** (*Trajectory* | None) – A trajectory from which to continue the cross-linking reaction. **Default:** None.
- **skip_initial_crosslink** (*bool*) – Whether or not to skip the initial cross-linking step when starting the simulation. **Default:** False.
- **nvt_steps_per_crosslink** (*int*) – The number of NVT molecular dynamics steps to perform per cycle. **Default:** 10000.
- **npt_steps_per_crosslink** (*int*) – The number of NPT molecular dynamics steps to perform per cycle. **Default:** 10000.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature that the cross-linking is performed at. **Default:** 300K.
- **pressure** (*PhysicalQuantity* of type pressure) – The pressure that the cross-linking is performed at. **Default:** 1 bar.
- **isotropic_pressure** (*bool*) – Whether or not pressure is applied isotropically. **Default:** True.
- **cycles_per_md_step** (*int*) – The number of cross-linking cycles performed between NPT-MD cycles. **Default:** 1.
- **cycles_to_terminate** (*int*) – The number of cycles the simulation will continue without any cross-linking reactions. None does not apply this termination condition. **Default:** None.
- **reaction_percentage** (*float*) – The target reaction percentage at the completion of the reaction. **Default:** 100.
- **bond_search_increment** (*PhysicalQuantity* of type length) – The increment to add to the bond search radius when no reactions are found in a cross-linking step. **Default:** 0.25 Angstrom.
- **max_bond_search_radii** (*PhysicalQuantity* of type length) – The maximum bond search radius to be reached during the cross-linking process. **Default:** 10 Angstrom.

- **use_fbmc_relax** (*bool*) – Whether or not to use force-bias Monte Carlo to relax the system after cross-linking. By default NVT molecular dynamics with a gradual relaxation of the reaction coordinates is used. Setting this variable to True changes the relaxation step to a combination of force-bias Monte Carlo and NVT molecular dynamics. **Default:** False.
- **fbmc_steps_per_crosslink** (*int*) – The number of force-bias Monte Carlo steps to perform per cycle. This step is only performed if force-bias Monte Carlo relaxation is selected with the input use_fbmc_relax. **Default:** 10000.
- **h_passivate_types** (*str | list | None*) – List of tags that specify atoms to be hydrogen passivated at the end of the cross-linking reaction. **Default:** None.
- **potential_builder** (*DreidingPotentialBuilder*|*UFFPotentialBuilder*|OPLSPot entialBuilder) – The potential builder used to create the potential for the cross-linking system. The default is to use the Dreiding potential without electrostatic interactions.
- **opls_type_list** (*list of str | None*) – Optional list of all of the OPLS atom types that can arise during cross-linking. Works only if the OPLSPotentialBuilder is used, and can increase the performance in this case. **Default:** None.

runCrosslinking(*trajectory_filename=None*)

Run the cross-linking simulation.

Parameters
trajectory_filename (*str | None*) – The file where the result trajectory is written.

Returns
The trajectory of the system as it is being cross-linked.

Return type
Trajectory

Usage Examples

Build a network polymer system based on an amine-epoxy reaction.

```
# Load the configuration to be reacted.
configuration = nlread('Amine_Epoxy_Crosslink.hdf5')[-1]

# Specify the reactions for primary amines with an epoxy using the two different␣
↪hydrogens.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H1', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H2': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H2', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H1': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)
```

(continues on next page)

(continued from previous page)

```
# Specify the reaction of a secondary amine with an epoxy.
reaction_c = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_C_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_C_H', 'REACT_C_N', 'REACT_A_O')],
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)

# Define the CrosslinkConnector, using the defined reactions and functional groups to␣
↪find the
# reactive atom groups in the configuration.
crosslink_connector = CrosslinkConnector(
    configuration,
    reactions=[reaction_a, reaction_b, reaction_c],
    functional_groups=[['REACT_A_C', 'REACT_A_O'], ['REACT_B_N', 'REACT_B_H1', 'REACT_B_
↪H2']],
    max_number_of_reactions=100
)

# Define the CrosslinkBuilder using the default Dreiding potential
polymer_builder = CrosslinkBuilder(
    crosslink_connector,
    cycles_to_terminate=None,
    reaction_percentage=100,
    nvt_steps_per_crosslink=1000,
    npt_steps_per_crosslink=1000,
    temperature=480*Kelvin,
    pressure=1*bar,
    bond_search_increment=0.5*Angstrom,
    max_bond_search_radii=6*Angstrom
)

# Run the polymer building simulation.
polymer_builder.runCrosslinking(trajectory_filename='Amine_Epoxy_Thermoset.hdf5')
```

`Amine_Epoxy_Crosslink.py` `Amine_Epoxy_Crosslink.hdf5`

Build a system of linear polymers of poly(ethylene oxide).

```
# Load the configuration to be reacted.
configuration = nlread('PEO_Crosslink.hdf5')[-1]

# Specify the reactions of ethylene glycol using both possible atom arrangements.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_C_A', 'REACT_O_B')],
    remove_atoms=['REACT_O_A', 'REACT_H_A', 'REACT_H_B'],
    bond_cutoff=5*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_O_A', 'REACT_C_B')],
    remove_atoms=['REACT_O_B', 'REACT_H_A', 'REACT_H_B'],
```

(continues on next page)

(continued from previous page)

```
    bond_cutoff=5*Angstrom
)

# Explicitly define the reactive atom groups in the configuration.
reactive_atom_groups = [{'REACT_C_A': 0, 'REACT_O_A': 2, 'REACT_H_A': 8},
                        {'REACT_C_A': 1, 'REACT_O_A': 5, 'REACT_H_A': 9},
                        {'REACT_C_A': 10, 'REACT_O_A': 12, 'REACT_H_A': 18},
                        {'REACT_C_A': 11, 'REACT_O_A': 15, 'REACT_H_A': 19},
                        {'REACT_C_A': 20, 'REACT_O_A': 22, 'REACT_H_A': 28},
                        {'REACT_C_A': 21, 'REACT_O_A': 25, 'REACT_H_A': 29},
                        {'REACT_C_A': 30, 'REACT_O_A': 32, 'REACT_H_A': 38},
                        {'REACT_C_A': 31, 'REACT_O_A': 35, 'REACT_H_A': 39},
                        {'REACT_C_A': 40, 'REACT_O_A': 42, 'REACT_H_A': 48},
                        {'REACT_C_A': 41, 'REACT_O_A': 45, 'REACT_H_A': 49},
                        {'REACT_C_A': 50, 'REACT_O_A': 52, 'REACT_H_A': 58},
                        {'REACT_C_A': 51, 'REACT_O_A': 55, 'REACT_H_A': 59},
                        {'REACT_C_A': 60, 'REACT_O_A': 62, 'REACT_H_A': 68},
                        {'REACT_C_A': 61, 'REACT_O_A': 65, 'REACT_H_A': 69},
                        {'REACT_C_A': 70, 'REACT_O_A': 72, 'REACT_H_A': 78},
                        {'REACT_C_A': 71, 'REACT_O_A': 75, 'REACT_H_A': 79},
                        {'REACT_C_A': 80, 'REACT_O_A': 82, 'REACT_H_A': 88},
                        {'REACT_C_A': 81, 'REACT_O_A': 85, 'REACT_H_A': 89},
                        {'REACT_C_A': 90, 'REACT_O_A': 92, 'REACT_H_A': 98},
                        {'REACT_C_A': 91, 'REACT_O_A': 95, 'REACT_H_A': 99},
                        {'REACT_C_B': 0, 'REACT_O_B': 2, 'REACT_H_B': 8},
                        {'REACT_C_B': 1, 'REACT_O_B': 5, 'REACT_H_B': 9},
                        {'REACT_C_B': 10, 'REACT_O_B': 12, 'REACT_H_B': 18},
                        {'REACT_C_B': 11, 'REACT_O_B': 15, 'REACT_H_B': 19},
                        {'REACT_C_B': 20, 'REACT_O_B': 22, 'REACT_H_B': 28},
                        {'REACT_C_B': 21, 'REACT_O_B': 25, 'REACT_H_B': 29},
                        {'REACT_C_B': 30, 'REACT_O_B': 32, 'REACT_H_B': 38},
                        {'REACT_C_B': 31, 'REACT_O_B': 35, 'REACT_H_B': 39},
                        {'REACT_C_B': 40, 'REACT_O_B': 42, 'REACT_H_B': 48},
                        {'REACT_C_B': 41, 'REACT_O_B': 45, 'REACT_H_B': 49},
                        {'REACT_C_B': 50, 'REACT_O_B': 52, 'REACT_H_B': 58},
                        {'REACT_C_B': 51, 'REACT_O_B': 55, 'REACT_H_B': 59},
                        {'REACT_C_B': 60, 'REACT_O_B': 62, 'REACT_H_B': 68},
                        {'REACT_C_B': 61, 'REACT_O_B': 65, 'REACT_H_B': 69},
                        {'REACT_C_B': 70, 'REACT_O_B': 72, 'REACT_H_B': 78},
                        {'REACT_C_B': 71, 'REACT_O_B': 75, 'REACT_H_B': 79},
                        {'REACT_C_B': 80, 'REACT_O_B': 82, 'REACT_H_B': 88},
                        {'REACT_C_B': 81, 'REACT_O_B': 85, 'REACT_H_B': 89},
                        {'REACT_C_B': 90, 'REACT_O_B': 92, 'REACT_H_B': 98}]

# Define the CrosslinkConnector, using the defined reactions and reactive atom groups.
crosslink_connector = CrosslinkConnector(
    configuration,
    reactions=[reaction_a, reaction_b],
    reactive_atom_groups=reactive_atom_groups,
    max_number_of_reactions=5
)
```

(continues on next page)

(continued from previous page)

```
# Define the CrosslinkBuilder object, using termination after 3 steps without reactions.
polymer_builder = CrosslinkBuilder(
    crosslink_connector,
    nvt_steps_per_crosslink=1000,
    npt_steps_per_crosslink=1000,
    temperature=350*Kelvin,
    pressure=90*bar,
    cycles_per_md_step=2,
    cycles_to_terminate=3,
    bond_search_increment=0.0*Angstrom,
)

# Run the polymer building simulation.
polymer_builder.runCrosslinking(trajectory_filename='PEO_Polymer.hdf5')
```

`PEO_Crosslink.py` `PEO_Crosslink.hdf5`

Note that in both these cases the examples have been created to be illustrative of the process of building the material. For high-quality simulations it may be necessary to increase the number of molecular dynamics steps or, as in the case of PEO, the size of the system.

Notes

The *CrosslinkBuilder* class manages the building and cross-linking of polymer systems by iteratively performing both cross-linking reactions and also controlling the motion of atoms between each cross-linking step[1][2]. In each cycle, relaxation of both the atomic positions and simulation cell can be performed. There are two different relaxation protocols implemented. The default relaxation protocol is based on molecular dynamics. First the position of the transfer atoms are relaxed with a constrained geometry optimization. After this the atomic positions are relaxed using NVT molecular dynamics. To avoid large forces on the reacted atoms, the reaction coordinates are also gradually scaled from reactant to product. The density of the system can then be further relaxed in an NPT molecular dynamics simulation. Using molecular dynamics in both stages allows for the preservation of velocities between cross-linking steps. An alternative relaxation method using Force-Bias Monte Carlo (FBMC) is also available. In this protocol, all atomic positions are initially optimized with a geometry optimization before being relaxed in an FBMC step. Because FBMC is not sensitive to large forces, no scaling of the reaction coordinates is necessary in this step. An NVT molecular dynamics step is then run before relaxing the density with NPT molecular dynamics. In this case, equilibrated velocities are lost during the FBMC step, and so additional NVT is required before NPT molecular dynamics to first equilibrate the velocities.

On initialization, the *CrosslinkBuilder* requires a *CrosslinkConnector* that contains the details of the possible reactions. There are also a number of optional arguments that control how the configuration is evolved between cross-linking steps. The number of NVT and NPT steps in each molecular dynamics cycle is specified with the arguments `nvt_steps_per_crosslink` and `npt_steps_per_crosslink`. If FMBC based relaxation is desired, this can be accessed by setting the argument `use_fbmc_relax` to `True`. The number of FBMC steps can then be set with `fbmc_steps_per_crosslink`. The temperature and pressure used in the dynamics simulations can be set with the arguments `temperature` and `pressure` respectively. The argument `isotropic_pressure` also sets whether pressure is applied isotropically, or independently for the three cell lengths. This can be useful in cases where the material is anisotropic, such as in building thermoset materials at a solid surface. It is also possible to skip some NPT molecular

[1] SpyrosăV. Kallivokas, AristotelisăP. Sgouros, and DorosăN. Theodorou. Molecular dynamics simulations of EPON-862/DETDA epoxy networks: structure, topology, elastic constants, and local dynamics. *Soft Matter*, 15:721, 2019. URL: https://pubs.rsc.org/en/content/articlelanding/2019/sm/c8sm02071j#!divAbstract, doi:10.1039/c8sm02071j.

[2] Chunyu Li and Alejandro Strachan. Molecular simulations of crosslinking process of thermosetting polymers. *Polymer*, 51:6058, 2010. URL: https://www.sciencedirect.com/science/article/abs/pii/S0032386110009225, doi:10.1016/j.polymer.2010.10.033.

dynamics steps at each cycle. This is done with the argument `cycles_per_md_step`. This takes an integer that specifies how many reaction and NVT molecular dynamics steps are done for every NPT molecular dynamics step. Skipping NPT molecular dynamics steps can be appropriate in cases where the density of the system does not significantly change with the progress of the reaction.

There are three separate termination criteria that can be set to determine when the cross-linking reaction process is complete. First, the argument `cycles_to_terminate` specifies a number of steps in which the simulation will terminate if no reactions are performed. If a reaction is performed at any step the counter is returned to zero. The default value of `None` allows for the simulation to continue through any number of unreactive steps. Secondly, the simulation can also terminate when a defined reaction extent is reached, which is specified through the argument `reaction_percentage`. The default value allows for the reaction to go to completion. Finally the reaction can be terminated when a specific bond search radius is reached. Using the argument `bond_search_increment`, an increase in the bond search radius used to determine possible reactions can be added if no reactions are found during a reaction step. Increasing the bond search radius during the simulation prioritizes reactions between closer reactive groups[3]. When the bond search radius reaches the limit specified by `max_bond_search_radii`, the simulation terminates.

To set the potential used to determine the motion of the atoms during the simulation, a `PotentialBuilder` object can be passed in that determines the forcefield used at each step. As the configuration is reacting, the object used must be able to automatically assign atomic forcefield types for the configuration. By default, a Dreiding potential ignoring electrostatic interactions is used. It should be noted that in both the `UFFPotentialBuilder` and `DreidingPotentialBuilder`, electrostatic interactions are calculated in each cycle using the QEq method. This may significantly slow down simulations in network materials which cannot be easily decomposed into a number of molecules. For the OPLS potential it is also possible to specify a list of types for the atoms in different stages of the reaction. This is intended to limit the search range that the `OPLSPotentialBuilder` must use to determine the types and speed up type assignment. If no list is given, the `OPLSPotentialBuilder` will search though all available types to determine the best fits of types to the structure.

Partially complete simulations can be restarted using the argument `restart_trajectory`. This takes a `Trajectory` from a previous calculation created by a *`CrosslinkBuilder`*. Restarting the calculation from the trajectory resets the count of the extent of the reaction and the bond search radius. Note that if a restart trajectory is given, the last frame of the trajectory should also be used to define the *`CrosslinkConnector`* object used to create the *`CrosslinkBuilder`*. When restarting a calculation it is also possible to skip the initial cross-linking step. This is useful in cases where the simulation terminated because of a problem with a reaction. Skipping the initial cross-linking step in the simulation allows for more movement of the atoms and can resolve problems with the reaction. Skipping the initial cross-linking step is done by setting the argument `skip_initial_crosslink` to `True`.

After the reaction is completed the final configuration can be passivated with hydrogen. The argument `h_passivate_types` takes a list of tags of atoms that are to be passivated. This requires that the tags be present in the final configuration. This is useful in cases such as in the reaction of acrylates, where radicals may be left at the conclusion of the simulation. After hydrogen atoms are added, a geometry optimization is then performed on the resulting configuration.

Once the *`CrosslinkBuilder`* object is defined, the actual cross-linking simulation is performed by calling the `runCrosslinking` method. This returns a trajectory of the resulting configuration after every geometry relaxation cycle, as well as the final structure after possible passivation. The argument `trajectory_filename` allows the specification of a file location where the trajectory can be written. If this is not given, the trajectory is stored in memory before being returned.

[3] Vikas Varshney, SoumyaăS. Patnaik, AjităK. Roy, and BarryăL. Farmer. A Molecular Dynamics Study of Epoxy-Based Networks: Cross-Linking Procedure and Prediction of Molecular and Material Properties. *Macromolecules*, 41:6837–6842, 2008. URL: https://pubs.acs.org/doi/10.1021/ma801153e, doi:10.1021/ma801153e.

CrosslinkConnector

class CrosslinkConnector(*configuration*, *reactions*, *functional_groups=None*, *reactive_atom_groups=None*, *max_number_of_reactions=None*, *random_seed=None*)

Create a class for performing cross-linking in a polymer system.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration*) – The configuration that is to be reacted.
- **reactions** (*CrosslinkReaction* | list of *CrosslinkReaction*) – The CrosslinkReaction objects that define the possible reactions.
- **functional_groups** (*list*) – List of lists of atom tags that defines the functional groups in the configuration that can react according to the defined reactions. Cannot be specified together with reactive_atom_groups.
- **reactive_atom_groups** (*list*) – List of dictionaries that identify the atoms in the configuration that react and their reaction tags. These are the same tags used to define atoms in the reaction definitions. Cannot be specified together with functional_groups.
- **max_number_of_reactions** (*int*) – The maximum number of reactions done in a cycle. None allows all available reactions. **Default:** None
- **random_seed** (*int*) – A seed for the random number generator used to select reactions when unequal reaction probabilities are specified.

crosslink(*configuration=None*)

Perform cross-linking on the stored configuration.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration*) – An optional configuration used if the atomic positions of the atoms changes between cross-linking steps. If not specified the current stored configuration is used.

Returns
Dictionary containing the number of reactions, lists of bonds formed and broken and atoms transferred.

Return type
dict

initialConfiguration()

Return a copy of the initial configuration on which the reactive groups are initially set.

Returns
The initial supplied configuration.

Return type
MoleculeConfiguration | *BulkConfiguration*

maximumBondSearchRadius()

Find the maximum bond search radius in the defined reactions.

Returns
The maximum bond search radius from each reaction.

Return type
PhysicalQuantity of type length

numberOfPossibleReactions()

Return the number of possible reactions found during the previous cross-linking step. This can include several reactions involving the same atoms, and so not all reactions may be able to be performed simultaneously.

Returns
Number of possible reactions.

Return type
int

ratioReacted()

Return the ratio of the current number of reactive groups to the initial number. This gives an indication of overall progress of the reaction. Numbers returned are in the range 0 to 1.

Returns
Ratio of reacted groups.

Return type
int

reactedConfiguration()

Return the current reacted configuration that has resulted from the performed cross-linking steps. If no cross-linking has been done the initial configuration is returned.

Returns
The reacted configuration.

Return type
MoleculeConfiguration | *BulkConfiguration*

Usage Examples

Define a cross-linker for use in cross-linking an epoxy-amine system

```
# Load the configuration to be reacted.
configuration = nlread('Amine_Epoxy_Crosslink.hdf5')[-1]

# Specify the reactions for primary amines with an epoxy using the two different␣
↪hydrogens.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H1', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H2': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H2', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H1': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)

# Specify the reaction of a secondary amine with an epoxy.
reaction_c = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_C_N')],
```

(continues on next page)

(continued from previous page)

```
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_C_H', 'REACT_C_N', 'REACT_A_O')],
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)

# Define the CrosslinkConnector, using the defined reactions and functional groups to␣
↪find the
# reactive atom groups in the configuration.
crosslink_connector = CrosslinkConnector(
    configuration,
    reactions=[reaction_a, reaction_b, reaction_c],
    functional_groups=[['REACT_A_C', 'REACT_A_O'], ['REACT_B_N', 'REACT_B_H1', 'REACT_B_
↪H2']],
    max_number_of_reactions=100
)
```

Amine_Epoxy_Crosslink.py Amine_Epoxy_Crosslink.hdf5

Define a cross-linker for creating linear polymers of poly(ethylene oxide)

```
# Load the configuration to be reacted.
configuration = nlread('PEO_Crosslink.hdf5')[-1]

# Specify the reactions of ethylene glycol using both possible atom arrangements.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_C_A', 'REACT_O_B')],
    remove_atoms=['REACT_O_A', 'REACT_H_A', 'REACT_H_B'],
    bond_cutoff=5*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_O_A', 'REACT_C_B')],
    remove_atoms=['REACT_O_B', 'REACT_H_A', 'REACT_H_B'],
    bond_cutoff=5*Angstrom
)

# Explicitly define the reactive atom groups in the configuration.
reactive_atom_groups = [{'REACT_C_A': 0, 'REACT_O_A': 2, 'REACT_H_A': 8},
                        {'REACT_C_A': 1, 'REACT_O_A': 5, 'REACT_H_A': 9},
                        {'REACT_C_A': 10, 'REACT_O_A': 12, 'REACT_H_A': 18},
                        {'REACT_C_A': 11, 'REACT_O_A': 15, 'REACT_H_A': 19},
                        {'REACT_C_A': 20, 'REACT_O_A': 22, 'REACT_H_A': 28},
                        {'REACT_C_A': 21, 'REACT_O_A': 25, 'REACT_H_A': 29},
                        {'REACT_C_A': 30, 'REACT_O_A': 32, 'REACT_H_A': 38},
                        {'REACT_C_A': 31, 'REACT_O_A': 35, 'REACT_H_A': 39},
                        {'REACT_C_A': 40, 'REACT_O_A': 42, 'REACT_H_A': 48},
                        {'REACT_C_A': 41, 'REACT_O_A': 45, 'REACT_H_A': 49},
                        {'REACT_C_A': 50, 'REACT_O_A': 52, 'REACT_H_A': 58},
                        {'REACT_C_A': 51, 'REACT_O_A': 55, 'REACT_H_A': 59},
                        {'REACT_C_A': 60, 'REACT_O_A': 62, 'REACT_H_A': 68},
                        {'REACT_C_A': 61, 'REACT_O_A': 65, 'REACT_H_A': 69},
                        {'REACT_C_A': 70, 'REACT_O_A': 72, 'REACT_H_A': 78},
                        {'REACT_C_A': 71, 'REACT_O_A': 75, 'REACT_H_A': 79},
```

(continues on next page)

(continued from previous page)

```
                        {'REACT_C_A': 80, 'REACT_O_A': 82, 'REACT_H_A': 88},
                        {'REACT_C_A': 81, 'REACT_O_A': 85, 'REACT_H_A': 89},
                        {'REACT_C_A': 90, 'REACT_O_A': 92, 'REACT_H_A': 98},
                        {'REACT_C_A': 91, 'REACT_O_A': 95, 'REACT_H_A': 99},
                        {'REACT_C_B': 0, 'REACT_O_B': 2, 'REACT_H_B': 8},
                        {'REACT_C_B': 1, 'REACT_O_B': 5, 'REACT_H_B': 9},
                        {'REACT_C_B': 10, 'REACT_O_B': 12, 'REACT_H_B': 18},
                        {'REACT_C_B': 11, 'REACT_O_B': 15, 'REACT_H_B': 19},
                        {'REACT_C_B': 20, 'REACT_O_B': 22, 'REACT_H_B': 28},
                        {'REACT_C_B': 21, 'REACT_O_B': 25, 'REACT_H_B': 29},
                        {'REACT_C_B': 30, 'REACT_O_B': 32, 'REACT_H_B': 38},
                        {'REACT_C_B': 31, 'REACT_O_B': 35, 'REACT_H_B': 39},
                        {'REACT_C_B': 40, 'REACT_O_B': 42, 'REACT_H_B': 48},
                        {'REACT_C_B': 41, 'REACT_O_B': 45, 'REACT_H_B': 49},
                        {'REACT_C_B': 50, 'REACT_O_B': 52, 'REACT_H_B': 58},
                        {'REACT_C_B': 51, 'REACT_O_B': 55, 'REACT_H_B': 59},
                        {'REACT_C_B': 60, 'REACT_O_B': 62, 'REACT_H_B': 68},
                        {'REACT_C_B': 61, 'REACT_O_B': 65, 'REACT_H_B': 69},
                        {'REACT_C_B': 70, 'REACT_O_B': 72, 'REACT_H_B': 78},
                        {'REACT_C_B': 71, 'REACT_O_B': 75, 'REACT_H_B': 79},
                        {'REACT_C_B': 80, 'REACT_O_B': 82, 'REACT_H_B': 88},
                        {'REACT_C_B': 81, 'REACT_O_B': 85, 'REACT_H_B': 89},
                        {'REACT_C_B': 90, 'REACT_O_B': 92, 'REACT_H_B': 98}]

# Define the CrosslinkConnector, using the defined reactions and reactive atom groups.
crosslink_connector = CrosslinkConnector(
    configuration,
    reactions=[reaction_a, reaction_b],
    reactive_atom_groups=reactive_atom_groups,
    max_number_of_reactions=5
)
```

PEO_Crosslink.py PEO_Crosslink.hdf5

Notes

The *CrosslinkConnector* class contains the necessary information to describe the possible reactions in a configuration. The class also contains methods to detect which groups are able to react together and to carry out these reactions by altering the bonding. This enables the creation of realistic amorphous structures that are formed through a chemical process. For instance it allows building polymer systems from monomers in a range of geometries, or post-synthetic cross-linking of linear polymer chains. This class can be used in a stand-alone manner to perform a discrete set of reactions on a single configuration. It can also be used together with the *CrosslinkBuilder* class to model the progressive changes in the configuration as reactions occur and the positions of the atoms evolve.

To create a CrosslinkConnector object 3 arguments are required. First the parameter `configuration` takes the configuration in which reactions are being modeled. The reactions themselves are given as the argument `reactions`. The takes a list of *CrosslinkReaction* objects that define all of the possible reactions in the given configuration. Lastly a list of the atoms that are able to react and the reactions they are able to participate in is required. This can be specified in one of two ways. One method is to use the parameter `functional_groups`. This argument takes a list of lists, with each list containing atom tags of the atoms that together form a reactive functional group. As an example, in the epoxy-amine system the epoxy and amine atoms create two different reactive functional groups. These groups are then specified as a list of lists of the tags on the atoms that make up each group. On initialization the CrosslinkConnector

finds all the reactive groups of each type by finding the groups of atoms that are located closest to each other. There can be any number of identical reactive groups in a configuration. The second way to specify the reactive atoms is to explicitly define them. This is done using the argument `reactive_atom_groups`. This argument takes a list of dictionaries, each dictionary representing one reactive group of atoms. The keys and values in the dictionary are the atom tags used in the reaction definitions and the indices of the atoms respectively. As the reacting atom groups are given explicitly, when `reactive_atom_groups` is set, tags for the reacting atoms do not need to be set on the configuration.

Note: The reactions and reactive atoms required by the arguments `reactions` and `reactive_atom_groups` respectively can be generated from the input configuration using a *ReactionTemplate* object. See the documentation of this object for more details.

The *CrosslinkConnector* also accepts two optional arguments on initialization. The first of these, `max_number_of_reactions`, controls the maximum number of reactions that are performed at each cross-linking step. This allows some control over the rate at which reactions are performed. This can be important at the start of a reaction simulation, where often a large number of reactions are possible. Without limiting the number of reactions, the initial structure may be locked into the configuration, as the performed reactions make the structure far more rigid. By default all possible reactions are carried out at each step. A seed to the random number generator used to determine whether or not low probability reactions are carried out can be given through the argument `random_seed`.

Performing a cross-linking step with the CrosslinkConnector is done through using the method `crosslink`. This takes an optional configuration to be reacted as input, and returns the number of reactions that were performed, as well as details of the bond changes as a result of the reactions. Supplying a configuration is useful in cases where the atomic positions of the atoms change in between cross-linking steps. If no configuration is supplied, either the initial configuration or the configuration produced by the last cross-linking step is used. Accessing the stored configurations is done through the methods `initialConfiguration` and `reactedConfiguration`. Information about the progress of the reactions can also be obtained through two methods. The method `ratioReacted` gives the ratio of reactive groups that have undergone a reaction compared to the total number of groups. In cases where the multi-step reactions are being modeled, the newly formed reactive groups are taken into account. The total number of possible reactions found after each reaction step can also be returned using the method `numberOfPossibleReactions`. The maximum bond cutoff radius in the defined reactions can also be returned with the function `maximumBondSearchRadius`.

CrosslinkReaction

class CrosslinkReaction(*add_bonds=None*, *remove_bonds=None*, *transfer_groups=None*, *remove_atoms=None*, *secondary_reaction_map=None*, *bond_cutoff=None*, *angle_exclusion=None*, *exclude_rings_below_size=None*, *relative_reaction_probability=None*)

Defines a cross-linking reaction between a group of atoms.

Parameters

- **add_bonds** (*list | None*) – List of tuples containing pairs of tags of atoms between which bonds are added.
- **remove_bonds** (*list | None*) – List of tuples containing pairs of tags of atoms between which bonds are removed.
- **transfer_groups** (*list | None*) – List of tuples containing triplets of atom tags of atoms to transfer. Order of tuple is atom transferred, atom transferred from then atom transferred to.
- **remove_atoms** (*list*) – A list of atom tags of atoms to be removed after the reaction.
- **secondary_reaction_map** (*dict*) – Map for how to change current types into new types for a following reaction.
- **bond_cutoff** (*PhysicalQuantity* of type length) – The maximum distance between reactive atoms. **Default:** 5*Angstrom

- **angle_exclusion** (*PhysicalQuantity* of type angle) – The minimum bond angle allowed when forming a new bond. **Default:** 30*Degrees
- **exclude_rings_below_size** (*int | All*) – Prevent reactions that form rings below a set size. The flag All excludes all reactions that form rings. The value 0 allows all ring forming reactions. **Default:** All.
- **relative_reaction_probability** (*float*) – Relative probability that reaction is performed. **Default:** 1

addBonds()

Returns
List of tuples containing atom tags of reactive atoms.

Return type
list

angleExclusion()

Returns
The minimum bond angle allowed when forming a new bond.

Return type
PhysicalQuantity of type angle

bondCutoff()

Returns
The maximum distance between reactive atoms.

Return type
PhysicalQuantity of type length

excludeRingsBelowSize()

Returns
The ring exclusion size. The flag All prohibits all ring formation.

Return type
int | All

nlinfo()

Returns
The nlinfo.

Return type
dict

relativeReactionProbability()

Returns
Return the relative probability that the reaction is performed.

Return type
float

removeAtoms()

Returns
A list of atoms tags of atoms to be removed after the reaction.

Return type
list

removeBonds()

Returns
List of tuples containing atom tags of bonds to be removed.

Return type
list

secondaryReactionMap()

Return a dictionary mapping changes from current types into new types for a following reaction.

Returns
A dictionary showing the pairing of reactant and product types.

Return type
dict

totalReactionTags()

Return a set of all the tags used to define the reaction.

Returns
The set of the reaction tags.

Return type
set

transferGroups()

Returns
Return a list of tuples containing atom tags of atoms to transfer. Order of tuple is atom transferred, atom transferred from then atom transferred to.

Return type
list

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define reactions for use in cross-linking an epoxy-amine system

```
# Load the configuration to be reacted.
configuration = nlread('Amine_Epoxy_Crosslink.hdf5')[-1]

# Specify the reactions for primary amines with an epoxy using the two different␣
↪hydrogens.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H1', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H2': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_B_N')],
```

(continues on next page)

(continued from previous page)

```
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_B_H2', 'REACT_B_N', 'REACT_A_O')],
    secondary_reaction_map={'REACT_B_N': 'REACT_C_N', 'REACT_B_H1': 'REACT_C_H', },
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)

# Specify the reaction of a secondary amine with an epoxy.
reaction_c = CrosslinkReaction(
    add_bonds=[('REACT_A_C', 'REACT_C_N')],
    remove_bonds=[('REACT_A_C', 'REACT_A_O')],
    transfer_groups=[('REACT_C_H', 'REACT_C_N', 'REACT_A_O')],
    exclude_rings_below_size=0,
    bond_cutoff=4*Angstrom
)
```

`Amine_Epoxy_Crosslink.py` `Amine_Epoxy_Crosslink.hdf5`

Define reactions for creating linear polymers of poly(ethylene oxide)

```
# Load the configuration to be reacted.
configuration = nlread('PEO_Crosslink.hdf5')[-1]

# Specify the reactions of ethylene glycol using both possible atom arrangements.
reaction_a = CrosslinkReaction(
    add_bonds=[('REACT_C_A', 'REACT_O_B')],
    remove_atoms=['REACT_O_A', 'REACT_H_A', 'REACT_H_B'],
    bond_cutoff=5*Angstrom
)
reaction_b = CrosslinkReaction(
    add_bonds=[('REACT_O_A', 'REACT_C_B')],
    remove_atoms=['REACT_O_B', 'REACT_H_A', 'REACT_H_B'],
    bond_cutoff=5*Angstrom
)
```

`PEO_Crosslink.py` `PEO_Crosslink.hdf5`

The given examples above demonstrate how reactions between groups of atoms in a configuration can be defined. The reaction definitions rely on atom tags that can be assigned to specific atoms. In the first case of the epoxy-amine reaction, the reaction is a 2-step process. Here an epoxy can react with a primary amine to first form a secondary amine and then another epoxy can react with the secondary amine to form a tertiary amine. To model these two stages requires using a secondary reaction map to correctly label atoms for the second reaction. As the resulting structure is also a network structure, all rings are allowed to be formed by reactions creating the three-dimensional network.

In the second example the reactions necessary for linking ethylene glycol into linear polymers of poly(ethylene oxide) is shown. This proceeds through a condensation reaction where the two hydroxy groups react to form an ether and a water molecule. This water molecule can be removed by setting `remove_atoms` argument.

Notes

The CrosslinkReaction class contains details of specific reactions that are performed by the *CrosslinkConnector* and *CrosslinkBuilder* classes. In this class reactions are specified using atomic tags. Tags are associated atoms according to the role that those atoms can play in specific reactions. Using these tags, reactions can be described through defining how the bonding of these atoms changes during the reaction. It is possible to describe changes to any number of atoms during a reaction, allowing for the description of complex reactions.

Reactions are defined when the object is initialized. The initialization function takes several arguments that define the reaction. The necessary argument `add_bonds` describes the bonds that are created during the reaction, while the optional argument `remove_bonds` describes bonds that are destroyed. Both these arguments require lists of tuples that contain the two atom tags between which bonds are added or removed. In reactions where single atoms are removed from one atom and added onto another, the motion of this atom can be described with the argument `transfer_groups`. This argument takes a list of tuples, containing tags of the atom being transferred, the atom it is bonded to in the reactant and then the atom it is bonded to in the product. This option is most commonly used for specifying proton transfer, as in the epoxy-amine example above. In some cases the reactions that are being modeled generate additional unwanted molecules, such as condensation reactions. In these cases the argument `remove_atoms` allows the specification of atoms that are removed from the configuration once they have reacted. In cases where multi-step reactions are to be modeled, atoms may change through a series of reactive types. One example of this is in epoxy-amine reactions, where primary amines first form secondary amines and then tertiary amines. Multi-step reaction pathways can be created with the `secondary_reaction_map` argument. This takes a dictionary which maps how the reactive type of the atoms change after the specific reaction.

The CrosslinkReaction class initialization function also contains some arguments that control when the reaction can happen[1]. The argument `bond_cutoff` sets a maximum bond length for newly formed bonds. Reactions that require creation of bonds longer than this cutoff are excluded. Similarly the argument `angle_exclusion` defines the minimum allowed bond angle that can be created when a new bond is formed. The exclusion of rings and cyclic structures is controlled through the parameter `exclude_rings_below_size`. In the case where two reacting groups are on the same molecule, reacting these two groups will form a ring or a cyclic structure. The argument `exclude_rings_below_size` defines the minimum size of any ring formed for the reaction to be accepted. The default value is `All`, which excludes any reaction that forms a cyclic structure. This value is appropriate in cases where molecules react to form linear polymer chains. A value of 0 allows all reactions that form cyclic structures. A small or zero value of `exclude_rings_below_size` is needed in cases where molecules react to form three dimensional network systems. Finally the relative probability of the reaction can be specified using the `relative_reaction_probability` argument. This is useful in cases where multiple reactions with different kinetics are being modeled. Setting different relative reactive probabilities allows slowing the rate of one reaction relative to another by passing over opportunities to perform that reaction. By default all reactions have a relative probability of 1, meaning they are performed whenever allowed.

[1] LaurenăJ. Abbott, KyleăE. Hart, and CorayăM. Colina. Polymatic: a generalized simulated polymerization algorithm for amorphous polymers. *Theor. Chem. Acc.*, 132:1334, 2013. URL: https://link.springer.com/article/10.1007/s00214-013-1334-z, doi:10.1007/s00214-013-1334-z.

CrystalInterfaceTrainingParameters

class CrystalInterfaceTrainingParameters(*reference_configurations*, *rattle=None*, *sample_size=None*, *atomic_rattling_amplitudes=None*, *random_seed=None*, *optimize=None*, *optimize_geometry_parameters=None*, *surface_termination_0=None*, *surface_termination_1=None*, *plane_0=None*, *plane_1=None*, *buffer_zone=None*, *vacuum=None*, *max_number_of_atoms=None*, *displacement_vector=None*, *thickness_max=None*, *strain_method=None*, *shortest_surface_lattice_vector=None*, *longest_surface_lattice_vector=None*, *minimum_surface_lattice_vector_angle=None*, *max_surface_area=None*, *grid_density=None*, *strain_max=None*, *angle_min=None*, *angle_max=None*, *angle_delta=None*, *layer_tolerance0=None*, *layer_tolerance1=None*, *configurational_descriptor_cutoff_0=None*, *configurational_descriptor_cutoff_1=None*, *log_filename_prefix=None*)

Class for storing parameters for generating a set of interface training configurations.

Parameters

- **reference_configurations** (Sequence of [*BulkConfiguration*]) – Two reference configurations to be used to generate the training set.
- **rattle** (*bool*) – Switch to turn on rattling of the interfaces to generate more configurations. **Default:** False.
- **sample_size** (*int*) – The number of training configurations to generate for each atomic rattling amplitude. **Default:** 5.
- **atomic_rattling_amplitudes** (*PhysicalQuantity* of type length | sequence of *PhysicalQuantity* of type length) – The list of maximum random displacements of the atomic positions. **Default:** `0.1 * Angstrom`.
- **random_seed** (*int*) – The random seed used for generating the displacements. **Default:** Generated automatically.
- **optimize** (bool | *Calculator*) – Switch to turn on optimization of interface configurations using a fast LCAO calculator with SingleZeta basis, KpointDensity(density_a=1.0*Angstrom, density_c=0.0*Angstrom) and NumericalAccuracyParameters(density_mesh_cutoff=60.0*Hartree). Alternatively a calculator object might be supplied to replace the default one. **Default:** False.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters* or None) – Parameters to be passed to a OptimizeGeometry object. **Default:** None.
- **surface_termination_0** (list of *PeriodicTableElement*) – List of elements considered for terminations of the left interface. **Default:** all elements.
- **surface_termination_1** (list of *PeriodicTableElement*) – List of elements considered for terminations of the left interface. **Default:** all elements.
- **plane_0** (*list of lists(3)*) – Planes given by the Miller indices h, k, and l to cleave the left bulk system. **Default:** all low index planes with maximum index 1.
- **plane_1** (*list of lists(3)*) – Planes given by the Miller indices h, k, and l to cleave the right bulk system. **Default:** all low index planes with maximum index 1.
- **buffer_zone** (*PhysicalQuantity* of type length) – Added spacing between interfaces. If the structures are rattled, the buffer should include the rattling amplitude. **Default:** 0.0 * Angstrom.

- **vacuum** (None | *PhysicalQuantity* of type length) – Vacuum to be added around the interface slab. **Default:** 0.0 * Angstrom.
- **max_number_of_atoms** (*int*) – Maximum number of atoms allowed in one interface configuration. **Default:** No restriction.
- **displacement_vector** (*PhysicalQuantity* of type fractional length | None) – This vector is added to the coordinates of the second surface configuration. **Default:** [0.25, 0.25, 0.0].
- **thickness_max** (*PhysicalQuantity* of type length) – Maximum thickness of either material in the final interface. This value is overwritten in case it is smaller than the thickness of two layers. **Default:** 9.0 * Angstrom.
- **strain_method** (*StrainFirst | StrainSecond | StrainBoth*) – Specifies which layer should be strained. If StrainFirst is given, then the first layer will be strained and the second layer will have no strain, while the opposite is true if StrainSecond is chosen. If StrainBoth is given, then each layer is strained to an intermediate lattice.
- **shortest_surface_lattice_vector** (*PhysicalQuantity* of type length) – The minimum length of either surface vector. **Default:** 0*Angstrom.
- **longest_surface_lattice_vector** (*PhysicalQuantity* of type length) – The maximum length of either surface vector. **Default:** 50*Angstrom.
- **minimum_surface_lattice_vector_angle** (*PhysicalQuantity* of type angle) – The smallest allowed angle between the surface lattice vectors.
- **max_surface_area** (*PhysicalQuantity* of type length**2) – The maximum surface area. **Default:** 200.0*Angstrom.
- **grid_density** (*2-tuple of type int*) – The number of times that the a and b lattice vectors of the first configuration should be repeated during the lattice match search. Larger values consider more possible repetitions.
- **strain_max** (*float*) – The largest engineering strain that will be considered a valid match. The default value of 1 corresponds to a 100% deformation. **Default:** 1.0.
- **angle_min** (*PhysicalQuantity* of type angle) – The minimum rotational angle of the second lattice with respect to the first. **Default:** 0.0 * Degrees.
- **angle_max** (*PhysicalQuantity* of type angle) – The maximum rotational angle of the second lattice with respect to the first. **Default:** 180.0 * Degrees.
- **angle_delta** (*PhysicalQuantity* of type angle) – The resolution of the rotational angles of the second lattice. Smaller values lead to a higher resolution search of rotational values. **Default:** 4.0 * Degrees.
- **layer_tolerance0** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.
- **layer_tolerance1** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.
- **configurational_descriptor_cutoff_0** (*float*) – Cut-off value of the ConfigurationalDescriptorBased duplicate detection for material 0. For crystals a large number is recommended to agressively filter duplicates. Range: (0, 1). **Default:** 0.9.
- **configurational_descriptor_cutoff_1** (*float*) – Cut-off value of the ConfigurationalDescriptorBased duplicate detection for material 1. For crystals a large number is recommended to agressively filter duplicates. Range: (0, 1). **Default:** 0.9.
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.

angleDelta()

Returns
The resolution of the rotational angles of the second lattice.

Return type
PhysicalQuantity of type angle

angleMax()

Returns
The maximum rotational angle of the second lattice with respect to the first.

Return type
PhysicalQuantity of type angle

angleMin()

Returns
The minimum rotational angle of the second lattice with respect to the first.

Return type
PhysicalQuantity of type angle

atomicRattlingAmplitudes()

Returns
Maximum rattling intensity.

Return type
PhysicalQuantity of type length | None

bufferZone()

Returns
Added spacing between interfaces.

Return type
PhysicalQuantity of type length | None

cleavePlanes0()

Returns
Planes given by the Miller indices h, k, and l to cleave the left bulk system.

Return type
list of lists(3).

cleavePlanes1()

Returns
Planes given by the Miller indices h, k, and l to cleave the right bulk system.

Return type
list of lists(3).

configuration0()

Returns
A configuration for the left half of the interface.

Return type
configuration_0: *BulkConfiguration*

configuration1()

Returns
A configuration for the right half of the interface.

Return type
configuration_1: *BulkConfiguration*

`configurationalDescriptorCutoff0()`

Returns
ConfigurationalDescriptorBased cut-off value.

Return type
float

`configurationalDescriptorCutoff1()`

Returns
ConfigurationalDescriptorBased cut-off value.

Return type
float

`dataTag()`

Returns
The selection tag added to the data in the training set.

Return type
str

`displacementVector()`

Returns
This vector is added to the coordinates of the second surface configuration.

Return type
PhysicalQuantity of type fractional length

`extractConfigurations()`

Function that extracts the configurations generated by CrystalInterfaceTrainingParameters. The configurations can be inspected before being passed as `TrainningSet` to the MomentTensorPotentialTraining.

Returns
Generated configurations.

Return type
ConfigurationDataContainer

`gridDensity()`

Returns
The number of times that the a and b lattice vectors of the first configuration should be repeated during the lattice match search.

Return type
2-tuple of type int

`layerTolerance0()`

Returns
Layers are detected by grouping atoms by their z-coordinates.

Return type
int

`layerTolerance1()`

Returns
Layers are detected by grouping atoms by their z-coordinates.

Return type
int

`logFilenameIdentifier()`

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

`logFilenamePrefix()`

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | `LogToStdOut` | None

`longestSurfaceLatticeVector()`

Returns
The maximum length of either surface vector.

Return type
PhysicalQuantity of type length

`maxNumberOfAtoms()`

Returns
Maximum number of atoms allowed in one interface configuration.

Return type
int

`maxSurfaceArea()`

Returns
The maximum surface area.

Return type
PhysicalQuantity of type length**2

`minimumSurfaceLatticeVectorAngle()`

Returns
The smallest allowed angle between the surface lattice vectors.

Return type
PhysicalQuantity of type angle

`optimize()`

Returns
Switch to turn on optimization of interface configurations using the LCAO calculator.

Return type
bool

`optimizeGeometryParameters()`

Returns
The optimize geometry parameters.

Return type
OptimizeGeometryParameters

`randomSeed()`

Returns
The random seed used for generating the displacements, or None if it should be generated automatically.

Return type
int | None

`rattle()`

Returns
Switch to turn on rattling of the interfaces to generate more configurations.

Return type
bool | None

`referenceConfigurations()`

Returns
The list of reference configurations to be used to generate the training set.

Return type
list(2) of [*BulkConfiguration*]

`sampleSize()`

Returns
The number of training configurations for each combination of list parameters.

Return type
int

`shortestSurfaceLatticeVector()`

Returns
The minimum length of either surface vector.

Return type
PhysicalQuantity of type length

`strainMax()`

Returns
The largest engineering strain that will be considered a valid match.

Return type
float

`strainMethod()`

Returns
Specifies which layer should be strained.

Return type
StrainFirst | StrainSecond | StrainBoth

surfaceTerminations0()

Returns
List of elements considered for terminations of the left interface.

Return type
list of [*PeriodicTableElement*]

surfaceTerminations1()

Returns
List of elements considered for terminations of the right interface.

Return type
list of [*PeriodicTableElement*]

thicknessMax()

Returns
Maximum thickness of either material in the final interface.

Return type
PhysicalQuantity of type length

uniqueString()
Return a unique string representing the state of the object.

vacuum()

Returns
Vacuum to be added around the interface.

Return type
PhysicalQuantity of type length | None

Usage Examples

Setup of a training set for a copper/iron interface using *CrystalInterfaceTrainingParameters*.

Note: The particular force-field used for optimization in below script is only for demonstration purposes and should be replaced by a higher quality method in actual training.

```
# Set up lattice
lattice = FaceCenteredCubic(3.61496*Angstrom)

# Define elements
elements = [Copper]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration_copper = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
```

(continues on next page)

(continued from previous page)

```
    )

# Set up lattice
lattice = BodyCenteredCubic(2.8665*Angstrom)

# Define elements
elements = [Iron]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration_iron = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define calculator for pre-optimization calculations.
potentialSet = EAM_CuFeNi_2009()
calculator_ff = TremoloXCalculator(parameters=potentialSet)

optimize_geometry_parameters = OptimizeGeometryParameters(
    max_forces=0.1*eV/Ang,
    max_steps=1000,
    max_step_length=0.2*Ang,
    constraints=[FixStrain(x=True, y=True, z=False)],
    trajectory_interval=1,
    optimizer_method=FIRE(),
    enable_optimization_stop_file=True,
    restart_strategy=NoRestart,
    trajectory_filename='interface-optimization-trajectory.hdf5',
)

reference_configurations = [bulk_configuration_copper, bulk_configuration_iron]

training_set = CrystalInterfaceTrainingParameters(
         reference_configurations,
         optimize=calculator_ff,
         optimize_geometry_parameters=optimize_geometry_parameters,
         rattle=True,
         sample_size=2,
         atomic_rattling_amplitudes=[0.3] * Angstrom,
         buffer_zone=0.3 * Angstrom,
         surface_termination_0=Copper,
         surface_termination_1=Iron,
         plane_0=[[1, 1, 1], [0, 1, 1], [0, 0, 1]],
         plane_1=[[0, 1, 1]],
         vacuum=10 * Angstrom,
         max_number_of_atoms=150,
         thickness_max=9*Angstrom,
         shortest_surface_lattice_vector=5 * Angstrom,
```

(continues on next page)

(continued from previous page)

```
        longest_surface_lattice_vector=12 * Angstrom,
        minimum_surface_lattice_vector_angle=60 * Degrees,
        strain_max=0.06,
)

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = KpointDensity(
    density_a=5.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=100.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=MethfesselPaxton(0.2*eV, 1),
    )

iteration_control_parameters = IterationControlParameters(
    tolerance=5e-05,
    damping_factor=0.3,
    number_of_history_steps=12,
    max_steps=200,
    non_convergence_behavior=StopCalculation(),
    )

algorithm_parameters = AlgorithmParameters(
    scf_restart_step_length=0.3*Angstrom,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    algorithm_parameters=algorithm_parameters,
    )

# Set up non-linear coefficients with optimization.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
   perform_optimization=True,
   energy_only=False,
)

# Set up parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
   basis_size=1000,
   outer_cutoff_radii=4.5*Angstrom,
   mtp_filename='mtp_Cu-Fe_interface.mtp',
   non_linear_coefficients_parameters=non_linear_coefficients_parameters,
)

# Set up MTP training.
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_study',
```

(continues on next page)

(continued from previous page)

```
    object_id='training',
    training_sets=training_set,
    calculator=calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters,
    train_test_split=0.8,
    random_seed=13345,
    number_of_processes_per_task=8,
    log_filename_prefix='fit_mtp_Cu-Fe_interface',
)
moment_tensor_potential_training.update()
nlprint(moment_tensor_potential_training)

# The MTP calculator can now be extracted from the MomentTensorPotentialTraining object.
mtp_calculator = moment_tensor_potential_training.momentTensorPotentialCalculators()[0]
```

interface-training-sets.py

Notes

The *CrystalInterfaceTrainingParameters* class can be used to generate training configurations by building interfaces from two crystalline bulk materials using the *InterfaceBuilder*. All parameters of the interface builder are available. For a specific type of interface, cleave planes and surface terminations can be individually selected for each material. Additionally, bulk configurations with two interfaces or slabs with just one interface can be created by adding vacuum. The former approach effectively doubles the interfaces to train on.

For the actual training data generation, a *CrystalInterfaceTrainingParameters* object needs to be passed into a *MomentTensorPotentialTraining* object.

It is recommended to pre-optimize the automatically generated interface configurations by specifying a fast calculator and a *OptimizeGeometryParameters* object. By default the LCAO method with a single zeta basis set is used.

To achieve a more broadly applicable training set, the generated interfaces can be randomly displaced to increase the number of configurations. The original configuration is always part of the set, sample_size determines the number of rattled configurations added. Alternatively the *crystalTrainingRandomDisplacements* protocol can be used with the trajectory of optimized configurations.

In order to speed up the following DFT reference calculations, a maximum thickness and number of atoms can be specified. This provides a very efficient way to generate training data for crystal materials without having to run long, expensive ab-initio MD simulations.

CrystalPropertyValidation

class CrystalPropertyValidation(*configuration*, *calculator*, *calculate_reference_data=True*, *calculate_volume_energy_plot=True*, *calculate_elastic_constants=True*, *calculate_phonon_bandstructure=True*, *calculate_delta_test=True*, *optimize_geometry_parameters=None*, *log_filename_prefix=None*, *dynamical_matrix_filename=None*, *dynamical_matrix_repetitions=(5, 5, 5)*)

Class for validating calculator accuracy based on properties of crystalline materials.

Parameters

- **configuration** (*BulkConfiguration*) – A bulk reference configuration with the reference calculator attached.
- **calculator** (*Calculator*) – The calculator that should be validated.
- **calculate_reference_data** (*bool*) – If the reference data should be calculated.
- **calculate_volume_energy_plot** (*bool*) – If the equation of state should be calculated and compared.
- **calculate_elastic_constants** (*bool*) – If the elastic moduli should be calculated and compared.
- **calculate_phonon_bandstructure** (*bool*) – If the phonon bandstructure should be calculated and compared.
- **calculate_delta_test** (*bool*) – If the *DeltaTest* should be calculated. Can only be done if the parameter *calculate_reference_data* is also True.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters* | None) – Parameters to used to optimize the geometry.
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this analysis. **Default:** None.
- **dynamical_matrix_filename** – The full or relative path to save the results for the dynamical matrix calculation. A suffix is added for the reference calculator and the calculator to be validated so that results can be saved to two separate files. See *nlsave()*. **Default:** None.
- **dynamical_matrix_repetitions** (`Automatic` | list of ints) – The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. If `use_wigner_seitz_scheme` is set to `True` the only values allowed for repetition are `[1, 1, 1]` or `Automatic`. **Default:** [5, 5, 5]

calculator()

Returns
The calculator to be validated.

Return type
Calculator

deltaTest()

Returns
The result of the delta test.

Return type
float

dynamicalMatrixRepetitions()

Returns
The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers.

Return type
list of ints

elastic()

Returns
Elastic constant calculated by the test calculator.

Return type
ElasticConstants

energy()

Returns
List of energies calculated for different volumes by the test calculator.

Return type
list

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

optimizeGeometryParameters()

Returns
The optimize geometry parameters.

Return type
OptimizeGeometryParameters

phononBandstructure()

Returns
The phonon bandstructure obtained with the test calculator.

Return type
PhononBandstructure

plotEnergyvsVolume()

Show a plot of the equation of state (energy vs. volume) curve calculated with relative and reference calculators.

referenceConfiguration()

Returns
The configuration used with the reference calculator.

Return type
BulkConfiguration

referenceElastic()

Returns
Elastic constant calculated by the reference calculator.

Return type
ElasticConstants

referenceEnergy()

Returns
List of energies calculated for different volumes by the reference calculator.

Return type
list

`referencePhononBandstructure()`

Returns
The phonon bandstructure obtained with the reference calculator.

Return type
PhononBandstructure

`referenceVolume()`

Returns
List of unit cell volumes included in the reference calculation.

Return type
list

`setCalculator(calculator)`

Set the calculator that should be tested against the reference calculator and calculate the properties.

Parameters
calculator (*Calculator*) – The calculator that should be validated.

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`testConfiguration()`

Returns
The configuration used with the test calculator.

Return type
BulkConfiguration

`uniqueString()`

Return a unique string representing the state of the object.

`volume()`

Returns
List of unit cell volumes included in the validation calculation.

Return type
list

Usage Examples

Calculate different properties of crystalline silicon with a Tersoff potential and a Moment Tensor Potential, print the results, and save the object to a hdf5 file:

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Setting up the reference calculator.
# A cheap force-field calculator is used so that this example script
# does not take too long.
potentialSet = Tersoff_Si_1988()
reference_calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(reference_calculator)

# Setting up the calculator that should be validated against the reference calcultor.
potentialSet = MTP_Si_2019b()
calculator = TremoloXCalculator(parameters=potentialSet)

crystal_property_validation = CrystalPropertyValidation(
    bulk_configuration,
    calculator,
    calculate_reference_data=True,
    calculate_volume_energy_plot=True,
    calculate_elastic_constants=True,
    calculate_phonon_bandstructure=True,
    calculate_delta_test=True,
    optimize_geometry_parameters=None,
    log_filename_prefix=None,
    dynamical_matrix_filename='dynamical_matrix.hdf5',
    dynamical_matrix_repetitions=(5, 5, 5),
)

nlsave('crystal_property_validation.hdf5', crystal_property_validation)
nlprint(crystal_property_validation)
crystal_property_validation.plotEnergyvsVolume()
```

si_crystal_property_validation.py

This example script shows how to read the output file. The script also shows how to use existing data to validate with

a number of different calculators.

```
crystal_property_validation = nlread('crystal_property_validation.hdf5')[0]

potentialSet = QuantumATK_MTP_Si_SiO2_2022()
calculator1 = TremoloXCalculator(parameters=potentialSet)

potentialSet = StillingerWeber_SiOCF_2005()
calculator2 = TremoloXCalculator(parameters=potentialSet)

for calculator in [calculator1, calculator2]:
    crystal_property_validation.setCalculator(calculator)
    nlsave('crystal_property_validation.hdf5', crystal_property_validation)
```

`multiple_calculator_crystal_property_validation.py`

Notes

Crystal properties, such as lattice parameters and the bulk modulus can be calculated using a reference calculator and compared to one or more other calculators that needed to be validated. To view these results use the *nlprint* method.

Before calculating the crystal properties of a given *BulkConfiguration*, the cell vectors are optimized using the *OptimizeGeometry* function with the reference and relative calculators. The optimization parameter can be provided using *OptimizeGeometryParameters*.

The *PhononBandstructure* can be visualized by selecting the output file in the **Lab Floor** of QuantumATK unpack using **Unpack** and opening it using the **Phonon Bandstructure Analyzer**.

The equation of state can be plotted using the `plotEnergyvsVolume` method or by unpacking in the **Lab Floor**.

Note: This class is only intended for validating with bulk configurations. It can not be used with molecules or surfaces.

CrystalStructurePrediction

class CrystalStructurePrediction(*initial_population*, *number_of_generations*, *selection_pressure=None*, *number_of_elites=None*, *number_to_promote=None*, *heredity_probability=None*, *permutation_probability=None*, *mutation_probability=None*, *sigma_lattice=None*, *max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *external_pressure=None*, *rng=None*, *write_population=None*, *population_format=None*, *log_filename_prefix=None*, *permutation_operator=None*, *heredity_operator=None*, *mutation_operator=None*, *kpoint_density=None*, *fitness_function=None*, *fitness_tolerance=None*, *active_learning=None*)

Function to perform a crystal structure search by applying a genetic algorithm on a given initial population.

Parameters

- **initial_population** (list of *BulkConfiguration* objects | `Population`) – A list of initial BulkConfigurations or a Population object.
- **number_of_generations** (*int*) – The number of times the population should be evolved.

•**selection_pressure** (*float*) – The pressure applied during the selection of the parents from the population. The higher the pressure the less likely it is to pick an individual with lower fitness. **Default:** 2.0.

•**number_of_elites** (*int*) – Only consider the number_of_elites fittest individuals when making the selection. If None, consider all individuals during selection. **Default:** None

•**number_to_promote** (*int*) – The number of fittest individuals to promote to the next generation. **Default:** 1.

•**heredity_probability** (*float*) – The unnormalized probability of using the heredity operator. **Default:** 0.6.

•**permutation_probability** (*float*) – The unnormalized probability of using the permutation operator. The permutation operator will only be used when there is more than one type of an element in the system. **Default:** 0.0.

•**mutation_probability** (*float*) – The probability of using the mutation operator. **Default:** 0.4.

•**sigma_lattice** (*float*) – The standard deviation of the random strain applied by the mutation operator. **Default:** 0.7.

•**max_forces** (*PhysicalQuantity* in units of force) – The maximum forces determining when the optimization should stop. **Default:** 0.01*eV/Angstrom

•**max_stress** (*PhysicalQuantity* in units of pressure) – The maximum stress determining when the optimization should stop. **Default:** 0.1*GPa

•**max_steps** (*uint*) – The maximum number of steps for local optimization. **Default:** 1000

•**max_step_length** (*PhysicalQuantity* in units of length) – The maximum step length the local optimizer may take. **Default:** 0.2*Angstrom

•**external_pressure** (*PhysicalQuantity* in units of pressure) – The target external pressure of the system. **Default:** 0*GPa

•**rng** (*numpy.random.RandomState*) – The random number generated used to generate the random number stream. If None, rng will be new random number generator, seeded from OS entropy pool **Default:** None.

•**write_population** (*bool*) – Enables saving the configuration of each individual configuration during the crystal structure prediction. **Default:** True.

•**population_format** (*str*) – Sets the storage format for populations. If None, the format is hdf5. **Default:** None.

•**log_filename_prefix** (*str*) – The filename prefix to use for logging the individual optimizations. **Default:** generation_.

•**permutation_operator** (*PermutationOperator.*) – The permutation operator.

•**heredity_operator** (*HeredityOperator.*) – The heredity operator.

•**mutation_operator** (*MutationOperator.*) – The mutation operator.

•**kpoint_density** (*PhysicalQuantity* of type length | None) – The target k-point sampling density. If None, a fixed k-point grid is used. **Default:** None.

•**fitness_function** (*callable*) – By default, the fitness is defined as the negative enthalpy of the system. This function allows for the fitness to be customized. The fitness_function should accept two arguments, configuration and enthalpy , and return the fitness (larger values are more fit). The fitness should be given as a floating point number (without units). The configuration argument is the geometry optimized individual to evaluate the fitness for. **Default:** None.

•**fitness_tolerance** (*float*) – Configurations whose fitness values differ by more than this amount are considered difference structures. This is used as part of the niching step, which removes duplicate configurations from the population. **Default:** 0.02.

•**active_learning** (*ActiveLearningSimulation*) – Can be used to specify an ActiveLearningSimulaiton object, which is then used to run the geometry optimization while training an MTP. The calculator on the configuration is ignored in this case. Note, this does not support restarting from file. **Default:** None.

activeLearning()

Returns
The active learning object, if specified, otherwise None.

Return type
ActiveLearningSimulation | None

evolvePopulation(*number_of_generations*)

Evolves the current population for `number_of_generations` generations.

Parameters
number_of_generations (*int*) – The number of times the genetic algorithm is applied to the population.

fitnesses()

Returns
Return the fitnesses of the current population sorted by descending fitness.

Return type
list of type float

kpointDensity()

Returns
The k-point density.

Return type
PhysicalQuantity of type length | None

population()

Returns
Return all configurations in the population sorted by descending fitness.

Return type
`Population`

symmetries()

Return the international symbol for each individual of the current population sorted by descending fitness.

Returns
a list of international symbols

Return type
list of type str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Determine the global minimum energy crystal structure for TiO2.

```
# ------------------------------------------------------------
# Initial Population
# ------------------------------------------------------------
population_size=10
```

(continues on next page)

(continued from previous page)

```
initial_population = generateInitialPopulation(
    elements=elements,
    population_size=population_size,
    calculator=calculator,
    volume=206.29*Angstrom**3,
    max_forces=0.01*eV/Angstrom,
    max_stress=0.1*GPa,
    max_steps=1000,
    max_step_length=0.2*Angstrom,
    external_pressure=0.0*GPa,
    log_filename_prefix="initial_population_",
)

# ------------------------------------------------------------
# Crystal Structure Prediction
# ------------------------------------------------------------

global_optimization = CrystalStructurePrediction(
    initial_population,
    number_of_generations=16,
    heredity_probability=0.85,
    mutation_probability=0.1,
    permutation_probability=0.05,
    number_to_promote=2,
    number_of_elites=int(0.6*population_size),
    selection_pressure=3.0,
    max_forces=0.005 * eV/Angstrom,
)
```

csp_TiO2.py

Find the crystal structure that most closely matches a target band gap, that is also stable (low enthalpy).

```
# Define custom fitness function.
def fitnessFunction(configuration, enthalpy):
    # Calculate the band gap.
    band_gap = Bandstructure(
        configuration=configuration,
        route=['G','G'],
        points_per_segment=3,
    ).indirectBandGap()

    # Target band gap.
    target_band_gap = 1.34 * eV

    # High fitness is low enthalpy and a band gap near the target. A parameter
    # alpha (ranged from zero to one) controls the relative importance of
    # enthalpy vs band gap error.
    alpha = 0.5
    fitness = -alpha*enthalpy - (1 - alpha) * abs(band_gap - target_band_gap)

    # The fitness should not have units.
    return fitness.inUnitsOf(eV)
```

(continues on next page)

(continued from previous page)

```
global_optimization = CrystalStructurePrediction(
    initial_population,
    number_of_generations=16,
    heredity_probability=0.85,
    mutation_probability=0.1,
    permutation_probability=0.05,
    number_to_promote=2,
    number_of_elites=int(0.6*len(initial_population)),
    selection_pressure=3.0,
    max_forces=0.01 * eV/Angstrom,
    fitness_function=fitnessFunction,
)
```

Run a crystal structure prediction in active learning.

```
# Read the initial training dataset for the active learning simulation.
dataset = nlread('Initial_training_data.hdf5', MomentTensorPotentialTraining)[-1]
initial_training_data = [TrainingSet(dataset, recalculate_training_data=False)]

# Define the MTP fitting parameters.
nl_parameters = NonLinearCoefficientsParameters(
    perform_optimization=False,
    initial_coefficients=Random,
)

fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=800,
    outer_cutoff_radii=4.5*Ang,
    mtp_filename='MTP_CSP_AL.mtp',
    non_linear_coefficients_parameters=nl_parameters,
    forces_cap=500.0*eV/Ang,
    use_element_specific_coefficients=True,
)

# Set up the active learning simulation.
al = ActiveLearningSimulation(
    fitting_parameters=fitting_parameters,
    initial_training_data=initial_training_data,
    mtp_study_filename='MTP_training_CSP_AL.hdf5',
    mtp_study_object_id='training',
    reference_calculator=calculator,
    candidate_threshold=1.0,
    retrain_threshold=3.0,
    check_interval=1,
    max_forces_check=10.0*eV/Ang,
    minimum_bond_length_percent=0.2,
    candidate_trajectory_filename='MTP_training_CSP_AL_candidates.hdf5',
    restart_simulation=False,
    extrapolation_selection_parameters=ExtrapolationSelectionParameters(
        extrapolation_grade_algorithm=QueryByCommittee,
```

(continues on next page)

(continued from previous page)

```
        committee_size=6,
        descriptor_cutoff=0.2,
        descriptor_basis_size=100,
    ),
)

# Read the initial population from file.
initial_population = nlread('Initial_population.hdf5', Population)[0]

# Run the crystal structure predicition with active learning.
global_optimization = CrystalStructurePrediction(
    initial_population,
    number_of_generations=10,
    selection_pressure=2.0,
    number_of_elites=10,
    number_to_promote=4,
    heredity_probability=50.0,
    permutation_probability=15.0,
    mutation_probability=35.0,
    sigma_lattice=0.7,
    max_forces=0.01*eV/Angstrom,
    max_stress=0.1*GPa,
    max_steps=1000,
    max_step_length=0.2*Angstrom,
    external_pressure=0*GPa,
    active_learning=al,
    log_filename_prefix='csp_al',
)
```

csp_active_learning_example.py

Notes

The goal of crystal structure prediction is to explore the state space of a compound and locate the structure with the maximum fitness for a given chemical composition. In QuantumATK, the default fitness function is set to the negative enthalpy, i.e. the algorithm converges to the structure with the minimal enthalpy (or energy if no external pressure is applied). The primary input to the calculation is the chemical formula, which determines the number of atoms in the unit cell. It is important that enough atoms are present in the unit cell to represent the crystal structure. Just because the correct stoichiometric ratio is used, does not ensure that it is possible to represent the minimum energy structure.

Custom Fitness Function

The default fitness function is the negative enthalpy. A custom fitness function may be provided using the `fitness_function` argument. The function takes two arguments: the minimized configuration and the enthalpy. It should return the fitness of the configuration (larger fitness means more favorable structure).

Custom fitness functions can be used to find structures that yield desired material properties or match known experimental values. Care must be taken to ensure that the resulting structures are physically plausible. This means that the enthalpy should be included in the fitness. Practically, this means that the fitness should be a weighted sum of the desired physical properties and the enthalpy. Otherwise, the evolutionary algorithm will find very high energy structures that are not stable or possible to synthesize.

Algorithm Details

The crystal structure prediction algorithm in QuantumATK uses a genetic algorithm. The genetic algorithm begins with an initial population of structures provided by the user. The initial population can be generated using the *generateInitialPopulation* function. Each step of the algorithm involves generating the next population of structures from the previous one. This evolution process is done in the following manner:

(A)Promote the `number_to_promote` fittest individuals to the next generation.

(B)Loop until there are `population_size` individuals in the new generation:

1.Pick which genetic operation should be performed. Each operation is given a user defined probability of being selected.

2.Select one (mutation and permutation operator) or two parent structures (heredity operator). The selection probability is weighted by the fitness of the parental structures.

3.Execute the genetic operation and create a new structure.

(C)Perform a local optimization of each individual of the population.

There are four types of genetic operators available:

- **Promotion**: this operation is always performed at the beginning of the evolution process to ensure that the best `number_to_promote` individuals are kept.
- **Mutation**: perturb the lattice vectors and atomic positions of the chosen configuration randomly.
- **Permutation**: swap the position of atoms of differing elements.
- **Heredity**: combine two parental structures to a new one. A random cut plane is used to divide the atoms of the two parent structures; all atoms below the plane in one structure are combined with the atoms above the plane in the other structure to create a new child structure. Care is taken to ensure that the total number of each element is not changed.

The algorithm stops after evolving the population `number_of_generations` times. Given the stochastic nature of the algorithm, this stopping criterion does not ensure that one has found the global solution. The stability of the search should be checked at least with respect to the chosen seed of the pseudo random number generator (`rng`), the `initial_population` and `number_of_generations`.

After generating a new population, each individual of the population is away from a local minimum in enthalpy, hence a geometry optimization must be performed to converge to a close local minimum. Since the global minimum is also a local minimum, one is guaranteed to find it for sufficiently large numbers of `number_of_generations`. These geometry (local) optimizations comprise nearly all of the computational work during crystal structure prediction.

Note: One key difference between a genetic algorithm and a random search is that the selection of the parent structure(s), on which the genetic operations are performed, is biased by the fitness of that structure. The higher the fitness (i.e. the lower the calculated enthalpy in this implementation), the more likely it is to pick this structure. The skewness of the probability distribution is altered by `selection_pressure`. A large value of `selection_pressure` makes it less likely to select a configuration with a lower fitness.

Restart support

A crystal structure prediction run can be restarted / resumed from a previous point in the following manner:

```
CSP = CrystalStructurePrediction(
    ...
    )

# Save the current state of the CSP object.
nlsave('my_file.hdf5', CSP, object_id='my_run')

...

# Read back a stored CSP object. The last population is restored
# and all given properties / attributes are set up.
CSP = nlread('my_file.hdf5', object_id='my_run')

# Run 10 more iterations of the genetic algorithm.
CSP.evolvePopulation(10)

# Run 5 more iterations of the genetic algorithm.
CSP.evolvePopulation(5)
```

Note: At this time it is not possible to resume runs that use a custom fitness function.

Analyzing Results

If `write_population` is `True` then after each step of the genetic algorithm the current population will be written to disk. The filename will start with `log_filename_prefix` and include the current generation number. For example, the initial population will be saved as `generation_0.hdf5` and after the first step a new file `generation_1.hdf5` will be saved. The crystal structures contained in these files can be viewed in QuantumATK using the Viewer.

Crystal Structure Prediction with Active Learning

Active learning simulation can be activated in crystal structure prediction by passing an *ActiveLearningSimulation* object via the parameter `active_learning`. This will use `runOptimizeGeometry` in the active learning simulation to optimize the individuals. This type of optimization can smooth the MTP based potential energy surface and can easily drag the system towards the local minimum.

Note that the calculator on the initial population is ignored in this case, and the trained MTP is used instead. Since the MTP is evolving during the active learning simulation, it is recommended to re-run the crystal structure prediction simulation multiple times, starting from the previous active learning state (using `restart_simulation=True` in *ActiveLearningSimulation*), until no more training configurations are added. Using multiple process groups for crystal structure prediction is currently not supported when run with active learning.

Since optimization trajectories tend to be shorter and configurations can change significantly between optimization steps, it is recommended to set the `check_interval` parameter to 1 and lower the candidate- and retrain thresholds to add sufficiently many new structures to the training set.

CrystalThermoChemistry

class CrystalThermoChemistry(*configuration*, *dynamical_matrix=None*, *qpoints=None*)

Thermo chemistry analysis of crystal configurations.

Parameters
configuration (*BulkConfiguration*) – The configuration.

:param dynamical_matrix:The dynamical matrix used to calculate the vibrational modes. :type dynamical_matrix:*DynamicalMatrix*

Parameters
qpoints (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The q-points for which the phonon density of states should be calculated. **Default:** The Monkhorst-Pack grid defined by the dynamical matrix repetitions.

energy()

Returns
The total energy.

Return type
PhysicalQuantity of type energy

entropy(*temperature*)

Calculate the entropy at the given temperature.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature to evaluate the entropy at.

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

freeEnergy(*temperature*)

The Helmholtz free energy including all contributions.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature.

Returns
The free energy at the specified temperature.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

qPointsGrid()

The q-points grid.

Returns
A grid object.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

zeroPointEnergy()

The zero point energy.

Returns
The zero point energy.

Return type
PhysicalQuantity of type energy

CurrentDensity

class CurrentDensity(*configuration=None*, *kpoints=None*, *energies=None*, *energy_weights=None*, *spin=None*, *electrode_temperatures=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*)

Constructor for the CurrentDensity object.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration to calculate the current density for.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The kpoints to integrate over. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.
- **energies** (list of *PhysicalQuantity* of type energy.) – The energies to integrate over. **Default:** An energy range that covers the bias window as specified by the `DoubleContourIntegralParameters` on the calculator attached to the configuration.
- **energy_weights** (`Current` | `Left` | `Right` | list of floats) – The weight of each energy point. The keyword `Current` gives an energy dependent weight, $f_R(E) - f_L(E)$, corresponding to the current integral. Left/Right gives an energy independent weight of 1, for current carrying states from Left/Right. **Default:** `Current`
- **spin** (`Spin.All` | `Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.RealUpDown` | `Spin.ImagUpDown` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin for which the current density should be calculated. **Default:** `Spin.Sum` for UNPOLARIZED, else `Spin.All`
- **electrode_temperatures** (list of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used the current calculation. Must be given as 2 temperatures - for example, [100, 200]*Kelvin. **Default:** The temperature used for the self-consistent calculation.

•**self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used for the current density. **Default:** RecursionSelfEnergy(storage_strategy=NoStorage())

•**energy_zero_parameter** (AverageFermiLevel | AbsoluteEnergy) – Specifies the choice for the energy zero. **Default:** AverageFermiLevel

•**infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the self-energy calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** 1.0e-6*eV.

electrodeFermiLevels()

Returns
The Fermi levels of the left and right electrodes in absolute energies.

Return type
PhysicalQuantity of type energy.

electrodeFermiTemperatures()

Returns
The Fermi temperature of the left and right electrodes.

Return type
PhysicalQuantity of type temperature.

energies()

Returns
The energies used in this current density

Return type
PhysicalQuantity of type energy

energyWeights()

Returns
The energy weights used in this current density

Return type
numpy.array

energyZero()

Returns
The energy zero used for the energy scale in this current density.

Return type
PhysicalQuantity of type energy.

evaluate(*x*, *y*, *z*, *spin=None*)
Evaluate in the point x,y,z.

Parameters

•**x** (*PhysicalQuantity* with type length) – The cartesian x coordinate.

•**y** (*PhysicalQuantity* with type length) – The cartesian y coordinate.

•**z** (*PhysicalQuantity* with type length) – The cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The vector grid value at the specified point for the given spin. For `Spin.All`, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity

`gridCoordinate`(*i*, *j*, *k*)

Return the grid coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coornate of the given grid index.

Return type
PhysicalQuantity of type length.

`infinitesimal`()

Returns
The infinitesimal used for calculating the current density.

Return type
PhysicalQuantity of type energy

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`scale`(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`shape`()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

`spin()`

Returns
The spins used in this current density.

Return type
tuple of spin component flags.

`spinProjection(spin=None)`

Query method to get a spin component of the VectorGridValue object.

Parameters
spin (`Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z` | `Spin.RealUpDown` | `Spin.ImagUpDown`) – The spin component for which to return the GridValues object. **Default:** `Spin.Sum`

Returns
A new `VectorGridValues` object for the specified spin.

Return type
`VectorGridValues`

`toArray()`

Returns
The values of the grid as a numpy array slicing off any units.

Return type
`numpy.array`.

`uniqueString()`

Return a unique string representing the state of the object.

`unit()`

Returns
The unit of the data in the grid.

Return type
A physical unit.

`unitCell()`

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

`volumeElement()`

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the *CurrentDensity* for a *DeviceConfiguration* and plot it density along z, integrated over the x,y plane:

```
# Calculate the current density at the fermi level from right to left going states
cd = CurrentDensity(
    configuration,
    energies = [0.0]*eV,
    energy_weights = Right,
    kpoints=MonkhorstPackGrid(1,11),
    )

# Calculate the volume element and x,y integration area
dX, dY, dZ = cd.volumeElement()
dAXY = numpy.linalg.norm( numpy.cross(dX,dY) )

# Calculate the current density along z integrated over x,y
shape = cd.shape()
# Take out spin up (index 0) z component (index 2) in every grid point.
cd_z = numpy.array( [ [ [ cd[i,j,k][0][2] for k in range(shape[2])] for j in␣
↪range(shape[1])] for i in range(shape[0]) ] )

# Sum across x and y dimensions
cd_z_integrated = numpy.array([cd_z[:,:,k].sum() * dAXY for k in range(shape[2])])


# Plot the data
import pylab
pylab.figure()
dz = dZ.norm()
z = numpy.arange(shape[2])*dz.inUnitsOf(Bohr)
pylab.plot(z,cd_z_integrated)
pylab.show()
```

current_density1.py

Calculate the *CurrentDensity* from right to left going states at the Fermi level. Plot the current density in the x-z plane as a contour plot:

```
from QuantumATK import *

# Calculate the current density at the fermi level from right to left going states
cd = CurrentDensity(
    configuration,
    energies = [0.0]*eV,
    energy_weights = Right,
    kpoints=MonkhorstPackGrid(1,11),
    )

# Calculate the volume element
dX, dY, dZ = cd.volumeElement()
dx = dX.norm()
dy = dY.norm()
dz = dZ.norm()
```

(continues on next page)

(continued from previous page)

```
# Calculate the current density along z integrated over x,y
shape = cd.shape()
# Take out spin up (index 0) z component (index 2) in every grid point.
cd_z = numpy.array( [ [ [ cd[i,j,k][0][2] for k in range(shape[2])] for j in␣
↪range(shape[1])] for i in range(shape[0]) ] )

# Sum across y dimensions
cd_xz = numpy.array([ [cd_z[i,:,k].sum() * dy for k in range(shape[2])] for i in␣
↪range(shape[0] )] )


# Make 2-d arrays for contour plot of the data
shape = cd.shape()
x = dx.inUnitsOf(Ang)*numpy.arange(shape[0])
z = dz.inUnitsOf(Ang)*numpy.arange(shape[2])
Z, X = numpy.meshgrid(z,x)
F = numpy.array(cd_xz).reshape(numpy.shape(Z))

#plot the current density in the (z,x) plane
import pylab
pylab.xlabel('z (Angstrom)',fontsize=12,family='sans-serif')
pylab.ylabel('x (Angstrom)',fontsize=12,family='sans-serif')
pylab.contourf(Z, X, F)
pylab.colorbar()
pylab.savefig('cd.png',dpi=100)
pylab.show()
```

current_density2.py

Notes

The current density is given by

$$\mathbf{J}(r) = -\frac{e\hbar}{4\pi m}\int dE \sum_{\mu,\nu} D_{\mu,\nu}(E)\phi_\nu(r)\nabla\phi_\mu(r)(f_L(E) - f_R(E)),$$

where $D(E) = G(E)\,\Gamma_{L/R}(E)\,G^\dagger(E)$, $\phi_{\nu/\mu}(r)$ are atomic basis orbitals, and $f_{L/R}(E)$ are the left and right Fermi functions. G is the retarded Green function, $\Gamma_{L/R}$ the broadening function from either left or right electrode[1].

- The positive current direction is from left to right (corresponding to electron propagation from right to left).
- Through the `energy_weights` keyword, it is possible for the user to define the spectral weight $w(E) = f_R(E) - f_L(E)$. This is particular useful when using the current density to analyze zero bias calculations. Setting the `energy weights` to `Right` is a short cut for setting the `energy_weights` to 1 for all states Setting the `energy weights` to `Left` is short cut for setting the `energy_weights` to -1 for all states. Note that a positive weight corresponds to calculating right to left going modes (positive current) while a negative weight corresponds to calculating left to right going modes (negative current).
- The current density does not include the non-local potential correction[Page 2575, 1] and spill in contributions from the electrodes. Thus, the area integrated current density is not constant along z but has a small fluctuating component from the non-local potential and larger errors near the electrodes from spill in components.

[1] C.ăLi, L.ăWan, Y.ăWei, and J.ăWang. Definition of current density in the presence of a non-local potential. *Nanotechnology*, 19(15):155401, 2008. URL: http://stacks.iop.org/0957-4484/19/i=15/a=155401.

CustomExchangeCorrelation

class CustomExchangeCorrelation(*exchange_correlation_functional*, *hubbard_term=None*, *number_of_spins=1*, *spin_orbit=None*, *dft_half_enabled=None*)

The *ExchangeCorrelation* class is used to define the type of exchange-correlation used in the calculation, and whether the calculation should be polarized or non-polarized.

Parameters

- **exchange_correlation_functional** (CustomExchangeCorrelationFunctional) – The exchange correlation to be used.
- **hubbard_term** (Onsite | OnsiteShell | Dual | DualShell | None) – The Hubbard term to add. **Default:** None
- **number_of_spins** (1 | 2 | 4) – The number of spins to be used in the calculation. **Default:** 1
- **spin_orbit** (*bool*) – If True spin-orbit coupling is considered. **Default:** False
- **dft_half_enabled** (*bool*) – Whether or not DFT-1/2 should be enabled. **Default:** False

cParameter(*grid_descriptor=None*)

return the param C.

Parameters
grid_descriptor (*NLEngine.GridDescriptor*) – The grid descriptor.

cloneAsPolarized()

Create a new functional with all the same settings, but polarized spin.

cloneAsSpinOrbit()

Create a new functional with all the same settings, but with Spin-Orbit coupling included.

correlation()

Returns
The exchange functional class

Return type
CustomExchangeCorrelationFunctional

densityCheckThreshold()

Returns
The density safe guard.

Return type
float

dftHalfEnabled()

Returns
Whether DFT-1/2 is enabled.

Type
bool

dielectricDependentHybridFunctional()

Returns
True, then a dielectric dependent hybrid functional is used.

Return type
bool

dielectricDependentHybridParameters()

Returns
The dielectric dependent hybrid hybrid parameters.

Return type
list

exchange()

Returns
The exchange functional class

Return type
CustomExchangeCorrelationFunctional

exchangeCorrelationFunctional()

Returns
The exchange functional class

Return type
CustomExchangeCorrelationFunctional

hubbardTerm()

Returns
The Hubbard term used.

Return type
Onsite | OnsiteShell | Dual | DualShell | None

maximumPotential()

Set maximum potential value allowed for the TB09. The value to use as the maximum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

minimumPotential()

Set minimum potential value allowed for the TB09. The value to use as the minimum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dicts

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a string representation of the *ExchangeCorrelation* instance.

Parameters
stream (A stream that supports strings being written to using *write*.) – The stream the exchange-correlation should be written to. **Default:** `sys.stdout`

numberOfSpins()

Returns
The number of spins.

Return type
`1 | 2 | 4`

params()

Get the parameters

spinOrbit()

Returns
Boolean determining if spin-orbit is enabled (True) or not (False).

Return type
bool.

spinType()

Returns
The spin type.

Return type
`NLEngine.POLARIZED | NLEngine.NONCOLLINEAR | NLEngine.UNPOLARIZED`

uniqueString()

Return a unique string representing the state of the object.

unscreenedExchange()

Whether the exchange functional in use is unscreened.

useLocalCParameter()

Returns
True, then a local c-parameter is used.

Return type
bool

D3Potential

class D3Potential(*file=None*, *S6=None*, *a1=None*, *S8=None*, *a2=None*, *S6r=None*, *r_cut=None*, *bond_rcut=None*, *triple_rcut=None*, *k3=None*, *taper_factor=None*, *ATM=None*)

Constructor of the potential.

Parameters

- **file** (*str*) – The name of the file that contains the D3 parameters.
- **S6** (*float*) – The dimensionless BJ/Z parameter S6
- **a1** (*float*) – The dimensionless BJ parameter a1 By setting S6r = 0 and a1!=0 or a2!=0 one chooses BJ damping

•**S8** (*float*) – The dimensionless BJ/Z parameter S8

•**a2** (*PhysicalQuantity of type length*) – The BJ parameter a2 By setting S6r = 0 and a1!=0 or a2!=0 one chooses BJ damping

•**S6r** (*float*) – The dimensionless Z parameter S6r By setting S6r != 0 and a1=a2=0 one chooses Z damping

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius for the interactions

•**bond_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the coordination numbers

•**triple_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the ATM interaction

•**k3** (*float*) – The dimensionless k3 smoothing parameter

•**taper_factor** (*float*) – The dimensionless taper_factor for bond_rcut and r_cut

•**ATM** (*bool*) – The parameter to switch on the Axilrod-Teller-Muto three-body terms

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a D3 potential for a SiC/SiO_2 interface to be used with an *MTPPotential* by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [9.228, 0.0, 0.0]*Angstrom
vector_b = [4.613999999999999, -2.663894142040934, 0.0]*Angstrom
vector_c = [0.0, 0.0, 9.185292029012857]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Carbon, Carbon, Carbon, Silicon, Silicon,
            Silicon, Carbon, Carbon, Carbon, Oxygen, Oxygen, Oxygen, Silicon,
```

(continues on next page)

(continued from previous page)

```
            Oxygen, Silicon]

# Define coordinates
fractional_coordinates = [[-0.            ,  0.666666666667,  0.            ],
                          [ 0.333333333333,  0.666666666667,  0.            ],
                          [ 0.666666666667,  0.666666666667,  0.            ],
                          [-0.            ,  0.666666666667,  0.211586087177],
                          [ 0.333333333333,  0.666666666667,  0.211586087177],
                          [ 0.666666666667,  0.666666666667,  0.211586087177],
                          [ 0.            ,  0.333333333333,  0.274787126204],
                          [ 0.333333333333,  0.333333333333,  0.274787126204],
                          [ 0.666666666667,  0.333333333333,  0.274787126204],
                          [ 0.            ,  0.333333333333,  0.486373213382],
                          [ 0.333333333333,  0.333333333333,  0.486373213382],
                          [ 0.666666666667,  0.333333333333,  0.486373213382],
                          [ 0.833330938737,  0.333338122527,  0.676754885757],
                          [ 0.833330938737,  0.833338122527,  0.676754885757],
                          [ 0.333330938737,  0.833338122527,  0.676754885757],
                          [-0.000002394597,  0.333338122527,  0.733058104206],
                          [-0.000002394597,  0.333338122527,  0.901967759553],
                          [-0.000002394597,  0.333338122527,  1.0708774149  ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new MTP potential set
mtp_potential_set = TremoloXPotentialSet('MTP')
mtp_potential_set.addParticleType(ParticleType('Si'))
mtp_potential_set.addParticleType(ParticleType('O'))
mtp_potential_set.addParticleType(ParticleType('C'))

mtp_potential_set.addPotential(MTPPotential(
    file = 'mtp-file.mtp',
    suppress_intercept = False,
))

# Create a new D3 potential set
d3_potential_set = TremoloXPotentialSet('D3')
d3_potential_set.addParticleType(ParticleType('Si'))
d3_potential_set.addParticleType(ParticleType('O'))
d3_potential_set.addParticleType(ParticleType('C'))
# D3BJ parameters for PBE
d3_potential_set.addPotential(D3Potential(
    S6=+1.0000000000e+00,
```

(continues on next page)

(continued from previous page)

```
    S8=+7.8750000000e-01,
    a1=+4.2890000000e-01,
    a2=+4.4407000000e+00*Bohr,
))

calculator = TremoloXCalculator(
    parameters=[mtp_potential_set, d3_potential_set]
)

bulk_configuration.setCalculator(calculator)
```

Define a D3 potential for a SiC/SiO_2 interface to be used in an LCAO-DFT *CombinedCalculator* by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [9.228, 0.0, 0.0]*Angstrom
vector_b = [4.613999999999999, -2.663894142040934, 0.0]*Angstrom
vector_c = [0.0, 0.0, 9.185292029012857]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Carbon, Carbon, Carbon, Silicon, Silicon,
            Silicon, Carbon, Carbon, Carbon, Oxygen, Oxygen, Oxygen, Silicon,
            Oxygen, Silicon]

# Define coordinates
fractional_coordinates = [[-0.            ,  0.666666666667,  0.            ],
                          [ 0.333333333333,  0.666666666667,  0.            ],
                          [ 0.666666666667,  0.666666666667,  0.            ],
                          [-0.            ,  0.666666666667,  0.211586087177],
                          [ 0.333333333333,  0.666666666667,  0.211586087177],
                          [ 0.666666666667,  0.666666666667,  0.211586087177],
                          [ 0.            ,  0.333333333333,  0.274787126204],
                          [ 0.333333333333,  0.333333333333,  0.274787126204],
                          [ 0.666666666667,  0.333333333333,  0.274787126204],
                          [ 0.            ,  0.333333333333,  0.486373213382],
                          [ 0.333333333333,  0.333333333333,  0.486373213382],
                          [ 0.666666666667,  0.333333333333,  0.486373213382],
                          [ 0.833330938737,  0.333338122527,  0.676754885757],
                          [ 0.833330938737,  0.833338122527,  0.676754885757],
                          [ 0.333330938737,  0.833338122527,  0.676754885757],
                          [-0.000002394597,  0.333338122527,  0.733058104206],
                          [-0.000002394597,  0.333338122527,  0.901967759553],
                          [-0.000002394597,  0.333338122527,  1.0708774149  ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
```

(continues on next page)

(continued from previous page)

```
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# -----------------------------------------------------------------
# Calculator
# -----------------------------------------------------------------

# Create a new D3 potential set
d3_potential_set = TremoloXPotentialSet('D3')
d3_potential_set.addParticleType(ParticleType('Si'))
d3_potential_set.addParticleType(ParticleType('O'))
d3_potential_set.addParticleType(ParticleType('C'))
# D3BJ parameters for PBE
d3_potential_set.addPotential(D3Potential(
    S6=+1.0000000000e+00,
    S8=+7.8750000000e-01,
    a1=+4.2890000000e-01,
    a2=+4.4407000000e+00*Bohr,
))

calculators = [
    TremoloXCalculator(parameters=d3_potential_set),
    LCAOCalculator()
]
calculator = CombinedCalculator(calculators)

bulk_configuration.setCalculator(calculator)
```

Notes

For most use cases it is more convenient to replace the the definition of the D3 potential through the *TremoloXPotentialSet* with a *DispersionD3Z* (zero damping)[1] or *DispersionD3BJ* (Becke-Johnson damping)[2] object which automatically supply the correct parameters for the chosen exchange-correlation functional.

D4Potential

class D4Potential(*file=None*, *fileD3=None*, *S6=None*, *a1=None*, *S8=None*, *a2=None*, *r_cut=None*, *bond_rcut=None*, *triple_rcut=None*, *taper_factor=None*, *ATM=None*, *EEQ_rcut=None*, *DSF_alpha=None*, *EEQ_only=None*)

Constructor of the potential.

Parameters

- **file** (*str*) – The name of the file that contains the D4 parameters.
- **fileD3** (*str*) – The name of the file that contains the involved D3 parameters.

[1] S.ăGrimme, J.ăAntony, S.ăEhrlich, and H.ăKrieg. A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu. *J. Chem. Phys.*, 2010. doi:10.1063/1.3382344.

[2] Stefan Grimme, Stephan Ehrlich, and Lars Goerigk. Effect of the damping function in dispersion corrected density functional theory. *Journal of Computational Chemistry*, 32(7):1456–1465, 2011.

- **S6** (*float*) – The dimensionless BJ/Z parameter S6
- **a1** (*float*) – The dimensionless BJ parameter a1 By setting S6r = 0 and a1!=0 or a2!=0 one chooses BJ damping
- **S8** (*float*) – The dimensionless BJ/Z parameter S8
- **a2** (*PhysicalQuantity of type length*) – The BJ parameter a2 By setting S6r = 0 and a1!=0 or a2!=0 one chooses BJ damping
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius for the interactions
- **bond_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the coordination numbers
- **triple_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the ATM interaction
- **taper_factor** (*float*) – The dimensionless taper_factor for bond_rcut and r_cut
- **ATM** (*bool*) – The parameter to switch on the Axilrod-Teller-Muto three-body terms
- **EEQ_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the EEQ interactions
- **DSF_alpha** (*PhysicalQuantity of type length**-1*) – Damping parameter in the DSF method. It determines how fast the Coulomb interactions decay with the particle distance. For alpha >0 the DSF approach is used to approximate erf(g r)/r If alpha <= 0, its just erf(g r)/r
- **EEQ_only** (*bool*) – The parameter to switch if just the EEQ term is applied (and not the C6 Dispersion term)

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a D4 potential ([1],[2],[3]) for a Si/SiO_2 interface to be used with an *MTPPotential* by adding particle types and interaction functions to the *TremoloXPotentialSet*.

[1] Eike Caldeweyher, Christoph Bannwarth, and Stefan Grimme. Extension of the D3 dispersion coefficient model. *The Journal of Chemical Physics*, 147(3):034112, 2017. URL: https://doi.org/10.1063/1.4993215, doi:10.1063/1.4993215.

[2] Eike Caldeweyher, Sebastian Ehlert, Andreas Hansen, Hagen Neugebauer, Sebastian Spicher, Christoph Bannwarth, and Stefan Grimme. A generally applicable atomic-charge dependent London dispersion correction. *The Journal of Chemical Physics*, 150(15):154122, 2019. URL: https://doi.org/10.1063/1.5090222, doi:10.1063/1.5090222.

[3] Eike Caldeweyher, Jan-Michael Mewes, Sebastian Ehlert, and Stefan Grimme. Extension and evaluation of the D4 London-dispersion model for periodic systems. *Phys. Chem. Chem. Phys.*, 22:8499–8512, 2020. URL: http://dx.doi.org/10.1039/D0CP00502A, doi:10.1039/D0CP00502A.

```
# -*- coding: utf-8 -*-
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [4.440892098500626e-16, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 21.495928431319584]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Oxygen, Oxygen,
            Oxygen, Silicon, Oxygen, Oxygen, Oxygen, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.17467925    ,  0.135152125   ,  0.235604166294],
                          [ 0.67467925    ,  0.635152125   ,  0.235604166294],
                          [ 0.92467925    ,  0.385152125   ,  0.298762638576],
                          [ 0.42467925    ,  0.885152125   ,  0.298762638576],
                          [ 0.67467925    ,  0.135152125   ,  0.361921110858],
                          [ 0.17467925    ,  0.635152125   ,  0.361921110858],
                          [ 0.42467925    ,  0.385152125   ,  0.42507958314 ],
                          [ 0.92467925    ,  0.885152125   ,  0.42507958314 ],
                          [ 0.17467925    ,  0.135152125   ,  0.488238055421],
                          [ 0.67467925    ,  0.635152125   ,  0.488238055421],
                          [ 0.05955       ,  0.614847875   ,  0.566340611359],
                          [ 0.392883333333,  0.114847875   ,  0.578342757834],
                          [ 0.273783      ,  0.819847875   ,  0.589473461329],
                          [ 0.845317      ,  0.599947875   ,  0.618508356925],
                          [ 0.511983      ,  0.220197875   ,  0.642334400174],
                          [ 0.726216666667,  0.410297875   ,  0.67136929577 ],
                          [ 0.94045       ,  0.220197875   ,  0.700404191366],
                          [ 0.607117      ,  0.599947875   ,  0.724230234614],
                          [ 0.17865       ,  0.819847875   ,  0.75326513021 ],
                          [ 0.05955       ,  0.114847875   ,  0.764395833706]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new MTP potential set
mtp_potential_set = TremoloXPotentialSet('MTP')
mtp_potential_set.addParticleType(ParticleType('Si'))
mtp_potential_set.addParticleType(ParticleType('O'))

mtp_potential_set.addPotential(MTPPotential(
    file = 'MTP_Si_SiO2_interface_small.mtp',
    suppress_intercept = False,
```

(continues on next page)

(continued from previous page)

```
))

# Create a new D4 potential set
d4_potential_set = TremoloXPotentialSet('D4')
d4_potential_set.addParticleType(ParticleType('Si'))
d4_potential_set.addParticleType(ParticleType('O'))
# D4BJ parameters for PBE.
d4_potential_set.addPotential(D4Potential(
    file=None,
    S6=+1.00000000,
    S8=+0.95948085,
    a1=+0.38574991,
    a2=+4.80688534*Bohr,
    EEQ_only=True
))

calculator = TremoloXCalculator(
    parameters=[mtp_potential_set, d4_potential_set]
)

bulk_configuration.setCalculator(calculator)
```

Notes

Currently only the electronegativity equilibrium model (EEQ) part of the D4 potential is available as a standalone model, which means it can only be used with `EEQ_only=True`.

DDHCustomExchangeCorrelation

class DDHCustomExchangeCorrelation(*screening_length=PhysicalQuantity(0.11, 1 / Bohr)*, *exx_fraction=0.25*, *number_of_spins=1*, *spin_orbit=None*, *ddh_parameters=None*, *local_ddh=None*, *ddh_variance=None*, *vacuum_exx_fraction=None*, *metallic_correction=None*)

Initialize the base object with the correct number of spins, and with the only Exchange and Correlation currently allowed for this class, i.e. the HSE ones.

Parameters

- **screening_length** (*PhysicalQuantity* of type inverse length) – The screening length defined for the coulomb kernel. **Default:** The HSE06 value of `0.11 * Bohr**-1`.
- **exx_fraction** (*float*) – The fraction of exact exchange to be added in with the PBE exchange **Default:** The HSE06 value of 0.25.
- **number_of_spins** (1 | 2) – The number of spins to be used in the calculation. **Default:** 1.
- **spin_orbit** (*bool*) – If `True` spin-orbit coupling is considered. **Default:** `False`.
- **ddh_parameters** (*list*) – The dielectric dependent hybrid functional parameters.
- **local_ddh** (*bool*) – If `True` the local algorithm is used. **Default:** `False`.
- **ddh_variance** (*float*) – The gaussian variance of the local dielectric estimator. **Default:** 2.12.

- **vacuum_exx_fraction** (*float*) – The fraction of exact exchange to be used in vacuum **Default:** The HSE06 value of 0.25.

cParameter(*grid_descriptor=None*)

return the param C.

Parameters
grid_descriptor (*NLEngine.GridDescriptor*) – The grid descriptor.

cloneAsPolarized()

Create a new functional with all the same settings, but polarized spin.

cloneAsSpinOrbit()

Create a new functional with all the same settings, but with Spin-Orbit coupling included.

correlation()

Returns
The correlation functional class

Return type
Correlation

defaultDielectricDependentHybridFractionVariance()

Returns
The default gaussian variance of the local dielectric estimator.

Return type
float

defaultExactExchangeFraction()

Returns
The default exact exchange fraction that is mixed in with PBE exchange.

Return type
float

defaultScreeningLength()

Returns
The default screening length defined for the coulomb kernel.

Return type
PhysicalQuantity of type inverse length

densityCheckThreshold()

Returns
The density safe guard.

Return type
float

dftHalfEnabled()

Returns
Whether DFT-1/2 is enabled.

Type
bool

dielectricDependentHybridFraction(*dielectric_estimator*)

Determines the dielectric dependent hybrid functional fraction.

Parameters
dielectric_estimator (*float*) – The dielectric estimator.

Returns
The exact exchange fraction.

Return type
float

dielectricDependentHybridFractionVariance()

Returns
The gaussian variance of the local dielectric estimator.

Return type
float

dielectricDependentHybridFunctional()

Overwrite base class method.

Returns
True, then a dielectric dependent hybrid functional is used.

Return type
dict

dielectricDependentHybridParameters()

Returns
The dielectric dependent hybrid hybrid parameters.

Return type
list

dielectricDependentHybridVacuumEstimator()

Returns
The estimator value corresponding to the vacuum exx fraction.

Return type
float

dielectricDependentHybridVacuumExactExchangeFraction()

Returns
The fraction of exact exchange to be used in vacuum.

Return type
float

exactExchangeFraction()

Returns
The part of exact exchange that is mixed in with PBE exchange.

Return type
float

exchange()

Returns
The exchange functional class

Return type
Exchange

hubbardTerm()

Returns
The Hubbard term used.

Return type
Onsite | OnsiteShell | Dual | DualShell | None

localDielectricDependentHybrid()

Returns
True, then local dielectric dependent hybrid functional algorihm.

Return type
bool

maximumPotential()

Set maximum potential value allowed for the TB09. The value to use as the maximum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

static metallicCorrection()

Returns
Always false, metallic corrections use a subclass.

Return type
bool

minimumPotential()

Set minimum potential value allowed for the TB09. The value to use as the minimum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a string representation of the *ExchangeCorrelation* instance.

Parameters
stream (A stream that supports strings being written to using *write*.) – The stream the exchange-correlation should be written to. **Default:** `sys.stdout`

numberOfSpins()

Returns
The number of spins.

Return type
`1 | 2 | 4`

params()

Get the parameters

screeningLength()

Returns
The screening length defined for the coulomb kernel.

Return type
PhysicalQuantity of type inverse length

spinOrbit()

Returns
Boolean determining if spin-orbit is enabled (True) or not (False).

Return type
bool.

spinType()

Returns
The spin type.

Return type
`NLEngine.POLARIZED | NLEngine.NONCOLLINEAR | NLEngine.UNPOLARIZED`

uniqueString()

Return a unique string representing the state of the object.

unscreenedExchange()

Whether the exchange functional in use is unscreened.

useLocalCParameter()

Returns
True, then a local c-parameter is used.

Return type
bool

Usage Examples

Defining the original dielectric dependent hybrid functional defined in[1]:

```
exchange_correlation = DDHCustomExchangeCorrelation(
    screening_length=0.11 * Bohr**-1,
    exx_fraction=0.25,
    number_of_spins=1,
    ddh_parameters=(0.121983, 0.0, 0.0, 0.0, 0.130711)
)
```

where the exact exchange mixing parameter is obtained as

$$\alpha = 0.121983 + 0.130711\bar{g}^4,$$

Performing a calculation using the local spatial dependent dielectric dependent hybrid functional defined in[2] specify:

```
exchange_correlation = DDHCustomExchangeCorrelation(
    screening_length=0.11 * Bohr**-1,
    exx_fraction=0.25,
    number_of_spins=1,
    ddh_parameters=(0.121983, 0.0, 0.0, 0.0, 0.130711)
    local_ddh=True,
    ddh_variance=2.12
)
```

DFTBDirectory

class DFTBDirectory(*path*)

Class for representing a DFTB parameter file directory.

Parameters
path (*str*) – The path of the directory to read files from.

fillingMethod(*element*)

Parameters
element (*PeriodicTableElement*) – The element for which to query the filling method.

Returns
The method used for setting up the initial occupation for the given element.

Return type
SphericalSymmetric | Anisotropic

onsiteSpinOrbitSplit(*element*)

Query the onsite spin-orbit split energies for a given element.

Parameters
element (*PeriodicTableElement*) – The element for which to query the spin-orbit split energies.

Returns
The onsite spin-orbit split for this element.

[1] Miguel A.ăL. Marques, Julien Vidal, Micael J.ăT. Oliveira, Lucia Reining, and Silvana Botti. Density-based mixing parameter for hybrid functionals. *Phys. Rev. B*, 83:035119, Jan 2011.

[2] Pedro Borlido, Miguel A.ăL. Marques, and Silvana Botti. Local Hybrid Density Functional for Interfaces. *Journal of Chemical Theory and Computation*, 14(2):939–947, 2018.

Return type
PhysicalQuantity of type energy

onsiteSpinSplit(*element*)

Query the onsite spin-split energies for a given element.

Parameters
element (*PeriodicTableElement*) – The element for which to query the spin splits.

Returns
The onsite spin-split energies for this element as a square matrix of size *n* where *n* is the number of subshells of the given element.

Return type
PhysicalQuantity of type energy

pairPotentials(*configuration*)

Query function for getting the pair potentials.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration defining which elements to load.

Returns
The pair-potentials.

Return type
list of *PairPotential*

setFillingMethod(*element*, *filling_method*)

Set the method used for setting up the initial occupation for the given element.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the filling method.
- **filling_method** (SphericalSymmetric | Anisotropic) – The method used for setting up the initial occupation.

setOnsiteSpinOrbitSplit(*element*, *split*)

Set the onsite spin-orbit split energies for a given element in the configuration.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the spin-orbit splits.
- **split** (*PhysicalQuantity* of type energy) – The onsite spin-split energies for this element as a square matrix of size *n* where *n* is the number of subshells of the given element.

setOnsiteSpinSplit(*element*, *split*)

Set the onsite spin split energies for a given element in the configuration.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the spin splits.
- **split** (*PhysicalQuantity* of type energy) – The on-site spin-split energies for each subshell pair of the element.

setVacuumLevel(*element*, *vacuum_level*)

Set the energy shift of the vacuum level for a given element.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the energy shift of the vacuum level.
- **vacuum_level** (*PhysicalQuantity* of type energy) – The energy shift of the vacuum level.

table(*configuration*)

Query function for getting the *SlaterKosterTable* representation of the parameter files in the directory.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration defining which elements to load.

Returns
A *SlaterKosterTable* instance representing the parameter files in the directory.

Return type
SlaterKosterTable

vacuumLevel(*element*)

Parameters
element (*PeriodicTableElement*) – The element for which to query the energy shift of the vacuum level.

Returns
The energy shift of the vacuum level for the given element.

Return type
PhysicalQuantity of type energy

Usage Examples

Set up a self-consistent tight-binding calculation using parameters in the format developed by the DFTB consortium.

```
# Set up a graphene crystal
bulk_configuration = BulkConfiguration(
    bravais_lattice=Hexagonal(2.4612*Angstrom, 6.709*Angstrom),
    elements=[Carbon, Carbon],
    cartesian_coordinates=[[ 0.        ,  0.        ,  0.        ],
                           [ 1.2306    ,  0.71050145,  0.        ]]*Angstrom
    )

# setup slater-koster calculator with dftb parameters

basis_set = DFTBDirectory("dftb")

pair_potentials = DFTBDirectory("/dftb")

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(4, 4, 1) )

calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    pair_potentials=pair_potentials,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
bulk_configuration.setCalculator(calculator)

initial_velocity = MaxwellBoltzmannDistribution(
        temperature=1000.0*Kelvin
    )
```

(continues on next page)

(continued from previous page)

```
method = NVEVelocityVerlet(
        time_step=1.0*femtoSecond,
        initial_velocity=initial_velocity
    )

molecular_dynamics = MolecularDynamics(
        bulk_configuration,
        constraints=[],
        trajectory_filename='trajectory.nc',
        steps=50,
        log_interval=1,
        method=method
    )
```

Notes

- QuantumATK can auto detect parameters stored in the directory `share/tightbinding/dftb` in the QuantumATK installation directory.
- Only some parameter sets include a spin-polarization splitting W. If this information is missing you can use the built-in database (see *Spin polarization*).
- The highest angular momentum per each element is officially not part of the DFTB Slater-Koster file format. However, some codes use the convention to parse this information from the third integer (s=1, p=2, d=3) in the first row of the onsite file. We follow the same convention. When the value is not present in the file, as a fall back we determine it by inspecting non-zero interactions.

DFTHalfParameters

class DFTHalfParameters(*element*, *half_shell_n=None*, *half_shell_l=None*, *fractional_charge=None*, *cutoff_radius=None*, *inner_radius=None*, *all_electron=None*)

Class for representing the DFT-1/2 parameters set for a given element in DFT.

Parameters

- **element** (*PeriodicTableElement*) – The element for which the DFT-1/2 parameters are defined.
- **half_shell_n** (*int*) – Integer specifying from which n-shell to remove the fractional charge. **Default:** The last occupied shell.
- **half_shell_l** (*int*) – Integer specifying from which l-shell to remove the fractional charge. **Default:** The last occupied shell.
- **fractional_charge** (*float | list of floats*) – Either a float value between 0 and 1 specifying how much charge to remove from the atomic calculation, or a list of two floats specifying how much charge to remove from up and down spin. **Default:** `[0.5, 0]`
- **cutoff_radius** (*PhysicalQuantity* of type length) – Radius specifying the radial cutoff. **Default:** `2.0 * Bohr`
- **inner_radius** – Radius specifying the inner radial cutoff, used in sh-LDA-1/2. A inner_radius larger than 0.0 specifies that a sh-LDA-1/2 metod should be used. **Default:** `0.0 * Bohr`

- **all_electron** (*bool*) – Boolean to determine whether to use an all-electron calculation (`True`) or a pseudopotential calculation (`False`). **Default:** `True`

allElectron()

Return the all_electron value.

Returns
Boolean to determine whether to use an all-electron calculation (`True`) or a pseudopotential calculation (`False`).

Return type
bool

cutoffRadius()

Return the cutoff radius.

Returns
The cutoff radius.

Return type
PhysicalQuantity of type length

element()

Return the element for which the DFT-1/2 parameters are defined.

Returns
The atomic element.

Return type
PeriodicTableElement

fractionalCharge()

Return the fractional_charge value.

Returns
The fractional charge to remove. It will either be a float or a list of two floats where the latter case specifies the up and down spin fractional charges to remove.

Return type
float | list of floats

halfShellL()

Return the half_shell_l value.

Returns
The integer for the angular quantum number of the state to remove a fractional electron from.

Return type
int

halfShellN()

Return the half_shell_n value.

Returns
The integer for the principal quantum number of the state to remove a fractional electron from.

Return type
int

innerRadius()

Return the inner cutoff radius.

Returns
The inner cutoff radius used in shell-DFT-1/2.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a *BasisSet* for Silicon with a set of custom *DFTHalfParameters*:

```
dft_half_parameters = DFTHalfParameters(
    element=Silicon,
    fractional_charge=[0.2, 0.0],
    cutoff_radius=4.0*Bohr)

basis_set = [
    LDABasis.Silicon_DoubleZetaPolarized(
        dft_half_parameters=dft_half_parameters)
]
```

Important: Adding a set of *DFTHalfParameters* to a *BasisSet* will only have an effect on the calculation if the *ExchangeCorrelation* has been defined with `dft_half_enabled = True`.

Notes

•More information on the DFT-1/2 method is given in *DFT-1/2 method*.

DampedDispersionPotential

class DampedDispersionPotential(*particleType1*, *particleType2*, *b*, *c*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**b** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**c** (*PhysicalQuantity of type length**6 * energy*) – Potential parameter.

- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *DampedDispersionPotential* defines a potential of the form

$$V_{ij}(r) = -\frac{C}{r^6}\frac{1}{\left(1 - e^{-br}\sum_{k=0}^{6}\frac{(br)^k}{k!}\right)}.$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

DefectCluster

class DefectCluster(*defect_list*)

Defines a defect cluster. Used as part of a *ChargedPointDefect* study.

Parameters

defect_list (sequence of [*Vacancy* | *Substitutional* | *Interstitial*]) – The list of basic defects which make up the cluster.

defectList()

Returns
The list of constituent defects.

Return type
list of [*Vacancy* | *Substitutional* | *Interstitial*]

generateDefectConfiguration(*reference_bulk_configuration*, *supercell_repetitions*, *use_ghost=None*)

Generate the supercell with the embedded defect.

Parameters

- **reference_bulk_configuration** (*BulkConfiguration*) – The bulk unit cell configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
The supercell configuration with the embedded defect.

Return type
BulkConfiguration

uniqueString()

Return a unique string representing the state of the object.

Notes

The *DefectCluster* object defines a defect cluster that consists of a number of point defects. These defects are given as input when the object is created. A *DefectCluster* can contain any number of vacancy, substitution or interstitial defects. The *DefectCluster* does not perform any symmetry analysis, and so is a definition of only a single possible configuration of the defect cluster.

A *DefectCluster* can be added to a *NamedPointDefect* to define a defect for use in a *ChargedPointDefectConfiguration* calculation.

See *Defining the point defect* for a description of how to use this object as part of a *ChargedPointDefect* study.

DefectDiffusionRates

class DefectDiffusionRates(*defect_diffusion_table*)

defect_diffusion_table, Usually

Columns:

initial defect (NamedPointDefect) final defect (NamedPointDefect) best_final defect (NamedPointDefect) [not used] initial cpdc (ChargedPointDefectConfiguration) [not used] final cpdc (ChargedPointDefectConfiguration) [not used] neb (single path from initial defect to final) diffusivity (DefectDiffusivity) (optional).

Parameters
defect_diffusion_table (Table) – The defect diffusion table.

static calculateDiffusivity(*defect_diffusivity*, *parameters_store=None*)

Calculate the macroscopic diffusivity of both defects forming the endpoints of the supplied NEB.

Parameters

•**defect_diffusivity** (DefectDiffusivity) – The defect diffusivity.

•**parameters_store** (*Store*) – The parameters store.

Returns
The calculated diffusivities of the initial and final defect.

Return type
tuple of (initial diffusivity, final diffusivity)

calculateRateAndTransitionMatrix(*temperature=None*, *defect_type=None*, *enable_finite_size_corrections=False*, *include_vibrations=False*, *assumed_transition_prefactor=PhysicalQuantity(10000000000000.0, 1 / s)*)

Calculate the rate matrix and discrete-time transition matrix.

Parameters

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the diffusivity should be calculated. **Default:** 300 * Kelvin.

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect.

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** False.

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. That includes the migration entropy, harmonic prefactor and vibrational contribution to the internal energy. **Default:** False.

•**assumed_transition_prefactor** (*PhysicalQuantity* of type frequency) – The assumed prefactor for this transition. This parameter is only used if include_vibrations is False. **Default:** 10**13 * Second**(-1)

Returns
The rate matrix and transition matrix.

Return type
tuple of (numpy array, *PhysicalQuantity* of type frequency)

static connectionName(*initial_defect_name*, *final_defect_name*)

The name of the connection

Parameters

•**initial_defect_name** (*str*) – Name of initial defect.

•**final_defect_name** (*str*) – Name of final defect.

Returns
name of connection.

Return type
str

createMapping()

Create mapping from defect name to specific image of NEB with the lowest forward barrier.

Returns
The mapping.

Return type
dict

graph()

Create the graph.

Returns
The network graph.

Return type
networkx.Graph

hasDiffusivity()

Returns
True, then the DefectDiffusionRates object has a table with DefectDiffusivity.

Return type
bool

hasVibrations()

Returns
True, then the all DefectDiffusivity objects have vibrational corrections.

Return type
bool

nodeOrEdgeObject(*name*)

The image of the defect with the name.

Parameters
name (*str*) – Name of defect or the path between defects.

Returns
The requested configuration or the NudgedElasticBand of the connection.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | NudgedElasticBand

tableInfo(*parameters_store=None*)

Generate the table info.

Parameters
parameters_store (*Store*) – The parameters store.

Returns
The table info.

Return type
list of dict

uniqueString()

Return a unique string representing the state of the object.

Notes

The *DefectDiffusionRates* object calculates the final diffusion network and diffusivities resulting from a series of defect diffusion calculations. The object is created by supplying a table of defects, defect configurations, NEB and optionally the *DefectDiffusivity* object for each transition. These resulting object can then be visualized in the *Defect Diffusion Rates Analyzer*.

DefectDiffusivity

class DefectDiffusivity(*neb*, *initial_defect*, *final_defect*, *charge_state=None*, *defect_vibrations=None*, *minimum_displacement=None*, *finite_difference=None*, *finite_difference_method=None*)

A class that stores thermodynamic data needed to calculate a defects diffusivity. It requires input from an updated *NudgedElasticBand* configuration and *ChargedPointDefectConfiguration* of the NEBs endpoints.

Parameters

- **neb** (*NudgedElasticBand*) – The updated nudged elastic band configuration.
- **initial_defect** (*ChargedPointDefectConfiguration*) – The initial defect.
- **final_defect** (*ChargedPointDefectConfiguration*) – The final defect.
- **charge_state** (*int*) – The charge state to calculate the diffusivity for.
- **defect_vibrations** (*ReducedDynamicalMatrix | UseDefault*) – Select method for calculating transition state vibrations. The flag `ReducedDynamicalMatrix` calculates the vibrational correction only for atoms that contribute significantly to the reaction coordinate. This approximation assumes that the contributions to the partition function from static atoms at the reactant and transition state cancel out. Choosing this method is recommended when the calculation of the complete dynamical matrix is unfeasible with a given phonon calculator. NOTE: This value will trigger a re-calculation of the vibrational correction of the supplied defects using the reduced dynamical matrix. This is necessary in order to treat all configurations, including the transition state, on equal footing. The flag `UseDefault` uses the same method as the optimized charge point defects to calculate the transition state vibrations. If no vibrations are included in these defects, vibrations are not calculated in the transition state. **Default:** UseDefault.
- **minimum_displacement** (*PhysicalQuantity* of type length | None) – The minimum distance that an atom must move during the reaction in order to be included in the calculation. If an atom moves further than this distance, it and its neighbors will be included in the dynamical matrix calculation that is needed for the frequencies and phonon density of states. Setting this value to zero will include all degrees of freedom. Setting this value to a larger number will reduce the amount of work needed to calculate the prefactor, but will decrease the accuracy. **Default:** `0.05 * Angstrom`
- **finite_difference** (*PhysicalQuantity* of type length) – The finite difference step size used when calculating the vibrational corrections. This parameter is only used when `minimum_displacement` is not None. **Default:** `0.01*Angstrom`
- **finite_difference_method** (`Central` | `Forward`) – The finite difference method to use when calculating the vibrational corrections. This parameter is only used when `minimum_displacement` is not None. **Default:** `Central`

calculateDiffusivity(*temperature=None*, *migration_distance=None*, *diffusion_dimensionality=None*, *coordination_number=None*, *correlation_factor=None*, *defect_type=None*, *enable_finite_size_corrections=False*, *include_vibrations=False*, *assumed_transition_prefactor=PhysicalQuantity(10000000000000.0, 1 / s)*)

Calculate the macroscopic diffusivity of both defects forming the endpoints of the supplied NEB.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the diffusivity should be calculated. **Default:** 300 * Kelvin.

•**migration_distance** (*PhysicalQuantity* of type distance) – The migration distance of the defect. By default, the initial and final defect configurations will be compared to find the atom that moves most, which will then be used to calculate the distance. **Default:** *Automatic*.

•**diffusion_dimensionality** (*int*) – The dimensionality of the diffusion. I.e 1, 2 or 3D. **Default:** 3.

•**coordination_number** (*int*) – The coordination number of the defect. One value that will be used for both the initial and final defect. With the automatic setting, the coordination number is determined based on the bonded neighbors of the atom moving the longest distance. **Default:** *Automatic*.

•**correlation_factor** (*float*) – The correlation factor of subsequent hops of the defect. The correlation factor can take values between 0 and 1. A value of 1 means no correlation. **Default:** 1.

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect.

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** False.

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. That includes the migration entropy, harmonic prefactor and vibrational contribution to the internal energy. **Default:** False.

•**assumed_transition_prefactor** (*PhysicalQuantity* of type frequency) – The assumed prefactor for this transition. This parameter is only used if include_vibrations is False. **Default:** 10**13 * Second**(-1)

Returns
The calculated diffusivities of the initial and final defect.

Return type
tuple of (initial diffusivity, final diffusivity)

calculateRateConstants(*temperature=None*, *defect_type=None*, *enable_finite_size_corrections=False*, *include_vibrations=False*, *assumed_transition_prefactor=PhysicalQuantity(10000000000000.0, 1 / s)*)

Calculate the rate constants between the defects forming the endpoints of the supplied NEB.

Parameters

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the diffusivity should be calculated. **Default:** 300 * Kelvin.

•**defect_type** (DeepLevelDefect | ShallowAcceptor | ShallowDonor) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with band_gap_calculator. Note that this parameter only has an effect if band_gap_calculator is different from formation_energy_calculator. **Default:** DeepLevelDefect.

•**enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** False.

•**include_vibrations** (*bool*) – Whether vibrational terms are included, if available. That includes the migration entropy, harmonic prefactor and vibrational contribution to the internal energy. **Default:** False.

•**assumed_transition_prefactor** (*PhysicalQuantity* of type frequency) – The assumed prefactor for this transition. This parameter is only used if include_vibrations is False. **Default:** 10**13 * Second**(-1)

Returns
The calculated rate constant of the initial and final defect.

Return type
tuple of (initial rate, final rate) as *PhysicalQuantity* of type frequency.

chargeState()

Returns
The charge state to calculate the diffusivity for.

Return type
int

defectVibrations()

Returns
Whether the harmonic transition state theory approximation was used.

Return type
bool

finalDefect()

Returns
The final defect.

Return type
ChargedPointDefectConfiguration

finalFrequencies()

Returns
The final vibrational frequencies.

Return type
PhysicalQuantity of type inverse time | None

finalPhononDOS()

Returns
The final defect phonon DOS.

Return type
PhononDensityOfStates | None

finiteDifference()

Returns
The finite difference step size used when calculating the vibrational corrections.

Return type
PhysicalQuantity of type length | None

finiteDifferenceMethod()

Returns
The finite difference method used when calculating the vibrational corrections.

Return type
`Central` | `Forward` | None

initialDefect()

Returns
The initial defect.

Return type
ChargedPointDefectConfiguration

initialFrequencies()

Returns
The initial vibrational frequencies.

Return type
PhysicalQuantity of type inverse time | None

initialPhononDOS()

Returns
The initial defect phonon DOS.

Return type
PhononDensityOfStates | None

migrationEnergies(*temperature=None*, *defect_type=None*, *enable_finite_size_corrections=False*, *include_vibrations=False*)

Calculate the migration energies of both defects from the charged point defect formation internal energies.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the diffusivity should be calculated. **Default:** 300 * Kelvin.
- **defect_type** (`DeepLevelDefect | ShallowAcceptor | ShallowDonor`) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with `band_gap_calculator`. Note that this parameter only has an effect if `band_gap_calculator` is different from `formation_energy_calculator`. **Default:** `DeepLevelDefect`.
- **enable_finite_size_corrections** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** `False`.
- **include_vibrations** (*bool*) – Whether vibrational terms are included. **Default:** `False`.

Returns
The calculated migration internal energies of the initial and final defect.

Return type
tuple of (initial migration internal energy, final migration internal energy)

migrationEnergiesFromNEB(*temperature=None*, *include_vibrations=False*)

Calculate the migration energies of both defects from the supplied NEB.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the diffusivity should be calculated. **Default:** 300 * Kelvin.
- **include_vibrations** (*bool*) – Whether vibrational terms are included. **Default:** `False`.

Returns
The calculated migration energies of the initial and final defect.

Return type
tuple of (initial migration internal energy, final migration internal energy)

minimumDisplacement()

Returns
The minimum displacement used to determine the vibrational contributions.

Return type
PhysicalQuantity of type length

neb()

Returns
The nudged elastic band configuration.

Return type
NudgedElasticBand

phononCalculator()

Returns
The calculator used for determining the phonon contributions.

Return type
Calculator

tsDefect()

Returns
The transition state defect.

Return type
ChargedPointDefectConfiguration | None

tsFrequencies()

Returns
The transition state vibrational frequencies.

Return type
PhysicalQuantity of type inverse time | None

tsIndex()

Returns
The transition state image index.

Return type
int

tsPhononDOS()

Returns
The transition state defect phonon DOS.

Return type
PhononDensityOfStates | None

uniqueString()

Return a unique string representing the state of the object.

vibrationsAvailable()

Returns
Whether the vibrational corrections are available.

Return type
bool

Notes

The *DefectDiffusivity* object calculates the diffusivity of a defect diffusion path. This object is created by supplying a *NudgedElasticBand* of the diffusion pathway, *ChargedPointDefectConfiguration* objects for the initial and final defects and the charge state of the defect transition. The barrier of the diffusion path is obtained from the transition state determined by the NEB. The defect diffusivity can then be calculated calling the `calculateDiffusivity` method with various parameters such as temperature.

DefectMigrationPaths

class DefectMigrationPaths(*initial_defect*, *final_defect=None*, *charge_state=None*, *minimum_migration_distance=None*, *maximum_migration_distance=None*, *maximum_total_migration_distance=None*)

Perform a defect migration between defect sites. The migration is a single hop between two stable defect sites, calculated with the nudged elastic band methodology. The initial and final defect configurations are defined in either a single ChargedPointDefectConfiguration object or two ChargedPointDefectConfiguration objects. If a single ChargedPointDefectConfiguration is provided, it is used as both the initial and final configuration. In both of these cases, DefectMigrationPaths will search over all symmetrically-equivalent sites for the final configuration to find the ones which are closest to the site of the initial configuration.

Parameters

- **initial_defect** (*ChargedPointDefectConfiguration*) – The named point defect providing the initial configuration for the migration. Must have a configuration stored on itself.
- **final_defect** (*ChargedPointDefectConfiguration*) – The optional named point defect providing the final configuration for the migration. Must have a configuration stored on itself. Note that initial and final defect studies must use compatible defects (ones in which the number of atoms added or removed for each species is the same). **Default:** *initial_defect* is used for the final configuration as well
- **charge_state** (*int*) – The integer charge state. **Default:** 0
- **minimum_migration_distance** (*PhysicalQuantity* of type length) – The minimum distance for accepting a defect migration path, based on the atom which is displaced the most during the reaction. **Default:** `0.1 * Angstrom`
- **maximum_migration_distance** (*PhysicalQuantity* of type length) – The maximum distance for accepting a defect migration path, based on the atom which is displaced the most during the reaction. **Default:** `3.5 * Angstrom`
- **maximum_total_migration_distance** (*PhysicalQuantity* of type length) – The maximum distance for accepting a defect migration path, based on the total displacement of all atoms during the reaction. **Default:** `30.0 * Angstrom`

chargeState()

Returns
The charge state.

Return type
int

finalDefect()

Returns
The final ChargedPointDefectConfiguration.

Return type
ChargedPointDefectConfiguration

generateEndpoints()

Method that finds the most likely transitions paths between two defects based on distance criteria.

Returns
Migration endpoint configurations.

Return type
list of lists (size 2) of *BulkConfiguration*

generateNEBs(*neb_calculator=None*, *image_distance=None*, *interpolation_algorithm=None*, *number_of_intermediate_images=None*, *use_endpoint_data=None*)

Method that sets up NEB calculations from a list of endpoint configurations using the image dependent pair potential interpolation algorithm.

Parameters

- **neb_calculator** (*Calculator*) – The calculator to be used for the NEB calculation. **Default:** The reference calculator of the initial *ChargedPointDefectConfiguration* with the correct charge state, if applicable.
- **image_distance** (*PhysicalQuantity* of type length) – Distance between the coordinates of successive configurations in the series of configurations. Mutually exclusive with number_of_intermediate_images. **Default:** 0.5*Angstrom.
- **interpolation_algorithm** (bool | *LinearInterpolation* | ImageDependentPairPotential) – The interpolation algorithm to use when generating images. No distance between successive images is greater than the image_distance parameter. The constraints are always automatically applied to be consistent with the initial defect. **Default:** *ImageDependentPairPotential*
- **number_of_intermediate_images** (*int*) – The number of images between the endpoints. Mutually exclusive with image_distance.
- **use_endpoint_data** (*bool | None*) – Whether to use endpoint energies, forces, stress to speed up the *NudgedElasticBand* calculation. By default, the endpoint data will be used if the supplied calculator is identical to the reference calculator on the *ChargedPointDefectConfiguration*. If the calculators differ, the endpoint data is invalid and can not be used. **Default:** Automatic

Returns
Unoptimized NEBs.

Return type
list of *NudgedElasticBand*

initialDefect()

Returns
The initial ChargedPointDefectConfiguration.

Return type
ChargedPointDefectConfiguration

maximumMigrationDistance()

Returns
The maximum distance for accepting a transition path reaction, based on the atom which is displaced the most during the reaction.

Return type
PhysicalQuantity of type length

maximumTotalMigrationDistance()

Returns
The maximum distance for accepting a transition path reaction, based on the total displacement of all atoms during the reaction.

Return type
PhysicalQuantity of type length

`minimumMigrationDistance()`

Returns
The minimum distance for accepting a transition path reaction, based on the atom which is displaced the most during the reaction.

Return type
PhysicalQuantity of type length

`uniqueString()`

Return a unique string representing the state of the object.

Notes

The *DefectMigrationPaths* object calculates symmetrically unique diffusion pathways between two defects. In doing so, it takes into account the symmetry of each defect, and the equivalent positions of the defect in the host material. The object is created by supply an initial defect as a *ChargedPointDefectConfiguration*. If no final defect is supplied self-diffusion is assumed, otherwise a second defect can be supplied with the argument `final_defect`. If paths for a non-neutral defect are required, the charge state can be given with the `charge_state` argument. Criteria for the minimum and maximum movement of the most displaced atom can be given, as well as the maximum total atomic movement. These criteria can be used to filter out unrealistic transition paths.

Once the object is created the endpoints for each path can be generated using the method `generate_endpoints`. Alternatively, an unoptimized *NudgedElasticBand* can be generated for each transition path using the method `generateNEBs`. Here, additional information can be given as to how the NEB is constructed, allowing for setting the calculator, the number of intermediate images and whether or not the calculated endpoint data, such as energies and forces, is used.

DefectPairGenerator

`class DefectPairGenerator(`*site_defect_generator*, *secondary_defect_generator*, *minimum_distance=None*, *maximum_distance=None*, *supercell_repetitions=None*, *symmetry_tolerance=None*`)`

A class which describes the possible defect clusters formed by pairs of point defects at various distances. The original two defects are placed together into the given host material supercell.

Parameters

- **`site_defect_generator`** (*SubstitutionalGenerator* | *VacancyGenerator*) – Generator for site defects that will constitute the first defect in the pair.
- **`secondary_defect_generator`** (*SubstitutionalGenerator* | *VacancyGenerator* | *InterstitialGenerator*) – Generator for vacancies or interstitials that will constitute the second defect in the pair and will be responsible for paired diffusion.
- **`minimum_distance`** (*PhysicalQuantity* of type length) – Minimum allowed distance from site defect to another. **Default:** `2.0 * Angstrom`.
- **`maximum_distance`** (*PhysicalQuantity* of type length) – Maximum allowed distance from site defect to another. **Default:** `3.0 * Angstrom`.

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the host configuration along the (a, b, c) directions. Note that using this parameter has the same effect as supercelling the host configuration prior to generating the point defects passed to this generator.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*, *site_element*, *secondary_element=None*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters

- **top_n_configurations** (*int*) – Number of most distinct configurations.
- **site_element** (*PeriodicTableElement*) – Site element to keep after filtering.
- **secondary_element** (*PeriodicTableElement*) – Secondary element to keep after filtering. **Default:** Same as site_element

Returns
A filtered defect list.

Return type
DefectPairGenerator

filterByLatticeSpecies(*site_element_list=None*, *secondary_element_list=None*)

Method for selecting unique defects by element. Valid only for defect types that are associated with a lattice site, e. g. Vacancies, but not Interstitials.

Parameters

- **site_element_list** (*PeriodicTableElement* | list of *PeriodicTableElement*) – List of elements to keep after filtering.
- **secondary_element_list** (*PeriodicTableElement* | list of *PeriodicTableElement*) – List of elements to keep after filtering. Only used if the secondary defect is associated with a lattice site.

Returns
A filtered defect list.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

- **point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.

•**keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`

•**select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)
Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)
Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

maximumDistance()

Returns
Maximum allowed distance from site defect to another.

Return type
PhysicalQuantity of type length.

minimumDistance()

Returns
Minimum allowed distance from site defect to another.

Return type
PhysicalQuantity of type length.

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

secondaryDefectGenerator()

Returns
Generator for vacancies or interstitials.

Return type
SubstitutionalGenerator | *VacancyGenerator* | *InterstitialGenerator*

siteDefectGenerator()

Returns
Generator for site defects.

Return type
SubstitutionalGenerator | *VacancyGenerator*

supercellRepetitions()

Returns
The supercell repetitions.

Return type
tuple (size 3) of int

symmetryTolerance()

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

uniqueDefects()

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

In this example a *Defects* object is created that contains vacancy and pair defects in 4H-SiC. The defect pairs consist of a vacancy and anti-site substitution on two neighboring atoms. Together, these defects can be used to model diffusion of a neighboring atom into a vacancy site. As the 4H polytype of silicon carbide has two different C and Si sites (*h* and *k*) there are a total of 4 pair defects that are created in the example.

In the example, the defects are created and added to a *Defects* object in a *Custom Block*. The defect configurations are then generated and are iterated over, calculating the phonon density of states of each defect.

A workflow for the calculation can be downloaded `here` with a corresponding python script `DefectPairGenerator_Example.py`.

Notes

The *DefectPairGenerator* object generates defect pairs for a specific host material. It is possible to create defects pairs that combine a site defect (vacancy or substitution) with either a vacancy, substitution or interstitial. Symmetry analysis is able to reduce the number of defects to the minimum of unique defects pairs.

The site defect and secondary defect are passed to the *DefectPairGenerator* as generators themselves. Using the `minimum_distance` and `maximum_distance` arguments the minimum and maximum distances between the defects can also be set. A supercell repetition for the defect can also be given. This is useful in cases where a pair may be desired that is not easily contained within the host unit cell. An example may be where the unit cell contains only one atom. Once the generator is created the defects contained within the defect can be filtered down to a desired subset of defects.

The unique defects can be returned with the `uniqueDefects` method as *NamedPointDefect* objects, which can be used as input for a *ChargedPointDefectConfiguration* object. The defects and generators can also be given to a *Defects* object to create the basic defect configurations.

Defects

class Defects(*defects*, *defect_generators=None*, *defect_generator_map=None*, *name='Defects'*)

A class which describes the list of defects.

Parameters

- **defects** (list of *NamedPointDefect*) – The list of defects.
- **defect_generators** (dict of (key, value) type (str, *BaseDefectGenerator*)) – Dictionary of defect defect_generators, used to construct the defects.
- **defect_generator_map** (dict of (key, value) type (*NamedPointDefect*, str)) – Mapping between the defects and the defect generators, used to construct them.
- **name** (*str*) – The name.

appendDefectPairsWithDefectConfigurations(*defect_pairs_table*, *required_defects_table*, *defect_configurations_table*)

Append the defect_configurations from the defect_configurations_table into the defect_pairs_table, as two new columns, one for initial and one for final.

Note that the required_defects_table is used as the defect_configurations_table is constructed from this table, and the required_defects_table knows which defects in defect_pairs_table correspond to which defect_configurations in defect_configurations_table.

Parameters

•**defect_pairs_table** (Table) – The defect migration pairs table.

•**required_defects_table** (Table) – The required defects table, that knows about the placement.

•**defect_configurations_table** (Table) – The defect configurations table.

Returns
The modified defect migration pairs table.

Return type
Table

generateDefectMigrationPairs(*host_configuration*, *supercell_repetitions=None*, *maximum_distance=None*)

Method to get the defect pairs table.

Parameters

•**host_configuration** (*BulkConfiguration*) – The host configuration.

•**supercell_repetitions** (*3-tuple of int | None*) – The supercell repetitions used in the defect material.

•**maximum_distance** (*PhysicalQuantity* of type length) – Maximum allowed distance.

Returns
The defect pairs table.

Return type
Table

requiredDefects(*defect_pairs_table*)

Create a table with a single column consisting of all unique NamedPointDefects in the table, either initial or final.

Parameters
defect_pairs_table (*Table*) – The defect pairs table (usually from generateDefectMigrationPairs).

Returns
The required defects table.

Return type
Table

totalDefects()

Create a table with just the input named point defects in it. This essentially just converts a Defects object to a table.

Returns
Table with all the named point defects.

Return type
Table

uniqueDefects()

Returns
The list of symmetrically unique point defects.

Return type
list of *NamedPointDefect*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

In this example the total energies of some defects in hexagonal boron nitride defects are calculated. Vacancy, anti-site substitutional and interstitial defects are defined in the *Defects* block. This is used to generate a table of defect configurations within a supercell. In each table iteration an *LCAOCalculator* is set, the geometry optimized, and then the total energy is calculated.

A workflow for the calculation can be downloaded `here` with a corresponding python script `Boron_nitride_defects.py`.

Notes

The *Defects* object manages the use of multiple defects for defect formation and diffusion calculations. The object takes at a minimum a sequence of defects in the form of *NamedPointDefect*. Optionally it also takes two dictionaries used to define the defect generators and which defects are created by which generator. The argument `defect_generators` takes a dictionary with the defect generators as the value. The argument `defect_generator_map` takes a second dictionary where the individual defects are the keys, and the keys from the `defect_generators` dictionary are the values.

The *Defects* object can create starting tables for different types of calculations. The method `totalDefects` returns a table of all the defects provided to the object. This is useful in cases where all the defects are to be calculated. For diffusion calculations the object can also generate a table of all of the possible transitions between defects using the `generateDefectMigrationPairs` method. This takes the compatibility of different defects and the minimum distance between them into account. If the generators are also supplied different symmetrically equivalent representations of the defects can be used to find the minimum distance between two defects. With this table, the minimal set of defects required to model the diffusion can be returned using the method `requiredDefects`. This method filters out defects that do not have any transition paths in the defect pair table.

DefectsParameters

class DefectsParameters(*defect_type=None*, *defect_element=None*, *include_voronoi_faces=None*, *include_voronoi_vertices=None*, *include_voronoi_ridges=None*, *miller_indices=None*, *split_distance=None*, *fractional_position=None*, *symmetry_tolerance=None*, *group_distance=None*, *filter_by_z_position=None*, *z_start=None*, *z_end=None*, *filter_by_distinct_configurations=None*, *top_n_configurations=None*, *filter_by_lattice_sites=None*, *filter_elements=None*)

A class which describes the list of defects.

Parameters

- **defect_type** (*int*) – The type of defect. **Default:** 0 (DefectType.SUBSTITUTIONAL)
- **defect_element** (PeriodicTableElement) – The element added as an interstitial or substitutional defect.
- **include_voronoi_faces** (*bool*) – Generate defects using Voronoi faces.
- **include_voronoi_vertices** (*bool*) – Generate defects using Voronoi vertices.
- **include_voronoi_ridges** (*bool*) – Generate defects using Voronoi ridges.
- **miller_indices** (*tuple*) – The Miller indices that define the direction of split interstitials.
- **split_distance** (*PhysicalQuantity* of type length) – The distance form the lattice position of split interstitial atoms.
- **fractional_position** (*sequence (size 3) of float*) – The position at which to insert the interstitial atom, given in fractional coordinates of the bulk unit cell.

- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.
- **group_distance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining groups of defects. **Default:** 0.1 * Angstrom.
- **filter_by_z_position** (*bool*) – Activate the filtering by Z position.
- **z_start** (*PhysicalQuantity* of type length) – The start of the position interval.
- **z_end** (*PhysicalQuantity* of type length) – The end of the position interval.
- **filter_by_distinct_configurations** (*bool*) – Activate the filtering using the MTP structural descriptor.
- **top_n_configurations** (*int*) – Number of most distinct configurations.
- **filter_by_lattice_sites** (*bool*) – Activate the filtering by lattice site elements.
- **filter_elements** (*iterable*) – The set of lattice elements that should be replaced.

defectGenerator(*host_configuration*)

The defect generator

Parameters
host_configuration (*BulkConfiguration*) – The host configuration.

Returns
Returns a generator of the defects stored in the collection.

Return type
BaseDefectGenerator based

uniqueDefects(*host_configuration*)

Return all symmetrically unique defects

Parameters
host_configuration (*BulkConfiguration*) – The host configuration.

Returns
Returns a list of the defects stored in the collection.

Return type
list of *NamedPointDefect*

uniqueString()

Return a unique string representing the state of the object.

classmethod voronoiTypes(*include_voronoi_vertices*, *include_voronoi_faces*, *include_voronoi_ridges*)

Returns
A list of the types of Voronoi points to use to generate interstitial defects.

Return type
list of VoronoiVertices, VoronoiFaces and VoronoiRidges

Notes

The *DefectsParameters* object enables the creation of an abstract defect definition. While defect generators such as *VacancyGenerator* or *InterstitialGenerator* require a specific pristine material, the *DefectsParameters* object only stores the parameters required to define a kind of point defect. This can be useful when applying the same kinds of defects across a number of configurations, such as when studying amorphous materials or materials with the same lattice.

The first required argument is the defect type. The type is represented by an integer as shown in the table below. Once the parameters are defined, the defect generator for a specific configuration can be obtained using the `defectGenerator` method. The unique defects for a given configuration can also be directly accessed using the `uniqueDefects` method.

Table 4.6: Generator types.

Argument	Defect Type
0	*SubstitutionalGenerator*
1	*VacancyGenerator*
2	*InterstitialGenerator*
3	*SplitInterstitialGenerator*
4	CustomInterstitialGenerator

DeformationPotential

class DeformationPotential(*configuration*, *hamiltonian_derivatives*, *dynamical_matrix*, *symmetry_label=None*, *kpoint_cartesian=None*, *kpoint_fractional=None*, *q_route=None*, *points_per_segment=None*, *q_path=None*, *electron_bands=None*, *phonon_modes=None*, *max_interaction_range=None*, *rotate_to_pure_spin_states=None*)

Constructor for the DeformationPotential object.

Parameters

- **configuration** (*BulkConfiguration*) – The BulkConfiguration for which to calculate the deformation potential.
- **hamiltonian_derivatives** (*HamiltonianDerivatives*) – The Hamiltonian derivatives for the configuration.
- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix for the configuration.
- **symmetry_label** (*str*) – The kpoint (as a symmetry point) for which to calculate the deformation potential. This option is mutually exclusive to `kpoint_cartesian` and `kpoint_fractional`. **Default:** G (Gamma-point)
- **kpoint_cartesian** (*PhysicalQuantity* of type inverse length) – The kpoint (in Cartesian coordinates) for which to calculate the deformation potential. This option is mutually exclusive to `symmetry_label` and `kpoint_fractional`. **Default:** `[0.0, 0.0, 0.0] * Angstrom**-1` Gamma-point
- **kpoint_fractional** (*list(3) of floats*) – The kpoint (in fractional coordinates) for which to calculate the deformation potential. This option is mutually exclusive to `kpoint_cartesian` and `symmetry_label`. **Default:** `[0.0, 0.0, 0.0]` Gamma-point
- **q_route** (list of symmetry points, e.g. `['G', 'X', 'G']`) – The route to take through the Brillouin-zone. This option is mutually exclusive to `q_path`. **Default:** None
- **points_per_segment** (*int*) – The number of points per segment of the route. This option is mutually exclusive to `q_path`. **Default:** 20
- **q_path** (*list of list of floats*) – List of fractional q-points. This option is mutually exclusive to `q_route` and `points_per_segment`. The shape of the list is `(:, 3)`, e.g. `[[0.0, 0.0, 0.0], [0.25, 0.0, 0.0], [0.5, 0.0, 0.0]]`. **Default:** None

•**electron_bands** (list of ints | `All`) – The band indices of the Bloch states to include. **Default:** `All` Include all bands.

•**phonon_modes** (list of ints | `All`) – Phonon modes to include. **Default:** `All` Include all phonon modes.

•**maximum_interaction_range** (*PhysicalQuantity* of type length) – Set the maximum range of the interactions in the *HamiltonianDerivatives*. **Default:** None

•**rotate_to_pure_spin_states** (*bool*) – Whether the eigenstates should be rotated to pure spin states or not. **Default:** `True` for Unpolarized, Polarized and Noncollinear, `False` for SpinOrbit

calculateDegenerateBandsDeformationPotential(*initial_bands_degenerate*, *final_bands_degenerate*, *phonon_modes_degenerate*, *qpoint_indices=None*, *spin=None*)

Method for fitting the deformation potential for a particular phonon mode.

Parameters

•**initial_bands_degenerate** (*list of ints*) – Quantum number (band index) of the degenerate initial Bloch states.

•**final_bands_degenerate** (*list of ints*) – Quantum number (band index) of the degenerate initial Bloch states.

•**phonon_modes_degenerate** (*list of ints*) – Phonon mode indices that are degenerate.

•**qpoint_indices** (*list of ints*) – Indices of the q-points where SVD is performed. **Default:** All q-points.

•**spin** (`Spin.Up` | `Spin.Down`) – The spin flag. **Default:** `Spin.Up`.

Returns
The deformation potential singular value decomposition.

Return type
PhysicalQuantity array of type energy / length

couplingMatrix()

Query method for the coupling matrix calculated in the Bloch basis.

Returns
Electron-phonon coupling matrix calculated with the *ElectronPhononCoupling* object. The shape of the coupling matrix is (number_of_spins, number_of_phonon_modes, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy

eigenvaluesK()

Query method for the electron Bloch state energies as a function of k.

Returns
The electron Bloch state energies as a function of k. The shape of the array is (number_of_spins, number_of_kpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy

eigenvaluesKMinusQ()

Query method for the electron Bloch state energies as a function of k-q.

Returns
The electron Bloch state energies as a function of k-q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy

`eigenvaluesKPlusQ()`

Query method for the electron Bloch state energies as a function of k+q.

Returns
The electron Bloch state energies as a function of k+q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy

`electronBands()`

Query method for the electron bands.

Returns
The list of electronic band indices used in the calculation.

Return type
list of int

`evaluate(`*coupling_matrix=None*`)`

Query method for the electron-phonon coupling matrix.

Returns
Electron-phonon coupling matrix (unscaled) calculated with the *ElectronPhononCoupling* object. The shape of the coupling matrix is (n_spins, n_modes, n_kpoints, n_qpoints, n_bloch_states, n_bloch_states).

Return type
PhysicalQuantity with the unit `eV / Angstrom`

`fitDeformationPotential(`*mode=None*, *initial_band=None*, *final_band=None*, *spin=None*, *first_q_index=None*, *last_q_index=None*, *coupling_matrix=None*, *include_phase_factor=None*`)`

Method for fitting the deformation potential for a particular phonon mode.

Parameters

- **mode** (*int*) – Phonon mode index. **Default:** The first mode calculated
- **initial_band** (*int*) – Quantum number (band index) of the initial Bloch state. **Default:** The first band calculated
- **final_band** (*int*) – Quantum number (band index) of the final Bloch state. **Default:** The last band calculated
- **spin** (`Spin.Up` | `Spin.Down`) – The spin flag. **Default:** `Spin.Up`
- **first_q_index** (*int*) – Index of the first q-point to consider in the fit. **Default:** 0
- **last_q_index** (*int*) – Index of the last q-point to consider in the fit. **Default:** Last index
- **coupling_matrix** (`'scaled'` | `'unscaled'`) – String indicating if the scaled or unscaled coupling matrix should be plotted. **Default:** `'unscaled'`
- **include_phase_factor** (*bool*) – Whether the phase factor is included or not. **Default:** `True`

Returns
Fitted zeroth, first-order deformation potentials, model deviation and R squared.

Return type
PhysicalQuantity, *PhysicalQuantity*, *PhysicalQuantity*, numpy.array

`inverseCharacteristicLength()`

Query method for inverse characteristic length.

Returns
The inverse characteristic length. The shape of the array is (number_of_degrees_of_freedom, number_of_qpoints)

Return type
PhysicalQuantity of type inverse length.

`kpoint()`

Query method for the k-point.

Returns
The k-point the deformation potential is calculated for.

Return type
numpy.array

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`

Returns
Structured information about the DeformationPotential.

Return type
dict

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`phaseFactor()`

Query method for phase factor.

Returns
The phase factor. list of length of number_of_spins. For each spin the shape of the coupling matrix is (number_of_qpoints, number_of_electron_bands, number_of_electron_bands).

Return type
unitless array.

`phononEnergies()`

Query method for the phonon energies.

Returns
The phonon energies as a function of q. The shape of the array is (number_of_degrees_of_freedom, number_of_qpoints), where number_of_degrees_of_freedom = 3*number_of_atoms.

Return type
PhysicalQuantity of type energy

phononModes()

Query method for the phonon modes.

Returns
The list of phonon mode indices used in the calculation.

Return type
list of int.

qRoute()

Query method for the q-route.

Returns
The route to taken through the Brillouin-zone in the calculation.

Return type
list of str

qpoints()

Query method for the q-points.

Returns
The list of fractional q-points used in the calculation.

Return type
list

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

unscaledCouplingMatrix()

Query method for the unscaled electron-phonon coupling matrix.

Returns
Electron-phonon coupling matrix calculated with the *ElectronPhononCoupling* object. The shape of the coupling matrix is (number_of_spins, number_of_phonon_modes, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity with the unit `eV / Angstrom`

Usage Examples

Calculate the deformation potential in graphene for different wave-vector paths in reciprocal space, specified by symmetry points connected by linear segments or a series of fractional q-points:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)
```

(continues on next page)

(continued from previous page)

```
elements = [Carbon, Carbon]

fractional_coordinates = [[0.0           , 0.0           , 0.5],
                          [0.333333333333, 0.666666666667, 0.5]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# Hamiltonian derivatives
# ------------------------------------------------------------
hamiltonian_derivatives = HamiltonianDerivatives(
    bulk_configuration,
    repetitions=(5, 5, 1),
    )

# ------------------------------------------------------------
# Dynamical matrix
# ------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    repetitions=(5, 5, 1),
    max_interaction_range=10*Angstrom,
    )

# ------------------------------------------------------------
# Deformation potential
# ------------------------------------------------------------
deformation_potential_1 = DeformationPotential(
    bulk_configuration,
    hamiltonian_derivatives,
    dynamical_matrix,
    symmetry_label='L',
    q_route=['G', 'M', 'L', 'A', 'G', 'K', 'H', 'A'],
    points_per_segment=30,
```

(continues on next page)

(continued from previous page)

```
    electron_bands=[0, 1, 2],
    phonon_modes=[0, 1],
    )

nlsave('graphene.nc', deformation_potential_1)

deformation_potential_2 = DeformationPotential(
    bulk_configuration,
    hamiltonian_derivatives,
    dynamical_matrix,
    kpoint_fractional=[0.0, 0.0, 0.5],
    q_path=[[0.0, 0.0, 0.0], [0.2, 0.0, 0.0], [0.3, 0.0, 0.0], \
            [0.4, 0.0, 0.0], [0.45, 0.0, 0.0], [0.5, 0.0, 0.0]],
    electron_bands=[0, 1, 2],
    phonon_modes=[0, 1],
    )

nlsave('graphene.nc', deformation_potential_2)
```

deformation_potential.py

DeltaTest

class DeltaTest(*elements*, *calculator*, *initial_spin=None*, *eos_data=None*)

Do a Delta test for a number of elements with a given calculator.

Parameters

- **elements** (*Sequence*) – Sequence of elements to be tested, e.g. *[Silicon, Hydrogen, Carbon]*.
- **calculator** (*Calculator*) – The calculator to be tested. Can be any QuantumATK calculator that supports total energy calculation.
- **initial_spin** (*InitialSpin*) – The inital spin to set on the configuration.

calculateEquationOfState()

Method to calculate the total energy as function of volume and storing the data on the object as a dictionary.

calculator()

Query method for the calculator.

delta(*element=None*, *reference=None*)

Get the delta value of the Delta test for a given element or the average among all elements.

Parameters

- **element** (*PeriodicTableElement*) – Element for which to return the delta value. If no element is specified the average value for all the elements will be returned.
- **reference** (str | *DeltaTest*) – The filename of the file that holds the reference data to test against, or an other *DeltaTest*, which must include the specific element. **Default:** WIEN2k data from https://molmod.ugent.be/deltacodesdft.

Returns
The delta value.

Return type
PhysicalQuantity of type energy.

deltaStatistics(*reference=None*)

Get statistics about the delta values of the Delta test.

Parameters
reference (str | *DeltaTest*) – The filename of the file that holds the reference data to test against, or an other *DeltaTest*, which must include the specific element. **Default:** WIEN2k data from https://molmod.ugent.be/deltacodesdft.

Returns
Dict with (element, delta), the minimum delta, element for the minimum, maximim delta, element for the maximum, average delta, standard deviation of delta, and failed elements.

Return type
(Dict with *PeriodicTableElement*, *PhysicalQuantity* of type energy, *PhysicalQuantity* of type energy, *PeriodicTableElement*, *PhysicalQuantity* of type energy, *PeriodicTableElement*, *PhysicalQuantity* of type energy, *PhysicalQuantity* of type energy, list of *PeriodicTableElement*).

elements()

Query method for the elements.

equationOfState(*element*)

Query method for the equation of state of a specific element.

equationOfStateParameters(*element*)

Query for the Birch-Murnaghan equation of state parameters for a specific element.

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *reference=None*)

Print a formatted report of the Delta test.

referenceEquationOfStateParameters(*element*, *reference=None*)

Query method for the reference eos parameters (Birch-Murnaghan parameters).

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Perform a DeltaTest for ATK-DFT vs. WIEN2k over Si and Ge.

```
# Perform a delta test for ATK-DFT over Si and Ge

# Setup calculator to use.
calculator = LCAOCalculator(
        iteration_control_parameters=IterationControlParameters(tolerance=1.0e-6, max_
↪steps=300),
        numerical_accuracy_parameters=NumericalAccuracyParameters(
            density_mesh_cutoff=GridSampling(60),
            k_point_sampling=[9,9,9]),
        exchange_correlation=GGA.PBE)

elements = [Silicon, Germanium]
# Construct the test.
```

(continues on next page)

(continued from previous page)

```
delta_test = DeltaTest(elements=elements, calculator=calculator)

# Print the delta test report.
nlprint(delta_test)

# Get the delta value for Si.
Si_delta = delta_test.delta(Silicon)

# Get the equation of state parameters for Si.
Si_equilibrium_volume, Si_bulk_modulus, Si_bulk_modulus_derivative, Si_bulk_minimum_
↪total_energy = delta_test.equationOfStateParameters(Silicon)

# Get the equation of state data E(V) for Ge.
Ge_volumes, Ge_total_energies = delta_test.equationOfState(Germanium)
```

Run DeltaTest against 3rd party code

By default, the DeltaTest is carried out against the WIEN2k data. Any other code can be compared to by providing the corresponding equation of state parameters in a file. This file must contain the reference equation of state parameters for each element to test organized in four columns: element symbol, equilibrium volume in units of cubic Angstrom, bulk modulus in units of GPa, and the normalized derivative of the bulk modulus which is unitless. The path to the reference data file is then given as a parameter to the DeltaTest constructor, e.g.

```
delta_test = DeltaTest(elements=[Silicon, Germanium],
    calculator=LCAOCalculator(),
    reference=/path/to/reference_file.txt
    )
```

Dense

class Dense

Cache self energies in dense matrix storage.

DensityOfStates

class DensityOfStates(*configuration*, *kpoints=None*, *energy_zero_parameter=None*, *bands_above_fermi_level=None*, *save_partial=None*, *enable_symmetry=None*, *processes_per_kpoint=None*, *method=None*)

Class for calculating the density of states for a configuration.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the density of states.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points for which to calculate the density of states. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.

•**energy_zero_parameter** (FermiLevel | AbsoluteEnergy) – Specifies the choice for the energy zero. **Default:** FermiLevel

•**bands_above_fermi_level** (int | All) – The number of bands above the Fermi level per principal spin channel. Must be a non-negative integer. **Default:** All (All bands are included), except for calculators of type *PlaneWaveCalculator*; in this case the default is equal to the number of bands above the Fermi level set by the calculator.

•**save_partial** (*bool*) – Save the partial contributions to the density of states. The *PlaneWaveCalculator* does not support calculation of partial contributions. **Default:** False

•**enable_symmetry** (*bool*) – Enable or disable the use of symmetry **Default:** True

•**processes_per_kpoint** (int | None) – The number of processes to use per kpoint. If None the same number of processes per k-point as used in the SCF calculation will be used. The parameter is only used by the *PlaneWaveCalculator*. **Default:** None

•**method** (None | Full | *KDotPExpansion3D*) – The method used for the eigenvalue computation.

The default is to perform an exact diagonalization to determine the eigenvalues at each requested k-point.

Alternatively, the k.p expansion method can be used to interpolate the eigenvalues at the requested k-points using the exact eigenvalues and eigenfunctions calculated in the SCF calculation.

NOTE: Only the *PlaneWaveCalculator* supports the k.p method at the moment.

calculateCarrierConcentration(*contribution=None*, *fermi_shift=None*, *temperature=None*)

Calculate the carrier concentration of the specified contribution.

Parameters

•**contribution** (Electron | Hole) – The contribution to the carrier contribution. **Default:** Electron

•**fermi_shift** (*PhysicalQuantity* of type energy.) – The shift in Fermi level. **Default:** 0.0 * eV

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature to use for the distribution. **Default:** 300 * Kelvin

Returns
The carrier concentration.

Return type
PhysicalQuantity of type inverse volume.

elements()

Query method for the elements in the configuration used for calculating the density of states.

Returns
A list of the elements.

Return type
list of *PeriodicTableElement*

energies()

Returns
The energies used for generating the last spectrum.

Return type
PhysicalQuantity of type energy.

energyMax()

Returns
The maximum energy eigenvalue.

Return type
PhysicalQuantity of type energy.

energyMin()

Returns
The minimum energy eigenvalue.

Return type
PhysicalQuantity of type energy.

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy.

evaluate(*spin=None*, *projection_list=None*, *normalization=None*)

Return the default density of states spectrum

Parameters

- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin of the DOS. **Default:** Spin.All
- **projection_list** (*ProjectionList*) – *ProjectionList* with projections to include in the density of states. **Default:** Total density of states.
- **normalization** (*False* | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** False

Returns
The density of states spectrum.

Return type
PhysicalQuantity of type reciprocal energy

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.All

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

gaussianSpectrum(*energies=None*, *spin=None*, *broadening=None*, *projection_list=None*)

Return the Density of States Spectrum using Gaussian broadening.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – List of energies for which the spectrum should be calculated. **Default:** Spectrum from *energyMin()* to *energyMax()*.
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin of the DOS. Must be Spin.All` for non-collinear and spin-orbit calculations. |DEFAULT| ``Spin.Up for unpolarized and polarized calculations.

- **broadening** (*PhysicalQuantity* of type energy) – Broadening of the Gaussian. **Default:** The default value is automatically determined based on the energy spectrum and k-point sampling. If the spectrum is calculated for a single k-point, the default value is `0.1 * eV`.
- **projection_list** (*ProjectionList*) – *ProjectionList* with projections to include in the density of states. **Default:** Total density of states.

Returns
The density of states spectrum.

Return type
PhysicalQuantity of type inverse energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Returns
The eigenvalue generation method.

Return type
None | Full | *KDotPExpansion3D*

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

processesPerKPoint()

Deprecated: from v2019

Returns
The number of processes per k-point used in the calculation.

Return type
int > 0

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spins()

Returns
The spins associated to this analysis object.

Return type
list of spin flags.

tetrahedronSpectrum(*energies=None*, *spin=None*, *projection_list=None*)

Return the Density of States Spectrum using the tetrahedron method.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – List of energies for which the spectrum should be calculated. **Default:** Spectrum from *energyMin()* to *energyMax()*.
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin of the DOS. Must be Spin.All` for non-collinear and spin-orbit calculations. |DEFAULT| ``UNPOLARIZED/POLARIZED: Spin.Up
- **projection_list** (*ProjectionList*) – ProjectionList with projections to include in the density of states. **Default:** Total density of states.

Returns
The density of states spectrum.

Return type
PhysicalQuantity of type inverse energy

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the Density of states of a silicon crystal:

```
# Set up Si in the diamond structure
si_crystal = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    cartesian_coordinates=[[ 0.      ,  0.      ,  0.      ],
                           [ 1.35765,  1.35765,  1.35765]]*Angstrom
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=
    NumericalAccuracyParameters(k_point_sampling=(4, 4, 4))
    )

si_crystal.setCalculator(calculator)

# Calculate the density of states
dos = DensityOfStates(si_crystal, kpoints = MonkhorstPackGrid(16,16,16))
# save the DOS to a file
nlsave('si_dos.nc', dos)
```

si_dos.py

Note: The spectrum can be shown by selecting the file si_dos.nc in the **LabFloor** of QuantumATK and using the **2D Plot** plugin.

Plot the Gaussian and tetrahedron spectrum within pylab:

```
# read DOS object from file
dos = nlread('si_dos.nc', DensityOfStates)[0]

# make list of energies
```

(continues on next page)

(continued from previous page)

```
energies = numpy.arange(-14,5,0.01)*eV

# calculate the DOS spectrum with two different methods
dos_t = dos.tetrahedronSpectrum(energies)
dos_g = dos.gaussianSpectrum(energies, broadening = 0.2*eV)

#plot the spectra using pylab
import pylab
pylab.figure()
pylab.plot(energies.inUnitsOf(eV), dos_t.inUnitsOf(eV**-1))
pylab.plot(energies.inUnitsOf(eV), dos_g.inUnitsOf(eV**-1))
pylab.xlabel("Energy (eV)")
pylab.ylabel("DOS (1/eV)")
pylab.show()
```

si_dos_plot.py

Notes

- The routine utilizes the symmetries of the crystal to reduce the computational load.
- The default spectrum is a Gaussian spectrum for a *MoleculeConfiguration* and a *BulkConfiguration* with less than 10 k-points. For a *BulkConfiguration* with more than 10 k-points, the tetrahedron method is used.
- The implementation of the tetrahedron method follows Ref.[1].
- The behavior of `bands_above_fermi_level` with regard to the spin types *unpolarized*, *polarized*, and *noncollinear* is described in *Notes* of the *Bandstructure* analysis object.

DensityProfile

class DensityProfile(*md_trajectory*, *start_time=None*, *end_time=None*, *resolution=None*, *atom_selection=None*, *direction_index=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the DensityProfile object.

Parameters

- **md_trajectory** (*MDTrajectory* | `AtomicConfiguration`) – The MDTrajectory or configuration to calculate the density profile for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** the last time frame.
- **resolution** (*PhysicalQuantity* of type length) – The bin size, which determines the resolution of the profile. **Default:** `2.0 * Angstrom`
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this atom selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All elements are considered
- **direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Can only be an element of [0, 1, 2]. **Default:** `2`

[1] P.ăE. Blochl, O.ăJepsen, and O.ăK. Andersen. Improved tetrahedron method for Brillouin-zone integrations. *Phys. Rev. B*, 49:16223–16233, Jun 1994. doi:10.1103/PhysRevB.49.16223.

- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the density profile.

zValues()

Return the positions of the bins associated with the density profile.

Usage Examples

Load an *MDTrajectory* and calculate the mass density profile along the A-vector of the cell:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

# Calculate the mass density profile along the A-vector of the cell.
density_profile = DensityProfile(md_trajectory,
                                 direction_index=0)

# Get the mass density in gram/ccm and the positions of the bin centers.
mass_density = density_profile.data().inUnitsOf(kiloGram/Meter**3)/1000.0
bin_centers  = density_profile.zValues().inUnitsOf(Angstrom)

# Plot the data using pylab.
import pylab

pylab.plot(bin_centers, mass_density,
           label='Mass density of alumina along A-vector')
pylab.xlabel('a (Angstrom)')
pylab.ylabel('d(a) (g/cm**3)')
pylab.legend()

pylab.show()
```

density_profile.py

DeviceAlgorithmParameters

class DeviceAlgorithmParameters(*initial_density_type=None*, *electrode_constraint=None*, *self_energy_calculator_real=None*, *self_energy_calculator_complex=None*, *non_equilibrium_method=None*, *equilibrium_method=None*, *store_grids=None*, *store_basis_on_grid=None*, *scf_restart_step_length=None*, *enforce_different_electrodes=None*, *neumann_self_energy_update_strategy=None*, *density_predictor=None*)

Class for representing algorithm parameters relevant for the device calculation.

Parameters

- **initial_density_type** (*EquivalentBulk* | *NeutralAtom* | RepeatedElectrode) – Select the method used for constructing the initial density. **Default:** *NeutralAtom*

- **electrode_constraint** (*ElectrodeConstraint.Off* | *ElectrodeConstraint.DensityMatrix*) – Constrain system variables to the electrode bulk values (see *ElectrodeConstraint* for the available options.) The default value differs if the class is used in a *DeviceConfiguration* or in a *SurfaceConfiguration*. **Default:**

  – *DeviceConfiguration*
  ElectrodeConstraint.Off

  – *SurfaceConfiguration*
  *ElectrodeConstraint.DensityMatrix(10.0 * Angstrom)*

- **self_energy_calculator_real** (*RecursionSelfEnergy* | *KrylovSelfEnergy* | *DirectSelfEnergy* | *SparseRecursionSelfEnergy*) – The SelfEnergyCalculator that should be used for calculating the real energies in the contour integral. **Default:** A default *RecursionSelfEnergy* instance

- **self_energy_calculator_complex** (*RecursionSelfEnergy* | *SparseRecursionSelfEnergy*) – The SelfEnergyCalculator that should be used for calculating the complex energies in the contour integral. **Default:** A default *RecursionSelfEnergy* instance

- **non_equilibrium_method** (*GreensFunction* | *SparseGreensFunction* | *ScatteringStates*) – The density matrix method employed in the real part of the contour integration. **Default:** A default *GreensFunction* instance

- **equilibrium_method** (*GreensFunction* | *SparseGreensFunction*) – The density matrix method employed in the complex part of the contour integration. **Default:** A default *GreensFunction* instance

- **store_grids** (*bool*) – Enable or disable the storage of grids that can be reused in the SCF process. True will increase the memory usage, but reduce the computation time. **Default:** `True`

- **store_basis_on_grid** (`Automatic` | `True` | `False`) – Control the storage of the basis functions on a real space grid. When enabled, increases the memory usage but reduces the computation time. **Default:** `Automatic`

- **scf_restart_step_length** (*PhysicalQuantity* of type length) – The maximum distance that an atom can move so that the calculation can be restarted from the previous initial state. **Default:** `0.1 * Ang`

- **enforce_different_electrodes** (*bool*) – Enforce separate calculations for the left and right electrodes, when enabled. When disabled, electrodes are treated identical if they differ by a constant translation. **Default:** `False`

- **neumann_self_energy_update_strategy** (`SurfaceAverage` | `VolumeAverage`) – Defines how the self-energies are updated during a self-consistent loop. Only relevant when Neumann boundary conditions in the transport direction are used. The self-energies are shifted rigidly to ensure the electrode is aligned in energy with the corresponding electrode repetition in the device, by inspection of the electrostatic potential. If `SurfaceAverage` is selected, a surface average at the boundary between device and electrode is used. If `VolumeAverage` is selected, a volume average on atoms in the minimal electrode and in the device atoms corresponding to the minimal electrode is used. In both cases, only regions within the atomic basis compact support are considered. **Default:** `SurfaceAverage`

- **density_predictor** (`DensityPredictor` | False) – The parameters for the density predictor to be used for the initial guess for density in a DFT calculation. This option is currently only used for DeviceLCAOCalculator. **Default:** `False`

densityPredictor()

Returns
The density predictor to be used for the initial guess for density in a DFT calculation.

Return type
`DensityPredictor` | False

electrodeConstraint()

Returns
The type of the electrode constraint.

Return type
`ElectrodeConstraint.Off` | *DensityMatrix*

enforceDifferentElectrodes()

Returns
If separate calculations for the left and right electrodes should be enforced.

Return type
bool

equilibriumMethod()

Returns
The equilibrium method for the complex contour integration.

Return type
GreensFunction | *SparseGreensFunction*

initialDensityType()

Returns
The initial density type.

Return type
EquivalentBulk | *NeutralAtom*

neumannSelfEnergyUpdateStrategy()

Returns
The strategy used to update self energy when using Neumann boundary conditions in the transport direction.

Return type
SurfaceAverage | VolumeAverage

nlinfo()

Returns
The nlinfo.

Return type
dict

nonEquilibriumMethod()

Returns
The non-equilibrium method for the real contour integration.

Return type
GreensFunction | *SparseGreensFunction* | *ScatteringStates*

scfRestartStepLength()

Returns
The SCF restart step length of this calculator.

Return type
PhysicalQuantity of type length

selfEnergyCalculatorComplex()

Returns
The self energy calculator for the complex energies.

Return type
RecursionSelfEnergy | *SparseRecursionSelfEnergy*

selfEnergyCalculatorReal()

Returns
The self energy calculator for the real energies.

Return type
RecursionSelfEnergy | *KrylovSelfEnergy* | *DirectSelfEnergy* | *SparseRecursionSelfEnergy*

storeBasisOnGrid()

Returns
If the storage of the basis functions on a real space grids is enabled or not

Return type
Automatic | True | False

storeGrids()

Returns
If the storage of grids is enabled or not

Return type
bool

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Setup a non-self-consistent calculation with an equivalent bulk initial density.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    initial_density_type=EquivalentBulk(electrode_constraint_length=7.*Ang)
    )

calculator = DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    iteration_control_parameters=NonSelfconsistent,
    )
```

Use a *RecursionSelfEnergy* for the real axis contour integral

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_real=RecursionSelfEnergy(storage_strategy=NoStorage())
    )

calculator = DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    electrode_voltages=(-0.5,0.5)*Volt
    )
```

Notes

The `initial_density_type` is only used for setting up the starting guess for the self-consistent loop, while the `electrode_constraint` will constrain the density matrix at each step during the self-consistent loop.

The `self_energy_calculator_complex` is used for evaluating the self energy at each point along the complex contour. The `self_energy_calculator_real` is only used for calculating the real axis integral for finite bias calculations. Thus for zero bias calculations, the keyword `self_energy_calculator_real` has no effect.

DeviceConfiguration

class DeviceConfiguration(*central_region*, *electrodes*, *equivalent_electrode_lengths=None*, *transverse_electrode_repetitions=None*)

A two-probe configuration consisting of a central region coupled to two electrodes.

Parameters

- **central_region** (*BulkConfiguration*) – The scattering region of the device.
- **electrodes** (sequence (size 2) of *BulkConfiguration*) – The semi-infinite regions on either side of the device.
- **equivalent_electrode_lengths** (sequence (size 2) of *PhysicalQuantity* of type length) – The lengths to use for the equivalent electrode regions in the central region. **Default:** Length of the actual electrodes given.
- **transverse_electrode_repetitions** (*sequence (size 2) of sequence (size 2) of int*) – Repetitions of the electrodes in the A and B directions, given as a list of two integers for both electrodes. **Default:** `[[1, 1], [1, 1]]`

addBonds(*bond_list*)

Add bonds.

Parameters
bond_list (*numpy.ndarray*) – The list of bonds to add.

addTags(*tags*, *indices=None*)

Add a set of tags to atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms.
- **indices** (*list | int | None*) – The list of indices to match atoms against. **Default:** All indices.

atomicMasses()

Returns
The masses of the atoms in the configuration.

Return type
PhysicalQuantity of type mass

atomicNumbers()

Returns
The list of atomic numbers associated with the elements.

Return type
list of ints

bias()

Calculate the bias of the attached calculator If no calculator is attached return None.

Returns
The bias.

Return type
PhysicalQuantity with electric potential units | None

bonds()

Returns
An array with the the two atom indices for each bond in the central region along with the vector which periodic images this bond connects.

Return type
array

bravaisLattice()

Returns
The bravais lattice of the central region.

Return type
BravaisLattice

calculator()

Returns
The calculator attached to the configuration, i.e. the calculator that will be used for both simulation and analysis.

Return type
Calculator

cartesianCoordinates()

The Cartesian coordinates of the atoms in the the central region of the configuration.

Returns
The Cartesian coordinates.

Return type
PhysicalQuantity of type length

centralRegion()

The central region of the device.

Returns
The central region.

Return type
BulkConfiguration

coarseGrainDescriptors()

Returns
The list of either elements or coarse grain particles for each element. Isotopes are returned as just their base element, UnitedAtoms and Particle are given as their specific type.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

copy()

Returns
A copy of the current configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndDeleteAtoms(*indices*)

Create a new configuration by deleting some atoms from this configuration.

Parameters
indices (*list of int*) – The indices of the atoms to delete.

Returns
The configuration with some atoms deleted.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndMerge(*other*)

Create a new configuration by merging this configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The other configuration.

Returns
The merged configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndShiftAtoms(*displacement*, *indices=None*)

Create a new configuration with some atoms translated.

Parameters

- **displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.
- **indices** (*list of int*) – The indices to shift. **Default:** All.

Returns
The configuration with the translation applied.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

crossSection()

Return the cross-sectional area of the central region.

Returns
The cross-sectional area.

Return type
float

deleteAtoms(*indices*)

Delete the specified atoms.

Parameters
indices (*array of int*) – The indices to delete.

deleteBonds(*bond_list=None*, *pair_selection=None*)

Delete bonds connected to atomic indices.

Parameters

- **bond_list** (*A two-dimensional sequence*) – The pairs of bondes indices.
- **pair_selection** (*list | None*) – Specifies two groups between which bonds are delete. Selectable groups are elements, index lists, tag names, or None (all atoms).

dielectricRegions()

Returns
The dielectric regions in the central region.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

electrodes()

The electrodes of the device.

Returns
The pair of electrodes belonging to this configuration.

Return type
list of *BulkConfiguration*

electrodesDisplacement()

The displacements of the `BravaisLattice` of the electrodes in the C-direction in order to match the `BravaisLattice` of the central_region.

Returns
The electrode displacements.

Return type
PhysicalQuantity of type length

electrodesWithCalculator()

Returns
A copy of the electrodes of the configuration with attached electrode calculator. If the configuration is updated, the calculators include a copy of the updated state.

Return type
list of *BulkConfiguration*

elements()

Returns
The elements in configuration.

Return type
list of *PeriodicTableElement*

externalPotential()

Returns
The external potential present in the central region.

Return type
AtomicShift | *AtomicCompensationCharge*

findBonds(*fuzz_factor=1.1*, *pair_selection=None*)

Find bonds in the configuration according to the combined covalent radii of the element pairs, multiplied with a fuzz factor. Optionally, find bonds only between two specified sub-groups of atoms. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.
- **pair_selection** (list(2) of type *PeriodicTableElement*, list of int, or str.) – Specifies two groups between which bonds are detected. Selectable groups are elements, index lists, tag names, or None (all atoms). By default bonds between all atoms in the configuration are taken into account.

fixedSpinDirections()

Returns
The fixed spin directions for the configuration.

Return type
FixedSpin | None

fractionalCoordinates()

Returns
The fractional coordinates of the central region.

Return type
array of floats

generateShifts()

Method for generating a list of origin shifts along two lattice directions.

It will create a list of 9 shifts of the supercell origin (-1 to 1) in x and y periodic directions.

Returns
The array of shifts.

Return type
PhysicalQuantity of type length

ghostAtoms()

Returns
The list of ghost atoms of the central region.

Return type
list of ints

improperDihedralIndices()

Returns
The list of atom indices for each improper dihedral or None if no improper dihedrals are defined. Improper dihedrals are mainly used in bonded force fields.

Return type
numpy array | None

indices()

Returns
The indices of the atoms.

Return type
list of int

indicesFromIsotopes(*isotopes*)

Parameters
isotopes (*list of type PeriodicTableElement or Isotope*) – The isotopes to select.

Returns
The indices of the selected isotopes.

Return type
list of type int

indicesFromTags(*tags=None*)

List the indices associated with a given collection of tags.

Parameters
tags (*list | str*) – A list of tags for which all matching indices should be extracted.

Returns
The list of indices corresponding to the specified tag name(s).

Return type
list of ints

magneticMoments()

Returns
The magnetic moments of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type Bohr magneton.

merge(*other*)

Merge configuration with another configuration.

NOTE: For devices this method shifts the atoms of the other configuration to the center.

Parameters
other (*AtomicConfiguration*) – A different AtomicConfiguration.

metallicRegions()

Returns
The metallic regions in the central region.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The configuration information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AtomicConfiguration object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger().

numberOfAtoms()

Returns
The total number of atoms in the configuration.

Return type
int

partialCharges(*indices=None*)

Get the list of partial atomic charges that can be used for representing electrostatic interactions in ATK-ForceField.

Parameters
indices (*list | int | None*) – The indices for which to return the partial charges. **Default:** All indices.

Returns
A PhysicalQuantity array of the atomic partial charge for each atom.

Return type
PhysicalQuantity of type charge | None

particleDescriptors()

Returns
The list of particle descriptors for each atom. This should return the exact same list as was given in the constructor argument elements.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

static periodicBoundaries()

Returns
The periodic boundary conditions of the configuration.

Return type
list

primitiveVectors()

Returns
The primitive lattice vectors.

Return type
PhysicalQuantity of type length

removeTags(*tags=None*, *indices=None*, *purge=False*)

Remove a set of tags from atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms. **Default:** All tags.
- **indices** (*list | int*) – The list of indices to match atoms against. **Default:** All indices.

•**purge** (*bool*) – When removing tags from the configuration, delete the tag completely when not associated with any atoms anymore. **Default:** False

repeat(*na=1*, *nb=1*, *nc=1*, *stack_systems=False*)

Repeat the derived class with the integer values na, nb, and nc along the three primitive unit cell vectors. The repeated system is constructed with a cell of the type UnitCell.

Parameters

•**na** (*int*) – The repetition along the a-axis. **Default:** 1.

•**nb** (*int*) – The repetition along the b-axis. **Default:** 1.

•**nc** (*int*) – The repetition along the c-axis. **Default:** 1.

•**stack_systems** (*bool*) – If True the basis atoms are repeated as a unit, i.e. a0 and b0 are repeated as: a0,b0,a1,b1, If False the basis atom are repeated individually, i.e. a0 and b0 are repeated as: a0,a1,, b0,b1, **Default:** True.

Returns
The repeated system.

Return type
DeviceConfiguration | *SurfaceConfiguration*

scalePartialCharges(*scale_factor*, *indices=None*)

Scale the partial charges with a given scale factor. These partial charges are used with Forcefield calculators.

Parameters

•**scale_factor** (*float*) – The factor for scaling charges.

•**indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

setBonds(*bond_list*, *skip_checks=False*)

Set the bonds on the configuration. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

•**bond_list** (*list(n, 2) | list(n, 5) | None*) – A list which contains for each bond the indices of the two connected atoms. Optionally, three more integers can be specified for each bond, which must be between -1 and 1, and which denote to which neighboring image cell the bond is connected. Without these additional indices, the minimum image convention is obeyed.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value.

setCalculator(*calculator*, *initial_state=None*, *initial_spin=None*)

Attach a *Calculator* to the configuration which will be used in calculations involving the configuration.

Parameters

•**calculator** (*Calculator*) – The calculator object that should be attached to the configuration.

•**initial_state** (*DeviceConfiguration* with a calculator | None) – The initial state to be used for this configuration. **Default:** No initial state.

•**initial_spin** (*InitialSpin* | None) – The initial InitialSpin object to be used for this configuration. **Default:** `InitialSpin()`, except when used with a calculator of type *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*. In this case `InitialSpin([0., 0., ...])` is used, i.e. the scaled spins for each atom are set to 0.

setCartesianCoordinates(*cartesian_coordinates*, *indices=None*, *skip_checks=False*)

Set the Cartesian coordinates of the atoms in the central region. Changes in the coordinates of the electrode extension will be synchronized with the corresponding coordinates in the electrodes.

Parameters

- **cartesian_coordinates** (*PhysicalQuantity* of type length) – The new coordinates of the atoms in each image.
- **indices** (*list*) – The indices of the atoms to set the positions of. **Default:** All indices.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False

setDielectricRegions(*dielectric_regions*)

Set the dielectric regions for the central region.

Parameters
dielectric_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

setExternalPotential(*external_potential*)

Set an external potential on the configuration that will be used in calculations involving the configuration.

Parameters
external_potential (*AtomicShift* | *AtomicCompensationCharge*) – The external potential to apply.

setImproperDihedralIndices(*improper_dihedral_indices*)

Set the list of atom indices for each improper dihedral in bonded force fields.

Parameters
improper_dihedral_indices (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

setMagneticField(*magnetic_field*)

Set local magnetic field. The spins will be forced to point in the directions given by the magnetic_field object. The magnetic field can be defined for each atom. This only has an effect for Noncollinear or Spinorbit calculations.

Parameters
magnetic_field (*FixedSpin*) – The magnetic field to be applied.

setMagneticMoments(*magnetic_moments=None*, *skip_checks=False*)

Function to set magnetic moments on the configuration.

Parameters

- **magnetic_moments** (*PhysicalQuantity* of type Bohr magneton.) – The magnetic_moments to set on the configuration. Has the dimensionality nx3. **Default:** None.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setMetallicRegions(*metallic_regions*)

Set the metallic regions for the central region.

Parameters
metallic_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setPartialCharges(*charge_list*, *indices=None*, *skip_checks=False*, *update_calculator=True*)

Set the partial charges on the configuration. The partial charges are used primarily to model electrostatic interactions in the TremoloX-calculator.

Parameters

•**charge_list** (*PhysicalQuantity of type charge | None*) – A list of atomic partial charges which contains a charge for each atom.

•**indices** (*list | int | None*) – The indices for which to set the partial charges. **Default:** All indices.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

•**update_calculator** (*bool*) – Whether to update and attached Forcefield calculator with the new charges. **Default:** True.

setVelocities(*velocities=None*, *skip_checks=False*)

Function to set velocities on the configuration.

Parameters

•**velocities** (*PhysicalQuantity* of type velocity | None) – The velocities to set on the configuration. Has the dimensionality nx3. **Default:** None.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

shiftAtoms(*displacement*, *indices=None*, *skip_checks=False*)

Translate some atoms.

Parameters

•**displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.

•**indices** (*list of int*) – The indices to shift. **Default:** All.

•**skip_checks** (*bool*) – True, if all consistency checks should be skipped.

shiftPartialCharges(*total_charge*, *indices=None*)

Shift the partial charges so that their sum is the given total charge value. These partial charges are used with Forcefield calculators.

Parameters

•**total_charge** (*PhysicalQuantity of type charge*) – The new total charge.

•**indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

symbols()

Returns
The element symbols of the configuration.

Return type
list of str

tags(*indices=None*)

List the tags associated with a given collection of indices. The list returned is the set union of tags associated with the given indices. If no collection of indices is provided, then all tags on the configuration are returned.

Parameters
indices (*list | int*) – The indices to check. **Default:** All indices.

Returns
The set union of tags present on the provided indices.

Return type
set

transverseElectrodeRepetitions()

Returns
The transverse electrode repetitions.

Return type
list (size 2) of list (size 2) of int

uniqueElements(*ordered=False*)

Parameters
ordered (*bool*) – If the elements should be returned in ascending order by atomic number.

Returns
The unique elements contained in the configuration.

Return type
list of *PeriodicTableElement*

uniqueString()

Return a unique string representing the state of the object.

update(*force_restart=False*)

A self-consistent solution is generated, using the currently set calculator.

Parameters
force_restart (*bool*) – Force the self-consistent calculation to restart. **Default:** False.

velocities()

Returns
The velocities of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type velocity

Usage Examples

Construct a device system consisting of a hydrogen molecule connected with two chains of Lithium atoms:

```
# Define A,B directions of lattice
vector_a = [5.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.0, 0.0]*Angstrom

#  setup electrode
electrode = BulkConfiguration(
    bravais_lattice=UnitCell(vector_a, vector_b,
                             [0.0, 0.0, 9.0]*Angstrom),
    elements=[Lithium, Lithium, Lithium],
    cartesian_coordinates=[[ 2.5,  2.5,  1.5],
                           [ 2.5,  2.5,  4.5],
                           [ 2.5,  2.5,  7.5]]*Angstrom
    )

# setup  Central region
```

(continues on next page)

(continued from previous page)

```
central_region = BulkConfiguration(
    bravais_lattice= UnitCell(vector_a, vector_b,
                              [0.0, 0.0, 22.0]*Angstrom),
    elements=[Lithium, Lithium, Lithium, Hydrogen, Hydrogen,
              Lithium, Lithium,  Lithium],
    cartesian_coordinates=[[  2.5,   2.5,   1.5],
                           [  2.5,   2.5,   4.5],
                           [  2.5,   2.5,   7.5],
                           [  2.5,   2.5,  10.5],
                           [  2.5,   2.5,  11.5],
                           [  2.5,   2.5,  14.5],
                           [  2.5,   2.5,  17.5],
                           [  2.5,   2.5,  20.5]]*Angstrom
     )

#setup Device configuration
device_configuration = DeviceConfiguration(
    central_region,
    [electrode, electrode]
    )
```

li_h2_li.py

Construct a device system from bulk lithium with transverse electrode repetitions:

```
# Set up the left minimal electrode with 2 Li atoms.
left_electrode_lattice = Hexagonal(3.111*Angstrom, 5.093*Angstrom)
left_electrode_elements = [Lithium, Lithium]
left_electrode_coordinates = [[ 0.333333333333,  0.666666666667,  0.25          ],
                              [ 0.666666666667,  0.333333333333,  0.75          ]]

left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    fractional_coordinates=left_electrode_coordinates,
    )

# Use the same electrode as the right electrode.
right_electrode = left_electrode

# Define transverse electrode repetitions; 2 repetitions
# in A and 3 in B for both electrodes.
transverse_electrode_repetitions = [[2, 3], [2, 3]]

# Create the repeated electrode and repeat it 4 times along
# C to create the central region. In this manner, the atoms in the electrode
# extensions will match the repeated electrodes.
repeated_left_electrode = left_electrode.repeat(
    transverse_electrode_repetitions[0][0],
    transverse_electrode_repetitions[0][1],
    1)
central_region = repeated_left_electrode.repeat(1, 1, 4)
```

(continues on next page)

(continued from previous page)

```
# Change one of the atoms at the center of the central region to make a non-homogeneous␣
↪device.
central_region_lattice = central_region.bravaisLattice()
central_region_elements = central_region.elements()
central_region_coordinates = central_region.fractionalCoordinates()

central_region_elements[24] = Hydrogen

central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    fractional_coordinates=central_region_coordinates,
    )

# Set up the device configuration with transverse repetitions.
device_configuration = DeviceConfiguration(
    central_region=central_region,
    electrodes=[left_electrode, right_electrode],
    transverse_electrode_repetitions=transverse_electrode_repetitions,
    )
```

`device_transverse.py`

Notes

ATK recognizes four types of atomic geometries:

•Molecules (*MoleculeConfiguration*)

•Bulk (*BulkConfiguration*)

•One Probe (*SurfaceConfiguration*)

•Two Probe (*DeviceConfiguration*)

A two-probe setup is created by attaching two bulk electrodes to the left and the right of a central region. The properties within the electrode regions are fixed at their bulk properties, whereas the central region is described self-consistently. The device configuration must fulfill the following properties:

•The electrodes and the central *BulkConfiguration* objects must have the same cell vectors in the A and B direction. If `transverse_electrode_repetitions` have been specified, the electrodes are assumed to be the minimal, irreducible ones in the A and B directions. In this case, the electrode unit cell vectors in the A and B directions multiplied by the transverse electrode repetitions must equal the central regions lattice vectors.

•The C direction must be perpendicular to the A and B directions.

•The first atoms in the central region must be positioned identically to the atoms in the left electrode (repeated by the correponding repetitions in A and B if `transverse_electrode_repetitions` have been specified) when the central region AB faces are aligned with the AB faces of the electrodes. See Fig. 4.14 for an example.

•Similarly, the last atoms in the central region must be positioned identically to the atoms in the right (repeated) electrode. See Fig. 4.14 for an example.

•The parameter `equivalent_electrode_lengths` can be used to control the number of electrode repeats along the C direction in the electrode extensions of the central region. The sum of the left and right equivalent electrode lenghts cannot exceed the lenght of the central region. Fig. 4.14 shows an example of two repeats of both the left and right electrodes along C in the corresponding electrode extensions. If the left and right electrode lengths are 5 and 7 `Angstrom`,

respectively, this example corresponds to setting the parameter to the value `equivalent_electrode_lengths=[2 * 5.0, 2 * 7.0] * Angstrom`.

If there is structural periodicity in the electrodes in the A and/or B directions, using the minimal, irreducible electrodes and specifying the corresponding `transverse_electrode_repetitions` can speed up the device calculation significantly and reduce memory usage.

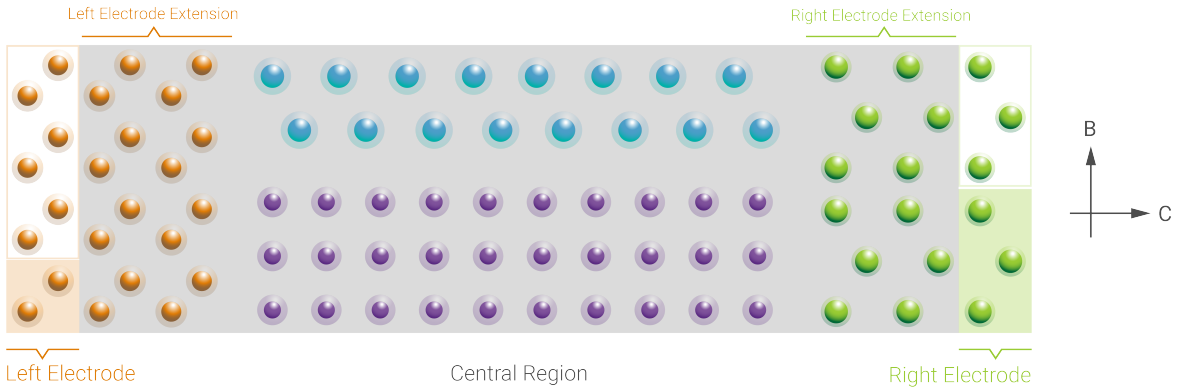


Fig. 4.14: Example of the constituents of a device system with 2 electrodes. The constituents are the left electrode, central region, and right electrode. The central region of the device is marked with a gray color, and the electrode extension regions of the central region are also shown. In this case, the repeated left electrode consists of four repetitions of the minimal electrode along the B direction, whereas the repeated right electrode consists of two repetitions along B of the corresponding minimal electrode, i.e., `transverse_electrode_repetitions=[[1, 4], [1, 2]]` (note that the A direction is not shown, and assumed not to have repetitions). Furthermore, the equivalent electrode lenghts in the central region correspond to twice the length of the electrodes in the C direction.

Attention: If `transverse_electrode_repetitions` have been specified, the `k_point_sampling` of the `electrode_calculators` in the device calculator (see *DeviceLCAOCalculator* and *DeviceSemiEmpiricalCalculator*) must be compatible with the `k_point_sampling` of the central region, i.e., the k-point density of the minimal electrodes along the A and B directions must be the same as the k-point density in the central region along A and B.

Vacuum basis sets (ghost atoms) can be included in the *DeviceConfiguration* by specifying a `ghost_atoms` list for the *BulkConfiguration* of the central and electrode regions.

DeviceDensityOfStates

class DeviceDensityOfStates(*configuration*, *energies=None*, *kpoints=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*)

Constructor for the Device Density of States object.

Parameters

- **configuration** (*DeviceConfiguration*) – The one or two-probe configuration with attached calculator for which the density of states. should be calculated.
- **energies** (*PhysicalQuantity* of type energy.) – The energies for which the density of states should be calculated. **Default:** `numpy.linspace(-2., 2., 200)*eV`

- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points for which the density of states should be calculated. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.
- **contributions** (`Left` | `Right` | `All`) – The density contributions to include in the density of states. **Default:** `All`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used for the density of states. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`.) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small energy, used to move the density of states calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

bias()

Returns
The applied bias.

Return type
PhysicalQuantity type voltage.

electrodeFermiLevels()

Returns
The Fermi levels of the left and right electrodes in absolute energies.

Return type
PhysicalQuantity of type energy.

electrodeFermiTemperatures()

Returns
The Fermi temperature of the left and right electrodes used in this density of states.

Return type
PhysicalQuantity of type temperature.

electrodeVoltages()

Returns
The left and right electrode_voltages used in this density of states spectrum.

Return type
PhysicalQuantity of type voltage.

elements()

Returns
The elements in the configuration used for generating the density of states.

Return type
list of *PeriodicTableElement*

energies()

Returns
The energies used in this density of states.

Return type
PhysicalQuantity of type energy.

energyZero()

Returns
The energy zero used for the energy scale in this density of states.

Return type
PhysicalQuantity of type energy.

evaluate(*spin=None*, *projection_list=None*)

Evaluate the density of states.

Parameters

- **spin** (Spin) – The spin component. **Default:** Spin.Sum
- **projection_list** (*ProjectionList*) – ProjectionList with projections to include in the density of states. **Default:** ProjectionList(range(n_left, n_central-n_right))

Returns
The density of states for the given spin component. For the spin Spin.All, a list containing the density of states for Spin.Sum, Spin.X, Spin.Y and Spin.Z is returned.

Return type
PhysicalQuantity | list(4) of *PhysicalQuantity*

infinitesimal()

Returns
The infinitesimal used for calculating the density of states.

Return type
PhysicalQuantity of type energy.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfElectrodeAtoms()

Returns
The number of atoms in each electrode in the configuration used for generating the density of states.

Return type
tuple of int.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spins()

Returns
The spins used in this density of states.

Return type
tuple of spin component flags.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *DeviceDensityOfStates* (DDOS) of a Li-H2-Li system:

```
# setup Huckel calculation for Li-H2-Li device

electrode_lattice = UnitCell([8., 0., 0.]*Angstrom,
                             [0., 8., 0.]*Angstrom,
                             [0., 0., 8.7]*Angstrom)

electrode_elements = [Lithium, Lithium, Lithium]
electrode_coordinates = [ [ 4.,  4., 1.45+i*2.9] for i in range(3)]*Angstrom

electrode = BulkConfiguration(
    bravais_lattice=electrode_lattice,
    elements=electrode_elements,
    cartesian_coordinates=electrode_coordinates
    )

lattice = UnitCell([8., 0., 0.]*Angstrom,
                   [0., 8., 0.]*Angstrom,
                   [0., 0., 37.376]*Angstrom)

elements = [Lithium,]*6 +[Hydrogen,]*2+[Lithium,]*6

coordinates = [ [ 4.,  4., 1.45+i*2.9] for i in range(6)] + \
              [ [ 4.,  4., 18.286+i*0.8] for i in range(2)] + \
              [ [ 4.,  4., 21.426+i*2.9] for i in range(6)]

central_region = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    cartesian_coordinates=coordinates*Angstrom
    )

device_configuration = DeviceConfiguration(
    central_region,
    [electrode, electrode]
    )

calculator = DeviceHuckelCalculator(
    iteration_control_parameters = NonSelfconsistent,
```

(continues on next page)

(continued from previous page)

```
    )

device_configuration.setCalculator(calculator)

device_configuration.update()
nlsave('device.nc', device_configuration)

#setup energies, and calculate DOS
energies = numpy.linspace(-2,3,100) * eV
dos = DeviceDensityOfStates(device_configuration, energies=energies)
nlsave('device.nc', dos)
```

device.py

Plot the projected density of states (PDOS) for the non-surface atoms of the central region:

```
# read the dos
dos = nlread('device.nc',DeviceDensityOfStates)[0]
conf = nlread('device.nc', DeviceConfiguration)[0]
energies = dos.energies()

# calculate projection on atoms in the central region
dos_projection = [dos.evaluate(projection_list = ProjectionList([i])) \
                  for i in range(3,11)]

# plot the projections
import pylab
pylab.figure()
for i in range(3,11):
    label = conf.elements()[i].symbol()+'('+str(i)+')'
    pylab.plot(energies.inUnitsOf(eV),dos_projection[i-3].inUnitsOf(eV**-1), \
               label = label)
pylab.legend()
pylab.xlabel("Energy (eV)")
pylab.ylabel("dos (1/eV)")
pylab.show()
```

dos.py

Calculate the contribution to the density of states from scattering states originating in the left electrode:

```
conf = nlread('device.nc',DeviceConfiguration)[0]

#setup energies
energies = numpy.linspace(-2,3,100) * eV

#calculate the dos from the Left electrode
dos = DeviceDensityOfStates(conf, energies=energies, contributions=Left)

#calculate the projections on the atoms in the central region
dos_projection = [
     dos.evaluate(projection_list = ProjectionList([i])) for i in range(3,11)
     ]
```

(continues on next page)

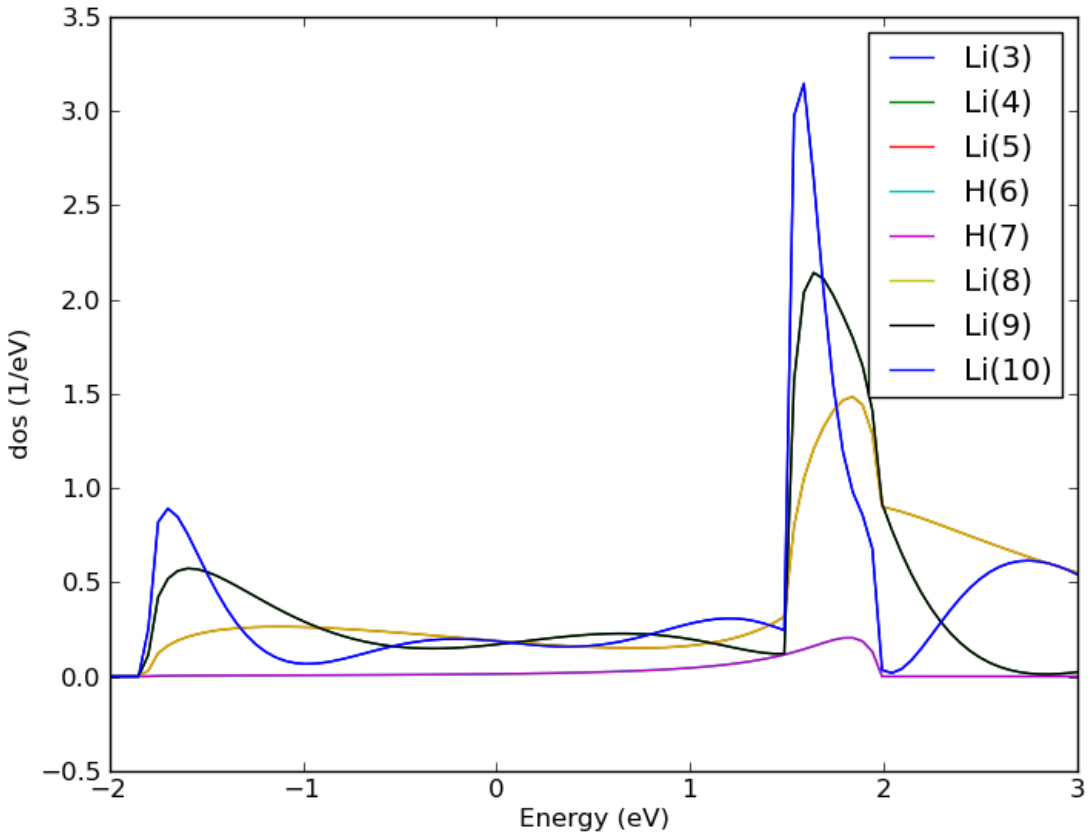


Fig. 4.15: The density of states of the Li-H2-Li systems projected onto the atoms in the central region. Note that half of the curves are not visible because they fall on top of the other curves due to symmetry.

(continued from previous page)

```
#plot the data
import pylab
pylab.figure()
for i in range(3,11):
    label = conf.elements()[i].symbol()+'('+str(i)+')'
    pylab.plot(
        energies.inUnitsOf(eV),
        dos_projection[i-3].inUnitsOf(eV**-1),
        label=label
        )
pylab.legend()
pylab.xlabel("Energy (eV)")
pylab.ylabel("dos (1/eV)")
pylab.show()
```

dos_left.py

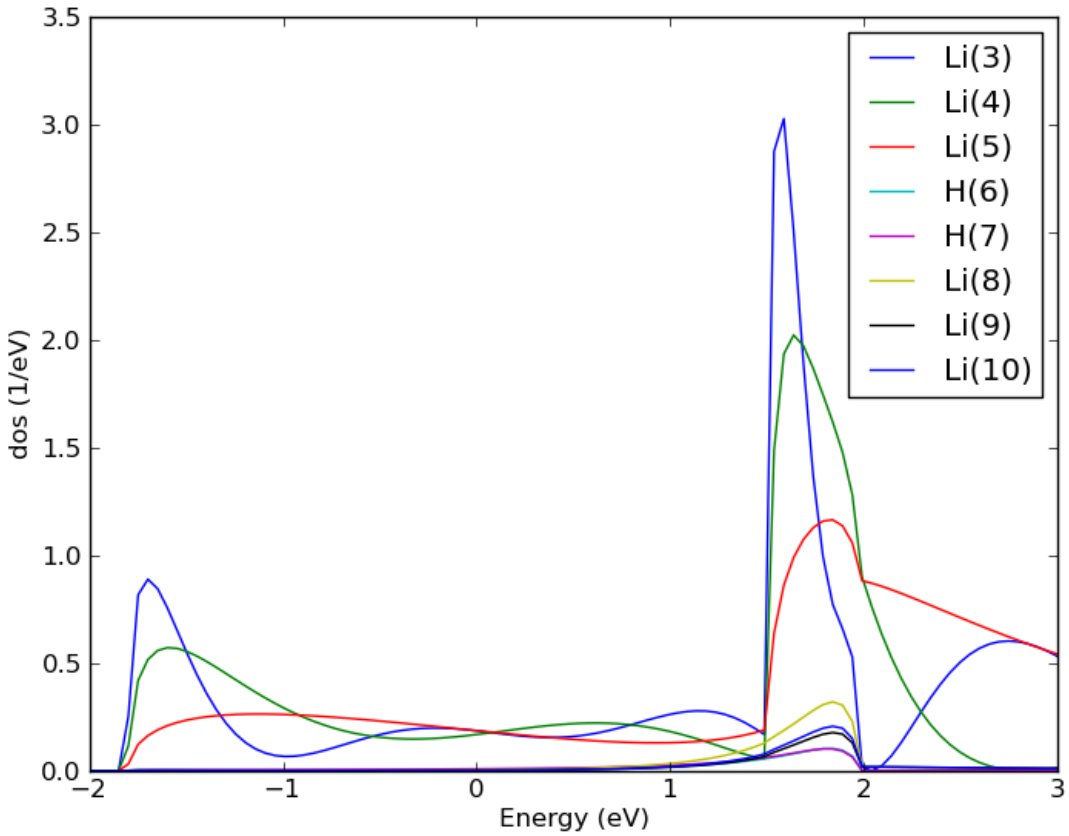


Fig. 4.16: The density of states from the left electrode of the Li-H2-Li systems projected onto the atoms in the central region. Note how the density of states in the right lithium atoms is reduced due do damping of the propagation by the hydrogen molecule.

Do the k-point average of the DOS:

```
dos = DeviceDensityOfStates(conf,
                            kpoints=MonkhorstPackGrid(4,4),
                            energies=energies,
                            contributions=Left)
```

Notes

The routine utilizes the symmetries of the configuration to reduce the computational load for k-point averaging.

The *DeviceDensityOfStates* (DDOS) $D(E)$ is computed via the spectral density matrix $\rho(E) = \rho^L(E) + \rho^R(E)$ (cf. *The non-equilibrium Greens function (NEGF) method*) where L/R denotes the contribution from the left/right electrode (see the keyword `contributions` above).

The *local density of states (LDOS)* is defined as

$$D(E, \mathbf{r}) = \sum_{ij} \rho_{ij}(E)\phi_i(\mathbf{r})\phi_j(\mathbf{r}),$$

where we note that the basis set orbitals $\phi_i(\mathbf{r})$ are real functions in QuantumATK through the use of solid harmonics.

The device density of states is then obtained by integrating the LDOS over all space:

$$D(E) = \int d\mathbf{r} D(E, \mathbf{r}) = \sum_{ij} \rho_{ij}(E) S_{ij},$$

where $S_{ij} = \int \phi_i(\mathbf{r})\phi_j(\mathbf{r}) d\mathbf{r}$ is the overlap matrix. Introducing $M_i(E) = \sum_j \rho_{ij}(E) S_{ij}$, we may write this as

$$D(E) = \sum_i M_i(E),$$

where thus $M_i(E)$ can be seen as the contribution to the DDOS from orbital i. $M_i(E)$ is a spectral *MullikenPopulation*, with

$$M_i = \int M_i(E) f\left(\frac{E-\mu}{k_B T}\right) dE.$$

$M_i(E)$ can be summed over orbitals of a particular angular momentum and/or over one or several atoms to give the projected device density of states (PDDOS). This can also be plotted in QuantumATK. The projection is specified via the `projection_list` keyword to the `evaluate()` method (see above).

DeviceHuckelCalculator

class DeviceHuckelCalculator(*electrode_calculators=None*, *basis_set=None*, *pair_potentials=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *device_algorithm_parameters=None*, *weighting_scheme=None*, *poisson_solver=None*, *contour_parameters=None*, *electrode_voltages=None*, *electrode_temperatures=None*, *checkpoint_handler=None*, *spin_polarization=None*, *dynamical_matrix_parameters=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*)

Class for representing calculations using the extended Huckel Model for DeviceConfigurations.

Parameters

•**electrode_calculators** (tuple of *HuckelCalculator*) – A tuple of *HuckelCalculator* objects containing a calculator for each electrode.

•**basis_set** (list of *HuckelBasisParameters*) – An object describing the basis set used for the Extended-Huckel calculation. **Default:** `HoffmannHuckelParameters.All`

•**pair_potentials** (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations. **Default:** No pair potential

•**numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 100),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1.0e-6,
    interaction_max_range=10*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(300.0*Kelvin))
```

•**iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self- consistent Huckel calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`.

•**device_algorithm_parameters** (*DeviceAlgorithmParameters*) – The *DeviceAlgorithmParameters* used for the device simulation. **Default:**

```
DeviceAlgorithmParameters(
    initial_density_type=NeutralAtom(
        electrode_constraint_length=10.0*Angstrom),
    electrode_constraint=ElectrodeConstraint.Off,
    self_energy_calculator_real=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1.0e-13,
        maximum_iteration=400),
    self_energy_calculator_complex=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1.0e-13,
        maximum_iteration=400),
    non_equilibrium_method=GreensFunction(),
    equilibrium_method=GreensFunction(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    enforce_different_electrodes=False)
```

Note that the electrode constraint for a SurfaceConfiguration will be set to `ElectrodeConstraint.DensityMatrix(electrode_constraint_length=10.0*Angstrom`.

•**weighting_scheme** (*HoffmannWeighting* | *WolfsbergWeighting*) – The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian. **Default:** *WolfsbergWeighting*

•**poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourier2DSolver* for a *DeviceConfiguration* without any metallic or dielectric `SpatialRegion`. For others: *MultigridSolver*.

The default boundary conditions are `[PeriodicBoundaryCondition, PeriodicBoundaryCondition, DirichletBoundaryCondition]`.

- **contour_parameters** (*ContourParameters*) – The parameters used for the complex contour integration. **Default:**

```
ContourParameters(
    equilibrium_contour=SemiCircleContour(
        integral_lower_bound=1.5*Hartree,
        circle_eccentricity=0.3,
        logarithmic_bunching=0.3,
        circle_points=30,
        fermi_line_points=10,
        fermi_function_poles=8),
    non_equilibrium_contour=RealAxisContour(
        real_axis_point_density=0.001*Hartree,
        real_axis_infinitesimal=0.001*Hartree,
        real_axis_kbt_padding_factor=5.0),
    method=DoubleContour())
```

- **electrode_voltages** (*PhysicalQuantity* of type electrical potential) – The voltages applied to the two electrodes. **Default:** `(0.0, 0.0) * Volt`
- **electrode_temperatures** (Sequence of *PhysicalQuantity* of type temperature) – The temperatures used in the Fermi-Dirac distribution of the electrodes. They represent the physical temperatures used for the integration of the transmission spectra. **Default:** `(300, 300) * Kelvin`
- **checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.
- **spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`
- **dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.
- **hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.
- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:** `ParallelParameters(processes_per_saddle_search=1)`

basisSet()

Returns
The basis set.

Return type
list of *HuckelBasisParameters*

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

contourParameters()

Returns
The contour integral parameters.

deviceAlgorithmParameters()

Returns
The device algorithm parameters.

Return type
DeviceAlgorithmParameters

dynamicalMatrixParameters()

This method is deprecated.

electrodeCalculators()

Returns
The electrode calculator of each electrode.

Return type
list of *SemiEmpiricalCalculator*

electrodeTemperatures()

Returns
The electrode temperatures.

Return type
Sequence of *PhysicalQuantity* of type temperature.

electrodeVoltages()

Returns
The electrode voltages as *PhysicalQuantity* of length 2.

Return type
PhysicalQuantity of type electrical potential

hamiltonianDerivativesParameters()

This method is deprecated.

hamiltonianParametrization()

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
SemiEmpiricalHamiltonianParametrization

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfSpins()

Returns
The number of spins.

Return type
int

numericalAccuracyParameters()

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setCheckpointHandler(*checkpoint_handler*)
Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
Unpolarized | Polarized | Noncollinear | SpinOrbit

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

weightingScheme()

Returns
The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian.

Return type
HoffmannWeighting | *WolfsbergWeighting*

Attention: The *DeviceHuckelCalculator* is being deprecated. Use the *DeviceSemiEmpiricalCalculator* with the *HuckelHamiltonianParametrization* instead.

Usage Examples

Define a *DeviceHuckelCalculator* with user defined *NumericalAccuracyParameters*, *MultigridSolver*, and *DoubleContourIntegralParameters*

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=10.*Units.Hartree,
    k_point_sampling=(3,2,100),
    interaction_max_range=10.*Angstrom,
    )

electrode_poisson_solver = MultigridSolver(
    boundary_conditions=[PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition]
    )

poisson_solver = MultigridSolver(
    boundary_conditions=[PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition,
                         DirichletBoundaryCondition]
    )
```

(continues on next page)

(continued from previous page)

```
electrode_calculator = HuckelCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=electrode_poisson_solver
    )

contour_parameters = DoubleContourIntegralParameters(
    integral_lower_bound=5.0*Units.Hartree
    )

device_calculator = DeviceHuckelCalculator(
    electrode_calculators=[electrode_calculator,electrode_calculator],
    contour_parameters=contour_parameters,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=poisson_solver,
    electrode_voltages=(0.2*Volt, -0.3*Volt)
    )
```

Perform a voltage sweep, and calculate I-V characteristics

```
calculator = DeviceHuckelCalculator()
device_configuration = DeviceConfiguration(...)

# Define voltages for voltage ramp, [0.0,0.1, ..., 1.0]*Volt
voltages = numpy.linspace(0.0,1.0,11)*Volt
for v in voltages:

    # Set the calculator on the configuration using the old calculation as starting␣
↪input.
    device_configuration.setCalculator(
        calculator(electrode_voltages=(0*Volt,v)),
        initial_state=device_configuration,
        )

    # Calculate the transmission
    t = TransmissionSpectrum(device_configuration)

    # Calculate the current.
    current = t.current()
    print t.bias(), t.current()
```

Perform a gate bias scan

```
calculator = DeviceHuckelCalculator()
device_configuration = DeviceConfiguration(...)
metal_region = BoxRegion(...)

# Define gate_voltages for scan, [0.0,0.1, ..., 1.0]*Volt
gate_voltage=numpy.linspace(0.0,1.0,11)*Volt
for v in gate_voltage:
    device_configuration.setMetallicRegions(
        [metallic_region(value = gate_voltage)]
```

(continues on next page)

(continued from previous page)

```
        )
    # Set the calculator on the configuration using the old calculation as starting␣
↪input.
    device_configuration.setCalculator(
        calculator(),
        initial_state=device_configuration,
    )

    # Calculate the transmission
    t = TransmissionSpectrum(device_configuration)

    # Calculate the conductance
    print t.bias(), t.conductance()
```

Notes

The parameters for the constructor of a *DeviceHuckelCalculator* object and the parameters of its electrode calculators must fulfill the following conditions: In case the user does not set an electrode parameter, QuantumATK will generate that parameter using the rules below.

- The *NumericalAccuracyParameters* must be the same for the electrodes and the device. The central region of the device does not use k-points in the C-direction and this parameter is only used for the electrodes. The electrodes need a very dense k-point sampling in the C direction.
- The `poisson_solver` must be set to either the *FastFourier2DSolver* (default, and normally recommended) or the *MultigridSolver* or *DirectSolver* in case electrostatic gates and/or dielectric regions are included. The same boundary conditions in the A and B directions must be used for the electrodes as for the device calculator. In the C directions the user setting is ignored and the program always uses `PeriodicBoundaryCondition` for the electrodes and `DirichletBoundaryCondition` for the device.
- The `electrode_voltages` give rise to a shift of the Fermi levels of the electrodes by $-eV_{\mathrm{bias}}$, where V_{bias} is the applied bias. Thus, a higher V_{bias} on the right electrode than the left gives rise to an electron flow from left to right, corresponding to an electrical current from right to left (the current will be negative in this case; see *TransmissionSpectrum*).

For the details of the extended-Hückel model, see the chapter on *Semi Empirical*.

DeviceLCAOCalculator

class DeviceLCAOCalculator(*electrode_calculators=None*, *basis_set=None*, *exchange_correlation=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *device_algorithm_parameters=None*, *poisson_solver=None*, *contour_parameters=None*, *electrode_voltages=None*, *electrode_temperatures=None*, *checkpoint_handler=None*, *dynamical_matrix_parameters=None*, *correction_extension=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*, *solvation_parameters=None*, *use_atom_pair_matrix=None*)

The constructor for the DeviceLCAOCalculator.

Parameters

- **electrode_calculators** (list of *LCAOCalculator*) – Two calculators, one for each electrode.

•**basis_set** (list | tuple | *BasisSet*) – An object describing the basis set used for the LCAO calculation. This keyword can be given in three ways.

1.As a single *BasisSet* object.

2.As a tuple (string, BasisSet). The string must match a tag on the atom in order to be applied.

3.As a sequence of entries matching either of the above formats

Default: `BasisGGASG15.Medium`

•**exchange_correlation** (*ExchangeCorrelation*) – The exchange correlation functional for this calculation. **Default:** `GGA.PBE`

•**numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The numerical accuracy parameters used for the DFT calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=75*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 100),
    radial_step_size=0.001*Bohr,
    density_cutoff=1e-6,
    interaction_max_range=20*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250*Hartree,
    occupation_method=FermiDirac(1000*Kelvin))
```

•**iteration_control_parameters** (*IterationControlParameters* | `NonSelfconsistent`) – The iteration control parameters used for the DFT calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:**

```
IterationControlParameters(
    tolerance=1e-4,
    max_steps=100,
    algorithm=PulayMixer(),
    damping_factor=0.1,
    number_of_history_steps=20,
    start_mixing_after_step=0,
    mixing_variable=HamiltonianVariable,
    linear_dependence_threshold=0.0,
    preconditioner=Preconditioner.Off)
```

•**device_algorithm_parameters** (*DeviceAlgorithmParameters*) – The DeviceAlgorithmParameters used for the device simulation **Default:**

```
DeviceAlgorithmParameters(
    initial_density_type=NeutralAtom(
        electrode_constraint_length=10.0*Angstrom),
    electrode_constraint=ElectrodeConstraint.Off,
    self_energy_calculator_real=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1e-13,
        maximum_iteration=400),
    self_energy_calculator_complex=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1e-13,
        maximum_iteration=400),
```

(continues on next page)

(continued from previous page)

```
    non_equilibrium_method=GreensFunction(),
    equilibrium_method=GreensFunction(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    enforce_different_electrodes=False)
```

Note that the electrode constraint for a SurfaceConfiguration will be set to `ElectrodeConstraint.DensityMatrix(electrode_constraint_length=10.0*Angstrom`.

- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourier2DSolver* for a *DeviceConfiguration* without any metallic or dielectric `SpatialRegion`. For others: *MultigridSolver*. The default boundary conditions are `[PeriodicBoundaryCondition, PeriodicBoundaryCondition, DirichletBoundaryCondition]`.

- **contour_parameters** (*ContourParameters*) – The parameters used for the complex contour integration. **Default:**

```
ContourParameters(
    equilibrium_contour=SemiCircleContour(
        integral_lower_bound=1.5*Hartree,
        circle_eccentricity=0.3,
        logarithmic_bunching=0.3,
        circle_points=30,
        fermi_line_points=10,
        fermi_function_poles=8),
    non_equilibrium_contour=RealAxisContour(
        real_axis_point_density=0.001*Hartree,
        real_axis_infinitesimal=0.001*Hartree,
        real_axis_kbt_padding_factor=5.0),
    method=DoubleContour())
```

- **electrode_voltages** (*PhysicalQuantity* of type electrical potential) – The voltages applied to the two electrodes. **Default:** `(0.0, 0.0) * Volt`.

- **electrode_temperatures** (Sequence of *PhysicalQuantity* of type temperature) – The temperatures used in the Fermi-Dirac distribution of the electrodes. They represent the physical temperatures used for the integration of the transmission spectra. **Default:** `(300, 300) * Kelvin`

- **checkpoint_handler** (*CheckpointHandler*) – The CheckpointHandler used for specifying the save-file and the time interval between saving the calculation during the SCF-loop. **Default:** A default *CheckpointHandler* object.

- **dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.

- **correction_extension** (*GrimmeDFTD2* | *GrimmeDFTD3*) – The correction extension to used, when calculating energy, forces and stress. **Default:** None.

- **hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.

- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:** `ParallelParameters(processes_per_saddle_search=1)`

- **solvation_parameters** (None | *CosmoSolvationParameters*) – Optional parameters for a COSMO solvation model. If this is specified, the COSMO solvation model contribution will be included in the SCF loop.

- **use_atom_pair_matrix** (*bool*) – If this flag is set to true, the density matrix/hamiltonian and overlap matrices will be stored in an atom-pair matrix format. **Default:** `False`

basisSet()

Returns
The basis set.

Return type
list

checkpointHandler()

Returns
The checkpoint handler.

Return type
CheckpointHandler

contourParameters()

Returns
The contour parameters.

Return type
ContourParameters

correctionExtension()

Returns
The correction extension or None if not set.

Return type
GrimmeDFTD2 | *GrimmeDFTD3* | None

deviceAlgorithmParameters()

Returns
The device algorithm parameters.

Return type
DeviceAlgorithmParameters

dynamicalMatrixParameters()
Return the DynamicalMatrixParameters. Deprecated.

Returns
None since the class DynamicalMatrixParameters is deprecated.

Return type
None

electrodeCalculators()

Returns
The electrode calculators.

Return type
list of *LCAOCalculator*

electrodeTemperatures()

Returns
The electrode temperatures.

Return type
Sequence of *PhysicalQuantity* of type temperature.

`electrodeVoltages()`

Returns
The electrode voltages.

Return type
PhysicalQuantity of type electrical potential

`exchangeCorrelation()`

Returns
The exchange-correlation.

Return type
ExchangeCorrelation

`hamiltonianDerivativesParameters()`
Return the HamiltonianDerivativesParameters. Deprecated.

Returns
None since the class HamiltonianDerivativesParameters is deprecated.

Return type
None

`isConverged()`

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

`iterationControlParameters()`

Returns
The iteration control parameters.

Return type
IterationControlParameters

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`numericalAccuracyParameters()`

Returns
The numerical accuracy parameters.

Return type
NumericalAccuracyParameters

parallelParameters()

Returns
The parallel parameters object.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver set on the calculator.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setBasisSet(*basis_set*)

Set the basis set.

Parameters
basis_set (*list*) – The basis set to use. The basis set is given as a sequence of *BasisSet* objects for each element in the configuration.

setCheckpointHandler(*checkpoint_handler*)

Set the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The checkpoint handler to use.

setCorrectionExtension(*correction_extension*)

Set the iteration correction extension.

Parameters
correction_extension (*GrimmeDFTD2* | *GrimmeDFTD3* | None) – The correction extension to use.

setExchangeCorrelation(*exchange_correlation*)

Set the exchange-correlation.

Parameters
exchange_correlation (*ExchangeCorrelation*) – The exchange-correlation to use.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The iteration control parameters to use.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The numerical accuracy parameters to use.

setParallelParameters(*parallel_parameters*)

Method for setting the parallel parameters.

Parameters
parallel_parameters (*ParallelParameters* | None) – The parallel parameters to set. If None a default version of the parameters is used.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver to set on the calculator.

setSolvationParameters(*solvation_parameters*)

Set the solvation parameters.

Parameters
solvation_parameters (*CosmoSolvationParameters*) – The solvent parameters to use.

solvationParameters()

Returns
The solvation parameters.

Return type
CosmoSolvationParameters

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible @private

useAtomPairMatrix()

Returns
If true, the density matrix/hamiltonian and overlap matrices will be stored in an atom-pair matrix format.

Return type
bool

static validateExchangeCorrelation(*exchange_correlation*)

Validate the exchange-correlation.

Parameters
exchange_correlation (*ExchangeCorrelation*) – The exchange-correlation to use.

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Usage Examples

Define a *DeviceLCAOCalculator* with user defined *NumericalAccuracyParameters*, *MultigridSolver*, and *DoubleContourIntegralParameters*:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=10.Hartree,
    k_point_sampling=MonkhorstPackGrid(3,2,100),
    interaction_max_range=10.*Angstrom,
    )

electrode_poisson_solver = MultigridSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

poisson_solver = MultigridSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [DirichletBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

electrode_calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=electrode_poisson_solver
    )

contour_parameters = DoubleContourIntegralParameters(
    integral_lower_bound=5.0*Hartree
    )

device_calculator = DeviceLCAOCalculator(
    electrode_calculators=[electrode_calculator,electrode_calculator],
    contour_parameters=contour_parameters,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=poisson_solver,
    electrode_voltages=(0.2*Volt, -0.3*Volt)
    )
```

Perform a voltage sweep and calculate I–V characteristics:

```
calculator = DeviceLCAOCalculator()
device_configuration = DeviceConfiguration(...)

# Define voltages for voltage ramp, [0.0,0.1, ..., 1.0]*Volt
voltages = numpy.linspace(0.0,1.0,11)*Volt
for v in voltages:
    # Set the calculator on the configuration using the old calculation as starting␣
↪input.
    device_configuration.setCalculator(
        calculator(electrode_voltages=(0*Volt,v)),
        initial_state=device_configuration,
```

(continues on next page)

(continued from previous page)

```
        )

    # Calculate the transmission
    t = TransmissionSpectrum(device_configuration)

    # Calculate the current.
    current = t.current()
    print t.bias(), t.current()
```

Perform a gate bias scan:

```
calculator = DeviceLCAOCalculator()
device_configuration = DeviceConfiguration(...)
metal_region = BoxRegion(...)

# Define gate_voltages for scan, [0.0,0.1, ..., 1.0]*Volt
gate_voltage=numpy.linspace(0.0,1.0,11)*Volt
for v in gate_voltage:
    device_configuration.setMetallicRegions(
        [metallic_region(value = gate_voltage)]
        )

    # Set the calculator on the configuration using the old calculation as starting␣
↪input.
    device_configuration.setCalculator(
        calculator(),
        initial_state=device_configuration,
        )

    # Calculate the transmission
    t = TransmissionSpectrum(device_configuration)

    # Calculate the conductance
    print t.bias(), t.conductance()
```

Notes

The parameters for the constructor of a *DeviceLCAOCalculator* object and the parameters of its electrode calculators must fulfill the conditions given below. In case the user does not set an electrode parameter, QuantumATK will automatically generate that parameter using these rules:

- The *NumericalAccuracyParameters* must be the same for the electrodes and the device. The central region of the device does not use k-points in the C direction and this parameter is only used for the electrodes. The electrodes need a very dense k-point sampling in the C direction. Note that if `transverse_electrode_repetitions` have been specified for the *DeviceConfiguration*, the k-point density of the minimal electrodes along the A and B directions must be the same as the k-point density in the central region along A and B.
- The `poisson_solver` parameter must be set to either the *FastFourier2DSolver* (default, and usually recommended) or the *MultigridSolver* or *DirectSolver* in case electrostatic gates and/or dielectric regions are included. The same boundary conditions in the A and B directions must be used for the electrodes as for the device calculator. In the C direction, the user setting is ignored and the program always uses *PeriodicBoundaryCondition* for the electrodes and *DirichletBoundaryCondition* for the device.

- The `electrode_voltages` parameter gives rise to a shift of the Fermi levels of the electrodes by $-e \cdot V_{\mathrm{bias}}$, where V_{bias} is the applied bias. Thus, a higher V_{bias} on the right electrode than the left gives rise to an electron flow from left to right, corresponding to an electrical current from right to left (the current will be negative in this case; see *TransmissionSpectrum*).

DevicePerformanceProfile

class DevicePerformanceProfile(*configuration*, *equilibrium_methods=None*, *non_equilibrium_methods=None*)

Class for performing timing and memory profiles of the different methods available for calculating the Greens function and lesser Greens function.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with an attached calculator to profile.
- **equilibrium_methods** (GreensFunction | SparseGreensFunction | sequence of (GreensFunction | SparseGreensFunction)) – The methods benchmarked for the equilibrium calculation. If no methods should be benchmarked for the equilibrium calculation, an empty list can be specified. **Default:**: (GreensFunction, SparseGreensFunction)
- **non_equilibrium_methods** – The methods benchmarked for the non-equilibrium calculation. If no methods should be benchmarked for the non-equilibrium calculation, an empty list can be specified. **Default:**: (GreensFunction, SparseGreensFunction)

equilibriumMethods()

Returns
The equilibrium methods profiled.

Return type
tuple of (GreensFunction | SparseGreensFunction)

memoryPeaks()

Get the peak memory usage across all processes for the different Greens function methods. Every process returns the same global peak memory usage.

Returns
The peak memory usages in MB.

Return type
dict

memoryPeaksSingleProcess()

Get the peak memory usage for the different Greens function methods. Each process returns its own local peak memory usage.

Returns
The peak memory usages in MB.

Return type
dict

memoryProfiles()

Get the memory usage over time for the different Greens function methods. Each process returns its own local memory usage.

Returns
The memory profiles in MB.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print out the profiling report.

Parameters
stream (*file*) – The stream to write to. This stream must support strings being written to it using write. **Default:** sys.stdout

nonEquilibriumMethods()

Returns
The non-equilibrium methods profiled.

Return type
tuple of (GreensFunction | SparseGreensFunction)

timings()

Get the timing results for the different Greens function methods.

Returns
The timing results in seconds.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Notes

This profiler helps to pick the optimal values for the arguments equilibrium_method and non_equilibrium_method in *DeviceAlgorithmParameters*. These arguments select the algorithms used to calculate the Greens function and lesser Greens function.

For both equilibrium and non-equilibrium contour points the following methods are profiled:

- *GreensFunction*
- *SparseGreensFunction*

For each method the elapsed time and memory usage are reported.

Note: The profile is generated by spawning a child process. Not all clusters support this.

Usage Example

Investigate the optimal methods to calculate the (lesser) Greens function for a carbon chain with the following script.

```
# -----------------------------------------------------------
# Left electrode
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
```

(continues on next page)

(continued from previous page)

```
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.8]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
left_electrode_elements = [Carbon, Carbon]
# Define coordinates
left_electrode_coordinates = [[ 3.  ,  3.  ,  1.45],
                              [ 3.  ,  3.  ,  4.35]]*Angstrom
# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )
# -----------------------------------------------------------
# Right electrode
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.8]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
right_electrode_elements = [Carbon, Carbon]
# Define coordinates
right_electrode_coordinates = [[ 3.  ,  3.  ,  1.45],
                               [ 3.  ,  3.  ,  4.35]]*Angstrom
# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )
# -----------------------------------------------------------
# Central region
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 71.5]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
central_region_elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon,
                           Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon,
                           Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon]
# Define coordinates
central_region_coordinates = [[  3.  ,   3.  ,   1.45],
                              [  3.  ,   3.  ,   4.35],
                              [  3.  ,   3.  ,   7.25],
```

(continues on next page)

(continued from previous page)

```
                                [  3.  ,   3.  ,  10.15],
                                [  3.  ,   3.  ,  13.05],
                                [  3.  ,   3.  ,  15.95],
                                [  3.  ,   3.  ,  18.85],
                                [  3.  ,   3.  ,  21.75],
                                [  3.  ,   3.  ,  24.65],
                                [  3.  ,   3.  ,  27.55],
                                [  3.  ,   3.  ,  30.45],
                                [  3.  ,   3.  ,  33.35],
                                [  3.  ,   3.  ,  38.15],
                                [  3.  ,   3.  ,  41.05],
                                [  3.  ,   3.  ,  43.95],
                                [  3.  ,   3.  ,  46.85],
                                [  3.  ,   3.  ,  49.75],
                                [  3.  ,   3.  ,  52.65],
                                [  3.  ,   3.  ,  55.55],
                                [  3.  ,   3.  ,  58.45],
                                [  3.  ,   3.  ,  61.35],
                                [  3.  ,   3.  ,  64.25],
                                [  3.  ,   3.  ,  67.15],
                                [  3.  ,   3.  ,  70.05]]*Angstrom
# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )
device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode]
    )
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#-------------------------------------
# Basis Set
#-------------------------------------
basis_set = [
    LDABasis.Carbon_SingleZeta,
    ]

#-------------------------------------
# Exchange-Correlation
#-------------------------------------
exchange_correlation = NCLDA.PZ

#-------------------------------------
# Poisson Solver Settings
#-------------------------------------
left_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                         [PeriodicBoundaryCondition,PeriodicBoundaryCondition],
```

(continues on next page)

(continued from previous page)

```
                          [PeriodicBoundaryCondition,PeriodicBoundaryCondition]]
    )
right_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                          [PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                          [PeriodicBoundaryCondition,PeriodicBoundaryCondition]]
    )

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )

#----------------------------------------
# Electrode Calculators
#----------------------------------------
left_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    poisson_solver=left_electrode_poisson_solver,
    )
right_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    poisson_solver=right_electrode_poisson_solver,
    )

#----------------------------------------
# Device Calculator
#----------------------------------------
calculator = DeviceLCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    iteration_control_parameters = iteration_control_parameters,
    electrode_calculators=
        [left_electrode_calculator, right_electrode_calculator],
    )

device_configuration.setCalculator(calculator)

# Create the performance profile and print a report.
device_performance_profile = DevicePerformanceProfile(device_configuration)
nlprint(device_performance_profile)
```

`device_performance_profile.py`

Here is an example of the output you might get by running with 2 MPI processes:

```
+-------------------------------------------------------------------------------+
| Device Performance Profile (2 processes)                                      |
+-------------------------------------------------------------------------------+
| Contour point timing (s):                                                     |
|                                  EQ        NEQ                                |
```

(continues on next page)

(continued from previous page)

```
| GreensFunction                  0.01       0.01                                |
| SparseGreensFunction            0.03       0.02                                |
|                                                                                |
| Fastest EQ method (by 5.1 times): GreensFunction                               |
| Fastest NEQ method (by 2.9 times): GreensFunction                              |
+--------------------------------------------------------------------------------+
| Peak memory usage/process (MB):                                                |
|                                  EQ        NEQ                                 |
| GreensFunction                292.73     295.70                                |
| SparseGreensFunction          295.27     301.52                                |
|                                                                                |
| Most memory-efficient EQ method (by 1.0 times): GreensFunction                 |
| Most memory-efficient NEQ method (by 1.0 times): GreensFunction                |
+--------------------------------------------------------------------------------+
```

We can see that, in this case, *GreensFunction* is faster for calculating both equilibrium and non-equilibrium contour points. In terms of memory efficienty, the two methods are almost identical.

It is important to note that the timings given are for a single contour point run in parallel across all MPI processes. In a real device calculation many contour points are used, and the best performance is obtained when running with 1 process per contour point. It is therefore appropriate to run *DevicePerformanceProfile* with the same number of processes that will be specified in `processes_per_contour_point`.

DeviceSemiEmpiricalCalculator

class DeviceSemiEmpiricalCalculator(*hamiltonian_parametrization=None*, *electrode_calculators=None*, *pair_potentials=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *device_algorithm_parameters=None*, *poisson_solver=None*, *contour_parameters=None*, *electrode_voltages=None*, *electrode_temperatures=None*, *checkpoint_handler=None*, *spin_polarization=None*, *parallel_parameters=None*)

Class for representing calculations using semi-empirical models for DeviceConfigurations.

Parameters

- **hamiltonian_parametrization** (*HuckelHamiltonianParametrization* | *SlaterKosterHamiltonianParametrization* | *NRLHamiltonianParametrization*) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.
- **electrode_calculators** (tuple of *SemiEmpiricalCalculator*) – A tuple of *SemiEmpiricalCalculator* objects containing a calculator for each electrode.
- **pair_potentials** (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations. **Default:** No pair potential
- **numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent semi-empirical calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 100),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1.0e-6,
```

(continues on next page)

(continued from previous page)

```
    interaction_max_range=10*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(300.0*Kelvin))
```

- **iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self- consistent semi-empirical calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`.

- **device_algorithm_parameters** (*DeviceAlgorithmParameters*) – The *DeviceAlgorithmParameters* used for the device simulation. **Default:**

```
DeviceAlgorithmParameters(
    initial_density_type=NeutralAtom(
        electrode_constraint_length=10.0*Angstrom),
    electrode_constraint=ElectrodeConstraint.Off,
    self_energy_calculator_real=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1.0e-13,
        maximum_iteration=400),
    self_energy_calculator_complex=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1.0e-13,
        maximum_iteration=400),
    non_equilibrium_method=GreensFunction(),
    equilibrium_method=GreensFunction(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    enforce_different_electrodes=False)
```

Note that the electrode constraint for a SurfaceConfiguration will be set to `ElectrodeConstraint.DensityMatrix(electrode_constraint_length=10.0*Angstrom`.

- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourier2DSolver* for a *DeviceConfiguration* without any metallic or dielectric `SpatialRegion`. For others: *MultigridSolver*. The default boundary conditions are `[PeriodicBoundaryCondition, PeriodicBoundaryCondition, DirichletBoundaryCondition]`.

- **contour_parameters** (*ContourParameters*) – The parameters used for the complex contour integration. **Default:**

```
ContourParameters(
    equilibrium_contour=SemiCircleContour(
        integral_lower_bound=1.5*Hartree,
        circle_eccentricity=0.3,
        logarithmic_bunching=0.3,
        circle_points=30,
        fermi_line_points=10,
        fermi_function_poles=8),
    non_equilibrium_contour=RealAxisContour(
        real_axis_point_density=0.001*Hartree,
        real_axis_infinitesimal=0.001*Hartree,
        real_axis_kbt_padding_factor=5.0),
```

(continues on next page)

(continued from previous page)

```
    method=DoubleContour())
```

- **electrode_voltages** (*PhysicalQuantity* of type electrical potential) – The voltages applied to the two electrodes. **Default:** `(0.0, 0.0) * Volt`
- **electrode_temperatures** (Sequence of *PhysicalQuantity* of type temperature) – The temperatures used in the Fermi-Dirac distribution of the electrodes. They represent the physical temperatures used for the integration of the transmission spectra. **Default:** `(300, 300) * Kelvin`
- **checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.
- **spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`
- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:** `ParallelParameters(processes_per_saddle_search=1)`

basisSet()

Returns
The basis set associated with the Hamiltonian parametrization.

Return type
BasisSet

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

contourParameters()

Returns
The contour integral parameters.

deviceAlgorithmParameters()

Returns
The device algorithm parameters.

Return type
DeviceAlgorithmParameters

electrodeCalculators()

Returns
The electrode calculator of each electrode.

Return type
list of *SemiEmpiricalCalculator*

electrodeTemperatures()

Returns
The electrode temperatures.

Return type
Sequence of *PhysicalQuantity* of type temperature.

`electrodeVoltages()`

Returns
The electrode voltages as *PhysicalQuantity* of length 2.

Return type
PhysicalQuantity of type electrical potential

`hamiltonianParametrization()`

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
`SemiEmpiricalHamiltonianParametrization`

`isConverged()`

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

`iterationControlParameters()`

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is `NonSelfconsistent`.

Return type
IterationControlParameters

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`numberOfSpins()`

Returns
The number of spins.

Return type
int

`numericalAccuracyParameters()`

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setCheckpointHandler(*checkpoint_handler*)

Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
Unpolarized | Polarized | Noncollinear | SpinOrbit

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Usage Examples

Define a *DeviceSemiEmpiricalCalculator* with user defined *NumericalAccuracyParameters*, *MultigridSolver*, *DoubleContourIntegralParameters*, and a default *HuckelHamiltonianParametrization*.

```
# Define the numerical accuracy parameters and Poisson solvers.
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=10. * Hartree,
    k_point_sampling=(3, 2, 100),
    interaction_max_range=10. * Angstrom,
```

(continues on next page)

(continued from previous page)

```
    )

electrode_poisson_solver = MultigridSolver(
    boundary_conditions=[PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition]
    )

poisson_solver = MultigridSolver(
    boundary_conditions=[PeriodicBoundaryCondition,
                         PeriodicBoundaryCondition,
                         DirichletBoundaryCondition]
    )

# Set up the default Huckel-type Hamiltonian parametrization
# and the electrode calculator.
hamiltonian_parametrization = HuckelHamiltonianParametrization()

electrode_calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=electrode_poisson_solver
    )

# Set up the contour parameters and the device calculator with
# non-zero electrode voltages.
contour_parameters = DoubleContourIntegralParameters(
    integral_lower_bound=5.0 * Hartree
    )

device_calculator = DeviceSemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    electrode_calculators=[electrode_calculator, electrode_calculator],
    contour_parameters=contour_parameters,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    poisson_solver=poisson_solver,
    electrode_voltages=(0.2 * Volt, -0.3 * Volt)
    )
```

Notes

The parameters for the constructor of a *DeviceSemiEmpiricalCalculator* object and the parameters of its electrode calculators must fulfill the following conditions: In case the user does not set an electrode parameter, QuantumATK will generate that parameter using the rules below.

- The *NumericalAccuracyParameters* must be the same for the electrodes and the device. The central region of the device does not use k-points in the C direction and this parameter is only used for the electrodes. The electrodes need a very dense k-point sampling in the C direction. Note that if `transverse_electrode_repetitions` have been specified for the *DeviceConfiguration*, the k-point density of the minimal electrodes along the A and B directions must be the same as the k-point density in the central region along A and B.
- The `poisson_solver` must be set to either the *FastFourier2DSolver* (default, and normally recommended) or the

MultigridSolver or *DirectSolver* in case electrostatic gates and/or dielectric regions are included. The same boundary conditions in the A and B directions must be used for the electrodes as for the device calculator. In the C directions the user setting is ignored and the program always uses `PeriodicBoundaryCondition` for the electrodes and `DirichletBoundaryCondition` for the device.

•The `electrode_voltages` give rise to a shift of the Fermi levels of the electrodes by $-eV_{\mathrm{bias}}$, where V_{bias} is the applied bias. Thus, a higher V_{bias} on the right electrode than the left gives rise to an electron flow from left to right, corresponding to an electrical current from right to left (the current will be negative in this case; see *TransmissionSpectrum*).

For the usage and details of the different Hamiltonian parametrizations, see *HuckelHamiltonianParametrization*, *SlaterKosterHamiltonianParametrization* and *NRLHamiltonianParametrization*.

DeviceSlaterKosterCalculator

class DeviceSlaterKosterCalculator(*electrode_calculators=None*, *basis_set=None*, *pair_potentials=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *device_algorithm_parameters=None*, *poisson_solver=None*, *contour_parameters=None*, *electrode_voltages=None*, *electrode_temperatures=None*, *checkpoint_handler=None*, *spin_polarization=None*, *dynamical_matrix_parameters=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*)

The constructor for the DeviceSlaterKosterCalculator.

Parameters

•**electrode_calculators** (list of *SlaterKosterCalculator*) – A list of *SlaterKosterCalculator* objects containing a calculator for each electrode.

•**basis_set** (*SlaterKosterTable* | *DFTBDirectory* | *HotbitDirectory*) – An object describing the basis set used for the SlaterKoster calculation.

•**pair_potentials** (*DFTBDirectory* | *HotbitDirectory* | list of *PairPotential*) – The repulsive pair potentials used for total energy and force calculations **Default:** No pair potential

•**numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent SlaterKoster calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 100),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1e-6,
    interaction_max_range=10*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250*Hartree,
    occupation_method=FermiDirac(300*Kelvin))
```

•**iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self-consistent SlaterKoster calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`

•**device_algorithm_parameters** (*DeviceAlgorithmParameters*) – The *DeviceAlgorithmParameters* used for the device simulation **Default:**

```
DeviceAlgorithmParameters(
    initial_density_type=NeutralAtom(
        electrode_constraint_length=10.*Angstrom),
    electrode_constraint=ElectrodeConstraint.Off,
    self_energy_calculator_real=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1e-13,
        maximum_iteration=400),
    self_energy_calculator_complex=RecursionSelfEnergy(
        storage_strategy=SaveInMemory(),
        tolerance=1e-13,
        maximum_iteration=400),
    non_equilibrium_method=GreensFunction(),
    equilibrium_method=GreensFunction(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    enforce_different_electrodes=False)
```

Note that the electrode constraint for a SurfaceConfiguration will be set to `ElectrodeConstraint.DensityMatrix(electrode_constraint_length=10.0*Angstrom`.

- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourier2DSolver* for a *DeviceConfiguration* without any metallic or dielectric `SpatialRegion`. For others: *MultigridSolver*. The default boundary conditions are `[PeriodicBoundaryCondition, PeriodicBoundaryCondition, DirichletBoundaryCondition]`.

- **contour_parameters** (*ContourParameters*) – The parameters used for the complex contour integration. **Default:**

```
ContourParameters(
    equilibrium_contour=SemiCircleContour(
        integral_lower_bound=1.5*Hartree,
        circle_eccentricity=0.3,
        logarithmic_bunching=0.3,
        circle_points=30,
        fermi_line_points=10,
        fermi_function_poles=8),
    non_equilibrium_contour=RealAxisContour(
        real_axis_point_density=0.001*Hartree,
        real_axis_infinitesimal=0.001*Hartree,
        real_axis_kbt_padding_factor=5.0),
    method=DoubleContour())
```

- **electrode_voltages** (*PhysicalQuantity* of type electrical potential) – The voltages applied to the electrodes as a *PhysicalQuantity* of length 2. **Default:** `(0.0, 0.0) * Volt`

- **electrode_temperatures** (Sequence of *PhysicalQuantity* of type temperature) – The temperatures used in the Fermi-Dirac distribution of the electrodes. They represent the physical temperatures used for the integration of the transmission spectra. **Default:** `(300, 300) * Kelvin`

- **checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.

- **spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`

- **dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.
- **hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.
- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:** `ParallelParameters(processes_per_saddle_search=1)`

basisSet()

Returns
The basis set associated with the Hamiltonian parametrization.

Return type
BasisSet

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

contourParameters()

Returns
The contour integral parameters.

deviceAlgorithmParameters()

Returns
The device algorithm parameters.

Return type
DeviceAlgorithmParameters

dynamicalMatrixParameters()
This method is deprecated.

electrodeCalculators()

Returns
The electrode calculator of each electrode.

Return type
list of *SemiEmpiricalCalculator*

electrodeTemperatures()

Returns
The electrode temperatures.

Return type
Sequence of *PhysicalQuantity* of type temperature.

electrodeVoltages()

Returns
The electrode voltages as *PhysicalQuantity* of length 2.

Return type
PhysicalQuantity of type electrical potential

hamiltonianDerivativesParameters()

This method is deprecated.

hamiltonianParametrization()

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
SemiEmpiricalHamiltonianParametrization

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfSpins()

Returns
The number of spins.

Return type
int

numericalAccuracyParameters()

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setBasisSet(*basis_set*)

Set the basis set.

Parameters
basis_set (*SlaterKosterTable* | *DFTBDirectory* | *HotbitDirectory*) – An object describing the basis set used for the calculation.

setCheckpointHandler(*checkpoint_handler*)

Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
Unpolarized | Polarized | Noncollinear | SpinOrbit

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Attention: The *DeviceSlaterKosterCalculator* is being deprecated. Use the *DeviceSemiEmpiricalCalculator* with the *SlaterKosterHamiltonianParametrization* instead.

Usage Examples

Define a *DeviceSlaterKosterCalculator* with a Vogl tight-binding basis (see also *SlaterKosterTable*)

```
calculator = Deviceslaterkostercalculator(
    basis_set=Vogl.Silicon_Basis,
    )
```

Restart a DeviceSlaterKosterCalculator calculation using the self-consistent state from a previous calculation

```
# Read in the DeviceConfiguration with the old SCF state
old_calculation=nlread("filename.nc", DeviceConfiguration)[0]

# Define the DeviceConfiguration with similar number of atoms
new_calculation=DeviceConfiguration(...)

# extract the old calculator
old_calculator = old_calculation.calculator()

# make a clone of the old calculator
new_calculator = old_calculator()

# Attach the calculator and use the old initial state
new_calculation.setCalculator(new_calculator, initial_state=old_calculation)
```

Notes

The parameters for the constructor of a DeviceSlaterKosterCalculator object and the parameters of its electrode calculators must fulfill the conditions below. If the user has not set an electrode parameter, QuantumATK will then generate the parameter using the rules below:

- The *NumericalAccuracyParameters* must be the same for the electrodes and the device. The central region of the device does not use k-points in the C-direction and this parameter is only used for the electrodes. The electrodes need a very dense k-point sampling in the C direction.
- The `poisson_solver` must be set to the *MultigridSolver* or the *DirectSolver* with the same boundary conditions in the A and B directions for the electrodes and device. In the C directions the user setting is ignored and the program always uses `PeriodicBoundaryCondition` for the electrodes and `DirichletBoundaryCondition` for the device.
- The `electrode_voltages` give rise to a shift of the Fermi levels of the electrodes by $-eV_{\mathrm{bias}}$, where V_{bias} is the applied bias. Thus, a positive V_{bias} on the right electrode, gives rise to an electron flow from left to right, corresponding to an electrical current from right to left.

For the details of the Slater-Koster model, see the chapter on *Semi Empirical*.

DiagonalizationSolver

class DiagonalizationSolver(*bands_above_fermi_level=None*, *processes_per_kpoint=None*, *optimize_for_speed_over_memory=None*)

Calculate the density matrix by direct diagonalization.

Parameters

- **bands_above_fermi_level** (int | `Automatic` | `All`) – The number of bands above the Fermi level. Must be either a non-negative integer, `All` or `Automatic`. When set to `All` the total number of bands will be equal to the total number of basis functions. When set to `Automatic` an adaptive method will be used. For most systems, `Automatic` provides a major speed-up as compared to `All`. If the number of states above the Fermi level changes significantly from one diagonalization step to another, the adapative method may lead to slowing down of the calculation. **Default:** `Automatic`
- **processes_per_kpoint** (*int*) – The number of processes to use per kpoint. Must be a positive integer. **Default:** The number will be determined automatically from the total number of k-points and processes such as to keep the number as small as possible. One may set this number manually in order to reduce the memory requirements for each process.
- **optimize_for_speed_over_memory** (*bool*) – Whether to optimize the diagonalization for speed over memory. If True, the solver will determine the most optimal algorithm in terms of speed. Note that optimizing for speed can increase the memory usage by up to 30 %. Note also that this flag only has an effect if `processes_per_kpoint=1`, or the automatic value for `processes_per_kpoint=1` turns out to be 1. It is intended to be used for medium-sized to large systems with many k-points. This option has no effect for gpu-accelerated diagonalization. **Default:** False

bandsAboveFermiLevel()

Returns
The initial guess for the number of bands above the Fermi level.

Return type
int | `All`

optimizeForSpeedOverMemory()

Returns
Whether the diagonalization should be optimized for speed over memory.

Return type
bool

processesPerKpoint()

Returns
Number of processes per kpoint.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Define a density matrix method that uses the *DiagonalizationSolver*:

```
algorithm_parameters = AlgorithmParameters(
    density_matrix_method = DiagonalizationSolver())
```

Notes

Diagonalization algorithm

The library used for diagonalization of the LCAO Hamiltonian depends on the value of the parameter `processes_per_kpoint`:

- LAPACK is used when a single processor is used for each k-point, i.e. `processes_per_kpoint=1`.
- ELPA[1] is used when more than one processor is used for each k-point, e.g. `processes_per_kpoint=2`.

GPU acceleration

When GPU acceleration is enabled (see *GPU acceleration of proprietary feature calculations*), *cuSolverDN <https://docs.nvidia.com/cuda/cusolver/index.html> ` is used in the case of single process per k-point, and `ELPA <http://elpa.mpcdf.mpg.de/>* with GPU acceleration enabled in the case of multiple processes per k-point.

Number of bands included in the diagonalization

When the parameter `bands_above_fermi_level` is set to `Automatic` (default), the diagonalization solver uses an adaptive method. The method determines the number of occupied bands to include when diagonalizing the Hamiltonian after each diagonalization step. Then, it increases the number of occupied states by a large enough margin such that the next diagonalization step is guaranteed to include all occupied bands.

Attention: If the number of bands above the Fermi level is too low such that not all occupied states are included, the diagonalization is repeated with an increased number of bands. This can slow down the calculation, and therefore setting `bands_above_fermi_level` manually is not recommended.

The behavior of `bands_above_fermi_level` with regard to the spin type UNPOLARIZED, POLARIZED, and NON-COLLINEAR is described in *Notes* of the *Bandstructure* analysis object.

[1] A.ăMarek, V.ăBlum, R.ăJohanni, V.ăHavu, B.ăLang, T.ăAuckenthaler, A.ăHeinecke, H.-J. Bungartz, and H.ăLederer. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter*, 26(21):213201, 2014. doi:10.1088/0953-8984/26/21/213201.

DielectricTensor

class DielectricTensor(*configuration*, *dynamical_matrix*, *optical_spectrum*, *born_effective_charge*, *phonon_modes=None*, *q_point=None*, *charge_sum_rule_method=None*, *polarity_tolerance=None*, *broadening=None*)

Constructor for the static dielectric tensor object.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the dielectric tensor.
- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to calculate the phonon modes from (incl. polar phonon splitting).
- **optical_spectrum** (*OpticalSpectrum*) – The optical spectrum object from which to get the electronic contribution to the dielectric tensor.
- **born_effective_charge** (*BornEffectiveCharge*) – The born effective charge object calculated by the user. This is used for the ionic contribution to the dielectric tensor.
- **phonon_modes** (list with positive ints | `All`) – The phonon modes to include. **Default:** `All` (All phonon modes are included).
- **q_point** (*list of 3 floats*) – The fractional q-point to use for phonon modes and polar correction. **Default:** `[0.0, 0.0, 0.0]`
- **charge_sum_rule_method** (`None`|`DistributeEvenly`|`DistributeRelative`.) – Keyword specifying if a charge sum-rule should be applied to the Born effective charges. Can be `None` for no sum-rule application. Alternatively the charge error is either distributed evenly (`DistributeEvenly`) or relative (`DistributeRelative`) to the Born effective charges on the atoms. **Default:** `None`
- **polarity_tolerance** (*float*) – Unit-less tolerance used to control when modes should be considered non-polar. This is helpful to avoid numerical errors for acoustic modes with vanishing frequency. **Default:** `1e-6`
- **broadening** (*PhysicalQuantity* of type energy) – The broadening parameter used for the spectra. **Default:** `0.25 * meV (~2 cm1)`

absorption(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The absorption tensor for the contribution. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
ndarray.

broadening()

Returns
The broadening parameter used for the spectra.

Return type
PhysicalQuantity of type energy.

dielectricTensor(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution (*Sum | Electronic | Ionic*) – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The real part of the dielectric constant tensor. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

electronicDielectricTensor()

Returns
The electronic dielectric constant tensor - i.e. excluding the ionic contribution. Dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

energies()

Returns
The energies used for the spectrum.

Return type
PhysicalQuantity of type energy.

evaluate()

Returns
The static dielectric constant tensor - i.e. including the ionic contribution. Dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

evaluateImaginaryPolarizability()

Function for evaluating the imaginary part of the polarizability.

Returns
A list of imaginary parts of the polarizability tensors [electronic, static, ionic].

Return type
PhysicalQuantity with the unit `Hartree**3 * vacuum_permitivity`

evaluatePolarizability()

Function for evaluating the real part of the polarizability.

Returns
A list of real parts of the polarizability tensors [electronic, static, ionic].

Return type
PhysicalQuantity with the unit `Hartree**3 * vacuum_permitivity`

extinctionCoefficient(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution (*Sum | Electronic | Ionic*) – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The extinction coefficient including the ionic contribution. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

imaginaryDielectricTensor(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The imaginary part of the dielectric constant tensor. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

imaginaryElectronicDielectricTensor()

Returns
The electronic dielectric constant tensor - i.e. excluding the ionic contribution. Dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

imaginaryIonicDielectricTensor()

Returns
The ionic part of the static dielectric constant tensor. Dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

imaginaryStaticDielectricTensor()

Returns
The static dielectric constant tensor - i.e. including the ionic contribution. Dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

imaginarySusceptibility(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
Imaginary part of the susceptibility for the contribution.

Return type
numpy.ndarray.

`infraredOscillatorStrength()`

Returns
The infrared oscillator strength tensor. Dimension is (M, 3, 3) where M is the number of phonon modes.

Return type
Dimensionless complex ndarray.

`ionicDielectricTensor()`

Returns
The ionic part of the static dielectric constant tensor. Dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
`stream` (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`opticalConductivity(contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>)`

Parameters
`contribution` – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The optical conductivity tensor for the contribution. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
ndarray.

`phononEigenvectors()`

Returns
The phonon eigenvectors.

Return type
numpy.ndarray.

`phononEnergies()`

Returns
The phonon eigenenergies.

Return type
PhysicalQuantity of type energy.

`phononModes()`

Query method for the phonon modes.

Returns
The list of phonon mode indices used in the calculation.

Return type
list of int.

`phononWaveNumbers()`

Returns
The phonon wave numbers calculated from the phonon frequencies.

Return type
PhysicalQuantity of type inverse length.

`polarity()`

Returns
The infrared polarity of the phonon modes. Dimension is (M, 3) where M is the number of phonon modes.

Return type
PhysicalQuantity with the unit `charge/(mass^0.5)`; specifically `elementary_charge / electron_mass^(0.5)`.

`qpoint()`

Returns
The q-point for which the phonon modes and Raman signals are calculated.

Return type
ndarray with 3 floats.

`realSusceptibility(`*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*`)`

Parameters
contribution – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
Real part of the susceptibility for the contribution.

Return type
numpy.ndarray.

`reflectivity(`*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*`)`

Parameters
contribution – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The reflectivity tensor for the contribution. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

refractiveIndex(*contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Sum'>*)

Parameters
contribution (*Sum | Electronic | Ionic*) – NLFlag to determine the contributions to include. If Sum, the sum of the electronic and ionic contribution is returned. If Electronic or Ionic, only the electronic or ionic contributions are returned.

Returns
The refractive index including the ionic contribution. If `contribution` is `Sum` or `Electronic`, the dimension is (3, 3, N), where N is the number of energies. If `contribution` is `Ionic`, the dimension is (M, 3, 3, N), where N is the number of energies and M is the number of phonon modes.

Return type
Dimensionless ndarray.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

staticDielectricTensor()

Returns
The static dielectric constant tensor - i.e. including the ionic contribution. Dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

uniqueString()

Return a unique string representing the state of the object.

vibrationalMode()

Returns
A VibrationalMode object for the calculated modes.

Return type
VibrationalMode

Usage Examples

Calculate the static *DielectricTensor* of a configuration including both ionic and electronic contributions.

```
dielectric_tensor = DielectricTensor(
  configuration,
  dynamical_matrix,
  optical_spectrum,
  born_effective_charge)
```

The dielectric tensor will be evaluated at the same energies as defined in the input optical spectrum object.

Notes

The static (low frequency) dielectric response $\epsilon(\omega) = \epsilon_0\epsilon_r(\omega)$ of polar materials consists of an electronic and a ionic part:

$$\epsilon_{r,\alpha\beta}(\omega) = \epsilon^{elec}_{r,\alpha\beta}(\omega) + \epsilon^{ionic}_{r,\alpha\beta}(\omega)$$

where ϵ_r is the unit-less relative permittivity and α, β are Cartesian directions. The electronic part can be calculated from the *OpticalSpectrum* while the ionic part is related to the *BornEffectiveCharge* and phonon modes. At high frequency only the electrons can follow the optical field and the dynamic (high frequency) dielectric constant $\epsilon^{\infty}_{r,\alpha\beta} = \epsilon^{elec}_{r,\alpha\beta}(0)$ is obtained by evaluating the electronic dielectric constant at zero frequency.

At low frequency the ions can also respond to an optical field. The ionic part is calculated as[1]

$$\epsilon^{ionic}_{r,\alpha\beta}(\omega) = \frac{1}{\epsilon_0\Omega_0}\sum_{\lambda}\frac{S^{\lambda}_{\alpha\beta}}{\omega^2_{\lambda} - \omega^2 - i\Gamma\omega}\,,$$

where we introduced a broadening Γ for each phonon mode Lorentz oscillator, Ω_0 is the unit cell volume, ϵ_0 is the vacuum permittivity and $S^{\lambda}_{\alpha\beta} = p^{*\lambda}_{\alpha}p^{\lambda}_{\beta}$ is the mode oscillator strength which is related to the infrared mode polarity:

$$p^{\lambda}_{\alpha} = \sum_{i,\beta} Z_{\alpha\beta}(i)e^{\lambda}_{\beta}/\sqrt{m_i}$$

$Z_{\alpha\beta}(i)$ is the Born effective charge tensor for atom i with mass m_i and e^{λ}_{β} is the eigendisplacement of phonon mode λ with frequency ω_{λ}.

Infrared spectrum

The refractive index, n and extinction coefficient, κ, are obtained from the real and imaginary parts of the square root of the dielectric tensor (see also *OpticalSpectrum*):

$$n + i\kappa = \sqrt{\epsilon_r}\,.$$

The reflectivity at the interface between a vacuum region and the material is given by

$$R = \frac{(1-n)^2 + \kappa^2}{(1+n)^2 + \kappa^2}$$

The infrared spectrum is obtained by performing a dense energy sampling in the range of the vibrational frequency range (typically up to 0.25 eV or correspondingly 2000 cm^{-1}, depending on the material of interest).

The term infrared spectrum is used frequently for either the imaginary part of the dielectric tensor, $\text{Im}(\epsilon_r)$, the absorption coefficient $\alpha = 2\frac{\omega}{c}\kappa$ where c is the speed of light, and the transmittance, $T = 1 - r$ throughout literature.

When $\alpha(\omega) = 0$, with corresponding $T(\omega) = 1$, the material is transparent at that frequency.

[1] Xavier Gonze and Changyol Lee. Dynamical matrices, Born effective charges, dielectric permittivity tensors, and interatomic force constants from density-functional perturbation theory. *Phys. Rev. B*, 55(16):10355, April 1997. doi:10.1103/PhysRevB.55.10355.

Usage Example

This example script illustrates the workflow of the *DielectricTensor* calculation, using a classical force field for the phonons and DFT+1/2 for the electronic structure, to obtain the infrared spectrum of SiO_2 (Quatz):

```
filename = 'Quartz.hdf5'

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------
# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697        ,  0.            ,  0.            ],
                          [ 0.            ,  0.4697        ,  0.666666666667],
                          [ 0.5303        ,  0.5303        ,  0.333333333333],
                          [ 0.4135        ,  0.2669        ,  0.1191        ],
                          [ 0.2669        ,  0.4135        ,  0.547567      ],
                          [ 0.7331        ,  0.1466        ,  0.785767      ],
                          [ 0.5865        ,  0.8534        ,  0.214233      ],
                          [ 0.8534        ,  0.5865        ,  0.452433      ],
                          [ 0.1466        ,  0.7331        ,  0.8809        ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
# Define a classical calculator
potentialSet = Tersoff_SiO_2007()
calculator_classical = TremoloXCalculator(parameters=potentialSet)
calculator_classical.setVerletListsDelta(0.25 * Angstrom)

bulk_configuration.setCalculator(calculator_classical)

constraints = [BravaisLatticeConstraint()]
bulk_configuration = OptimizeGeometry(
    bulk_configuration,
    max_forces=0.001 * eV / Angstrom,
    max_stress=0.01 * GPa,
    max_steps=500,
    max_step_length=0.2 * Angstrom,
    constraints=constraints,
```

(continues on next page)

(continued from previous page)

```
    trajectory_filename=None,
)
nlsave(filename, bulk_configuration, object_id='bulk_configuration_classical')

# ------------------------------------------------------------
# Dynamical Matrix
# ------------------------------------------------------------
nrep = 5
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    filename=filename,
    object_id='dynamical_matrix',
    repetitions=(nrep, nrep, nrep),
    atomic_displacement=0.01 * Angstrom,
    acoustic_sum_rule=True,
    force_tolerance=1e-08 * Hartree / Bohr**2,
    processes_per_displacement=1,
    log_filename_prefix=None,
    )
dynamical_matrix.update()

# ------------------------------------------------------------
# Set relaxed configuration with DFT calculator
# ------------------------------------------------------------
exchange_correlation = GGAHalf.PBE

k_point_sampling = MonkhorstPackGrid(
    na=6,
    nb=6,
    nc=5,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=125.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=FermiDirac(100.0*Kelvin*boltzmann_constant),
    )

basis_set = [
    BasisGGAPseudoDojo.Oxygen_Medium,
    BasisGGAPseudoDojo.Silicon_Medium,
    ]

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    basis_set=basis_set,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave(filename, bulk_configuration, object_id='bulk_configuration_GGAHalf')
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
# Optical spectrum
# -----------------------------------------------------------
energies = numpy.linspace(0, 0.17358, 1001) * eV
optical_spectrum = OpticalSpectrum(
    configuration=bulk_configuration,
    kpoints=MonkhorstPackGrid(26, 26, 20),
    energies=energies,
    broadening=0.1 * eV,
    bands_below_fermi_level=1000,
    bands_above_fermi_level=1000,
)
nlsave(filename, optical_spectrum, object_id='optical_spectrum')

# -----------------------------------------------------------
# Born effective charges
# -----------------------------------------------------------
# Setup Born effective class object.
nk, nk_off = 7, 3
# Perform calculation.
born_effective_charge = BornEffectiveCharge(
   configuration=bulk_configuration,
   kpoints_a=MonkhorstPackGrid(nk, nk_off, nk_off),
   kpoints_b=MonkhorstPackGrid(nk_off, nk, nk_off),
   kpoints_c=MonkhorstPackGrid(nk_off, nk_off, nk),
   atomic_displacement=0.01 * Angstrom)
nlsave(filename, born_effective_charge, object_id='born_effective_charge')

# -----------------------------------------------------------
# Dielectric tensor
# -----------------------------------------------------------
dielectric_tensor = DielectricTensor(
        configuration=bulk_configuration,
        optical_spectrum=optical_spectrum,
        dynamical_matrix=dynamical_matrix,
        born_effective_charge=born_effective_charge,
        )
nlsave(filename, dielectric_tensor, object_id='dielectric_tensor')
```

dielectric_tensor_infrared_spectrum.py

Notice that the energies at interest are defined in the input to the *OpticalSpectrum* object.

It is important to note that this example is only intended to give an overview of the capabilities of *DielectricTensor*.

DiffractionPeak

class DiffractionPeak(*q_value*, *cutoff_radius=None*, *included_atoms=None*)

Class to calculate the intensity of the isotropic diffraction peak at the given q-value.

Parameters

- **q_value** (*PhysicalQuantity* of type inverse length) – The q-value at which the diffraction peak should be calculated.
- **cutoff_radius** (*PhysicalQuantity* of type length) – The cutoff radius for the atomic distances that should be included in the scattering function. **Default:** 10.0 Ang.
- **included_atoms** (PeriodicTableElement | *str* | *list*) – If not None, this parameter specifies the atoms between which the scattering function should be calculated. Can be specified as element, list of elements, or list of indices. **Default:** All atoms.

calculateCVAndDerivatives(*configuration*)

Calculate the value, derivative, and virial of the diffraction peak.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | DistributedConfiguration) – The configuration for which the diffraction peak should be calculated.

Returns
A tuple containing the value of the diffraction intensity (float), the derivatives with respect to atomic positions (PhysicalQuantity of type inverse length), and the derivatives with respect to the cell components (virial) (PhysicalQuantity of type inverse volume).

Return type
tuple

classmethod diffractionPlotData(*configuration*, *q_values*, *cutoff_radius=None*, *included_atoms=None*)

Helper method to create the X-ray diffraction data for a range of q-values, e.g. to create a plot.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the scattering plot should be calculated.
- **q_values** (*PhysicalQuantity* of type inverse length) – The q-values at which the x-ray diffraction should be calculated.
- **cutoff_radius** (*PhysicalQuantity* of type length) – The cutoff radius for the atomic distances that should be included in the scattering function. **Default:** 10.0 Ang.
- **included_atoms** (PeriodicTableElement | *str* | *list*) – If not None, this parameter specifies the atoms between which the scattering function should be calculated. Can be specified as element, list of elements, or list of indices. **Default:** All atoms.

Returns
The isotropic X-ray diffraction intensity at the given q-values.

Return type
numpy.ndarray

Usage Examples

Use the *DiffractionPeak* as collective variable in a *SteeredMolecularDynamics* to accelerate the crystallization in an amorphous cobalt silicide configuration using the MomentTensorPotential.

```
# Set up the diffraction peak intensity at q=2.1/Ang as collective variable.
# The diffraction intensity is only calculated for Cobalt atoms.
diffraction_peak = DiffractionPeak(
    cutoff_radius=10.0*Ang,
    q_value=2.1/Ang,
    included_atoms=[Cobalt],
)

# Set up the Steered MD with the diffraction peak intensity as collective variable.
# The object will be used as post-step-hook in the MD simulation.
hook_function = SteeredMolecularDynamics(
    collective_variable=diffraction_peak,
    velocity=300/1000/ps,
    spring_constant=0.5*eV,
    measurement_interval=50,
)

method = NVTNoseHoover(
    initial_velocity=MaxwellBoltzmannDistribution(
        temperature=1500.0*Kelvin
    ),
    reservoir_temperature=1500.0*Kelvin,
    thermostat_timescale=500.0*fs,
)

md_trajectory = MolecularDynamics(
    configuration=last_image_2,
    constraints=[
        FixCenterOfMass()
    ],
    trajectory_filename='CoSi2_bulk_crystallization.hdf5',
    steps=1200000,
    post_step_hook=hook_function,
    measurement_hook=hook_function.measurements,
    log_interval=5000,
    method=method
)
```

crystallization_example.py

Calculate and plot the diffraction intensities for a SiO2 cristobalite crystal.

```
# Set up the q-values at which the diffraction intensity should be plotted.
n_points = 100
q_values = numpy.linspace(1.0, 6.0, n_points) / Ang

xray_intensities = DiffractionPeak.diffractionPlotData(sio2_cristobalite, q_values,␣
↪cutoff_radius=12.0*Ang)
plot_model = Plot.PlotModel(Ang**-1)
```

(continues on next page)

(continued from previous page)

```
line = Plot.Line(q_values, xray_intensities)
plot_model.addItem(line)
plot_model.setLimits()

plot_model.xAxis().setLabel('q')
plot_model.yAxis().setLabel('Intensity')

Plot.show(plot_model)
```

plot_diffraction_sio2_cristobalite.py

Notes

The *DiffractionPeak* object calculates the intensity of the isotropic X-ray diffraction (XRD) peak at a given scattering vector q. The intensity of selected XRD peaks can be used as collective variable to accelerate the crystallization of bulk and interface materials in biased MD simulations[1].

The diffraction intensity is calculated as[Page 2704, 1]:

$$I(Q) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N} f_i(Q) f_j(Q) \frac{\sin(Q\dot{R}_{ij})}{Q\dot{R}_{ij}} W(R_{ij})$$

where Q is the magnitude of the scattering vector, N is the number of included atoms, $W(R_{ij})$ is defined as

$$W(R) = \frac{\sin(\pi R/R_c)}{\pi R/R_c}$$

and $f_i(Q)$ are the form factors which are calculated as described in[2] and *XRayScattering*.

Currently, the class can be used as collective variable in the *SteeredMolecularDynamics* hook object.

Before running a crystallization simulation, one should calculate the scattering plot over a range of q-vectors for the targeted crystal material (see second example above) to identify a characteristic peak for that material and use the corresponding q-value for the accelerated simulation. During the accelerated simulations the bias potential will try to change (e.g. increase) the value of the collective variable and thereby drive the configuration e.g. towards a larger value of the diffraction intensity. This normally results in a more pronounced ordering of the the atoms, which facilitates crystallization. Note that this collective variable does not introduce any bias with respect to the orientation of the crystal, as it acts only isotropically. Via the keyword `included_atoms` the peak intensity calculation can be restricted to a subset of atoms, e.g. only a specific element, or a group of atoms identified by their indices. This can e.g. be used if the difference in scattering intensity between disordered and crystalline structure is more pronounced for a sublattice, or if one wants to study crystallization in the presence of an interface, where the acceleration should only act on the atoms on one side of the interface.

Note, that currently this class does not support MPI domain decomposition in *MolecularDynamics()*, which is why the MD simulation should be run with OpenMP threading instead.

[1] H.ăNiu, P.M. Piaggi, M.ăInvernizzi, and Parrinello M. Molecular dynamics simulations of liquid silica crystallization. *Proc. Natl. Acad. Sci.*, 115(21):5348–5352, 2018. doi:https://doi.org/10.1073/pnas.1803919115.

[2] G.ăGutierrez and B.ăJohansson. Molecular dynamics study of structural properties of amorphous Al2O3. *Phys. Rev. B*, 65:104202, Feb 2002. doi:10.1103/PhysRevB.65.104202.

DimerConfiguration

class DimerConfiguration(*configuration*, *displacement=None*, *dimer_separation=None*, *eigenmode=None*)

Object that stores the dimer configuration consisting of the central image and the displaced images.

Parameters

- **configuration** (MoleculeConfiguration | BulkConfiguration) – The original configuration.
- **displacement** (*PhysicalQuantity* of type position) – A vector used to displace the original configuration. The displacement should be an estimate for the uphill mode leading to the closest minimum on the potential energy surface to the saddle point. Can also be random when exploring the potential energy surface.
- **dimer_separation** (*PhysicalQuantity* of type length) – The distance separating the displaced dimer images from the central image. **Default:** *1e-4 Angstrom*
- **eigenmode** (*numpy array of shape (n, 3)*) – The eigenmode to use for the initial search direction. If not supplied, the displacement vector is used.

calculator()

Returns
The calculator attached to the central configuration.

Return type
Calculator

centralEnergy()

Returns
The energy of the central configuration.

Return type
PhysicalQuantity with energy units

centralForces()

Returns
The forces of the central configuration.

Return type
PhysicalQuantity with force units

configuration()

Returns
The central configuration.

Return type
BulkConfiguration | Molecule

dimerForces()

Returns
The forces of the displaced dimer images.

Return type
List of length 2 of *PhysicalQuantity* with force units

setCalculator(*calculator*)

Attach a calculator to the *DimerConfiguration*.

Parameters
calculator (*Calculator* | None) – The calculator that should be attached to the configuration.

DirectSelfEnergy

class DirectSelfEnergy(*save_self_energies=None*, *lambda_min=None*, *sparse_threshold=None*, *storage_strategy=None*)

Self-energy calculator based on the direct method. The direct method uses direct diagonalization to get all the electrode modes.

Parameters

- **storage_strategy** (*SaveInMemory* | *StoreOnDisk* | *NoStorage*) – The way self energies are stored between iterations. **Default:** *SaveInMemory()*
- **sparse_threshold** (*float > 0*) – Self energies (in Hartree) smaller than this value are treated as 0 when storing as sparse matrix. **Default:** `1.e-12`
- **save_self_energies** – **Deprecated:** from v2017.0, use `storage_strategy` instead.
- **lambda_min** – **Deprecated:** from v12.2

lambdaMin()

Deprecated: from v12.2

nlinfo()

Returns
The nlinfo.

Return type
dict

saveSelfEnergies()

Deprecated: from v2017.0

sparseThreshold()

Returns
Values treated as 0.

Return type
float

storageStrategy()

Returns
The way self energies are stored between iterations.

Return type
SaveInMemory | *StoreOnDisk* | *NoStorage*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define that the self energy on the real contour is calculated with direct diagonalization.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_real=DirectSelfEnergy()
)
```

Examples on how to use the `storage_strategy` parameter can be found in the *Usage Examples* of *RecursionSelfEnergy*.

Notes

- *DirectSelfEnergy* uses a diagonalization scheme to find all propagating and decaying modes of the electrodes. The self energy matrix is then determined from the modes[1].
- The direct method implementation in QuantumATK is stable towards a sparse electrode Hamiltonian with null spaces. The implementation uses projection operators which removes any null spaces which could be problematic.

Reference

DirectSolver

class DirectSolver(*boundary_conditions=None*, *solvent_dielectric_constant=None*, *workspace_path=None*)

The direct Poisson solver.

The solvers required virtual memory can be reduced by allocating workspace on disk. Usage of a disk workspace has a small negative impact on the solvers performance.

Parameters

- **boundary_conditions** (list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*) – A list of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:**

– *MoleculeConfiguration*: `[[MultipoleBoundaryCondition()] * 2] * 3`

– *BulkConfiguration*: `[[PeriodicBoundaryCondition()] * 2] * 3`

– *SurfaceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition(), NeumannBoundaryCondition()]]
```

– *DeviceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition()] * 2]
```

[1] S.ăSanvito, C.ăJ. Lambert, J.ăH. Jefferson, and A.ăM. Bratkovsky. General Greens-function formalism for transport calculations with spd Hamiltonians and giant magnetoresistance in Co- and Ni-based magnetic multilayers. *Phys. Rev. B*, 59:11936–11948, May 1999. doi:10.1103/PhysRevB.59.11936.

- **solvent_dielectric_constant** (*float*) – The dielectric constant of the media surrounding the configuration. Must be a number larger than 0. **Default:** `1.0`
- **workspace_path** (*str of maximum length 255*) – Path to an existing directory usable as workspace. **Default:** Disk workspace disabled.

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

nlinfo()

Returns
The nlinfo.

Return type
dict

solventDielectricConstant()

Returns
The solvent dielectric constant.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

workspacePath()

Returns
The workspace path.

Return type
str

Usage Examples

Define a *DirectSolver* with specified boundary conditions on the 6 faces:

```
poisson_solver = DirectSolver(
    boundary_conditions=[
        [DirichletBoundaryCondition(), NeumannBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Define a *DirectSolver* with specified boundary conditions in the A, B, and C-direction, but similar boundary conditions on opposite faces:

```
poisson_solver = DirectSolver(
    boundary_conditions=[
        [NeumannBoundaryCondition(), NeumannBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Notes

- By setting the `solvent_dielectric_constant` it is possible to perform calculations of solvents. In this case the volume of the configuration is defined by inscribing each atom in a sphere of size given by the van der Waals radius of the element. Inside the volume of the configuration the dielectric constant is 1, outside the volume of the configuration the dielectric constant is equal to the value of `solvent_dielectric_constant`.
- The direct solver requires a significant amount of memory. When called in an environment with multiple MPI processes, the required memory will however be distributed over all available processes.
- See also *Poisson solvers*.

DirichletBoundaryCondition

class DirichletBoundaryCondition(*value=None*)

A Dirichlet type boundary condition.

Parameters
value (PhysicalQuantity of type energy | None) – The constraint value to fix the Hartree difference potential to.
Default: The potential of the electrode for device calculations, `0.0 * eV` otherwise.

value()

Returns
The constraint value.

Return type
PhysicalQuantity of type energy | None

DispersionD3BJ

class DispersionD3BJ(*xc=None*, *S6=None*, *a1=None*, *S8=None*, *a2=None*, *r_cut=None*, *bond_rcut=None*, *triple_rcut=None*, *k3=None*, *taper_factor=None*, *ATM=None*, *tags=None*)

The D3BJ dispersion potential.

Parameters

- **xc** (*str*) – Exchange-correlation functional name
- **S6** (*float*) – The dimensionless BJ/Z parameter S6
- **a1** (*float*) – The dimensionless BJ parameter a1
- **S8** (*float*) – The dimensionless BJ/Z parameter S8
- **a2** (*PhysicalQuantity of type length*) – The BJ parameter a2
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius for the interactions
- **bond_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the coordination numbers
- **triple_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the ATM interaction
- **k3** (*float*) – The dimensionless k3 smoothing parameter
- **taper_factor** (*float*) – The dimensionless taper_factor for bond_rcut and r_cut
- **ATM** (*bool*) – The parameter to switch on the Axilrod-Teller-Muto three-body terms
- **tags** (*str | list of type str*) – If tags is a string, the potential set acts only on particles with the given tag, if tag is list of strings of length 2, the potential acts only on particles with different tags.

actOnlyBetweenTaggedRegions(*tag1*, *tag2*)

If this potential set consists only of twobody interactions, calling this function makes all twobody interactions act between particles of the two given, different tags. If this potential contains other interactions than twobody interactions, calling this function will raise a RuntimeError. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters

- **tag1** (*str*) – The first tag.
- **tag2** (*str*) – The second tag.

actOnlyOnTaggedRegion(*tag*)

Calling this function makes this potential set act only on particles that have the given tag. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters
tag (*str*) – The tag on which this potential set should act.

addAnalyzer(*analyzer*)

Adds an analyzer to this potential set.

Parameters
analyzer (*TremoloBaseAnalyzer*) – The analyzer to should be added.

addOption(*option*)

Add an option to the potential.

Parameters
option (*TremoloOption*) – The option that will be added.

addParticleType(*particle*)

Add a particle type definition to the potential set.

Parameters
particle (ParticleType) – Instance of ParticleType.

addPotential(*potential*)

Add a potential to this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential that will be added.

findPotentials(*potentialType=None*, *particleTypes=None*)

Return a list of potentials in this potential set, that fullfil certain conditions.

Parameters

- **potentialType** (*TremoloBasePotential*) – A potential class (e.g. LennardJonesPotential).
- **particleTypes** (*str | ParticleType | ParticleIdentifier | sequence of type str/ ParticleType/ParticleIdentifier*) – A string or a sequence of strings that determine the particle types which the potential should act on.

Returns
The list of potentials that are of type potentialType and act on the particle types that are given by particleTypes.

Return type
list of type TremoloBasePotential

getCoulombSolver()

Returns
The current Coulomb potential or None.

Return type
CoulombSolver

getOptions()

Return a list of all options that are stored in this potential set.

getParticleType(*symbol*)

Return the particle definition with the given symbol

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

Returns
The ParticleType that belongs to the given identifier or None if no such particle was found.

Return type
ParticleType | None

getParticleTypes()

Return a list of all particle types that are stored in this potential set.

getPotentials()

Return a list of all potentials that are stored in this potential set.

getUsedTags()

Returns
The tags this potential set acts on.

Return type
set of type str

merge(*otherPotentialSet*)

Merges the current potential set with another potential set.

Parameters
otherPotentialSet (*PotentialSetBase*) – The potential set, the current potential should be merged with.

static mergePotentials(*potentials*)

Merges a list of potential sets and returns the merged potential set.

Parameters
potentials (*list of type PotentialSetBase*) – List of potential sets to be merged.

Returns
The potential set that was created by merging all potentials in the list.

Return type
PotentialSetBase

removeParticleType(*symbol*)

Remove a given particle type from the potential set.

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

removePotential(*potential*)

Remove a potential from this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential to be removed, this potential must have been retrieved by the findPotentials method.

setCoulombSolver(*coulomb*)

Sets the coulomb potential. Previously set Coulomb solvers will be overwritten.

Parameters
coulomb (CoulombSolver | *None*) – The new Coulomb solver. Must either be None or an object that was derived from the CoulombSolver class.

setPreferredUnitSystem(*unitSystem*)

Set the preferred unit system for this potential set. The choice here may be overridden if some potentials require a specific unit system.

Parameters
unitSystem (*TremoloUnitSystem*) – The unit system, e.g. (eVolt, kCalPerMole, SI).

setTags(*tags*)

Makes this potential set act only on particles with a given tag Note: This operation can not be reverted easily and will change the names of your particles. Use it only when you are finished building the potential set!

Parameters
tags (*str | list of type str*) – If tags is a string, the potential set acts only on particles with the given tag, if tag is list of strings of length 2, the potential acts only on particles with different tags.

supportedElements()

Return the chemical elements that are supported by this potential set. The elements are returned as a list that contains the element symbols.

Returns
The list of the symbols of the supported elements.

Return type
list of type str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Create a potential set that implements the D3 dispersion correction with Becke-Johnson damping[1]. This can be combined with a *MTPPotential* (MTP) fitted to Density Functional Theory (DFT) calculations in a *TremoloXCalculator*. Doing so can correct for some systematic errors in DFT.

```
potentialSet = TremoloXPotentialSet(name = 'New Potential Set')
potentialSet.addParticleType(ParticleType('N'))
potentialSet.addParticleType(ParticleType('Au'))
potentialSet.addParticleType(ParticleType('B'))
_potential = MTPPotential(
    file = 'MTP_parameters.mtp',
    suppress_intercept = False,
    group_name = '',
)
potentialSet.addPotential(_potential)

d3_potential_set = DispersionD3BJ(xc='PBE')

calculator = TremoloXCalculator(parameters=[potentialSet, d3_potential_set])
```

Notes

Parameters are available for the following functionals:

B97-D3, BP86, BOP, BLYP, mPWLYP, OLYP, PBE, PBEsol, revPBE, BPBE, OPBE, rPW86PBE, SSB, revSSB, TPSS, oTPSS, M06-L, B3LYP, B3PW91, BHLYP, PBE0, revPBE0, PBE38, revPBE38, TPSSh, TPSS0, PW6B95, MPW1B95, PWB6K, MPWB1K, B1B95, BMK, M05, M052X, M06, M062X, M06HF, CAM-B3LYP,LC-wPBE, B2PLYP, B2GPPLYP, PWPB95, DSD-BLYPB2PLYP, B3LYP, B97D, BLYP, BP86, DSDBLYP, PBE0, PBE, PW6B95, PWPB95, revPBE0, revPBE38, revPBE, rPW86PBE, TPSS0, TPSS, B1B95, B2GPPLYP, B3PW91, BHLYP, BMK, BOP, BPBE, CAMB3LYP, LCwPBE, MPW1B95, MPWB1K, mPWLYP, OLYP, OPBE, oTPSS, PBE38, PBEsol, PTPSS, PWB6K, revSSB, SSB, TPSSh, HCTH120, PBEhPBE, RPBE, PW91, BPBE, OPBE, OLYP, XLYP, mPWLYP, mPWPW91, HCTH/407, revTPSS, SCAN, tHCTH, MN12L, B3PW91, B3P86, BHLYP, B1P86, B1LYP, B1B95, MPW1B95, MPWB1K, MPW1PW91, MPW1KCIS, MPWKCIS1K, PBEh1PBE, PBE1KCIS, X3LYP, O3LYP, B97-1, B97-2, B98, HISS, HSE03, HSE06, TPSSh, revTPSSh, revTPSS0, TPSS1KCIS, BMK, tHCTHhyb, M11, SOGGA11X, N12SX, MN12SX, MN15, LC-whPBE, B2PLYP, B2GPPLYP, MPW2PLYP, PWPB95, DSD-BLYP, DSD-PBEP86, DSD-PBEB95

[1] Stefan Grimme, Stephan Ehrlich, and Lars Goerigk. Effect of the damping function in dispersion corrected density functional theory. *Journal of Computational Chemistry*, 32(7):1456–1465, 2011.

DispersionD3Z

class DispersionD3Z(*xc=None*, *S6=None*, *S6r=None*, *S8=None*, *r_cut=None*, *bond_rcut=None*, *triple_rcut=None*, *k3=None*, *taper_factor=None*, *ATM=None*, *tags=None*)

The D3Z dispersion potential.

Parameters

- **xc** (*str*) – Exchange-correlation functional name
- **S6** (*float*) – The dimensionless BJ/Z parameter S6
- **S6r** (*float*) – The dimensionless Z parameter S6r
- **S8** (*float*) – The dimensionless BJ/Z parameter S8
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius for the interactions
- **bond_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the coordination numbers
- **triple_rcut** (*PhysicalQuantity of type length*) – The cutoff radius for the ATM interaction
- **k3** (*float*) – The dimensionless k3 smoothing parameter
- **taper_factor** (*float*) – The dimensionless taper_factor for bond_rcut and r_cut
- **ATM** (*bool*) – The parameter to switch on the Axilrod-Teller-Muto three-body terms
- **tags** (*str | list of type str*) – If tags is a string, the potential set acts only on particles with the given tag, if tag is list of strings of length 2, the potential acts only on particles with different tags.

actOnlyBetweenTaggedRegions(*tag1*, *tag2*)

If this potential set consists only of twobody interactions, calling this function makes all twobody interactions act between particles of the two given, different tags. If this potential contains other interactions than twobody interactions, calling this function will raise a RuntimeError. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters

- **tag1** (*str*) – The first tag.
- **tag2** (*str*) – The second tag.

actOnlyOnTaggedRegion(*tag*)

Calling this function makes this potential set act only on particles that have the given tag. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters
tag (*str*) – The tag on which this potential set should act.

addAnalyzer(*analyzer*)

Adds an analyzer to this potential set.

Parameters
analyzer (*TremoloBaseAnalyzer*) – The analyzer to should be added.

addOption(*option*)

Add an option to the potential.

Parameters
option (*TremoloOption*) – The option that will be added.

addParticleType(*particle*)

Add a particle type definition to the potential set.

Parameters
particle (ParticleType) – Instance of ParticleType.

addPotential(*potential*)

Add a potential to this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential that will be added.

findPotentials(*potentialType=None*, *particleTypes=None*)

Return a list of potentials in this potential set, that fullfil certain conditions.

Parameters

- **potentialType** (*TremoloBasePotential*) – A potential class (e.g. LennardJonesPotential).
- **particleTypes** (*str | ParticleType | ParticleIdentifier | sequence of type str/ ParticleType/ParticleIdentifier*) – A string or a sequence of strings that determine the particle types which the potential should act on.

Returns
The list of potentials that are of type potentialType and act on the particle types that are given by particleTypes.

Return type
list of type TremoloBasePotential

getCoulombSolver()

Returns
The current Coulomb potential or None.

Return type
CoulombSolver

getOptions()

Return a list of all options that are stored in this potential set.

getParticleType(*symbol*)

Return the particle definition with the given symbol

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

Returns
The ParticleType that belongs to the given identifier or None if no such particle was found.

Return type
ParticleType | None

getParticleTypes()

Return a list of all particle types that are stored in this potential set.

getPotentials()

Return a list of all potentials that are stored in this potential set.

getUsedTags()

Returns
The tags this potential set acts on.

Return type
set of type str

merge(*otherPotentialSet*)

Merges the current potential set with another potential set.

Parameters
otherPotentialSet (*PotentialSetBase*) – The potential set, the current potential should be merged with.

static mergePotentials(*potentials*)

Merges a list of potential sets and returns the merged potential set.

Parameters
potentials (*list of type PotentialSetBase*) – List of potential sets to be merged.

Returns
The potential set that was created by merging all potentials in the list.

Return type
PotentialSetBase

removeParticleType(*symbol*)

Remove a given particle type from the potential set.

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

removePotential(*potential*)

Remove a potential from this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential to be removed, this potential must have been retrieved by the findPotentials method.

setCoulombSolver(*coulomb*)

Sets the coulomb potential. Previously set Coulomb solvers will be overwritten.

Parameters
coulomb (CoulombSolver | *None*) – The new Coulomb solver. Must either be None or an object that was derived from the CoulombSolver class.

setPreferredUnitSystem(*unitSystem*)

Set the preferred unit system for this potential set. The choice here may be overridden if some potentials require a specific unit system.

Parameters
unitSystem (*TremoloUnitSystem*) – The unit system, e.g. (eVolt, kCalPerMole, SI).

setTags(*tags*)

Makes this potential set act only on particles with a given tag Note: This operation can not be reverted easily and will change the names of your particles. Use it only when you are finished building the potential set!

Parameters
tags (*str | list of type str*) – If tags is a string, the potential set acts only on particles with the given tag, if tag is list of strings of length 2, the potential acts only on particles with different tags.

supportedElements()

Return the chemical elements that are supported by this potential set. The elements are returned as a list that contains the element symbols.

Returns
The list of the symbols of the supported elements.

Return type
list of type str

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Create a potential set that implements the D3 dispersion correction with zero damping[1]. This can be combined with a *MTPPotential* (MTP) fitted to Density Functional Theory (DFT) calculations in a *TremoloXCalculator*. Doing so can correct for some systematic errors in DFT.

```
potentialSet = TremoloXPotentialSet(name = 'New Potential Set')
potentialSet.addParticleType(ParticleType('N'))
potentialSet.addParticleType(ParticleType('Au'))
potentialSet.addParticleType(ParticleType('B'))
_potential = MTPPotential(
    file = 'MTP_parameters.mtp',
    suppress_intercept = False,
    group_name = '',
)
potentialSet.addPotential(_potential)

d3_potential_set = DispersionD3Z(xc='PBE')

calculator = TremoloXCalculator(parameters=[potentialSet, d3_potential_set])
```

Notes

Parameters are available for the following functionals:

B97-D3, BP86, BOP, BLYP, mPWLYP, OLYP, PBE, PBEsol, revPBE, BPBE, OPBE, rPW86PBE, SSB, revSSB, TPSS, oTPSS, M06-L, B3LYP, B3PW91, BHLYP, PBE0, revPBE0, PBE38, revPBE38, TPSSh, TPSS0, PW6B95, MPW1B95, PWB6K, MPWB1K, B1B95, BMK, M05, M052X, M06, M062X, M06HF, CAM-B3LYP LC-wPBE, B2PLYP, B2GPPLYP, PWPB95, DSD-BLYP

DoubleContour

`class DoubleContour`

A simple flag-like class representing a double contour.

[1] S.ăGrimme, J.ăAntony, S.ăEhrlich, and H.ăKrieg. A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu. *J. Chem. Phys.*, 2010. doi:10.1063/1.3382344.

Usage Example

The *DoubleContour* class is passed on to the *ContourParameters* to specify a DoubleContour should be used to calculate the density matrix.

```
contour_parameters = ContourParameters(method=DoubleContour())
```

Notes

When there is a finite bias in the system there are different equivalent ways to obtain the density matrix (see *ContourParameters*). In principle the left and right estimates of the density matrix are equivalent:

$$D = D^L + \Delta^R \, ,$$
$$D = D^R + \Delta^L \, ,$$

where D^L, (D^R) is the equilibrium density matrix, calculated using one of the equilibrium methods, and Δ^L (Δ^R) the non-equilibrium density matrix contribution.

Due to the finite accuracy of the integration along the real axis these results are not exactly the same. Therefore, if the double contour option is specified both the left and right part are calculated and mixed together to obtain an accurate and reliable calculation of the density matrix at finite bias.

Weighting Scheme

Calculate the non-equilibrium atom Mulliken populations

$$\alpha_i^L = (\Delta D^L S)_{ii},$$

$$\alpha_i^R = (\Delta D^R S)_{ii}.$$

From the α coefficients, calculate the weights

$$w_{ij}^L = \frac{\sqrt{\alpha_i^L \alpha_j^L}}{\sqrt{\alpha_i^L \alpha_j^L} + \sqrt{\alpha_i^R \alpha_j^R}},$$

$$w_{ij}^R = \frac{\sqrt{\alpha_i^R \alpha_j^R}}{\sqrt{\alpha_i^L \alpha_j^L} + \sqrt{\alpha_i^R \alpha_j^R}}.$$

The weighted estimate for the density matrix is then

$$D_{ij} = w^L (D^L + \Delta D^R) + w^R (D^R + \Delta D^L)$$

This weighting scheme guarantees at each site that the equilibrium part of the density matrix is the largest contribution. Since the equilibrium part of the density matrix is calculated from a complex contour it is usually much more accurate than the non-equilibrium part.

Bound states in the bias window

Assume the right Fermi level is higher than the left Fermi level, then a bound state in the bias window will only be included in D^R. The occupation of the bound state is then given by w^R, which favors occupation of bound states at sites that are strongly connected with the right electrode.

DreidingAnglePotential

class DreidingAnglePotential(*particleType1*, *particleType2*, *particleType3*, *A*, *B*, *r_cut*)

Constructor of the potential.

This potential is not symmetric, i.e. activating particleType1-particleType2-particleType3 interactions will not also activate particleType3-particleType2-particleType1 interactions.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the donator particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the hydrogen like particle type (central particle when calculating angles)
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of tehe acceptor particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a hydrogen-bond potential for the O-H–O hydrogen bond in water.

```
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = TremoloXPotentialSet(name='Dreiding_hbond_angle')
potentialSet.addParticleType(ParticleType.fromElement(Oxygen))
potentialSet.addParticleType(ParticleType.fromElement(Hydrogen))

_potential = DreidingAnglePotential(
        particleType1=ParticleIdentifier('O', []),
        particleType2=ParticleIdentifier('H', []),
        particleType2=ParticleIdentifier('O', []),
        A=7.0*kiloCaloriePerMol,
        B=2.75*Angstrom,
        r_cut=7.5*Angstrom,
)
potentialSet.addPotential(_potential)
calculator = TremoloXCalculator(parameters=potentialSet)
```

dreiding_angle.py

Notes

The Dreiding angle potential was originally introduced in[1] to model hydrogen bonding interactions between hydrogen bond donors and acceptors. It has the potential form

$$U = \sum_i \sum_{j \neq i} \sum_{\substack{k>i \\ k \neq j}} U_{ijk}$$

$$U_{ijk} = A_{ijk} \left[5 \left(\frac{B_{ijk}}{r_{ik}} \right)^{12} - 6 \left(\frac{B_{ijk}}{r_{ik}} \right)^{10} \right] \min(0, \cos(\theta_{ijk}))^4.$$

The $\min(0, \cos(\theta_{ijk}))$ part of this equation excludes all triple terms where the angle between the three particles is smaller than 90 degrees (and thus $\cos(\theta_{ijk}) > 0$). This was the expression used in the original paper (Page 2720, 1).

DreidingPotentialBuilder

class DreidingPotentialBuilder(*use_bonds=None*, *include_bond_stretching=None*, *include_bond_angles=None*, *include_torsions=None*, *include_inversions=None*, *include_h_bonds=None*, *include_lennard_jones=None*, *include_electrostatic=None*, *lennard_jones_cutoff=None*, *lennard_jones_smoothing_length=None*, *h_bond_scale=None*, *h_bond_cutoff=None*, *electrostatic_pairwise_cutoff=None*, *electrostatic_smoothing_length=None*, *electrostatic_spme_cutoff=None*, *electrostatic_accuracy=None*, *electrostatic_scale=None*, *use_electrostatic_direct_sum=None*, *ignore_missing_types=None*)

[1] StephenăL. Mayo, BarryăD. Olafson, and WilliamăA. Goddard. DREIDING: A generic force field for molecular simulations. *J. Phys. Chem.*, 94(26):8897–8909, 1990. doi:10.1021/j100389a010.

This object builds a DREIDING potential using TremoloX.

Parameters

•**use_bonds** – Controls if any bonding (stretching, angle, torsion and inversion) terms are to be included. **Default:** True.

•**include_bond_stretching** (*bool*) – Controls if bonding stretching terms are included. **Default:** use_bonds.

•**include_bond_angles** (*bool*) – Controls if angle bending terms are included. **Default:** use_bonds.

•**include_torsions** (*bool*) – Controls if bond torsion terms are included. **Default:** use_bonds.

•**include_inversions** (*bool*) – Controls if bond inversion torsion terms are included. **Default:** use_bonds.

•**include_h_bonds** (*bool*) – Controls if hydrogen bonding terms are included. **Default:** use_bonds.

•**include_lennard_jones** (*bool*) – Controls if Lennard-Jones terms are included. **Default:** True.

•**include_electrostatic** (*bool*) – Controls if electrostatic terms are included. **Default:** True.

•**lennard_jones_cutoff** (*PhysicalQuantity* of type distance) – Distance beyond which the Lennard-Jones terms are truncated. **Default:** 10.0 * Angstrom.

•**lennard_jones_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the Lennard-Jones terms are brought to zero at the cutoff. **Default:** 2.0 * Angstrom.

•**h_bond_scale** (*float*) – Scaling for the strength of the hydrogen bond term to allow for setting atomic charges. **Default:** 1.0.

•**h_bond_cutoff** (*PhysicalQuantity* of type distance) – Distance beyond which the Hydrogen bond terms are truncated. **Default:** 7.5 * Angstrom.

•**electrostatic_pairwise_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in pairwise summation. **Default:** 12 * Angstrom

•**electrostatic_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the electrostatic terms are brought to zero at the cutoff when using the direct electrostatic sum. **Default:** 2.0 * Angstrom

•**electrostatic_spme_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in SPME summation. **Default:** 7.5 * Angstrom

•**electrostatic_accuracy** (*float*) – Sets the relative accuracy of the SPME summation. **Default:** 0.0001

•**electrostatic_scale** (*float*) – Set the relative strength of the electrostatic terms. **Default:** 1.0

•**use_electrostatic_direct_sum** (*bool*) – Use the direct Coulomb sum to calculate the electrostatic terms in MoleculeConfigurations. **Default:** False

•**ignore_missing_types** (*bool*) – Whether or not molecules missing Dreiding types are ignored. **Default:** False

assignAtomTypes(*configuration*, *overwrite=True*)

Function that gets the atom types based on the bonding pattern.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration of the system that is having types assigned. **Default:** True

•**overwrite** (*bool*) – Whether or not the existing Dreiding types are overwritten.

checkAtomTypes(*configuration*, *add_missing_terms=False*)

Function that tests that each atom has only one valid Dreiding type.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration whose types are being checked.
- **add_missing_terms** (*bool*) – Whether or not to add terms for missing Dreiding types. **Default:** False

createCalculator(*configuration*, *check_types=True*, *atomic_charges=None*, *charge_averaging=None*)

Create and return a calculator with the Dreiding potential.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration the potential is to be used with.
- **check_types** (*bool*) – Flag as to whether or not the correctness of the types are checked. **Default:** True
- **atomic_charges** (*PhysicalQuantity of type charge*) – Custom atomic charges to be used in the potential.
- **charge_averaging** (*NLFlag | None*) – The type of averaging of charges to be used, if any. Accepted values are None, AtomConnectivity to average over atoms bonded to the same elements and AtomTypes to average over atoms of the same forcefield type. **Default:** None

Returns
The calculator containing the potential.

Return type
TremoloXCalculator

dreidingType(*index*, *configuration*, *add_missing_terms=False*)

Gives the Dreiding type on an atom in a configuration, ignoring other tags.

Parameters

- **index** (*int*) – Index of the atom for which the type is sought.
- **configuration** (*AtomicConfiguration*) – The configuration containing the atom.
- **add_missing_terms** (*bool*) – Whether or not generic Dreiding types are considered.

Returns
The type of the atom.

Return type
str

electrostaticAccuracy()

Returns
The relative accuracy of the electrostatic summation for SPME.

Return type
float

electrostaticPairwiseCutoff()

Returns
The cutoff used in pairwise electrostatic summation.

Return type
PhysicalQuantity (length)

electrostaticScale()

Returns
Scaling factor for the electrostatic terms.

Return type
float

electrostaticSmoothingLength()

Returns
Distance at which electrostatic terms are splined when using direct summation.

Return type
PhysicalQuantity (length)

electrostaticSpmeCutoff()

Returns
The cutoff used in SPME electrostatic summation.

Return type
PhysicalQuantity (length)

hBondCutoff()

Returns
Distance at which hydrogen bond terms are ignored.

Return type
PhysicalQuantity (length)

hBondScale()

Returns
Scaling of the hydrogen bond energy because of included charges.

Return type
float

ignoreMissingTypes()

Returns
Whether or not missing types are ignored on molecules.

Return type
bool

includeBondAngles()

Returns
Whether or not bond angle terms are included in the potential set.

Return type
bool

includeBondStretching()

Returns
Whether or not bond stretching terms are included in the potential set.

Return type
bool

includeElectrostatic()

Returns
Whether or not electrostatic terms are included in the potential set.

Return type
bool

includeHBonds()

Returns
Whether or not hydrogen-bond terms are included in the potential set.

Return type
bool

includeInversions()

Returns
Whether or not bond inversion terms are included in the potential set.

Return type
bool

includeLennardJones()

Returns
Whether or not Lennard Jones terms are included in the potential set.

Return type
bool

includeTorsions()

Returns
Whether or not bond torsion terms are included in the potential set.

Return type
bool

isDreidingType(*type_label*)

Function that tests if a type is in the potential

Parameters
type_label (*str*) – The type being tested.

Returns
Whether or not the tag is defined in the potential.

Return type
bool

lennardJonesCutoff()

Returns
Distance at which Lennard Jones terms are ignored.

Return type
PhysicalQuantity (length)

lennardJonesSmoothingLength()

Returns
Distance at which Lennard Jones terms are splined.

Return type
PhysicalQuantity (length)

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

useBonds()

Returns
Whether or not bond terms are included in the potential set.

Return type
bool

useElectrostaticDirectSum()

Returns
Whether or not direct Coulomb sums are used for molecules.

Return type
bool

Usage Examples

Investigate the diffusion of chloroform molecules in LTA zeolites using the DREIDING potential.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('LTA_Chloroform.hdf5')[-1]

# ------------------------------------------------------------
# DREIDING Calculator
# ------------------------------------------------------------
potential_builder = DreidingPotentialBuilder()
potential_builder.assignAtomTypes(bulk_configuration)
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)

method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=initial_velocity,
```

(continues on next page)

(continued from previous page)

```
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='LTA_Chloroform_Trajectory.hdf5',
    steps=100000,
    log_interval=500,
    method=method
)

bulk_configuration = md_trajectory.lastImage()
```

LTA_Chloroform.py LTA_Chloroform.hdf5

Notes

The DreidingPotentialBuilder class enables the creation of a *TremoloXCalculator* that implements the DREIDING potential of Mayo, Olafson and Goddard[1]. This is a bonded valence forcefield that represents the potential energy of a molecule as a collection of simple functions based on bond lengths, bond angles, torsion angles, inversion angles and inter-atomic distances. The overall mathematical form of the DREIDING potential can be given as:

$$E(\mathbf{x}) = \sum_{i}^{bonds} k_i(r_i - r_0)^2 + \sum_{i}^{angles} k_i(\theta_i - \theta_0)^2 + \sum_{i}^{torsions} \sum_{n=1}^{3} k_{i,n}(1 - \cos(n\phi_i - \delta_{i,n}))$$
$$+ \sum_{i}^{inversion} k_i(\chi_i - \chi_0)^2 + \sum_{i}^{H-bonds} A_i \left[5 \left(\frac{B_i}{r_{ac}} \right)^{12} - 6 \left(\frac{B_i}{r_{ac}} \right)^{10} \right] \cos^4(\theta_{abc})$$
$$+ \sum_{ij}^{atoms} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \sum_{ij}^{atoms} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Here r, θ, ϕ and χ represent inter-atomic and bond distances, angles, torsions and inversions respectively in the configuration.

To create a DREIDING calculator, an instance of the *DreidingPotentialBuilder* must first be created. The constructor for this class has a number of options that controls the way in which the calculator is built. Individual types of interactions can be excluded by setting the `include_` options to `False`. By default all possible terms are included. All valence terms can also be excluded by setting the argument `use_bonds` to False. This is appropriate for configurations consisting of single atoms that are not bonded to each other.

As the *DreidingPotentialBuilder* class uses the bonding in the configuration to determine appropriate potentials, bonds should be set on the configuration. This can be done with the `findBonds()` method in either the *MoleculeConfiguration* or *BulkConfiguration* class. This automatically assigns bonds between atoms that are within the covalent radii of each other. Specific bonds can also be assigned with the `setBonds()`, which takes a list of the indices of pairs of atoms that are to be considered bonded.

There are then a number of options that control the non-bonding interactions. For hydrogen bonding, the cutoff used in calculating hydrogen bonding interactions can be set with the parameter `h_bond_cutoff`. The strength of the hydrogen bonding interactions can also be scaled using the `h_bond_scale` parameter. In the original DREIDING

[1] StephenăL. Mayo, BarryăD. Olafson, and WilliamăA. Goddard. DREIDING: A generic force field for molecular simulations. *J. Phys. Chem.*, 94(26):8897–8909, 1990. doi:10.1021/j100389a010.

paper it was mentioned that it may be beneficial to scale down the strength of hydrogen bonding when combined with atomic partial charges, as part of the hydrogen bonding effect may be replicated by the electrostatic interaction. By default, the DREIDING potential builder adds unscaled hydrogen bonding potentials to the calculator. Lennard-Jones interactions can be similarly controlled. The parameter `lennard_jones_cutoff` sets the distance beyond which Lennard-Jones interactions are truncated. To avoid integration problems in molecular dynamics, Lennard-Jones interactions are brought to zero at the cutoff using a spline function. The width of this spline can be set using the `lennard_jones_smoothing_length` parameter.

Electrostatic interactions are a little more complicated. By default the *DreidingPotentialBuilder* adds a *CoulombSPME* electrostatic solver for calculators to be used with a bulk configuration, and a *CoulombDSF* electrostatic solver to calculators to be used with a molecule configuration. When adding an *CoulombSPME* solver the cutoff used for calculating the real-space interactions can be set using the `electrostatic_spme_cutoff` argument. Likewise the accuracy of the smoothed particle mesh Ewald solver can be set using the `electrostatic_accuracy` argument. In the case where a *CoulombDSF* solver is added to the potential, the cutoff used can be set using the `electrostatic_pairwise_cutoff` argument. In cases where molecular electrostatics are to be compared with values calculated in a bulk configuration, an explicit splined Coulomb potential can be used for molecules instead of the damped shifted force potential. In this case a *CoulombN2Spline* electrostatic solver can be added by using `use_electrostatic_direct_sum` argument. As with the damped shifted force potential the cutoff is controlled using the `electrostatic_pairwise_cutoff` argument, and like the Lennard-Jones potential is brought to zero with a spline whose length is defined by the `electrostatic_smoothing_length` argument. For all Coulomb solvers the total electrostatic energies and forces can be scaled up or down using the `electrostatic_scale` argument.

Once the *DreidingPotentialBuilder* object is instantiated, a calculator is created using the `createCalculator` function. This takes a configuration and returns the appropriate DREIDING calculator for that configuration. To use electrostatic potentials in the calculator, atomic partial charges have to be assigned to each atom. As the DREIDING forcefield does not define specific charges, by default charges are calculated with the *QEqAtomicCharges* class. This implements the QEq charge equilibration method of Rappe and Goddard[2]. These charges are then added to the calculator. It is also possible to add pre-calculated charges from different sources using the `atomic_charges` argument. This argument takes a list of charges that have the same ordering as the atoms in the configuration. The partial atomic charges can also be averaged according to a few different schemes. This is controlled by the `charge_averaging` parameter, which takes an NLFlag which indicates the type of averaging. The flag `AtomConnectivity` averages charges on each atom based on the element and the other elements it is bonded to. The flag `AtomType` averages according to the assigned DREIDING type.

When creating a DREIDING calculator for a configuration, the configuration must have the appropriate tags for each atom type. Tags used by the DREIDING forcefield all begin with the prefix `DREI_`. The next two characters in the tag are for the atomic symbol, followed by a possible character to indicate the hybridization of the atom. Tags can be assigned manually, or they can be assigned automatically. Automatic typing can be done with the `assignAtomTypes` function. This takes the configuration to have types added, as well as a possible flag to overwrite existing types. The atom types are determined by the element and in some cases the bonding of the atom. The validity of the existing types on a configuration can be checked by calling the `checkAtomTypes` function. This function raises a `NLValueError` if a problem is found with the tags on the given configuration. The DREIDING type of a specific atom can also be found using the `dreidingType` function.

[2] AnthonyăK. Rappe and WillianăA. GoddardăIII. Charge equilibration for molecular dynamics simulations. *J. Phys. Chem.*, 95:3358, April 1991. URL: https://pubs.acs.org/doi/10.1021/j100161a070, doi:10.1021/j100161a070.

DynamicalMatrix

class DynamicalMatrix(*configuration*, *filename*, *object_id*, *calculator=None*, *repetitions=None*, *atomic_displacement=None*, *acoustic_sum_rule=None*, *finite_difference_method=None*, *constraints=None*, *constrain_electrodes=None*, *use_equivalent_bulk=None*, *max_interaction_range=None*, *force_tolerance=None*, *processes_per_displacement=None*, *log_filename_prefix=None*, *use_wigner_seitz_scheme=None*, *use_symmetry=None*, *polar_phonon_splitting_parameters=None*, *use_internal_coordinates=None*, *bond_list=None*, *symmetrize=None*)

Constructor for the DynamicalMatrix object.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration* | *DeviceConfiguration*) – The configuration for which to calculate the dynamical matrix.
- **calculator** (Calculators) – The calculator to be used in the dynamical matrix calculations. **Default:** The calculator attached to the configuration.
- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **repetitions** (Automatic | list of ints) – The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. Each repetition value must be odd. **Default:** `Automatic`
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01 * Angstrom`
- **acoustic_sum_rule** (*bool*) – Control if the acoustic sum rule should be invoked. **Default:** `True`
- **finite_difference_method** (Forward | Central) – The finite difference scheme to use. **Default:** `Central`
- **constraints** (*list of type int*) – List of atomic indices that will be constrained, e.g. `[0, 2, 10]`. **Default:** None
- **constrain_electrodes** (*bool*) – Control if the electrodes and electrode extensions should be constrained in case of a *DeviceConfiguration*. **Default:** `False`
- **use_equivalent_bulk** (*bool*) – Control if a *DeviceConfiguration* should be treated as a *BulkConfiguration*. **Default:** `True`
- **max_interaction_range** (*PhysicalQuantity* of type length) – Set the maximum range of the interactions. **Default:** All atoms are included
- **force_tolerance** (*PhysicalQuantity* of type energy per length squared) – All force constants below this value will be truncated to zero. **Default:** `1e-8 * Hartree/Bohr**2`
- **processes_per_displacement** (int | AllProcesses | *ProcessesPerNode*) – The number of processes assigned to calculating a single displacement. **Default:** `1` for force-field calculators, `AllProcesses` otherwise.
- **log_filename_prefix** (*str or None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. If a classical calculator is used, no per-displacment log files will be generated. **Default:** `"forces_displacement_"`
- **use_wigner_seitz_scheme** (*bool*) – Control if the real space Dynamical Matrix should be extended according to the Wigner Seitz construction. `use_wigner_seitz_scheme=True`. **Default:** `False` for ForceField calculator, `True` otherwise.

- **use_symmetry** (*bool*) – If enabled, only the symmetrically unique atoms are displaced and the remaining force constants are calculated using symmetry. **Default:** `True`
- **polar_phonon_splitting_parameters** (`PolarPhononSplittingParameters | None`) – If given polar phonon splitting will be included in the dynamical matrix. An optical_spectrum and born_effective_charge tasks will be added to the work flow. **Default:** `None`, i.e. polar phonon splitting is not included.
- **use_internal_coordinates** (*bool*) – Control if the eigensystem should be solved in internal coordinates for a *MoleculeConfiguration* **Default:** `False`
- **bond_list** (*list of list of type int*) – List of og sublists with bonding atomic indices that will be used to generate internal coordinates, e.g. `[[0, 1], [0, 2]]`. **Default:** `None`

acousticSumRule()

Returns
Return if the acoustic sum rule is invoked.

Return type
bool

atomicDisplacement()

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity with length unit

bondList()

Returns
The bond_list optionally supplied for an internal coordinate scheme calculation.

Return type
list | None

constrainElectrodes()

Returns
Boolean determining if the electrodes and electrode extensions are constrained in case of a DeviceConfiguration.

Return type
bool

constraints()

Returns
The list of constrained atoms.

Return type
list of int

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

`enablePolarPhononSplitting()`

Returns
Return true if the polar phonon splitting is enabled.

Return type
bool

`filename()`

Returns
The filename where the study object is stored.

Return type
str

`finiteDifferenceMethod()`

Returns
The finite difference scheme to use.

Return type
`Central | Forward`

`forceTolerance()`

Returns
The force tolerance

Return type
PhysicalQuantity with an energy per length squared unit e.g. Hartree/Bohr**2

`logFilenamePrefix()`

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

`maxInteractionRange()`

Returns
The maximum interaction range.

Return type
PhysicalQuantity with length unit

`nlinfo()`

Returns
Structured information about the DynamicalMatrix.

Return type
dict

`nlprint(`*stream=None*`)`

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
`stream` (*`python stream`*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

phononEigensystem(*q_point=None*, *constrained_atoms=None*, *enable_polar_phonon_splitting=None*, *D=None*)
Calculate the eigenvalues and eigenvectors for the dynamical matrix at a specified q-point.

Parameters

- **q_point** (*list of 3 floats*) – The fractional q-point to use. **Default:** `[0.0, 0.0, 0.0]`
- **constrained_atoms** (*list of ints.*) – List of atoms being constrained. The matrix elements from these atoms will not be included in the calculation of the eigensystem. **Default:** [] (empty list)
- **enable_polar_phonon_splitting** (*bool*) – Boolean controling if polar optical splitting should be included.

Returns
The eigenvalues and eigenvectors of the dynamical matrix.

Return type
2-tuple containing the eigenvalues and eigenvectors

polarPhononSplittingParameters()

Returns
The polar optical splitting parameters.

Return type
PolarOpticalSplittingParameters

processesPerDisplacement()

Returns
The number of processes per displacement.

Return type
int | `AllProcesses` | *ProcessesPerNode*

realSpaceDynamicalMatrix()

Returns the real space dynamical matrix. The shape of the matrix is (N, M), where N is the number of degrees of freedom (3 * number of atoms), and M = N * R, where R is the total number of repetitions.

Each subblock D[i*N:(i+1)*N, :] corresponds to the matrix elements between the center block (where the atoms have been displaced) and a neighbouring cell translated from the central cell by translations[i] in fractional coordinates.

Returns
The real-space dynamical matrix as a sparse matrix together with a list of translation vectors in fractional coordinates. The real space dynamical matrix is given in units of (meV / hbar)**2.

Return type
(scipy.sparse.csr_matrix, list of list(3) of integers)

reciprocalSpaceDynamicalMatrix(*q_point=None*)

Evaluate the reciprocal space dynamical matrix for a given q-point in reciprocal space.

Parameters
q_point (*list of floats*) – The fractional q-point to use. **Default:** `[0.0, 0.0, 0.0]`

Returns
The dynamical matrix for `q_point`.

Return type
PhysicalQuantity of units `(meV / hbar)**2`

repetitions()

Returns
The number of repetitions in the A, B, and C-directions for the supercell that is used in the finite displacement calculation.

Return type
list of three int.

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

symmetry()

Returns
True if the use of crystal symmetry to reduce the number of displacements is enabled.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the DynamicalMatrix study object.

useEquivalentBulk()

Returns
Boolean determining if a DeviceConfiguration is treated as a BulkConfiguration.

Return type
bool

useInternalCoordinates()

Returns
Return True if the use of internal coordinates is enabled.

Return type
bool

wignerSeitzScheme()

Returns
Boolean to control if the real space Dynamical Matrix should be extended according to the Wigner-Seitz construction.

Return type
bool

Usage Examples

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Calculate the *DynamicalMatrix* for a system repeated five times in the B direction and three times in the C direction.

```
dynamical_matrix = DynamicalMatrix(
    configuration,
    filename='DynamicalMatrix.hdf5',
    object_id='dynamical_matrix',
    repetitions=(1,5,3)
    )
dynamical_matrix.update()
```

When using `repetitions=Automatic`, the cell is repeated such that all atoms within a pre-defined, element-pair dependent interaction range are included.

```
dynamical_matrix = DynamicalMatrix(
    configuration,
    filename='DynamicalMatrix.hdf5',
    object_id='dynamical_matrix',
    repetitions=Automatic
)
dynamical_matrix.update()
```

The default number of repetitions i.e. `repetitions=Automatic` can be found before a calculation using the function *checkNumberOfRepetitions()*.

```
(nA, nB, nC)  = checkNumberOfRepetitions(configuration)
```

The maximum interaction range between two atoms can be specified manually, by using the `max_interaction_range` keyword.

```
dynamical_matrix = DynamicalMatrix(
    configuration,
```

(continues on next page)

(continued from previous page)

```
    filename='DynamicalMatrix.hdf5',
    object_id='dynamical_matrix',
    repetitions=Automatic,
    max_interaction_range=12.0*Ang,
)
dynamical_matrix.update()
```

Notes

The *DynamicalMatrix* is calculated using the finite difference method in a repeated cell, which is sometimes also referred to as frozen- phonon or super-cell method.

In the following, we denote the atoms in the unit cell by μ and the atoms in the repeated cell by i. Furthermore, denote the dynamical matrix elements, $D_{\mu\alpha,i\beta}$, where α, β are the Cartesian directions, i.e. x, y, z. A dynamical matrix element is given by

$$D_{\mu\alpha,i\beta} = \frac{1}{\sqrt{m_\mu m_i}} \frac{dF_{i\beta}}{dr_{\mu\alpha}},$$

where $F_{i\beta}$ is the force on atom i in direction β due to a displacement of atom μ in direction α.

The derivative is calculated by either `forward` or `central` finite differences, where in the following we will focus on the latter. Atom μ is displaced by Δr_α and $-\Delta r_\alpha$, and the changes in the force, $\Delta F_{i\beta}$ are calculated to approximate the dynamical matrix element

$$D_{\mu\alpha,i\beta} \approx \frac{1}{\sqrt{m_\mu m_i}} \frac{F_{i\beta}(\Delta r_\alpha) - F_{i\beta}(-\Delta r_\alpha)}{2\Delta r_\alpha}.$$

The default is to use `repetitions=Automatic`. In this case the cell is repeated such that all atoms within a pre-defined element-dependent distance from the atoms in the unit cell are included in the repeated cell. The repetitions used is written to the output file. These default interaction ranges are suitable for most systems.

However, if you are using long-ranged interactions, e.g. classical potentials with electrostatic interactions in *TremoloXCalculator*, it might be necessary to increase the number of repetitions.

For a 1D or 2D system, the unit cell should not be repeated in the confined directions. This is only discovered by the `repetitions=Automatic`, if there is enough vacuum in the unit cell in the directions that should not be repeated. That is typically 10-20 Å vacuum depending on the elements and their interaction ranges. Thus, for confined systems it is recommended to check the repetitions used and possibly use manual instead of automatic repetition.

DynamicalMatrix calculations using DFT or Semi-Empirical calculators have functionality to fully resume partially completed calculations by re-running the same script or reading the study object from file and calling `update()` on it. The study object will automatically detect which displacement calculations have already been carried out and only run the ones that are not yet completed. To ensure highest performance this functionality is not available for ATK-ForceField calculations.

Notes for DFT

In ATK-DFT the number of repetitions of the unit cell in super cell must ensure that the change in the force on atoms outside the super cell is zero for every atomic displacement in the center cell. An equivalent discussion of the number of k-points of the super cell and the number of repetitions can be found for *HamiltonianDerivatives* in the section *Notes for DFT*. Consider a system with e.g. x and y as confined directions and the k-point sampling of the unit cell $(1, 1, N_{\mathrm{C}})$, see Fig. 4.23 (a). Assume that the number of repetitions in the C-direction is known for the change in the force on atoms outside the super cell to be zero. Then the number of repetitions must be $(1, 1, \text{repetitions in C})$. Furthermore, the k-point sampling of the super cell becomes $(1, 1, \frac{N_{\mathrm{C}}}{\text{repetitions in C}})$.

Note: From QuantumATK-2019.03 onwards, the k-point sampling and density-mesh-cutoff will be **automatically** adapted to the given number of repetitions when setting up the super cell inside *DynamicalMatrix* and *HamiltonianDerivatives*. That means you can specify the calculator settings for the unit cell and use it with any desired number of repetitions in dynamical matrix and hamiltonian derivatives calculations.

When calculating the *DynamicalMatrix* with ATK-DFT, accurate results may require a higher precision than usual by increasing the `density_mesh_cutoff` in *NumericalAccuracyParameters* and decreasing the `tolerance` in *IterationControlParameters*, e.g.

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=150.0*Hartree
    )
iteration_control_parameters = IterationControlParameters(
    tolerance=1e-6
    )
```

Notes for the simplified supercell method (`use_wigner_seitz_scheme=True`)

The simplified supercell method is an approximation which allows to obtain the dispersion of vibrational eigenmodes with a force calculation of the unit cell only, i.e. having `repetitions=[1,1,1]` (the poor mans frozen phonon calculation). It is valid if the unit cell is large enough, i.e. if it accommodates 200 atoms and more. The convergence should be checked with respect to the number of atoms per unit cell or by a conventional frozen phonon calculation.

Idea of the simplified supercell method

In large unit cells, the force of *atom i* due to a displacement of *atom j* is small for distances of half the length of the unit cell vectors. Due to the translational invariance, a displaced *atom i* contributes from two sides to the force on *atom j*, which can be exactly decomposed if the distance between the atoms is large enough. As an example, we will look at a 1D chain with 6 atoms per unit cell (see Fig. 4.17).

All atoms are at their equilibrium positions, besides *atom 1* which is displaced along the x direction. The force on *atom 5*, for example, can be regarded as repulsive due to the displacement of *atom 1* in the unit cell with translation $T = 0$ along the x-axis. However, there is also an attractive contribution from the image of the displaced *atom 1* in the unit cell with translation $T = 1$. To decompose the two contributions we construct a Wigner-Seitz cell centered at the undisplaced position of *atom 1* and check the distance to the nearest neighbor representations of *atom 5*. The representation of *atom 5* inside the cell at $T = 0$ is not part of the Wigner-Seitz cell, and thus this contribution is neglected. In contrast, the representation of *atom 5* at $T = -1$ is inside the Wigner-Seitz cell and we keep this contribution. If an atom is at the face (like *atom 4*), the edge or a vertex of the Wigner-Seitz cell, the force is included weighted by the inverse multiplicity of this Wigner-Seitz grid point. This construction is repeated for all atoms inside the unit cell. In this way it is possible to approximately calculate the entries of the *DynamicalMatrix* for all repetitions of the cell from $T = -1$ to $T = 1$ and thereby get the dispersion, by only calculating the forces in a single unit cell.

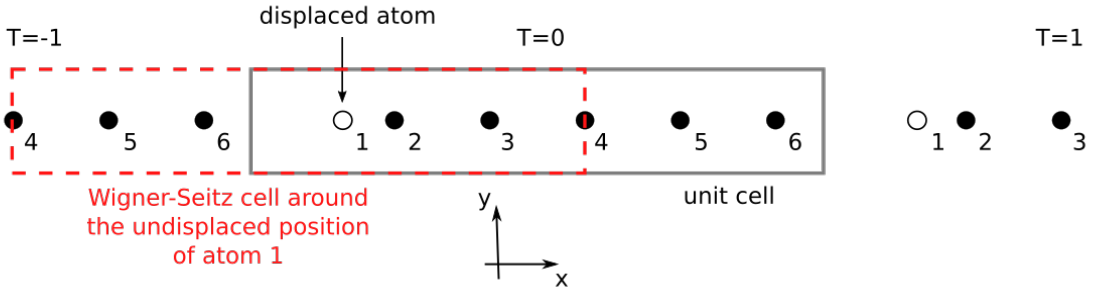


Fig. 4.17: A 1D chain with 6 atoms per unit cell with atom 1 being displaced. The periodically-repeated atoms to the left ($T = -1$) and to the right ($T = 1$) are indicated. The Wigner-Seitz cell centered at the undisplaced position of atom 1 is drawn red / dashed.

Note: The Wigner-Seitz construction ensures that the phonon spectrum at the Γ-point is preserved. Hence, a separate Γ-point calculation is not necessary.

Notes for polar phonon splitting

In polar materials the covalency will change during ionic displacements since a macroscopic electric field is generated. This gives rise to an extra force constant term in the dynamical matrix. The full dynamical matrix is $\mathbf{D}(\mathbf{q}) = \mathbf{D}^{FC}(\mathbf{q}) + \mathbf{D}^{NA}(\mathbf{q})$. Here $\mathbf{D}^{FC}(\mathbf{q})$ is the usual force constant dynamical matrix discussed above and the second term $\mathbf{D}^{NA}(\mathbf{q})$ is due to polar phonon interaction. The non-analytic (NA) contribution from the macroscopic field to the dynamical matrix is:

$$\mathbf{D}^{NA}_{i\mu j\nu}(\mathbf{q}) = \frac{e}{\epsilon_0 \Omega \sqrt{M_i M_j}} \frac{[\mathbf{Z}(i)\cdot\hat{\mathbf{q}}]_\mu [\mathbf{Z}(j)\cdot\hat{\mathbf{q}}]_\nu}{\hat{\mathbf{q}}\epsilon^\infty\hat{\mathbf{q}}} .$$

Here $\mathbf{Z}(i)$ is the Born effective charge tensor (a matrix for atom i), $\hat{\mathbf{q}} = \mathbf{q}/|\mathbf{q}|$ is a unit vector, M_i is the mass of atom i, Ω is the unit cell volume, e is the electron charge and ϵ_0 is the vacuum permittivity. The term non-analytical refers to the fact that it approaches different values from different $\mathbf{q}$-space directions as $q \to 0$. It is obtained from the *BornEffectiveCharge* calculation and the high-frequency dielectric tensor ϵ^∞ obtained from the *OpticalSpectrum*.

The Dynamical Matrix study object will automatically include a calculation of the BornEffectiveCharges and OpticalSpectrum or it can be given by the user from an existing calculation. To include polar phonon splitting in the dynamical matrix study object one has to give `PolarPhonon_splitting_parameters` as input:

```
dynamical_matrix = DynamicalMatrix(
    configuration,
    filename='DynamicalMatrix.hdf5',
    object_id='dynamical_matrix',
    repetitions=(1,5,3),
    polar_phonon_splitting_parameters=pps_parameters,
)
dynamical_matrix.update()
```

Here the polar polar_phonon_splitting_parameters is a container of polar phonon splitting parameters. It can either be setup by defaults

```
pps_parameters = PolarPhononSplittingParameters()
```

Or it can be set with specific user-defined variables for optical spectrum and Born effective charges.

```
pps_parameters = PolarPhononSplittingParameters(
    kpoints=MonkhorstPackGrid(25, 25, 25),
    broadening=0.1 * eV,
    bands_below_fermi_level=35,
    bands_above_fermi_level=35,
    kpoints_a=MonkhorstPackGrid(19, 9, 9),
    kpoints_b=MonkhorstPackGrid(9, 19, 9),
    kpoints_c=MonkhorstPackGrid(9, 9, 19),
    atomic_displacement=0.01 * Ang,
)
```

Finally it can be defined from existing calculations of the *OpticalSpectrum* and *BornEffectiveCharge*:

```
pps_parameters = PolarPhononSplittingParameters(
    optical_spectrum=optical_spectrum,
    born_effective_charge=born_effective_charge,
)
```

This last option enables the combination of force constants obtained from a classical calculator with polar phonon splitting obtained from ATK-DFT.

Notes for internal coordinates

For molecules, it can be beneficial to shift the dynamical matrix into an internal coordinate basis when conducting studies of or including the vibrational modes of a *MoleculeConfiguration*. A (potentially) redundant internal coordinate basis is generated by converting the bond connectivity of the molecule into unique stretch, bend, and dihedral internal coordinates, from which a non-redundant set of $3N - 6$ (in the general case) linear combinations of internal coordinates are derived. This scheme thus allows a non-redundant description of the vibrational modes and results in the same number of vibrational modes as the number of degrees of freedom in the molecule.

The internal coordinate scheme assumes that the bond connectivity of the system connects all atoms in the configuration, so that at least $3N - 6$ unique internal coordinates are generated. If fewer bonds are present, or if only a smaller number of bonds can be automatically discovered, the internal coordinate scheme will yield fewer vibrational modes than the degrees of freedom in the molecule. The `use_internal_coordinates` argument should therefore only be activated when **all** bonds in a molecule can be automatically discovered with the `findBonds()` method of a *MoleculeConfiguration* object or are set manually on the configuration. This can be done from the Builder or in scripting. Alternatively, a `bond_list` argument can be passed to a *DynamicalMatrix* object with the desired bonding behaviour, in which all atom indices are included and interconnected in a single molecule fragment.

DynamicalMatrixParameters

class DynamicalMatrixParameters(*calculator=None*, *repetitions=None*, *atomic_displacement=None*, *acoustic_sum_rule=None*, *finite_difference_method=None*, *constraints=None*, *constrain_electrodes=None*, *use_equivalent_bulk=None*, *max_interaction_range=None*, *force_tolerance=None*, *processes_per_displacement=None*, *log_filename_prefix=None*, *use_wigner_seitz_scheme=None*, *use_symmetry=None*, *polar_phonon_splitting_parameters=None*, *use_internal_coordinates=None*, *bond_list=None*)

Parameters

- **calculator** (`Calculators`) – The calculator to be used in the dynamical matrix calculations. **Default:** The calculator attached to the configuration.
- **repetitions** (`Automatic` | list of ints) – The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. Each repetition value must be odd. **Default:** `Automatic`
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01 * Angstrom`
- **acoustic_sum_rule** (*bool*) – Control if the acoustic sum rule should be invoked. **Default:** `True`
- **finite_difference_method** (`Forward` | `Central`) – The finite difference scheme to use. **Default:** `Central`
- **constraints** (*list of type int*) – List of atomic indices that will be constrained, e.g. `[0, 2, 10]`. **Default:** None
- **constrain_electrodes** (*bool*) – Control if the electrodes and electrode extensions should be constrained in case of a *DeviceConfiguration*. **Default:** `False`
- **use_equivalent_bulk** (*bool*) – Control if a *DeviceConfiguration* should be treated as a *BulkConfiguration*. **Default:** `True`
- **max_interaction_range** (*PhysicalQuantity* of type length) – Set the maximum range of the interactions. **Default:** All atoms are included
- **force_tolerance** (*PhysicalQuantity* of type energy per length squared) – All force constants below this value will be truncated to zero. **Default:** `1e-8 * Hartree/Bohr**2`
- **processes_per_displacement** (int | *ProcessesPerNode*) – The number of processes assigned to calculating a single displacement. **Default:** 1 for force-field calculators, *ProcessesPerNode* otherwise.
- **log_filename_prefix** (*str or None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. If a classical calculator is used, no per-displacment log files will be generated. **Default:** `"forces_displacement_"`
- **use_wigner_seitz_scheme** (*bool*) – Control if the real space Dynamical Matrix should be extended according to the Wigner Seitz construction. `use_wigner_seitz_scheme=True`. **Default:** `False` for ForceField calculator, `True` otherwise.
- **use_symmetry** (*bool*) – If enabled, only the symmetrically unique atoms are displaced and the remaining force constants are calculated using symmetry. **Default:** `True`
- **polar_phonon_splitting_parameters** (`PolarPhononSplittingParameters` | `None`) – If given polar phonon splitting will be included in the dynamical matrix. An optical_spectrum and born_effective_charge tasks will be added to the work flow. **Default:** `None`, i.e. polar phonon splitting is not included.

- **use_internal_coordinates** (*bool*) – Control if the eigensystem should be solved in internal coordinates for a *MoleculeConfiguration* **Default:** `False`
- **bond_list** (*list of list of type int*) – List of og sublists with bonding atomic indices that will be used to generate internal coordinates, e.g. `[[0, 1], [0, 2]]`. **Default:** `None`

acousticSumRule()

Returns
Control if the acoustic sum rule should be invoked.

Return type
bool

atomicDisplacement()

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity of type length

bondList()

Returns
List of og sublists with bonding atomic indices that will be used to generate internal coordinates, e.g. `[[0, 1], [0, 2]]`.

Return type
list of list of type int

calculator()

Returns
The calculator to use for the dynamical matrix. If None the configuration calculator is used.

Return type
Calculator

constrainElectrodes()

Returns
Control if the electrodes and electrode extensions should be constrained in case of a *DeviceConfiguration*.

Return type
bool

constraints()

Returns
List of atomic indices that will be constrained, e.g. `[0, 2, 10]`.

Return type
list of type int

finiteDifferenceMethod()

Returns
The finite difference scheme to use.

Return type
`Forward|Central`

`forceTolerance()`

Returns
All force constants below this value will be truncated to zero.

Return type
PhysicalQuantity of type energy per length squared

`logFilenamePrefix()`

Returns
Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. If a classical calculator is used, no per-displacment log files will be generated.

Return type
str or None

`maxInteractionRange()`

Returns
Set the maximum range of the interactions.

Return type
PhysicalQuantity of type length

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`polarPhononSplittingParameters()`

Returns
If given polar phonon splitting will be included in the dynamical matrix. An optical_spectrum and born_effective_charge tasks will be added to the work flow.

Return type
`None`, i.e. polar phonon splitting is not included.

`processesPerDisplacement()`

Returns
The number of processes assigned to calculating a single displacement.

Return type
1 for force-field calculators, *ProcessesPerNode* otherwise.

`repetitions()`

Returns
The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. Each repetition value must be odd.

Return type
`Automatic` | list of ints

setAcousticSumRule(*acoustic_sum_rule*)

Parameters
acoustic_sum_rule (*bool*) – Control if the acoustic sum rule should be invoked.

setAtomicDisplacement(*atomic_displacement*)

Parameters
atomic_displacement (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method.

setBondList(*bond_list*)

Parameters
bond_list (*list of list of type int*) – List of og sublists with bonding atomic indices that will be used to generate internal coordinates, e.g. `[[0, 1], [0, 2]]`. **Default:** `None`

setCalculator(*calculator*)

Parameters
calculator (`Calculators`) – The calculator to be used in the dynamical matrix calculations.

setConstrainElectrodes(*constrain_electrodes*)

Parameters
constrain_electrodes (*bool*) – Control if the electrodes and electrode extensions should be constrained in case of a *DeviceConfiguration*.

setConstraints(*constraints*)

Parameters
constraints (*list of type int*) – List of atomic indices that will be constrained, e.g. `[0, 2, 10]`.

setFiniteDifferenceMethod(*finite_difference_method*)

Parameters
finite_difference_method (`Forward` | `Central`) – The finite difference scheme to use.

setForceTolerance(*force_tolerance*)

Parameters
force_tolerance (*PhysicalQuantity* of type energy per length squared) – All force constants below this value will be truncated to zero.

setLogFilenamePrefix(*log_filename_prefix*)

Parameters
log_filename_prefix (*str or None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. If a classical calculator is used, no per-displacment log files will be generated.

setMaxInteractionRange(*max_interaction_range*)

Parameters
max_interaction_range (*PhysicalQuantity* of type length) – Set the maximum range of the interactions.

setPolarPhononSplittingParameters(*polar_phonon_splitting_parameters*)

Parameters
polar_phonon_splitting_parameters (`PolarPhononSplittingParameters` | `None`) – If given polar phonon

splitting will be included in the dynamical matrix. An optical_spectrum and born_effective_charge tasks will be added to the work flow.

setProcessesPerDisplacement(*processes_per_displacement*)

Parameters
processes_per_displacement (int | `AllProcesses` | *ProcessesPerNode*) – The number of processes assigned to calculating a single displacement.

setRepetitions(*repetitions*)

Parameters
repetitions (`Automatic` | list of ints) – The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. Each repetition value must be odd.

setUseEquivalentBulk(*use_equivalent_bulk*)

Parameters
use_equivalent_bulk (*bool*) – Control if a *DeviceConfiguration* should be treated as a *BulkConfiguration*.

setUseInternalCoordinates(*use_internal_coordinates*)

Parameters
use_internal_coordinates (*bool*) – Control if the eigensystem should be solved in internal coordinates for a *MoleculeConfiguration*

setUseSymmetry(*use_symmetry*)

Parameters
use_symmetry (*bool*) – If enabled, only the symmetrically unique atoms are displaced and the remaining force constants are calculated using symmetry.

setUseWignerSeitzScheme(*use_wigner_seitz_scheme*)

Parameters
use_wigner_seitz_scheme (*bool*) – Control if the real space Dynamical Matrix should be extended according to the Wigner Seitz construction. `use_wigner_seitz_scheme=True`.

uniqueString()
Return a unique string representing the state of the object.

useEquivalentBulk()

Returns
Control if a *DeviceConfiguration* should be treated as a *BulkConfiguration*.

Return type
bool

useInternalCoordinates()

Returns
Control if the eigensystem should be solved in internal coordinates for a *MoleculeConfiguration*

Return type
`False`

useSymmetry()

Returns
If enabled, only the symmetrically unique atoms are displaced and the remaining force constants are calculated using symmetry.

Return type
bool

`useWignerSeitzScheme()`

Returns
Control if the real space Dynamical Matrix should be extended according to the Wigner Seitz construction. `use_wigner_seitz_scheme=True`.

Return type
bool

EAMfssetflPotential

`class EAMfssetflPotential(`*file*, *splineType=None*, *tags=None*`)`

Constructor of the potential.

Parameters

- **file** (*str*) – The name of the file that contains the EAM parameters. The file must be in the EAM/FS setfl format.
- **splineType** – The used spline type. Either EAMPotential.akima, EAMPotential.fivepoint, EAMPotential.bessel or EAMPotential.natural.
- **tags** (*None or sequence of str*) – A list of tags (or None) that will be used for all particle types inside the EAM potential file.

`classmethod getAllParameterNames()`

Return the names of all used parameters as a list.

`getAllParameters()`

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

`static getDefaults()`

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

`getParameter(`*parameterName*`)`

Get the current value of the parameter parameterName.

`setParameter(`*parameterName*, *value*`)`

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define an EAM Finnis-Sinclair potential for HCP titanium by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration for HCP titanium
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(2.9506*Angstrom, 4.6835*Angstrom)

# Define elements
elements = [Titanium, Titanium]

# Define coordinates
fractional_coordinates = [[ 0.33333333, 0.66666667, 0.25],
                          [ 0.66666667, 0.33333333, 0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='EAMFS_Ti_1992')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='Ti',
                                          mass=47.867*atomic_mass_unit))

# Add the EAM potential to the potential set
potentialSet.addPotential(EAMfssetflPotential(file='Ti-Ackland-1992.eam.fs',
                                              splineType='natural'))

# Create a TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

Notes

The *EAMfssetflPotential* is a many-body potential implementing the **embedded atom method (EAM)** in the **Finnis-Sinclair (FS)** variant. It is primarily designed for metals and alloys[1], and it is similar to the *EAMsetflPotential* and the *SuttonChenPotential*.

The potential energy is defined as

$$V = \sum_{i<j} \phi_{ij}(r_{ij}) + \sum_i \left(\sum_{j \neq i} \rho_{ij}(r_{ij}) \right) ,$$

where F is the embedding function, ρ is a generalized density arising from the neighbor atoms, and ϕ is a repulsive pair potential.

In contrast to the *EAMsetflPotential*, the density function ρ does not depend on the neighbor type j alone, but is pair-dependent instead.

These functions are usually given in tabulated form and read from the file specified by the `file` argument.

EAMsetflPotential

class EAMsetflPotential(*file*, *splineType=None*, *tags=None*)

Constructor of the potential.

Parameters

- **file** (*str*) – The name of the file that contains the EAM parameters. The file must be in the EAM/alloy setfl format.
- **splineType** – The used spline type. Either EAMPotential.akima, EAMPotential.fivepoint, EAMPotential.bessel or EAMPotential.natural.
- **tags** (*None or sequence of str*) – A list of tags (or None) that will be used for all particle types inside the EAM potential file.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

[1] M.ăS. Daw and M.ăI. Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Phys. Rev. B*, 29:6443–6453, Jun 1984. doi:10.1103/PhysRevB.29.6443.

Usage Examples

Define an EAM potential for a copper by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration for FCC copper
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(3.61496*Angstrom)

# Define elements
elements = [Copper]

# Define coordinates
fractional_coordinates = [[ 0.0, 0.0, 0.0]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='EAM_Cu_2001b')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='Cu',
                                          mass=63.546*atomic_mass_unit))

# Add the EAM potential to the potential set
potentialSet.addPotential(EAMsetflPotential(file='Cu-Mishin-2001.eam.alloy',
                                            splineType='natural'))

# Create a TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

Notes

The *EAMsetflPotential* is a many-body potential implementing the **embedded atom method (EAM)**. It is primarily designed for metals and alloys[1], and it is similar to the *EAMfssetflPotential* and the *SuttonChenPotential*.

The potential energy is defined as

$$V = \sum_{i<j} \phi_{ij}(r_{ij}) + \sum_{i} \left(\sum_{j \neq i} \rho_j(r_{ij}) \right) ,$$

where F is the embedding function, ρ is a generalized density arising from the neighbor atoms, and ϕ is a repulsive pair potential.

These functions are usually given in tabulated form and read from the file specified by the `file` argument.

EEMOption

class EEMOption(*chargeNeutrality=False*)

Constructor of the option.

Parameters
chargeNeutrality (*bool*) – Whether or not charge neutrality should be enforced.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

[1] M.ăS. Daw and M.ăI. Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Phys. Rev. B*, 29:6443–6453, Jun 1984. doi:10.1103/PhysRevB.29.6443.

EEMSinglePotential

class EEMSinglePotential(*particleType*, *chi*, *J*, *a=PhysicalQuantity(0.0, Ang)*)

Constructor of the potential.

Parameters

- **particleType** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **chi** (*PhysicalQuantity of type energy / charge*) – Potential parameter.
- **J** (*PhysicalQuantity of type 1 / length*) – Potential parameter.
- **a** (*PhysicalQuantity of type length*) – Potential parameter.

static generateFromD4(*particleType*, *Z*, *mCN*, *MaxCN=None*)

Create an EEMSinglePotential based on Z and the modified coordination number mCN using the D4 parameters and formulae from J. Chem. Phys. 150, 154122 (2019); doi: 10.1063/1.5090222 where chi = EN - kappa*sqrt(mCN) for MaxCN = None and otherwise chi = EN - kappa*sqrt(log(1+exp(MaxCN))-log(1+exp(MaxCN-mCN))) according to Phys. Chem. Chem. Phys., 2020, 22, 8499-8512; DOI: 10.1039/D0CP00502A

Parameters

- **Z** (*integer*) – Atomic number
- **mCN** (*real*) – Modified coordination number

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

ELPASolver

class ELPASolver(*bands_above_fermi_level=None*, *processes_per_kpoint=None*, *optimize_for_speed_over_memory=None*)

Calculate the density matrix by direct diagonalization.

Parameters

- **bands_above_fermi_level** (int | `Automatic` | `All`) – The number of bands above the Fermi level. Must be either a non-negative integer, `All` or `Automatic`. When set to `All` the total number of bands will be equal to the total number of basis functions. When set to `Automatic` an adaptive method will be used. For most systems, `Automatic`

provides a major speed-up as compared to `All`. If the number of states above the Fermi level changes significantly from one diagonalization step to another, the adapative method may lead to slowing down of the calculation. **Default:** `Automatic`

- **processes_per_kpoint** (*int*) – The number of processes to use per kpoint. Must be a positive integer. **Default:** The number will be determined automatically from the total number of k-points and processes such as to keep the number as small as possible. One may set this number manually in order to reduce the memory requirements for each process.
- **optimize_for_speed_over_memory** (*bool*) – Whether to optimize the diagonalization for speed over memory. If True, the solver will determine the most optimal algorithm in terms of speed. Note that optimizing for speed can increase the memory usage by up to 30 %. Note also that this flag only has an effect if `processes_per_kpoint=1`, or the automatic value for `processes_per_kpoint=1` turns out to be 1. It is intended to be used for medium-sized to large systems with many k-points. This option has no effect for gpu-accelerated diagonalization. **Default:** False

bandsAboveFermiLevel()

Returns
The initial guess for the number of bands above the Fermi level.

Return type
int | `All`

optimizeForSpeedOverMemory()

Returns
Whether the diagonalization should be optimized for speed over memory.

Return type
bool

processesPerKpoint()

Returns
Number of processes per kpoint.

Return type
int

uniqueString()
Return a unique string representing the state of the object.

Notes

ELPASolver is an alias for *DiagonalizationSolver*.

EffectiveBandstructure

class EffectiveBandstructure(*configuration*, *primitive_configuration*, *route=None*, *points_per_segment=None*, *kpoints=None*, *energies=None*, *bands_above_fermi_level=None*, *projections=None*, *symmetrize=None*, *primitive_cell_matching_tolerance=None*, *processes_per_kpoint=None*)

Analysis class for calculating the effective band structure of a bulk supercell configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration of the supercell with attached calculator for which to calculate the effective band structure.

•**primitive_configuration** (*BulkConfiguration*) – The primitive cell configuration of the underlying crystal for which the band structure should be calculated for.

•**route** (*list of str*) – The route to take through the Brillouin zone of the primitive cell as a list of symmetry points identifiers, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `kpoints`. **Default:** Unit cell dependent route.

•**points_per_segment** (*int*) – The number of points per segment of the route. **Default:** `20`.

•**kpoints** (*list*) – A list of fractional coordinates of 3-dimensional k-points at which to calculate the effective band structure, e.g. `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. The shape should be (*K*, 3) where *K* is the number of k-points. This option is mutually exclusive to `route`, and `points_per_segment`. **Default:** Unit cell dependent route.

•**energies** (*PhysicalQuantity*) – The energies where the spectral function should be be evaluated. **Default:** A list of 501 regularly spaced energies from -10 eV to 10 eV.

•**bands_above_fermi_level** (int | `All`) – The number of bands above the Fermi level (of the supercell configuration) to include in the effective band structure. If `All` the number of bands will be taken as that used in the ground state calculation. **Default:** `All`

•**projections** (list of *Projection* | *Projection* | *ProjectionGenerator* | `NoProjection`) – The projections used for calculating the weights. **Default:** `NoProjection`.

•**symmetrize** (*bool*) – Whether to calculate the symmetrized effective band structure. If `True` the effective band structure will be calculated for all symmetry equivalent k-points and the end result will be averaged. The symmetries used are those of the primitive configuration. If `False` the effective band structure will not be symmetrized and it may thus depend on the specific k-point path chosen. **Default:** `False`

•**primitive_cell_matching_tolerance** (*float*) – Tolerance for the transformation matrix between the primitive configuration and the supercell configuration. **Default:** 1e-3

•**processes_per_kpoint** (*int*) – The number of processes to use per k-point. If `None` the same number of processes per k-point as used for the ground state calculation will be used. **Default:** `None`

allBandEnergies(*spin=None*)

Parameters
spin (`Spin.All` | `Spin.Up` | `Spin.Down`) – The spin the bandstructure should be returned for. **Default:** `Spin.All`

Returns
The band energies of the (super cell) configuration for a given spin. The shape is (*K*, *B*) where *K* is the number of k-points and *B* is the number of bands.

Return type
`PhysicalQuantity` of type energy

energies()

Returns
The energies at which the effective bandstructure is calculated.

Return type
`PhysicalQuantity` of type energy

energyZero()

The energy zero. It is set to the spin-averaged FermiLevel.

Returns
The energy zero.

Return type
`PhysicalQuantity` of type energy

evaluate(*spin=None*)

Parameters
spin (Spin.All | Spin.Up | Spin.Down) – The spin the effective bandstructure should be returned for. **Default:** Spin.All

Returns
The effective bandstructure as a band density for a given spin. The shape is (*K*, *E*) or (*S*, *K*, *E*) where *S* is the number of spins, *K* is the number of k-points, and *E* is the number of energies.

Return type
numpy.array

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.All

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

fermiTemperature()

Returns
The Fermi temperature used in the calculation of the effective bandstructure.

Return type
PhysicalQuantity of type temperature

kpoints()

Returns
The list of 3-dimensional fractional k-points at which the energies of the bands are calculated. The shape is (*K*, 3) where *K* is the number of k-points.

Return type
list of lists of floats

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveConfiguration()

Returns
The primitive configuration which defines the band path.

Return type
BulkConfiguration

projectedEffectiveBandstructures(*spin=None*)

Parameters
spin (Spin.All | Spin.Up | Spin.Down | Spin.Sum) – The spin the effective bandstructure should be returned for.
Default: Spin.All

Returns
The effective bandstructure as a band density for a given spin for each projection. The shape is (*P*, *K*, *E*) or (*P*, *S*, *K*, *E*) where *P* is the number of projections, *S* is the number of spins, *K* is the number of k-points, and *E* is the number of energies.

Return type
numpy.array

projections()

Returns
The projections.

Return type
list of *Projection*

route()

Returns
The route through the Brillouin-zone as a list of symmetry points of the unit cell.

Return type
list of str

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

superCellConfiguration()

Returns
The (super cell) configuration for which the effective bandstructure is calculated.

Return type
BulkConfiguration

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the effective bandstructure of an InGaAs alloy.

```
from AddOns.AlloyPlugins.SubstitutionalAlloy.SubstitutionalAlloyWidget import␣
↪substitutionalAlloy

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

#---------------------------------------
# Basis Set
#---------------------------------------

basis_set = [
    LDABasis.Gallium_SingleZetaPolarized,
    LDABasis.Arsenic_SingleZetaPolarized,
    LDABasis.Indium_SingleZetaPolarized,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------

exchange_correlation = MGGA.TB09LDA(c=1.01)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(3, 3, 3),
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

# Set up lattice with exp. lattice const. of In0.53Ga0.47As
lattice = FaceCenteredCubic(5.8687*Angstrom)

# Define elements
elements = [Indium, Arsenic]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
primitive_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# Attach calculator to primitive bulk configuration
primitive_configuration.setCalculator(calculator)

# Repeat InAs
super_inas = primitive_configuration.repeat(3,3,3)

# List of indices corresponding to Indium atoms
indium_indices = []
symbols = super_inas.symbols()

for index,atom in enumerate(symbols):
    if atom=='In':
        indium_indices.append(index)

# Call the substitutional alloy generator.
super_ingaas = substitutionalAlloy(super_inas, indium_indices, 'fixed_fraction', 47,␣
↪Gallium)

# Set calculator and update.
super_ingaas.setCalculator(calculator)
super_ingaas.update()
nlsave('InGaAs_3x3x3.nc', super_ingaas)

# ------------------------------------------------------------
# EffectiveBandstructure
# ------------------------------------------------------------
effective_bandstructure = EffectiveBandstructure(
    super_ingaas,
    primitive_configuration,
    energies = numpy.linspace(-2,2.75,501)*eV,
    route=['L','G','X'],
    points_per_segment=101,
    )
nlsave('InGaAs_3x3x3.nc', effective_bandstructure)
```

Notes

The implementation of effective band structure in QuantumATK follows the works by Popescu and Zunger[1],[2] and Haverkort *et al.*[3]. From the bandstructure of a super cell containing many bands one can unfold the bandstructure to only contain the bands corresponding to the primitive cell. In the unfolding procedure, one calculates a spectral weight as $|\langle e^{i\mathbf{k}\cdot\mathbf{r}}|\psi_{j,\mathbf{K}}\rangle|$, with $|\psi_{j,\mathbf{K}}\rangle$ being an eigenstate of the super cell at k-point K . If the super cell is simply a copy of a simple unit cell (e.g. a super cell of InAs) the spectral weights will either be 0 or 1, and the band unfolding can be done explicitly. However, in the case of a disordered alloy as e.g. In0.53Ga0.47As the band weights can be anywhere in the interval [0;1].

[1] V.ăPopescu and A.ăZunger. Effective Band Structure of Random Alloys. *Phys. Rev. Lett.*, 104:236403, Jun 2010. doi:10.1103/PhysRevLett.104.236403.

[2] V.ăPopescu and A.ăZunger. Extracting E versus P k effective band structure from supercell calculations on alloys and impurities. *Phys. Rev. B*, 85:085201, Feb 2012. doi:10.1103/PhysRevB.85.085201.

[3] M.ăW. Haverkort, I.ăS. Elfimov, and G.ăA. Sawatzky. Electronic structure and self energies of randomly substituted solids using density functional theory and model calculations. *arXiv*, pages 1109.4036, 2011. URL: http://arxiv.org/abs/1109.4036.

It is currently a requirement that the configuration can be divided into an integer number of smaller unit cells with the size given by the primitive_configuraiton. Each of these smaller cells must have the same number of basis functions. It is currently thus not possible to e.g. replace some In atoms with Hydrogen atoms, both being described with e.g. a DoubleZetaPolarized basis set, since these will have a different number of basis functions. It is, however, possible to replace an Indium atom with a Gallium since they will have the same number of basis functions.

When calculating the effective band structure of a random alloy one should perform a sample average by calculating the effective bandstructure for many different configurations and averaging the results. This procedure is described in the tutorial *Effective band structure of random alloy InGaAs*.

EffectiveMass

class EffectiveMass(*configuration*, *symmetry_label=None*, *kpoint_cartesian=None*, *kpoint_fractional=None*, *bands_below_fermi_level=None*, *bands_above_fermi_level=None*, *band_index=None*, *stencil_order=None*, *delta=None*, *direction_cartesian=None*, *direction_fractional=None*, *method=None*)

Constructor for the effective mass object.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the effective mass.
- **symmetry_label** (*str*) – The k-point (as a symmetry point) at which to calculate the effective mass. This option is mutually exclusive to `kpoint_cartesian` and `kpoint_fractional`. **Default:** `'G'` (Gamma-point)
- **kpoint_cartesian** (*PhysicalQuantity* of type inverse length) – The k-point (in Cartesian coordinates) at which to calculate the effective mass. This option is mutually exclusive to `symmetry_label` and `kpoint_fractional`. **Default:** `[0.0, 0.0, 0.0] * Angstrom**-1`
- **kpoint_fractional** (*list(3) of floats*) – The k-point (in fractional coordinates) at which to calculate the effective mass. This option is mutually exclusive to kpoint_cartesian and symmetry_label. **Default:** `[0.0, 0.0, 0.0]`
- **bands_below_fermi_level** (*int*) – The number of bands below the Fermi level per principal spin channel to include. This option is mutually exclusive to `band_index`. **Default:** `1`
- **bands_above_fermi_level** (*int*) – The number of bands above the Fermi level per principal spin channel to include. This option is mutually exclusive to `band_index`. **Default:** `1`
- **band_index** (*int*) – The band index at which to calculate the effective mass. This option is mutually exclusive to `bands_below_fermi_level` and `bands_above_fermi_level`.
- **stencil_order** (*int*) – The number of points to use in the second order derivative stencil. **Default:** `5`
- **delta** (*PhysicalQuantity* of type inverse length) – The distance between neighboring points in the stencil. **Default:** `0.001 * Angstrom**-1`
- **direction_cartesian** (*PhysicalQuantity* of type inverse length) – Direction in Cartesian coordinates along which to calculate the effective mass. This option is mutually exclusive to `direction_fractional`. **Default:** `[1, 0, 0] * Angstrom**-1`
- **direction_fractional** (*list(3) of floats*) – Direction in fractional coordinates along which to calculate the effective mass. This option is mutually exclusive to `direction_cartesian`.
- **method** (`Numerical` | `Analytical`) – The method for calculating the effective mass. The option `Analytical` is not supported by *PlaneWaveCalculator*. **Default:** `Numerical`

energies()

Returns
The band energies.

Return type
PhysicalQuantity of type energy

`evaluate`(*spin=None*, *band=None*)

Return the effective mass.

Parameters

- **spin** (`Spin.Up | Spin.Down | Spin.All`) – The spin for which to evaluate the effective mass. For non-collinear and spin-orbit calculations use `Spin.All`. **Default:** `Spin.All`
- **band** (*int*) – The band for which to evaluate the effective mass. **Default:** All bands

Returns
The effective mass.

Return type
PhysicalQuantity with the unit `electron_mass`

`kPoint`()

Returns
The k-point for which the effective mass was calculated.

Return type
`numpy.array`

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString`()

Return a unique string representing the state of the object.

Usage Examples

The effective mass of electrons and holes is related to the local curvature of the electronic band energy, $E(k)$:

$$m_{\text{eff}} \propto \frac{1}{\frac{\partial^2 E(k)}{\partial k^2}}.$$

A finite-difference (FD) representation of derivatives of $E(k)$ is used to compute the effective mass from band structures. The FD approximation for the local derivative of a function $f(x)$ is basically derived from a Taylor expansion of $f(x)$ around the central point, x. The nth derivative is therefore approximated as a sum over function values multiplied by a weight:

$$\frac{\partial^n f(X_0)}{\partial X^n} = f^{(0)}(X_0) \approx \sum_i w_i^{(n)} f(x_1)/h^2$$

The 5-point FD stencil uses function values in the five points x, $x \pm h$, and $x \pm 2h$ to approximate the second-order derivative, $f''(x)$. This is *exact* up to and including $O(h^4)$ contributions to the curvature (for $h < 1$). Beyond that, accuracy improves as the spacing, h, gets smaller.

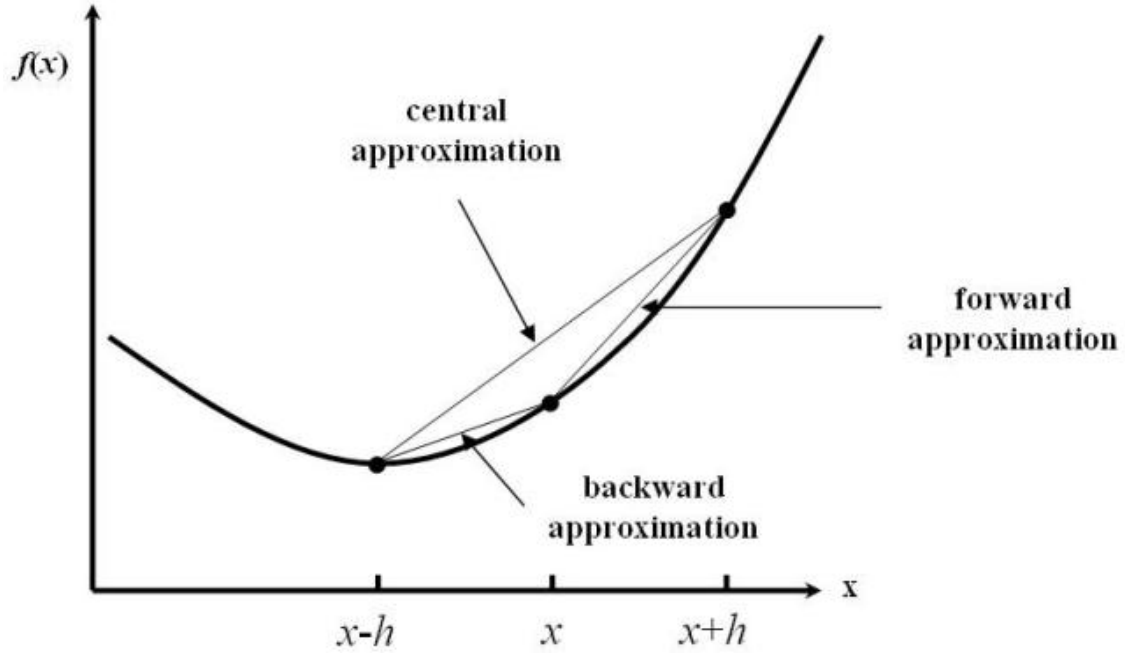


Fig. 4.18: Finite-difference stencils: Numerical approximations for local derivatives of continuous functions.

The FD method is in general much more accurate than a simple parabolic fit to band energies. The QuantumATK default is the 5-point FD central stencil with a distance (`delta`) of 0.001 Å^{-1} between each point. This gives very accurate results in most cases. The FD accuracy can in principle be improved by increasing `stencil_order` and decreasing `delta`, but the latter should not be so small that numerical noise starts dominating the error. The figure below shows the error on computed effective masses for some realistic combinations of `stencil_order` and `delta`. For more information on 5-point stencils, see the relevant Wikipedia entry.

Calculate the effective mass in the silicon conduction band minimum:

```
# -----------------------------------------------------------
# Bulk configuration
# -----------------------------------------------------------
```

(continues on next page)

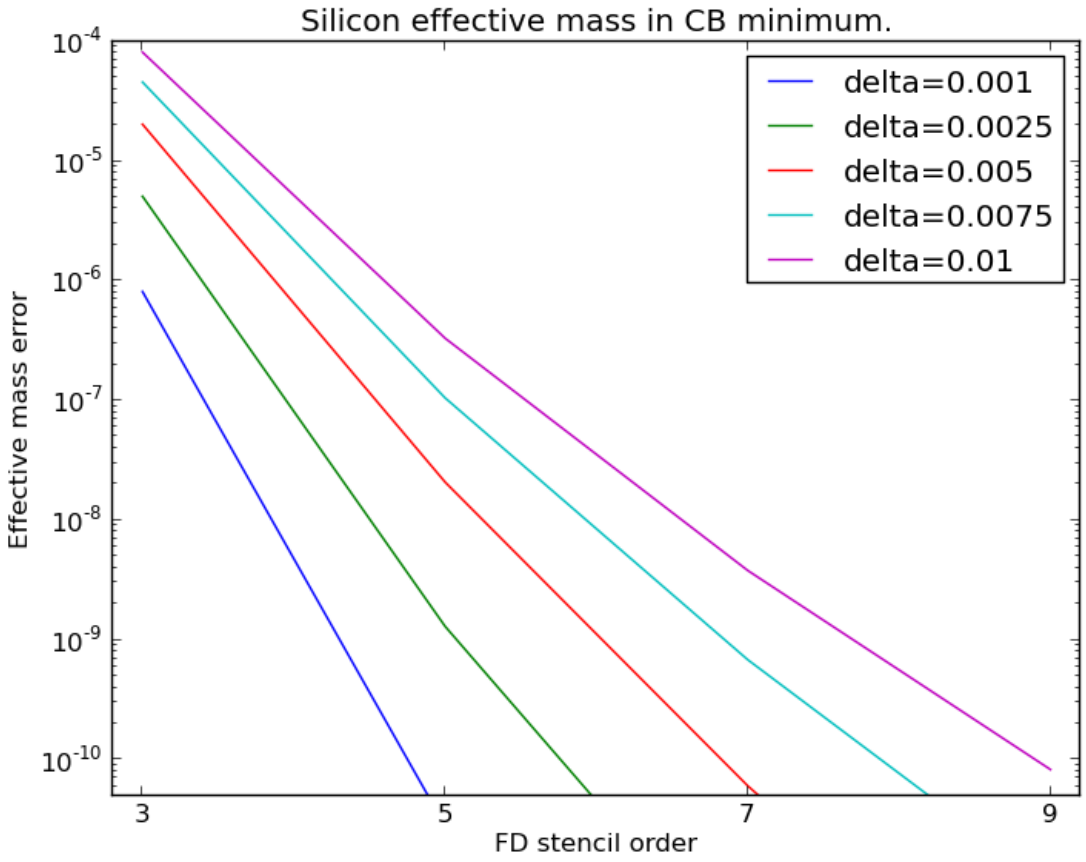


Fig. 4.19: Errors in the silicon effective mass as a function of the number of points in the FD stencil. Plotted for different distances between the points.

(continued from previous page)

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 5),
    density_mesh_cutoff=25.0*Hartree,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------
# Effective mass
# ------------------------------------------------------------
effective_mass = EffectiveMass(
    configuration=bulk_configuration,
    kpoint_fractional=[0.425000, 0.000000, 0.425000],
    bands_below_fermi_level=1,
    bands_above_fermi_level=1,
    stencil_order=5,
    delta=0.001000*Angstrom**-1,
    )


effective_mass.nlprint()
```

si_effective_mass.py

Notes

- Note that the k-points are given in units of the reciprocal vectors $\mathbf{k}_A$, $\mathbf{k}_B$, and $\mathbf{k}_C$. E.g. the symmetry point X has the fractional k-point coordinates $(\frac{1}{2}, 0, \frac{1}{2})$ since it is described by $X = \frac{1}{2}\mathbf{k}_A + \frac{1}{2}\mathbf{k}_C$. The k-point used in the calculations is thus close to the X-point, where the conduction band has its minimum.
- For each band index, three masses are written in the output. The three masses are the eigenvalues of the effective mass tensor given by $[M^{-1}]_{ij} = \hbar^{-2}\frac{\partial^2 E(k)}{\partial k_i \partial k_j}$.
- In the example above, the three masses of the lowest conduction band are $m^* = (0.19, 0.19, 0.9)m_e$, in good agreement with experimental values.
- For `bands_above_fermi_level` set to a valid integer, N_{ba}, the total number of bands above the Fermi level will be N_{ba} for the spin type *unpolarized*, and $2N_{\text{ba}}$ for *polarized* and *noncollinear*. This is because there is only one spin channel in the former case and two spin channels in the latter two cases. *Spin-orbit* calculations are categorized as non-collinear calculations and therefore follow the same band selection as *noncollinear*. The same behavior applies for the relation between `bands_below_fermi_level` and the total number of bands below the Fermi level. The case where `bands_above_fermi_level` is 2 and `bands_below_fermi_level` is 1, are illustrated in Fig. 4.20.

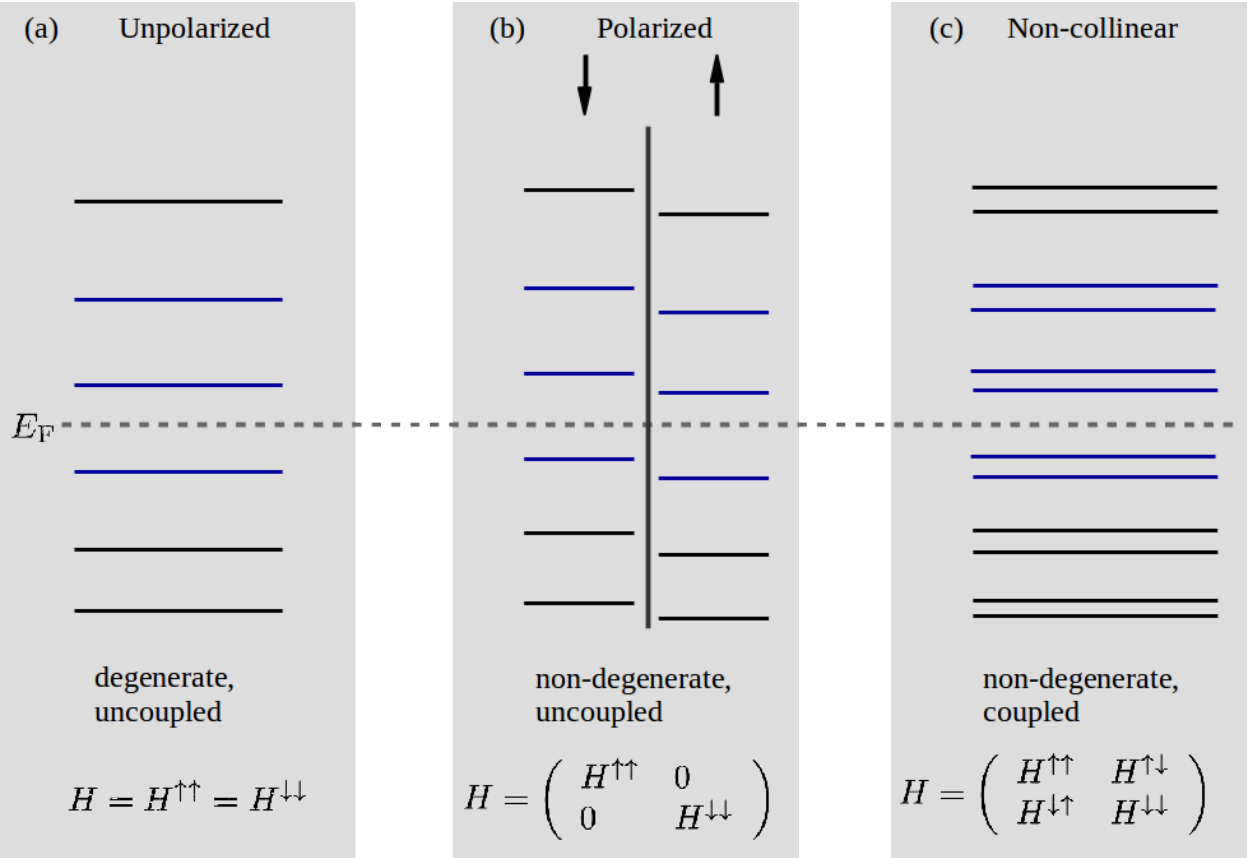


Fig. 4.20: The total number of bands selected (blue lines) by specifying 2 bands above the Fermi level and 1 band below the Fermi level for **(a)** an unpolarized, **(b)** a polarized, and **(c)** a non-collinear calculation. The Fermi level E_F is shown as a horizontal dashed gray line. The vertical separation line in **(b)** signifies that the bands can be categorized according to spin up or down since there is no coupling. The structure of the Hamiltonian for each of the 3 spin types is shown at the bottom.

EffectivePotential

class EffectivePotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the effective potential for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the effective potential should be calculated.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy Œ length^{-1}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`primitiveVectors()`

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

`scale(scale)`

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`shape()`

Returns
The number of grid points in each direction.

Return type
tuple of three int.

`spin()`

Returns
The spin the effective potential is calculated for, always `Spin.All`.

Return type
`Spin.All`

`spinProjection(spin=None)`

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the effective potential and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define the calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)
```

(continues on next page)

(continued from previous page)

```
# Calculate and save the effective potential
effective_potential = EffectivePotential(molecule_configuration)
nlsave('results.nc', effective_potential)
```

nh3_effective_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the effective potential of a DFT calculation. It is the potential that causes the fictitious system of non-interacting electrons to reproduce the electron density of the true system of interacting electrons. See *Effective potential* for more information.

Eigensolutions

class Eigensolutions(*spin_type*, *eigenvalues*, *eigenvectors*, *kpoint*, *fermi_level*)

A container class to store eigensolutions for LCAO and semi-empirical calculations. This class is returned by *calculateEigensolutions()*.

Parameters

•**spin_type** (Unpolarized | Polarized | Noncollinear | SpinOrbit) – The spin polarization type of the configuration for which the eigensolutions were calculated.

•**eigenvalues** (*PhysicalQuantity* of type energy with shape (nspin, nstates) where nspin is the number of spin components and nstates the number of eigensolutions.) – The eigenvalues per spin component.

•**eigenvectors** (*numpy.ndarray with shape (nspin, norbs, nstates) where nspin is the number of spin components, norbs the number of orbitals and nstates the number of eigensolutions.*) – The eigenvectors per spin component.

•**kpoint** (*list(3) of floats*) – The k-point (in fractional coordinates) for which the eigensolutions were calculated.

•**fermi_level** (*PhysicalQuantity* of type energy.) – The fermi level of the configuration for which the eigensolutions were calculated.

eigenvalues()

Returns
the eigenvalues as absolute energy.

Return type
PhysicalQuantity of type energy.

eigenvectors()

Returns
the eigenvectors.

Return type
numpy.ndarray.

fermiLevel()

Returns
the fermi level.

Return type
PhysicalQuantity of type energy.

kpoint()

Returns
the k-point.

Return type
list(3) of floats

spinType()

Parameters
spin_type (Unpolarized | Polarized | Noncollinear | SpinOrbit) – The spin polarization type of the configuration for which the eigensolutions were calculated.

uniqueString()

Return a unique string representing the state of the object.

Eigenstate

class Eigenstate(*configuration=None*, *quantum_number=None*, *projection_list=None*, *spin=None*, *density_mesh_cutoff=None*)

A class for calculating the wave function of a particular orbital in a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the eigenstate should be calculated.
- **quantum_number** (*int*) – The quantum number of the desired eigenstate. **Default:** 0
- **projection_list** (*ProjectionList*) – Specifies the atoms and orbitals within the configuration to include in the calculation. **Default:** ProjectionList(All)
- **spin** (Spin.Up | Spin.Down | Spin.All) – The spin to calculate the eigenstate for. **Default:** Spin.All
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* of type length) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* of type length) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* of type length) – The Cartesian z coordinate.

•**spin** (Spin.Up | Spin.Down | Spin.All) – The spin component to project on. **Default:** The spin the eigenstate was calculated for.

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Up, Spin.Down) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-3/2}$ Œ length^{-1}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

•**downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.

•**downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.

•**downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate the wave function in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* of type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* of type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* of type length) – The Cartesian z coordinate.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component to project on. **Default:** The spin the eigenstate was calculated for.

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Up`, `Spin.Down`) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-3/2}$

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

`quantumNumber()`

Returns
The quantum number of the desired eigenstate.

Return type
int

`scale(scale)`

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`shape()`

Returns
The number of grid points in each direction.

Return type
tuple of three int.

`spin()`

Returns
The spin the eigenstate is calculated for.

Return type
`Spin.Up` | `Spin.Down` | `Spin.All`

`spinProjection(spin=None)`

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z` | `Spin.Up` | `Spin.Down` | `Spin.RealUpDown` | `Spin.ImagUpDown`) – The spin component to project on. **Default:** The spin the object was created with. If the spin was `Spin.All`, `Spin.Sum` will be used for the projection.

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

`toArray()`

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate and save an eigenstate for water molecule:

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,   0.000000000000,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# define a a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# calculate and save the eigenstate with quantum number 0
eigenstate = Eigenstate(molecule_configuration, 0)
nlsave('eigenstate.nc', eigenstate)
```

eigenstate.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

Using a projection list means that only the part of the Hamiltonian involving the atoms in the projection list will be used in the diagonalization to find the eigenstates.

Notice that the *Eigenstate* method neglects interactions across the unit cell boundaries. This makes sense physically for a *MoleculeConfiguration* where the interactions are artifacts of the calculation. However, for a *BulkConfiguration* and a *DeviceConfiguration*, the interactions across cell boundaries represent something real and care should be taken when interpreting the results. For *BulkConfiguration*, see *BlochState* for an alternative analysis object. For *DeviceConfiguration*, see *TransmissionEigenstate* for an alternative analysis object.

Note: For more advanced options and calculating multiple `BlochState` objects `calculateBlochStates()` is advisable.

Eigenvalues

class Eigenvalues(*configuration*, *kpoints=None*, *energy_zero_parameter=None*, *bands_above_fermi_level=None*, *enable_symmetry=None*, *processes_per_kpoint=None*, *method=None*, *interpolation_method=None*, *diagonalization_method=None*)

Class for calculating the eigenvalues for a configuration.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the eigenvalues.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – The k-points for which to perform the calculation. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **energy_zero_parameter** (`FermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `FermiLevel`
- **bands_above_fermi_level** – **Deprecated:** from 2023.09, see `diagonalization_method` and *FullDiagonalizationSolver* instead.
- **enable_symmetry** (*bool*) – Enable or disable the use of symmetry **Default:** `True`
- **processes_per_kpoint** (int | `None`) – The number of processes to use per kpoint. The parameter is only used by the *PlaneWaveCalculator*, or if an iterative diagonalzation solver is used (see parameter `diagonalization_method`). **Default:** For *PlaneWaveCalculator*, the default is the same number of processes per k-point as used in the SCF calculation. Otherwise, the default is `1`.
- **method** – **Deprecated:** from v2023.09, use `interpolation_method` parameter instead.
- **interpolation_method** (`None` | `Full` | *KDotPExpansion1D* | *KDotPExpansion3D*) – The method used for the bandstructure calculation.

  Default is to perform an exact eigenvalue solution for each k-point.

  Alternatively, the k.p expansion method can be use to interpolate the eigenvalues at the requested k-points using the exact eigensolution computed for a small subset of the k-points. There are two modes for the k.p method. 1) Using *KDotPExpansion1D*, for each bandstructure path segment, a few exact eigensolution are calculated and the eigenvalues of the remaining k-points are interpolated using a 1D correction term. 2) Using *KDotPExpansion3D*, the eigenvalues for the requested k-points are interpolated from exact solution on a 3D grid. The bands, grid and wave functions of a previous ground state computation can be used in this approach.

Default: `Full` meaning that all k-points will be calculated exactly.

- **diagonalization_method** – Method used for diagonalizing the hamiltonian.

This parameter allows to choose between a full diagonalization solver and an iterative subspace solver. The full diagonalization solver evaluates all bands from the lowest energy one to a given number of bands above fermi level. The iterative subspace solver allows to evaluate a given number of bands around fermi level, or around an energy of choice.

The full diagonalization solver is more robust, but can be proibitively expensive for very large systems.

The iterative solver can deal with very large systems (tens of thousands atoms and beyond) and greatly outperforms when calculating a small number of eigenvalues, but it is also inherently less robust.

Note: the exact method used when selecting *FullDiagonalizationSolver* is defined by the calculator (see *AlgorithmParameters*). *IterativeDiagonalizationSolver* is not supported for *PlaneWaveCalculator*

Default: `FullDiagonalizationSolver`

Type
FullDiagonalizationSolver | *IterativeDiagonalizationSolver*

densityOfStates()

Returns
A DensityOfStates object constructed from the calculated eigenvalues. This is only possible, if the kpoint used in this Eigenvalues object are of the type *MonkhorstPackGrid* or *RegularKpointGrid*.

Return type
DensityOfStates

eigenvalues(*spin=None*)

Returns the calculated eigenvalues of the Hamiltonian.

Parameters
spin (`Spin.All` | `Spin.Up` | `Spin.Down`) – The spin of the eigenvalues. For Noncollinear calculations, only `Spin.All` is valid. **Default:** `Spin.All`

Returns
The eigenvalues. If spin is `Spin.All` the shape of the array is (n-spins, n-kpoints, n-bands) for Unpolarized and Polarized calculations, and (n-kpoints, n-bands) for Noncollinear calculations. When spin is `Spin.Up` or `Spin.Down` the shape is (n-kpoints, n-bands).

Return type
PhysicalQuantity of type energy.

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy.

evaluate(*spin=None*)

Returns the irreducible kpoints and the corresponding eigenvalues of the Hamiltonian.

Parameters
spin (`Spin.All` | `Spin.Up` | `Spin.Down`) – The spin of the eigenvalues. For Noncollinear caclulations, only `Spin.All` is valid. **Default:** `Spin.All`

Returns
The irriducible kpoints and the corresponding eigenvalues. When spin is `Spin.All` the shape of the eigenvalues array

is (n-spins, n-kpoints, n-bands) for Unpolarized and Polarized calculations, and (n-kpoints, n-bands) for Noncollinear calculations. When spin is `Spin.Up` or `Spin.Down` the shape is (n-kpoints, n-bands).

Return type
numpy.ndarray, *PhysicalQuantity* of type energy.

fermiLevel(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. Must be either `Spin.Up`, `Spin.Down`, or `Spin.All`. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** `Spin.All`

Returns
The Fermi level in absolute energy.

Return type
`PhysicalQuantity` of type energy

interpolationMethod()

Returns
The eigenvalue generation method.

Return type
`None` | `Full` | *KDotPExpansion3D*

irreducibleKpoints()

Returns
The irriducible kpoints for which the eigenvalues are calculated.

Return type
numpy.ndarray

irreducibleWeights()
Get the weights of the irreducible k-points for which the eigenvalues are calculated.

Returns
The weights.

Return type
numpy.ndarray

kpoints()

Returns
The kpoints given as input.

Return type
class:*~.MonkhorstPackGrid* | *RegularKpointGrid* | numpy.ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Deprecated: Deprecated. Use interpolationMethod instead.

Deprecated: Use `interpolationMethod()` instead.

Returns
The method used for the bandstructure calculation.

Return type
None | Full | *KDotPExpansion1D* | *KDotPExpansion3D*

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spins()

Returns
The spins associated to this analysis object.

Return type
list of spin flags.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate and save the eigenvalues for a bulk silicon configuration:

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306 * Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates,
    )

# k-point sampling
```

(continues on next page)

(continued from previous page)

```
k_point_sampling = MonkhorstPackGrid(
    na=3,
    nb=3,
    nc=3,
    )

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

# LCAO calculator
calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Calculate and save the eigenvalues
eigenvalues = Eigenvalues(bulk_configuration)
nlprint(eigenvalues)
nlsave('eigenvalues.hdf5', eigenvalues)
```

eigenvalues.py

By default the eigenvalues will be calculated for the same kpoints as used in the Calculator. It is, however, possible to calculate the eigenvalues at an arbitrary list of kpoints or *MonkhorstPackGrid* or *RegularKpointGrid* as:

```
# Calculate the eigenvalues at two selected k-points.
eigenvalues = Eigenvalues(
        bulk_configuration,
        kpoints=[[0.0, 0.0, 0.0], [0.1, 0.2, 0.3]])

# Calculate the eigenvalues using a MonkhorstPackGrid.
eigenvalues = Eigenvalues(
        bulk_configuration,
        kpoints=MonkhorstPackGrid(5, 5, 5))

# Calculate the eigenvalues using a RegularKpointGrid.
eigenvalues = Eigenvalues(
        bulk_configuration,
        kpoints=RegularKpointGrid(5, 5, 5))
```

If an *Eigenvalues* object has been calculated using either a *MonkhorstPackGrid* or *RegularKpointGrid* it is possible to construct a *DensityOfStates* object using the already calculated eigenvalues as:

```
# Calculate the eigenvalues using a MonkhorstPackGrid.
eigenvalues = Eigenvalues(
        bulk_configuration,
        kpoints=MonkhorstPackGrid(5, 5, 5))

# Calculate a DensityOfStates object.
density_of_states = eigenvalues.densityOfStates()
```

ElasticConstants

class ElasticConstants(*configuration*, *optimizer=None*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *eta_max=None*, *n_eta=None*, *enable_symmetry=None*, *fit_order=None*)

Class to calculate the elastic constants of a bulk material. Uses finite strain and calculates the elastic constants from the linear stress response.

Parameters

- **configuration** (*BulkConfiguration*.) – The *BulkConfiguration* for which the elastic constants should be calculated.
- **optimizer** (`FIRE` | `LBFGS` | `NoOptimization`) – The optimizer method used for optimizing the internal coordinates after each strain. Set this to NoOptimization to disable optimization of the strained cell. **Default:** `LBFGS`
- **max_forces** (*PhysicalQuantity* of type force) – The force threshold for the geometry optimization. **Default:** `0.005 * eV / Ang`
- **max_steps** (*int*) – The maximum number of steps used by the optimizer. **Default:** `200`
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** `0.2 * Ang`
- **eta_max** (*float*) – The maximum strain that can be be applied to the cell. Must be positive. **Default:** `0.005`
- **n_eta** (*int*) – The total number of different strain magnitudes for each strain vector. Must be larger than 1. **Default:** `3`
- **enable_symmetry** (*bool*) – Flag to select if only the independent elastic constants are calculated for each crystal system, or if all 21 constants are calculated. **Default:** `True` if the configuration has standard orientation, `False` otherwise.
- **fit_order** (*int*) – The order of the polynomial fit of the stress-strain dependency. Must be in the interval 1 to (*n_eta* - 1). **Default:** `1`

bulkModulusHill()

Returns
The Hill type bulk modulus.

Return type
PhysicalQuantity of type pressure

bulkModulusReuss()

Returns
The Reuss type bulk modulus.

Return type
PhysicalQuantity of type pressure

bulkModulusVoigt()

Returns
The Voigt type bulk modulus.

Return type
PhysicalQuantity of type pressure

enableSymmetry()

Returns
`True` if only the independent elastic constants are calculated for each crystal system, and `False` if all 21 constants are calculated

Return type
bool

etaMax()

Returns
The maximum strain that can be be applied to the cell.

Return type
float

evaluate()

Returns
The elastic constants as a shape (6, 6) array.

Return type
PhysicalQuantity of type pressure

fitOrder()

Returns
The order of the polynomial fit of the stress-strain dependency.

Return type
int

maxForces()

Returns
The force threshold for the geometry optimization.

Return type
PhysicalQuantity of type force

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nEta()

Returns
The total number of different strain magnitudes for each strain vector.

Return type
int

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

poissonRatio()

Returns
The Poisson ratios in various directions as a shape (3, 3) array.

Return type
numpy.array

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shearModulusHill()

Returns
The Hill type shear modulus.

Return type
PhysicalQuantity of type pressure

shearModulusReuss()

Returns
The Reuss type shear modulus.

Return type
PhysicalQuantity of type pressure

shearModulusVoigt()

Returns
The Voigt type shear modulus.

Return type
PhysicalQuantity of type pressure

uniqueString()

Return a unique string representing the state of the object.

youngsModulus()

Returns
The Youngs moduli.

Return type
PhysicalQuantity of type pressure

Usage Examples

Calculate the elastic constants of bulk silicon using the Stillinger-Weber potential[1], print the results, and save the object to an nc file:

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

```
# Set up bulk configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[ 0.  ,  0.  ,  0.  ],
                            [ 0.25,  0.25,  0.25]])

# Define the calculator
calculator = TremoloXCalculator(parameters=StillingerWeber_Si_1985())
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Calculate the elastic constants
elastic = ElasticConstants(bulk_configuration,
                           optimizer=LBFGS(),
                           max_forces=0.001*eV/Angstrom,
                           eta_max=0.002,
                           n_eta=3,
                           enable_symmetry=True,
                           fit_order=1)
# Print the results
nlprint(elastic)
# Save the elastic constants analysis object
nlsave('elastic_constants.nc', elastic)
```

elastic_constants.py

Notes

- Set `optimizer=None` if the atoms in the strained cell should not be optimized before calculating the *Stress*. If the cell only contians one atom, the optimization is automatically disabled.
- Other properties, such as bulk modulus, shear modulus, Youngs modulus, or Poisson ratios can be calculated from the elastic constants. To view these results use the method *nlprint*.
- An *ElasticConstants* calcualtion can be performed using any calculator that supports the calculation of *Stress*.
- Before calculating the elastic constants of a given *BulkConfiguration*, the cell vectors should be optimized using the *OptimizeGeometry* function.

ElectricFieldConstraint

class ElectricFieldConstraint(*electric_field=None*, *born_effective_charge_parameters=None*, *charge_sum_rule_method=None*, *polarization_parameters=None*, *update_strategy=None*, *optimize_for_field_direction=None*, *enable_stress_correction=None*)

Add a response to an applied electic field to energy, forces and stress.

Parameters

- **electric_field** (*PhysicalQuantity* of type Volt / length) – A vector representing the screened electric field inside the material. **Default:** [0.0, 0.0, 0.0] * Volt / Meter

- **born_effective_charge_parameters** (*BornEffectiveChargeParameters*) – A container with the parameters to be passed to *BornEffectiveCharge* when evaluating the electric field correction. **Default:** `BornEffectiveChargeParameters()`
- **charge_sum_rule_method** (`Disabled` | `DistributeEvenly` | `DistributeRelative`.) – Keyword specifying if a charge sum-rule should be applied to the Born effective charges. Can be `Disabled` for no sum-rule application. Alternatively the charge error is either distributed evenly (`DistributeEvenly`) or relative (`DistributeRelative`) to the Born effective charges on the atoms. **Default:** `Disabled`
- **polarization_parameters** (*PolarizationParameters*) – A container with the parameters to be passed to *Polarization* when evaluating the electric field enthalpy. **Default:** `PolarizationParameters()`
- **update_strategy** (`FixElectricFieldCorrection()` | *UpdateElectricFieldCorrection*) – The strategy used to update Polarization and Born Effective Charge during the optimization or molecular dynamics. `FixElectricFieldCorrection()` will calculate *Polarization* and *BornEffectiveCharge* upfront based on the initial configuration and keep the value fixed. *UpdateElectricFieldCorrection* will calculate *Polarization* and *BornEffectiveCharge* at each step, unless the change in configuration coordinates is below a given threshold. **Default:** `FixElectricFieldCorrection()`
- **optimize_for_field_direction** (*bool*) – For an electric field along a Cartesian direction and orthorhombic cell, it is possible to speed up the calculation by evaluating only *Polarization* components along the direction of the field. If `True`, this optimization is enforced when possible. **Default:** `True`
- **optimize_for_field_direction** – For an electric field along a Cartesian direction and orthorhombic cell, it is possible to speed up the calculation by evaluating only *Polarization* components along the direction of the field. If `True`, this optimization is enforced when possible. **Default:** `True`
- **enable_stress_correction** (*bool*) – Whether also the stress tensor should include the electric field correction. **Default:** `True`

bornEffectiveChargeParameters()

Returns
The container with the *BornEffectiveCharge* parameters.

Return type
BornEffectiveChargeParameters

calculatedBornEffectiveCharge()

Retrieve the BornEffectiveCharge analysis available on this object, if any.

Returns
The born effective charge and the configuration for which it was evaluated. In the case of NEB, a list is returned.

Return type
None | (`AtomicConfiguration`, *BornEffectiveCharge*) | list of (`AtomicConfiguration`, *BornEffectiveCharge*)

calculatedPolarization()

Retrieve the last Polarization calculated to evaluate the correction, if any.

Returns
The polarization and the configuration for which it was evaluated. In the case of NEB, a list is returned.

Return type
None | (`AtomicConfiguration`, *Polarization*) | list of `AtomicConfiguration`, list of *Polarization*

chargeSumRuleMethod()

Returns
Whether a charge sum-rule should be applied to the Born effective charges.

Type
`Disabled | DistributeEvenly | DistributeRelative`.

`electricField()`

Returns
A vector representing the screened electric field inside the material. **Default:** [0.0, 0.0, 0.0] * Volt / Meter

Return type
PhysicalQuantity of type Volt / length

`enableStressCorrection()`

Returns
Whether the stress tensor should include the electric field correction.

Type
bool

`frozenDegreesOfFreedom(local_atoms=None)`

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

`optimizeForFieldDirection()`

Returns
Whether the optimization with respect to *Polarization* components is enabled or not.

Return type
bool

`polarizationParameters()`

Returns
The container with the *Polarization* parameters.

Return type
Polarization

`uniqueString()`
Return a unique string representing the state of the object.

`updateStrategy()`

Returns
The strategy used to update Polarization and Born Effective Charges during the optimization or molecular dynamics.

Return type
`FixPolarizationToInitialValue | UpdatedPolarizationDuringTrajectory`

Usage Examples

Optimize the geometry of a tetragonal $BaTiO_3$ with an applied electric field opposite to the initial polarization:

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog, ELECTRIC_FIELD_CONSTRAINT=True)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleTetragonal(3.97816645571*Angstrom, 4.07906435409*Angstrom)

# Define elements
elements = [Barium, Titanium, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.000000000008 ,  0.000000000035, -0.010189450414],
                          [ 0.500000000154,  0.500000000008,  0.465308176297],
                          [ 0.500000000067,  0.500000000004 ,  1.011196206321],
                          [ 0.500000000058,  0.000000000049,  0.498397341501],
                          [ 0.000000000038,  0.500000000045,  0.498397341501]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
    LDABasis.Oxygen_SingleZetaPolarized,
    LDABasis.Titanium_SingleZetaPolarized,
    LDABasis.Barium_SingleZetaPolarized,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = LDA.PZ

k_point_sampling = MonkhorstPackGrid(
    na=2,
    nb=2,
    nc=2,
    )
```

(continues on next page)

(continued from previous page)

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=80.0*Hartree,
    k_point_sampling=k_point_sampling
    )

iteration_control_parameters = IterationControlParameters(
    tolerance=1e-02,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('bulk_configuration.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------

# Use default parameters for the Born effective charges and
# Polarization.
born_effective_charge_parameters = BornEffectiveChargeParameters()
polarization_parameters = PolarizationParameters()

# Setup an electric field constraint object.
efield = ElectricFieldConstraint(
    [0.0, 0.0, 0.1] * Volt / Ang,
    born_effective_charge_parameters=born_effective_charge_parameters,
    polarization_parameters=polarization_parameters,
    update_strategy=UpdateElectricFieldCorrection(max_displacement=0.*Angstrom))

constraints = [efield]

bulk_configuration = OptimizeGeometry(
    bulk_configuration,
    max_forces=0.05*eV/Ang,
    max_steps=200,
    max_step_length=0.2*Ang,
    constraints=constraints,
    trajectory_filename='optimize_bulk_configuration.hdf5',
    trajectory_interval=1*Second,
    restart_strategy=RestartFromTrajectory(),
    optimize_cell=False,
    enable_optimization_stop_file=True,
    )
nlsave('optimize_bulk_configuration.hdf5', bulk_configuration)
```

(continues on next page)

(continued from previous page)

```
nlprint(bulk_configuration)
```

`efield_optimization.py`

In the example above the parameter `update_strategy=UpdateElectricFieldCorrection(max_displacement=0.*Angstrom)` implies that electric field derived force and stress terms (see *section below*) are recalculated at each optimization step. The polarization and Born effective charges along the optimization trajectory are printed in the output log, as long as the log region `ELECTRIC_FIELD_CONSTRAINT` is active (refer to *setVerbosity* documentation).

Note: The order of the constraint specified in input to *OptimizeGeometry*, *OptimizeNudgedElasticBand* and *MolecularDynamics* matters. Constraint objects modify energies, forces or stress in the order they are specified. Constraints which fix energy, forces or stress to a given value, like *FixAtomConstraints*, should always be specified after *ElectricFieldConstraint*.

ElectricFieldConstraint can be used in the same way in *OptimizeNudgedElasticBand* and *MolecularDynamics*.

Notes

ElectricFieldConstraint is used to model the effect of an external electric field in DFT calculations with periodic boundary conditions.

Within the DFT formalism, the electric enthalpy of an insulator under a uniform electric field is expressed as[1]:

$$E^{\varepsilon}[\mathbf{R}, \varepsilon] = E^{0}[\mathbf{R}] - \mathbf{P}[\mathbf{R}] \cdot \varepsilon$$

where ε represents the electric field, $\mathbf{R}$ the atomic coordinates, E^0 the Kohn-Sham energy functional and $\mathbf{P}$ the macroscopic polarization[2].

Only the dominant first-order response to the field is taken into account, i.e. the field-induced polarization of the electronic wave-function, which introduces an explicit dependency of polarization and Kohn-Sham energy on the electric field, is neglected.

The force on an atom i is:

$$F_i^{\varepsilon}[\mathbf{R}] = F_i^{0} + Z_i^{*}[\mathbf{R}] \cdot \varepsilon$$

where the second term of the equation is a field-induced force dependent on the Born Effective Charge tensor Z_i^*.

Similarly, the field-induced stress is:

$$\sigma_{\alpha\beta}^{\varepsilon} = -\sum_i F_{i,\alpha} R_{i,\beta}$$

The field-induced correction to energy, forces and stress can then be determined by calculating a *Polarization* and *BornEffectiveCharge* analysis at a given atomic positions.

The *ElectricFieldConstraint* evaluates such corrections and can be used in all methods which accept constraint objects, i.e. *OptimizeGeometry*, *OptimizeNudgedElasticBand* and *MolecularDynamics*.

Note: *ElectricFieldConstraint* adds a large overhead to calculation time, mainly due to the evaluation of *BornEffectiveCharge*. The Born effective charges and the polarization must in principle be recalculated for any change in

[1] Huaxiang Fu and LăBellaiche. First-principles determination of electromechanical responses of solids under finite electric fields. *Physical review letters*, 91(5):057601, 2003.

[2] R.ăD. King-Smith and D.ăVanderbilt. Theory of polarization of crystalline solids. *Phys. Rev. B*, 47:1651–1654, Jan 1993. doi:10.1103/PhysRevB.47.1651.

atomic coordinates. However, *ElectricFieldConstraint* allows to limit the computational burden by performing such calcualtions more sparsely via the **update_strategy** parameter. The validity of such approach will depend on the system and the quantities under evaluation, and should be verified case by case. For reasonable static geometries, a single evaluation of the Born effective charges may be sufficient.

ElectricFieldGradients

class ElectricFieldGradients(*configuration*, *atom_indices=None*)

Class for calculating and representing the electric field gradients at the atomic nuclei.

Parameters

- **configuration** (*BulkConfiguration*) – Configuration with a calculator that supports EFG calculations.
- **atom_indices** (All | list of ints.) – The indices of the atoms for which to calculate the electric field gradient. **Default:** All i.e. all atoms in the configuration.

atomIndices()

Returns
The atom indices.

Return type
list of int.

evaluate()

Get the electric field gradient tensors for each atom.

Returns
A list of electric field gradient tensors.

Return type
list

evaluatePrincipalAxis()

Determine the principel axis (eigenvectors and eigenvalues of the EFG tensor) for each atom.

Returns
The principel axis for each atom.

Return type
two ndarray

evaluatePrincipelAxisComponents()

Determine the principel axis components for each atom.

Returns
The principel axis components and asymmetry parameter for each atom.

Return type
tuple of 2 lists

evaluateQuadrupoleCouplingConstants(*quadrupole_moments=None*)

Determine the quadrupole coupling constants for each atom.

Parameters
quadrupole_moments (*list | None | Automatic*) – List of quadrupole moments for each atom in units of Barns. If Automatic, the quadrupole moment is taken from Pekka Pyykkö (2018): Year-2017 nuclear quadrupole moments,

Molecular Physics, DOI: 10.1080/00268976.2018.1426131 select non zero quadrupole moment for isotope of highest abbundance. **Default:** 1 Barn = 1.e-28*m^2

Returns
The quadrupole coupling constants and asymmetry parameter for each atom.

Return type
tuple of 2 lists

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`quadrupoleMoments(atom_index, quadrupole_moments=None)`

Returns the quadrupole moment of the atom index in the configuration.

Parameters

- **atom_index** (*int*) – The atom index in the configuration.
- **quadrupole_moments** (*None | Automatic*) – If Automatic, the quadrupole moment is taken from Pekka Pyykkö (2018): Year-2017 nuclear quadrupole moments, Molecular Physics, DOI: 10.1080/00268976.2018.1426131 select non zero quadrupole moment for isotope of highest abbundance. **Default:** 1 Barn = 1.e-28*m^2.

Returns
The quadrupole moment of the atom index in the configuration.

Return type
PhysicalQuantity

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Calculate the electric field gradient of silicon, save it to a file, and print out the quadrupole coupling constant:

```
# Set up silicon crystal.
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306 * Angstrom),
    elements=[Silicon, Silicon],
    cartesian_coordinates=[[ 0.     , 0.     , 0.],
```

(continues on next page)

(continued from previous page)

```
                            [ 1.35765, 1.35765, 1.35765]] * Angstrom
    )
# Setup calculator.
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(6, 6, 6),
    )
# Setup PAW data set.
basis_set = [
    PAWPBEGPAW.Silicon,
    ]

# Setup plane wave calculator.
calculator = PlaneWaveCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Electric Field Gradients.
efg_analysis = ElectricFieldGradients(
    configuration=bulk_configuration,
    atom_indices=All,
    )

nlsave('si_efg.hdf5', efg_analysis)

# Print out the quadrupole coupling constant.
print('Quadrupole coupling constants')
print(efg_analysis.evaluateQuadrupoleCouplingConstants())
```

si_efg.py

Notes

The quadrupole coupling constant is defined as

$$C_q = \frac{eV_{zz}q}{h},$$

where e is the absolute value of the electron charge, h the plank constant and q is the nuclear quadrupolar momentum. V_{zz} is the largest eigenvalue of the diagonalized traceless electric field gradient tensor at the nucleus:

$$V_{ij}(\mathbf{r}) = \int d\mathbf{r}' \frac{\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|^3}\left[\delta_{ij} - 3\frac{(\mathbf{r}_i-\mathbf{r}'_i)(\mathbf{r}_j-\mathbf{r}'_j)}{|\mathbf{r}-\mathbf{r}'|^2}\right],$$

where $\rho(\mathbf{r})$ is the charge neutral nuclear and all electron density. i, j denote Cartesian coordinates

ElectroOpticalTensor

class ElectroOpticalTensor(*configuration*, *dielectric_tensor*, *raman_spectrum*, *second_harmonics_generation_susceptibility=None*, *phonon_modes=None*, *use_principal_axes=None*, *apply_symmetries=None*)

Constructor for the electro-optical (Pockels) tensor object.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the electro-optical response.
- **dielectric_tensor** (*DielectricTensor*) – The dielectric_tensor object evaluated at zero energy.
- **raman_spectrum** (*RamanSpectrum*) – The raman_spectrum object calculated by the user.
- **second_harmonics_generation_susceptibility** (*SecondHarmonicsGenerationSusceptibility*) – The second_harmonics_generation_susceptibility object evaluated at zero energy. If None only the ionic terms are calculated. **Default:** None
- **phonon_modes** (list with positive ints | `All`) – The phonon modes to include. **Default:** `All` (All phonon modes are included).
- **use_principal_axes** (*bool*) – Keyword flag specifying if the expression for coinciding crystal and principal axes should be used. **Default:** False
- **apply_symmetries** (*bool*) – Keyword flag specifying if the symmetries should be enforced. **Default:** True

applySymmetries()

Returns
True if tensors are symmetrized.

Return type
bool

clampedElectronicElectroOpticalTensor()

Returns
The clamped electronic part of the electro-optical/Pockels tensor. Dimension is (3,3,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedElectronicElectroOpticalVoigtTensor(*contribution='Average'*)

Returns
The clamped ionic part of the electro-optical/Pockels tensor in Voigt notation. Dimension is (6,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedIonicElectroOpticalTensor()

Returns
The clamped ionic part of the electro-optical/Pockels tensor. Dimension is (3,3,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedIonicElectroOpticalTensorModes()

Returns
The clamped ionic part of the electro-optical/Pockels tensor for individual modes. Dimension is (3,3,3,M) where M is the number of phonon modes.

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedIonicElectroOpticalVoigtTensor(*contribution='Average'*)

Returns
The clamped ionic part of the electro-optical/Pockels tensor in Voigt notation. Dimension is (6,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedIonicElectroOpticalVoigtTensorModes(*contribution='Average'*)

Returns
The clamped ionic part of the electro-optical/Pockels tensor in Voigt notation for individual modes. Dimension is (6,3,M) where M is the number of phonon modes.

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedTotalElectroOpticalTensor()

Returns
The sum of clamped electronic and ionic electro-optical/Pockels tensor. Dimension is (3,3,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

clampedTotalElectroOpticalVoigtTensor(*contribution='Average'*)

Returns
The sum of clamped electronic and ionic electro-optical/Pockels tensor in Voigt notation. Dimension is (6,3).

Return type
PhysicalQuantity with the unit `picoMeter / Volt`.

dynamicDielectricTensor()

Returns
The dynamic dielectric constant tensor - i.e. excluding the ionic contribution. Dimension is (3,3).

Return type
Dimensionless ndarray.

imaginaryRefractiveIndex()

Returns
The imaginary part of the refractive index including the ionic contribution. Dimension is (3,3).

Return type
Dimensionless ndarray.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`phononEnergies`()

Returns
The phonon eigenenergies.

Return type
PhysicalQuantity of type energy.

`phononModes`()

Query method for the phonon modes.

Returns
The list of phonon mode indices used in the calculation.

Return type
list of int.

`phononWaveNumbers`()

Returns
The phonon wave numbers calculated from the phonon frequencies.

Return type
PhysicalQuantity of type inverse length.

`realRefractiveIndex`()

Returns
The real part of the refractive index including the ionic contribution. Dimension is (3,3).

Return type
Dimensionless ndarray.

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`staticDielectricTensor`()

Returns
The static dielectric constant tensor - i.e. including the ionic contribution. Dimension is (3,3).

Return type
Dimensionless ndarray.

`uniqueString`()

Return a unique string representing the state of the object.

usePrincipalAxes()

Returns
True if expression for coinciding crystal and principal axes was used.

Return type
bool

Usage Examples

Calculate the static *ElectroOpticalTensor* of a configuration including both ionic and electronic contributions.

```
electro_optical_tensor = ElectroOpticalTensor(
  configuration,
  dielectric_tensor,
  raman_spectrum,
  second_harmonics_generation_susceptibility,
  phonon_modes,
  use_principal_axes,
  apply_symmetries)
```

The clamped electro-optical (Pockels) tensor will be evaluated from the zero energy *DielectricTensor* and the *RamanSpectrum* objects. The *SecondHarmonicsGenerationSusceptibility* is optional as input for addition of the electronic contribution to the tensor.

Notes

The change in the inverse static dielectric tensor, ϵ_r^{-1}, in the presence of an external electric field, E_α, is:

$$\Delta(\epsilon_r^{-1})_{ij} = \sum_{\alpha=1}^{3} r_{ij\alpha} E_\alpha + ...$$

where the rank 3 tensor, r_{ijk}, is the electro-optical (EO) tensor. The clamped EO tensor (neglecting piezoelectric terms) has two contributions[1]: an electronic term related to the modification of the electronic structure for fixed ions positions (displacements $\mathbf{u} = 0$) and a ionic term related to field-induced ion displacements.

The inverse modification in the equation above can be related to the direct variation through the zero-field dielectric tensor as

$$\Delta(\epsilon_r^{-1})_{ij} = -\sum_{m,n=1}^{3} \left((\epsilon_r^{\infty})^{-1}\right)_{im} \Delta\epsilon_{r,mn} \left((\epsilon_r^{\infty})^{-1}\right)_{nj},$$

$$\Delta\epsilon_{r,mn} = \Delta\epsilon_{r,mn}^{el} + \Delta\epsilon_{r,mn}^{ion}$$

The electronic and ionic change in the dielectric tensor refers to the partial derivatives $\frac{d\epsilon_{r,mn}}{dE_\gamma} = \left.\frac{\partial\epsilon_{r,ij}}{\partial E_\gamma}\right|_{u=0} + \left.\frac{\partial\chi^{(1)}_{r,ij}}{\partial E_\gamma}\right|_{E_\gamma=0}$. The optical carrier frequency is assumed high so that the dynamic (high frequency) dielectric tensor

[1] M.ăVeithen, X.ăGonze, and Ph. Ghosez. Nonlinear optical susceptibilities, Raman efficiencies, and electro-optic tensors from first-principles density functional perturbation theory. *Phys. Rev. B*, 71(12):125107, mar 2005. doi:10.1103/PhysRevB.71.125107.

ϵ_r^∞ can be used in the projection[2]. The electronic part is related to the second harmonic generation susceptibility, $\chi^{(2)}$:

$$\left.\frac{\partial \epsilon_{r,ij}}{\partial E_\gamma}\right|_{u=0} = 2\chi^{(2)}_{ij\gamma}$$

The ionic part is obtained from the Raman susceptibility, α^λ_{ij}, and infrared mode polarity, p^λ_α, of phonon λ with angular frequency ω_λ:

$$\left.\frac{\partial \chi^{(1)}_{r,ij}}{\partial E_\gamma}\right|_{E_\gamma=0} = \sum_\lambda \left.\frac{\partial \chi^{(1)}_{r,ij}}{\partial u_\lambda}\right|_{E_\gamma=0} \frac{\partial u_\lambda}{\partial E_\gamma} = \frac{1}{\Omega_0}\sum_\lambda \frac{\alpha^\lambda_{ij} p^\lambda_\gamma}{\omega^2_\lambda}$$

where we defined the Raman tensor $\alpha^\lambda_{ij} = \Omega_0 \left.\partial\chi^{(1)}_{r,ij}/\partial u_\lambda\right|_{E_\gamma=0}$, and used that $\partial u_\lambda/\partial E_\gamma = p^\lambda_\gamma/\omega^2_\lambda$. Ω_0 is the unit cell volume. The infrared mode polarity is given by:

$$p^\lambda_\alpha = \sum_{i,\beta} Z_{\alpha\beta}(i) e^\lambda_\beta/\sqrt{m_i},$$

where $Z_{\alpha\beta}(i)$ is the Born effective charge tensor for atom i with mass m_i and e^λ_β is the eigendisplacement of phonon mode λ with frequency ω_λ.

These factors are obtained from the *RamanSpectrum*, and *DielectricTensor* analysis objects, respectively. From these relations we can evaluate the clamped EO tensor. Inserting the partial derivatives we see that

$$r^{el}_{ij\gamma} = -\sum_{m,n=1}^{3}\left((\epsilon_r^\infty)^{-1}\right)_{im} 2\chi^{(2)}_{mn\gamma}\left((\epsilon_r^\infty)^{-1}\right)_{nj},$$

$$r^{ion}_{ij\gamma} = -\sum_{m,n=1}^{3}\left((\epsilon_r^\infty)^{-1}\right)_{im} \frac{1}{\Omega_0}\sum_\lambda \frac{\alpha^\lambda_{mn} p^\lambda_\gamma}{\omega^2_\lambda}\left((\epsilon_r^\infty)^{-1}\right)_{nj}$$

In Voigt notation we contract the first two indices into six-vector notation, $r_{ij\alpha} \to r_{l\alpha}$, where $l = 1, ..., 6$ for $\{ij\} = xx, yy, zz, yz = zy, zx = xz, xy = yx$.

Usage Example

This example script illustrates the full workflow of the *ElectroOpticalTensor* calculation and display a summary of results:

```
filename = 'ElectroOpticalTensorWorkflow.hdf5'

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleTetragonal(4.070385412109747*Angstrom, 4.115226913895633*Angstrom)

# Define elements
```

(continues on next page)

[2] W.ăD. Johnston. Nonlinear Optical Coefficients and the Raman Scattering Efficiency of LO and TO Phonons in Acentric Insulating Crystals. *Phys. Rev. B*, 1(8):3494–3503, apr 1970. doi:10.1103/PhysRevB.1.3494.

(continued from previous page)

```
elements = [Barium, Titanium, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.028650982641, -0.028665746616,  0.087957144726],
                          [ 0.544026489212,  0.455959167129,  0.605611994272],
                          [ 0.515281929165,  0.484708002041,  1.05622848206 ],
                          [ 0.514348270569, -0.004594374366,  0.570135591311],
                          [ 0.00458017807 ,  0.48564221155 ,  0.570135494531]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(
    na=10,
    nb=10,
    nc=10,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=105.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=FermiDirac(100.0*Kelvin*boltzmann_constant),
    )

iteration_control_parameters = IterationControlParameters(
    tolerance=1e-05,
    max_steps=1000,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave(filename, bulk_configuration)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
constraints = [SpaceGroupConstraint()]

bulk_configuration = OptimizeGeometry(
    bulk_configuration,
```

(continues on next page)

(continued from previous page)

```
    max_forces=0.01*eV/Ang,
    max_stress=0.05*GPa,
    max_steps=2000,
    max_step_length=0.2*Ang,
    constraints=constraints,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
)
nlsave(filename, bulk_configuration)
nlprint(bulk_configuration)

# ---------------------------------------------------------------
# Optical Spectrum
# ---------------------------------------------------------------
kpoint_grid = MonkhorstPackGrid(
    na=25,
    nb=25,
    nc=25,
    )

optical_spectrum = OpticalSpectrum(
    configuration=bulk_configuration,
    kpoints=kpoint_grid,
    energies=numpy.linspace(0, 0, 1)*eV,
    broadening=0.1*eV,
    bands_below_fermi_level=1000,
    bands_above_fermi_level=1000,
    )
nlsave(filename, optical_spectrum)

# ---------------------------------------------------------------
# Born Effective Charge
# ---------------------------------------------------------------
born_effective_charge = BornEffectiveCharge(
    configuration=bulk_configuration,
    kpoints_a=MonkhorstPackGrid(30, 10, 10),
    kpoints_b=MonkhorstPackGrid(10, 30, 10),
    kpoints_c=MonkhorstPackGrid(10, 10, 30),
    atomic_displacement=0.01*Angstrom,
    )
nlsave(filename, born_effective_charge)

# ---------------------------------------------------------------
# Dynamical Matrix
# ---------------------------------------------------------------
polar_phonon_splitting_parameters = PolarPhononSplittingParameters(
    optical_spectrum=optical_spectrum, born_effective_charge=born_effective_charge)
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    filename=filename,
    object_id='dynamical_matrix',
    repetitions=(5, 5, 5),
```

(continues on next page)

(continued from previous page)

```
    atomic_displacement=0.01*Angstrom,
    acoustic_sum_rule=True,
    finite_difference_method=Central,
    force_tolerance=1e-08*Hartree/Bohr**2,
    processes_per_displacement=1,
    log_filename_prefix='forces_displacement_',
    use_wigner_seitz_scheme=False,
    polar_phonon_splitting_parameters=polar_phonon_splitting_parameters,
    )
dynamical_matrix.update()

# ------------------------------------------------------------
# Dielectric Tensor
# ------------------------------------------------------------
dielectric_tensor = DielectricTensor(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    optical_spectrum=optical_spectrum,
    born_effective_charge=born_effective_charge,
)
nlsave(filename, dielectric_tensor)

# ------------------------------------------------------------
# Susceptibility Derivatives
# ------------------------------------------------------------
susceptibility_derivatives = SusceptibilityDerivatives(
    bulk_configuration,
    filename=filename,
    object_id='susceptibility_derivatives',
    atomic_displacement=0.01*Angstrom,
    finite_difference_method=Central,
    broadening=0.1*eV,
    bands_below_fermi_level=1000,
    bands_above_fermi_level=1000,
    processes_per_displacement=1,
    log_filename_prefix='susceptibility_displacement_',
)
susceptibility_derivatives.update()

# ------------------------------------------------------------
# Raman Spectrum
# ------------------------------------------------------------
raman_spectrum = RamanSpectrum(
    bulk_configuration,
    dynamical_matrix,
    susceptibility_derivatives,
    qpoint=[0.0, 0.0, 0.0],
    phonon_modes=None,
    method=PolarizationDependent,
    polarization_in=[1, 0, 0],
    polarization_out=[1, 0, 0],
    broadening=0.25 * meV,
```

(continues on next page)

(continued from previous page)

```
    )
nlsave(filename, raman_spectrum)

# ------------------------------------------------------------
# Electro Optical Tensor
# ------------------------------------------------------------
electro_optical_tensor = ElectroOpticalTensor(
    configuration=bulk_configuration,
    dielectric_tensor=dielectric_tensor,
    raman_spectrum=raman_spectrum,
    second_harmonics_generation_susceptibility=None,
)
nlsave(filename, electro_optical_tensor)
nlprint(electro_optical_tensor)
```

electro_optical_tensor.py

It is important to note that this example is only intended to give an overview of the capabilities of *ElectroOpticalTensor*; A typical output from the `nlprint` command looks like this:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| Dielectric Tensor Analysis                                                    |
|                                                                               |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
|                                                                               |
| Raman Spectrum Analysis                                                       |
|                                                                               |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
|                                                                               |
| Electro-optical Tensor Analysis                                               |
|                                                                               |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Electro-Optical Tensor Report                                                 |
| In units of [pm/V]:                                                           |
+-------------------------------------------------------------------------------+
|                                                                               |
| Total                                                                         |
|              x             y             z                                  |
| xx        0.00000       0.00000       5.43361                               |
| yy        0.00000       0.00000       5.43361                               |
| zz        0.00000       0.00000       8.79489                               |
| yz        0.00000       0.00000       0.00000                               |
| xz        0.00000       0.00000       0.00000                               |
| xy        0.00000       0.00000       0.00000                               |
|                                                                               |
| Electronic                                                                    |
|              x             y             z                                  |
| xx        0.00000       0.00000       0.00000                               |
| yy        0.00000       0.00000       0.00000                               |
```

(continues on next page)

(continued from previous page)

```
| zz        0.00000        0.00000        0.00000                                |
| yz        0.00000        0.00000        0.00000                                |
| xz        0.00000        0.00000        0.00000                                |
| xy        0.00000        0.00000        0.00000                                |
|                                                                                 |
| Ionic                                                                           |
|             x              y              z                                     |
| xx        0.00000        0.00000        5.43361                                |
| yy        0.00000        0.00000        5.43361                                |
| zz        0.00000        0.00000        8.79489                                |
| yz        0.00000        0.00000        0.00000                                |
| xz        0.00000        0.00000        0.00000                                |
| xy        0.00000        0.00000        0.00000                                |
|                                                                                 |
+---------------------------------------------------------------------------------+
| Ionic Electro-optical tensor in units of [pm/V]:                                |
|                                                                                 |
| Ionic part for mode 4, wavenumber: 165.0261 cm^-1                               |
|             x              y              z                                     |
| xx        0.00000        0.00000        1.28989                                |
| yy        0.00000        0.00000        1.28989                                |
| zz        0.00000        0.00000        1.75727                                |
| yz        0.00000        0.00000        0.00000                                |
| xz        0.00000        0.00000        0.00000                                |
| xy        0.00000        0.00000        0.00000                                |
|                                                                                 |
| Ionic part for mode 7, wavenumber: 252.5027 cm^-1                               |
|             x              y              z                                     |
| xx        0.00000        0.00000        3.35205                                |
| yy        0.00000        0.00000        3.35205                                |
| zz        0.00000        0.00000        5.77609                                |
| yz        0.00000        0.00000        0.00000                                |
| xz        0.00000        0.00000        0.00000                                |
| xy        0.00000        0.00000        0.00000                                |
|                                                                                 |
| Ionic part for mode 14, wavenumber: 657.9398 cm^-1                              |
|             x              y              z                                     |
| xx        0.00000        0.00000        0.79166                                |
| yy        0.00000        0.00000        0.79166                                |
| zz        0.00000        0.00000        1.26153                                |
| yz        0.00000        0.00000        0.00000                                |
| xz        0.00000        0.00000        0.00000                                |
| xy        0.00000        0.00000        0.00000                                |
+---------------------------------------------------------------------------------+
```

It gives the total, electronic and ionic electro-optical tensors, and the ionic resolved in specified mode contributions, in Voigt notation. It is important to check the mode contributions as acoustic modes can give spurious effects numerically.

Depending on the system one could have different calculators for the phonon and electronic properties - i.e. DFT+1/2 for a better band gap and optical properties. One should carefully examine the phonon displacements, band gap and Born effective charges during the analysis.

Furthermore it is possible to also include the *SecondHarmonicsGenerationSusceptibility* in the Raman spectrum and EO tensor calculations which could be important in polar materials lacking inversion symmetry.

ElectrodeConstraint

This module is used to constrain variables in a device system to the bulk electrode values, up to a certain distance. Electrode constraints are turned off with the keyword `ElectrodeConstraint.Off`. Alternatively, a constraint on the density matrix can be defined by declaring an instance of *ElectrodeConstraint.DensityMatrix*.

class DensityMatrix(*electrode_constraint_length=None*)

Class for representing a Density Matrix Constraint for Device simulations. See *ElectrodeConstraint* documentation a usage example.

Parameters
electrode_constraint_length (*PhysicalQuantity* of type length) – The part of the density matrix that is constraint by the electrode density matrix **Default:** `0.0 * Angstrom`

electrodeConstraintLength()

Returns
The electrode constraint length.

Return type
PhysicalQuantity of type length

Usage Examples

Setup a calculation with a constrained density matrix.

```
electrode_constraint = ElectrodeConstraint.DensityMatrix(
    electrode_constraint_length=6.*Ang
    )
device_algorithm_parameters = DeviceAlgorithmParameters(
    electrode_constraint=electrode_constraint
    )

calculator = DeviceHuckelCalculator(
    device_algorithm_parameters = device_algorithm_parameters,
    )
```

ElectronDensity

class ElectronDensity(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the electron density for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the density should be calculated.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type length^{-4}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-3}

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electron density is calculated for, always `Spin.All`.

Return type
`Spin.All`

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the electron density for a NH_3 molecule and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.       ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define the calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the electron density together with the configration.
electron_density = ElectronDensity(molecule_configuration)
nlsave('results.hdf5', electron_density)
nlsave('results.hdf5', molecule_configuration)
```

nh3_density.py

Read in the electron density from a file and calculate the integrated density, which should equal the number of valence electrons in the calculation: $\int n(x,y,z)dydzdx = N_{elec}$. In this case this equals $N_{elec} = 3 \cdot n_H + n_N = 3 + 5 = 8$ since there are 5 (1) valence electrons included in the default PseudoDojo pseudo-potential for Nitrogen (Hydrogen).

```
# Import an electron density
density = nlread('results.hdf5', ElectronDensity)[0]

# Get the shape of the data.
```

(continues on next page)

(continued from previous page)

```
shape = density.shape()

# Find the volume elements.
dX, dY, dZ = density.volumeElement()

# Calculate the unit area in the y-z plane.
dAYZ = numpy.cross(dY,dZ)[0]

# calculate density along x integrated over y,z
n_x = [ density[i,:,:].sum() * dAYZ for i in range(shape[0]) ]

# Integrate the density along x to get the number of electrons.
dx = dX.norm()
number_of_electrons = 0

for i in range(shape[0]):
    number_of_electrons += n_x[i]*dx

print('Total number of electrons = {:.5f}'.format(number_of_electrons))

 # Alternative to calculating the integral
dX, dY, dZ = density.volumeElement()

# Volume of a single grid point
dV = numpy.dot(numpy.cross(dX, dY), dZ)

density_unit = density.unit()
density_array = density.toArray() * density_unit
integral = density_array.sum() * dV

print('Alternative method:')
print('Total number of electrons= {:.5f}'.format(integral))
```

nh3_density_integral.py

Read in the electron density from a file and calculate the first moment of both the negatively charged electron density and the positively charged ions. The sum of the two contributions give the dipole moment for NH_3:

```
# Import an electron density and molecule configuration
density = nlread('results.hdf5', ElectronDensity)[0]
molecule_configuration = nlread('results.hdf5', MoleculeConfiguration)[0]

# Volume of a single grid point
dX, dY, dZ = density.volumeElement()
dV = numpy.dot(numpy.cross(dX, dY), dZ)

# Setup array for the first moment of the electron density.
electron_moment = numpy.zeros(3) * Ang * elementary_charge

# Loop over unit cell vectors:
for axis_index, axis in enumerate(['a', 'b', 'c']):
    (axis_values, density_values) = density.axisProjection(
        projection_type='sum',axis=axis, coordinate_type=Cartesian)
```

(continues on next page)

(continued from previous page)

```
    # The minus sign accounts for the negative charge of electrons.
    moment = numpy.sum(axis_values * density_values) * dV * (-1*elementary_charge)
    electron_moment[axis_index] += moment

# Calculate the dipole moment for the positive nuclei
elements = molecule_configuration.elements()
cartesian_coordinates = molecule_configuration.cartesianCoordinates()

# Get the valence charge for each element from the basis set.
basis_set = molecule_configuration.calculator().basisSet()
ionic_charges = {b.element() : b.numberOfValenceElectrons() * elementary_charge for b in␣
↪basis_set}

# Calculate the first moment of the positive ions:
ion_moment = numpy.zeros(3) * Ang * elementary_charge
for element, coordinate in zip(elements, cartesian_coordinates):
    ion_moment += ionic_charges[element] * coordinate

# Calculate the total dipole moment taking both negative electrons and positive ions␣
↪into account.
total_dipole_moment = ion_moment + electron_moment

# Get the norm of the dipole moment in units of Debye (1 D = 0.2081943 * e*Ang)
dipole_moment = numpy.linalg.norm(total_dipole_moment.inUnitsOf(Angstrom*elementary_
↪charge)) / 0.2081943

nlprint('Dipole moment = {:.3f} D'.format(dipole_moment))
```

For more examples on working with 3D grids, see *HartreePotential*.

Notes

- This class inherits from the *GridValues* class.
- Returns the electron density $n(\mathbf{r})$, as defined in *The Hartree potential and the electrostatic potential*.
- When PAW potentials are used, $n(\mathbf{r})$ is the smooth pseudo valence density.

ElectronDifferenceDensity

class ElectronDifferenceDensity(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the electron difference density for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the difference density should be calculated.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type length^{-4}

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- `downsampling_a` (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- `downsampling_b` (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- `downsampling_c` (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- `x` (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- `y` (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- `z` (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- `spin` (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-3}

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- `i` (*int*) – The grid index in the A direction.
- `j` (*int*) – The grid index in the B direction.
- `k` (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electron difference density is calculated for, always `Spin.All`.

Return type
`Spin.All`

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the electron difference density and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.      ,  0.      ,  0.124001],
                           [ 0.      ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define the calculator
iteration_control_parameters = IterationControlParameters()

calculator = HuckelCalculator(
    iteration_control_parameters=iteration_control_parameters,
    )

molecule_configuration.setCalculator(calculator)

# Calculate and save the electron difference density
difference_density = ElectronDifferenceDensity(molecule_configuration)
nlsave('results.nc', difference_density)
```

nh3_difference_density.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the electron difference density $\delta n(\mathbf{r})$, as defined in *The Hartree potential and the electrostatic potential*.

ElectronLocalizationFunction

class ElectronLocalizationFunction(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the electron localization function for a configuration.

Parameters

•**configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the electron localization function should be calculated.

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type=None*, *axis=None*, *spin=None*, *projection_point=None*, *coordinate_type=None*)

Project a 3D Grid onto one of the cartesian axes.

Parameters

•**projection_type** (`'sum'` | `'average'` | `'avg'` | `'line'`) – Which type of projection to perform. **Default:** `'sum'`

•**axis** (`'a'` | `'b'` | `'c'`) – Onto which axis to project the data. **Default:** `'c'`

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.RealUpDown` | `Spin.ImagUpDown`) – Which spin component to project. **Default:** `Spin.Up`

•**projection_point** (*numpy.array*) – Coordinates of the point through which to take a lineout. **Default:** `numpy.array([0.0, 0.0, 0.0])`

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data.

Return type
tuple of numpy arrays

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* of type length.) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* of type length.) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* of type length.) – The Cartesian z coordinate.

•**spin** (`Spin.All` | `Spin.Up` | `Spin.Down` | `Spin.RealUpDown` | `Spin.ImagUpDown` | `Spin.UpUp` | `Spin.DownDown`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Up`, `Spin.Down`, `Spin.RealUpDown`, `Spin.ImagUpDown`) components is returned.

Return type
PhysicalQuantity of type unitless

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- `downsampling_a` (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- `downsampling_b` (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- `downsampling_c` (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- `x` (*PhysicalQuantity* of type length) – The Cartesian x coordinate.
- `y` (*PhysicalQuantity* of type length) – The Cartesian y coordinate.
- `z` (*PhysicalQuantity* of type length) – The Cartesian z coordinate.
- `spin` (`Spin.Up` | `Spin.Down` | `Spin.RealUpDown` | `Spin.All` | `Spin.ImagUpDown` | `Spin.UpUp` | `Spin.DownDown`) – The spin component to project on. **Default:** The spin this object was calculated for.

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Up`, `Spin.Down`, `Spin.RealUpDown`, `Spin.ImagUpDown`) components is returned.

Return type
PhysicalQuantity of type unitless

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- `i` (*int*) – The grid index in the A direction.
- `j` (*int*) – The grid index in the B direction.
- `k` (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electron localization function is calculated for, always Spin.All.

Return type
Spin.All

spinProjection(*spin=None*)

Query method to get a spin component of the ElectronLocalizationFunction object.

Parameters
spin (Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component for which to return the electron localization function object. **Default:** Spin.Up

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the electron localization function and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 6.13508 ,  5.790587,  2.75    ],
                           [ 6.13508 ,  6.73176 ,  2.336663],
                           [ 6.95016 ,  5.32    ,  2.336663],
                           [ 5.32    ,  5.32    ,  2.336663]]*Angstrom
    )


# Define the calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the effective potential
elf = ElectronLocalizationFunction(molecule_configuration)
nlsave('results.nc', molecule_configuration)
nlsave('results.nc', elf)
```

nh3_elf.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

- The electron localization function (ELF) is a 3D function with values between 0 and 1. The positions where the electron is localized has a high ELF value. The function was originally defined by Becke for molecular systems[1], however, the QuantumATK implementation is also valid for bulk system.
- To export the grid data of an electron localization function use the method *nlprint*.
- Calculation of the electron localization function is only supported for the *LCAOCalculator* .

ElectronPhononCoupling

class ElectronPhononCoupling(*configuration*, *hamiltonian_derivatives*, *dynamical_matrix*, *kpoints_fractional=None*, *kpoints_cartesian=None*, *qpoints_fractional=None*, *qpoints_cartesian=None*, *electron_bands=None*, *phonon_modes=None*, *calculate_velocities=None*, *max_interaction_range=None*, *rotate_to_pure_spin_states=None*, *energy_tolerance=None*, *initial_state_energy_range=None*, *store_dense_coupling_matrices=None*, *coupling_matrix_tolerance=None*, *use_kpoint_symmetry=None*, *hamiltonian_derivatives_constraints=None*)

Class for calculating the electron-phonon coupling matrix for a BulkConfiguration.

Parameters

- **configuration** (*BulkConfiguration*) – The BulkConfiguration for which to calculate the electron-phonon coupling.
- **hamiltonian_derivatives** (*HamiltonianDerivatives*) – The HamiltonianDerivatives for the configuration.
- **dynamical_matrix** (*DynamicalMatrix*) – The DynamicalMatrix for the configuration.
- **kpoints_fractional** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – List of k-points. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. This option is mutually exclusive to `kpoints_cartesian`. **Default:** `[[0.0, 0.0, 0.0]]`
- **kpoints_cartesian** (sequence of sequence (size 3) of *PhysicalQuantity* of type inverse length) – List of Cartesian k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], ...] * Angstrom**-1`. This option is mutually exclusive to `kpoints_fractional`. **Default:** `kpoints_fractional` is used.
- **qpoints_fractional** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – List of q-points. This can either be given as a q-point grid or a list of fractional q-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. This option is mutually exclusive to `qpoints_cartesian`. **Default:** `[[0.0, 0.0, 0.0]]`
- **qpoints_cartesian** (sequence of sequence (size 3) of *PhysicalQuantity* of type inverse length) – List of Cartesian q-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], ...] * Angstrom**-1`. This option is mutually exclusive to `qpoints_fractional`. **Default:** `qpoints_fractional` is used.
- **electron_bands** (list of int | `All` | `Automatic`) – The band indices of the Bloch states to include. If `Automatic` the states within the initial_state_energy_range are selected. **Default:** `All` if initial_state_energy_range is None. Otherwise `Automatic`.
- **phonon_modes** (list of int | `All`) – Phonon modes to include. **Default:** `All`
- **calculate_velocities** (*bool*) – Boolean to control if the velocities should be calculated or not. **Default:** `True`

[1] A.ăD. Becke and K.ăE. Edgecombe. A simple measure of electron localization in atomic and molecular systems. *J. Chem. Phys.*, 1990.

•**maximum_interaction_range** (*PhysicalQuantity* of type length) – Set the maximum range of the interactions. **Default:** No maximum interaction range.

•**rotate_to_pure_spin_states** (*bool*) – Boolean to control if the eigenstates should be rotated to pure spin states or not. This is only relevant for non-collinear or spin-orbit calculations. As default for non-collinear calculations the eigenstates are rotated to pure spin states (`True`) and for spin-orbit no eigenstates are rotated (`False`).

•**energy_tolerance** (*PhysicalQuantity* of type energy) – Energy tolerance for calculating matrix elements. The matrix elements are only calculated if $(E_{final} - E_{initial} \pm E_{phonon})$ is smaller in magnitude than the `energy_tolerance`, where $E_{initial}$ is the initial state energy with wavevector k, E_{final} is the final state energy with wavevector k $\pm$ q, and E_{phonon} is the phonon energy at phonon wavevector q. **Default:** `0.1 * eV`

•**initial_state_energy_range** (*PhysicalQuantity* of type energy) – Energy range for the initial state. Electron-phonon coupling elements are only calculated if the energy of the initial state is within the initial state energy range. **Default:** `[-100, 100] * eV` (all matrix elements are calculated included).

•**store_dense_coupling_matrices** (*bool*) – Boolean to control if the electron-phonon coupling elements should be stored as dense matrices. This should only be set to True if a large energy_tolerance (> 0.1*eV) is used. **Default:** `False`

•**coupling_matrix_tolerance** (*PhysicalQuantity* of type energy) – Tolerance for storing the coupling matrix elements. The matrix element is only stored, if it is larger in magnitude than the `coupling_matrix_tolerance`. This parameter is only used, if the parameter `store_dense_coupling_matrices` is `False`. **Default:** `1e-9 * eV`

•**use_kpoint_symmetry** (*bool.*) – Boolean to determine if symmetries are used to reduce k-point selection. **Default:** `True` for kpoints of type MonkhorstPackGrid or RegularKpointGrid. *False* ` for k-point list, which is a requirement in general.

•**hamiltonian_derivatives_constraints** – List of atomic indices, where the hamiltonian derivatives will be constrained, e.g. `[0, 2, 10]`. **Default:** Empty list `[]`

couplingMatrixTolerance()

Query method for the coupling_matrix_tolerance.

Returns
The coupling_matrix_tolerance used in the calculation.

Return type
PhysicalQuantity of type energy.

couplingMatrixVsQ(*spin_index=None*, *phonon_mode=None*, *k_index=None*, *initial_band=None*, *final_band=None*)

Query method for the (scaled) coupling matrix vs. q at specified spin, phonon mode, k-index, initial band and final band.

The scaled coupling matrix element is given by $\mathbf{g}_{kq}^{\lambda nn'} = \langle n'k+q|\Delta H_{q\lambda}|nk\rangle$, where $\Delta H_{q\lambda}$ is the perturbation of the Hamiltonian due to a phonon with momentum q in phonon mode λ, and $|nk\rangle$ is a Bloch state with momentum k and band index n.

Parameters

•**spin_index** (*int*) – The spin index.

•**phonon_mode** (*int*) – The phonon mode.

•**k_index** (*int*) – The k-point index.

•**initial_band** (*int*) – The initial band.

•**final_band** (*int*) – The final band.

Returns
The electron-phonon coupling matrix vs. q. The shape of the array is (number_of_qpoints).

Return type
PhysicalQuantity of type energy.

`eigenvaluesK()`

Query method for the electron Bloch state energies as a function of k.

Returns
The electron Bloch state energies as a function of k. The shape of the array is (number_of_spins, number_of_kpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy.

`eigenvaluesKMinusQ()`

Query method for the electron Bloch state energies as a function of k-q.

Returns
The electron Bloch state energies as a function of k-q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy.

`eigenvaluesKMinusQVsQ(spin_index=None, k_index=None, band=None)`

Query method for the eigenvalues_k_minus_q vs. q at specified spin, k-index, band.

Parameters

- **spin_index** (*int*) – The spin index.
- **k_index** (*int*) – The k-point index.
- **band** (*int*) – The electron band.

Returns
The electron Bloch state energies at k-q as a function of q. The shape for the returned array is (*N*,), where *N* is the number of q-points.

Return type
PhysicalQuantity of type energy.

`eigenvaluesKPlusQ()`

Query method for the electron Bloch state energies as a function of k+q.

Returns
The electron Bloch state energies as a function of k+q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands).

Return type
PhysicalQuantity of type energy.

`eigenvaluesKPlusQVsQ(spin_index=None, k_index=None, band=None)`

Query method for the eigenvalues_k_plus_q vs. q at specified spin, k-index, band.

Parameters

- **spin_index** (*int*) – The spin index.
- **k_index** (*int*) – The k-point index.
- **band** (*int*) – The electron band.

Returns
The electron Bloch state energies at k+q as a function of q. The shape for the returned array is (*N*,), where *N* is the number of q-points.

Return type
PhysicalQuantity of type energy.

`electronBands()`

Query method for the electron bands.

Returns
The list of electronic band indices used in the calculation.

Return type
list of int.

`energyDependentInverseLifeTime(`*energies*, *phonon_modes=None*, *electron_bands=None*, *temperature=None*, *fermi_shift=None*, *integration_method=None*, *energy_broadening=None*`)`

Generator method for the inverse life time as a function of energy.

Parameters

- **energies** (`PhysicalQuantity` of type energy) – The energies to calculate the scattering rate for, given as an energy list.
- **phonon_modes** (*List of indices or NLFlag All.*) – Phonon modes to include. **Default:** All.
- **electron_bands** (*List of indices or NLFlag All.*) – Electronic bands to include. **Default:** All (Include all the calculated electron bands).
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`
- **fermi_shift** (*PhysicalQuantity* of energy unit.) – Shift of the Fermi level. **Default:** `0.0 * eV`
- **integration_method** (`GaussianBroadening | TetrahedronMethod`) – The method to use for calculating the q-integrals. **Default:** `TetrahedronMethod` if a `MonkhorstPackGrid` or `RegularKpointGrid` was used for the q-point sampling. Otherwise `GaussianBroadening(0.1*eV)`
- **energy_broadening** (*PhysicalQuantity* of type energy) – The energy broadening of delta-functions. **Default:** `3.0e-2*eV`

Returns
The inverse relaxation time. The length is len(`energies`).

Return type
PhysicalQuantity of type inverse length.

`energyDependentInverseMeanFreePath(`*energies*, *phonon_modes=None*, *electron_bands=None*, *temperature=None*, *fermi_shift=None*, *integration_method=None*, *energy_broadening=None*`)`

Generator method for the inverse mean free time as a function of energy.

param energies
The energies to calculate the scattering rate for, given as an energy list.

type energies
`PhysicalQuantity` of type energy

param phonon_modes
Phonon modes to include. **Default:** All.

type phonon_modes
List of indices or NLFlag All.

param electron_bands
Electronic bands to include. **Default:** All (Include all the calculated electron bands).

type electron_bands
List of indices or NLFlag All.

param temperature
The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`

type temperature
PhysicalQuantity of type temperature

param fermi_shift
Shift of the Fermi level. **Default:** `0.0 * eV`

type fermi_shift
PhysicalQuantity of energy unit.

param integration_method
The method to use for calculating the q-integrals. **Default:** `TetrahedronMethod` if a `MonkhorstPackGrid` or `RegularKpointGrid` was used for the q-point sampling. Otherwise `GaussianBroadening(0.1*eV)`

type integration_method
`GaussianBroadening | TetrahedronMethod`

param energy_broadening
The energy broadening of delta-functions. **Default:** `3.0e-2*eV`

type energy_broadening
`PhysicalQuantity` of type energy

Returns
The energy dependent inverse mean free path in total and mode resolved, and the raw data. The shapes are (n_energies), (n_modes, n_energie), and (n_spins, n_modes, n_kpoints, n_bloch_states) where n_energies is the number of energy points and n_modes is the number of phonon modes. The shape of the last array is (n_spins, n_modes, n_kpoints, n_bands)

Return type
PhysicalQuantity of type inverse length, *PhysicalQuantity* of type inverse length, *PhysicalQuantity* of type inverse length.

`energyTolerance()`

Query method for the energy_tolerance.

Returns
The energy_tolerance used in the calculation.

Return type
PhysicalQuantity of type energy.

`evaluate(`*spin_index=None*, *phonon_mode=None*, *k_index=None*, *initial_band=None*, *final_band=None*`)`

Return the electron-phonon coupling matrix vs. q at specified spin, phonon mode, k-index, initial band and final band.

The scaled coupling matrix element is given by $\mathbf{g}_{kq}^{\lambda nn'} = \langle n'k+q|\Delta H_{q\lambda}|nk\rangle$, where $\Delta H_{q\lambda}$ is the perturbation of the Hamiltonian due to a phonon with momentum q in phonon mode λ, and $|nk\rangle$ is a Bloch state with momentum k and band index n.

Parameters

- **spin_index** (*int*) – The spin index.
- **phonon_mode** (*int*) – The phonon mode.
- **k_index** (*int*) – The k-point index.
- **initial_band** (*int*) – The initial band.
- **final_band** (*int*) – The final band.

Returns
The electron-phonon coupling matrix vs. q. The shape of the array is (number_of_qpoints).

Return type
PhysicalQuantity of type energy.

hamiltonianDerivativesConstraints()

Returns
List of atomic indicies, where the hamiltonian derivative are constrained.

Return type
list

initialStateEnergyRange()

Query method for the initial state energy range.

Returns
The initial_state_energy_range used in the calculation. This is a list of two energies [Emin, Emax] defining the allowed energy range for the initial states.

Return type
PhysicalQuantity of type energy.

inverseCharacteristicLength()

Query method for inverse characteristic length.

Returns
The inverse characteristic length. The shape of the array is (number_of_degrees_of_freedom, number_of_qpoints)

Return type
PhysicalQuantity of type inverse length.

inverseLifeTime(*phonon_modes=None*, *electron_bands=None*, *temperature=None*, *fermi_shift=None*, *integration_method=None*, *refinement=None*)

Calculates the inverse life time due to electron-phonon coupling.

Parameters

- **phonon_modes** (*List of indices or NLFlag All.*) – Phonon modes to include. **Default:** All.
- **electron_bands** (*List of indices or NLFlag All.*) – Electronic bands to include. **Default:** All (Include all the calculated electron bands).
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`
- **fermi_shift** (*PhysicalQuantity* of energy unit.) – Shift of the Fermi level. **Default:** `0.0 * eV`
- **integration_method** (`GaussianBroadening | TetrahedronMethod`) – The method to use for calculating the q-integrals. **Default:** `TetrahedronMethod` if a `MonkhorstPackGrid` or `RegularKpointGrid` was used for the q-point sampling. Otherwise `GaussianBroadening(0.1*eV)`

•**refinement** (*Positive int*) – The number of times the q-grid is refined in each direction. **Default:** 1 (no interpolation)

Returns
A phonon mode, k-point, and electron band resolved life time.

Return type
PhysicalQuantity of type inverse time of shape (phonon modes, k-points, electron bands).

inverseMeanFreePath(*phonon_modes=None*, *electron_bands=None*, *temperature=None*, *fermi_shift=None*, *integration_method=None*, *refinement=None*)

Calculates the inverse mean free path due to electron-phonon coupling.

Parameters

•**phonon_modes** (*List of indices or NLFlag All.*) – Phonon modes to include. **Default:** All.

•**electron_bands** (*List of indices or NLFlag All.*) – Electronic bands to include. **Default:** All (Include all the calculated electron bands).

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`

•**fermi_shift** (*PhysicalQuantity* of energy unit.) – Shift of the Fermi level. **Default:** `0.0 * eV`

•**integration_method** (`GaussianBroadening | TetrahedronMethod`) – The method to use for calculating the q-integrals. **Default:** `TetrahedronMethod` if a `MonkhorstPackGrid` or `RegularKpointGrid` was used for the q-point sampling. Otherwise `GaussianBroadening(0.1*eV)`

Returns
A phonon mode, k-point, and electron band resolved inverse mean free path.

Return type
PhysicalQuantity of type inverse time of shape (phonon modes, k-points, electron bands).

inverseSpinLifeTime(*phonon_modes=None*, *electron_bands=None*, *temperature=None*, *fermi_shift=None*, *integration_method=None*, *refinement=None*)

Calculates the inverse spin life time due to electron-phonon coupling. This can only be done for spin-orbit calculations.

Parameters

•**phonon_modes** (*List of indices or NLFlag All.*) – Phonon modes to include. **Default:** All.

•**electron_bands** (*List of indices or NLFlag All.*) – Electronic bands to include. **Default:** All (Include all the calculated electron bands).

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`

•**fermi_shift** (*PhysicalQuantity* of energy unit.) – Shift of the Fermi level. **Default:** `0.0 * eV`

•**integration_method** (`GaussianBroadening | TetrahedronMethod`) – The method to use for calculating the q-integral. **Default:** `TetrahedronMethod`

•**refinement** (*Positive int*) – The number of times the q-grid is refined in each direction. **Default:** 1 (no interpolation)

Returns
An averaged inverse spin life time, and phonon mode, k-point, and electron band resolved spin life time.

Return type
Two *PhysicalQuantity* of type inverse time of shape (1), and (phonon modes, k-points, electron bands).

kpoints()

Query method for the k-points in fractional coordinates.

Returns
The list of fractional k-points used in the calculation.

Return type
list

kpointsCartesian()

Query method for the k-points in Cartesian coordinates.

Returns
The list of Cartesian k-points used in the calculation.

Return type
PhysicalQuantity of type inverse length.

kpointsWeights()

Returns
The unreduced fractional k-points. The shape is (k-points).

Return type
list of lists of floats

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

phononEnergies()

Query method for the phonon energies.

Returns
The phonon energies as a function of q. The shape of the array is (number_of_degrees_of_freedom, number_of_qpoints), where number_of_degrees_of_freedom = 3*number_of_atoms.

Return type
PhysicalQuantity of type energy.

phononModes()

Query method for the phonon modes.

Returns
The list of phonon mode indices used in the calculation.

Return type
list of int.

qpoints()

Query method for the q-points in fractional coordinates.

Returns
The list of fractional q-points used in the calculation.

Return type
list

qpointsCartesian()

Query method for the q-points in Cartesian coordinates.

Returns
The list of Cartesian q-points used in the calculation.

Return type
PhysicalQuantity of type inverse length.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

storeDenseCouplingMatrices()

Query method for the store_dense_coupling_matrices Boolean.

Returns
The Boolean determining if the electron-phonon coupling matrix is stored in a dense format (True) or in a sparse format (False).

Return type
bool.

uniqueString()

Return a unique string representing the state of the object.

unscaledCouplingMatrixVsQ(*spin_index=None*, *phonon_mode=None*, *k_index=None*, *initial_band=None*, *final_band=None*)

Query method for the unscaled coupling matrix vs. q at specified spin, phonon mode, k-index, initial band and final band. The unscaled coupling matrix is defined as $\mathbf{M}_{kq}^{\lambda nn'} = \mathbf{g}_{kq}^{\lambda nn'}/l_\lambda$, where $\mathbf{g}_{kq}^{\lambda nn'} = \langle n'k+q|\Delta H_{q\lambda}|nk\rangle$ is the (scaled) coupling matrix. Here $\Delta H_{q\lambda}$ is the perturbation of the Hamiltonian due to a phonon with momentum q in phonon mode λ, and $|nk\rangle$ is a Bloch state with momentum k and band index n. The characteristic length of phonon mode λ is given by $l_\lambda = \sqrt{\hbar/(2\omega_\lambda \mathbf{e}_\lambda \cdot \mathbf{m} \cdot \mathbf{e}_\lambda)}$, where $\mathbf{e}_\lambda$ is the phonon eigenvector of mode λ with corresponding eigenfrequency ω_λ and $\mathbf{m}$ is a diagonal matrix with the atomic masses.

Parameters

- **spin_index** (*int*) – The spin index.
- **phonon_mode** (*int*) – The phonon mode.
- **k_index** (*int*) – The k-point index.
- **initial_band** (*int*) – The initial band.
- **final_band** (*int*) – The final band.

Returns
The unscaled electron-phonon coupling matrix vs. q. The shape of the array is (number_of_qpoints).

Return type
PhysicalQuantity of type energy / length.

useKpointSymmetry()

Returns
Whether the used k-points should be reduced by symmetries.

Return type
bool

velocitiesK()

Query method for the velocities as function of k.

Returns
The electron velocities as a function of k. The shape of the array is (number_of_spins, number_of_kpoints, number_of_electron_bands, 3).

Return type
PhysicalQuantity of type Meter/Second.

velocitiesKMinusQ()

Query method for the velocities as function of k-q.

Returns
The electron velocities as a function of k-q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands, 3).

Return type
PhysicalQuantity of type Meter/Second

velocitiesKMinusQVsQ(*spin_index=None*, *k_index=None*, *band=None*)

Query method for the velocities_k_minus_q vs. q at specified spin, k-index, band.

Parameters

- **spin_index** (*int*) – The spin index.
- **k_index** (*int*) – The k-point index.
- **band** (*int*) – The electron band.

Returns
The electron Bloch state velocities at k-q as a function of q. The shape for the returned array is (*N*, 3), where *N* is the number of q-points.

Return type
PhysicalQuantity of type velocity.

velocitiesKPlusQ()

Query method for the velocities as function of k+q.

Returns
The electron velocities as a function of k+q. The shape of the array is (number_of_spins, number_of_kpoints, number_of_qpoints, number_of_electron_bands, 3).

Return type
PhysicalQuantity of type Meter/Second

velocitiesKPlusQVsQ(*spin_index=None*, *k_index=None*, *band=None*)

Query method for the velocities_k_plus_q vs. q at specified spin, k-index, band.

Parameters

- **spin_index** (*int*) – The spin index.
- **k_index** (*int*) – The k-point index.

•**band** (*int*) – The electron band.

Returns
The electron Bloch state velocities at k+q as a function of q. The shape for the returned array is $(N, 3)$, where N is the number of q-points.

Return type
PhysicalQuantity of type velocity.

Usage Examples

Calculate the electron-phonon coupling in graphene for k-point around the K-point, which is $(1/3, 1/3, 0)$ in fractional coordinates, and for phonon q-point around the Γ point.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)

elements = [Carbon, Carbon]

fractional_coordinates = [[0.0           , 0.0           , 0.5],
                          [0.333333333333, 0.666666666667, 0.5]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
# Use only Gamma-point for HamiltonianDerivatives and DynamicalMatrix
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(1, 1, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# Hamiltonian derivatives
# ------------------------------------------------------------
hamiltonian_derivatives = HamiltonianDerivatives(
    bulk_configuration,
    repetitions=(5, 5, 1),
    )
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------------
# Dynamical matrix
# -----------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    repetitions=(5, 5, 1),
    max_interaction_range=10*Angstrom,
    )

# -----------------------------------------------------------------
# Calculator
# -----------------------------------------------------------------
# Use k-point sampling for the ElectronPhononCoupling object.
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# -----------------------------------------------------------------
# Electron Phonon Coupling
# -----------------------------------------------------------------
k_a = numpy.linspace(0.32333, 0.34333, 20)
k_b = numpy.linspace(0.32333, 0.34333, 20)
k_c = numpy.linspace(0, 0, 1)
kpoints = [[a, b, c] for a in k_a for b in k_b for c in k_c]

q_a = numpy.linspace(-0.02, 0.02, 20)
q_b = numpy.linspace(-0.02, 0.02, 20)
q_c = numpy.linspace(0, 0, 1)
qpoints = [[a, b, c] for a in q_a for b in q_b for c in q_c]

electron_phonon_coupling = ElectronPhononCoupling(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    kpoints_fractional=kpoints,
    qpoints_fractional=qpoints,
    electron_bands=All,
    phonon_modes=All,
    energy_tolerance=0.01*eV,
    initial_state_energy_range=[-100,100]*eV,
    store_dense_coupling_matrices=False,
    )

nlsave('graphene.nc', electron_phonon_coupling)
```

(continues on next page)

(continued from previous page)

By specifying the input parameter energy_tolerance=0.01*eV as above, you will only be calculating a matrix element, if the initial- and final energies are separated by the phonon energy, within the specified energy tolerance. When using a small energy_tolerance, it will also be advantagerous to use a sparse data structure. This is achieved by setting store_dense_coupling_matrices=False, as above. These are the default options. If you will only use the ElectronPhononCoupling object as input to a *Mobility* calculation, it is advantageous and faster to only calculate the matric elements for those (k,q)-pairs which are relevant for the mobility calculation. In the equations used for the mobility, the electron-phonon coupling matrix element always occurs together with delta-functions, like $\delta(E_i(k) - E_f(k+q) + \hbar\omega_\lambda(q))$ or $\delta(E_i(k) - E_f(k+q) - \hbar\omega_\lambda(q))$, where $E_i(k)$ and $E_f(k+q)$ are the energies of the initial- and final Bloch state, respectively, and $\hbar\omega_\lambda(q)$ is the phonon energy.

If you which to visually ananlyze the coupling matrix elements with the ElectronPhononCoupling analyzer, it is advantagerous to calculate the matrix elements for all (k,q)-points. This is achieved by specifying a large energy tolerance, energy_tolerance=100*eV, like in the script below. In that case it will also be advantagerous to use a dense matrix representation. This can be specified by setting store_dense_coupling_matrices=True.

```
electron_phonon_coupling = ElectronPhononCoupling(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    kpoints_fractional=kpoints,
    qpoints_fractional=qpoints,
    electron_bands=All,
    phonon_modes=All,
    energy_tolerance=100*eV,
    initial_state_energy_range=[-100,100]*eV,
    store_dense_coupling_matrices=True,
    )

nlsave('graphene.nc', electron_phonon_coupling)
```

Often it will only be relevant to consider initial states close to the Fermi energy. For materials with complicated Fermi surfaces, it is difficult to provide a list of k-point spanning only the Fermi surface. It is therefor possible to specify initial_state_energy_range=[-0.1,0.1]*eV, relative to the Fermi energy (see below). Only matrix elements, with inital states with energies $E_i(k)$ in the specified range will be calculated.

```
electron_phonon_coupling = ElectronPhononCoupling(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    kpoints_fractional=kpoints,
    qpoints_fractional=qpoints,
    electron_bands=All,
    phonon_modes=All,
    energy_tolerance=0.01*eV,
    initial_state_energy_range=[-0.1,0.1]*eV,
    store_dense_coupling_matrices=False,
    )
```

(continues on next page)

(continued from previous page)

```
nlsave('graphene.nc', electron_phonon_coupling)
```

Theoretical notes

The theoretical details of the electron-phonon coupling in QuantumATK is described in[1].

In order to calculate the electron-phonon coupling three main ingredients are needed:

- An updated configuration in order to get the Hamiltonian of the system and the electronic Bloch states.
- A *DynamicalMatrix* object for calculating the phonon modes.
- A *HamiltonianDerivatives* object for calculating the electron-phonon coupling matrix.

The configuration used as input to the *ElectronPhononCoupling* constructor must be exactly the same as used for the *HamiltonianDerivatives* and *DynamicalMatrix* objects.

From the dynamical matrix D we can obtain the phonon frequencies $\omega_{\lambda,\mathbf{q}}$ and eigenvectors $\mathbf{u}_{\lambda,\mathbf{q}}$ for a particular phonon band index λ and phonon wave number $\mathbf{q}$:

$$D(\mathbf{q})\mathbf{u}_{\lambda,\mathbf{q}} = \omega^2_{\lambda,\mathbf{q}}\mathbf{u}_{\lambda,\mathbf{q}},$$

where the phonon mode vectors are normalized, i.e. $\mathbf{u}^T \cdot \mathbf{u} = 1$. We further define mass-scaled phonon mode vectors according to

$$\mathbf{v}_{\lambda,\mathbf{q}}(\alpha) = \mathbf{u}_{\lambda,\mathbf{q}}(\alpha)\sqrt{\frac{\hbar}{2M_\alpha\omega_{\lambda,\mathbf{q}}}},$$

where M_α is the mass of the atom with index α. Let now $j = (\alpha, \mu)$ denote a combined index for atomic site (α) and orbital index (μ), and let an electronic Bloch state be expanded in the LCAO basis as:

$$|\mathbf{k}\rangle = \sum_j c_j^{\mathbf{k}}|j\mathbf{k}\rangle,$$

where

$$|j\mathbf{k}\rangle = \frac{1}{N}\sum_n e^{2\pi i\mathbf{k}\cdot\mathbf{R}_n}|j\mathbf{R}_n\rangle,$$

and $|j\mathbf{R}_n\rangle$ is an LCAO basis orbital in the nth cell displaced by lattice vector $\mathbf{R}_n$ from the central cell with index 0. The electron-phonon coupling matrix for a given phonon mode, $\{\mathbf{q}, \lambda\}$ is then obtained as[2, Page 2826, 1]:

$$g^\lambda_{ij}(\mathbf{k},\mathbf{q}) = \sum_{ij}(c_i^{\mathbf{k}+\mathbf{q}})^* c_j^{\mathbf{k}}\sum_{mn} e^{2\pi(i\mathbf{k}\cdot(\mathbf{R}_n-\mathbf{R}_m)-i\mathbf{q}\cdot\mathbf{R}_m)}\langle i\mathbf{R}_m|\mathbf{v}_{\mathbf{q},\lambda}\cdot\nabla H_0(r)|j\mathbf{R}_n\rangle,$$

where the (ij)-sum runs over combined atom site and orbital indices, and the (mn)-sum runs over repeated unit cells in the *HamiltonianDerivatives* calculation. Specifically the derivative of the Hamiltonian $\nabla H_0(r)$ is being calculated in the *HamiltonianDerivatives* object, where the subscript 0 indicates that the derivatives are only calculated for atoms in the unit cell with index 0.

[1] T.ăGunst, T.ăMarkussen, K.ăStokbro, and M.ăBrandbyge. First-principles method for electron-phonon coupling and electron mobility: Applications to two-dimensional materials. *Phys. Rev. B*, 93:035414, Jan 2016. doi:10.1103/PhysRevB.93.035414.

[2] K.ăKaasbjerg, K.ăS. Thygesen, and K.ăW. Jacobsen. Phonon-limited mobility in n-type single-layer MoS2 from first principles. *Phys. Rev. B*, 85:115317, Mar 2012. doi:10.1103/PhysRevB.85.115317.

Spin life time calculations

The spin life time is an important parameter for spintronics devices. At technologically relevant temperatures (> 100 K) the spin life time will, in many materials, be limited by electron-phonon interactions. These are mediated by spin-orbit coupling through the so-called Elliot-Yafet mechanism. The implementation in QuantumATK follows the theory in Ref.[3]. The phonon-limited spin life time can be calculated from an *ElectronPhononCoupling* object, provided that the calculator attached to the configuration, and used for the *HamiltonianDerivatives* object, has spin set to Noncollinear Spin Orbit. The inverse spin life time is calculated as:

```
# Calculate inverse spin life time
average_inv_time, all_inv_times = electron_phonon_coupling.inverseSpinLifeTime(
    phonon_modes=All,
    electron_bands=All,
    temperature=300*K,
    fermi_shift=0.0*eV,
    integration_method=TetrahedronMethod)

# Spin life time
T_1 = 1/average_inv_time
```

The function returns two physical quantities of unit inverse time. The first is a band-averaged inverse relaxation time given by

$$\langle\frac{1}{T_1}\rangle = \frac{\sum_n \int_0^{k_F} [1/T_1(n,k)\frac{\partial f}{\partial \epsilon}|_{\epsilon=\epsilon_F}]d^3k}{\sum_n \int_0^{k_F} [\frac{\partial f}{\partial \epsilon}|_{\epsilon=\epsilon_F}]d^3k}$$

where $f = f(\epsilon_k)$ is the Fermi-Dirac distribution function, and the band- and k-resolved inverse spin life times are calculated as

$$\frac{1}{T_1(n,k)} = \frac{2\pi}{\hbar}\sum_{q\lambda n'} |g^{q\lambda}_{k+qn'\uparrow,kn\downarrow}|^2\{[f(\epsilon_{k+q,n'}) + n_{q\lambda}]\delta(\epsilon_{k,n} - \epsilon_{k+q,n'} + \hbar\omega_{q\lambda}) + [1 + n_{q\lambda} - f(\epsilon_{k+q,n'})]\delta(\epsilon_{k,n} - \epsilon_{k-q,n'} - \hbar\omega_{q\lambda})\}$$

where $\hbar\omega_{q\lambda}$ are the phonon energies for phonon mode λ and momentum q with Bose-Einstein occupation factors $n_{q\lambda}$. Notice that only matrix elements between spin up states ($\uparrow$) and spin-down states ($\downarrow$) are included.

The sum (integral) over q can be performed in two different ways, determined by the input variable `integration_method` which can either be `TetrahedronMethod` for a tetrahedron integration or `GaussianBroadening(0.01*eV)` for a gaussian integration, where the δ-functions are replaced by gaussians with a certain broadening (in this case `0.01 eV`). For three-dimensional sampling the `TetrahedronMethod` converges significantly faster, thus reducing the required number of q-points. For one-dimensional sampling, however, the GaussianBroadening is recomended.

[3] O.ăD. Restrepo and W.ăWindl. Full First-Principles Theory of Spin Relaxation in Group-IV Materials. *Phys. Rev. Lett.*, 109:166604, 2012. doi:10.1103/PhysRevLett.109.166604.

ElectronicInverseParticipationRatio

class ElectronicInverseParticipationRatio(*configuration*, *kpoints=None*, *energy_zero_parameter=None*)

Class for calculating the electronic inverse participation ratio.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration for which to calculate the electronic inverse participation ratio.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – The k-points on which to calculate the electronic inverse participation ratio. **Default:** The Gamma point (0.0, 0.0, 0.0).
- **energy_zero_parameter** (`FermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `FermiLevel`

energies(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the energies should be returned for. **Default:** `Spin.All`

Returns
The energies for each orbital.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy reference in absolute energy.

Return type
PhysicalQuantity of type energy.

fermiLevel(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. **Default:** `Spin.All`

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

inverseParticipationRatio(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the inverse participation ratio should be returned for. **Default:** `Spin.All`

Returns
The electronic inverse participation ratio for each orbital and k-point. The array has size (n_spins, n_kpoints, n_orbitals)

Return type
ndarray

kpoints()

Returns
The k-points on which to calculate the electronic inverse participation ratio.

Return type
ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

printTable(*stream=None*, *energy_min=None*, *energy_max=None*)

Print a string containing an ASCII table with the electronic inverse participation ratio data, restricted in an energy window.

Parameters

- **stream** (*A stream that supports strings being written to using 'write'.*) – The stream to write to.
- **energy_min** (*PhysicalQuantity* of type energy) – Lower limit of the energy window with respect to the energy reference. **Default:** -5 * eV
- **energy_max** (*PhysicalQuantity* of type energy) – Upper limit of the energy window with respect to the energy reference. **Default:** +5 * eV

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *ElectronicInverseParticipationRatio* at the Gamma point for silicon bulk with a vacancy defect and print out information for the most localized state:

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[0.,   0.,   0.  ],
                          [0.25, 0.25, 0.25]]

# Set up configuration
unit_configuration = BulkConfiguration(
```

(continues on next page)

(continued from previous page)

```
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
reps = 3
bulk_configuration = unit_configuration.repeat(reps, reps, reps)
bulk_configuration = bulk_configuration.copyAndDeleteAtoms(5)

# Setup the calculator.
calculator = LCAOCalculator(
    exchange_correlation=GGA.PBE,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
        k_point_sampling=MonkhorstPackGrid(1, 1, 1))
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Compute the electronic inverse participation ratio
electronic_ipr = ElectronicInverseParticipationRatio(
    configuration=bulk_configuration
    )

print('\nNumber of atoms N:', bulk_configuration.numberOfAtoms())
print(f'Expected IPR for delocalized states 1/N: {1 / bulk_configuration.
↪numberOfAtoms():.3f}')

# Obtain results
energies = electronic_ipr.energies().inUnitsOf(eV)
ipr = electronic_ipr.inverseParticipationRatio()
idx = numpy.argmax(ipr)
print(f'State at energy {energies.ravel()[idx]:.2f} eV',
      f'has the highest IPR: {ipr.ravel()[idx]:.3f}')
```

electronic_ipr.py

Notes

The following definition is used to calculate the electronic inverse participation ratio for a system of N atoms[1]

$$p_i^{(n)} = \sum_{i=1}^{N} |q_i^{(n)}|^2,$$
$$q_i^{(n)} = \sum_{\mu \in i, \nu} \mathrm{Re}[S_{\mu\nu} c_{\mu n}^* c_{\nu n}]$$

where $\mathbf{S}$ is the overlap matrix and $c_{\mu n}$ are the n-th wavefunction coefficients in the LCAO basis. The inverse participation ratio is $1/N$ for completely delocalized states and greater for localized states. The current definition does not

[1] S.ăI. Simdyankin, S.ăR. Elliott, Z.ăHajnal, T.ăA. Niehaus, and Th. Frauenheim. Simulation of physical properties of the chalcogenide glass As2S3 using a density-functional-based tight-binding method. *Physical Review B*, Apr 2004. URL: http://dx.doi.org/10.1103/PhysRevB.69.144202, doi:10.1103/physrevb.69.144202.

pose an upper limit to the value of the inverse participation ratio.

Note: The electronic inverse participation ratio can only be computed for configurations with *LCAOCalculator* or *SemiEmpiricalCalculator*, since the current definition is based on a localized basis orbital representation.

ElectrostaticDifferencePotential

class ElectrostaticDifferencePotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the electrostatic difference potential for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the electrostatic difference potential should be calculated.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** The density mesh cutoff set on the calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

`derivatives`(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy Œ charge^{-1} Œ length^{-1}

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy Œ charge^{-1}

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electrostatic difference potential is calculated for, always Spin.All.

Return type
Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the electrostatic difference potential and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )
```

(continues on next page)

(continued from previous page)

```
# Define a calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the electrostatic difference potential
potential = ElectrostaticDifferencePotential(molecule_configuration)
nlsave('electrostatic_difference_pot.nc', potential)
```

nh3_electrostatic_difference_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the electrostatic difference potential $\delta V_{\mathbf{E}}(\mathbf{r})$, as defined in *The Hartree potential and the electrostatic potential*.

ElectrostaticPotential

class ElectrostaticPotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the electrostatic potential for a configuration.

Parameters

•**configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration for which the electrostatic potential should be calculated.

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** The density mesh cutoff set on the calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy Œ charge^{-1} Œ length^{-1}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The value at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy Œ charge^{-1}

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

•**i** (*int*) – The grid index in the A direction.

•**j** (*int*) – The grid index in the B direction.

•**k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the electrostatic potential is calculated for, always `Spin.All`.

Return type
`Spin.All`

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the electrostatic potential and save it to a file:

```
# Set up a configuration.
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define a calculator.
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the electrostatic potential.
potential = ElectrostaticPotential(molecule_configuration)
nlsave('electrostatic_pot.nc', potential)
```

nh3_electrostatic_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the electrostatic potential $V_{\mathbf{E}}(\mathbf{r})$, as defined in *The Hartree potential and the electrostatic potential*.

EmtPotential

class EmtPotential(*particleType*, *E0*, *s0*, *V0*, *eta2*, *kappa*, *l*, *nu0*)

Constructor of the potential

Parameters

•**particleType** (*string*) – Identifier of the particle type.

•**E0** (*PhysicalQuantity of type energy*) – Potential parameter.

•**s0** (*PhysicalQuantity of type length*) – Potential parameter.

•**V0** (*PhysicalQuantity of type energy*) – Potential parameter.

•**eta2** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**kappa** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**l** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**nu0** (*PhysicalQuantity of type length**-3*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameter of the current potential, returns a dictionary consisting of the parameter name and default value.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a EmtPotential for a copper crystal.

```
# Set up lattice
lattice = FaceCenteredCubic(3.61496*Angstrom)

# Define elements
elements = [Copper]

# Define coordinates
fractional_coordinates = [[0.0, 0.0, 0.0]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

potential_set = TremoloXPotentialSet(name='EMT_Potential')

potential_set.addParticleType(ParticleType(
    symbol='Cu',
    mass=63.546*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=29,
    tags=[]
))

# Define a new EMT potential and add it to the potential set.
```

(continues on next page)

(continued from previous page)

```
emt_potential = EmtPotential(
    ParticleIdentifier('Cu', []),
    -3.51*eV,
    2.67*Bohr,
    2.476*eV,
    1.652*1/Bohr,
    2.74*1/Bohr,
    1.906*1/Bohr,
    0.0091*1/Bohr**3,
)
potential_set.addPotential(emt_potential)

# Set up a new TremoloXCalculator and attach it to the configuration.
calculator = TremoloXCalculator(potential_set)
bulk_configuration.setCalculator(calculator)
```

Notes

This potential implements *Effective Medium Theory (EMT)*[1]. The basic idea of EMT is to calculate the energy of an atom in an arbitrary environment by first calculating it in some properly chosen reference system, the effective medium, and then estimate the energy difference between the real system and the reference system.

The energy function is given by

$$E = \sum_i E_{c,i}(n_i) + \Delta E_{AS}(i)$$

where E_c represents the cohesive energy and it is defined as

$$E_c(s_i) = E_0 f[\lambda(s_i - s_0)]$$

with

$$f(x) = (1 + x)\exp(-x)$$

$$s_i = s_0 - \frac{1}{\beta\eta_2}\log\left(\frac{\sigma_{1,i}}{12}\right)$$

$$\sigma_{1,i} = \frac{1}{\gamma_1}\sum_{j\neq i}\exp[-\eta_2(r_{ij} - \beta s_0)]\chi_{ij}\Theta(r_{ij})$$

$$\beta = \frac{(16\pi/3)^{1/3}}{\sqrt{2}}$$

$$\Theta(x) = (1 + \exp(a(r - r_c)))$$

ΔE_{AS} represents the atomic-sphere correction, and it is defined as

$$\Delta E_{AS}(i) = \frac{1}{2}\left[\sum_{j\neq i}\chi_{ij}V_{ij}(r_{ij}) - 12V(\beta s_i)\right]$$

[1] K.ăW. Jacobsen, P.ăStoltze, and J.ăK. Norskov. A semi-empirical effective medium theory for metals and alloys. *Surf. Sci.*, 366(2):394–402, oct 1996. doi:10.1016/0039-6028(96)00816-3.

with

$$V(r_{ij}) = -\frac{V_0}{\gamma_2} \exp\left[-\frac{\kappa}{\beta}(r_{ij} - \beta s_0)\right] \Theta(r_{ij})$$

The normalization constants γ_1 and γ_2 are chosen such that $E_c = E_0$ and $\Delta E_{AS} = 0$ in the equilibrium reference system. The factor $\chi_{ij} = n_{0,i}/n_{0,j}$ corrects for interactions between particles of different type. Last, the cutoff radius r_c is chosen based on the largest s_0 present in the simulation and the constant *a* defines the steepness of the cutoff. The remaining parameters are given by the user.

EndToEndDistances

class EndToEndDistances(*md_trajectory*, *polymer_tags=None*, *end_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *gui_worker=None*)

Create an analyzer to calculate the end to end distance of polymer chains.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or AtomicConfiguration that contains the polymers.
- **polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used. Tags can also be set automatically with the flag Automatic
- **end_tags** (*list of type str | None*) – A list of tags for the ends of each polymer chain. If not specified default tags are used.
- **start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

autocorrelation()

Returns
The end to end vector autocorrelation function. This is returned as a 2D array indexed by polymer chain and then lag time.

Return type
PhysicalQuantity of type length

distances()

Returns
The end to end distance for each polymer chain. This is returned as a 2D array indexed by trajectory image then polymer chain.

Return type
PhysicalQuantity of type length

polymerTags()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

times()

Returns
Times for each image used in the analysis.

Return type
PhysicalQuantity of type time

vectors()

Returns
The end to end vector for each polymer chain. This is returned as a 3D array indexed by trajectory image, polymer chain then vector components.

Return type
PhysicalQuantity of type length

Usage Examples

Calculate the end-to-end distance and distance vector autocorrelation function of polystyrene chains during a molecular dynamics simulation starting from a well equilibrated configuration.

```
# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('Polystyrene.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)
```

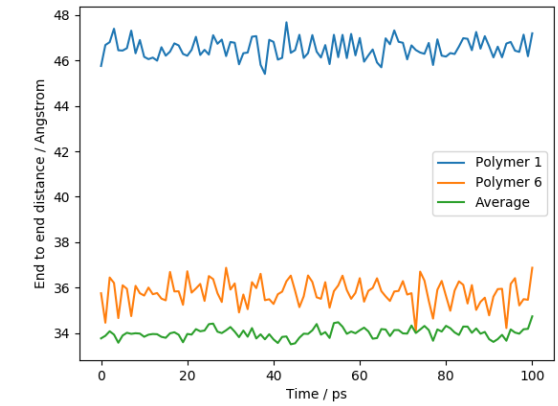
(continues on next page)

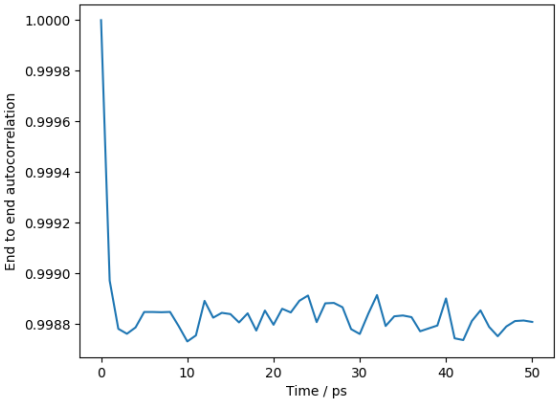
(continued from previous page)

```
constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Polystyrene_Trajectory.hdf5',
    steps=100000,
    log_interval=10,
    trajectory_interval=1000,
    method=method
)

# ------------------------------------------------------------
# Calculate End to End Distance
# ------------------------------------------------------------
analyzer = EndToEndDistances(
      md_trajectory,
      start_time=0*ps,
      end_time=100*ps,
      end_tags=['HEAD_CONNECT', 'TAIL_CONNECT']
)
end_end_distance = analyzer.distances()
end_end_distance_average = end_end_distance.mean(axis=1)
end_end_correlation = analyzer.autocorrelation().mean(axis=0)
time = analyzer.times().convertTo(ps)

# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
import pylab
pylab.figure()
pylab.plot(time, end_end_distance[:, 0], label='Polymer 1')
pylab.plot(time, end_end_distance[:, 5], label='Polymer 6')
pylab.plot(time, end_end_distance_average, label='Average')
pylab.xlabel('Time / ps')
pylab.ylabel('End to end distance / Angstrom')
pylab.legend()
pylab.savefig('End_To_End_Distances_Plot.png')

pylab.figure()
pylab.plot(time[:51], end_end_correlation[:51])
pylab.xlabel('Time / ps')
pylab.ylabel('End to end autocorrelation')
pylab.savefig('End_To_End_Autocorrelation_Plot.png')
```

`EndToEndDistances_Example.py` `Polystyrene.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate the end to end distances of two individual polymers, as well as the average. The end to end vector autocorrelation is also calculated. These are shown in the following two plots.





Note that the high value of the end to end vector autocorrelation indicates that there is little overall movement in the polymer chains, as the configuration is below the glass transition temperature.

Notes

This class enables the calculation of the end-to-end distances of different polymer chains, as well as their autocorrelation functions from a *MDTrajectory* or *BulkConfiguration*.

The end-to-end distance is simply defined as the Cartesian distance between the two end backbone atoms in the polymer chain. For this calculation, the polymer chain is unwrapped from the simulation cell and considered as one complete molecule. Given that this distance is a vector quantity, an autocorrelation function can be calculated that measures the evolution of the direction of this vector. The end-to-end autocorrelation function $R_{ee}(t)$ can be defined as:

$$R_{ee}(t) = \left\langle \frac{\mathbf{d}_0 \cdot \mathbf{d}_t}{|\mathbf{d}_0||\mathbf{d}_t|} \right\rangle$$

To calculate each of these shape measures a *EndToEndDistances* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration* and calculates the end to end distances and autocorrelation function for the specified configurations. The specific frames in the *MDTrajectory* to analyzed can be selected using the `start_time`, `end_time` and `time_resolution` arguments. The polymer chains analyzed in the configuration can be specified with the `polymer_tags` argument. This takes a list of tags of polymer molecules in the configuration. By default polymers tagged with `POLYMER_MOLECULE_#` are analyzed, where `#` is a numerical index. Tags indicating the start and end of each polymer chain also need to be added. The argument `end_tags` takes a list of strings that contain the tags for the end

atoms. These tags can be specified in any order. By default the tags HEAD_CONNECT and TAIL_CONNECT are used. All the default polymer and end tags are placed on polymer configurations built with the *PolymerMonteCarloBuilder*.

Once the *EndToEndDistances* object is created and the values of each of the descriptors has been calculated, the values can be queried using the remaining methods. The method `vectors()` returns a list of the end-to-end vectors, indexed by trajectory image, then polymer chain and finally vector components. Similarly the method `distances()` returns a list of the scalar end-to-end distances, indexed by trajectory image and then polymer chain. The method `autocorrelation()` returns a list of the autocorrelation functions, indexed first by polymer chain and then by lag time.

EquilibriumContour

class EquilibriumContour(*args*)

Check the parameters given, and which of them are using the default value.

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

This class is the abstract base class for the different Contour types that are used to compute the equilibrium density matrix.

At this point there are two different EquilibriumContours available: *OzakiContour* and *SemiCircleContour*.

EquivalentBulk

class EquivalentBulk(*electrode_constraint_length=None*, *iteration_control_parameters=None*, *algorithm_parameters=None*)

Class for representing an Equivalent Bulk initial Density Matrix for Device simulations.

Parameters

- **electrode_constraint_length** (*PhysicalQuantity* of type length) – Length over which the atoms in the electrode copy within the central region will be initialized with a linear combination of the converged density matrix of the electrodes and the equivalent bulk density matrix. The weight of the electrode density matrix is 1 at the boundary and 0 at the distance `electrode_constraint_length` from the boundary. **Default:** `10.0 * Angstrom`
- **iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the equivalent bulk run. It is not allowed to use `NonSelfconsistent` for *EquivalentBulk*.
- **algorithm_parameters** (*AlgorithmParameters*) – The *AlgorithmParameters* used for the equivalent bulk.

algorithmParameters()

Returns
The algorithm parameters.

Return type
AlgorithmParameters

electrodeConstraintLength()

Returns
The electrode constraint length.

Return type
PhysicalQuantity of type length

initializeCentralRegionDensityMatrix(*device_builder*, *downgraded_calculator*, *schroedinger_container*)

Method for initializing the density matrix of the central region.

With this initial density type, the Hamiltonian is also initialized. This function carries out a self-consistent calculation of the isolated central region of the device and copies over the final density matrix to the initial density matrix used in the device calculation.

Parameters

- **device_builder** (DeviceLCAOBuilder | DeviceSemiEmpiricalBuilder) – The device builder.
- **downgraded_calculator** (*LCAOCalculator* | *HuckelCalculator* | *SlaterKosterCalculator*) – The calculator to use for the isolated central region calculation.
- **schroedinger_container** (NLEngine.CompositeSchroedingerContainer) – The device Schroedinger container, the Hamiltonian and density matrix of which will be initialized.

iterationControlParameters()

Returns
The iteration control parameters.

Return type
IterationControlParameters

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Setup a self-consistent calculation where the initial density matrix is constructed by first performing a self-consistent bulk calculation of the central region, i.e. a calculation with periodic boundary conditions in the C-direction.

The initial density matrix elements of atoms within a distance of 7 Angstrom from the electrodes are set to the electrode density matrix elements.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    initial_density_type=EquivalentBulk(electrode_constraint_length=7.*Ang))
    calculator=DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    )
```

Notes

•The *EquivalentBulk* initial density type often gives a very good initial guess for the density matrix. It can also be used when the central region is not periodic.

•The *EquivalentBulk* density can also be used for obtaining the transmission spectrum, by setting the *IterationControlParameters* of the :ref:` DeviceLCAOCalculator_c` to `NonSelfconsistent_c`. This approach is usually very accurate and fast for systems at zero bias.

•If the `electrode_constraint_length` is longer than the length of the electrode, still only the electrode copy atoms within the central region will be affected. However, the weighting of the electrode density matrix will be larger. For instance, if `electrode_constraint_length` is infinite, the initial density matrix of the electrode copy atoms in the central region will be the electrode density matrix.

EvolutionarySQS

class EvolutionarySQS(*alloy*, *cluster_max_diameters*, *repetitions=None*, *seed_configuration=None*, *number_of_generations=None*, *population_size=None*, *number_to_promote=None*, *heredity_probability=0.5*, *permutation_probability=0.5*, *n_best=10*, *rng=None*)

Generate a special quasi-random structure (SQS) using genetic optimization.

Parameters

•**alloy** (*AlloyConfiguration*) – An alloy configuration.

•**cluster_max_diameters** (*dict*) – A dictionary that maps the cluster order to the maximum diameter for clusters of that order. They keys should be integers greater than equal to two and the values are positive PhysicalQuantities with units of distance (typically at least as large as the nearest neighbor distance).

•**repetitions** (*list of type int*) – The number of times to repeat the alloy configuration. **Default:** (1, 1, 1)

•**seed_configuration** (*BulkConfiguration*) – An initial guess of the SQS structure.

•**number_of_generations** (*int*) – The number of times the population should be evolved.

•**population_size** (*int*) – The number of individuals which are part of a population. The population size must be at least 5, but it is recommeded to use a large population size (>= 20) to obtain higher quality results.

•**number_to_promote** (*int*) – The number of most fittest individuals to promote to the next generation. **Default:** 2

•**heredity_probability** (*float*) – The unnormalized probabilty of using the heredity operator. **Default:** 0.5

•**permutation_probability** (*float*) – The unnormalized probabilty of using the permutation operator. The permutation operator will only be used when there is more than one type of an element in the system. **Default:** 0.5

•**n_best** (*int*) – The n best structures the correlation functions of which are closest to the truely random one.

•**rng** (*numpy.random.RandomState*) – The random number generated used to generate the random number stream. **Default:** A new random number generator (seeded from OS entropy pool)

activeLearning()

Returns
The active learning object, if specified, otherwise None.

Return type
ActiveLearningSimulation | None

alloyConfiguration()

Returns
The alloy configuration. This is the same as the alloy configuration that was initially passed in.

Return type
AlloyConfiguration

bestStructure()

Returns
The SQS structure that best matches the random correlation function.

Return type
BulkConfiguration

bestStructures()

Returns
A list of the configurations sorted in order of increasing error.

Return type
list of type BulkConfiguration

evolvePopulation(*number_of_generations*)

Evolves the current population for `number_of_generations` generations.

Parameters
number_of_generations (*int*) – The number of times the genetic algorithm is applied to the population.

fitnesses()

Returns
Return the fitnesses of the current population sorted by descending fitness.

Return type
list of type float

kpointDensity()

Returns
The k-point density.

Return type
PhysicalQuantity of type length | None

objectiveFunction(*configuration*)

Evaluate the objective function for a configuration.

Parameters
configuration (BulkConfiguration) – The configuration to evaluate.

Returns
The SQS objective function.

Return type
float

population()

Returns
Return all configurations in the population sorted by descending fitness.

Return type
Population

symmetries()

Return the international symbol for each individual of the current population sorted by descending fitness.

Returns
a list of international symbols

Return type
list of type str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up an evolutionary SQS optimization for a 64 atom SiGe alloy.

```
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 21.7224]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define site types
site_a = AlloySite(Germanium=0.5, Silicon=0.5)

# Define sites
sites = [
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
    site_a, site_a, site_a, site_a, site_a, site_a, site_a, site_a,
]

# Define coordinates
cartesian_coordinates = [[  0.     ,   0.     ,   0.     ],
                         [  0.     ,   0.     ,   5.4306 ],
                         [  0.     ,   0.     ,  10.8612 ],
                         [  0.     ,   0.     ,  16.2918 ],
                         [  1.35765,   1.35765,   1.35765],
                         [  1.35765,   1.35765,   6.78825],
                         [  1.35765,   1.35765,  12.21885],
                         [  1.35765,   1.35765,  17.64945],
                         [  2.7153 ,   2.7153 ,   0.     ],
                         [  2.7153 ,   2.7153 ,   5.4306 ],
                         [  2.7153 ,   2.7153 ,  10.8612 ],
                         [  2.7153 ,   2.7153 ,  16.2918 ],
                         [  4.07295,   4.07295,   1.35765],
                         [  4.07295,   4.07295,   6.78825],
                         [  4.07295,   4.07295,  12.21885],
                         [  4.07295,   4.07295,  17.64945],
                         [  2.7153 ,   0.     ,   2.7153 ],
```

(continues on next page)

(continued from previous page)

```
                          [  2.7153 ,   0.     ,   8.1459 ],
                          [  2.7153 ,   0.     ,  13.5765 ],
                          [  2.7153 ,   0.     ,  19.0071 ],
                          [  4.07295,   1.35765,   4.07295],
                          [  4.07295,   1.35765,   9.50355],
                          [  4.07295,   1.35765,  14.93415],
                          [  4.07295,   1.35765,  20.36475],
                          [  0.     ,   2.7153 ,   2.7153 ],
                          [  0.     ,   2.7153 ,   8.1459 ],
                          [  0.     ,   2.7153 ,  13.5765 ],
                          [  0.     ,   2.7153 ,  19.0071 ],
                          [  1.35765,   4.07295,   4.07295],
                          [  1.35765,   4.07295,   9.50355],
                          [  1.35765,   4.07295,  14.93415],
                          [  1.35765,   4.07295,  20.36475]]*Angstrom


alloy_configuration = AlloyConfiguration(
    bravais_lattice=lattice,
    sites=sites,
    cartesian_coordinates=cartesian_coordinates,
)


# Define the maximum diameter of each cluster.
cluster_max_diameters = {
    2: 4.9*Angstrom,
    3: 4.4*Angstrom,
    4: 2.5*Angstrom,
}

# Perform the SQS optimization.
sqs = EvolutionarySQS(
    alloy_configuration,
    cluster_max_diameters,
    number_of_generations=200,
    population_size=100,
    number_to_promote=10,
    heredity_probability=0.5,
    permutation_probability=0.5,
)

# Extract and save the best SQS structure after the optimization
best_sqs_configuration = sqs.bestStructure()
nlsave('sqs_sige_optimized.hdf5', best_sqs_configuration)
```

sige_evolutionary_sqs.py

Notes

EvolutionarySQS can be used to generate an optimized random alloy, based on the special quasi-random structure (SQS) method[1].

The desired composition and structure of an alloy can be set up via the *AlloySite* and *AlloyConfiguration* classes, which are used as input to *EvolutionarySQS*.

QuantumATK uses a genetic algorithm (similar to *CrystalStructurePrediction*) to permute the positions of the atoms in order to obtain correlation functions that match the correlation functions of an ideal random alloy configuration. The objective function that is optimized is given by equation (3) in[Page 2852, 1]. For a given configuration the value of the objective function can be queried by the method `objectiveFunction()`. After the optimization, the bulk configuration with the best correlation functions can be extracted via the `bestStructure ()` method. This configuration can then be used in further calculations.

ExactExchangeParameters

class ExactExchangeParameters(*aux_basis_tolerance=None*, *number_of_waves=None*, *maximum_wave_vector=None*, *integral_tolerance=None*, *relative_screening_tolerance=None*, *use_admm=None*, *basis_set_admm=None*, *auxiliary_basis_set=None*)

Class for representing the parameters for setting the numerical accuracy of an exact exchange calculation.

Parameters

- **aux_basis_tolerance** (*dict of string to double, or just a single double used for all element.*) – The auxiliary basis is constructed by taking a product basis of the regular basis and throwing away functions that are linearly dependent up to this tolerance. **Default:** `1.0e-3`
- **number_of_waves** (*int*) – Parameter for the radial FFT. Will determine accuracy of the exchange integrals. **Default:** `1024`
- **maximum_wave_vector** (*double*) – Parameter for the radial FFT. Will determine accuracy of the exchange integrals. **Default:** `50.0`
- **integral_tolerance** (*double*) – The integral tolerance for throwing away integral contributions to the exchange matrix **Default:** `1.0e-4`
- **relative_screening_tolerance** – The tolerance used to throw away contributions to the exchange matrix. Contributions that have an estimated contribution below this tolerance is not calculated. The tolerance is relative to the exchange_integral_tolerance, thus the actual tolerance used is relative_screening_tolerance * exchange_integral_tolerance. The screening tolerance is smaller than the exchange_integral_tolerance due to accumulation of errors. **Default:** `1.0e-2`
- **use_admm** (*boolean*) – If true, we use the admm approximation for exact exchange.
- **basis_set_admm** (*BasisSet*) – If we use ADMM, we have a second smaller basis set for exchange, which is provided here.
- **auxiliary_basis_set** (`NLEngine.BasisSet`) – If this is provided, we use this as an auxiliary basis set and not the product space.

auxBasisTolerance()

[1] A.ăvanăde Walle, P.ăTiwary, M.ădeăJong, D.L. Olmsted, M.ăAsta, A.ăDick, D.ăShin, Y.ăWang, L.-Q. Chen, and Z.-K. Liu. Efficient stochastic generation of special quasirandom structures. *Calphad*, 42:13–18, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0364591613000540.

Returns
The auxiliary basis is constructed by taking a product basis of the regular basis and throwing away functions that are linearly dependent up to this tolerance.

Return type
dict of string to double, or just a single double used for all element.

`auxiliaryBasisSet()`

Returns
The auxiliary basis set used for PARI, if this is None, the auxiliary basis set used will be the product basis of the exchange basis set.

Return type
NLEngine.BasisSet

`basisSetAdmm()`

Returns
The Auxiliary Density Matrix Method (ADMM) exchange basis set.

Return type
list

`integralTolerance()`

Returns
The integral screening tolerance for throwing away integral contributions to the exchange matrix

Return type
double

`maximumWaveVector()`

Returns
Parameter for the radial FFT. Will determine accuracy of the exchange integrals.

Return type
double

`minAuxBasisTolerance()`

Returns
The auxiliary basis is constructed by taking a product basis of the regular basis and throwing away functions that are linearly dependent up to this tolerance.

Return type
double

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`numberOfWaves()`

Returns
Parameter for the radial FFT. Will determine accuracy of the exchange integrals.

Return type
int

relativeScreeningTolerance()

Returns
The integral screening tolerance for throwing away integral contributions to the exchange matrix

Return type
double

uniqueString()

Return a unique string representing the state of the object.

useAdmm()

Returns
True, then the Auxiliary Density Matrix Method (ADMM) is used.

Return type
bool

ExchangeCorrelation

class ExchangeCorrelation(*exchange*, *correlation*, *hubbard_term=None*, *number_of_spins=1*, *spin_orbit=None*, *dft_half_enabled=None*)

The *ExchangeCorrelation* class is used to define the type of exchange-correlation used in the calculation, and whether the calculation should be polarized or non-polarized.

Parameters

- **exchange** (Exchange) – The exchange to be used.
- **correlation** (Correlation) – The correlation to be used.
- **hubbard_term** (Onsite | OnsiteShell | Dual | DualShell | None) – The Hubbard term to add. **Default:** None.
- **number_of_spins** (1 | 2 | 4) – The number of spins to be used in the calculation. **Default:** 1.
- **spin_orbit** (*bool*) – If True spin-orbit coupling is considered. **Default:** False.
- **dft_half_enabled** (*bool*) – Whether or not DFT-1/2 should be enabled. **Default:** False.

cParameter(*grid_descriptor=None*)

return the param C.

Parameters
grid_descriptor (*NLEngine.GridDescriptor*) – The grid descriptor.

cloneAsPolarized()

Create a new functional with all the same settings, but polarized spin.

cloneAsSpinOrbit()

Create a new functional with all the same settings, but with Spin-Orbit coupling included.

correlation()

Returns
The correlation functional class

Return type
Correlation

densityCheckThreshold()

Returns
The density safe guard.

Return type
float

dftHalfEnabled()

Returns
Whether DFT-1/2 is enabled.

Type
bool

dielectricDependentHybridFunctional()

Returns
True, then a dielectric dependent hybrid functional is used.

Return type
bool

dielectricDependentHybridParameters()

Returns
The dielectric dependent hybrid hybrid parameters.

Return type
list

exchange()

Returns
The exchange functional class

Return type
Exchange

hubbardTerm()

Returns
The Hubbard term used.

Return type
Onsite | OnsiteShell | Dual | DualShell | None

maximumPotential()

Set maximum potential value allowed for the TB09. The value to use as the maximum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

minimumPotential()

Set minimum potential value allowed for the TB09. The value to use as the minimum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`nlprint(stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)`

Print a string representation of the *ExchangeCorrelation* instance.

Parameters
stream (A stream that supports strings being written to using *write*.) – The stream the exchange-correlation should be written to. **Default:** `sys.stdout`

`numberOfSpins()`

Returns
The number of spins.

Return type
`1 | 2 | 4`

`params()`

Get the parameters

`spinOrbit()`

Returns
Boolean determining if spin-orbit is enabled (True) or not (False).

Return type
bool.

`spinType()`

Returns
The spin type.

Return type
`NLEngine.POLARIZED | NLEngine.NONCOLLINEAR | NLEngine.UNPOLARIZED`

`uniqueString()`

Return a unique string representing the state of the object.

`unscreenedExchange()`

Whether the exchange functional in use is unscreened.

`useLocalCParameter()`

Returns
True, then a local c-parameter is used.

Return type
bool

Usage Examples

Define a spin-polarized LDA+U calculation for nickel with a Hubbard U of 4.6 eV on each nickel atom and using the **Dual** representation for the Hubbard term (see also *Dual representation*):

```
exchange_correlation = ExchangeCorrelation(
    exchange=DiracBloch,
    correlation=PerdewZunger,
    hubbard_term=Dual,
    number_of_spins=2
)
basis_set = [LDABasis.Nickel_SingleZeta(hubbard_u=[4.6, 0.0]*eV)]
calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    basis_set=basis_set
)
```

Below is a similar calculation as above, but using the **DualShell** representation for the Hubbard term. In the shellwise Hubbard representation, occupations are summed over all basis set orbitals that have the same angular momentum and the complete basis is orthogonalized (see also *DualShell representation*). The Hubbard U must be the same for all orbitals that have equal angular momentum:

```
exchange_correlation = ExchangeCorrelation(
    exchange=DiracBloch,
    correlation=PerdewZunger,
    hubbard_term=DualShell,
    number_of_spins=2
)
basis_set = [LDABasis.Nickel_DoubleZetaPolarized(hubbard_u=[4.6, 0.0, 4.6, 0.0, 0.0]*eV)]
calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    basis_set=basis_set
)
```

Notes

A complete list of available keywords for the `exchange` and `correlation` parameters is available in the section *References*.

Parameters for TB09-MGGA exchange-correlation

The *ExchangeCorrelation* class can take parameters. Currently, this is only relevant for the TB09 meta-GGA functional, which takes a parameter `c` (cf. *Meta-GGA*).

If no value is specified, the value of `c` is automatically computed according to Eq. (3) in[1]. The calculated value can then be recovered using the *calculateTB09C* function.

The following piece of code selects the TB09 functional with `c=1.0` (recovering the Becke–Johnson potental[2]):

[1] F.ăTran and P.ăBlaha. Accurate band gaps of semiconductors and insulators with a semilocal exchange-correlation potential. *Phys. Rev. Lett.*, 102:226401, 2009. doi:10.1103/PhysRevLett.102.226401.

[2] AxelăD. Becke and ErinăR. Johnson. A simple effective potential for exchange. *J. Chemi. Phys.*, 2006. doi:http://dx.doi.org/10.1063/1.2213970.

```
exchange_correlation = MGGA.TB09LDA(c=1.0)
calculator = LCAOCalculator(exchange_correlation=exchange_correlation)
```

Simulations of complex systems, e.g. interfaces or two-probe device systems with different materials in the electrodes and the central region, may require setting the c parameter to different values in the various parts of the system, e.g. one c for the left electrode, another c for the central region, and yet another c for the right electrode. In such cases, the c parameter for TB09-MGGA exchange-correlation functionals can be defined as a sequence of **BoxRegions** (cf. *BoxRegion*), as demonstrated in the following script. The configuration is a bulk configuration that consists of layers of iron, magnesium-oxide, and again iron. Indeed, it is the central region of a magnetic tunneling junction device configuration.

```
from QuantumATK import *

# Set up lattice
vector_a = [2.866, 0.0, 0.0]*Angstrom
vector_b = [0.0, 2.866, 0.0]*Angstrom
vector_c = [0.0, 0.0, 34.6764]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Iron, Iron, Iron, Iron, Iron, Iron, Iron, Iron, Magnesium,
            Oxygen, Oxygen, Magnesium, Magnesium, Oxygen, Oxygen, Magnesium,
            Magnesium, Oxygen, Iron, Iron, Iron, Iron, Iron, Iron, Iron,
            Iron]

# Define coordinates
fractional_coordinates = [[ 0.25          ,  0.25          ,  0.020662467846],
                          [ 0.75          ,  0.75          ,  0.061987403537],
                          [ 0.25          ,  0.25          ,  0.103312339228],
                          [ 0.75          ,  0.75          ,  0.144637274919],
                          [ 0.25          ,  0.25          ,  0.18596221061 ],
                          [ 0.75          ,  0.75          ,  0.227287146301],
                          [ 0.25          ,  0.25          ,  0.268612081992],
                          [ 0.75          ,  0.75          ,  0.309937017683],
                          [ 0.25          ,  0.25          ,  0.373380743099],
                          [ 0.75          ,  0.75          ,  0.373380743099],
                          [ 0.25          ,  0.25          ,  0.43669037155 ],
                          [ 0.75          ,  0.75          ,  0.43669037155 ],
                          [ 0.25          ,  0.25          ,  0.5           ],
                          [ 0.75          ,  0.75          ,  0.5           ],
                          [ 0.25          ,  0.25          ,  0.56330962845 ],
                          [ 0.75          ,  0.75          ,  0.56330962845 ],
                          [ 0.25          ,  0.25          ,  0.626619256901],
                          [ 0.75          ,  0.75          ,  0.626619256901],
                          [ 0.75          ,  0.75          ,  0.690062982317],
                          [ 0.25          ,  0.25          ,  0.731387918008],
                          [ 0.75          ,  0.75          ,  0.772712853699],
                          [ 0.25          ,  0.25          ,  0.81403778939 ],
                          [ 0.75          ,  0.75          ,  0.855362725081],
                          [ 0.25          ,  0.25          ,  0.896687660772],
                          [ 0.75          ,  0.75          ,  0.938012596463],
                          [ 0.25          ,  0.25          ,  0.979337532154]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
# c parameter regions.

c_region_0 = BoxRegion(
    0.9,
    xmin = 0.0*Angstrom, xmax = 2.866*Angstrom,
    ymin = 0.0*Angstrom, ymax = 2.866*Angstrom,
    zmin = 0.0*Angstrom, zmax = 12.13674*Angstrom,
)

c_region_1 = BoxRegion(
    1.4,
    xmin = 0.0*Angstrom, xmax = 2.866*Angstrom,
    ymin = 0.0*Angstrom, ymax = 2.866*Angstrom,
    zmin = 12.13674*Angstrom, zmax = 22.53966*Angstrom,
)

c_region_2 = BoxRegion(
    0.9,
    xmin = 0.0*Angstrom, xmax = 2.866*Angstrom,
    ymin = 0.0*Angstrom, ymax = 2.866*Angstrom,
    zmin = 22.53966*Angstrom, zmax = 34.6764*Angstrom,
)

c_regions = [c_region_0, c_region_1, c_region_2]

# Define the exchange correlation with the c regions.
exchange_correlation=MGGA.TB09LDA(c=c_regions)

# Define the calculator.
calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    )

# Attach the calculator on the configuration.
bulk_configuration.setCalculator(calculator)

# Calculate the exchange correlation potential.
exchange_correlation_potential = ExchangeCorrelationPotential(bulk_configuration)

# Save on file.
nlsave('Fe-MgO-Fe.nc', exchange_correlation_potential)
```

It is important that the regions defining the `c` parameter cover the entire simulation volume. The parameter is set to `c = 1.0` at all points that do not fall into any region. The difference in `c` between two adjacent regions should not be too

large, otherwise problems in the SCF convergence may occur. As a rule of thumb, the jump in the exchange-correlation potential between adjacent regions should be of the same order as the fluctuations of the exchange-correlation potential in a region of constant `c`.

Potential problems in spin polarized calculations

For molecular systems containing hydrogen, the default initial scaled spin values of 1.0 for a spin polarized calculation may cause the density of one of the spin components to be zero. For some of the GGA functionals, relying for their implementation on the Libxc library, there is a known issue for such systems. This has been confirmed using Becke88 exchange, but may also be present in other functionals. The zero-electron density causes numerical errors in the exchange-correlation potential, resulting in ill-behaved convergence. The practical solution to this issue is to explicitly set the initial scaled spin value to 0.999, or use a random initial spin.

Hubbard U

Further details of the Hubbard U correction can be found in *XC+U mean-field Hubbard term*.

DFT-1/2

Further details of the DFT-1/2 correction can be found in *DFT-1/2 method*.

Abbreviations

Most common exchange-correlation functionals may be specified through abbreviations of the format `XCFAMILY[U|Half].XCTYPE`, where `XCFAMILY` determines the *class* of approximation to use, e.g. `LDA` or `GGA`, and also determines if the calculation should be spin polarized (collinear or non-collinear), and if it should include spin-orbit coupling. The table below gives all possibilities for `XCFAMILY` without the Hubbard U or DFT-1/2 models.

Non-polarized	Spin-polarized	Non-collinear	Spin-orbit
LDA	LSDA	NCLDA	SOLDA
GGA	SGGA	NCGGA	SOGGA
MGGA	SMGGA	NCMGGA	SOMGGA

Note: Appending `U` or `Half` to `XCFAMILY` enables the use of the Hubbard U or DFT-1/2 model, respectively: e.g., `SGGAU` for spin-polarized GGA with Hubbard U parameters, `LDAHalf` for non-polarized LDA with DFT-1/2 parameters. `Half` may only be appended to `XCFAMILY` in the first two rows of the table (LDA and GGA functionals).

Also note that calculations including spin-orbit coupling, e.g. `SOGGA`, are inherently done using a non-collinear representation of electron spin, like with `NCGGA`, but add extra computational complexity.

The value of `XCTYPE` sets the specific parametrization of the functional type according to the abbreviations given in tables below, e.g. `GGA.PBE` for the PBE variant of GGA. The abbreviations are shorthand notations for a full, predefined *`ExchangeCorrelation`* class, creating well-known combinations of exchange and correlation functionals. Therefore, setting

```
exchange_correlation=LSDA.PZ
```

is identical to specifying

```
exchange_correlation=ExchangeCorrelation(
   exchange=DiracBloch,
   correlation=PerdewZunger,
   number_of_spins=2,
)
```

Table 4.7: Available abbreviations for LDA functionals. The exchange part is always DiracBloch.

XCTYPE	Correlation
Wigner	Parametrization
XA	SlatersXAlpha_c
RPA	RandomPhaseApproximation
HL	HedinLundqvist
PZ	PerdewZunger
PW	PerdewWang

Table 4.8: Available abbreviations for GGA functionals.

XCTYPE	Exchange	Correlation
PBE	PerdewBurkeErnzerhofExchange	PerdewBurkeErnzerhofCorrelation
RPBE	RevisedPerdewBurkeErnzerhofExchange	PerdewBurkeErnzerhofCorrelation
PBES	PerdewBurkeErnzerhofSolids	PerdewBurkeErnzerhofCorrelationSolids
BLYP	Becke88	LeeYangParr
XLYP	XuGoddard	LeeYangParr
PW91	PerdewWang91	PerdewWang91Correlation
BPW91	Becke88	PerdewWang91Correlation
BP86	Becke86	Perdew86

Table 4.9: Available abbreviations for MGGA functionals. Note that the correlation is LDA-based for TB09.

XCTYPE	Exchange	Correlation
TB09LDA	TB09Exchange	PerdewZunger
SCAN	SCANExchange	SCANCorrelation

Table 4.10: Available exchange functionals.

Exchange	Family
DiracBloch	LDA
Exchange1D	LDA
Exchange2D	LDA
AdamoBarone	GGA
ArmientoKummel	GGA
ArmientoMattsson	GGA
Bayesian	GGA
Becke86	GGA
Becke862D	GGA
Becke86MGC2D	GGA
Becke88	GGA
Becke882D	GGA
Becke88DionVanDerWaals	GGA
Becke88ProtonTransfer	GGA
BerlandHyldgaard	GGA
BPCCAC	GGA
C09xForRutgersChalmersVdW	GGA
ChiodoConstantinFabianoExchange	GGA
ConstantinAiryGas	GGA
ConstantinFabianoLaricchiaSalaExchange	GGA
DelCampoGazquezTrickeyVela	GGA
FabianoConstantinSalaExchange	GGA
FilatovThiel97	GGA
Gill96	GGA
GradientModifiedBecke86	GGA
HaasTranBlahaSchwarz	GGA
HammerHansenNorskov	GGA
HandyCohen	GGA
HJSB88	GGA
HJSB97X	GGA
HJSPBE	GGA
HJSPBESOL	GGA
KealTozer1	GGA
KressDePristo	GGA
LacksGordon93	GGA
LocalAiryGas	GGA
Madsen	GGA
MinisotaN12Exchange	GGA
ModifiedAdamoBarone	GGA
ModifiedFilatovThiel97	GGA
OuYangLevy2	GGA
PedrozaSilvaCapelleJSJR	GGA
PerdewBurkeErnzerhof2D	GGA
PerdewBurkeErnzerhofExchange	GGA
PerdewBurkeErnzerhofRegularizedExchange	GGA
PerdewBurkeErnzerhofSolids	GGA

continues on next page

Table 4.10 – continued from previous page

Exchange	Family
PerdewBurkeErnzerhofVanDerWaalsK1	GGA
PerdewBurkeErnzerhofVanDerWaalsOpt	GGA
PerdewWang86	GGA
PerdewWang86Refitted	GGA
PerdewWang91	GGA
RevisedBecke86	GGA
RevisedKressDePristo	GGA
RevisedPerdewBurkeErnzerhofExchange	GGA
SecondOrderGGA	GGA
SecondOrderGGA2011Exchange	GGA
ShortRangeGGA	GGA
ShortRangeGGASFAT	GGA
ShortRangePBE	GGA
SwartSolaBickelhauptDispersion	GGA
SwartSolaBickelhauptExchange	GGA
SwartSolaBickelhauptPBEExchange	GGA
TognettiCortonaAdamoExchange	GGA
VanLeeuwenBaerends	GGA
VanLeeuwenBaerendsModified	GGA
VelaMedelTrickey84GE	GGA
VelaMedelTrickey84PBE	GGA
VelaMedelTrickeyGE	GGA
VelaMedelTrickeyPBE	GGA
WuCohen	GGA
XuGoddard	GGA
BalancedLocalization	MGGA
GVT4Exchange	MGGA
LocalTauApproximation	MGGA
M06Local	MGGA
M11LExchange	MGGA
ManbyKnowles	MGGA
ManbyKnowlesB	MGGA
MinisotaMN12LExchange	MGGA
ModifiedTaoPerdewStaroverovScuseriaExchange	MGGA
PerdewKurthZupanBlaha	MGGA
SCANExchange	MGGA
SunXiaoBulikScuseriaPerdewMS1	MGGA
SunXiaoBulikScuseriaPerdewMS2	MGGA
SunXiaoRuzsinszkyMS0	MGGA
TaoPerdewStaroverovScuseriaExchange	MGGA
TaoPerdewStaroverovScuseriaRevisedExchange	MGGA
TB09Exchange	MGGA
TwoDimensionalPRHG	MGGA
TwoDimensionalPRHGCorrected	MGGA

Table 4.11: Available correlation functionals.

Correlation	Family
Attacalite	LDA
CasulaSorellaSenatore1D	LDA
Gombas	LDA
GunnarssonLundqvist	LDA
HedinLundqvist	LDA
Loos1D	LDA
ModifiedPerdewWang	LDA
ModifiedPerdewZunger	LDA
OrtizBallone	LDA
OrtizBallonePerdewWang	LDA
PerdewWang	LDA
PerdewWangRPA	LDA
PerdewZunger	LDA
PittalisRasanenMarques	LDA
ProynovSalahubModifiedLSD1	LDA
ProynovSalahubModifiedLSD2	LDA
RagotCortona	LDA
RandomPhaseApproximation	LDA
SlatersXAlpha	LDA
VonBarthHedin	LDA
VoskoWilkNussair	LDA
VoskoWilkNussair1	LDA
VoskoWilkNussair2	LDA
VoskoWilkNussair3	LDA
VoskoWilkNussair4	LDA
VoskoWilkNussairRPA	LDA
WignerParametrization	LDA
ArmientoMattssonCorrelation	GGA
ChiodoConstantinFabianoCorrelation	GGA
CohenHandy	GGA
ConstantinFabianoLaricchiaSalaCorrelation	GGA
ConstantinFabianoSalaZPBEINT	GGA
ConstantinFabianoSalaZPBESOL	GGA
ExtendedXuGoddard	GGA
FabianoConstantinSalaCorrelation	GGA
FilatovThiel97Correlation	GGA
LangrethMehl	GGA
LeeYangParr	GGA
MinisotaN12Correlation	GGA
MinisotaN12SXCorrelation	GGA
OneParameterB88	GGA
OneParameterG96	GGA
OneParameterPBE	GGA
OneParameterXAlpha	GGA
PedrozaSilvaCapelleJRGX	GGA
Perdew86	GGA
PerdewBurkeErnzerhofCorrelation	GGA
PerdewBurkeErnzerhofCorrelationSolids	GGA
PerdewBurkeErnzerhofRegularizedCorrelation	GGA
PerdewRuszinszkyCsonkaConstantin	GGA

continues on next page

Table 4.11 – continued from previous page

Correlation	Family
PerdewWang91Correlation	GGA
SecondOrderGGA2011Correlation	GGA
SwartSolaBickelhauptPBE	GGA
TognettiCortonaAdamo	GGA
TognettiCortonaAdamoRevised	GGA
WilsonIvanov	GGA
WilsonIvanovInitial	GGA
WilsonLevy	GGA
BC95Correlation	MGGA
CancioChou	MGGA
ColleSalvetti	MGGA
M06LocalCorrelation	MGGA
M11LCorrelation	MGGA
MinisotaMN12LCorrelation	MGGA
PerdewKurthZupanBlahaCorrelation	MGGA
SCANCorrelation	MGGA
TaoPerdewStaroverovScuseriaCorrelation	MGGA
TaoPerdewStaroverovScuseriaRevisedCorrelation	MGGA
VSXCCorrelation	MGGA

References

ATK uses the Libxc library for efficient implementations of exchange and correlation functionals. In the following tables, we list the parameterization that are available in QuantumATK, along with the corresponding entity used in the Libxc manual, and with references to the original articles.

Table 4.12: Available LDA exchange parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
DiracBloch	XC_LDA_X	Exchange. PAM Dirac, (Mathematical) Proceedings of the Cambridge Philosophical Society 26, 376 (1930) F Bloch, Zeitschrift fuer Physik 57, 545 (1929)
Exchange1D	XC_LDA_X_1D	Exchange in 1D N. Helbig, J. I. Fuks, M. Casula, M. J. Verstraete, M. A. L. Marques, I. V. Tokatly and A. Rubio, Phys. Rev. A 83, 032503 (2011)
Exchange2D	XC_LDA_X_2D	Exchange in 2D PAM Dirac, (Mathematical) Proceedings of the Cambridge Philosophical Society 26, 376 (1930) F Bloch, Zeitschrift fuer Physik 57, 545 (1929)

Table 4.13: Available LDA correlation parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
Attacalite	XC_LDA_C_2D_ (was XC_LDA_C_AM	Attacalite et al., for 2D systems C Attacalite et al., Phys. Rev. Lett. 88, 256601 (2002) C Attacalite, PhD thesis
CasulaSorellaSenatore1D	XC_LDA_C_1D_ (was XC_LDA_C_AM	Casula, Sorella, and Senatore 1D correlation M Casula, S Sorella, and G Senatore, Phys. Rev. B 74, 245427 (2006)
Gombas	XC_LDA_C_GOMBAS (was XC_LDA_C_AM	Gombas parametrization P. Gombas, Pseudopotentiale (Springer-Verlag, Wien, New York, 1967)
Gunnarsson-Lundqvist	XC_LDA_C_GL	Gunnarson & Lundqvist O Gunnarsson and BI Lundqvist, Phys. Rev. B 13, 4274 (1976)
HedinLundqvist	XC_LDA_C_HL	Hedin & Lundqvist L. Hedin and B.I. Lundqvist, , J. Phys. C 4, 2064 (1971)
Loos1D	XC_LDA_C_1D_	P-F Loos correlation LDA P-F Loos, arXiv:1207.6849v1 [cond-mat.str-el] (2012)
Modified-PerdewWang	XC_LDA_C_PW_	Perdew & Wang (Modified) Added extra digits to some constants as in the PBE routine (http://dft.uci.edu/pubs/PBE.asc) JP Perdew and Y Wang, Phys. Rev. B 45, 13244 (1992)
Modified-PerdewZunger	XC_LDA_C_PZ_	Perdew & Zunger (Modified) Modified to improve the matching between the low and high rs parts Perdew and Zunger, Phys. Rev. B 23, 5048 (1981)
OrtizBallone	XC_LDA_C_OB_	Ortiz & Ballone (PZ). G Ortiz and P Ballone, Phys. Rev. B 50, 1391 (1994) , G Ortiz and P Ballone, Phys. Rev. B 56, 9970(E) (1997) Perdew and Zunger, Phys. Rev. B 23, 5048 (1981)
OrtizBallonePerdewWang	XC_LDA_C_OB_	Ortiz & Ballone (PW) G Ortiz and P Ballone, Phys. Rev. B 50, 1391 (1994) G Ortiz and P Ballone, Phys. Rev. B 56, 9970(E) (1997) JP Perdew and Y Wang, Phys. Rev. B 45, 13244 (1992)
PerdewWang	XC_LDA_C_PW	Perdew & Wang JP Perdew and Y Wang, Phys. Rev. B 45, 13244 (1992)
PerdewWangRPA	XC_LDA_C_PW_	Perdew & Wang fit of the RPA JP Perdew and Y Wang, Phys. Rev. B 45, 13244 (1992)
PerdewZunger	XC_LDA_C_PZ	Perdew & Zunger Perdew and Zunger, Phys. Rev. B 23, 5048 (1981)
PittalisRasanen-Marques	XC_LDA_C_2D_ (was XC_LDA_C_PRM	Pittalis, Rasanen & Marques correlation in 2D S Pittalis, E Rasanen, and MAL Marques, Phys. Rev. B 78, 195322 (2008)
ProynovSalahub-ModifiedLSD1	XC_LDA_C_ML1 (was XC_LDA_C_PRM	Modified LSD (version 1) of Proynov and Salahub EI Proynov and D Salahub, Phys. Rev. B 49, 7874 (1994)
ProynovSalahub-ModifiedLSD2	XC_LDA_C_ML2 (was XC_LDA_C_PRM	Modified LSD (version 2) of Proynov and Salahub EI Proynov and D Salahub, Phys. Rev. B 49, 7874 (1994)
Random-PhaseApproximation	XC_LDA_C_RPA	Random Phase Approximation M Gell-Mann and KA Brueckner, Phys. Rev. 106, 364 (1957)
RagotCortona	XC_LDA_C_RC0	Ragot-Cortona S Ragot and P Cortona, J. Chem. Phys. 121, 7671 (2004)
SlatersXAlpha	XC_LDA_C_XALPHA	Slaters X-alpha JC Slater, Phys. Rev. 81, 385 (1951)
VonBarthHedin	XC_LDA_C_vBH	von Barth & Hedin U von Barth and L Hedin, J. Phys. C: Solid State Phys. 5, 1629 (1972)
VoskoWilkNusair	XC_LDA_C_VW	Vosko, Wilk, & Nussair SH Vosko, L Wilk, and M Nusair, Can. J. Phys. 58, 1200 (1980)
VoskoWilkNusair1	XC_LDA_C_VW	Vosko, Wilk, & Nussair (1) SH Vosko, L Wilk, and M Nusair, Can. J. Phys. 58, 1200 (1980)
VoskoWilkNusair2	XC_LDA_C_VW	Vosko, Wilk, & Nussair (2) SH Vosko, L Wilk, and M Nusair, Can. J. Phys. 58, 1200 (1980)
VoskoWilkNusair3	XC_LDA_C_VW	Vosko, Wilk, & Nussair (3) SH Vosko, L Wilk, and M Nusair, Can. J. Phys. 58, 1200 (1980)
VoskoWilkNusair4	XC_LDA_C_VW	Vosko, Wilk, & Nussair (4) SH Vosko, L Wilk, and M Nusair, Can. J. Phys. 58, 1200 (1980)

Table 4.14: Available GGA exchange parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
AdamoBarone	XC_GGA_X_mP'	Modified form of PW91 by Adamo & Barone C Adamo and V Barone, J. Chem. Phys. 108, 664 (1998)
ArmientoMattsson	XC_GGA_X_AM	Armiento & Mattsson 05 exchange R Armiento and AE Mattsson, Phys. Rev. B 72, 085108 (2005) AE Mattsson, R Armiento, J Paier, G Kresse, JM Wills, and TR Mattsson, J. Chem. Phys. 128, 084714 (2008)
Bayesian	XC_GGA_X_BA'	Bayesian best fit for the enhancement factor JJ Mortensen, K Kaasbjerg, SL Frederiksen, JK Norskov, JP Sethna, and KW Jacobsen, Phys. Rev. Lett. 95, 216401 (2005)
Becke86	XC_GGA_X_B86	Becke 86 AD Becke, J. Chem. Phys 84, 4524 (1986)
Becke862D	XC_GGA_X_2D_	Becke 86 in 2D. G Vilhena and MAL Marques, unpublished AD Becke, J. Chem. Phys 84, 4524 (1986)
Becke86MGC2D	XC_GGA_X_2D_	Becke 86 MGC for 2D systems S Pittalis, E Rasanen, JG Vilhena, and MAL Marques, Phys. Rev. A 79, 012503 (2009) AD Becke, J. Chem. Phys 85, 7184 (1986)
Becke88	XC_GGA_X_B88	Becke 88 AD Becke, Phys. Rev. A 38, 3098 (1988)
Becke882D	XC_GGA_X_2D_	Becke 88 in 2D. G Vilhena, MAL Marques, unpublished AD Becke, Phys. Rev. A 38, 3098 (1988)
Becke88DionVanDerWaals	XC_GGA_X_OP'	Becke 88 reoptimized to be used with vdW functional of Dion et al J Klimes, DR Bowler, and A Michaelides, J. Phys.: Condens. Matter 22, 022201 (2010)
Becke88ProtonTransfer	XC_GGA_X_MB	Modified Becke 88 for proton transfer V Tognetti and C Adamo, J. Phys. Chem. A 113, 14415-14419 (2009)
BPCCAC	XC_GGA_X_BPCCAC	BPCCAC (GRAC for the energy) E Bremond, D Pilard, I Ciofini, H Chermette, C Adamo, and P Cortona, Theor Chem Acc 131, 1184 (2012)
C09xForRutgersChalmersVdW	XC_GGA_X_C09	C09x to be used with the VdW of Rutgers-Chalmers VR Cooper, PRB 81, 161104(R) (2010)
ConstantinAiryGas	XC_GGA_X_AIR	Constantin et al based on the Airy gas LA Constantin, A Ruzsinszky, and JP Perdew, Phys. Rev. B 80, 035125 (2009)
FilatovThiel97	XC_GGA_X_FT9	Filatov & Thiel 97 (version A) M Filatov and W Thiel, Mol. Phys. 91, 847 (1997)
Gill96	XC_GGA_X_G96	Gill 96 PMW Gill, Mol. Phys. 89, 433 (1996)
GradientModifiedBecke86	XC_GGA_X_B86	Becke 86 Xalfa,beta,gamma (with mod. grad. correction) AD Becke, J. Chem. Phys 84, 4524 (1986) AD Becke, J. Chem. Phys 85, 7184 (1986)
HaasTranBlahaSchwarz	XC_GGA_X_HT	Haas, Tran, Blaha, and Schwarz P Haas, F Tran, P Blaha, and K Schwarz, Phys. Rev. B 83, 205117 (2011)
HammerHansenNorskov	XC_GGA_X_RPI	Hammer, Hansen & Norskov (PBE-like) B Hammer, LB Hansen and JK Noerskov, Phys. Rev. B 59, 7413 (1999)
HandyCohen	XC_GGA_X_OP'	Handy & Cohen OPTX 01 NC Handy and AJ Cohen, Mol. Phys. 99, 403 (2001)
HJSB88	XC_GGA_X_HJS	HJS screened exchange B88 version NTM Henderson, BG Janesko, and GE Scuseria, J. Chem. Phys. 128, 194105 (2008)
HJSB97X	XC_GGA_X_HJS	HJS screened exchange B97x version NTM Henderson, BG Janesko, and GE Scuseria, J. Chem. Phys. 128, 194105 (2008)
HJSPBE	XC_GGA_X_HJS	HJS screened exchange PBE version NTM Henderson, BG Janesko, and GE Scuseria, J. Chem. Phys. 128, 194105 (2008)

continues on next page

Table 4.14 – continued from previous page

Keyword	Libxc entity	Description and references
HJSPBESOL	XC_GGA_X_HJS	HJS screened exchange PBE_SOL version NTM Henderson, BG Janesko, and GE Scuseria, J. Chem. Phys. 128, 194105 (2008)
KealTozer1	XC_GGA_X_KT	Keal and Tozer version 1 TW Keal and DJ Tozer, J. Chem. Phys. 119, 3015 (2003)
KressDePristo	XC_GGA_X_DK	dePristo & Kress 87 (version R1) AE DePristo and JD Kress, J. Chem. Phys. 86, 1425 (1987)
LacksGordon93	XC_GGA_X_LG	Lacks & Gordon 93 DJ Lacks and RG Gordon, Phys. Rev. A 47, 4681 (1993)
LocalAiryGas	XC_GGA_X_LA	Local Airy Gas L Vitos, B Johansson, J Kollar, and HL Skriver, Phys. Rev. B 62, 10046-10050 (2000)
Madsen	XC_GGA_X_PBE	Madsen (PBE-like) G Madsen, Phys. Rev. B 75, 195108 (2007)
ModifiedAdamoBarone	XC_GGA_X_MP	Adamo & Barone modification to PBE C Adamo and V Barone, J. Chem. Phys., 116, 5933 (2002)
ModifiedFilatovThiel97	XC_GGA_X_FT	Filatov & Thiel 97 (version B) M Filatov and W Thiel, Mol. Phys. 91, 847 (1997)
PedrozaSilvaCapelleJSJR	XC_GGA_X_PBE	JSJR reparametrization by Pedroza, Silva & Capelle LS Pedroza, AJR da Silva, and K. Capelle, Phys. Rev. B 79, 201106(R) (2009)
PerdewBurkeErnzerhof2D	XC_GGA_X_2D_	Perdew, Burke & Ernzerhof exchange in 2D. G Vilhena and MAL Marques, unpublished JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997)
PerdewBurkeErnzerhofExchange	XC_GGA_X_PBE	Perdew, Burke & Ernzerhof exchange JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997)
PerdewBurkeErnzerhofRegularizedExchange	XC_GGA_X_RGE	Regularized PBE A Ruzsinszky, GI Csonka, and G Scuseria, J. Chem. Theory Comput. 5, 763 (2009)
PerdewBurkeErnzerhofSolids	XC_GGA_X_PBE	Perdew, Burke & Ernzerhof exchange (solids) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997) JP Perdew, et al., Phys. Rev. Lett. 100, 136406 (2008) arXiv:0707.2088v1
PerdewBurkeErnzerhofVanDerWaalsK1	XC_GGA_X_PBE	PBE reparametrization for vdW J Klimes, DR Bowler, and A Michaelides, J. Phys.: Condens. Matter 22, 022201 (2010)
PerdewBurkeErnzerhofVanDerWaalsOpt	XC_GGA_X_OP	PBE reparametrization for vdW J Klimes, DR Bowler, and A Michaelides, J. Phys.: Condens. Matter 22, 022201 (2010)
PerdewWang86	XC_GGA_X_PW	Perdew & Wang 86. JP Perdew and Y Wang, Phys. Rev. B 33, 8800 (1986)
PerdewWang86Refitted	XC_GGA_X_RPW	Refitted Perdew & Wang 86 ED Murray, K Lee and DC Langreth, J. Chem. Theory Comput. 5, 2754-2762 (2009)
PerdewWang91	XC_GGA_X_PW	Perdew & Wang 91 JP Perdew, JA Chevary, SH Vosko, KA Jackson, MR Pederson, and C Fiolhais, Phys. Rev. B 46, 6671 (1992)
RevisedBecke86	XC_GGA_X_B86	Becke 86 Xalfa,beta,gamma (reoptimized) AD Becke, J. Chem. Phys 84, 4524 (1986) AD Becke, J. Chem. Phys 107, 8554 (1997)
RevisedKressDePristo	XC_GGA_X_DK	dePristo & Kress 87 (version R2) AE DePristo and JD Kress, J. Chem. Phys. 86, 1425 (1987)

continues on next page

Table 4.14 – continued from previous page

Keyword	Libxc entity	Description and references
RevisedPerdew-BurkeErnzer-hofExchange	XC_GGA_X_PBE	Perdew, Burke & Ernzerhof exchange (revised) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997) Y Zhang and W Yang, Phys. Rev. Lett 80, 890 (1998)
Secon-dOrderGGA	XC_GGA_X_SO(	Second-order generalized gradient approximation Y Zhao and DG Truhlar, J. Chem. Phys. 128, 184109 (2008) Minnesota Functional Module
Secon-dOrderGGA2011I change	XC_GGA_X_SO(	Second-order generalized gradient approximation 2011 R Peverati, Y Zhao, and DG Truhlar, J. Phys. Chem. Lett. 2, 1991-1997 (2011) Minnesota Functional Module
Short-RangeGGA	XC_GGA_X_ITY	Short-range recipe for exchange GGA functionals H Iikura, T Tsuneda, T Yanai, and K Hirao, J. Chem. Phys. 115, 3540 (2001)
ShortRangePBE	XC_GGA_X_WP BEH	Short-range part of the PBE (default w=0 gives PBEh) J Heyd, GE Scuseria, and M Ernzerhof, J. Chem. Phys. 118, 8207 (2003)
SwartSolaBick-elhauptDisper-sion	XC_GGA_X_SSB	Swarta, Sola and Bickelhaupt dispersion M Swart, M Sola, and FM Bickel-haupt, J. Chem. Phys. 131, 094103 (2009)
SwartSolaB-ickelhauptEx-change	XC_GGA_X_SSB	Swarta, Sola and Bickelhaupt M Swart, M Sola, and FM Bickelhaupt, J. Chem. Phys. 131, 094103 (2009)
SwartSolaBick-elhauptPBEEx-change	XC_GGA_X_SSB	Swarta, Sola and Bickelhaupt correction to PBE M Swart, M Sola, and FM Bickelhaupt, J. Comp. Meth. Sci. Engin. 9, 69 (2009)
WuCohen	XC_GGA_X_WC	Wu & Cohen Z Wu and RE Cohen, Phys. Rev. B 73, 235116 (2006)
XuGoddard	XC_GGA_X_XPB	xPBE reparametrization by Xu & Goddard X. Xu and WA Goddard III, J. Chem. Phys., 121, 4068 (2004)

Table 4.15: Available GGA correlation parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
ArmientoMattsson-Correlation	XC_GGA_C_AM	Armiento & Mattsson 05 correlation R Armiento and AE Mattsson, Phys. Rev. B 72, 085108 (2005) AE Mattsson, R Armiento, J Paier, G Kresse, JM Wills, and TR Mattsson, J. Chem. Phys. 128, 084714 (2008)
CohenHandy	XC_GGA_C_OP	Optimized correlation functional of Cohen and Handy AJ Cohen and NC Handy, Mol. Phys. 99, 607-615 (2001)
ExtendedXu-Goddard	XC_GGA_C_XPI	xPBE reparametrization by Xu & Goddard X. Xu and WA Goddard III, J. Chem. Phys., 121, 4068 (2004)
FilatovThiel97Correlation	XC_GGA_C_FT9	Filatov & Thiel correlation M Filatov & W Thiel, Int. J. Quant. Chem. 62, 603-616 (1997) M Filatov & W Thiel, Mol Phys 91, 847 (1997)
LangrethMehl	XC_GGA_C_LM	Langreth and Mehl correlation DC Langreth and MJ Mehl, Phys. Rev. Lett. 47, 446 (1981)
LeeYangParr	XC_GGA_C_LYI	Lee, Yang & Parr C Lee, W Yang and RG Parr, Phys. Rev. B 37, 785 (1988) B Miehlich, A Savin, H Stoll and H Preuss, Chem. Phys. Lett. 157, 200 (1989)
OneParameterB88	XC_GGA_C_OP_	One-parameter progressive functional (B88 version) T Tsuneda, T Suzumura, and K Hirao, J. Chem. Phys. 110, 10664 (1999)
OneParameterG96	XC_GGA_C_OP_	One-parameter progressive functional (G96 version) T Tsuneda, T Suzumura, and K Hirao, J. Chem. Phys. 111, 5656 (1999)
OneParameterPBE	XC_GGA_C_OP_	One-parameter progressive functional (PBE version) T Tsuneda, T Suzumura, and K Hirao, J. Chem. Phys. 111, 5656 (1999)
OneParameterXAlpha	XC_GGA_C_OP_PHA	One-parameter progressive functional (XALPHA version) T Tsuneda, T Suzumura, and K Hirao, J. Chem. Phys. 111, 5656 (1999)
PedrozaSilvaCapelleJRGX	XC_GGA_C_PBE	JRGX reparametrization by Pedroza, Silva & Capelle LS Pedroza, AJR da Silva, and K. Capelle, Phys. Rev. B 79, 201106(R) (2009)
Perdew86	XC_GGA_C_P86	Perdew 86 JP Perdew, Phys. Rev. B 33, 8822 (1986)
PerdewBurkeErnzerhofCorrelation	XC_GGA_C_PBE	Perdew, Burke & Ernzerhof correlation JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997)
PerdewBurkeErnzerhofCorrelationSolids	XC_GGA_C_PBE	Perdew, Burke & Ernzerhof correlation SOL JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 77, 3865 (1996) JP Perdew, K Burke, and M Ernzerhof, Phys. Rev. Lett. 78, 1396(E) (1997) JP Perdew, et al., Phys. Rev. Lett. 100, 136406 (2008) arXiv:0707.2088v1
PerdewBurkeErnzerhofRegularizedCorrelation	XC_GGA_C_RGE	Regularized PBE A Ruzsinszky, GI Csonka, and G Scuseria, J. Chem. Theory Comput. 5, 763 (2009)
PerdewWang91Correlation	XC_GGA_C_PW	Perdew & Wang 91 JP Perdew, JA Chevary, SH Vosko, KA Jackson, MR Pederson, DJ Singh, and C Fiolhais, Phys. Rev. B 46, 6671 (1992) JP Perdew, JA Chevary, SH Vosko, KA Jackson, MR Pederson, DJ Singh, and C Fiolhais, Phys. Rev. B 48, 4978(E) (1993)
SecondOrderGGA2011Correlation	XC_GGA_C_SOC	Second-order generalized gradient approximation 2011 R Peverati, Y Zhao, and DG Truhlar, J. Phys. Chem. Lett. 2, 1991-1997 (2011) Minnesota Functional Module
SwartSolaBickelhauptPBE	XC_GGA_C_SPE	PBE correlation to be used with the SSB exchange M Swart, M Sola, and FM Bickelhaupt, J. Chem. Phys. 131, 094103 (2009)
TognettiCortonaAdamo	XC_GGA_C_TCA	Tognetti, Cortona, Adamo V Tognetti, P Cortona, and C Adamo, J. Chem. Phys. 128, 034101 (2008)
TognettiCortonaAdamoRevised	XC_GGA_C_REV	Tognetti, Cortona, Adamo (revised) V Tognetti, P Cortona, and C Adamo, Chem. Phys. Lett. 460, 536-539 (2008)
WilsonIvanov	XC_GGA_C_WI	Wilson & Ivanov LC Wilson & S Ivanov, Int. J. Quantum Chem. 69, 523-532 (1998)
WilsonIvanovInitial	XC_GGA_C_WIC	Wilson & Ivanov initial version LC Wilson & S Ivanov, Int. J. Quantum Chem. 69, 523-532 (1998)
WilsonLevy	XC_GGA_C_WL	Wilson & Levy LC Wilson and M Levy, Phys. Rev. B 41, 12930 (1990)

Table 4.16: Available MGGA exchange parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
M05	XC_MGGA_X_M	M05 functional of Minnesota Y Zhao, NE Schultz, and DG Truhlar, J. Chem. Phys. 123, 161103 (2005)
M052X	XC_MGGA_X_M	M05-2X functional of Minnesota Y Zhao, NE Schultz, and DG Truhlar, J. Chem. Theory Comput. 2, 364 (2006)
M06	XC_MGGA_X_M	M06 functional of Minnesota Y Zhao and DG Truhlar, Theor. Chem. Acc. 120, 215 (2008)
M062X	XC_MGGA_X_M	M06-2X functional of Minnesota Y Zhao and DG Truhlar, Theor. Chem. Acc. 120, 215 (2008)
M06HF	XC_MGGA_X_M	M06-HF functional of Minnesota Y Zhao and DG Truhlar, J. Phys. Chem. A 110, 13126 (2006)
M06Local	XC_MGGA_X_M	M06-Local functional of Minnesota Y. Zhao and D. G. Truhlar, J. Chem. Phys. 125, 194101 (2006) Y Zhao and DG Truhlar, Theor. Chem. Account 120, 215 (2008)
M08HX	XC_MGGA_X_M	M08-HX functional of Minnesota Y Zhao and DG Truhlar, J. Chem. Theory Comput. 4, 1849-1868 (2008)
M08SO	XC_MGGA_X_M	M08-SO functional of Minnesota Y Zhao and DG Truhlar, J. Chem. Theory Comput. 4, 1849-1868 (2008)
PerdewKurthZupanBlaha	XC_MGGA_X_P	Perdew, Kurth, Zupan, and Blaha JP Perdew, S Kurth, A Zupan, and P. Blaha, Phys. Rev. Lett. 82, 2544-2547 (1999)
SCANExchange	XC_MGGA_X_S	J. Sun, A. Ruzsinszky, and J. P. Perdew, Phys. Rev. Lett. 115, 036402 (2015)
TaoPerdewStaroverovScuseriaExchange	XC_MGGA_X_T	Perdew, Tao, Staroverov & Scuseria exchange JP Perdew, J Tao, VN Staroverov and GE Scuseria, Phys. Rev. Lett. 91, 146401 (2003) JP Perdew, J Tao, VN Staroverov and GE Scuseria, J. Chem. Phys. 120, 6898 (2004)
TaoPerdewStaroverovScuseriaRevisedExchange	XC_MGGA_X_R	Revised Perdew, Tao, Staroverov & Scuseria exchange JP Perdew, A Ruzsinszky, GI Csonka, LA Constantin, and J Sun, Phys. Rev. Lett. 103, 026403 (2009) JP Perdew, A Ruzsinszky, GI Csonka, LA Constantin, and J Sun, Phys. Rev. Lett. 106, 179902(E) (2011)
TB09Exchange	XC_MGGA_X_T	Tran & Blaha 89, meta-GGA F Tran and P Blaha, Phys. Rev. Lett. 102, 226401 (2009)

Table 4.17: Available MGGA correlation parametrizations in QuantumATK.

Keyword	Libxc entity	Description and references
PerdewZunger-Correlation	XC_MGGA_C_M	M06-Local functional of Minnesota Perdew and Zunger, Phys. Rev. B 23, 5048 (1981) Reused from PerdewZunger class in order to allow inclusion in MGGA.
SCANCorrelation	XC_MGGA_C_S	J. Sun, A. Ruzsinszky, and J. P. Perdew, Phys. Rev. Lett. 115, 036402 (2015)
TaoPerdew-StaroverovScuseriaCorrelation	XC_MGGA_C_T	Perdew, Tao, Staroverov & Scuseria correlation JP Perdew, J Tao, VN Staroverov and GE Scuseria, Phys. Rev. Lett. 91, 146401 (2003) JP Perdew, J Tao, VN Staroverov and GE Scuseria, J. Chem. Phys. 120, 6898 (2004)

ExchangeCorrelationPotential

class ExchangeCorrelationPotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the exchange-correlation potential for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the exchange-correlation potential should be calculated
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy Œ length^{-1}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`primitiveVectors`()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

`scale`(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`shape`()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

`spin()`

Returns
The spin the exchange-correlation potential is calculated for, always `Spin.All`.

Return type
`Spin.All`

`spinProjection(spin=None)`

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

`toArray()`

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

`uniqueString()`

Return a unique string representing the state of the object.

`unit()`

Returns
The unit of the data in the grid.

Return type
A physical unit.

`unitCell()`

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

`volumeElement()`

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the exchange-correlation potential and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 6.13508 ,  5.790587,  2.75    ],
                           [ 6.13508 ,  6.73176 ,  2.336663],
                           [ 6.95016 ,  5.32    ,  2.336663],
                           [ 5.32    ,  5.32    ,  2.336663]]*Angstrom
    )


# Define the calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the effective potential
xc_potential = ExchangeCorrelationPotential(molecule_configuration)
nlsave('results.nc', molecule_configuration)
nlsave('results.nc', xc_potential)
```

nh3_xc_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the exchange-correlation potential of a DFT calculation. See *Exchange-correlation potential* for more information.

ExternalMachineLearnedFFCalculator

class ExternalMachineLearnedFFCalculator(*ase_calculator_cls=None*, **ase_calculator_args*, ***ase_calculator_kwargs*)

Class for representing an ASE calculator.

Parameters

•**ase_calculator_cls** (`ase.calculators.Calculator | function() | None`) – An ASE calculator class constructor or a function that returns an ASE calculator which should be called. **Default:** None.

•**ase_calculator_args** (*tuple*) – The arguments that should be passed to the ASE calculator class constructor.

•**ase_calculator_kwargs** (*dict*) – The keyword arguments that should be passed to the ASE calculator class constructor.

dynamicalMatrixParameters()

This method is deprecated.

Returns
None always.

Return type
None

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Method to upgrade the current calculator based on the given configuration.

Parameters
configuration (AtomicConfiguration sub-class) – The configuration the calculator will be used for.

Usage Examples

The *ExternalMachineLearnedFFCalculator* allows you to use an external machine-learned force field based on an ASE calculator in QuantumATK. The ASE calculator class constructor is passed as the first argument, and any additional arguments for the ASE calculator can be passed afterward.

Below is an example using the MatterSim pre-trained potential (requires MatterSim to be installed in a virtual environment).:

```
# Import external force field calculator
from mattersim.forcefield import MatterSimCalculator

# Create a ExternalMachineLearnedFFCalculator with the MatterSimCalculator
calculator = ExternalMachineLearnedFFCalculator(
   MatterSimCalculator,
   compute_stress=True,
   device='cuda'
)
```

Alternatively, *ExternalMachineLearnedFFCalculator* can also be used as a wrapper.:

```
# Wrap the MatterSimCalculator with the ExternalMachineLearnedFFCalculator
WrappedMatterSimCalculator = ExternalMachineLearnedFFCalculator(MatterSimCalculator)

# Initialize the wrapped calculator with additional arguments
calculator = WrappedMatterSimCalculator(compute_stress=True, device='cuda')
```

Notes

External third-party ASE calculators can be used with the *ExternalMachineLearnedFFCalculator*. The external calculator must be installed in a virtual environment. It can then be imported and used by the *ExternalMachineLearnedFFCalculator*. Any additional arguments required by the external calculator can be passed to the wrapped calculator. The *ExternalMachineLearnedFFCalculator* will pass these arguments to the ASE calculator during initialization. The wrapped calculator can be used in QuantumATK as a regular calculator.

ExternalPotential

class ExternalPotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the external potential for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the external potential should be calculated.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

`derivatives`(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy Œ length^{-1}

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the external potential is calculated for, always Spin.All.

Return type
Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the external potential and save it to a file:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 6.13508 ,  5.790587,  2.75    ],
                           [ 6.13508 ,  6.73176 ,  2.336663],
                           [ 6.95016 ,  5.32    ,  2.336663],
                           [ 5.32    ,  5.32    ,  2.336663]]*Angstrom
    )
```

(continues on next page)

(continued from previous page)

```
# Add metallic regions (without them the external potential would be constant)
metallic_region_0 = BoxRegion(
    0*Volt,
    xmin = 0*Angstrom, xmax = 12*Angstrom,
    ymin = 0*Angstrom, ymax = 12*Angstrom,
    zmin = 0*Angstrom, zmax = 1*Angstrom
)

metallic_region_1 = BoxRegion(
    1*Volt,
    xmin = 0*Angstrom, xmax = 12*Angstrom,
    ymin = 0*Angstrom, ymax = 12*Angstrom,
    zmin = 5*Angstrom, zmax = 6*Angstrom
)

metallic_regions = [metallic_region_0, metallic_region_1]
molecule_configuration.setMetallicRegions(metallic_regions)

# Define the calculator
calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the effective potential
external_potential = ExternalPotential(molecule_configuration)
nlsave('results.nc', molecule_configuration)
nlsave('results.nc', external_potential)
```

nh3_external_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

- This class inherits from the *GridValues* class.
- Returns the electrostatic potential due to the electrodes and gates in the system. Note that other external electrostatic potentials, e.g. from pseudopotentials, are **not** included.
- See *External potential* (for *DFT: LCAO* calculators) or *Tight-binding total energy* (for *Semi Empirical* calculators) for more information on the theoretical background.

ExtrapolationScheme

class ExtrapolationScheme(*include_first_order_term=None*, *include_second_order_term=None*, *include_third_order_term=None*, *minimum_number_of_atoms=None*)

A container for parameters defining the extrapolation scheme to infinite supercell size to use.

The extrapolation is obtained by fitting the calculated formation energies against the inverse length 1/L of each supercell, where L is the cube root of the supercell volume. A polynomial of up to third order is fitted. The first, second and third order terms can each be included or not in the fit (if not, they are set to zero). In all cases the fitted zeroth order term gives the extrapolation to infinite supercell size.

Parameters

- **include_first_order_term** (*bool*) – Whether to include the 1/L term (monopole-monopole) in the extrapolation fit. **Default:** True
- **include_second_order_term** (*bool*) – Whether to include the 1/L^2 term in the extrapolation fit. **Default:** False
- **include_third_order_term** (*bool*) – Whether to include the 1/L^3 term (monopole-quadrupole) in the extrapolation fit. **Default:** True
- **minimum_number_of_atoms** (*int*) – The minimum number of atoms for the supercell to be included in the fit. Any supercell with fewer atoms will be discarded. **Default:** 8

minimumNumberOfAtoms()

Returns
The minimum number of atoms for the supercell to be included in the fit. Any supercell with fewer atoms will be discarded.

Return type
int

termsToInclude()

Returns
Whether to include the first, second and third order terms in the extrapolation fit.

Return type
tuple (size 3) of bool

ExtrapolationSelectionParameters

class ExtrapolationSelectionParameters(*extrapolation_grade_algorithm=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.MaxvolStandard'>*, *selection_basis_size=None*, *descriptor_cutoff=0.2*, *descriptor_basis_size=100*, *committee_size=6*, *write_atomic_error_estimates=False*)

Additional parameters for candidate selection and extrapolation grade calculations.

Parameters

- **extrapolation_grade_algorithm** (*MaxvolStandard | MaxvolLinearized | MaxForce | QueryByCommitteeForces | QueryByCommitteeEnergy*) – Select which extrapolation grade algorithm should be used. The MaxForce algorithm is only available with MD active learning. For multi-element systems QueryByCommitteeForces is the recommended method. **Default:** MaxvolStandard
- **selection_basis_size** (*int*) – The basis size used for candidate selection and extrapolation grade calculation. Only used if MaxvolStandard is used for the extrapolation grade. **Default:** Same size as MTP basis size specified in fitting parameters.
- **descriptor_cutoff** (*float*) – The cutoff value for the distances in normalized descriptor space, above which two configurations are considered different. Only used by query-by-committee and max. force algorithms. **Default:** 0.2
- **descriptor_basis_size** (*int*) – The MTP basis size that should be used for the configurational descriptor. Only used by query-by-committee and max. force algorithms. **Default:** 100
- **committee_size** (*int*) – The number of committee members that should be used when the query-by-committee method is selected. Only used by query-by-committee algorithms. **Default:** 6

- **write_atomic_error_estimates** (*bool*) – Write the per-atom error estimates as measurements to the trajectory whenever the extrapolation grade is calculated. This is only available with query-by-committee methods. Only used by query-by-committee algorithms. **Default:** False

committeeSize()

Returns
The number of committee members when the query-by-committee method is used for the extrapolation grade.

Return type
int

descriptorBasisSize()

Returns
The MTP basis size that is used in the descriptor.

Return type
int

descriptorCutoff()

Returns
The descriptor cutoff value.

Return type
float

extrapolationGradeAlgorithm()

Returns
Which extrapolation grade algorithm should be used.

Return type
MaxvolStandard | MaxvolLinearized | MaxForce | QueryByCommitteeForces | QueryByCommitteeEnergy

selectionBasisSize()

Returns
The basis size used for candidate selection and extrapolation grade calculation.

Return type
int | None

uniqueString()

Return a unique string representing the state of the object.

writeAtomicErrorEstimates()

Returns
True if the per-atom error estimates should be written to the trajectory.

Return type
bool

FENEBondPotential

class FENEBondPotential(*particleType1*, *particleType2*, *k*, *r_0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **k** (*PhysicalQuantity of type energy*) – Potential parameter.
- **r_0** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The FENE (finite extensible non-linear elastic) potential[1] is a bonded potential where the potential energy becomes infinitely large if the distance between two bonded particles is larger than a given distance. This potential is part of the well-known Kremer-Grest model to simulate coarse-grained bead-spring polymers. It is defined as follows:

$$U_{ij} = \begin{cases} k_{ij} \ln(1 - (r_{ij}/r_0)^2), & r_{ij} < r_0 \\ \infty, & r_{ij} \geq r_0. \end{cases}$$

In the actual implementation, we have replaced ∞ by $k_{ij} \ln(DBL\_MIN) = k_{ij} \ln(2.2 \cdot 10^{-308}) = -708 k_{ij}$. This has no effect on the actual dynamics but will not destroy all measurements as would be the case when the potential energy becomes infinity.

[1] Kurt Kremer and GaryăS. Grest. Dynamics of entangled linear polymer melts: A moleculardynamics simulation. *J. Chem. Phys.*, 92(8):5057–5086, 1990. URL: https://doi.org/10.1063/1.458541, doi:10.1063/1.458541.

FIRE

class FIRE(*maximum_timestep=PhysicalQuantity(2.0, fs)*, *scf_initial_guess=None*)

Constructor for the FIRE optimizer method.

Parameters

- **maximum_timestep** (*PhysicalQuantity* of type time) – The maximum timestep. **Default:** 2.0*femtosecond
- **scf_initial_guess** (`UsePreviousConfiguration` | `DIISExtrapolation`) – Determines how the state of the system at each MD step is used as an initial guess for the next MD step, in the case when a self-consistent calculation is performed at each step. `UsePreviousConfiguration` means that the density matrix (or valence density/wave functions in a plane-wave calculation) of the previous configuration will be used as an initial guess for the next SCF calculation. For LCAO and semi-empirical calculations, it is possible and recommended to use `DIISExtrapolation`, which calculates an improved initial guess using the DIIS extrapolation method for the previous density matrices.

  This parameter has no effect when the calculator attached to `configuration` is a force-field calculator. **Default:** `UsePreviousConfiguration` for *PlaneWaveCalculator*, `DIISExtrapolation(4)` for *LCAOCalculator* and *SemiEmpiricalCalculator*.

scfInitialGuess()

Returns
The SCF initial guess used for the optimizer.

Return type
`DIISExtrapolation` | `UsePreviousConfiguration` | None

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Optimize the geometry of a water molecule using the FIRE algorithm.

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,  -1.70000000e-05,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Define a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# Perform optimization using the FIRE algorithm.
OptimizeGeometry(molecule_configuration, optimizer_method=FIRE())
```

fire.py

Notes

The *Fast Inertial Relaxation Engine (FIRE)* is an optimization algorithm which is based on molecular dynamics with additional modifications on the velocity and an adaptive timestep. Details about this algorithm can be found in[1].

FastFourier2DSolver

class FastFourier2DSolver(*boundary_conditions=None*, *order=None*)

The Hybrid FFT2D + Finite-Difference-1D Poisson solver.

Parameters

•**boundary_conditions** (list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*) – A list of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:**

–*MoleculeConfiguration*: `[[PeriodicBoundaryCondition()] * 2] * 3`

–*BulkConfiguration*: `[[PeriodicBoundaryCondition()] * 2] * 3`

–*SurfaceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition(), NeumannBoundaryCondition()]]
```

–*DeviceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition()] * 2]
```

•**order** (*int*) – The order of the finite difference kernel used for solving in the C-direction. Must be either 1 or 2. **Default:** 2

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

nlinfo()

Returns
The nlinfo.

Return type
dict

[1] E.ăBitzek, P.ăKoskinen, F.ăGahler, M.ăMoseler, and P.ăGumbsch. Structural Relaxation Made Simple. *Phys. Rev. Lett.*, 97:170201, Oct 2006. doi:10.1103/PhysRevLett.97.170201.

order()

Returns
The order.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a *FastFourier2DSolver* with periodic boundary conditions:

```
poisson_solver = FastFourier2DSolver(
    boundary_conditions=[
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Define a *FastFourier2DSolver* with Dirichlet boundary conditions in the C-direction:

```
poisson_solver = FastFourier2DSolver(
    boundary_conditions=[
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Notes

The *FastFourier2DSolver* uses a FFT in the A, B directions and a finite difference operator in the C direction to solve Poissons equation[1].

The *FastFourier2DSolver* is the default solver for a *DeviceConfiguration* that has no metallic and dielectric regions.

[1] T.ăOzaki, K.ăNishio, and H.ăKino. Efficient implementation of the nonequilibrium green function method for electronic transport calculations. *Phys. Rev. B*, 81:035116, 2010. doi:10.1103/PhysRevB.81.035116.

Reference

FastFourierSolver

class FastFourierSolver(*boundary_conditions=None*)

The FFT-based Poisson solver.

Parameters
boundary_conditions (list of *PeriodicBoundaryCondition*) – A list of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:** `[[PeriodicBoundaryCondition()] * 2] * 3`

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a *FastFourierSolver* with periodic boundary conditions on the 6 faces:

```
poisson_solver = FastFourierSolver()

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

FatBandstructure

class FatBandstructure(*configuration*, *route=None*, *points_per_segment=None*, *kpoints=None*, *bands_above_fermi_level=None*, *projections=None*, *diagonalization_method=None*)

Analysis class for calculating the fat band structure (or projected band structure) for a bulk configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the fat band structure.
- **route** (*list of str*) – The route to take through the Brillouin-zone as a list of symmetry points of the unit cell, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `kpoints`. **Default:** Unit-cell dependent route.

•**points_per_segment** (*int*) – The number of points per segment of the route given as a positive integer. This option is mutually exclusive to kpoints. **Default:** 20.

•**kpoints** (*list of lists of floats*) – A list of 3-dimensional fractional k-points at which to calculate the energies of the bands e.g. [[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]. The shape is (K, 3) where K is the number of k-points. This option is mutually exclusive to route, and points_per_segment. **Default:** Unit-cell dependent route.

•**bands_above_fermi_level** – **Deprecated:** from v2023.09, see diagonalization_method and *FullDiagonalizationSolver* instead.

•**projections** (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for calculating the weights. **Default:** ProjectOnUpDownSpin.

•**diagonalization_method** – Method used for diagonalizing the hamiltonian.

This parameter allows to choose between a full diagonalization solver and an iterative subspace solver. The full diagonalization solver evaluates all bands from the lowest energy one to a given number of bands above fermi level. The iterative subspace solver allows to evaluate a given number of bands around fermi level, or around an energy of choice.

The full diagonalization solver is more robust, but can be proibitively expensive for very large systems.

The iterative solver can deal with very large systems (tens of thousands atoms and beyond) and greatly outperforms when calculating a small number of eigenvalues, but it is also inherently less robust.

Note: the exact method used when selecting *FullDiagonalizationSolver* is defined by the calculator (see *AlgorithmParameters*). *IterativeDiagonalizationSolver* is not supported for *PlaneWaveCalculator*

Default: FullDiagonalizationSolver

Type
FullDiagonalizationSolver | *IterativeDiagonalizationSolver*

conductionBandEdge()

Calculate the conduction band edge.

Returns
The conduction band edge.

Return type
PhysicalQuantity of type energy

directBandGap()

Calculate the direct band gap.

Returns
The direct band gap.

Return type
PhysicalQuantity of type energy

energyZero()

The energy zero is equal to the Fermi level. For fixed spin moment calculations it is the average of the Fermi level for spin up and spin down.

Returns
The energy zero.

Return type
PhysicalQuantity of type energy

evaluate()

Get the eigenvalues.

Returns
The eigenvalues. The shape is (K, B), where K is the number of k-points and B the number of bands. B also includes the spin channel for polarized AND unpolarized calculations. Spin up and spin down solutions are interleaved, i.e. up and down bands alternate. In contrast to the behavior of the *Bandstructure* class, an unpolarized calculation is treated as a polarized one with identical up and down components. This is done, since different projections could be chosen for the up and down states even for unpolarized calculations (see *Projection*).

Return type
list of PhysicalQuantity of type energy

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.All

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

fermiTemperature()

Returns
The Fermi temperature used in this bandstructure.

Return type
PhysicalQuantity of type temperature

indirectBandGap()

Calculate the indirect band gap.

Returns
The indirect band gap.

Return type
PhysicalQuantity of type energy

kpoints()

Returns
The list of 3-dimensional fractional k-points at which the energies of the bands are calculated. The shape is (K, 3) where K is the number of k-points.

Return type
list of lists of floats

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

projections()

Returns
The projections.

Return type
list of *Projection*

route()

Returns
The route through the Brillouin-zone as a list of symmetry points of the unit cell.

Return type
list of str

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

valenceBandEdge()

Calculate the valence band edge.

Returns
The valence band edge.

Return type
PhysicalQuantity of type energy

weights()

Returns
The weights for a given projection. Each projection provides a weight for all the eigenvalues. The shape is (*P*, *K*, *B*), where *P* is the number of projections, *K* is the number of k-points, and *B* is the number of spin-interleaved bands as in the output of the `evaluate` method.

Return type
`numpy.ndarray` of doubles

Usage Example

Calculate the shell resolved fat band structure for bulk silicon.

```
lattice = FaceCenteredCubic(5.4306*Angstrom)
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=[Silicon, Silicon],
    fractional_coordinates=[(0.0, 0.0, 0.0), (0.25, 0.25, 0.25)])

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(7, 7, 7)

calculator = LCAOCalculator(
        numerical_accuracy_parameters=NumericalAccuracyParameters(
            k_point_sampling=k_point_sampling))

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# FatBandstructure
# ------------------------------------------------------------
fat_bandstructure = FatBandstructure(
    configuration=bulk_configuration,
    projections=ProjectOnShellsByElement
    )
```

In the example above we used an abbreviation to generate the list of projections (see *Projection* documentation). Alternatively, if we would obtain the same result by specifying:

```
# ------------------------------------------------------------
# FatBandstructure
# ------------------------------------------------------------
projections = [Projection(l_quantum_numbers=[0]),
               Projection(l_quantum_numbers=[1]),
               Projection(l_quantum_numbers=[2])]

fat_bandstructure = FatBandstructure(
        configuration=bulk_configuration,
        projections=projections)
```

If we want to build more complex projections we can use the algebraic operations, as documented in *Projection*. In the following example we calculate the fat band structure for bulk gallium arsenide, projecting on the s shells of gallium and p shells of arsenic

```
lattice = FaceCenteredCubic(5.6537*Angstrom)
elements = [Gallium, Arsenic]
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]
bulk_configuration = BulkConfiguration(
```

(continues on next page)

(continued from previous page)

```
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# -----------------------------------------------------------
# Calculator
# -----------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(7, 7, 7)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# -----------------------------------------------------------
# FatBandstructure
# -----------------------------------------------------------
p1 = Projection(l_quantum_numbers=[0], atoms=[Gallium])
p2 = Projection(l_quantum_numbers=[1], atoms=[Arsenic])

fat_bandstructure = FatBandstructure(
    configuration=bulk_configuration,
    projections=p1+p2
    )
```

Notes

The *FatBandstructure* (or Projected band structure) is used to visualize the contribution of different orbitals to the bandstructure of a bulk system. Given a projection on a subspace, for each eigenvalue we can calculate a weight associated to the projection. The weights are calculated as follows: the eigenstates and eigenvalues are calculated by solving the Schrödinger equation, similarly to an ordinary bandstructure calculation

$$H\psi_{n\mathbf{k}} = E\psi_{n\mathbf{k}}$$

We then define an operator $\hat{\mathbf{P}}_M$ which performs a projection onto the subspace M. The weight $\omega^M_{n\mathbf{k}}$ associated to this projection for the eigenpair $(E_{n\mathbf{k}}, \psi_{n\mathbf{k}})$ is given by

$$\omega^M_{n\mathbf{k}} = \langle\psi_{n\mathbf{k}}|\hat{\mathbf{P}}_M|\psi_{n\mathbf{k}}\rangle$$

It follows from the definition and form the normalization condition on $\psi_{n\mathbf{k}}$ that the weight is 1.0 for any eigenpair if the projection operator is the identity. The projection operator is defined according to the description contained in the documentation for *Projection*.

Note: The *FatBandstructure* will return weights between 0 and 1 (unitless) for projections on up/down spin, and

between -1 to 1 (in units of $\frac{\hbar}{2}$) for projections involving the Pauli spin matrices. See the *Usage Example* in *Projection* for some more information.

FeastSolver

class FeastSolver(*number_of_eigenvalues=None*, *lambda_min=None*, *lambda_max=None*)

Calculate the density matrix using the feast algorithm

Parameters

- **number_of_eigenvalues** (*int > 0*) – The estimated number of eigenvalues. **Default:** 1.5 × number of electrons.
- **lambda_min** (*PhysicalQuantity* of type energy) – The lower bound on the eigenvalue spectrum to calculate. **Default:** `-20.0 * Hartree`
- **lambda_max** (*PhysicalQuantity* of type energy) – The upper bound on the eigenvalue spectrum to calculate. **Default:** `0.0 * Hartree`

lambdaMax()

Returns
Upper bound on the eigenvalue spectrum.

Return type
PhysicalQuantity of type energy

lambdaMin()

Returns
Lower bound on the eigenvalue spectrum.

Return type
PhysicalQuantity of type energy

numberOfEigenvalues()

The estimated number of eigenvalues.

Returns
None when the estimated number of eigenvalues is set to 1.5 × number of electrons.

Return type
int | None

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Define a density matrix method that uses the Feast Solver

```
density_matrix_method = FeastSolver(
    lambda_min=-1.0*Hartree,
    lambda_max= 1.0*Hartree,
    number_of_eigenvalues=10,
    )
```

Notes

The FeastSolver is useful if only a small part of the eigenvalue spectrum (eigenvector space) is needed. The parameters lambda_min and lambda_max specify the limits of the sought eigenvalue spectrum and number_of_eigenvalues is an estimate for the number of eigenvalues to be found within this interval. The algorithm works most efficiently if the estimate is approximately a factor 1.5 larger than the actual number of eigenvalues within the interval. For more details on the Feast algorithm, visit the Feast website.

FermiDirac

class FermiDirac(*broadening*)

Parameters
broadening (*PhysicalQuantity* of type energy or temperature) – The broadening of the Fermi-Dirac distribution.

broadening()

Returns
The broadening of the Fermi-Dirac distribution.

Return type
PhysicalQuantity of type energy

Usage Examples

Use the Fermi-Dirac occupation function with a broadening of 0.1 eV on an LCAOCalculator:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    occupation_method=FermiDirac(0.1*eV))

calculator = LCAOCalculator(numerical_accuracy_parameters=numerical_accuracy_parameters)
```

Notes

Note: For comparison of different occupation methods and suggestions for which one to choose, see *Occupation Methods*.

In the Fermi-Dirac smearing scheme one effectively considers the system at finite temperature. This means that one replaces the integer occupation numbers when calculating e.g. the electron density in DFT by fractional occupation numbers given by the Fermi-Dirac distribution,

$$f(x) = \frac{1}{e^{(\epsilon-\mu)/\sigma}+1},$$

where ϵ is the energy of a given state, μ is the chemical potential/Fermi level and σ is a broadening parameter. For finite temperature calculations $\sigma = k_{\mathrm{B}}T$ with T the temperature.

The Fermi-Dirac smearing scheme also corresponds to replacing the Dirac delta-function in the density of states by a smeared function given by

$$\tilde{\delta}(x) = \frac{1}{4\sigma}\frac{1}{\cosh^2(x/2\sigma)}$$

In the Fermi-Dirac smearing scheme the contribution to the generalized entropy from a state with occupation f is[1]

$$S(f) = -[f \ln f + (1-f)\ln(1-f)].$$

The total energy can extrapolated to zero broadening by adding to the total internal energy the following correction term

$$\Delta E_{\sigma\to 0}(\sigma) = -\frac{1}{2}\sum_i \sigma S(f_i),$$

where i runs over all states.

FermiSurface

class FermiSurface(*configuration*, *sampling=None*, *band_indices=None*)

The FermiSurface is used for calculating a surface associated with an energy grid in the reciprocal k-space. For a given *BulkConfiguration*, its unit cell is used to calculate the Brillouin zone in k-space. The bounding box of the Brillouin zone is then used to define the k-point sampling space, which is given as input to a Bandstructure calculation. The resulting bands make up the energy grid an energy isosurface can be obtained. For the band corresponding to the Fermi energy, this isosurface corresponds to the so-called Fermi surface.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration (with an attached calculator) the band surface should be calculated for.
- **sampling** (*int*) – The sampling, n, used for the 3D grid from which the Fermi surface is extracted. n sampling points are used to discretize the longest grid axis. The length of the resulting sampling intervals defines the discretization of the two remaining axis.
- **band_indices** (*list of int*) – The indices of the bandstructure band from which the surface is sampled.

bandIndices()

Returns
The band indices, .i.e. the bandstrucure indices for which surfaces can be calculated.

Return type
numpy.ndarray

bands()

Returns
The bandstrucure bands associated with the given band indices.

Return type
numpy.ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

[1] N.ăD. Mermin. Thermal Properties of the Inhomogeneous Electron Gas. *Phys. Rev.*, 137:A1441–A1443, Mar 1965. doi:10.1103/PhysRev.137.A1441.

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

sampling()

Returns
The sampling value of the band surface.

Return type
int

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

FiniteBiasSpinTransferTorque

class FiniteBiasSpinTransferTorque(*configuration*, *filename*, *object_id*, *atom_indices=None*, *bias_voltages=None*, *kpoints=None*, *initial_spin=None*, *potential_drop_region_tags=None*, *potential_drop_method=None*, *potential_profile=None*, *number_of_processes_per_task=None*, *log_filename_prefix=None*, *initial_state=None*)

configuration:
The device calculation that the finite bias spin transfer torque should be calculated for.

configuration:
DeviceConfiguration

Parameters

- **filename** (*str*) – The full or relative filename path the study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file.
- **atom_indices** (*list of int | list of str*) – The indices of the atomic sites to calculate the spin transfer torque for. This may be defined as a list of integers or a list of tags. **Default:** All atoms in the configuration
- **bias_voltages** (*PhysicalQuantity* of type voltage) – A list of voltages. Each voltage will in turn represent the value 1.0 in the potential profile when converting the latter to the voltage drop profile accross the device. I.e. if the potential profile drops by 1.0 from the left to the right lead the voltage drop from the left to the right lead will equal each given voltage in turn. **Default:** `numpy.linspace(-0.5, 0.5, 11)*Volt`
- **kpoints** (*MonkhorstPackGrid* | *KpointDensity*) – The k-points for which the spin transfer torque should be calculated. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **initial_spin** (*InitialSpin*) – The initial spins of the device for the non-collinear calculation. **Default:** Initial spin where all spins are pointing in the x-direction (theta=90, phi=0) except for the atoms in atom_indices, which will be pointing in the z-direction (theta=0, phi=0)

- **potential_drop_region_tags** (*list of str*) – List of tags defining regions where the potential drops. A potential profile will be generated automatically based on the tagged regions. It is a requirement that the tagged regions do not overlap in the z-direction. **Default:** None
- **potential_drop_method** (*str*) – Parameter describing how the potential drops as function of length. This should either be linear or exponential **Default:** exponential
- **potential_profile** (List of (*PhysicalQuantity* of type length, float)) – Defines the potential model as a piecewise linear function, defined as a list of (coordinate, value) pairs, where the values define the potential drop in an interval $(+V_b/2, V_b/2)$ with V_b the applied bias.

  For example, a list *[(0*Ang, 1), (5*Ang, 1), (10*Ang, 0.), (15*Ang, 0.), (20*Ang, -1.), (25*Ang, -1.)]* represents a potential ramp from $V_b/2$ to $-V_b/2$ between coordinates 5 and 20 Angstrom, with a flat 0 potential region in the middle. It is required that the leftmost and rightmost values are 1 and -1 respectively.

  If `potential_profile` this will be used for the non-self consistent potential drop. **Default:** [(0*Ang, 1), (L_C, -1)], where L_C is the central region length.
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of processes does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively. **Default:** `None`
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of study object. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'srhrecombination_'`

atomIndices()

Returns
The indices of the atomic sites to calculate the spin transfer torque for.

Return type
list of int

atomResolvedSpinTransferTorkance(*atom_indices=None*)

Returns
Linear response spin transfer torkance array of shape (3, n_atom_indices), where the 3 corresponds to x, y, z directions.

Return type
PhysicalQuantity of type energy per voltage (charge).

atomResolvedSpinTransferTorques(*atom_indices=None*, *bias_voltages=None*)

Parameters

- **atom_indices** (list of int | `All`) – List of atom indices to sum the STT contribution over. If `All` or `None`, the indices used at construction will be used.
- **bias_voltages** (*PhysicalQuantity* of type voltage) – A list of voltages for which to get the result. **Default:** All the bias voltages used at construction.

Returns
Spin transfer torque array of shape (n_bias_points, 3, n_atom_indices), where the 3 corresponds to x, y, z directions.

Return type
PhysicalQuantity of type energy

biasVoltages()

Returns
The bias voltages.

Return type
PhysicalQuantity of type Volt

`calculator`()

Returns
The calculator attached to the configuration.

Return type
DeviceLCAOCalculator

`dependentStudies`()

Returns
The list of dependent studies.

Return type
list of `Study`

`filename`()

Returns
The filename where the study object is stored.

Return type
str

`initialSpin`()

Returns
The initial spin to be used for the non-collinear calculation.

Return type
InitialSpin

`kpointResolvedSpinTransferTorkance`(*atom_indices=None*)

Parameters
atom_indices (list of int | `All`) – List of atom indices to sum the STT contribution over. If `All` or `None`, the indices used at construction will be used.

Returns
Spin transfer torkance array of shape (3, n_kpoints), where the 3 corresponds to x, y, z directions. The torkance is summed over atom indices.

Return type
PhysicalQuantity of type energy per voltage (charge).

`kpointResolvedSpinTransferTorques`(*bias*, *atom_indices=None*)

Parameters

- **bias** (*PhysicalQuantity* of type Volt) – The bias voltage to get result from.
- **atom_indices** (list of int | `All`) – List of atom indices to sum the STT contribution over. If `All` or `None`, the indices used at construction will be used.

Returns
Spin transfer torque array of shape (3, n_kpoints), where the 3 corresponds to x, y, z directions. The STT is summed over atom indices.

Return type
PhysicalQuantity of type energy

kpoints()

Returns
The k-points for which the spin transfer torque is calculated.

Return type
MonkhorstPackGrid

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

potentialProfile()

Returns
Defines the potential model as a piecewise linear function, defined as a list of (coordinate, value) pairs.

Return type
List of (*PhysicalQuantity* of type length, float)

potentialProfileCoordinates()

Returns
The coordinates in the potential profile model.

Return type
list of *PhysicalQuantity* of type length

`potentialProfileValues()`

Returns
The scaling values in the potential profile model.

Return type
list of float

`saveToFileAfterUpdate()`

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

`totalSpinTransferTorkance()`

Returns
Spin transfer torque array of shape (n_bias_points, 3), where the 3 corresponds to x, y, z directions.

Return type
PhysicalQuantity of type energy per voltage (charge).

`totalSpinTransferTorques()`

Returns
Spin transfer torque array of shape (n_bias_points, 3), where the 3 corresponds to x, y, z directions.

Return type
PhysicalQuantity of type energy

`uniqueString()`

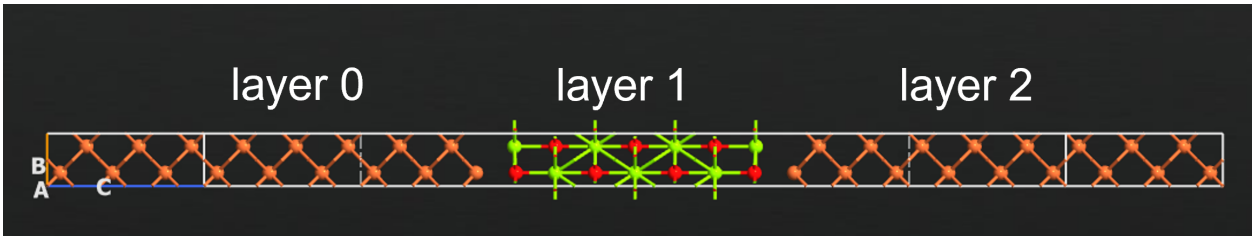
Return a unique string representing the state of the object.

`update()`

Run the calculations for the study object.

Usage Examples

In this example we will calculate the spin transfer torque (STT) in a Fe-MgO-Fe magnetic tunneling junction (MTJ) as shown below. The left Fe, MgO, and right Fe regions have been assigned the tags layer 0, layer 1, and layer 2 respectively. We will be calculating the STT in free layer, which we will define as layer 2, for an angle of 90 degree between the magnetization in the left (layer 0) and free (layer 2) magnetic regions.

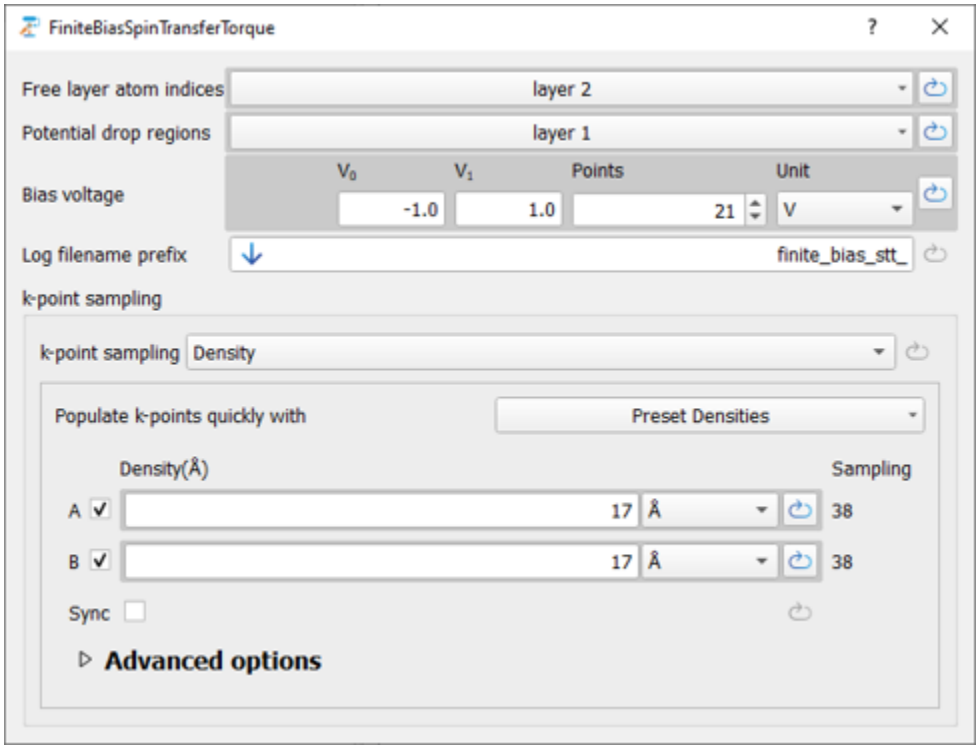


The spin transfer torque is evaluated in a non-selfconsistent finite bias approximation. The potential drop across the MgO barrier is represented as a linear ramp across the tunneling barrier[1], avoiding time consuming and often hard to

[1] Ivan Rungger, Andrea Droghetti, and Maria Stamenova. *Non-equilibrium Greens Function Methods for Spin Transport and Dynamics*, pages 1–27. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-42913-7_75-1, doi:10.1007/978-3-319-42913-7_75-1.

converge non-collinear calculations at finite bias. The specification of the potential drop region is most easily done using tags.

The figure below shows how to setup the calculation using the work flow builder and the FiniteBiasSpinTransferTorque editor.



The free layer atoms, where the STT will be evaluated can be specified in the editor using tags, setting it to layer 2 in this example. The potential drop region is likewise specified using tags, in this case layer 1.

A workflow for the calculation can be downloaded `here` with a corresponding python script `Fe-MgO-Fe-STT-Medium.py`. Note that the DeviceLCAOCalculator needs to have spin set to Noncollinear.

Note: The calculation will take several hours, even when running on multiple cores.

Initial spin

When setup like above, the STT will be calculated for a spin configuration defined as

- The spin in the free layer (layer 2) is oriented along the z-axis
- The spin in the left Fe layer (layer 0) is rotated with $\theta = 90^\circ$ and $\phi = 0^\circ$ pointing in the x-direction.
- The spin in the potential drop region is scaled to zero.

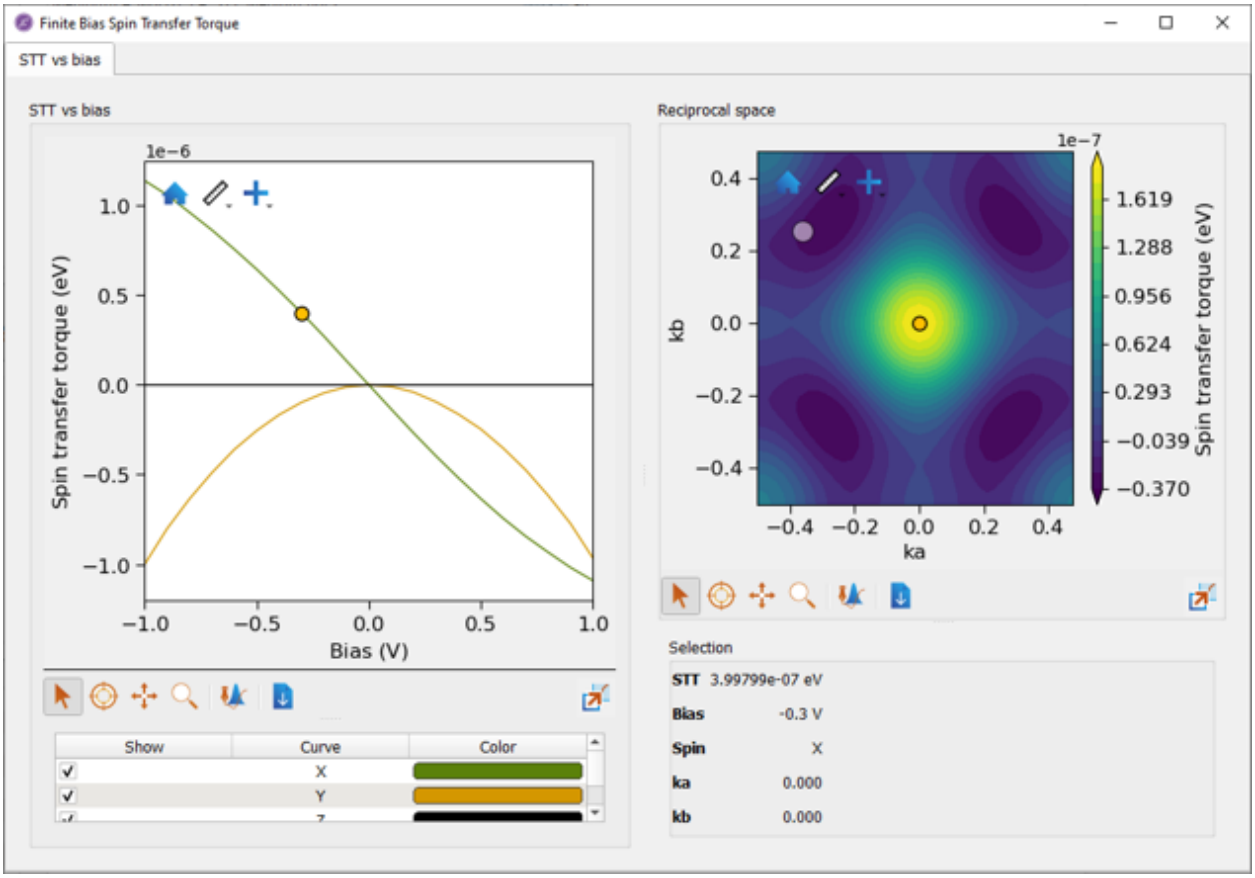
This means that the in-plane component of the STT vector will be in the x-direction, while the out-of-plane component will be in the y-direction.

It is possible to define a custom spin orientation, by inserting an **InitialSpin** block before the **FiniteBiasSpinTransferTorque** work-flow block.

Analyze results

When the calculation has finished the results can be visualized with the **Finite bias spin transfer torque analyzer** as illustrated below.

The total STT on the free layer atoms



The close to linear behavior of the in-plane (x-) component for small bias and the quadratic behavior of the out-of-plane (y-) component are consistent with literature results[2, Page 2902, 1].

Note: Results can be sensitive to numerical parameters such as k-point sampling or real axis contour accuracy. Convergence with respect to those parameters should be carefully checked.

[2] Xingtao Jia, KeăXia, Youqi Ke, and Hong Guo. Nonlinear bias dependence of spin-transfer torque from atomic first principles. *Phys. Rev. B*, 84:014401, Jul 2011. URL: https://link.aps.org/doi/10.1103/PhysRevB.84.014401, doi:10.1103/PhysRevB.84.014401.

Notes

The atom-resolved torque on an atom a is then calculated as[3] :

$$\mathbf{T}_{CD,a} = \sum_{\alpha \in a} \sum_{\beta} \Re[\rho_{CD,\alpha\beta}\, \sigma \times \mathbf{B}_{XC,\alpha\beta}]$$

where $\rho_{CD} = \rho_{neq} - \rho_{eq}$ is the current driven density matrix, given by the difference between the density matrix at finite bias and at equilibrium. The α sum runs over orbital indices on atom a while the β sum runs over all orbitals in the system.

Note that the non-equilibrium contour parameters used to evaluate the density matrix (see *ContourParameters*) are taken from the calculator attached to the input configuration.

References

FixAtomConstraints

class FixAtomConstraints(*atom_selection*, *fix_cartesian_direction=None*)

This constraint fixes the positions of the selected atoms during a *MolecularDynamics()* or *OptimizeGeometry()* simulation. The atoms are selected by specifying either the indices of the atoms or by passing one or more tags.

Parameters

- **atom_selection** (*list of ints | list of str | str*) – The list of indices or the list of tags of the atoms whose positions should be fixed.
- **fix_cartesian_direction** (*list(3) of bools*) – List of three directions to constrain the group of atoms in. If the Bravais lattice does not change (or a MoleculeConfiguration is used) then the three flags correspond to constraining the x, y, and z Cartesian coordinates. If the Bravais lattice does change, then the three flags correspond to constraining the fractional coordinates along the a, b, and c axes. **Default:** [True, True, True]

atomIndices()

Returns
The indices of the fixed atoms. An exception is raised if the constraint has not been initialized with a configuration.

Return type
list of type int

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by the constraints.

Return type
int

[3] BranislavăK. Nikolic, Kapildeb Dolui, MarkoăD. Petrovic, Petr Plechac, Troels Markussen, and Kurt Stokbro. *First-Principles Quantum Transport Modeling of Spin-Transfer and Spin-Orbit Torques in Magnetic Multilayers*, pages 1–35. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-50257-1_112-1, doi:10.1007/978-3-319-50257-1_112-1.

tags()

Returns
The tags used for defining the fixed atoms.

Return type
list of type string

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run an MD simulation of a decane molecule with one end of the molecule fixed.

```
# Define elements
elements = [Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen,
            Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon,
            Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen,
            Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen,
            Carbon, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 2.949218676646,  0.371893089344, -5.977841255567],
                         [ 2.0915751053  ,  0.325878896596, -5.255719576195],
                         [ 1.450283574002,  1.214213807408, -5.492967455154],
                         [ 1.451118166918, -0.526603584715, -5.601873618684],
                         [ 1.822966280268,  0.248365819288, -4.006203357873],
                         [ 2.452472374579, -0.657496389029, -3.794106921656],
                         [ 2.454588788042,  1.119399272419, -3.683412523973],
                         [ 1.11002977129 ,  0.182932682631, -2.941890060606],
                         [ 0.514074186602,  1.11809089051 , -3.137372769128],
                         [ 0.511494198217, -0.719003710087, -3.252072509036],
                         [ 0.842624096707,  0.10516011283 , -1.690762149425],
                         [ 1.491345296758, -0.78355395752 , -1.471599662887],
                         [ 1.494967653129,  0.95670152228 , -1.362236499313],
                         [ 0.132859680402,  0.039658959846, -0.626048074315],
                         [-0.482794358104,  0.9589027956  , -0.827177016793],
                         [-0.487026068393, -0.844146311539, -0.940982573926],
                         [-0.132380369437, -0.038781999777,  0.626344899449],
                         [ 0.483113914589, -0.95814352964 ,  0.827391526337],
                         [ 0.487761818334,  0.84474606991 ,  0.941323554846],
                         [-0.842177289594, -0.104271778821,  1.690893077731],
                         [-1.490395532317,  0.784892174015,  1.472217773116],
                         [-1.494937459238, -0.955400183967,  1.362016363174],
                         [-1.109932511965, -0.182939754747,  2.942043484298],
                         [-0.515413707857, -1.119341268799,  3.136533689731],
                         [-0.509862540826,  0.71770586625 ,  3.253250779134],
                         [-1.823114817881, -0.248387001392,  4.00605213455 ],
                         [-2.451181382366,  0.658629838415,  3.794745029733],
                         [-2.456019886489, -1.118078262521,  3.682204447835],
                         [-2.092097378568, -0.326797885494,  5.255445507069],
                         [-1.454808166348, -1.218140992161,  5.492314817703],
```

(continues on next page)

(continued from previous page)

```
                          [-1.448149189819,  0.522735169126,  5.602345190585],
                          [-2.95020692258 , -0.368818356258,  5.97713874924 ]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

potentialSet = Tersoff_CH_2010()
calculator = TremoloXCalculator(parameters=potentialSet)

molecule_configuration.setCalculator(calculator)
molecule_configuration.update()

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

fix_atom_indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
fix_atom_constraints = FixAtomConstraints(fix_atom_indices)
# Or equivalently as a tag:
# molecule_configuration.addTags('frozen', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# fix_atom_constraints = FixAtomConstraints('frozen')

md_trajectory = MolecularDynamics(
    molecule_configuration,
    constraints=[fix_atom_constraints],
    trajectory_filename='decane_constraints.hdf5',
    steps=500,
    log_interval=10,
    method=method
)

molecule_configuration = md_trajectory.lastImage()
```

In the following example only the z-coordinates of the end atoms are fixed.

```
fix_atom_indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
fix_atom_constraints = FixAtomConstraints(
    fix_atom_indices,
    fix_cartesian_direction=[False, False, True]
```

(continues on next page)

(continued from previous page)

```
)

md_trajectory = MolecularDynamics(
    molecule_configuration,
    constraints=[fix_atom_constraints],
    trajectory_filename='decane_z_constraints.hdf5',
    steps=500,
    log_interval=10,
    method=method
)
```

Notes

- *FixAtomConstraints* can be used in *OptimizeGeometry* and *MolecularDynamics* simulations to constrain the coordinates of the selected atoms to their initial positions by setting the velocities and forces on these atoms to zero.
- It is not recommended to run cell-optimizations (i.e. optimization simulations with `optimize_cell=True`) or NPT simulations (e.g. *NPTBerendsen*) when fixed atoms are used.
- With the `fix_cartesian_direction` parameter you fix individual cartesian components of the selected atoms.
- You can specify more than one *FixAtomConstraints* object in the `constraints` list, e.g. one object which fixes only certain cartesian components of some atoms, and another one, which completely fixes another group of atoms.
- If specifying one or more tags instead of atom indices, the constrained atoms are specified by tags (that should be set on the configuration beforehand) rather than indices. The conversion of tag names to atom indices is deferred to the time when the constraint object is paired up with a configuration and not when initializing the object. That means the same constraint can be used for multiple configurations with different numbers of atoms (e.g. in an active learning simulation) provided the constrained tag names are present in all configurations.

FixCenterOfMass

class FixCenterOfMass(*fix_cartesian_direction=None*)

This constraint fixes the center-of-mass of the system in a *MolecularDynamics()* simulation.

Parameters
fix_cartesian_direction (*list(3) of bools*) – Flags to apply the position constraints of this group of atoms only to the selected certain cartesian directions. **Default:** [True, True, True]

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by the constraints.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run an MD simulation of a decane molecule with the center-of-mass of the molecule fixed.

```
# Define elements
elements = [Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen,
            Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon,
            Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen,
            Hydrogen, Carbon, Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen,
            Carbon, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 2.949218676646,  0.371893089344, -5.977841255567],
                         [ 2.0915751053  ,  0.325878896596, -5.255719576195],
                         [ 1.450283574002,  1.214213807408, -5.492967455154],
                         [ 1.451118166918, -0.526603584715, -5.601873618684],
                         [ 1.822966280268,  0.248365819288, -4.006203357873],
                         [ 2.452472374579, -0.657496389029, -3.794106921656],
                         [ 2.454588788042,  1.119399272419, -3.683412523973],
                         [ 1.11002977129 ,  0.182932682631, -2.941890060606],
                         [ 0.514074186602,  1.11809089051 , -3.137372769128],
                         [ 0.511494198217, -0.719003710087, -3.252072509036],
                         [ 0.842624096707,  0.10516011283 , -1.690762149425],
                         [ 1.491345296758, -0.78355395752 , -1.471599662887],
                         [ 1.494967653129,  0.95670152228 , -1.362236499313],
                         [ 0.132859680402,  0.039658959846, -0.626048074315],
                         [-0.482794358104,  0.9589027956  , -0.827177016793],
                         [-0.487026068393, -0.844146311539, -0.940982573926],
                         [-0.132380369437, -0.038781999777,  0.626344899449],
                         [ 0.483113914589, -0.95814352964 ,  0.827391526337],
                         [ 0.487761818334,  0.84474606991 ,  0.941323554846],
                         [-0.842177289594, -0.104271778821,  1.690893077731],
                         [-1.490395532317,  0.784892174015,  1.472217773116],
                         [-1.494937459238, -0.955400183967,  1.362016363174],
                         [-1.109932511965, -0.182939754747,  2.942043484298],
                         [-0.515413707857, -1.119341268799,  3.136533689731],
                         [-0.509862540826,  0.71770586625 ,  3.253250779134],
                         [-1.823114817881, -0.248387001392,  4.00605213455 ],
                         [-2.451181382366,  0.658629838415,  3.794745029733],
                         [-2.456019886489, -1.118078262521,  3.682204447835],
                         [-2.092097378568, -0.326797885494,  5.255445507069],
                         [-1.454808166348, -1.218140992161,  5.492314817703],
                         [-1.448149189819,  0.522735169126,  5.602345190585],
                         [-2.95020692258 , -0.368818356258,  5.97713874924 ]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

potentialSet = Tersoff_CH_2010()
calculator = TremoloXCalculator(parameters=potentialSet)
```

(continues on next page)

(continued from previous page)

```
molecule_configuration.setCalculator(calculator)
molecule_configuration.update()

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=False
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

# Run MD with FixCenterOfMass constraint.
md_trajectory = MolecularDynamics(
    molecule_configuration,
    constraints=[FixCenterOfMass()],
    trajectory_filename='decane_fix_center_of_mass_md.nc',
    steps=500,
    log_interval=10,
    method=method
)

molecule_configuration = md_trajectory.lastImage()
```

Notes

- *FixCenterOfMass* can be used in *MolecularDynamics* simulations to constrain the center-of-mass of the configuration.
- In principle, it is sufficient to set `remove_center_of_mass_momentum=False` in the initial velocity object (e.g. *MaxwellBoltzmannDistribution*), to achieve this. Sometimes, however, numerical inaccuracies in the force calculation (e.g when using Ewald summation techniques, as in *CoulombSPME*) can give rise to a finite force on the center-of-mass, resulting in an unwanted drift of the system. In this case it may be convenient to apply a *FixCenterOfMass* constraint, to remove this effect.
- When using the *Langevin* integrator, the stochastic force will cause a diffusion of the center-of-mass vector, which can be avoided by using a *FixCenterOfMass* constraint.
- With the `fix_cartesian_direction` parameter you can fix individual cartesian components of the center-of-mass-vector.
- *FixCenterOfMass* constraints cannot be used together with *FixAtomConstraints* in the same MD simulation.

FixElectricFieldCorrection

class FixElectricFieldCorrection(*polarization=None*, *born_effective_charge=None*)

A class to control the update of Polarization and Born Effective Charges analysis during optimization and molecular dynamics with electric field.

It keeps the Polarization and Born Effective Charge fixed to the value calculated with the first configuration passed at initialization of the optimization or molecular dynamics.

Parameters

- **polarization** (*Polarization* | list of *Polarization*) – Provide the value of Polarization, instead of being calculated internally. When used together with *NudgedElasticBand*, it must be a list with one entry per image. Single entries can be either None or *Polarization*. **Default:** The Polarization is calculated internally.
- **born_effective_charge** (*BornEffectiveCharge* | list of *BornEffectiveCharge*) – Provide the value of Born Effective Charge, instead of being calculated internally. When used together with NEB, it must be a list with one analysys object per image. **Default:** The Born Effective Charge is calculated internally.

FixStrain

class FixStrain(*x=False*, *y=False*, *z=False*)

When used as input argument for a stress calculation the stress is set to zero for the indicated components of the stress tensor.

When used as input argument for an optimization the strain is kept fixed along the indicated directions.

Parameters

- **x** (*bool*) – Indicating constraint along the x-axis. **Default:** False
- **y** (*bool*) – Indicating constraint along the y-ayis. **Default:** False
- **z** (*bool*) – Indicating constraint along the z-azis. **Default:** False

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

toList()

Returns
The constraints as a list of Booleans

Return type
list of type bool

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Optimize a bulk system by constraining along the x- and z-axis:

```
# Set up lattice
lattice = SimpleCubic(3.731*Angstrom)

# Define elements
elements = [Silver, Cerium]

# Define coordinates
cartesian_coordinates = [[ 0.    ,  0.    ,  0.    ],
                         [ 1.8655,  1.8655,  1.8655]]*Angstrom

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

bulk_configuration.setCalculator(LCAOCalculator())

# Run the optimization
constraints = [FixStrain(x=True, y=False, z=True)]
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        constraints=constraints,
        )
```

fixstrain.py

FixedSpin

class FixedSpin(*spin_directions=None*, *fixed_spin_energy=None*)

Class for representing a fixed spin configuration.

The spins are fixed by adding an energy penalty on given atoms if their spins are not aligned with the desired direction.

Parameters

- **spin_directions** (list of tuple of (int | *PeriodicTableElement*, *PhysicalQuantity* of type angle, *PhysicalQuantity* of type angle)) – The fixed spin direction for each atom should be given as a tuple with three numbers, corresponding to the atom index and the spherical coordinates theta and phi defining the direction. Alternatively, the spin directions can also be given by element. **Default:** `(index, 0.0 * Degrees, 0.0 * Degrees)` for each atom
- **fixed_spins_energy** (*PhysicalQuantity* of type energy) – Scaling of the energy penalty if the spins are not aligned with the fixed spin direction. **Default:** `5.0 * eV`

fixedSpinEnergy()

Returns
The fixed spin energy

Return type
PhysicalQuantity of type energy

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Setup a FixedSpin object using atom indices:

```
fixed_spin = FixedSpin(spin_directions=[(0, 60*Degrees, 30*Degrees),
                                        (2, -45*Degrees, 330*Degrees)])
```

Setup a FixedSpin object by element:

```
fixed_spin = FixedSpin(spin_directions=[(Hydrogen, 90*Degrees, 30*Degrees),
                                        (Carbon, 54*Degrees, 300*Degrees)])
```

The following example demonstrates how to run a calculation with the FixedSpin object for a chromium dimer. Note that the chosen parameters might not necessarily represent a physically relevant calculation.

Warning: If FixedSpin is used in conjunction with InitialSpin, the latter should define collinear spin directions for all atoms in the system.

```
 # Define the elements and coordinates.
elements = [Chromium, Chromium]
cartesian_coordinates = [[0.0, 0.0, 0.0],
                         [0.0, 0.0, 1.77]]*Angstrom

# Set up the Cr dimer configuration.
configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Set up an LCAO calculator with non-collinear spin and
# the LDA exchange-correlation.
calculator = LCAOCalculator(
    exchange_correlation=NCLDA.PZ,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
        density_mesh_cutoff=75.0*Hartree,
        occupation_method=FermiDirac(300.0*Kelvin),
    )
)

# Set an initial spin of 1.0 in the direction (theta, phi) = (0, 0).
spin_list = [
    (0, 1.0, 0*Degrees, 0*Degrees),
    (1, 1.0, 0*Degrees, 0*Degrees)
]
initial_spin = InitialSpin(scaled_spins=spin_list)

# Set the spin to be fixed in the direction (theta_0, phi_0) = (0, 0)
# for the first atom and in the direction (theta_1, phi_1) = (15 Degrees, 0)
```

(continues on next page)

(continued from previous page)

```
# for the second atom.
theta_0_fixed = 0*Degrees
theta_1_fixed = 15*Degrees
fixed_spin_list = [(0, theta_0_fixed, 0*Degrees), (1, theta_1_fixed, 0*Degrees)]
fixed_spin = FixedSpin(spin_directions=fixed_spin_list, fixed_spin_energy=5*eV)

# Set the fixed spins as a local magnetic field.
configuration.setMagneticField(fixed_spin)

# Set the calculator on the configuration and run the calculation.
configuration.setCalculator(calculator, initial_spin=initial_spin)
configuration.update()
```

ForceBiasMonteCarlo

class ForceBiasMonteCarlo(*temperature=None*, *max_atom_displacement=None*, *heating_rate=None*, *random_seed=None*, *max_random_attempts=None*)

Set up the ForceBiasMonteCarlo object.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature | None) – The temperature at which the Monte Carlo simulation should be run. **Default:** 300*Kelvin
- **max_atom_displacement** (*PhysicalQuantity* of type length | None) – The maximum distance an atom can move in each Cartesian direction during a single step. **Default:** 0.1*Angstrom
- **heating_rate** (*PhysicalQuantity* of type temperature | None) – The change in temperature per step. **Default:** 0*Kelvin
- **random_seed** (*int | None*) – The seed for the random generator. Must be between 0 and 2**32. **Default:** The default random seed
- **max_random_attempts** (*int*) – The maximum attempts used in the rejection sampling of the probability distribution. **Default:** 500

monteCarloStep(*configuration*, *forces*, *stress*, *constraints=None*)

Perform a Monte Carlo step and apply the new positions to the configuration.

Parameters

- **configuration** (*MoleculeConfiguration | BulkConfiguration | DeviceConfiguration | SurfaceConfiguration*) – The configuration on which the Monte Carlo step should be performed.
- **forces** (*PhysicalQuantity* of type energy/length) – The atomic forces for the given configuration.
- **stress** (*PhysicalQuantity* of type pressure) – The current stress in the unit cell.
- **constraints** (*list of type BaseConstraint | None*) – The list of constraints to apply.

monteCarloTimeStep()

Returns
The average time that elapses between each step according to the time-stamped force bias MC (tfMC) formalism or None if not initialized.

Return type
PhysicalQuantity of type time | None

randomSeed()

Returns
The random seed used to initialize the random number generator.

Return type
int

reservoirTemperature()

Returns
The current reservoir temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Use the *ForceBiasMonteCarlo* object to run a *TimeStampedForceBiasMonteCarlo* simulation:

```
# ------------------------------------------------------------
# Time-Stamped Force-Bias Monte Carlo
# ------------------------------------------------------------

method = ForceBiasMonteCarlo(
    reservoir_temperature=500.0*Kelvin,
    max_atom_displacement=0.3*Ang,
)

mc_trajectory = TimeStampedForceBiasMonteCarlo(
    bulk_configuration,
    constraints=[],
    trajectory_filename='tfmc_trajectory.hdf5',
    steps=500,
    log_interval=50,
    method=method
)
```

tfmc_example.py

Notes

The *ForceBiasMonteCarlo* class is used in the *TimeStampedForceBiasMonteCarlo* function to run a time-stamped force-bias Monte Carlo (TFMC) simulation[1].

A Monte Carlo step is taken by displacing each Cartesian component α of the position of atom i by a distance $\zeta_{i,\alpha}\Delta_i$. The maximum displacement magnitude for each atom is given by $\Delta_i = \Delta\,(m_{\text{min}}/m_i)^{1/4}$[2], where Δ is specified by the argument `max_atom_displacement`, and m_i are the atomic masses.

[1] ErikăC. Neyts and Annemie Bogaerts. Combining molecular dynamics with Monte Carlo simulations: implementations and applications. *Theor. Chem. Acc.*, 132(2):1320, 2012. doi:10.1007/s00214-012-1320-x.

[2] KristofăM. Bal and ErikăC. Neyts. On the time scale associated with Monte Carlo simulations. *J. Chem. Phys.*, 141(20):204104, 2014. doi:10.1063/1.4902136.

$\zeta_{i,\alpha}$ is sampled from the interval $[-1, 1]$ with a probability:

$$P(\zeta_{i,\alpha}) = \begin{cases} \frac{\exp(\gamma_{i,\alpha}(2\zeta_{i,\alpha}+1))-\exp(-\gamma_{i,\alpha})}{\exp(\gamma_{i,\alpha})-\exp(-\gamma_{i,\alpha})} & \zeta_{i,\alpha} \in [-1, 0[\\ \frac{\exp(\gamma_{i,\alpha})-\exp(\gamma_{i,\alpha}(2\zeta_{i,\alpha}-1))}{\exp(\gamma_{i,\alpha})-\exp(-\gamma_{i,\alpha})} & \zeta_{i,\alpha} \in]0, 1] \end{cases}$$

where $\gamma_{i,\alpha} = 0.5F_{i,\alpha}\Delta/k_BT$, $F_{i,\alpha}$ is the force component on atom *i*, and T is the temperature at which the configurations are sampled, specified by the argument `reservoir_temperature`.

By adjusting the `max_atom_displacement` parameter one can tune accuracy vs. efficiency of the simulation. A small value results in a more accurate sampling of the canonical ensemble, whereas a large value increases the efficiency at which the phase space is sampled. Typically, values in the range $0.1R_{eq} - 0.3R_{eq}$ are a reasonable choice, where R_{eq} represents an equilibrium bond length in the system.

If a non-zero `heating_rate` is specified, the reservoir temperature will be changed by the given value after each Monte Carlo step, resulting in an increase or decrease of the temperature during sampling.

ForceBiasMonteCarloNPTBerendsen

class ForceBiasMonteCarloNPTBerendsen(*temperature=None*, *pressure=None*, *barostat_factor=None*, *compressibility=None*, *coupling_mask=None*, *max_atom_displacement=None*, *heating_rate=None*, *compression_rate=None*, *random_seed=None*, *max_random_attempts=None*)

Set up the ForceBiasMonteCarlo object with NPT Berendsen pressure control.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature | None) – The temperature at which the Monte Carlo simulation should be run. **Default:** `300*Kelvin`
- **pressure** (*PhysicalQuantity* of type pressure) – The reservoir pressure in the simulation. The pressure can be given either as a scalar value, as a vector of length 3 denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. A scalar value will result in isotropic pressure coupling, whereas for all other representations, each pressure component will be coupled to the barostat independently. **Default:** `1.0*bar`
- **barostat_factor** (*float*) – Defines the barostat timescale by taking the product of this factor and the time step. **Default:** `500`
- **compressibility** (*PhysicalQuantity* of type pressure**-1) – The estimated compressibility of the system relating volume changes to pressures changes. **Default:** `1.0e-4*bar**-1`
- **coupling_mask** (*array of bools*) – The mask determining which elements of the stress tensor the barostat should couple to. Can be given as a vector of length 3, denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. By default the barostat couples only to the diagonal elements. **Default:** True for diagonal elements, False for shear components.
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `None`
- **max_atom_displacement** (*PhysicalQuantity* of type length | None) – The maximum distance an atom can move in each Cartesian direction during a single step. **Default:** `0.1*Angstrom`
- **compression_rate** (*PhysicalQuantity* of type pressure | None) – The change in the pressure per step. A value of None disables the compression of the system. **Default:** `None`
- **random_seed** (*int | None*) – The seed for the random generator. Must be between 0 and 2**32. **Default:** The default random seed

- **max_random_attempts** (*int*) – The maximum attempts used in the rejection sampling of the probability distribution.
  Default: 500

monteCarloStep(*configuration*, *forces*, *stress*, *constraints=None*)

Perform a Monte Carlo step, applying new positions to the configuration and scaling the unit cell.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration on which the Monte Carlo step should be performed.
- **forces** (*PhysicalQuantity* of type energy/length) – The atomic forces for the given configuration.
- **stress** (*PhysicalQuantity* of type pressure) – The current stress in the unit cell.
- **constraints** (*list of type BaseConstraint | None*) – The list of constraints to apply.

monteCarloTimeStep()

Returns
The average time that elapses between each step according to the time-stamped force bias MC (tfMC) formalism or None if not initialized.

Return type
PhysicalQuantity of type time | None

randomSeed()

Returns
The random seed used to initialize the random number generator.

Return type
int

reservoirPressure()

Returns
The current reservoir temperature.

Return type
PhysicalQuantity of type temperature

reservoirTemperature()

Returns
The current reservoir temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Use the *ForceBiasMonteCarloNPTBerendsen* object to run a *TimeStampedForceBiasMonteCarlo* simulation:

```
# ------------------------------------------------------------
# Time-Stamped Force-Bias Monte Carlo
# ------------------------------------------------------------

method = ForceBiasMonteCarloNPTBerendsen(
    temperature=500*Kelvin,
    max_atom_displacement=0.1*Angstrom,
    pressure=1*bar,
    barostat_factor=500,
    compressibility=0.0001*bar**-1,
)

mc_trajectory = TimeStampedForceBiasMonteCarlo(
    bulk_configuration,
    constraints=[],
    trajectory_filename='tfmc_trajectory.hdf5',
    steps=500,
    log_interval=50,
    method=method
)

bulk_configuration = mc_trajectory.lastImage()
```

tfmc_example.py

Notes

The *ForceBiasMonteCarloNPTBerendsen* class is used in the *TimeStampedForceBiasMonteCarlo* function to run a time-stamped force-bias Monte Carlo (TFMC) simulation[1]. This class builds upon the method defined in the *ForceBiasMonteCarlo* to add pressure control via a Benendsen barostat. This allows the unit cell of the material to change during the simulation in response to an external pressure.

A Monte Carlo step is taken by displacing each Cartesian component, using the same algorithm as in the *ForceBiasMonteCarlo* class. In addition to the atomic displacements, the unit cell is also scaled at each step according to the Berendsen barostat, as described in Ref.[2]. The target pressure in the simulation is specified with the argument `pressure`. If a single scalar pressure value is given, isotropic pressure coupling is applied, scaling all cell vectors by the same factor. If a pressure vector (of length 3 or 6, in Voigt notation), or a 3x3-tensor is given, anisotropic pressure coupling is applied, allowing each unit cell vector to be scaled independently.

By adjusting the `max_atom_displacement` parameter one can tune accuracy vs. efficiency of the simulation. A small value results in a more accurate sampling of the canonical ensemble, whereas a large value increases the efficiency at which the phase space is sampled. Typically, values in the range $0.1R_{eq} - 0.3R_{eq}$ are a reasonable choice, where R_{eq} represents an equilibrium bond length in the system.

The response rate of the unit cell to the Berendsen barostat is controlled by two factors. The first of these is the barostat scaling factor, specified with `barostat_factor`. This factor, given as a dimensionless scalar, is combined with the

[1] ErikăC. Neyts and Annemie Bogaerts. Combining molecular dynamics with Monte Carlo simulations: implementations and applications. *Theor. Chem. Acc.*, 132(2):1320, 2012. doi:10.1007/s00214-012-1320-x.

[2] H.ăJ.ăC. Berendsen, J.ăP.ăM. Postma, W.ăF. van Gunsteren, A.ăDiNola, and J.ăR. Haak. Molecular dynamics with coupling to an external bath. *J. Chem. Phys.*, 81(8):3684–3690, 1984. doi:http://dx.doi.org/10.1063/1.448118.

Monte Carlo time step to give a time scale of how rapidly the unit cell adjusts to changes in pressure. The second variable is the compressibility of the material, which is specified with the argument `compressibility`. This gives a measure of the pressure required to create a change in volume. The default value of this parameter is the compressibility of water, which is usually sufficient in most simulations, as reasonable changes in the compressibility do not significantly alter the simulation.

In cases where anisotropic pressure is specified, different cell dimensions can be coupled or uncoupled with the barostat by using the argument `coupling_mask`. This takes a 3x3 matrix of boolean values specifying whether or not the cell dimension is coupled to the barostat.

The target temperature and pressure during the simulation can also be scaled. If a non-zero `heating_rate` is specified, the reservoir temperature will be changed by the given value after each Monte Carlo step, resulting in an increase or decrease of the temperature during sampling. As with the temperature, the target pressure can also be scaled using the argument `compression_rate.` Note that due to technical reasons, the MC time step will be kept constant during the simulation, also when using a finite heating rate, although strictly speaking the time scale depends on the temperature. This means that time-dependent quantities calculated from heating or cooling simulations should be considered only an approximation.

ForceCappedEquilibration

class ForceCappedEquilibration(*temperature=None*, *md_steps_per_force_capped_simulation=None*, *md_time_step=None*, *force_capped_simulations=None*, *starting_factor=None*, *ending_factor=None*, *fixed_indices=None*, *lennard_jones_cutoff=None*)

Force-capped equilibration for polymer melt systems.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the force-capped equilibration will be run. **Default:** 300 Kelvin
- **md_steps_per_force_capped_simulation** (*int*) – The number of MD steps that are run at each level of force capping. **Default:** 40000
- **md_time_step** (*PhysicalQuantity* of type time) – The timestep used in the molecular dynamics simulation. **Default:** 0.5 fs
- **force_capped_simulations** (*int*) – The number of force-capped simulations. Each simulation is run at a different force-capping factor. **Default:** 4
- **starting_factor** (*float*) – The starting position for the inclusion of force capping, expressed as a fraction of the sigma value in each potential. **Default:** 1.04
- **ending_factor** (*float*) – The ending position for the inclusion of force capping, expressed as a fraction of the sigma value in each potential. **Default:** 0.8
- **fixed_indices** (*list of int | None*) – A list of indices that should be fixed during the force-capped equilibration.
- **lennard_jones_cutoff** (*PhysicalQuantity* of type length) – The cutoff beyond which non-bonding interactions are ignored. **Default:** 8 Angstrom

runEquilibration(*configuration*, *initial_bond_only_step=False*, *trajectory_filename=None*, *log_interval=None*)

Run the force-capped equilibration protocol on the given configuration.

Parameters

- **configuration** (BulkConfiguration) – The starting configuration for the equilibration.

- **initial_bond_only_step** (*bool*) – Run an initial equilibration cycle without any non-bonding interactions. **Default:** False
- **trajectory_filename** (*str | None*) – The name of the output trajectory file.
- **log_interval** (*int | None*) – The frequency of saving configurations to output trajectory. **Default:** Number of MD steps

Returns
The equilibrated configuration.

Return type
BulkConfiguration

Usage Examples

Create a polymer model of poly(methyl methacrylate) and use the ForceCappedEquilibration object to remove atomic overlaps in the structure created by the PolymerMonteCarloBuilder.

```
# Load the monomer for the polymer
monomer = nlread('Methyl_Methacrylate.hdf5')[-1]

# Define the polymer sequence as containing 10 chains of 100 monomers each, joined␣
↪atactically.
sequence = PolymerSequence(
    [monomer],
    number_of_monomers=100,
    number_of_chains=10,
)

# Build the initial polymer structure to a correct density for the given polymer.
builder = PolymerMonteCarloBuilder(
    sequence,
    density=1.18*gram*cm**-3,
)
configuration = builder.buildPolymerConfiguration()

# Place an OPLS potential on the configuration as the basis of the force capped␣
↪equilibration.
potential = OPLSPotentialBuilder()
calculator = potential.createCalculator(configuration)
configuration.setCalculator(calculator)

# Equilibrate the structure and save the resulting configuration to disk.
equilibrator = ForceCappedEquilibration()
equilibrated_structure = equilibrator.runEquilibration(configuration)
nlsave('Final_Configuration.hdf5', equilibrated_structure)
```

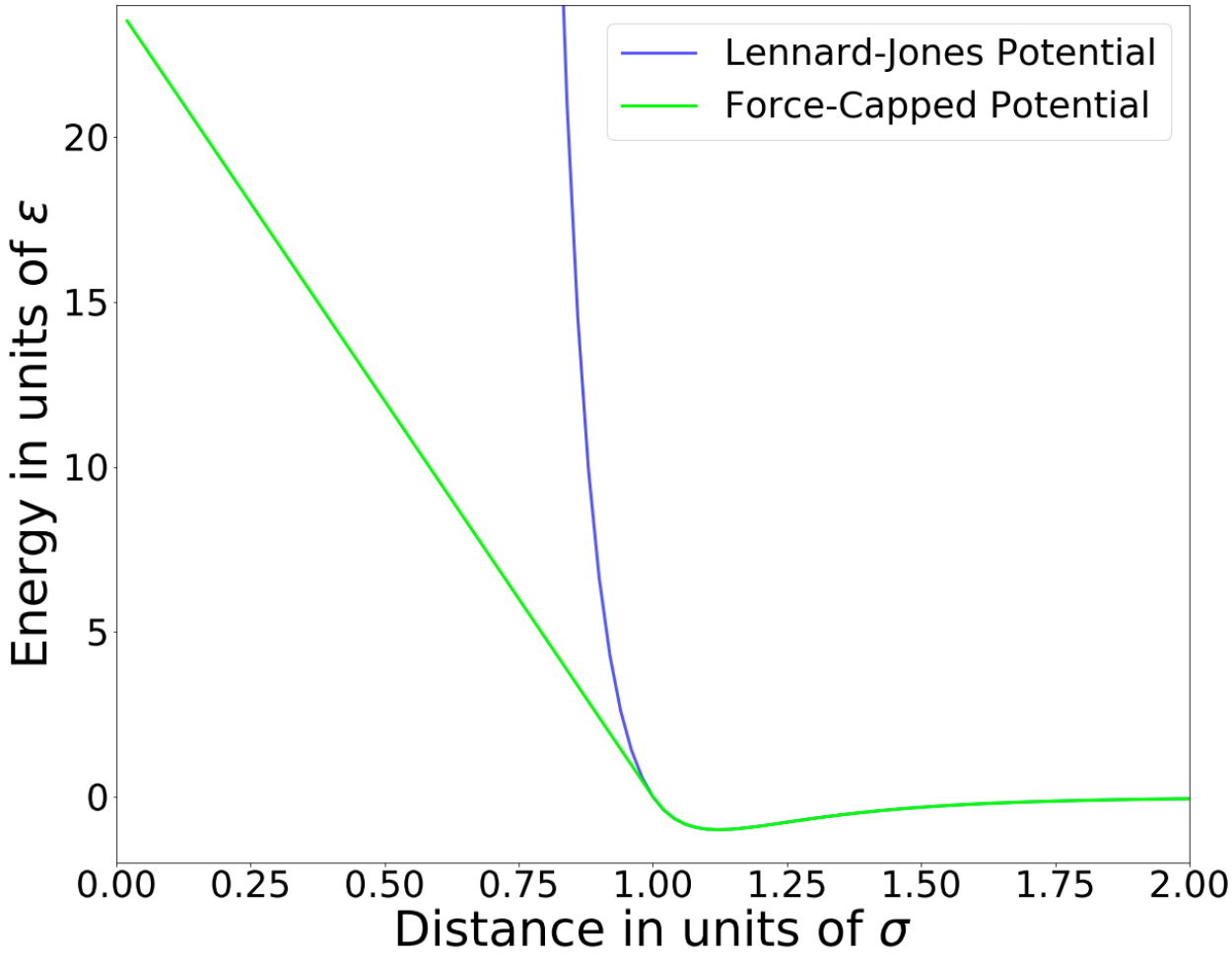
ForceCappedEquilibration_Example.py Methyl_Methacrylate.hdf5

Notes

When creating realistic models of polymer systems, it is important to ensure that the internal degrees of freedom in the polymer chain are properly randomized. In QuantumATK, this is achieved with the *PolymerMonteCarloBuilder*. Once the chains are generated, it is then necessary to pack them correctly into a unit cell so that none of the atoms overlap. Overlapping atoms generally result in very large interatomic forces that make molecular dynamics and other types of simulations impossible. To ensure that the polymer chains are well packed, one approach is to use the `ForceCappedEquilibration` class[1].

The `ForceCappedEquilibration` class contains an equilibration protocol that combines optimization and molecular dynamics. This separates overlapping polymer chains and returns a structure that is ready for further simulation. In order to do this the input configuration needs to contain a TremoloX calculator that contains *LennardJonesSplinePotential* potentials. Both the *DreidingPotentialBuilder* and the *OPLSPotentialBuilder* create potentials that contain *LennardJonesSplinePotential* potentials to model van der Waals non-bonding interactions.

The force-capped equilibration protocol avoids the large forces associated with overlapping or close atoms by assuming a constant non-bonding interaction force at particle distances below a certain switching value. An example of such as potential is shown in the figure below, along with the reference Lennard-Jones potential. Here force-capping has been applied where the potential crosses the x-axis. Beyond this point both potentials are identical at larger separation.



[1] Rolf Auhl, Ralf Everaers, GaryăS. Grest, Kurt Kremer, and StevenăJ. Plimpton. Equilibration of long chain polymer melts in computer simulations. *J. Chem. Phys.*, 119(24):12718–12728, 2003. URL: https://doi.org/10.1063/1.1628670, doi:10.1063/1.1628670.

In QuantumATK the Lennard-Jones function is defined as:

$$V(r) = 4\epsilon \left(\left[\frac{\sigma}{r}\right]^{12} - \left[\frac{\sigma}{r}\right]^{6} \right)$$

Here σ is the position at which the potential is zero and ϵ is the depth of the potential minimum. The potential minimum is located at $2^{1/6}\sigma$. In a molecule the values for σ and ϵ usually differ between pairs of atomic types to reflect differing chemical environments. Since σ varies between potentials, the position where force-capping is applied is defined relative to the σ value for that potential. During equilibration, this position where force-capping is applied is reduced over a number of cycles. This makes the applied potential increasingly similar to the full Lennard-Jones potential as the equilibration continues and the atomic overlaps are resolved. As electrostatic interactions are difficult to treat in this manner, any defined electrostatic interactions in the input configuration are ignored during force-capped equilibration and can be switched on after the force-capped equilibration.

The number of force-capped equilibration cycles is controlled by the parameter `force_capped_simulations`. A new force-capped potential is used in each cycle of equilibration. The range of force-capping factors is controlled by the input parameters `starting_factor` and `ending_factor`, which select the starting and final ratios of sigma. If `force_capped_simulations` is larger than two, the remaining factors are evenly spaced between the starting and ending factors.

During each equilibration cycle a geometry optimization, followed by an NVT molecular dynamics run, are performed. These can be modified using some additional parameters. The parameter `temperature` sets the temperature at which the molecular dynamics are carried out. This allows the simulation temperature to be set to the desired temperature of the final polymer model. The number of molecular dynamics steps in each cycle is controlled with the parameter `md_steps_per_force_capped_simulation`. The longer each MD simulation is run, the slower the non-bonded interactions are introduced, which can reduce deformation of the chain structure for long polymer melts[Page 2921, 1].

Once an instance of the *ForceCappedEquilibration* class is defined, the equilibration is carried out using the `runEquilibration` function. As well as an input configuration, this takes an optional input argument `initial_bond_only_step`. Setting this argument to `True` causes an additional initial equilibration step to be carried out in which no non-bonding interactions are taken into account. This can be useful in cases where the polymer system has some highly strained internal degrees of freedom, such as torsions or bond angles, that need to be relaxed before packing of the polymer chains can be considered. By default this initial equilibration step is not performed. It is also possible to output trajectories of the molecular dynamics steps using the parameters `trajectory_filename` and `log_interval` to set the file name and the number of steps between images respectively.

Forces

class Forces(*configuration*)

Class for calculating the forces on the atomic configuration.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – Configuration with a calculator that supports forces calculations.

evaluate()

Returns
The atomic forces. The shape of the force matrix is (number_of_atoms, 3).

Return type
PhysicalQuantity with the unit `eV / Angstrom`

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate and print the forces on a water molecule:

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,  -1.70000000e-05,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# define a a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# calculate and print the forces
forces = Forces(molecule_configuration)
forces.nlprint()
```

forces.py

Notes

To print the data of the *Forces*, use the method *nlprint*.

FourierAnglePotential

class FourierAnglePotential(*particleType1*, *particleType2*, *particleType3*, *K*, *C*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **K** (*PhysicalQuantity of type energy*) – Potential parameter.
- **C** (*Sequence of floats*) – Potential parameter. The length of this list must not exceed 5.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

FourierImproperPotential

class FourierImproperPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *K*, *C*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.
- **K** (*PhysicalQuantity of type energy*) – Potential parameter.

•**C** (*Sequence of floats*) – Potential parameter. The length of this list must not exceed 10.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

FourierTorsionPotential

class FourierTorsionPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *K*, *C*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.

•**K** (*PhysicalQuantity of type energy*) – Potential parameter.

•**C** (*Sequence of floats*) – Potential parameter. The length of this list must not exceed 5.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

FragmentCalculator

class FragmentCalculator(*full_system_calculator*, *fragment_generator*, *number_of_processes_per_fragment=None*, *use_global_fermi_level=None*, *workload_partitioning_algorithm=None*)

Class for representing calculations using the divide-and-conquer linear scaling fragment method with a given atomic scale calculator.

Parameters

•**full_system_calculator** (*LCAOCalculator*) – The calculator that each of the fragment configurations will base their calculator on.

•**fragment_generator** (*FragmentGenerator*) – The object that describes how the fragments should be created and is responsible for creating them.

•**number_of_processes_per_fragment** (*positive int*) – The number of processes that will be used in the calculation of each fragment. If the total number of processes does not divide evenly into the number of fragments, some fragment process groups will have less than this number of processes. **Default:** 1

•**use_global_fermi_level** (*bool*) – Boolean controlling if a global Fermi level should be used in the calculation. This should be `True` for calculations involving metals or interface with an expected charge transfer. For pure semi-conductor systems, setting it to `False` can improve convergence and performance.

•**workload_partitioning_algorithm** (*KarmarkarKarpAlgorithm | GreedyAlgorithm | CompleteGreedyAlgorithm | Interleaved | tuple*) – The algorithm to partition the fragments by their estimated workload into the process groups such that the sum of workloads in each process group is as equal as possible. One may alternatively provide a tuple with the partitioning algorithm in the first entry and an orbital power (positive integer) in the second entry. The orbital power is the power that the number of orbitals of a fragment is raised to when calculating the fragment weight. **Default:** (Interleaved, 3)

fragmentGenerator()

Returns
The object that describes how the fragments are created and is responsible for creating them.

Return type
FragmentGenerator

fullSystemCalculator()

Returns
The calculator for the full system.

Return type
LCAOCalculator

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Query method for the IterationControlParameters.

Returns
The iteration control parameters set on the calculator.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfProcessesPerFragment()

Returns
The number of processes that is used in the calculation of each fragment. If the total number of processes does not divide evenly into the number of fragments, some fragment process groups will have less than this number of processes. The number of processes per fragment is capped at the number of processes available.

Return type
int

requestedNumberOfProcessesPerFragment()

Returns
The number of processes asked for (on construction of the calculator) to be used in the calculation of each fragment. This number can be higher than the number of processes available and will be tried to be honored when a calculator is read in again from file and the number of processes per fragment should be re-evaluated.

Return type
int

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

upgrade(*full_system_configuration*)

Upgrade the full system calculator with defaults matching the configuration and create the fragment configurations for the given configuration.

Parameters
full_system_configuration (*BulkConfiguration*) – The configuration to upgrade the calculator defaults for and the configuration to create fragment configurations for.

useGlobalFermiLevel()

Returns
Boolean controlling if a global Fermi level should be used in the calculation.

Return type
bool

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

workloadPartitioningAlgorithm()

Parameters
workload_partitioning_algorithm (*KarmarkarKarpAlgorithm | GreedyAlgorithm | CompleteGreedyAlgorithm | Interleaved*) – The algorithm to partition the tasks by their estimated workload into the process groups such that the sum of workloads in each process group is as equal as possible. **Default:** KarmarkarKarpAlgorithm

Notes

The *FragmentCalculator* is a linear scaling method based on the *LCAOCalculator*. The *FragmentCalculator* can be used to simulate large systems with more than 10,000 atoms using DFT. The method shows good parallel scaling with respect to both computation time and memory reduction.

The underlying method implemented in the *FragmentCalculator* is based on the work of Lin-Wang Wang et. al,[1],[2]. The QuantumATK implementation largely follows the concepts and ideas presented in these articles. The main difference is that while the work of Wang. et al is based on a plane-wave method, the QuantumATK implementation is currently implemented for LCAOCalculations.

Below we will present the main concepts behind the *FragmentCalculator* and highlight differences with respect to the original works.

Scaling with system size

The computation time for an SCF step of any traditional DFT method (LCAO and plane-wave) scales as $t_{scf}^{DFT} \propto O(N^3)$, where N is the number of atoms in the system. This is due to the expensive calculation of eigenstates when solving the Kohn-Sham eigenvalue problem.

The main idea behind the *FragmentCalculator* method is to divide a large system into a number smaller systems (fragments). The number of fragments, $K_f = k \cdot N$ will be proportional to the total number of atoms, with a proportionality factor that depends on the specific type of fragmentation strategy. Denoting the size of the small, fragment, systems as N_f, the time for solving the eigenvalue problem for the K_f fragments scales as $t_{scf}^{Fragment} \propto k \cdot N \cdot O(N_f^3)$. Since the fragment size N_f is constant we see that the calculation time scales linearly with system size N through the number of fragments.

[1] Lin-Wang Wang, Zhengji Zhao, and Juan Meza. Linear-scaling three-dimensional fragment method for large-scale electronic structure calculations. *Phys. Rev. B*, 77:165113, Apr 2008. URL: https://link.aps.org/doi/10.1103/PhysRevB.77.165113, doi:10.1103/PhysRevB.77.165113.

[2] Zhengji Zhao, Juan Meza, and Lin-Wang Wang. A divide-and-conquer linear scaling three-dimensional fragment method for large scale electronic structure calculations. *Journal of Physics: Condensed Matter*, 20(29):294203, jun 2008. URL: https://doi.org/10.1088/0953-8984/20/29/294203, doi:10.1088/0953-8984/20/29/294203.

Equally import to the scaling of computation time is the scaling of the required memory with system size. For LCAO, the memory required for storing the eigenvectors scales as $O(N^2)$, since each eigenvector grows linearly with system size, and a linearly increasing number of eigenstates are required to construct the density. For the *FragmentCalculator* the required memory scales linearly with system size, since the memory required for the eigenstates is constant for an individual fragment, while the number of fragments scale linearly with system size. The memory required for the real-space density and effective potential scales linear with system size for both normal LCAO and the fragment method. It should be noted, that the fragment method has a significant memory overhead for small systems.

The linear scaling of both the calculation time and required memory implies that very large systems can be studied with the *FragmentCalculator*, provided that the number of processors is increased proportionally.

Division into fragments

Following Refs.[Page 2928, 1],[Page 2928, 2] we divide the original, big system into a number of fragments. The actual fragmentation is defined using a *FragmentGenerator*. The *FragmentGenerator* must be defined in connection with the *FragmentCalculator*. and both classes are dependent on each other. The *FragmentGenerator* provides three mutually exclusive options for fragmentation:

1. `fractional_cuts`
2. `number_of_minimal_fragments`
3. `fragment_lengths`

More details about fragmentation is provided here: *FragmentGenerator*.

The fragmentation specified by one of the three options defines how to cut out the minimal fragments. In addition to the interior of a fragment, one can add a buffer region to fragment. The buffer region is included in the fragment Kohn-Sham equation, but is not included when calculating the electron density. Including 1-3 Å of buffer region usually improves the quality of the fragment calculation of both total energy and forces.

When cutting out a fragment, potentially including buffer atoms, a vacuum region is added in the directions of fragmentation. The boundary atoms facing vacuum are passivated with hydrogen atoms positioned at the center of the cut bonds.

Following Refs.[Page 2928, 1],[Page 2928, 2] the fragments have different signs, either +1 or -1, depending on their type. The total electron density in a given point is calculated as

$$\rho(\mathbf{r}) = \sum_F \alpha_F \rho_F(\mathbf{r})$$

where the sum runs over fragments covering the point $\mathbf{r}$ and where the fragment sign α_F is either +1 or -1. The fragment density $\rho_F(\mathbf{r})$ is obtained from solving a (small) fragment Kohn-Sham equation:

$$\left[-\frac{1}{2}\nabla^2 + V_{eff}(\mathbf{r}) + V_{F,p}(\mathbf{r})\right] \psi_{F,i}(\mathbf{r}) = \epsilon_{F,i}\psi_{F,i}(\mathbf{r})$$

where $V_{eff}(\mathbf{r})$ is the usual effective potential and $V_{F,p}(\mathbf{r})$ is a passivation potential taken as the difference between the neutral atom potentials of the fragment configuration and the full configuration.

1D, 2D, and 3D fragmentation and scaling considerations

The *FragmentGenerator* and *FragmentCalculator* can be used to divide the initial configuration into 1D, 2D or 3D fragments depending on the input to *FragmentGenerator* as described in the *FragmentGenerator* manual page.

The performance improvement of the *FragmentCalculator* compared to normal LCAO depends on the fragmentation and weather it is 1D, 2D, or 3D fragments.

For **1D** fragmentation, the fragments generated will either be single fragments, denoted (1,1,1) or double fragments, denoted (1,1,2) for fragmentation along the C-direction. The total number of fragments is thus $N_f = 2 \cdot N_{single}$, where N_{single} is the number of single fragments.

Consider now a system consisting of a primitive unit cell repeated N_C times along the C-direction. The number of orbitals in the primitive cell is denoted P. For the fragment calculation, we divide this system into $N_f = N_C$ number of minimal fragments. If we only take into account the $O(N^3)$ cost of solving the Kohn-Sham eigenvalue problem, the idealized speedup of the fragment method compared to normal LCAO is

$$S_{1D} = \frac{T_{fragment}}{T_{LCAO}} = \frac{N_C \left(P^3 + (2P)^3\right)}{(N_C P)^3} = \frac{9}{N_C^2}$$

It is thus evident that for less than 3 minimal fragments, the fragment calculation will always be slower than a normal LCAO calculation. In the ideal case, the fragment calculation will be faster than normal LCAO for $N_C > 3$. In practice, the cross-over point will often be observed at somewhat larger $N_C \sim 5 - 6$ due to other computational costs related to real-space grid operations and other overhead.

For **2D** fragmentation along B and C directions, the fragments are (1,1,1), (1,2,1), (1,1,2), and (1,2,2). For each single fragment there will be 4 fragments in total. The idealized speedup for a primitive configuration repeated N_B and N_C times along the B and C directions is in this case

$$S_{2D} = \frac{T_{fragment}}{T_{LCAO}} = \frac{N_B N_C \left(\cdot P^3 + 2 \cdot (2P)^3 + (4P)^3\right)}{(N_B N_C P)^3} = \frac{81}{N_B^2 N_C^2}$$

Again, we see an idealized break-even is seen for three repetitions along each of the direction, $N_B = N_C = 3$.

For **3D** fragmentation the fragments are (1,1,1), (2,1,1), (1,2,1), (1,1,2), (2,2,1), (2,1,2), (1,2,2), (2,2,2) i.e. 8 fragments in total for each single fragment. A 5x5x5 fragmentation thus leads to 1000 fragments in total. The idealized speedup for a primitive configuration repeated N_A, N_B and N_C times along the A, B and C directions is in this case

$$S_{3D} = \frac{T_{fragment}}{T_{LCAO}} = \frac{N_A N_B N_C \left(\cdot P^3 + 3 \cdot (2P)^3 + 3(4P)^3 + (8P)^3\right)}{(N_A N_B N_C P)^3} = \frac{729}{N_A^2 N_B^2 N_C^2}$$

Again, we see an idealized break-even is seen for three repetitions along each of the direction, $N_A = N_B = N_C = 3$, but in practice the fragment method will be faster at a higher number of repetitions.

Recommended fragment sizes

The fragment method is an approximation to the full solution. In order to be a good approximation, the size of the fragments needs to be of a certain size. We recommend fragment sizes in the range of 5-10 Å in order to achieve accurate, yet fast results. In addition we recommend to use a buffer region of size 2-3 Å. Optimizing these numbers might vary from problem to problem, but these recommendation should serve as a starting point for further optimizations.

Load imbalance of the process groups

When going to higher dimension fragmentation the size difference between the smallest fragment and the largest fragment becomes quite significant. Most prominently for 3D fragmentation the size difference between the (1,1,1) fragment and its (2,2,2) fragment is a factor of 8. If one at the same time have many process groups as a result of having many processes, the number of fragments per process group is small and it will be increasingly important to partition the fragments optimally between the process group. A bad parallel scaling with increasing number of processes might be a sign that the used partition of fragments for the process groups leads to a bad load balance. To try to improve the workload partition of the process groups one can change the `workload_partitioning_algorithm` from `Interleaved` to `KarmarkarKarpAlgorithm`. The number partitioning algorithms Karmarkar-Karp, Greedy and complete Greedy are described in[3]. The complete Greedy algorithm will find the most optimal partition of the fragments, but it is not recommended to use since it is very slow and scales bad.

Other numerical settings

Fragment method calculations generally use more SCF step to converge than a similar LCAO calculation. In order to achieve stable convergence we recommend:

- Decrease the `damping_factor` in *IterationControlParameters* to 0.05 or below.
- Decrease the broadening used in the `occupation_method` in :ref:NumericalAccuracyParameters_c` from default 1000 Kelvin to 300 Kelvin or below.

Supported analysis

Not all analysis objects are supported by the *FragmentCalculator*. At present, the following objects are available:

- TotalEnergy
- Forces
- Stress
- OptimizeGeometry
- MolecularDynamics
- ElectronDensity
- ElectronDifferenceDensity
- HartreeDifferencePotential
- DynamicalMatrix
  - PhononBandstructure
  - PhononDensityOfStates
  - VibrationalMode

[3] RichardăE. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106:181–203, Dec 1998. URL: https://doi.org/10.1016/S0004-3702(98)00086-1, doi:10.1016/S0004-3702(98)00086-1.

References

FragmentGenerator

class FragmentGenerator(*vacuum_length=None*, *buffer_length=None*, *fractional_coordinates_cuts=None*, *number_of_minimal_fragments=None*, *fragment_lengths=None*, *fuzz_factor=None*, *passivation_parameters=None*)

Class for generating fragments of a bulk configuration.

Parameters

- **vacuum_length** (float of *PhysicalQuantity* type length) – The thickness at each boundary that should be covered with vacuum. **Default:** `5.0 * Angstrom`.
- **buffer_length** (float of *PhysicalQuantity* type length) – The thickness at each boundary that should be filled with existing adjacent atoms from the adjacent fragments. **Default:** `0.0 * Angstrom`.
- **fractional_coordinates_cuts** (*list of size 3 of floats in range (0, 1) or None*) – The cuts in fractional coordinates along the primitive vectors of the cell. NOTE: A None element means that the given dimension should not be fragmented. This option is mutually exclusive to `number_of_minimal_fragments` and `fragment_lengths`.
- **number_of_minimal_fragments** (*tuple (size 3) of either None or positive int*) – The number of fragments in each directions of the cell. NOTE: A None element means that the given dimension should not be fragmented. This option is mutually exclusive to `fractional_coordinates_cuts` and `fragment_lengths`.
- **fragment_lengths** (float of *PhysicalQuantity* type length | list (size 3) of None or floats of *PhysicalQuantity* type length) – The length of each fragment dimension. NOTE: A None element means that the given dimension should not be fragmented. If a single *PhysicalQuantity* is provided, this will be used in all directions. This option is mutually exclusive to `number_of_minimal_fragments` and `fractional_coordinates_cuts`.
- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond. **Default:** `1.1`
- **passivation_parameters** (Flag All or NoPassivation of `NLFlag` or a list of tuples (size 2) with an element and a flag or basis set.) – Parameter(s) to specify passivation. **Default:** `All`

bufferLength()

Returns
The buffer_length.

Return type
PhysicalQuantity of type length

fractionalCoordinatesCuts()

Returns
The fractional_coordinates_cuts.

Return type
list of tuples of floats in range (0, 1) or None

fragmentLengths()

Returns
The fragment_lengths.

Return type
list (size 3) of None or floats of *PhysicalQuantity* type length

fragmentationDimensionality()

Returns
Which of the 3 spacial directions to create fragments in.

Return type
A tuple (size 3) of bool.

fuzzFactor()

Returns
The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.

Return type
float

minimalFragmentLength()

Returns
The minimal fragment length.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
Structured information about the FragmentGenerator.

Return type
dict

numberOfMinimalFragments()

Returns
The number_of_minimal_fragments.

Return type
tuple (size 3) of either None or positive int or None

passivationParameters()

Returns
Parameter(s) to specify passivation.

Return type
Flag All or NoPassivation of `NLFlag` or a list of tuples (size 2) with an element and a flag or basis set.

uniqueString()

Return a unique string representing the state of the object.

vacuumLength()

Returns
The vacuum_length.

Return type
PhysicalQuantity of type length

Usage Examples

The *FragmentGenerator* must be defined in connection with the *FragmentCalculator* to specify the fragmentation of a larger system configuration. Both classes are fully dependent on each other. The *FragmentGenerator* provides three mutually exclusive options for fragmentation:

1. `fractional_cuts`
2. `number_of_minimal_fragments`
3. `fragment_lengths`

Note that these fragmentation approaches determine the dimensions of the minimal fragments (also termed single fragments). The `fractional_cuts` option instructs some cuts along the primitive vectors of the unit cell. This is illustrated in Fig. 4.21 as the fractional divisions along the C primitive vector.

```
# Cut the C primitive vector into 3 equal lengths.
fractional_cuts = [None, None, [1/3, 2/3]]
```

Note: The `None` inputs in the `fractional_cuts` vector indicate that no fragmentation should be performed along the A- and B directions. For the C-direction, the fractional cuts at 0 and 1 are implicit and should not be specified.

The `number_of_minimal_fragments` option specifies a certain number of divisions along the primitive vectors of the unit cell as illustrated in Fig. 4.21.

```
# Divide the full system into 3 single fragments along the C primitive vector.
number_of_minimal_fragments = [1, 1, 3]
```

Here, the numbers 1 again implies that no fragmentation will be performed along the A- and B directions.

The `fragment_lengths` option simply divides the primitive vectors in lengths of physical units accordingly,

$$L = |\vec{\mathbf{C}}|/\text{int}\left(\frac{|\vec{\mathbf{C}}|}{L_{input}}\right)$$

```
# For the purpose of this example, say L = 5 Angstrom.
L = 5 * Angstrom
fragment_lengths = [None, None, L]
```

After choosing a fragmentation method it can be given as an argument for the *FragmentGenerator*. It is only possible to use one method.

```
# Create the fragment generator with the fragmentation method.
fragment_generator = FragmentGenerator(
    fragment_lengths=fragment_lengths,
    )
```

Each fragment has its own configuration where the actual fragment atoms are embedded in vacuum. A buffer region may also be included containing atoms from neighboring fragments. The vacuum and buffer regions are specified as input arguments of the *FragmentGenerator* and only added in the given directions where fragmentation occurs.

```
# Define fragmentation method.
number_of_minimal_fragments = [1, 1, 3]
```

(continues on next page)

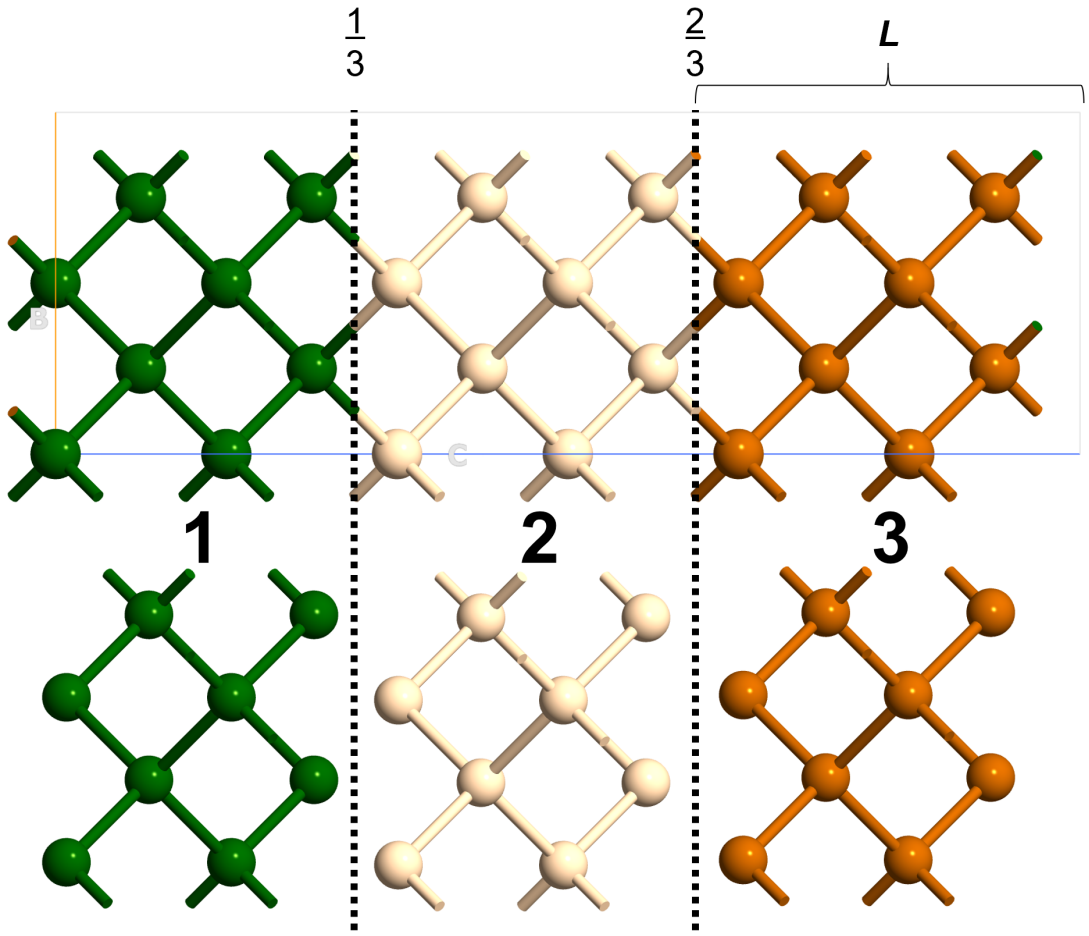


Fig. 4.21: Fragmentation methods when using *FragmentGenerator*.

(continued from previous page)

```
# Define vacuum and buffer lengths.
vacuum_length = 6.0 * Angstrom
buffer_length = 3.0 * Angstrom

# Create the fragment generator with the vacuum and buffer lengths and fragmentation␣
↪method.
fragment_generator = FragmentGenerator(
    number_of_minimal_fragments=number_of_minimal_fragments,
    vacuum_length=vacuum_length,
    buffer_length=buffer_length,
    )
```

Upon fragmentation multiple bonds might be cut between surfaces of fragments. In order to avoid occurrence of metallic surface states, passivation might be required. In some cases one might want to passivate certain elements of fragment configurations or passivate with a particular fractional hydrogen basis set. Therefore, *FragmentGenerator* can take an input `passivation_parameters` which applies for all fragment surfaces. Three options are available.

1. `All` flag indicating passivation of all elements.
2. `NoPassivation` flag indicating no passivation of any elements
3. A list of elements with respective passivation type in form of either a hydrogen basis set or a flag as in the two points above

```
# Define fragmentation method.
number_of_minimal_fragments = [1, 1, 3]

# Define passivation parameters. Say that we have a GaAs configuration and we
# want to passivate Ga with 1.25 fractional pseudo hydrogens and As with
# 0.75 fractional pseudo hydrogens.
passivation_parameters = [
    (Gallium, FractionalGGABasis.fractional_1_25_HydrogenBasis),
    (Arsenic, FractionalGGABasis.fractional_0_75_HydrogenBasis),
    ]

# Create fragment generator with passivation.
fragment_generator = FragmentGenerator(
    number_of_minimal_fragments=number_of_minimal_fragments,
    passivation_parameters=passivation_parameters,
    )
```

In all cases, the passivating Hydrogen atoms are positioned in the center of the bond cut by the surface. The definition of when two atoms form a bond is determined by a `fuzz_factor`. This is equivalent to the `fuzz_factor` used in the Viewer and Builder for visualizing bonds (this can be modified under **Graphics Properties**). The default value is 1.1, but in some cases this might need to be increased to e.g. 1.15 in order to include all relevant bonds:

```
# Define fragmentation method.
number_of_minimal_fragments = [1, 1, 3]

# Create fragment generator with passivation.
fragment_generator = FragmentGenerator(
    number_of_minimal_fragments=number_of_minimal_fragments,
    fuzz_factor=1.15,
    )
```

1D, 2D, and 3D fragmentation

The *FragmentGenerator* and *FragmentCalculator* can be used to divide the initial configuration into 1D, 2D or 3D fragments depending on the input to *FragmentGenerator* as illustrated below:

```
# 1D fragmentation: Divide the configuration into 5 single fragments.
# Total number of fragment = 10.
fragment_generator = FragmentGenerator(
   number_of_minimal_fragments=[None, None, 5],
   )


# 2D fragmentation: Divide the configuration into 5x5 = 25 single fragments.
# Total number of fragment = 100.
fragment_generator = FragmentGenerator(
   number_of_minimal_fragments=[None, 5, 5],
   )

# 3D fragmentation: Divide the configuration into 5x5x5 = 125 single fragments.
# Total number of fragment = 1000.
fragment_generator = FragmentGenerator(
   number_of_minimal_fragments=[5, 5, 5],
   )
```

For **1D** fragmentation, the fragments generated will either be single fragments, denoted (1,1,1) or double fragments, denoted (1,1,2) for fragmentation along the C-direction.

For **2D** fragmentation along B and C directions, the fragments are (1,1,1), (1,2,1), (1,1,2), and (1,2,2). For each single fragment there will be 4 fragments in total. Dividing the configuration into 5x5 single fragments thus results in 100 fragments in total.

For **3D** fragmentation the fragments are (1,1,1), (2,1,1), (1,2,1), (1,1,2), (2,2,1), (2,1,2), (1,2,2), (2,2,2) i.e. 8 fragments in total for each single fragment. A 5x5x5 fragmentation thus leads to 1000 fragments in total.

The large number of fragments when performing 3D fragmentation implies that the fragment calculation can be efficiently parallelized over many CPUs, although full parallelization will lead to significant load imbalance.

FreeVolume

class FreeVolume(*md_trajectory*, *polymer_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *probe_size=PhysicalQuantity(0.0, Ang)*, *overlap_ratio=1.0*, *samples=10000*, *random_seed=None*, *gui_worker=None*)

Create an analyzer for the free volume in a polymer structure.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or AtomicConfiguration that contains the polymers.
- **polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used. Tags can also be set automatically with the flag Automatic
- **start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.

- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **probe_size** (*PhysicalQuantity* of type length) – The size of the probe to add into calculating the excluded volume. **Default:** 0 Angstrom.
- **overlap_ratio** (*float*) – The fraction of the probe and atom radii that is considered to be part of the polymer. **Default:** 1.0.
- **samples** (*int*) – The number of volume samples to attempt.
- **random_seed** (*int | None*) – The random seed used in the random number generator. If None is given then it will be generated using values from the system entropy pool. **Default:** None.
- **gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

freeVolumes()

Returns
The free volume for each image in the given trajectory.

Return type
PhysicalQuantity of type volume

polymerTags()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

ratio()

Returns
The ratio of free to total volume for each image in the given trajectory.

Return type
ndarray

times()

Returns
Times for each image used in the analysis.

Return type
PhysicalQuantity of type time

totalVolumes()

Returns
The total unit cell volume for each image in the given trajectory.

Return type
PhysicalQuantity of type volume

Usage Examples

Calculate the free volume of a polystyrene melt during a molecular dynamics simulation starting from a well equilibrated configuration.

```
# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('Polystyrene.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Polystyrene_Trajectory.hdf5',
    steps=100000,
    log_interval=10,
    trajectory_interval=1000,
    method=method
)

# ------------------------------------------------------------
# Calculate End to End Distance
# ------------------------------------------------------------
analyzer = FreeVolume(
      md_trajectory,
      start_time=0*ps,
```

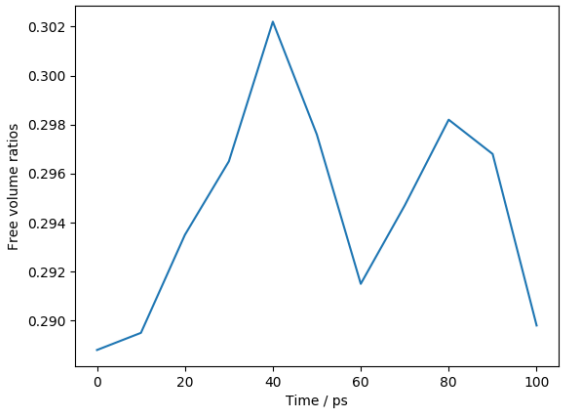
(continues on next page)

(continued from previous page)

```
      end_time=100*ps,
      time_resolution=10*ps,
      overlap_ratio=1.0,
      samples=10000
)
free_volume_ratio = analyzer.ratio()
time = analyzer.times().convertTo(ps)

# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
import pylab
pylab.figure()
pylab.plot(time, free_volume_ratio)
pylab.xlabel('Time / ps')
pylab.ylabel('Free volume ratios')
pylab.savefig('Free_Volume_Ratio_Plot.png')
```

`FreeVolume_Example.py Polystyrene.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate free volume ratio of 11 images in the 101 image trajectory. This is shown in the following plot.



Notes

This class enables the calculation of the free volumes of different polymer systems from a *MDTrajectory* or *BulkConfiguration*.

The free volume of a polymer system is defined as the volume not occupied by atoms. The free volume of polymer structures has been shown to be related to important macroscopic properties, such as the glass transition temperature[1]. Assuming atoms can be represented as hard spheres, the free volume can be estimated using a simple Monte Carlo sampling procedure. This process randomly selects a number of positions within the structure, and tests whether or not the position is within the hard sphere radius of any atom. The ratio of the number of empty points to total number of selected positions gives the free volume ratio.

[1] RonaldăP. White and Jane E.ăG. Lipson. Polymer free volume and its connection to the glass transition. *Macromolecules*, 49:3987, May 2016. URL: https://pubs.acs.org/doi/10.1021/acs.macromol.6b00215, doi:10.1021/acs.macromol.6b00215.

To calculate the hard sphere free volume a *FreeVolume* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration* and calculates the end-to-end distances and autocorrelation function for the specified configurations. The specific frames in the *MDTrajectory* to be analyzed are selected using the `start_time`, `end_time` and `time_resolution` arguments. There are also a arguments that control the Monte Carlo procedure used to calculate the free volume. The size of the probe placed at each point can be set using the `probe_size` argument. By default a probe size of zero is used, equivalent to an infinitesimally small probe. The relative size of both the probe and atoms can be scaled using the `overlap_ratio` argument. The total number of Monte Carlo trials per configuration is set with the `samples` argument. Increasing this both increases the accuracy of the calculation and the calculation time. Finally the `random_seed` argument allows passing in a seed to the random number generator, so that calculations can be repeated.

Once the *FreeVolume* object is created and the volumes of each of the configurations has been calculated, the values can be queried using the remaining methods. The method `ratio()` returns a list of the free volume ratios of each configuration. Similarly the method `freeVolumes()` returns a list of the absolute free volumes. The method `totalVolumes()` returns a list of the total volume of each configuration, which is calculated simply from the lattice parameters.

FullDiagonalizationSolver

class FullDiagonalizationSolver(*bands_above_fermi_level=None*)

Constructor for information regarding the full diagonalization solver.

Parameters

bands_above_fermi_level (Non-negative int | `All`) – The number of bands above the Fermi level per principal spin channel. **Default:** `All` (All bands are included), except for calculators of type *PlaneWaveCalculator*; in this case the default number of bands above the Fermi level is determined by the bands_per_electron parameter on *NumericalAccuracyParameters* on the *PlaneWaveCalculator*. The default value of bands_per_electron=1.2 means that if there are 10 (valence) bands below the Fermi level, there will be two bands above. If more bands are included in the calculator, they will also be calculated as default.

bandsAboveFermiLevel()

Returns

Number of bands above the Fermi level

Return type

int

uniqueString()

Return a unique string representing the state of the object.

GWCalculator

class GWCalculator(*ground_state_calculator*, *energy_cutoff=None*, *tau_precision_parameter=None*, *abc_tolerance=None*, *polarizability_compression_parameter=None*, *w_compression_parameter=None*, *include_gamma_point=None*)

Class for representing GW calculators.

Parameters

- **ground_state_calculator** (*LCAOCalculator*) – The DFT calculator that is used to generate the Kohn-Sham states on which the G0W0 algorithm is based.
- **energy_cutoff** (*PhysicalQuantity of type energy.*) – The energy cutoff decides how many bands will be included in the Greens functions of the GW calculation. At the gamma point the Kohn-Sham energy spectrum will

be checked, and all bands that have an energy above the Fermi level that is below the energy cutoff will be included. **Default:** `50.0 eV`

- **tau_precision_parameter** (*float*) – The parameter that decided how many time points will be taken. Given a (e_max : e_min) energy range, we have a minimal and maximal number of points we can include. If the precision parameters is 1 it will take the maximum, if it is 0 it will take the minimal. If in between we will take the closest integer that respects this fractional value. **Default:** `1.0`

- **abc_tolerance** (*float*) – If the norm of certain atom-pair IJ blocks in the auxiliary basis coefficients is below this threshold we will throw it away and reduce the overlap_sparsity. **Default:** `1.0e-4`

- **polarizability_compression_parameter** (*float*) – This parameter determines how aggresively the polarizability will be compressed, to save memory. **Default:** `1.0e-7`

- **w_compression_parameter** (*float*) – This parameter determines how aggresively the screened coulomb potential, W, will be compressed, to save memory. **Default:** `1.0e-5`

- **include_gamma_point** (*boolean*) – If this is true, we include the gamma point with a gamma point correction. If False, we skip the Gamma point. This can be done if you want to perform an extrapolation to infinite kpoint sampling or if you want to verify that the gamma point correction is correct. **Default:** `True`

abcTolerance()

Returns
The abc tolerance: If the norm of certain pair IJ blocks in the auxiliary basis coefficients is below this threshold, the abc_tolerance, we will throw it away and reduce the overlap_sparsity.

Return type
float

energyCutoff()

Returns
The energy cutoff decides how many bands will be included in the Greens functions of the GW calculation. At the Gamma point the Kohn-Sham energy spectrum will be checked, and all bands that have an energy above the Fermi level that is below the energy cutoff will be included.

Return type
PhysicalQuantity of type energy.

groundStateCalculator()

Returns
The DFT calculator used to construct the zero order KS states.

Return type
LCAOCalculator

includeGammaPoint()

Returns
A bool, which if true, means we skip the gamma point and use no gamma point correction. This can be done if you want to perform an extrapolation to infinite kpoint sampling or if you want to verify that the gamma point correction is correct.

classmethod isADMMSupported(*is_hybrid*, *use_admm*)

Check for ADMM. The ADMM method cannot be combined with G0W0 for now.

classmethod isMGGASupported(*is_mgga*)

Check for ADMM. The ADMM method cannot be combined with G0W0 for now.

classmethod isPseudoPotentialTypeSupported(*is_paw*)

Check if the pseudo-potential, for now GW calculations can only be done with normconserving.

classmethod isSpinTypeSupported(*spin_type*)

Check if the spin type is supported.

nlinfo()

Returns
A dict with info about this calculator.

Return type
dict

numberOfOccupiedBands()

Returns
The number of bands below the fermi level in the DFT calculation. Set only after update has been called.

numericalAccuracyParameters()

Returns
The NumericalAccuracyParameters object used for the DFT calculation.

Return type
NumericalAccuracyParameters

polarizabilityCompressionParameter()

Returns
This parameter determines how aggresively the polarizability will be compressed, to save memory.

Return type
float

classmethod supportedCalculatorTypes()

Return a list of supported calculators.

Returns
List of supported calculators.

Return type
list

tauPrecisionParameter()

Returns
The parameter that decided how many time points will be taken. Given a (e_max : e_min) energy range, we have a minimal and maximal number of points we can include. If the precision parameters is 1 it will take the maximum, if it is 0 it will take the minimal. If in between we will take the closest integer that respects this fractional value.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Function that calls upgrade on the underlying ground_state_calculator object.

wCompressionParameter()

Returns
This parameter determines how aggresively the screened coulomb potential will be compressed, to save memory.

Return type
float

Usage Examples

To run a GW calculation one first has to construct a regular *LCAOCalculator*, which is then set on a *GWCalculator* object:

```
lcao_calculator = LCAOCalculator(
    basis_set=BasisGGAPseudoDojo.High,
    exchange_correlation=HybridGGA.HSE06)

gw_calculator = GWCalculator(
    lcao_calculator,
    energy_cutoff=50*eV,
    polarizability_compression_parameter=1.0e-07,
    w_compression_parameter=1.0e-05)

bulk_configuration.setCalculator(gw_calculator)
bulk_configuration.update()
```

Once the update function is called, the self energies are calculated and stored on the *GWCalculator* object. To get the bandstructure or density of states we just call the standard analysis objects:

```
bandstructure = Bandstructure(bulk_configuration, bands_above_fermi_level=4)
nlsave("gw_bandstructure.hdf5", bandstructure)

pdos= ProjectedDensityOfStates(
    bulk_configuration,
    MonkhorstPackGrid(21,21,21),
    projections=ProjectOnShellsBySite,
    bands_above_fermi_level=4)
nlsave("gw_pdos.hdf5", pdos)
```

For GW calculations it is important to have a good basis set. To get a basis set the matches plane-wave accuracy we have developed the *BasisSetOptimizer* class. For a reference system a plane wave calculation is performed and the LCAO basis set is optimized to match the plane wave calculation. The basis set is then stored in a file:

```
basis_set_optimizer = BasisSetOptimizer(
    bulk_configuration,
    orbitals_per_shell_map={Silicon: [6,6,5,4]},
    energy_cutoff=10*eV,
    tolerance=1.0e-5)

opt_basis_set = basis_set_optimizer.optimize()
nlsave("opt_basis_set.hdf5", opt_basis_set)
```

The bulk_configuration object that is passed into the *BasisSetOptimizer* object should have the updated plane-wave calculator attached.

The saved basis set can then be used in the GW calculation by setting it on the *LCAOCalculator* object:

```
opt_basis_set = nlread("opt_basis_set.hdf5")[0]
lcao_calculator = LCAOCalculator(
    basis_set=opt_basis_set,
    exchange_correlation=HybridGGA.HSE06)

gw_calculator = GWCalculator(
    lcao_calculator,
    energy_cutoff=50*eV,
    polarizability_compression_parameter=1.0e-07,
    w_compression_parameter=1.0e-05)

bulk_configuration.setCalculator(gw_calculator)
bulk_configuration.update()
```

More detailed information about the GW implementation can be found in the *G0W0 technical notes*.

GasSolubility

class GasSolubility(*gas_solutes*, *solvent_components*, *solvent_density=None*, *temperature=None*, *parameters=None*)

Calculate the solubility of gases using the COSMO-RS model.

Parameters

- **gas_solutes** (*CosmoRealSpecies* | Sequence of *CosmoRealSpecies*) – The gases for which the solubility is calculated.
- **solvent_components** (*CosmoRealSolvent* | *MoleculeConfiguration* | *CosmoRSMixture*) – The solvent in which the gas is dissolved in.
- **solvent_density** (*PhysicalQuantity* of type mass per volume) – The density of the solvent. If not specified solvent density is estimated from the components.
- **temperature** (*PhysicalQuantity* of type temperature) – The solvent temperature. **Default:** `298.15 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

gasSolutes()

Returns
The gases the solubility is calculated for.

Return type
Sequence of *CosmoRealSpecies*

henryConstantCP()

Return the Henrys constant of the given gas species in terms of concentration per pressure.

Returns
The Henry constant.

Return type
Sequence of *PhysicalQuantity* of mole per volume pressure

henryVolatilityPX()

Return the Henry volatility of the given gas species in terms of pressure per mole fraction.

Returns
The Henry volatility.

Return type
Sequence of *PhysicalQuantity* of mole fraction per pressure

parameters()

Returns
The COSMO-RS parameters

Return type
CosmoRSParameters

solventComponents()

Returns
The components of the solvent.

Return type
CosmoRSMixture

solventDensity()

Returns
The solvent density, if specified. If not returns None, and the solvent density is taken from the solvent components.

Return type
PhysicalQuantity of type mass per volume

temperature()

Returns
The solvent and gas temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Calculate the gas solubility of SO_2 in water.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
sulfur_dioxide = database.exportSpecies('sulfur dioxide')
sulfur_dioxide.setAntoineA(4.37798 * bar)
sulfur_dioxide.setAntoineB(966.6 * Kelvin)
sulfur_dioxide.setAntoineC(-42.07 * Kelvin)

# Determine the gas solubility.
gas_solubility = GasSolubility(
    sulfur_dioxide,
    water,
```

(continues on next page)

(continued from previous page)

```
    solvent_density=1*kiloGram/liter,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Retrieve the Henry constant.
h_cp = gas_solubility.henryConstantCP()[0].convertTo(molar/bar)
nlprint(f'The Henry constant of SO2 in water is {h_cp}')

# Retrieve the Henry volatility.
h_px = gas_solubility.henryVolatilityPX()[0].convertTo(bar)
nlprint(f'The Henry volatility of SO2 in water is {h_px}')
```

water_so2_example.py

Notes

The *GasSolubility* object allows one to estimate a measure of gas solubility. This measure is named Henrys constant and comes in a variety forms. Henrys law states that the amount of gas dissolved, c_{gas} is directly proportional to the partial pressure, p, of the gas above the condensed phase. The proportionality constant H^{cp} can be given as

$$H^{cp} = \frac{c_{gas}}{p}$$

The relation may also be defined as the partial pressure of a gas is proportional to its concentration. In this form the proportionality constant is referred to as the Henry volatility. Assuming concentration is given in mole fraction x_{gas}, then the proportionality constant K^{px} can be given as

$$K^{px} = \frac{p}{x_{gas}} = \frac{1}{V_m H^{cp}}$$

Here V_m is the molar volume of the solvent. Both measures can be calculated by the *GasSolubility* as demonstrated in the **Usage Example**.

In COSMO-RS, the Henrys constant is calculated by estimating the difference in chemical potential between the solvated and gaseous states. This is expressed as:

$$H^{cp} = \frac{\exp([\mu^{gas} - \mu^{solv}]/RT)}{p_0 V_m}$$

Here p_0 is a reference pressure, defined to be 1 bar. If the *Antoine* parameters are known for the gaseous species then the partial pressure, p, can be estimated as:

$$\log_{10}(p) = A - \frac{B}{C + T}$$

Here T is the temperature and A, B and C are the Antoine equation parameters. Since the partial pressure of a substance in COSMO-RS is defined as $p = \exp([\mu_{liq} - \mu_{gas}]/RT)$ this can be used to simplify the calculation of solubility to the difference in chemical potential between the solvated and pure liquid states. This can be expressed as:

$$H^{cp} = \frac{\exp([\mu^{liq} - \mu^{solv}]/RT)}{pV_m} = \frac{1}{\gamma p V_m}$$

Here γ is the activity coefficient of the substance in the solvent and p is the estimated gas partial pressure. The Antoine parameters are provided as demonstrated in the **Usage Example**. These parameters are then used in the semi-empirical Antoine equation which describes the relation between the vapor pressure of a pure substance and temperature.

GatePotentialAlignment

class GatePotentialAlignment(*reference_electrode=None*, *use_intrinsic_chemical_potential=None*, *work_function_difference=None*, *gate_work_function=None*, *reference_electrode_work_function=None*)

A class for representing the potential alignment of a gate electrode with respect to a reference electrode.

Parameters

- **reference_electrode** (NLFlag.Left | NLFlag.Right) – The reference electrode direction. **Default:** NLFlag.Left
- **use_intrinsic_chemical_potential** (*bool*) – If set to False, the chemical potential of the electrode specified in reference_electrode is used to determine work function differences. If True, the chemical potential of an equivalent intrinsic bulk configuration is used. **Default:** False
- **work_function_difference** – The difference between the work function of the gate electrode and the work function of the reference electrode. A zero difference will keep the gate and the electrode aligned. A positive difference will introduce or increase an energy barrier between the gate electrode and the reference electrode and a negative difference will reduce or remove an energy barrier between the gate electrode and the reference electrode. Alternatively, the gate electrode and the reference electrode work function can be directly specified instead of their difference the parameters reference_electrode_work_function and gate_work_function. **Default:** 0 * eV
- **gate_work_function** – The work function of the gate electrode, used to determine the difference with the reference electrode work function. The option is mutually exclusive with work_function_difference and if specified, also reference_electrode_work_function must be given.
- **reference_electrode_work_function** – The work function of the reference electrode, used to determine the difference with the gate work function. If use_intrinsic_chemical_potential is set to True, the work function of an intrinsic equivalent bulk configuration should be set. The option is mutually exclusive with work_function_difference and if specified, also gate_work_function must be given.

referenceElectrode()

Returns
The electrode used to set the reference energy.

Return type
NLFlag.Left | NLFlag.Right

uniqueString()

Return a unique string representing the state of the object.

useIntrinsicChemicalPotential()

Returns
Wheter the chemical potential of a bulk configuration equivalent to the electrode but intrinsic is used.

Return type
bool

workFunctionDifference()

Returns
The difference between the work function of the gate electrode and the work function of the reference electrode.

Return type
PhysicalQuantity of type energy.

GaussPotential

class GaussPotential(*particleType1*, *particleType2*, *b1*, *b2*, *b3*, *c1*, *c2*, *c3*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **b1** (*PhysicalQuantity of type length**-2*) – Potential parameter.
- **b2** (*PhysicalQuantity of type length**-2*) – Potential parameter.
- **b3** (*PhysicalQuantity of type length**-2*) – Potential parameter.
- **c1** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c2** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c3** (*PhysicalQuantity of type energy*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *GaussPotential* defines a pairwise potential of the form

$$V(r) = c_1 e^{-b_1 r^2} + c_2 e^{-b_2 r^2} + c_3 e^{-b_3 r^2} \ .$$

GaussianAnglePotential

class GaussianAnglePotential(*particleType1*, *particleType2*, *particleType3*, *K*, *sigma*, *theta0*, *d*, *r_cut*, *N=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type (central particle when calculating angles)
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **K** (*PhysicalQuantity of type energy*) – Potential parameter.
- **sigma** (*PhysicalQuantity of type angle**-2*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter.
- **d** (*PhysicalQuantity of type length**2*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius
- **N** (*int*) – Potential parameter. Must not be 0.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The Gaussian angle potential has the form

$$U_{ijk} = -K_{ijk} \exp\left(-\sigma_{ijk}(\theta_{ijk} - \theta_{ijk}^{(0)})^2\right) \begin{cases} \exp\left(-\frac{d_{ijk}}{(\bar{r}_{ijk} - r_{ijk}^{cut})^2}\right), & \bar{r}_{ijk} \leq r_{ijk}^{cut} \\ 0, & \bar{r}_{ijk} > r_{ijk}^{cut} \end{cases}$$

where

$$\bar{r}_{ijk} = (r_{ij}^{N_{ijk}} + r_{jk}^{N_{ijk}})^{1/N_{ijk}}.$$

GaussianProcessNEBParameters

class GaussianProcessNEBParameters(*kernel=None*, *length_scale=None*, *magnitude=None*, *noise_variance=None*, *max_displacement=None*, *convergence_factor=None*, *max_iterations=None*, *training_interval=None*, *trajectory_filename=None*, *history=None*, *min_bond_fuzz_factor=None*)

Class that stores all the parameters associated with optimizing a NudgedElasticBand using a Gaussian process regression surrogate model.

Parameters

•**kernel** (*InverseAtomDistance | RadialBasisFunction | Matern52*) – The kernel to use in the Gaussian process regression. **Default:** Matern52.

•**length_scale** (*PhysicalQuantity* of type length) – The kernel length scale hyperparameter. **Default:** 1.0 Angstrom.

•**magnitude** (*PhysicalQuantity* of type energy) – The covariance functions magnitude hyperparameter. **Default:** 1.0 eV.

•**noise_variance** (*float*) – The Gaussian process models noise variance. Usually a small value to avoid numerical problems. **Default:** 1e-10.

•**max_displacement** (*PhysicalQuantity* of type length) – The maximum displacement of the surrogate path from the last known path before the next reference points are calculated. If set to *Automatic*, the value is calculated as half of the total length of the last known path. **Default:** Automatic.

•**convergence_factor** (*float*) – The convergence factor used to determine if the surrogate optimization has converged. This is relative to the *max_forces* criterion of the true path. It is recommended to choose this value to be smaller than 1.0. **Default:** 0.5.

•**max_iterations** (*int*) – The maximum number of iterations of the surrogate optimization. **Default:** 20.

•**training_interval** (*int*) – The number of iterations between optimizing the Gaussian Process model hyperparameters. A value of 0 means that the hyperparameters are never optimized. **Default:** 0.

•**trajectory_filename** (*str*) – The filename to save the trajectory of the surrogate optimization. None means that the trajectory is not saved. **Default:** None.

•**history** (*non-negative int*) – The number of most recent configurations to use in training and fitting the model. If 0, all configurations are used. The smaller the number, the faster the training and fitting, but the less accurate the model. **Default:** 0

•**min_bond_fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine if two atoms are too close to each other. This is used to determine if a configuration is unphysical. A higher value corresponds to a more strict criterion and more configurations classified as unphysical, while a lower value corresponds to a more relaxed criterion. If the factor is 0, the check for unphysical structures is not done. **Default:** 0.7

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

GaussianSmearing

class GaussianSmearing(*broadening*)

Parameters
broadening (*PhysicalQuantity* of type energy or temperature) – The broadening of the distribution.

broadening()

Returns
The broadening.

Return type
PhysicalQuantity of type energy

Usage Examples

Use the Gaussian smearing occupation function with a broadening of 0.1 eV on an LCAOCalculator:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    occupation_method=GaussianSmearing(0.1*eV))

calculator = LCAOCalculator(numerical_accuracy_parameters=numerical_accuracy_parameters)
```

Notes

Note: For comparison of different occupation methods and suggestions for which one to choose, see *Occupation Methods*.

In the Gaussian smearing scheme[1] one replaces the delta function in the density of states by a Gaussian distribution:

$$\tilde{\delta}(x) = \frac{1}{\sigma\sqrt{\pi}} e^{-(x/\sigma)^2},$$

where σ is the broadening. This means that the integer occupation numbers are replaced by fractional occupations given by the distribution

$$f(\epsilon) = \frac{1}{2}\left[1 - \mathrm{erf}\left(\frac{\epsilon - \mu}{\sigma}\right)\right],$$

where ϵ is the energy of the state and μ is the Fermi level.

In the Gaussian smearing scheme the generalized entropy is given by

$$S = \sum_i \frac{1}{2\sqrt{\pi}} \exp\left(-\left(\frac{\epsilon_i - \mu}{\sigma}\right)^2\right)$$

The total energy at zero broadening can be estimated by adding the correction given by

$$\Delta E_{\sigma\to 0}(\sigma) = -\frac{1}{2}\sigma S(\sigma)$$

[1] C.ă-L. Fu and K.ă-M. Ho. First-principles calculation of the equilibrium ground-state properties of transition metals: Applications to Nb and Mo. *Phys. Rev. B*, 28(10):5480–5486, November 1983. doi:10.1103/PhysRevB.28.5480.

General1Potential

class General1Potential(*particleType1*, *particleType2*, *A*, *rho*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type length * energy*) – Potential parameter.
- **rho** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *General1Potential* defines a pair potential of the form

$$V_{ij}(r) = \frac{A}{r} e^{-r/\rho} .$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

General2Potential

class General2Potential(*particleType1*, *particleType2*, *A*, *C*, *rho*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type length**2 * energy*) – Potential parameter.
- **C** (*PhysicalQuantity of type length * energy*) – Potential parameter.
- **rho** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *General2Potential* defines a pair potential of the form

$$V_{ij}(r) = \frac{A}{r^2} e^{-r/\rho} - \frac{C}{r} .$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$

•In the range $[0, 1]$ when $r_i < r < r_{cut}$

•0 when $r \geq r_{cut}$

General3Potential

class General3Potential(*particleType1*, *particleType2*, *A*, *B*, *sigma*, *r_i=None*, *r_cut=None*)
Constructor of the potential.

Parameters

•**particleType1** (*ParticleType or ParticleIdentifier*) – Identifier of the first particle type.

•**particleType2** (*ParticleType or ParticleIdentifier*) – Identifier of the second particle type.

•**A** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**sigma** (*PhysicalQuantity of type length*) – Potential parameter.

•**r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()
Return the names of all used parameters as a list.

getAllParameters()
Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()
Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)
Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)
Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

The *General3Potential* defines a pair potential of the form

$$V_{ij}(r) = Ae^{B(\sigma - r)}\,.$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

GeneralStiwe2Potential

class GeneralStiwe2Potential(*particleType1*, *particleType2*, *p*, *A*, *B*, *gamma*, *q*, *D*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **p** (*float*) – Potential parameter.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type length**p*) – Potential parameter.
- **gamma** (*PhysicalQuantity of type length*) – Potential parameter.
- **q** (*float*) – Potential parameter.
- **D** (*PhysicalQuantity of type length**q*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The *GeneralStiwe2Potential* represents the a more general form for the two-body contribution of the Stillinger-Weber potential[1].

The Stillinger-Weber potential can be written as:

$$V = \sum_{i<j} v_2^{ij}(r_{ij}) + \sum_{i<j<k} \left[v_3^{ijk}(r_{ji}, r_{jk}, \theta_{ijk}) + v_3^{jik}(r_{ij}, r_{ik}, \theta_{jik}) + v_3^{ikj}(r_{ki}, r_{kj}, \theta_{ikj}) \right].$$

The general form of the two-body part is implemented as

$$v_2^{ij}(r) = A \cdot (B \cdot r^{-p} - D \cdot r^{-q}) \cdot \exp\left[\frac{\gamma}{r - r_{\mathrm{cut}}} \right].$$

For the three-body part two types are available, which can be selected by the `type` argument.

Type 1:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1} \right] \cdot (\cos(\theta) - cos(\theta_0))^{\alpha}.$$

If not specified otherwise, α is set to 2, as in the original Stillinger-Weber potential[Page 2957, 1].

Type 2:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1} \right] \cdot (\cos(\theta) - cos(\theta_0)) \sin(\theta) \cos(\theta).$$

GeneralizedDavidsonSolver

class GeneralizedDavidsonSolver(*absolute_tolerance=None*, *relative_tolerance=None*, *maximum_number_of_restarts=None*, *initialization_method=None*)

Iterative eigensolver that uses a blocked generalized Davidson algorithm to improve on an initial guess for the eigenstates of the Hamiltonian.

Parameters

- **absolute_tolerance** (*float*) – The absolute tolerance up to which the residuals have to be converged for the algorithm to end.
- **relative_tolerance** (*float*) – The amount the residual of the states has to be reduced compared to the residual of the initial guess. The reduction factor must be between 0.0 and 1.0. **Default:** `0.3`
- **maximum_number_of_restarts** (*int*) – The maximum number of restarts in the Davidson algorithm. Must be a positive integer. **Default:** `0`
- **initialization_method** (*RandomBlochWaveInitialization* | *BasisSetInitialization*) – The method used to obtain the initial states for the Generalized Davidson algorithm, see *Initialization methods*. **Default:** `BasisSetInitialization()`

absoluteTolerance()

Returns
The absolute tolerance up to which the residuals have to be converged for the algorithm to end.

Return type
float

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

initializationMethod()

Returns
The method used to obtain the initial states for the residual minimizer.

Return type
RandomBlochWaveInitialization | BasisSetInitialization

maximumNumberOfRestarts()

Returns
The maximum number of restarts in the Davidson algorithm.

Return type
int

relativeTolerance()

Returns
The amount the residual of the states has to be reduced compared to the residual of the initial guess.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Define a density matrix method that uses the default *GeneralizedDavidsonSolver*:

```
algorithm_parameters = AlgorithmParameters(
   density_matrix_method=GeneralizedDavidsonSolver())
```

A more robust eigensolver can be defined by tightening the tolerances and increasing the maximum_number_of_restarts.

```
density_matrix_method = GeneralizedDavidsonSolver(
    relative_tolerance=0.001,
    absolute_tolerance=1.0e-7,
    maximum_number_of_restarts=4)

algorithm_parameters = AlgorithmParameters(
   density_matrix_method=density_matrix_method)
```

A different initial guess for the states can be chosen by setting the initialization_method:

```
density_matrix_method = GeneralizedDavidsonSolver(
    initialization_method=RandomBlochWaveInitialization(),
)

algorithm_parameters = AlgorithmParameters(
    density_matrix_method=density_matrix_method)
```

Notes

Algorithm

The class implements the blocked generalized Davidson algorithm[1],[2]. It is the default eigensolver for plane wave calculations.

One can increase the stability of the algorithm by lowering the `relative_factor`, which determines how much the residual of the current guess for the eigenstates has to be reduced in this SCF step. One can also use the `absolute_tolerance` keyword, which determines how small the norm of the residuals of the wave functions need to be. At the same time one should increase the `maximum_number_of_restarts`, which determines how many restarts are allowed. When a restart is performed, a new generalized Davidson optimization is started with the current best guess for the eigenstates as starting vectors.

GilbertDamping

class GilbertDamping(*configuration*, *kpoints=None*, *broadenings=None*)

Class for calculating the Gilbert damping for a BulkConfiguration.

Parameters

- **configuration** (*BulkConfiguration*) – The BulkConfiguration for which to calculate the inter-site coupling matrix.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points with which to calculate the Gilbert damping. **Default:** The k-point sampling from the calculator.
- **broadenings** (*PhysicalQuantity* of type energy.) – Energy level broadening due finite life time. **Default:** `numpy.linspace(0.01, 0.1, 11)*eV`

alpha()

Returns
Dimensionless Gilbert damping tensor

Return type
numpy.array of shape (3,3).

broadenings()

Returns
Energy level broadening due finite life time.

Return type
PhysicalQuantity of type energy.

dampingConstant()

Returns
Dimensionless Gilbert damping constant

Return type
float

[1] ErnestăR. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics*, 17(1):87–94, 1975. doi:http://dx.doi.org/10.1016/0021-9991(75)90065-0.

[2] RonaldăB. Morgan and DavidăS. Scott. Generalizations of Davidsons Method for Computing Eigenvalues of Sparse Symmetric Matrices. *SIAM Journal on Scientific and Statistical Computing*, 7(3):817–825, 1986. doi:10.1137/0907054.

dampingRate()

Returns
Gilbert damping constant

Return type
PhysicalQuantity of type inverse time

dampingRateTensor()

Returns
Gilbert damping tentor

Return type
PhysicalQuantity of type inverse time

kpoints()

Returns
The k-points with which to calculate the Heisenberg exchange constants.

Return type
class:~.*MonkhorstPackGrid* | *RegularKpointGrid*

magneticMoment()

Returns
The total magnetic moment.

Return type
PhysicalQuantity of type Bohr magneton

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *GilbertDamping* for iron.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = BodyCenteredCubic(2.8665*Angstrom)

# Define elements
elements = [Iron]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
    BasisGGAPseudoDojoSO.Iron_Medium,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = SOGGA.PBE

k_point_sampling = MonkhorstPackGrid(
    na=13,
    nb=13,
    nc=13,
    force_timereversal=False,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=120.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
```

(continues on next page)

(continued from previous page)

```
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('gilbert_damping_iron.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Gilbert Damping
# ------------------------------------------------------------
kpoint_grid = MonkhorstPackGrid(
    na=38,
    nb=38,
    nc=38,
    force_timereversal=False,
    )

gilbert_damping = GilbertDamping(
    configuration=bulk_configuration,
    kpoints=kpoint_grid,
    broadenings=numpy.linspace(0.01, 0.1, 11)*eV,
)
nlsave('gilbert_damping_iron.hdf5', gilbert_damping)
```

gilbert_damping.py

Notes

The unitless Gilbert damping tensor is calculated using Kamberskys torque-torque correlation model following refs.[1][2][3] :

$$\alpha_{\mu,\nu} = \frac{g}{m\pi} \sum_k w_k \sum_{ij} W_{ij}(k) T^{\nu}_{ij}(k) \left(T^{\mu}_{ij}(k)\right)^{\dagger}$$

where $g = 2.0023$ is the g-factor, m is the magnetization (has unit of Bohr magnetons). The sum over i, j runs over the band indices and the k sum is over the Brillouin zone with w_k being the k-point weight. The matrix elements of the torque operator are

$$T^{\nu}_{ij}(k) = \langle \psi_{ik} | \left[\sigma^{\nu}, H_{SO}\right] | \psi_{jk} \rangle$$

where σ^{ν} is a Pauli spin matrix, H_{SO} is the spin-orbit contribution to the Hamiltonian and ψ_{ik} is the Bloch eigenstate of band n evaluated at k-point k. The spectral overlap function is

$$W_{ij}(k) = \int \left(-\frac{\partial f(\epsilon, \epsilon_F)}{\partial \epsilon}\right) A_{ik}(\epsilon, \Lambda) A_{jk}(\epsilon, \Lambda) d\epsilon$$

[1] K.ăGilmore, Y.ăU. Idzerda, and M.ăD. Stiles. Identification of the Dominant Precession-Damping Mechanism in Fe, Co, and Ni by First-Principles Calculations. *Phys. Rev. Lett.*, 99:027204, Jul 2007. URL: https://link.aps.org/doi/10.1103/PhysRevLett.99.027204, doi:10.1103/PhysRevLett.99.027204.

[2] DăThonig and JăHenk. Gilbert damping tensor within the breathing Fermi surface model: anisotropy and non-locality. *New Journal of Physics*, 16(1):013032, jan 2014. doi:10.1088/1367-2630/16/1/013032.

[3] Danny Thonig, Yaroslav Kvashnin, Olle Eriksson, and Manuel Pereiro. Nonlocal Gilbert damping tensor within the torque-torque correlation model. *Phys. Rev. Materials*, 2:013801, Jan 2018. URL: https://link.aps.org/doi/10.1103/PhysRevMaterials.2.013801, doi:10.1103/PhysRevMaterials.2.013801.

where $f(\epsilon, \epsilon_F)$ is the Fermi-Dirac distribution function (ϵ_F is the Fermi energy), and $A_{ik}(\epsilon, \Lambda) = \frac{1}{\pi} \frac{\Lambda}{(\epsilon_{ik}-\epsilon)^2+\Lambda^2}$ is a Lorentzian spectral function with broadening Λ and centered around the band energy ϵ_{ik}.

Assuming that the spectral life-time broadening Λ is due to electron-phonon coupling, it is reasonable to assume that $\Lambda > k_B T$ in which case we can approximate the derivative of the Fermi-Dirac distribution can be approximated by a δ-function resulting in the simplified expression

$$W_{ij}(k) \approx A_{ik}(\epsilon_F, \Lambda) A_{jk}(\epsilon_F, \Lambda)$$

which is implemented in QuantumATK.

The unitless damping tensor can be obtained as

```
# Unitless damping tensor.
alpha_tensor = gilbert_damping.alpha()
```

When the magnetic moment is oriented along the z-direction as will usually be the case, the quantity of interest will often be calculated from the torque operator $\langle \psi_{ik} | [\sigma^-, H_{SO}] | \psi_{jk} \rangle$, where $\sigma^- = \sigma^x - i\sigma^y$. This corresponds to the sum of the tensor elements $\alpha = \alpha_{xx} + \alpha_{yy}$ This number is obtained for each value of the broadening as

```
# Sum of xx and yy components of the dimensionless damping tensor.
alpha = gilbert_damping.dampingConstant()
```

The relation between the unitless damping tensor and the physical damping rate in units of inverse time is:

$$\lambda = \gamma M \mu_0 \alpha$$

where the gyro-magnetic ratio is $\gamma = g\mu_B/\hbar$ and the magnetization $M = \frac{m}{V}$ is the magnetic moment divided by unit cell volume. μ_0 is the vacuum permeability and μ_B is the Bohr magneton.

The damping rate in units of inverse time is obtained as

```
# Damping rate tensor in units of inverse time
damping_rate_tensor = gilbert_damping.dampingRateTensor()

# Sum of xx and yy components of the damping rate tensor.
damping_rate = gilbert_damping.dampingRate()
```

The life-time broadening, Λ

The life-time broadening Λ can in principle have several origins such as electron-phonon scattering, impurity scattering or alloy scattering. It is often assumed that the electron-phonon coupling is the dominant scattering mechanism, at least at temperatures $T \gg 0K$. In QuantumATK it is possible to calculate the scattering rate, or inverse life time, due to electron phonon coupling. See the *ElectronPhononCoupling* object for more information.

GrainBoundaryGenerator

class GrainBoundaryGenerator(*rotation_axis*, *rotation_angle*, *sigma*, *boundary_plane*, *minimum_electrode_length=None*, *buffer_layer=None*, *overlap_distance=None*, *reduce_by_symmetry=True*)

Template for generating grain boundaries.

Parameters

- **rotation_axis** (*array of ints*) – The rotation axis to use.

•**rotation_angle** (*PhysicalQuantity* of type angle) – The rotation angle to use.

•**sigma** (*int*) – The sigma to use.

•**boundary_plane** (*array of ints*) – The boundary plane.

•**minimum_electrode_length** (*PhysicalQuantity* of type length) – The minimum length of the electrode extension. **Default:** `0.0*Angstrom`.

•**buffer_layer** (*PhysicalQuantity* of type length) – The minimum distance from the grain boundary to the electrode extension. **Default:** `0.0*Angstrom`.

•**overlap_distance** (*PhysicalQuantity* of type length) – The distance where atoms should be considered overlapping. **Default:** `0.0*Angstrom`.

•**reduce_by_symmetry** (*bool*) – Reduce the size using symmetries. If False, the cross section will always be rectangular. **Default:** True.

boundaryPlane()

Returns
The boundary plane.

Return type
array of ints

bufferLayer()

Returns
The minimum distance from the grain boundary to the electrode extension.

Return type
PhysicalQuantity of type length

bulkConfiguration(*bulk_configuration*, *repeats=None*, *center_only=True*, *origin_index=None*)

Generate grain boundary configuration.

Parameters

•**bulk_configuration** (BulkConfiguration) – The pristine structure to use.

•**repeats** (*array of ints*) – How to repeat the grain boundary. **Default:** `[1, 1, 1]`.

•**center_only** (*bool*) – True, if atoms should only be removed at the central grain boundary. **Default:** True.

•**origin_index** (*int*) – The atom to be used as the coincident site when creating the grain boundary.

Returns
The generated configuration with the grain boundary.

Return type
BulkConfiguration

deviceConfiguration(*bulk_configuration*, *origin_index=None*)

Generate device configuration with grain boundary.

Parameters

•**bulk_configuration** (BulkConfiguration) – The pristine structure to use.

•**origin_index** (*int*) – The atom to be used as the coincident site when creating the grain boundary.

Returns
The generated device.

Return type
DeviceConfiguration

minimumElectrodeLength()

Returns
The minimum length of the electrode extension.

Return type
PhysicalQuantity of type length

overlapDistance()

Returns
The distance where atoms should be considered overlapping.

Return type
PhysicalQuantity of type length

reduceBySymmetry()

Returns
Boolean determining if the structure should be reduced in size using symmetries.

Return type
bool

rotationAngle()

Returns
The rotation angle to use.

Return type
PhysicalQuantity of type angle

rotationAxis()

Returns
The rotation axis to use.

Return type
array of ints

sigma()

Returns
The sigma to use.

Return type
int

static sigmaOptions(*rotation_axis*, *max_sigma=99*)
Get the sigma and rotation angle options for a specified rotation axis.

Parameters

- **rotation_axis** (*array of ints*) – The rotation axis to use.
- **max_sigma** (*int*) – The upper limit of sigma to return.

Returns
The supported sigmas and angles.

Return type
list of tuples

`title()`

Returns
The title representation of the grain boundary generator.

Return type
str

`uniqueString()`

Return a unique string representing the state of the object.

Usage Example

Generate three examples using tausonite as the input configuration:

•configuration_1: A grain boundary in a BulkConfiguration, with strontium as the origin_index atom,

•configuration_2 :A grain boundary in a BulkConfiguration, with titanium as the origin_index atom.

•configuration_3 :A grain boundary in a DeviceConfiguration, with strontium as the origin_index atom.

```
# Set up a Tausonite configuration.
fractional_coordinates = [[ 0.5,  0.5,  0.5],
                          [ 0. ,  0. ,  0. ],
                          [ 0.5,  0. ,  0. ],
                          [ 0. ,  0.5,  0. ],
                          [ 0. ,  0. ,  0.5]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=SimpleCubic(3.90528 * Angstrom),
    elements=[Strontium, Titanium, Oxygen, Oxygen, Oxygen],
    fractional_coordinates=fractional_coordinates
)

# Instantiate a generator for BulkConfiguration.
generator = GrainBoundaryGenerator(
    rotation_axis=[0, 0, 1],
    rotation_angle=36.8698976458 * Degrees,
    sigma=5,
    boundary_plane=[-2, -1, 0],
    overlap_distance=0.5 * Angstrom,
    reduce_by_symmetry=True,
)

# Generate a grain boundary in a BulkConfiguration, with strontium as the origin_index␣
↪atom.
configuration_1 = generator.bulkConfiguration(
    bulk_configuration,
    repeats=[1, 1, 1],
    origin_index=0
)
```

(continues on next page)

(continued from previous page)

```
# Generate a grain boundary in a BulkConfiguration, with titanium as the origin_index␣
↪atom.
configuration_2 = generator.bulkConfiguration(
    bulk_configuration,
    repeats=[1, 1, 1],
    origin_index=1
)

# Instantiate a generator for DeviceConfiguration.
generator = GrainBoundaryGenerator(
    rotation_axis=[0, 0, 1],
    rotation_angle=36.8698976458 * Degrees,
    sigma=5,
    boundary_plane=[-2, -1, 0],
    minimum_electrode_length=5.0 * Angstrom,
    buffer_layer=5.0 * Angstrom,
    overlap_distance=0.5 * Angstrom,
    reduce_by_symmetry=True,
)

# Generate a grain boundary in a DeviceConfiguration, with strontium as the origin_index␣
↪atom.
configuration_3 = generator.deviceConfiguration(
    bulk_configuration,
    origin_index=0
)
```

grain_boundary_generator.py

GreensFunction

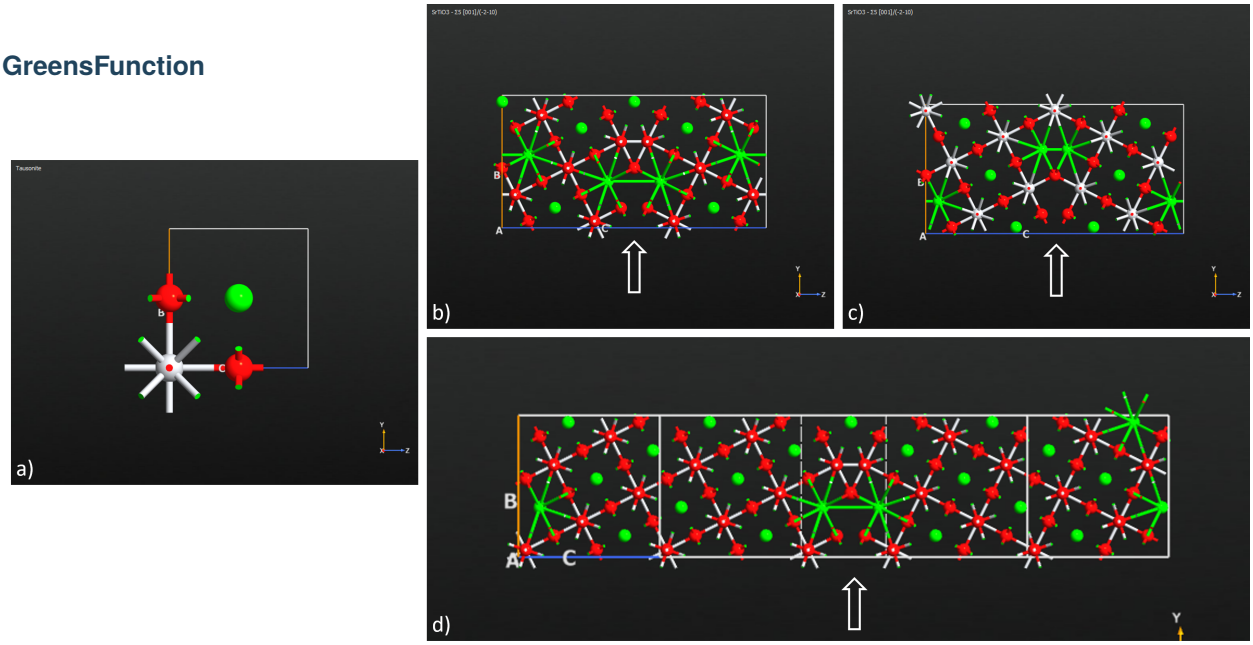


Fig. 4.22: Atom color legend: green - strontium, grey - titanium, red - oxygen. a) The input configuration, b) configuration_1, c) configuration_2, and d) configuration_3. The arrows indicate the grain boundary in figures b)-d).

class Greens

The block tridiagonal inversion method used to calculate the

Greens function and lesser

Greens function.

The method distributes the evaluation of the (lesser) Greens function for multiple contour points, i.e., energy-points and k-points, over all available processes.

The evaluation of the (lesser) Greens function in each contour

point, can be performed by multiple processes in parallel.

Parameters
processes_per_contour_point (*int*) – The number of processes used to calculate each contour point. Must be positive. The best performance is obtained when set to 1. Increasing the number of processes per contour point reduces the required memory. **Default:** 1

processesPerContourPoint()

Returns
The processes per contour point

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Setup a device calculation that computes the Greens function and lesser Greens function via block tridiagonal inversion using 2 MPI processes for each contour point.

```
greens_function_method = GreensFunction(
   processes_per_contour_point=2)
   device_algorithm_parameters = DeviceAlgorithmParameters(
       equilibrium_method = greens_function_method,
       non_equilibrium_method = greens_function_method,
       )

calculator = DeviceHuckelCalculator(
    device_algorithm_parameters = device_algorithm_parameters,
    )
```

Notes

•The method implements the $\mathcal{O}(N)$ block tridiagonal inversion algorithm[1].

•GPU acceleration is supported, see *GPU acceleration of proprietary feature calculations*.

GridData

class GridData(*values*, *axes*, *display=None*)

GridData contains a grid of arbitrary data spanned over a number of axes.

The simplest case is a one-dimensional grid data of scalar quantities:

```
GridData(values=[1, 2, 3, 4], axes=[0.1, 0.2, 0.3, 0.4])
```

It also supports *PhysicalQuantity* as data and it can handle multiple dimensions:

```
GridData(
    values=[[1, 2, 3], [4, 5, 6]] * eV,
    axes=[
        [0.1, 0.2] * Angstrom,
        [0.4, 0.5, 0.6] * Degrees
    ]
)
```

The values and the axes can also be given names to easily distinguish them:

```
GridData(
    values=('Energy', [[1, 2, 3], [4, 5, 6]] * eV),
    axes=[
        ('Distance', [0.1, 0.2] * Angstrom),
        ('Angles', [0.4, 0.5, 0.6] * Degrees)
    ]
)
```

The names can be used for visualization as well as grid operations, see the `grid()` method for additional information.

A default visualization for the grid data can be specified through the optional display option, see the `setDisplay()` method.

Parameters

•**values** (*sequence of named values*) – The (optionally named) grid values at each data point.

•**axes** (*sequence of named axes*) – The (optionally named) axes.

•**display** (*dict*) – Optional default `display` options.

axes()

The grid axes.

Returns
The grid axes.

Return type
tuple of *PhysicalQuantity*

[1] D.ăE. Petersen, H.ăH.ăB. Sorensen, P.ăC. Hansen, S.ăSkelboe, and K.ăStokbro. Block Tridiagonal Matrix Inversion and Fast Transmission Calculations. *J. Comput. Phys.*, 227(6):3174–3190, 2008. doi:10.1016/j.jcp.2007.11.035.

axesNames()

The axes names.

Returns
The axes names.

Return type
tuple of str

axis(*name=None*)

Retrieve the named axis.

Parameters
name (*str | NoneType*) – The name of the axis to retrieve.

Returns
The named axis.

Return type
PhysicalQuantity

display()

The default display option.

Returns
The default display options.

Return type
dict | NoneType

classmethod fromPoints(*values*, *points*, *display=None*)

Create GridData from points.

Parameters

- **values** (*sequence of values*) – The values.
- **points** (*sequence of sequence of axis values*) – The points.
- **display** (*NoneType | dict*) – The optional default display options.

Returns
The created GridData.

Return type
GridData

classmethod fromSequence(*values*, *axes*, *display=None*)

Create GridData from sequences of values by using the axes to reshape it.

Parameters

- **values** (*sequence of named values*) – The (optionally named) grid values at each data point.
- **axes** (*sequence of named axes*) – The (optionally named) axes.
- **display** (*NoneType | dict*) – The optional default display options.

Returns
The created GridData.

Return type
GridData

grid(*names=None*, *averages=None*, *sums=None*)

Create a reduced grid by performing averages or sums along several axes.

Averaging over an axis named `Angle` can then be done by calling:

```
averaged_data = data.grid(averages=('Angle'))
```

Or similarly, summing over the axes `Distance` and `Angle`:

```
summed_data = data.grid(sums=('Distance', 'Angle'))
```

Parameters

- **names** (*tuple of str*) – The named values to include.
- **averages** (*tuple of str*) – The names of axes to average over.
- **sums** (*tuple of str*) – The names of axes to sum over.

Returns
The reduced grid.

Return type
GridData

names()

The names of the values.

Returns
The names of the values.

Return type
tuple of str

nlinfo()

Returns
Structured information about the GridData.

Return type
dict

nlprint(*stream*)

Print a string containing an ASCII description of the GridData.

Parameters
stream (*file-like*) – The stream to write to. This should be an object that supports strings being written to it using a `write` method.

setDisplay(*value=None*, *x=None*, *y=None*, *averages=None*, *sums=None*)

Set the default display options.

In case of a grid with two axes, the display could be a line, averaged over one of them:

```
data.setDisplay(x='NameAxis1', averages=('NameAxis2'))
```

See the `grid()` method for additional information.

Parameters

- **value** (*str*) – The name of the value to display.

- **x** (*str*) – The name of the axis to consider as x axis.
- **y** (*str*) – The name of the axis to consider as y axis.
- **averages** (*tuple of str*) – The names of the axes to average over.
- **sums** (*tuple of str*) – The names of the axes to sum over.

uniqueString()
Return a unique string representing the state of the object.

value(*name=None*)
Retrieve the named value.

Parameters
name (*str | NoneType*) – The name of the value to retrieve.

Returns
The named value.

Return type
PhysicalQuantity

values()
The values.

Returns
The values.

Return type
tuple of *PhysicalQuantity*

Usage Examples

Here is a short overview over the different ways to use the GridData class. The simplest case is a one-dimensional grid data:

```
GridData(values=[1, 2, 3, 4], axes=[0.1, 0.2, 0.3, 0.4])
```

It also supports *PhysicalQuantity* and the `fromSequence()` constructor can reshape the `values` array automatically based on the provided axes:

```
GridData.fromSequence(
    values=[1, 2, 3, 4, 5, 6],
    axes=[
        [0.1, 0.2] * Angstrom,
        [0.4, 0.5, 0.6] * Degrees
    ]
)
```

The grid can scale up to multiple values and dimension, names can be used to easily identify each of them:

```
GridData(
    values=[
        ('Energy', [[1, 2, 3], [4, 5, 6]] * eV),
        ('Temperature', [[200, 300, 400], [250, 350, 450]] * Kelvin)
    ],
    axes=[
```

(continues on next page)

(continued from previous page)

```
        ('Distance', [0.1, 0.2] * Angstrom),
        ('Angle', [0.4, 0.5, 0.6] * Degrees)
    ]
)
```

The names can be used for visualization as well as grid operations. Averaging over all angles can then be done by calling:

```
averaged_data = data.grid(averages=('Angle'))
```

Or similarly, summing over all distances,

```
summed_data = data.grid(sums=('Distance'))
```

Quick example - Total energy of diatomic molecules

In the following ATKPython example script, we compute the total energy of several diatomic molecules, for different atomic numbers and bond lengths. They can then be saved as values on a two-dimensional grid.

```
atomic_numbers_dict = {
    Hydrogen: 1,
    Nitrogen: 7,
    Oxygen: 8,
    Fluorine: 9,
    Chlorine: 17
}

bond_lenghts = [0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0]

total_energies = []
atomic_numbers = []

for element in atomic_numbers_dict:
    molecule = MoleculeConfiguration(
        elements=[element, element],
        cartesian_coordinates=[[0, 0, 0], [1, 0, 0]] * Angstrom,
        padding_length=5 * Angstrom
    )
    molecule.setCalculator(LCAOCalculator())
    atomic_numbers.append(atomic_numbers_dict[element])

    for bond_lenght in bond_lenghts:
        molecule.setCartesianCoordinates(
            [[bond_lenght, 0, 0]] * Angstrom,
            indices=[1]
        )
        total_energy = TotalEnergy(molecule)
        total_energies.append(total_energy.evaluate())

grid = GridData.fromSequence(
        values=('Total energy', total_energies),
```

(continues on next page)

(continued from previous page)

```
        axes=[
            ('Atomic number', atomic_numbers),
            ('Bond lenght', bond_lenghts),
        ]
    )
```

grid_data.py

Realistic example - Sputtering yield of a surface process

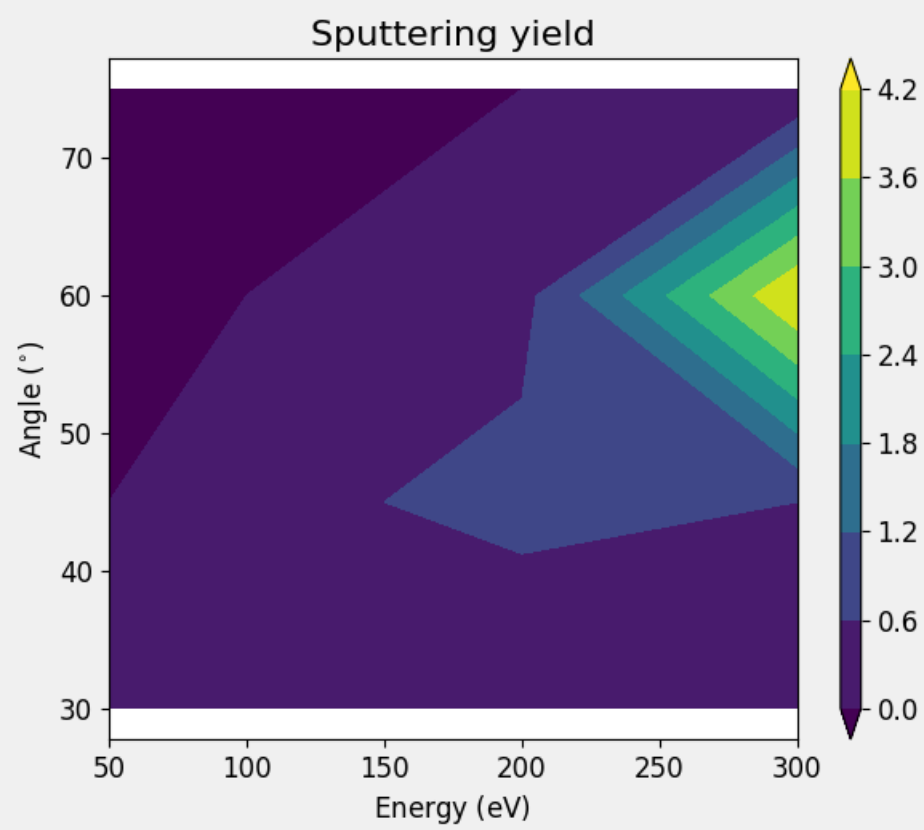
A more realistic example could be a study of the sputtering yield using *SurfaceProcessSimulation*. However this script will take significantly longer to execute, compared to the previous one. Also, the purpose of these scripts is only to document the GridData class, not to show solutions to real case studies.

This script simulates the shooting of an argon atom to a silicon surface, with different kinetic energies and incident angles; finally, it repeats each event 5 times. The following code snippet contains the section of the script that creates the GridData object, the entire script is available for download below.

```
sputtering_yield = [item.sputteringYield() for item in iteration_items]

grid_data = GridData.fromSequence(
    values=(
        ('Sputtering yield', sputtering_yield)
    ),
    axes=(
        ('Energy', energies),
        ('Angle', angles),
        ('Repeat', repeats)
    ),
    display={
        'x': 'Energy',
        'y': 'Angle',
        'averages': ('Repeat',)
    }
)
```

multi_sps.py

The sputtering yield is saved on a grid, with the incident angles and energies on two of the axes, the last axis is defined by the repeats and is averaged over by the default display option. The data can be visualized with a contour plot.



GridSampling

class GridSampling(*na*, *nb=None*, *nc=None*)

Class for representing the grid sampling for real space grids.

The constructor takes the number of grid points in the three unitcell directions.

Parameters

- **na** (*int*) – Number of grid points to use along the primitive direction a. Must be larger than 3.
- **nb** (*int*) – Number of grid points to use along the primitive direction b. Must be larger than 3. **Default:** The value of na
- **nc** (*int*) – Number of grid points to use along the primitive direction c. Must be larger than 3. **Default:** The value of na

nA()

Returns
The sampling along the direction of unit cell vector a.

Return type
int

nB()

Returns
The sampling along the direction of unit cell vector b.

Return type
int

nC()

Returns
The sampling along the direction of unit cell vector c.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

sampling()

Returns
The grid sampling data as a tuple of three integers.

Return type
tuple of int

setNA(*value*)

Set the sampling in the direction parallel to unit vector a to a specified value.

Parameters
value (*int*) – The value to set the a sampling to. Must be larger than 3.

setNB(*value*)

Set the sampling in the direction parallel to unit vector b to a specified value.

Parameters
value (*int*) – The value to set the b sampling to. Must be larger than 3.

setNC(*value*)

Set the sampling in the direction parallel to unit vector c to a specified value.

Parameters
value (*int*) – The value to set the c sampling to. Must be larger than 3.

setSampling(*sampling*)

Set the sampling in all directions to a specified value.

Parameters
sampling (*int | tuple of int*) – The number of grid points in the three unitcell directions as a tuple of three integers larger than 3. If a single number is given, it is used for all three directions.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Use GridSampling to set density mesh cutoff.

```
numerical_accuracy_parameters=NumericalAccuracyParameters(
    density_mesh_cutoff=GridSampling(30,32,31),
    )
```

Use uniform GridSampling

```
# Set same grid sampling in each direction.
numerical_accuracy_parameters=NumericalAccuracyParameters(
    density_mesh_cutoff=GridSampling(30),
    )
```

GridValues

class GridValues(*data=None*, *unit=None*, *spin=None*, *spin_type=None*, *up_component_is_sum=None*)

Constructor - for internal use.

Parameters

- **data** (NLEngine.RealGrid3D | NLEngine.ComplexGrid3D) – The underlying backengine grid.
- **unit** (PhysicalQuantity) – The unit of the quantity stored on the grid.
- **spin** (Spin.All | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown | Spin.X | Spin.Y | Spin.Z | Spin.Sum) – The spin component. **Default:** Spin.All
- **spin_type** (NLEngine.UNPOLARIZED | NLEngine.POLARIZED | NLEngine.NONCOLLINEAR) – The spin type of the GridValues object. **Default:** NLEngine.UNPOLARIZED
- **up_component_is_sum** (*bool*) – In the case that spin_type is UNPOLARIZED, GridValues always assumes that the given data grid is the Spin.Sum grid. This flag determines how the Spin.Up/Spin.Down components are extracted from that grid.

  If this flag is true, the GridValues assumes the Spin.Up component is identical to the Spin.Sum component (as is e.g. the case for potential-type GridValues). As a result, a spin projection to Spin.Up or Spin.Down will return the originally given Spin.Sum data unchanged.

  If the flag is False, the Spin.Sum data will be divided by two when projecting to Spin.up or Spin.Down (as is the case for density-type GridValues).

  No effect if spin_type is not UNPOLARIZED.

  Default: False

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* with type length.) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* with type length.) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* with type length.) – The Cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The gradient at the specified point for the given spin. An error is raised, if the underlying data does not contain the spin projection of the requested type. In case that Spin.All is used, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned. Note that in the spin unpolarized case the Spin.X, Spin.Y and Spin.Z entries are zero, and for the spin polarized case the Spin.X and Spin.Y are zero.

Return type
PhysicalQuantity

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

•**downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.

- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown`) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The value at the specified point for the given spin. An error is raised, if the underlying data does not spin projection of the requested type. In case `Spin.All` is used, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned. Note that in the spin unpolarized case the `Spin.X`, `Spin.Y` and `Spin.Z` entries are zero, and for the spin polarized case the `Spin.X` and `Spin.Y` are zero.

Return type
PhysicalQuantity

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin the object was created with. If the spin was Spin.All, Spin.Sum will be used for the projection.

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

We distinguish between three types of spin treatment: Unpolarized, polarized and non-collinear. In the spin-unpolarized calculations we have no explicit treatment of spin and we assign the same values to the `Spin.Up` and `Spin.Down` projections. In spin-polarized calculation we treat spin in a collinear fashion and we distinguish between `Spin.Up` and `Spin.Down` components. Using the up and down components, the projection in z-direction can be computed, which can be accessed with `Spin.Z`. In the case of non-collinear calculations, we have spinors as solutions, and the spin is no longer aligned along a quantization axis. Non-collinear grid data can have four parts, up(-up), down(-down), the real part of up-down, and the imaginary part of up-down, accessible with the spin projections `Spin.Up`, `Spin.Down`, `Spin.RealUpDown` and `Spin.ImagUpDown`. Using these four components, the spin projection along the x-, y-, and z-direction can be computed. They can be retrieved using `Spin.X`, `Spin.Y`, and `Spin.Z`. For all three spin-types the `Spin.All` and `Spin.Sum` is supported.

Polarized example:

```
# Extract spin components from electron density.
spin_up   = electron_density.spinProjection(spin=Spin.Up)
spin_down = electron_density.spinProjection(spin=Spin.Down)
spin_sum  = electron_density.spinProjection(spin=Spin.Sum)

# Get data as numpy arrays.
up = spin_up.toArray()
down = spin_down.toArray()
sums = spin_sum.toArray()

# Print the total electron density for each component.
print "Up       ", (up).sum()
print "Down     ", (down).sum()
print
print "Up + Down", (up + down).sum()
print "Sum      ", (sums).sum()
```

The *GridValues* constructor is made for internal use. Constructing a generic grid must be done using the *gridValues()* function:

```
# Setup generic grid values object with random data.
data = numpy.random.random([30, 30, 30]) * Angstrom**-3
cell = SimpleCubic(4.0*Angstrom)
grid_values = gridValues(data, cell)
```

Notes

The following classes inherit from the *GridValues* object:

- *BlochState*
- *CurrentDensity*
- *EffectivePotential*
- *Eigenstate*
- *ElectronDensity*
- *ElectronDifferenceDensity*
- *ElectronLocalizationFunction*
- *ElectrostaticDifferencePotential*
- *ElectrostaticPotential*
- *ExchangeCorrelationPotential*
- *ExternalPotential*
- *HartreeDifferencePotential*
- *HartreePotential*
- *LocalDeviceDensityOfStates*
- *STM*
- *TransmissionEigenstate*

GrimmeDFTD2

class GrimmeDFTD2(*global_scale_factor=None*, *damping_factor=None*, *maximum_neighbour_distance=None*, *element_parameters=None*)

Class for determining the Grimme DFT-D2 dispersion correction of the energy, forces, and stress.

Parameters

- **global_scale_factor** (*float*) – The global scale factor of the DFT functional. **Default:** `0.75`
- **damping_factor** (*float*) – The damping factor. **Default:** `20.0`
- **maximum_neighbour_distance** (*PhysicalQuantity* of type length.) – The max interaction distance to consider. **Default:** `30.0 * Angstrom`
- **element_parameters** (*dict*) – Parameter for updating *R0* and *C6* values for specific elements. This should be given as a dictionary mapping *PeriodicTableElement* objects to tuples of *R0* and *C6* parameters. **Default:** No elements are updated.

dampingFactor()

Returns
The damping factor.

Return type
float

elementParameters()

This method gets the element parameters as a dictionary mapping *PeriodicTableElement* objects to tuples of *R0* and *C6* parameters.

Returns
The element parameters.

Return type
dict

globalScaleFactor()

Returns
The global scale factor.

Return type
float

maximumNeighbourDistance()

Returns
The maximum neighbour distance.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

setDampingFactor(*damping_factor*)

Set the damping factor.

Parameters
damping_factor (*float*) – Set the damping factor.

setElementParameters(*element_parameters*)

Set parameters *R0* and *C6* values for specific elements. This should be given as a dictionary mapping *PeriodicTableElement* objects to tuples of *R0* and *C6* parameters.

Parameters
element_parameters (*dict*) – Set the element parameters.

setGlobalScaleFactor(*global_scale_factor*)

Set the global scale factor.

Parameters
global_scale_factor (*float*) – Set the global scale factor.

setMaximumNeighbourDistance(*maximum_neighbour_distance*)

Set maximum neighbour distance.

Parameters
maximum_neighbour_distance (*PhysicalQuantity* of type length) – Set maximum neighbour distance.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Extend an *LCAOCalculator* with the DFT-D2 semiempirical dispersion correction by Grimme[1]:

```
# Grimme DFTD2
# Change/update R0 and C6 for Silicon
correction_extension = GrimmeDFTD2(
    global_scale_factor=0.75,
    damping_factor=20.0,
    maximum_neighbour_distance=30.0*Ang,
    element_parameters={
        Silicon: [1.716*Ang, 9.23*J*nm**6/mol],
        },
    )
# Apply correction to calculator
calculator = LCAOCalculator(correction_extension=correction_extension)
```

Notes

Recommended values of the `global_scale_factor` (s_6) for standard GGA functionals available in the ATK-DFT package: 0.75 (PBE), 1.2 (BLYP), 1.05 (BP86). More can be found in the original publication[Page 2985, 1].

GrimmeDFTD3

class GrimmeDFTD3(*global_scale_factor_6=None*, *damping_factor_6=None*, *global_scale_factor_8=None*, *damping_factor_8=None*, *maximum_neighbour_distance=None*, *include_three_body_term=None*, *exchange_correlation=None*)

Class for determining the Grimme DFT-D3 dispersion correction of the energy, forces, and stress.

Parameters

- **global_scale_factor_6** (*float*) – The global scale factor for the 6th order pole. **Default:** Selected based on the exchange-correlation functional.
- **damping_factor_6** (*float*) – The damping factor for the 6th order pole. **Default:** Selected based on the exchange-correlation functional.
- **global_scale_factor_8** (*float*) – The global scale factor for the 8th order pole. **Default:** Selected based on the exchange-correlation functional.
- **damping_factor_8** (*float*) – The damping factor for the 8th order pole. **Default:** Selected based on the exchange-correlation functional.
- **maximum_neighbour_distance** (*PhysicalQuantity* of type length) – The maximum interaction distance to consider. **Default:** `30.0 * Angstrom`
- **include_three_body_term** (*bool*) – If True, the three-body term is included. **Default:** `False`
- **exchange_correlation** (*ExchangeCorrelation*) – The exchange-correlation functional for selecting the Grimme DFT-D3 parameters. Should be one of `GGA.BLYP`, `SGGA.BLYP`, `NCGGA.BLYP`, `SOGGA.BLYP`, `GGA.PBE`, `SGGA.PBE`, `NCGGA.PBE`, `SOGGA.PBE`, `GGA.RPBE`, `SGGA.RPBE`, `NCGGA.RPBE`, `SOGGA.RPBE`, `HybridGGA.HSE06`, `HybridSGGA.HSE06`, `HybridNCGGA.HSE06`, `HybridSOGGA.HSE06`, `HybridGGA.PBE0`, `HybridSGGA.PBE0`, `HybridNCGGA.PBE0`,

[1] S.ăGrimme. Semiempirical GGA-type density functional constructed with a long-range dispersion correction. *J. Comput. Chem.*, 27(15):1787–1799, 2006. doi:10.1002/jcc.20495.

HybridSOGGA.PBE0, HybridGGA.B3LYP, HybridSGGA.B3LYP, HybridNCGGA.B3LYP or HybridSOGGA.B3LYP. **Default:** GGA.PBE

dampingFactor6()

Returns
The damping factor for the 6th order pole.

Return type
float

dampingFactor8()

Returns
The damping factor for the 8th order pole.

Return type
float

exchangeCorrelation()

Returns
The exchange-correlation functional for selecting default Grimme DFT-D3 parameters.

Return type
ExchangeCorrelation

globalScaleFactor6()

Returns
The global scale factor for the 6th order pole.

Return type
float

globalScaleFactor8()

Returns
The global scale factor for the 8th order pole.

Return type
float

includeThreeBodyTerm()

Returns
If True, the three-body term is included.

Return type
bool

maximumNeighbourDistance()

Returns
The neighbour cut off distance.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

setDampingFactor6(*damping_factor*)

Set the damping factor for the 6th order pole.

Parameters
damping_factor (*float*) – The damping factor for the 6th order pole.

setDampingFactor8(*damping_factor*)

Set the damping factor for the 8th order pole.

Parameters
damping_factor (*float*) – The damping factor for the 8th order pole.

setMaximumNeighbourDistance(*maximum_neighbour_distance*)

Set the neighbour cut off distance.

Parameters
maximum_neighbour_distance (*PhysicalQuantity* of type length) – The neighbour cut off distance.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Extend an *LCAOCalculator* with the DFT-D3 semiempirical dispersion correction by Grimme an co-workers[1]:

```
# Grimme DFT-D3
correction_extension = GrimmeDFTD3(exchange_correlation=GGA.RPBE)

# Apply correction to calculator
calculator = LCAOCalculator(
    exchange_correlation=GGA.RPBE,
    correction_extension=correction_extension,
    )

# Grimme DFT-D3. Note that we disable the three-body
# term since it's computationally very expensive.
correction_extension = GrimmeDFTD3(
    include_three_body_term=False,
    exchange_correlation=GGA.RPBE,
    )

# Apply correction to calculator
calculator = LCAOCalculator(
    exchange_correlation=GGA.RPBE,
    correction_extension=correction_extension,
    )
```

[1] S.ăGrimme, J.ăAntony, S.ăEhrlich, and H.ăKrieg. A consistent and accurate ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu. *J. Chem. Phys.*, 2010. doi:10.1063/1.3382344.

Notes

The default scaling and damping parameters used by the *GrimmeDFTD3* object are those optimized for the exchange-correlation functional given to the object, see Table 4.18. For PBE, BLYP, and rPBE, the appropriate parameters are *automatically* set. For other functionals, the scaling and damping parameters should be manually specified.

Attention: The DFT-D3 method contains both two-body and three-body terms. The three-body term can be prohibitively expensive to compute for bulk systems, although it typically contributes no more than 5–10% of the total dispersion energy. It is therefore disabled by default. Note also that the calculation of forces and stress can be computationally very expensive with the DFT-D3 method, even if the three-body term has been disabled. Therefore, for such calculations it is recommended to use the *GrimmeDFTD2* method instead.

Table 4.18: Scaling parameters s_6 and s_8, and damping parameters d_6 and d_8, for the DFT-D3 method.

Functional	scaling_parameter_6	damping_parameter_6	scaling_parameter_8	damping_parameter_8
PBE	1.0	1.217	0.7220	1.0
BLYP	1.0	1.094	1.6820	1.0
revPBE	1.0	0.923	1.0100	1.0
PBEsol	1.0	1.345	0.6120	1.0
rPBE	1.0	0.872	0.5140	1.0
B3LYP	1.0	1.261	1.7030	1.0
PBE0	1.0	1.287	0.9280	1.0

GrueneisenCoefficient

class GrueneisenCoefficient(*configuration*, *filename*, *object_id*, *calculator=None*, *cell_displacement_factor=None*, *route=None*, *points_per_segment=None*, *path=None*, *dynamical_matrix_parameters=None*, *optimize_geometry_parameters=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*)

Calculate the Grueneisen parameter and thermal expansion coefficient.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration for which the Grueneisen coefficient should be calculated.
- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **calculator** (Calculators) – The calculator to be used. **Default:** The calculator attached to the configuration.
- **cell_displacement_factor** (*float*) – The factor for which the initial configuration cell volume expands or shrinks. E.g. a value of 0.1 means that the volume expands and shrinks by 10%. **Default:** 0.1
- **route** (*list*) – The route to take through the Brillouin-zone. Must be given as a list of symmetry points of the unit cell, e.g. [G, X, G].
- **points_per_segment** (*int*) – The number of points per segment of the route. **Default:** 20
- **path** (*list*) – A list of Fourier-points (in fractional coordinates) in which the bandstructure should be calculated. This options is mutually exclusive to route, and points_per_segment.

- **dynamical_matrix_parameters** (*DynamicalMatrixParameters*) – The DynamicalMatrix parameters. **Default:** `DynamicalMatrixParameters()`.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters for the geometry optimization.
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'molecular_reaction_path_'`.
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively.

cellDisplacementFactor()

Returns
The factor for which the initial configuration cell expands or contracts.

Return type
float

configuration()

Returns
The initial configuration.

Return type
BulkConfiguration

contractedConfiguration()

Returns
The contracted configuration.

Return type
BulkConfiguration

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

dynamicalMatrixParameters()

Returns
The parameters for the dynamical matrix.

Return type
DynamicalMatrixParameters

expandedConfiguration()

Returns
The expanded configuration.

Return type
BulkConfiguration

filename()

Returns
The filename where the study object is stored.

Return type
str

grueneisenCoefficient(*temperature*)

Compute the Grueneisen coefficient at the given temperature.

Parameters
temperature (PhysicalQuantity in units of temperature) – The temperature condition to consider.

Returns
The Grueneisen coefficient.

Return type
float

linearThermalExpansionCoefficient(*temperature*)

Compute the linear thermal expansion at the given temperature. The linear thermal expansion is per lattice vector.

Parameters
temperature (PhysicalQuantity in units of temperature) – The temperature condition to consider.

Returns
The linear thermal expansion coefficient.

Return type
PhysicalQuantity in units of inverse temperature

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlinfo()

Get only the printable results from the extraction results - plain text.

Returns
The study information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometryParameters()

Returns
The parameters for the geometry optimization.

Return type
OptimizeGeometryParameters

phononEnergies()

Returns
The eigenvalues for the phonon band structure. The shape is (Q, B) where Q is the number of q-points and B is the number of bands.

Return type
PhysicalQuantity of type energy

qpoints()

Get the list of Fourier points used in the band structure.

Returns
The fractional coordinates of the Fourier points.

Return type
numpy.ndarray

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

update()

Perform the update. Here, the mode and q point specific Grueneisen coefficients are calculated and used to calculate the temperature dependent part in the thermal expansion coefficient and macroscopic Grueneisen parameter.

volumetricThermalExpansionCoefficient(*temperature*)

Compute the volumetric thermal expansion at the given temperature. The thermal expansion is called volumetric as we assume the expansion is isotropic.

Parameters
temperature (PhysicalQuantity in units of temperature) – The temperature condition to consider.

Returns
The volumetric thermal expansion coefficient.

Return type
PhysicalQuantity in units of inverse temperature

Usage Examples

Calculate the temperature-dependent Grueneisen coefficient and linear thermal expansion for silicon FCC.

```
# %% GrueneisenCoefficient

grueneisen_coefficient = GrueneisenCoefficient(
    configuration=bulk_configuration,
    filename='grueneisen_coefficient_example.hdf5',
    object_id='gc',
    points_per_segment=10,
)
grueneisen_coefficient.update()

print('%15s%30s%30s' % ('Temperature (K)', 'Grueneisen', 'Linear CTE (1/K)'))


temperatures = numpy.linspace(300, 1200, 10) * Kelvin
for temperature in temperatures:
    grueneisen = grueneisen_coefficient.grueneisenCoefficient(temperature)
    linear_thermal_expansion_coefficient = (
        grueneisen_coefficient.linearThermalExpansionCoefficient(temperature)
    )
    print(
        '%15.0f%30.5f%30.5E' % (
            temperature.inUnitsOf(Kelvin),
            grueneisen,
            linear_thermal_expansion_coefficient.inUnitsOf(Kelvin**-1)
        )
    )
```

grueneisen_coefficient_example.py

Notes

The *GrueneisenCoefficient* object calculates two properties of a bulk material; the Grüneisen and thermal expansion coefficients denoted γ and α, respectively. The thermal expansion coefficient can generally be given either as linear or volumetric, denoted by subscripts L and V, respectively. If a material expands with equal rates in all directions (also called an isotropic material), the following simple relation is valid,

$$\alpha_V \approx 3\alpha_L$$

This assumption is made when calculating α_V from α_L in the current implementation.

The Grüneisen coefficient is a dimensionless parameter with different valid formulations. Here, the following expression is used,

$$\gamma = \frac{\alpha_L B}{C_V}$$

where B is the Hill bulk modulus and C_V is the heat capacity at constant volume. The Grüneisen coefficient and heat capacity can further be refined into their mode index (i) and $\mathbf{q}$-point microscopic components denoted $\gamma(\mathbf{q}, i)$ and $C_V(\mathbf{q}, i, T)$ defined as,

$$\gamma(\mathbf{q}, i) = -\frac{V_0}{2[h\nu_i(\mathbf{q})]^2} \left\langle \mathbf{e}_i(\mathbf{q}) \middle| \frac{dD(\mathbf{q})}{dV} \middle| \mathbf{e}_i(\mathbf{q}) \right\rangle$$

$$C_V(\mathbf{q}, i, T) = \frac{[h\nu_i(\mathbf{q})]^2}{V_0 k_B T^2} \frac{\exp h\nu_i(\mathbf{q})}{[\exp h\nu_i(\mathbf{q}) - 1]^2}$$

where V_0 is the initial configuration cell volume, $\mathbf{e}$ are the phonon eigensystem eigenvectors, $dD(\mathbf{q})$ is the change in *DynamicalMatrix* elements at specific $\mathbf{q}$-points and dV is the volume change around V_0. h and k_B are the Planck and Boltzmann constants, respectively. Once the microscopic quantities are calculated, the temperature dependent linear thermal expansion coefficient is determined according to,

$$\alpha_L(T) = \frac{\sum_{i,\mathbf{q}} \gamma(\mathbf{q}, i) C_V(\mathbf{q}, i, T)}{B}$$

and normalized per cell based on the *DynamicalMatrix* repetitions. The (macroscopic) temperature-dependent Grüneisen coefficient is calculated as,

$$\gamma(T) = \frac{\sum_{i,\mathbf{q}} \gamma(\mathbf{q}, i) C_V(\mathbf{q}, i, T)}{\sum_i C_V(\mathbf{q}, i, T)}$$

HSECustomExchangeCorrelation

class HSECustomExchangeCorrelation(*screening_length=PhysicalQuantity(0.11, 1 / Bohr)*, *exx_fraction=0.25*, *number_of_spins=1*, *spin_orbit=None*)

Initialize the base object with the correct number of spins, and with the only Exchange and Correlation currently allowed for this class, i.e. the HSE ones.

Parameters

- **screening_length** (float *PhysicalQuantity* of type inverse length) – The screening length defined for the coulomb kernel. **Default:** The HSE06 value of `0.11 * Bohr**-1`
- **exx_fraction** (*float*) – The fraction of exact exchange to be added in with the PBE exchange **Default:** The HSE06 value of 0.25
- **number_of_spins** (1 | 2) – The number of spins to be used in the calculation. **Default:** `1`
- **spin_orbit** (*bool*) – If `True` spin-orbit coupling is considered. **Default:** `False`

cParameter(*grid_descriptor=None*)

return the param C.

Parameters
grid_descriptor (*NLEngine.GridDescriptor*) – The grid descriptor.

cloneAsPolarized()

Create a new functional with all the same settings, but polarized spin.

cloneAsSpinOrbit()

Create a new functional with all the same settings, but with Spin-Orbit coupling included.

correlation()

Returns
The correlation functional class

Return type
Correlation

defaultExactExchangeFraction()

Returns
The default exact exchange fraction that is mixed in with PBE exchange.

Return type
float

defaultScreeningLength()

Returns
The default screening length defined for the coulomb kernel.

Return type
PhysicalQuantity of type inverse length

densityCheckThreshold()

Returns
The density safe guard.

Return type
float

dftHalfEnabled()

Returns
Whether DFT-1/2 is enabled.

Type
bool

dielectricDependentHybridFunctional()

Returns
True, then a dielectric dependent hybrid functional is used.

Return type
bool

dielectricDependentHybridParameters()

Returns
The dielectric dependent hybrid hybrid parameters.

Return type
list

exactExchangeFraction()

Returns
The part of exact exchange that is mixed in with PBE exchange.

Return type
float

exchange()

Returns
The exchange functional class

Return type
Exchange

hubbardTerm()

Returns
The Hubbard term used.

Return type
Onsite | OnsiteShell | Dual | DualShell | None

maximumPotential()

Set maximum potential value allowed for the TB09. The value to use as the maximum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

minimumPotential()

Set minimum potential value allowed for the TB09. The value to use as the minimum allowed value for the potential in points with density lower than threshold.

Returns
The density safe guard.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a string representation of the *ExchangeCorrelation* instance.

Parameters
stream (A stream that supports strings being written to using *write*.) – The stream the exchange-correlation should be written to. **Default:** sys.stdout

numberOfSpins()

Returns
The number of spins.

Return type
1 | 2 | 4

params()

Get the parameters

screeningLength()

Returns
The screening length defined for the coulomb kernel.

Return type
PhysicalQuantity of type inverse length

spinOrbit()

Returns
Boolean determining if spin-orbit is enabled (True) or not (False).

Return type
bool.

spinType()

Returns
The spin type.

Return type
NLEngine.POLARIZED | NLEngine.NONCOLLINEAR | NLEngine.UNPOLARIZED

uniqueString()

Return a unique string representing the state of the object.

unscreenedExchange()

Whether the exchange functional in use is unscreened.

useLocalCParameter()

Returns
True, then a local c-parameter is used.

Return type
bool

HTSTEvent

class HTSTEvent(*neb*, *assumed_prefactor=None*, *minimum_displacement=None*, *finite_difference=None*, *finite_difference_method=None*, *constraints=None*, *saddle_image_index=None*, *spline_estimate=None*, *dynamical_matrix_log_filename_prefix=None*, *dynamical_matrix_filename=None*, *dynamical_matrix_object_id_prefix=None*, *dynamical_matrix_processes_per_displacement=None*)

Calculate harmonic transition state theory (HTST) reaction rates using configurations from a *NudgedElasticBand* object.

Parameters

- **neb** (NudgedElasticBand) – The nudged elastic band configuration to use.
- **assumed_prefactor** (*PhysicalQuantity* of type per Second) – A fixed value for the prefactors (both forward and reverse). If assumed_prefactor is None, then the vibrational frequencies at the minima and saddle point will be calculated. **Default:** None.

- **minimum_displacement** (*PhysicalQuantity* of type length) – The minimum distance that an atom must move during the reaction in order to be included in the prefactor calculation. If an atom moves further than this distance, it and its neighbors will be included in the dynamical matrix calculation that is needed for calculating the prefactor. Setting this value to zero will include all degrees of freedom. Setting this value to a larger number will reduce the amount of worked needed to calculate the prefactor, but will decrease the accuracy. Note that this parameter is only used when `assumed_prefactor` is None. **Default:** `0.05 * Angstrom`.
- **finite_difference** (*PhysicalQuantity* of type length) – The finite difference step size used when estimating the force constants. **Default:** `0.01*Angstrom`.
- **finite_difference_method** (`Central | Forward`) – The finite difference method to use when estimating the force constants needed to calculate the harmonic prefactor. **Default:** `Central`.
- **constraints** (*[type]*) – [description].
- **saddle_image_index** (*int*) – The index of the image that is assumed to be the saddle point. If `saddle_image_index` is None, then the image with the largest energy will be chosen. **Default:** `None`.
- **spline_estimate** (*bool*) – Obtain a rough estimate of the prefactor using a spline to obtain the vibrational frequency at the minimum in the direction of the saddle point. This requires that the NEB configuration has either had its update method called or has already been optimized so that the energy and forces of each image are available. This option is incompatible with `assumed_prefactor`. **Default:** `False`.
- **dynamical_matrix_log_filename_prefix** (*str | LogToStdOut*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (`".log"`). A value of `LogToStdOut` means all logging will go to stdout. **Default:** `"htst_"`.
- **dynamical_matrix_filename** (*str | LogToStdOut*) – Name of the file to save the dynamical matrix. LogToStdOut if no file needed.
- **dynamical_matrix_object_id_prefix** (*str*) – The prefix of the object id to use when saving. **Default:** `"htst_"`.
- **dynamical_matrix_processes_per_displacement** (*int | Automatic*) – The number of processes assigned to calculating a single displacement **Default:** 1 process per displacement.

:param : [description]. :type : [type]

forwardBarrier()

Returns
The forward (reactant to product) barrier.

Return type
PhysicalQuantity of type energy

forwardPrefactor()

Returns
The forward (reactant to product) vibrational prefactor.

Return type
PhysicalQuantity of type per second

forwardRate(*temperature*)

Calculate the forward (reactant to product) reaction rate using HTST.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The reaction temperature.

Returns
The forward reaction rate.

Return type
PhysicalQuantity of type per second

classmethod fromConfigurations(*reactant_configuration*, *saddle_configuration*, *product_configuration*, *assumed_prefactor=None*, *minimum_displacement=PhysicalQuantity(0.05, Ang)*, *finite_difference=PhysicalQuantity(0.01, Ang)*, *finite_difference_method=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Central'>*, *dynamical_matrix_log_filename_prefix='htst_'*, *dynamical_matrix_filename=None*, *dynamical_matrix_object_id_prefix='htst_'*, *dynamical_matrix_processes_per_displacement=1*)

This is an alternate constructor for this class that uses explicit configurations for the reactant minimum configuration, saddle point configuration, and product minimum configuration as opposed to extracting these configurations from a NudgedElasticBand object.

Parameters

- **reactant_configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration at the reactant/initial minimum.
- **saddle_configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration at the saddle point.
- **product_configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration at the product/final minimum.
- **assumed_prefactor** (*PhysicalQuantity* of type per Second) – A fixed value for the prefactors (both forward and reverse). If `assumed_prefactor` is None, then the vibrational frequencies at the minima and saddle point will be calculated. **Default:** `None`
- **minimum_displacement** (*PhysicalQuantity* of type length) – The minimum distance that an atom must move during the reaction in order to be included in the prefactor calculation. If an atom moves further than this distance, it and its neighbors will be included in the dynamical matrix calculation that is needed for calculating the prefactor. Setting this value to zero will include all degrees of freedom. Setting this value to a larger number will reduce the amount of worked needed to calculate the prefactor, but will decrease the accuracy. Note that this parameter is only used when `assumed_prefactor` is None. **Default:** `0.05*Angstrom`
- **finite_difference** (*PhysicalQuantity* of type length) – The finite difference step size used when estimating the force constants. **Default:** `0.01*Angstrom`
- **finite_difference_method** (`Central` | `Forward`) – The finite difference method to use when estimating the force constants needed to calculate the harmonic prefactor. **Default:** `Central`
- **dynamical_matrix_log_filename_prefix** (*str | None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). A value of None means all logging will go to stdout. **Default:** `"htst_"`
- **dynamical_matrix_object_id_prefix** (*str*) – The prefix of the object id to use when saving. **Default:** `"htst_"`
- **dynamical_matrix_filename** – Name of the file to save the dynamical matrix. None if no file needed.
- **dynamical_matrix_processes_per_displacement** (*int*) – The number of processes assigned to calculating a single displacement **Default:** 1 process per displacement.

Type
str | None

static htstRate(*barrier*, *temperature*, *prefactor*)

Calculate the HTST reaction rate at the specified temperature.

Parameters

- **barrier** (*PhysicalQuantity* of type energy) – The energy barrier.
- **temperature** (*PhysicalQuantity* of type temperature) – The reaction temperature.
- **prefactor** (*PhysicalQuantity* of type per second) – The harmonic prefactor (attempt frequency).

Returns
The HTST reaction rate.

Return type
PhysicalQuantity of type per second

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

productConfiguration()

Returns
The product configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

productEnergy()

Returns
The energy of the product configuration.

Return type
PhysicalQuantity of type energy

productFrequencies()

Returns
The angular vibrational frequencies at the product configuration.

Return type
PhysicalQuantity of type radians per second

reactantConfiguration()

Returns
The reactant configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

reactantEnergy()

Returns
The energy of the reactant configuration.

Return type
PhysicalQuantity of type energy

reactantFrequencies()

Returns
The angular vibrational frequencies at the reactant configuration.

Return type
PhysicalQuantity of type radians per second

reverseBarrier()

Returns
The reverse (product to reactant) barrier.

Return type
PhysicalQuantity of type energy

reversePrefactor()

Returns
The reverse (product to reactant) vibrational prefactor.

Return type
PhysicalQuantity of type per second

reverseRate(*temperature*)

Calculate the reverse (product to reactant) reaction rate using HTST.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The reaction temperature.

Returns
The reverse reaction rate.

Return type
PhysicalQuantity of type per second

saddleConfiguration()

Returns
The saddle configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

saddleEnergy()

Returns
The energy of the saddle configuration.

Return type
PhysicalQuantity of type energy

saddleFrequencies()

Returns
The angular vibrational frequencies at the saddle configuration.

Return type
PhysicalQuantity of type radians per second

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Logging

If the dynamical matrix is calculated at the minima and saddle points, the output from the calculator for each displacement is written to a separate file on the disk. By default, the displacements at the initial minimum will have the pattern `htst_reactant_displacement_%i.log` where the `%i` is an integer representing the displacement number. The displacements and the saddle are named `htst_saddle_displacement_%i.log` and displacements at the final minimum are named `htst_product_displacement_%i.log`. The prefix `htst_` can be modified by changing the `dynamical_matrix_log_filename_prefix` parameter. Alternatively, all file-based logging can be disabled by setting `dynamical_matrix_log_filename_prefix` to `None`.

Notes

This class calculates reaction rates using harmonic transition state theory (HTST),

$$k_{\mathrm{HTST}} = \frac{\prod_i^{3N} \nu_i^{\mathrm{min}}}{\prod_i^{3N-1} \nu_i^{\ddagger}} \exp\left[-\left(E^{\ddagger} - E^{\mathrm{min}}\right)/k_{\mathrm{B}}T\right],$$

where k_{HTST} is the HTST reaction rate, N is the number of atoms, ν_i^{min} and $\nu_i^{\ddagger}$ are the positive normal mode frequencies at the minimum and at the saddle point, E^{min} $E^{\ddagger}$ are the energies at the minimum and at the saddle point, k_{B} is Boltzmanns constant and T is the temperature. The ratio of the product of the normal mode frequencies (the term in front of the exponential) is called the prefactor or the attempt frequency.

Determining the prefactor is an expensive calculation. Formally, it requires calculating the eigenvalues of the dynamical matrix (without any repetitions) of the minima and of the saddle point. However, it can be approximated by reducing the dimensionality of the system to include only those atoms that move between the minimum and saddle point and their near neighbors. This approximation can be controlled using the `minimum_displacement` parameter. The default value of is 0.05 Angstrom and typically estimates the prefactor to within 5-10% of the true value. Note that the prefactor is temperature *independent*. This means that it only needs to be calculated once and then all calls to `forwardRate(temperature)` and `reverseRate(temperature)` can compute the rate inexpensively.

When calculating the eigenvalues of the dynamical matrix, the `finite_difference_method` parameter controls the order of the finite difference method used to compute the force constants. The `finite_difference` parameter controls the step length. The Forward method uses a first order finite difference scheme that only needs one displacement per degree of freedom, while the Central method uses a second order finite difference scheme that needs two displacements per degree of freedom, but is much more accurate. The central difference scheme is recommended when using DFT due to numerical noise that arises from having a discrete grid.

Alternatively, one can disable the prefactor calculation entirely by passing a frequency using the `assume_prefactor` parameter. In most systems, this value does not change significantly between different reactions and mostly depends upon the mass of the atoms involved. For lighter elements (about the first three rows of the periodic table) the prefactor is approximately $10^{13}\,\mathrm{s}^{-1}$ while for heavier elements, it about $10^{12}\,\mathrm{s}^{-1}$.

Lastly, there is a very crude way to estimate the prefactor by using a spline interpolation along the NEB reaction coordinate to determine the vibrational frequency in the direction of the saddle point. This option is enabled by setting `spline_estimate` to `True`. This very simple approach often gives a reasonable estimate of the prefactor, but is not guaranteed to give any particular level of accuracy and should not be relied upon.

HTSTParameters

class HTSTParameters(*assumed_prefactor=None*, *minimum_displacement=None*, *finite_difference=None*, *finite_difference_method=None*, *spline_estimate=None*, *dynamical_matrix_processes_per_displacement=1*)

This class stores parameters for the *HTSTEvent* calculation.

Parameters

- **assumed_prefactor** (*PhysicalQuantity* of type per Second) – A fixed value for the prefactors (both forward and reverse). If `assumed_prefactor` is None, then the vibrational frequencies at the minima and saddle point will be calculated. **Default:** `None`
- **minimum_displacement** (*PhysicalQuantity* of type length) – The minimum distance that an atom must move during the reaction in order to be included in the prefactor calculation. If an atom moves further than this distance, it and its neighbors will be included in the dynamical matrix calculation that is needed for calculating the prefactor. Setting this value to zero will include all degrees of freedom. Setting this value to a larger number will reduce the amount of work needed to calculate the prefactor, but will decrease the accuracy. Note that this parameter is only used when `assumed_prefactor` is None. **Default:** `0.05 * Angstrom`
- **finite_difference** (*PhysicalQuantity* of type length) – The finite difference step size used when estimating the force constants needed to calculate the harmonic prefactor. **Default:** `0.01*Angstrom`
- **finite_difference_method** (`Central | Forward`) – The finite difference method to use when estimating the force constants needed to calculate the harmonic prefactor. **Default:** `Central`
- **spline_estimate** (*bool*) – Obtain a rough estimate of the prefactor using a spline to obtain the vibrational frequency at the minimum in the direction of the saddle point. This requires that the NEB configuration has either had its update method called or has already been optimized so that the energy and forces of each image are available. This option is incompatible with `assumed_prefactor`. **Default:** `False`
- **dynamical_matrix_processes_per_displacement** (*int*) – The number of processes assigned to calculating a single displacement **Default:** 1 process per displacement.

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate harmonic transition state theory prefactors using forward finite differences (`finite_difference_method=Forward`) with a small step length (`finite_difference=1e-4*Angstrom`) and include all atoms in the dynamical matrix calculation (`minimum_displacement=0.0*Angstrom`).

```
htst_parameters = HTSTParameters(
    finite_difference=1e-4*Angstrom,
    finite_difference_method=Forward,
    minimum_displacement=0.0*Angstrom,
    )
```

`parameters.py`

Notes

These parameters are normally passed to the *AdaptiveKineticMonteCarlo* class. For more details on harmonic transition state theory, see the *HTSTEvent* class.

HalgrenLipscomb

class HalgrenLipscomb(*constraints=None*)

Use the Halgren-Lipscomb method for interpolating images.

Parameters
constraints (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained. **Default:** []

constraints()

Returns
The constrained atoms.

Return type
list of type int

setConstraints(*constraints*)

Set the constraints.

Parameters
constraints (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained.

tags()

Returns
The tags used for defining the constraints.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

HamiltonianDerivatives

class HamiltonianDerivatives(*configuration*, *filename*, *object_id*, *repetitions=None*, *atomic_displacement=None*, *constraints=None*, *use_equivalent_bulk=None*, *constrain_electrodes=None*, *log_filename_prefix=None*, *processes_per_displacement=None*, *use_wigner_seitz_scheme=None*, *auxiliary_basis_set=None*, *compensate_fermi_level_shift=None*)

Constructor for the HamiltonianDerivatives object.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration* | *DeviceConfiguration*) – The configuration for which to calculate the Hamiltonian derivatives.
- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **repetitions** (`Automatic` | list of ints) – The number of repetitions of the system in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`, or `Automatic`. Each repetition value must be odd. **Default:** `Automatic`
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01 * Angstrom`
- **constraints** (*list of type int*) – List of atomic indices that will be constrained, e.g. `[0, 2, 10]`. **Default:** Empty list `[]`
- **use_equivalent_bulk** (*bool*) – Control if a *DeviceConfiguration* should be treated as a *BulkConfiguration*. **Default:** `True`
- **constrain_electrodes** (*bool*) – Control if the electrodes and electrode extensions should be constrained in case of a *DeviceConfiguration*. **Default:** `False`
- **log_filename_prefix** (*str or None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). **Default:** `"hamiltonian_displacement_"`
- **processes_per_displacement** (int | *ProcessesPerNode*) – The number of processes assigned to calculating a single displacement. **Default:** *ProcessesPerNode*
- **use_wigner_seitz_scheme** (*bool*) – Control if the real space HamiltonianDerivative Matrix should be extended according to the Wigner Seitz construction. `use_wigner_seitz_scheme=True` is only supported for simple orthorhombic, simple tetragonal and simple cubic lattices. **Default:** `False`
- **auxiliary_basis_set** (list | tuple | *BasisSet*) – A different (typically smaller) basis set to be used for evaluating the derivative of the effective potential. This auxiliary basis set will be used in the self-consistent updates, and can potentially speed up the calculations significantly, although at the cost of accuracy. The Hamiltonian derivatives matrices will be evaluated in the basis set on the calculator. This option is only available for LCAOCalculators. **Default:** `None` i.e. no auxiliary basis set is used.
- **compensate_fermi_level_shift** (*bool*) – If `True`, possible changes in the Fermi level due to finite size effects are compensated, according to eq.17 in ref.[1]. This requires the evaluation of the Fermi level for each displaced configuration. In the case of non-selfconsistent semi-empirical calculations, setting it to `False` will give a large speedup.

  Note: for *LCAOCalculator* this parameter is ignored, as the hamiltonian derivatives are calculated differently, and it is not necessary to adjust for finite size effects. **Default:** `True`

[1] T.ăFrederiksen, M.ăPaulsson, M.ăBrandbyge, and A.-P. Jauho. Inelastic transport theory from first principles: Methodology and application to nanoscale devices. *Phys. Rev. B*, 75:205413, May 2007. doi:10.1103/PhysRevB.75.205413.

`atomicDisplacement()`

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity with length unit.

`auxiliaryBasisSet()`

Returns
A different (typically smaller) basis set to be used for evaluating the derivative of the effective potential.

Return type
list | tuple | *BasisSet*

`compensateFermiLevelShift()`

Returns
Whether a change in Fermi level for displaced configurations should be compensated.

Return type
bool

`constrainElectrodes()`

Returns
Boolean determining if the electrodes and electrode extensions are constrained in case of a DeviceConfiguration.

Return type
bool

`constraints()`

Returns
The list of constrained atoms.

Return type
list of int

`dependentStudies()`

Returns
The list of dependent studies.

Return type
list of `Study`

`filename()`

Returns
The filename where the study object is stored.

Return type
str

`logFilenamePrefix()`

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

nlinfo()

Returns
Structured information about the HamiltonianDerivatives.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

processesPerDisplacement()

Returns
The number of processes per displacement.

Return type
int | *ProcessesPerNode*

repetitions()

Returns
The number of repetitions of the system in the A, B, and C-directions.

Return type
list of three int.

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the HamiltonianDerivatives study object.

useEquivalentBulk()

Returns
Boolean determining if a DeviceConfiguration is treated as a BulkConfiguration.

Return type
bool.

wignerSeitzScheme()

Returns
Boolean to control if the real space Dynamical Matrix should be extended according to the Wigner-Seitz construction.

Return type
bool

Usage Examples

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Calculate the Hamiltonian derivatives for a system repeated five times in the B direction and three times in the C direction.

```
hamiltonian_derivatives = HamiltonianDerivatives(
    configuration,
    filename='HamiltonianDerivatives.hdf5',
    object_id='hamiltonian_derivatives',
    repetitions=(1,5,3),
    )
hamiltonian_derivatives.update()
```

When using `repetitions=Automatic`, the cell is repeated such that all atoms within a pre-defined, element-pair dependent interaction range are included.

```
hamiltonian_derivatives = HamiltonianDerivatives(
    configuration,
    filename='HamiltonianDerivatives.hdf5',
    object_id='hamiltonian_derivatives',
    repetitions=Automatic,
    )
hamiltonian_derivatives.update()
```

The default number of repetitions i.e. `repetitions=Automatic` can be found before a calculation using the function *checkNumberOfRepetitions()*.

```
(nA, nB, nC)  = checkNumberOfRepetitions(configuration)
```

Notes

The Hamiltonian derivatives are calculated using the central finite difference method in a repeated cell constituting a super cell. That is, the Hamiltonian derivatives are calculated for each atom in the central cell by displacing the atom in the supercell and determining the Hamiltonian for two displacements from its original position along each of the Cartesian directions. In DFT, the derivatives of the Hamiltonian $\hat{H}$ can be expressed as the derivatives of the effective potential V_{eff} since the kinetic term does not contribute. Thus the Hamiltonian derivatives for the i and j basis functions can be approximated as

$$\langle i|\frac{\partial \hat{H}}{\partial R_{I,\alpha}}|j\rangle = \langle i|\frac{\partial V_{\text{eff}}}{\partial R_{I,\alpha}}|j\rangle \approx \langle i|\frac{V_{\text{eff}}(R_{I,\alpha}+\delta) - V_{\text{eff}}(R_{I,\alpha}-\delta)}{2\delta}|j\rangle,$$

where $R_{I,\alpha}$ is the α cartesian coordinate for atom I in the central unit cell, and δ is the atomic displacement. For Slater-Koster calculators, the Hamiltonian derivatives are described by the on-site and off-site parameters and their distance dependence, hence

$$\langle i|\frac{\partial \hat{H}}{\partial R_{I,\alpha}}|j\rangle \approx \frac{\partial}{\partial R_{I,\alpha}}\left(\langle i|\hat{H}|j\rangle\right) \approx \frac{H_{ij}(R_{I,\alpha}+\delta) - H_{ij}(R_{I,\alpha}-\delta)}{2\delta}.$$

Terminated HamiltonianDerivatives calculations can be resumed by re-running the same script or reading the study object from file and calling `update()` on it. The study object will automatically detect which displacement calculations have already been carried out and only run the calculations that are not yet completed.

Notes for DFT

When calculating the Hamiltonian derivatives both the number of sampled k-points for the super cell and repetitions in the confined the directions must be 1. In the following the number of sampled k-points for the super cell and repetitions in the non-confined directions will be adressed. To simplify things a system with only one non-confined direction will be used, see Fig. 4.23 (a), but the relations for the one non-confined direction applies to all non-confined directions. The bulk configuration in Fig. 4.23 (a) is converged in the total energy with respect to the number of k-points of $(1, 1, N_{\text{C}})$. The number of repetitions in the super cell in the non-confined direction, see Fig. 4.23 (b), is chosen large enough such that the change in the effective potential $\text{d}V_{\text{eff}}$ goes to zero at the boundaries of the super cell in the non-confined directions for every atomic displacement, confer Fig. 4.23 (c). For this system five repetitions of the unit cell in the confined direction is enough for the change in the effective potential to go to zero at the boundaries. The recommended k-point sampling for the non-confined direction is the number of k-points in the non-confined direction for the unit cell divided by the number of repetitions in the non-confined direction. The k-point sampling then becomes $(1, 1, \frac{N_{\text{C}}}{\text{repetitions in C}})$ and in this particular case $(1, 1, N_{\text{C}}/5)$.

Note: From QuantumATK-2019.03 onwards, the k-point sampling and density-mesh-cutoff will be **automatically** adapted to the given number of repetitions when setting up the super cell inside *DynamicalMatrix* and *HamiltonianDerivatives*. That means you can specify the calculator settings for the unit cell and use it with any desired number of repetitions in dynamical matrix and hamiltonian derivatives calculations.

The Hamiltonian derivatives calculations generally requires a low tolerance in the *IterationControlParameters* settings, e.g. a `tolerance` of 1e-6. Finally, it should be noted that the *HuckelCalculator* currently does not support calculation of the Hamiltonian derivatives.

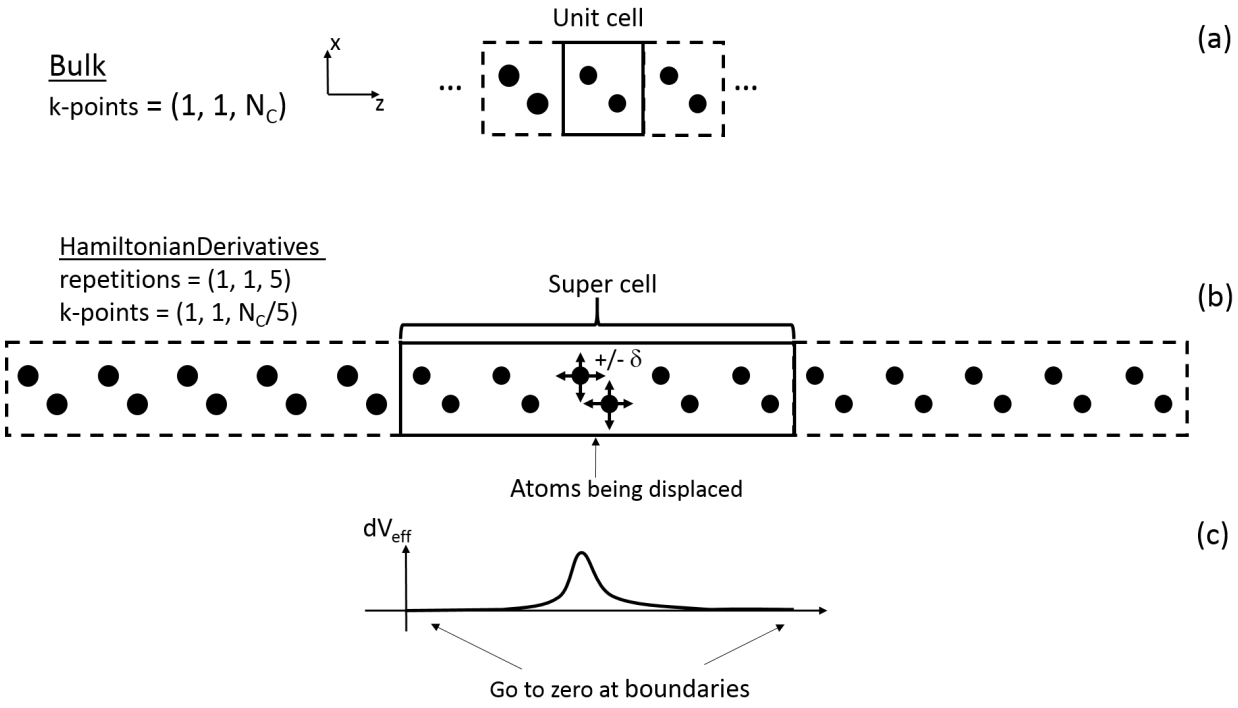


Fig. 4.23: **(a)** a unit cell for a 1D system confined in the x and y -directions. The k-point sampling is correspondingly $(1, 1, N_{\mathrm{C}})$, where N_{C} is the sampling in the z -direction. **(b)** a super cell created by five repetitions in the z -direction of the unit cell in **(a)**. In the calculation of the Hamiltonian derivatives the atoms in the center unit cell of the super cell (the two atom indicated with arrows) are displaced by $\pm\delta$ in each of the Cartesian directions. **(c)** the change in the effective potential $\mathrm{d}V_{\mathrm{eff}}$ in the super cell as an atom is displaced δ . The number of repetitions must ensure that the change in the effective potential goes to zero at the boundaries of the super cell in the non-confined directions for every atomic displacement.

HarmonicAnglePotential

class HarmonicAnglePotential(*particleType1*, *particleType2*, *particleType3*, *k*, *theta0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **k** (*PhysicalQuantity of type energy / angle**2*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter (the equilibrium angle).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a HarmonicAnglePotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethane_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=-0.3*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)
# Set up a new angle potential for H-C-C angles and add it to the potential set.
angle_potential = HarmonicAnglePotential(
    particleType1='H',
    particleType2='C',
```

(continues on next page)

(continued from previous page)

```
    particleType3='C',
    k=2.1682*eV,
    theta0=1.9111355,
)
potential_set.addPotential(angle_potential)

# Set up a new angle potential for H-C-H angles and add it to the potential set.
angle_potential = HarmonicAnglePotential(
    particleType1='H',
    particleType2='C',
    particleType3='H',
    k=1.51774*eV,
    theta0=1.9111355,
)
potential_set.addPotential(angle_potential)
# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the HarmonicAnglePotential block of the script is shown. The full script can be found found in the file `ethane_bonded_potential.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- An angle potential is calculated for a triple *(a,b,c)* of atoms, which are connected by two adjacent bonds *(a,b)* and *(b,c)*. The potential acts on the angle θ formed by these two bonds:

$$\theta(\mathbf{r}_{a,b}, \mathbf{r}_{b,c}) = \arccos\left(\frac{\mathbf{r}_{a,b} \cdot \mathbf{r}_{c,b}}{r_{a,b} r_{c,b}}\right)$$

where $\mathbf{r}_{i,j}$, $\{i,j\} \subset \{a,b,c\}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ the corresponding bond distance.

The potential is calculated as

$$V(\theta) = k\,(\theta - \theta_0)^2$$

HarmonicBondPotential

class HarmonicBondPotential(*particleType1*, *particleType2*, *k*, *r_0=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **k** (*PhysicalQuantity of type energy / length^2*) – Potential parameter (strength of the bond).

•**r_0** (*PhysicalQuantity of type length*) – Potential parameter (equilibrium distance).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a HarmonicBondPotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethane_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=-0.3*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)
# Set up a new bond potential for C-C bonds and add it to the potential set.
bond_potential = HarmonicBondPotential(
    particleType1='C',
    particleType2='C',
    k=13.44287*eV/Ang**2,
    r_0=1.526*Ang,
)
potential_set.addPotential(bond_potential)

# Set up a new bond potential for C-H bonds and add it to the potential set.
bond_potential = HarmonicBondPotential(
    particleType1='C',
    particleType2='H',
    k=14.74379*eV/Ang**2,
    r_0=1.090*Ang,
)
potential_set.addPotential(bond_potential)
```

(continues on next page)

(continued from previous page)

```
# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the HarmonicBondPotential block of the script is shown. The full script can be found found in the file `ethane_bonded_potential.py`.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This bond potential is calculated as

$$V(r) = k\,(r - r_0)^2$$

where r is the distance between the atoms connected by this bond and r_0 is the equilibrium distance.

HarmonicChargedPointDefect

class HarmonicChargedPointDefect(*charged_point_defect*, *relax_atomic_coordinates=None*, *phonon_calculator=None*, *dynamical_matrix_processes_per_task=None*, *assumed_formation_entropy=None*, *resume=None*)

Constructor for the HarmonicChargedPointDefect object.

Parameters

•**charged_point_defect** (*ChargedPointDefect*) – The charged point defect study for which the phonon contributions will be calculated. The study must only contain one supercell repetitions value. The atomic partial entropies used for calculating the entropy contributions will be extracted for each element from the `atomic_chemical_potentials` of the charged point defect object.

•**relax_atomic_coordinates** (*bool*) – Whether to relax the atomic coordinates of the defect supercells. May be needed if configurations in the ChargedPointDefect were not fully relaxed. **Default:** `True`

•**phonon_calculator** (*Calculator*) – The calculator used for performing phonon calculations. The calculator must contain a basis set for the elements in the defect configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration` in the `charged_point_defect` study; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `relaxation_calculator` in the `charged_point_defect` study.

•**dynamical_matrix_processes_per_task** (int | `AllProcesses` | *ProcessesPerNode*) – The number of processes that will be used to execute each task for DynamicalMatrix study objects. If this value is greater than or equal to the total number of available processes, each single task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. Dynamical calculations are run in serial, independently on the tasks and not affected by those. In other words, if there is more than 1 phonon calculation each is run in serial

versus the others, but each one uses this parameter to allow internal parallelism (by computing the displacements in parallel). **Default:** `AllProcesses`, all available processes execute each task collaboratively.

- **assumed_formation_entropy** (*PhysicalQuantity* of type entropy) – The assumed formation entropy of the defect associated with this HarmonicChargedPointDefect object. Setting this parameter will disable all phonon contributions. **Default:** Phonon contributions will be calculated.
- **resume** (*bool*) – Whether reruning the simulation will resume non converged results **Default:** False

assumedFormationEntropy()

Returns
The assumed formation entropy of the defect associated with this HarmonicChargedPointDefect object. If None, the phonon contributions are calculated explicitly.

Return type
PhysicalQuantity | None

atomNumberDifferences()

Returns
The atom number differences for each element.

Return type
dict of type {*PeriodicTableElement* : int}

atomicChemicalPotentials()

Returns
The atomic chemical potential for all the required elements. If an element for which the atomic chemical potential needs to be calculated is not present, it is calculated from the pristine cell at each supercell size.

Return type
list of *AtomicChemicalPotential*

chargeStates()

Returns
The list of all calculated and not calculated charge states. Any charge state which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of int

chargedPointDefect()

Returns
The charged point defect study, for which the harmonic corrections will be calculated.

Return type
ChargedPointDefect

defectConfiguration(*supercell_repetitions*, *charge_state*, *symmetry_index=None*, *unit_cell_translation=None*)

Retrieve the defect supercell configuration for a given supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

- **symmetry_index** (*int*) – The index of the symmetry operation for the pristine unit cell to apply to the defect supercell configuration. Note that this can only be used if the calculate_defect_symmetry option has been enabled for the calculation. **Default:** No transformation (i.e., the identity operation)
- **unit_cell_translation** (*sequence (size 3) of int*) – An additional translation to apply in terms of the unit cell vectors. **Default:** No translation

Returns
The defect supercell configuration. If not available, returns None.

Return type
BulkConfiguration | None

defectSymmetry(*supercell_repetitions*, *charge_state*)

Calculate the symmetry information for the defect supercell.

This result will only be available after the calculation ran successfully.

Parameters

- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **defect_charge_state** (*int*) – The charge state of the defect.

Returns
The symmetry information for the defect supercell. If not available, returns None.

Return type
dict | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

dynamicalMatrixProcessesPerTask()

Returns
The number of processes that will be used to execute each task for DynamicalMatrix study objects.

Return type
int

filename()

Returns
The filename where the study object is stored.

Return type
str

formationEnergy(*charge_state*, *electronic_chemical_potential=None*, *electronic_chemical_potential_reference=None*, *defect_type=None*, *supercell_repetitions=None*, *enable_finite_size_corrections=None*, *temperature=None*)

Determines the formation energy with harmonic corrections for a given charge state q of the defect, which is defined as

$E_f^{\mathrm{harmonic},q} = E_f^q + E_{\mathrm{vib}}^{\mathrm{def}} - E_{\mathrm{vib}}^{\mathrm{pr}}$,

where the first term is the formation energy in the given `charged_point_defect` study, and the second and third terms correspond to the defect and pristine vibrational energies, respectively.

The vibrational energy is defined as

$E_{\mathrm{vib}} = G + TS,$

where *G* is the vibrational free energy, *T* is the temperature and *S* is the vibrational entropy.

If `assumed_formation_entropy` has been specified, the vibrational energies wont be included, i.e., the method returns E_f^q from the `charged_point_defect` object.

Parameters

- **`charge_state`** (*int*) – The charge state of the defect.
- **`electronic_chemical_potential`** (*PhysicalQuantity* of type energy) – The electronic chemical potential relative to `electronic_chemical_potential_reference`. **Default:** `0.0 * eV`
- **`electronic_chemical_potential_reference`** (`ValenceBandEdge | ConductionBandEdge`) – The reference level for the electronic chemical potential. **Default:** `ValenceBandEdge`
- **`defect_type`** (`DeepLevelDefect | ShallowAcceptor | ShallowDonor`) – The assumption of the type of defect to use for aligning the transition levels in the band gap calculated with `band_gap_calculator`; see the documentation of *ChargedPointDefect* for more details. Note that this parameter only has an effect if `band_gap_calculator` is different from `formation_energy_calculator`. **Default:** `DeepLevelDefect`
- **`supercell_repetitions`** (sequence (size 3) of int | *ExtrapolationScheme*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions. If ExtrapolationScheme, the extrapolation to the infinite supercell size limit is used. **Default:** The largest calculated supercell.
- **`enable_finite_size_corrections`** (*bool*) – Whether to include finite size corrections for the supercell in the calculation of the formation energy. **Default:** `True`
- **`temperature`** (*PhysicalQuantity* of type temperature) – The temperature used for calculating the formation energy. **Default:** `300.0 * Kelvin`

Returns
The calculated formation energy.

Return type
PhysicalQuantity of type energy

`formationEntropy`(*temperature=None*)

Determines the formation entropy for a given defect, defined as

$S_f^{\mathrm{harmonic}} = S_{\mathrm{vib}}^{\mathrm{def}} - S_{\mathrm{vib}}^{\mathrm{pr}} - \sum_i \Delta n_i s_i$

where the first and second terms correspond to the defect and pristine entropies, respectively, and the third term is the sum of the atomic partial entropies.

If `assumed_formation_entropy` has been specified, the method returns that value instead.

Parameters
`temperature` (*PhysicalQuantity* of type temperature) – The temperature used for calculating the formation entropy. **Default:** `300.0 * Kelvin`

Returns
The calculated formation entropy, or the user-specified entropy if `assumed_formation_entropy` has been given.

Return type
PhysicalQuantity of type entropy

isUpdated()

Returns whether the defect was successfully updated

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometryParameters()

Returns
Parameters required to optimize the geometry.

Return type
OptimizeGeometryParameters

phononCalculator()

Returns
The calculator which will be used for determining the phonon contributions.

Return type
Calculator

pointDefect()

Returns
The point defect associated with the charged point defect study.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

pristineSymmetryData()

Retrieve the symmetry information for the pristine unit cell.

This result will only be available after the calculation ran successfully.

Returns
The symmetry information for the pristine unit cell. If not available, returns None.

Return type
dict | None

resume()

Returns
Whether non converged simulations are to be resumed

Return type
bool

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

supercellRepetitionsList()

Returns
The list of all calculated and not calculated supercells of the bulk unit cell, each given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. Any supercell which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of tuple (size 3) of int

tasksFinished()

Returns
finished task, unfinished tasks

Return type
tuple of list of str, list of str

uniqueString()

Return a unique string representing the state of the object.

update()

Performs the calculations for the harmonic charged point defect study, i.e., the dynamical matrix calculations for the pristine and defect configurations and the corresponding phonon density of states. If `assumed_formation_entropy` is specified, phonon contributions are assumed to be fixed and wont be calculated.

The defect phonon density of states is calculated for the supercell with a zero charge state.

HarmonicCosineAnglePotential

class HarmonicCosineAnglePotential(*particleType1*, *particleType2*, *particleType3*, *k*, *cosTheta0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **k** (*PhysicalQuantity of type energy*) – Potential parameter.
- **cosTheta0** (*float*) – Potential parameter (cosine of the equilibrium angle)

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a HarmonicCosineAnglePotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethane_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=-0.3*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)

# Set up a new cosine angle potential for H-C-C angles and add it to the potential set.
angle_potential = HarmonicCosineAnglePotential(
    particleType1='H',
```

(continues on next page)

(continued from previous page)

```
    particleType2='C',
    particleType3='C',
    k=2.1682*eV,
    cosTheta0=-0.3338,
)
potential_set.addPotential(angle_potential)

# Set up a new cosine angle potential for H-C-H angles and add it to the potential set.
angle_potential = HarmonicCosineAnglePotential(
    particleType1='H',
    particleType2='C',
    particleType3='H',
    k=1.51774*eV,
    cosTheta0=-0.3338,
)
potential_set.addPotential(angle_potential)

# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the HarmonicCosineAnglePotential block of the script is shown. The full script can be found found in the file `ethane_bonded_potential_cosine_angle.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- An angle potential is calculated for a triple *(a,b,c)* of atoms, which are connected by two adjacent bonds *(a,b)* and *(b,c)*. The potential acts on the angle θ formed by these two bonds:

$$\theta(\mathbf{r}_{a,b}, \mathbf{r}_{b,c}) = \arccos\left(\frac{\mathbf{r}_{a,b} \cdot \mathbf{r}_{c,b}}{r_{a,b} r_{c,b}}\right)$$

where $\mathbf{r}_{i,j}$, $\{i,j\} \subset \{a,b,c\}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ the corresponding bond distance.

The potential is calculated as

$$V(\theta) = \frac{k}{2}\left(\cos(\theta) - \cos(\theta_0)\right)^2$$

where θ_0 is the equilibrium angle.

HarmonicUreyBradleyAnglePotential

class HarmonicUreyBradleyAnglePotential(*particleType1*, *particleType2*, *particleType3*, *k*, *theta0*, *k_ub*, *S_0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **k** (*PhysicalQuantity of type energy / angle**2*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter (the equilibrium angle).
- **k_ub** (*PhysicalQuantity of type energy / length**2*) – Potential parameter.
- **S_0** (*PhysicalQuantity of type length*) – Potential parameter (the equilibrium distance)

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a HarmonicUreyBradleyAnglePotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='water_urey_bradley')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Oxygen)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen)
)
```

(continues on next page)

(continued from previous page)

```
# Set up a new Urey-Bradley angle potential for H-O-H angles and add it to the potential␣
↪set.
angle_potential = HarmonicUreyBradleyAnglePotential(
    particleType1='H',
    particleType2='O',
    particleType3='H',
    k=4.33641*eV,
    theta0=1.824218,
    k_ub=5.0*eV/Ang**2,
    S_0=1.54*Ang,
)
potential_set.addPotential(angle_potential)

# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the HarmonicUreyBradleyAnglePotential block of the script is shown. The full script can be found found in the file `ethane_bonded_potential_urey_bradley.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- The angle potential is calculated for a triple *(a,b,c)* of atoms, which are connected by two adjacent bonds *(a,b)* and *(b,c)*. The potential acts on the angle θ formed by these two bonds:

$$\theta(\mathbf{r}_{a,b}, \mathbf{r}_{b,c}) = \arccos\left(\frac{\mathbf{r}_{a,b} \cdot \mathbf{r}_{c,b}}{r_{a,b} r_{c,b}}\right)$$

where $\mathbf{r}_{i,j}$, $\{i,j\} \subset \{a,b,c\}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ the corresponding bond distance.

This angle potential is similar to the *HarmonicAnglePotential*, but includes another term that depends on the distance of the outer atoms *(a,c)* The potential is calculated as

$$V(\theta) = k\left(\theta - \theta_0\right)^2 + k^{UB}\left(r_{a,c} - S_0\right)^2$$

where θ is the angle formed between *(a,b)* and *(b,c)* and θ_0 is the equilibrium angle.

HartreeDifferencePotential

class HartreeDifferencePotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the Hartree difference potential for a configuration.

Parameters

configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration for which the Hartree difference potential should be calculated.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

•**x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.

•**y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.

•**z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.

•**spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy Œ length^{-1}

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **`downsampling_a`** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **`downsampling_b`** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **`downsampling_c`** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **`x`** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **`y`** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **`z`** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **`spin`** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **`i`** (*int*) – The grid index in the A direction.
- **`j`** (*int*) – The grid index in the B direction.
- **`k`** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the Hartree difference potential is calculated for, always `Spin.All`.

Return type
`Spin.All`

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the Hartree difference potential and save it to a file:

```
# Set up a configuration.
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.       ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define a calculator.
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the Hartree difference potential.
potential = HartreeDifferencePotential(molecule_configuration)
nlsave('hartree_difference_pot.nc', potential)
```

nh3_hartree_difference_potential.py

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the Hartree difference potential $\delta V_H(\mathbf{r})$, as defined in *The Hartree potential and the electrostatic potential*.

HartreePotential

class HartreePotential(*configuration*, *density_mesh_cutoff=None*)

A class for calculating the Hartree potential for a configuration.

Parameters

•**configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration for which the Hartree potential should be calculated.

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** The density mesh cutoff set on the calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

•**axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c

•**spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All

•**projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.

•**coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

`derivatives`(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The gradient at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy Œ length^{-1}

`downsample`(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`evaluate`(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type energy

`gridCoordinate`(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the Hartree potential is calculated for, always Spin.All.

Return type
Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the Hartree potential and save it to a file:

```
# Set up a configuration.
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )
```

(continues on next page)

(continued from previous page)

```
# Define a calculator.
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the Hartree potential.
potential = HartreePotential(molecule_configuration)
nlsave('hartree_pot.nc', potential)
```

nh3_hartree_potential.py

Read in the Hartree potential from a file and calculate the average along the z-axis:

```
# Import a HartreePotential object.
potential = nlread('hartree_pot.hdf5', HartreePotential)[0]

# Calculate the average along z axis.
v_z = numpy.apply_over_axes(numpy.mean, potential[:,:,:], [0,1]).flatten()

# Add unit.
v_z = v_z * potential.unit()

# Get the volume element of the grid.
dX, dY, dZ = potential.volumeElement()
dz = dZ.norm()

# Print out the result.
shape = potential.shape()
for i in range(shape[2]):
    print(dz*i, v_z[i])
```

nh3_hartree_potential_average.py

For more examples on working with 3D grids, see *ElectronDensity*.

Notes

•This class inherits from the *GridValues* class.

•Returns the Hartree potential $V_H(\mathbf{r})$, as defined below in *The Hartree potential and the electrostatic potential*.

The Hartree potential and the electrostatic potential

•The Hartree potential $V_H(\mathbf{r})$ is calculated from the Poisson equation as

$$\nabla^2 V_H[n](\mathbf{r}) = -\frac{e^2}{4\pi\epsilon_0} n(\mathbf{r}),$$

where $n(\mathbf{r})$ is the valence electron density, e is the unit charge, and ϵ_0 is the vacuum permittivity. More details on solving this equation are given in *Poisson solvers*.

•In a similar fashion, the Hartree difference potential is calculated from the electron difference density as

$$\nabla^2 \delta V_H[\delta n](\mathbf{r}) = -\frac{e^2}{4\pi\epsilon_0} \delta n(\mathbf{r}),$$

with the electron difference density $\delta n(\mathbf{r})$ being defined through the relation

$$n(\mathbf{r}) = \delta n(\mathbf{r}) + \sum_{I}^{N_{\mathrm{atoms}}} n_I(\mathbf{r}),$$

where $n_I(\mathbf{r})$ is the compensation charge of atom I and N_{atoms} is the number of atoms in the system.:footcite:*Soler2002*

Note that the electron density and electron difference density are calculated differently for *DFT: LCAO* calculators (see *Electron density*) and *Semi Empirical* calculators (see *Electron density*).

•The electrostatic potential is given by

$$V_{\mathbf{E}}(\mathbf{r}) = -\frac{V_H(\mathbf{r})}{e},$$

and the electrostatic difference potential by

$$\delta V_{\mathbf{E}}(\mathbf{r}) = -\frac{\delta V_H(\mathbf{r})}{e}.$$

Note that the Hartree potential and Hartree difference potential have units of energy (default: Hartree), while the electrostatic potential and electrostatic difference potential have units of electric potential (default: Volt). Furthermore, they are of opposite sign since $V_H(\mathbf{r})$ is the potential energy that an electron (i.e., a negative unit charge) has in an electrostatic potential $V_{\mathbf{E}}(\mathbf{r})$.

The quantities defined above can be calculated with the following classes:

•$n(\mathbf{r})$: *ElectronDensity*

•$\delta n(\mathbf{r})$: *ElectronDifferenceDensity*

•$V_H(\mathbf{r})$: *HartreePotential*

•$\delta V_H(\mathbf{r})$: *HartreeDifferencePotential*

•$V_{\mathbf{E}}(\mathbf{r})$: *ElectrostaticPotential*

•$\delta V_{\mathbf{E}}(\mathbf{r})$: *ElectrostaticDifferencePotential*

HeisenbergExchange

class HeisenbergExchange(*configuration*, *kpoints=None*, *atom_indices=None*, *exchange_interaction_range=None*, *number_of_contour_points=None*, *integral_lower_bound=None*, *precalculate_eigenstates=None*)

Class for calculating the inter-site Heisenberg exchange coupling matrix for a BulkConfiguration.

Parameters

•**configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The BulkConfiguration for which to calculate the inter-site coupling matrix.

•**kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points with which to calculate the Heisenberg exchange constants. **Default:** *MonkhorstPackGrid(1, 1, 1)*

•**atom_indices** (`All` | list of ints.) – The atom indices for which to calculate the Heisenberg exchange constants. **Default:** `All` i.e. all atoms in the configuration.

•**exchange_interaction_range** (*list of ints*) – The range of the included interactions. Interactions are calculated between the primitive cell of the input configuration and translated cells given by the exchange_interaction_range. The exchange_interaction_range determines the number of translations of the configuration in the A, B, and C-directions given as a list of three positive integers, e.g. `[3, 3, 3]`. For a `MoleculeConfigurtion` `[1, 1, 1]` will always be used. **Default:** `[5, 5, 5]` for *BulkConfiguration* and [1, 1, 1] for a *MoleculeConfiguration*

- **number_of_contour_points** (*int >= 2*) – The number of points on the circle contour. **Default:** 30
- **integral_lower_bound** (*PhysicalQuantity* of type energy) – The distance between the lowest Fermi-level and the lowest energy circle contour point. **Default:** An energy determined on the chosen pseudopotentials or 1.5 Hartree for the semi-empirical calculators.
- **precalculate_eigenstates** (*bool*) – Boolean determining if the eigenstates and energies at all the k-points should be pre-calculated and stored for the real-space Greens functions. Setting this option to True will reduce the computation time, but requires more memory. **Default:** False

DMIVectors()

Returns
The Dzyaloshinskii-Moria interaction (DMI) vectors in the configuration. The DMI is only calculated if spin-orbit coupling is included. The shape of the array is [n_translations, n_atoms, n_atoms, 3], where n_translations is the number of translations, given by the product of the repetitions,

Return type
PhysicalQuantity of type energy.

atomIndices()

Returns
The atom indices.

Return type
list of int.

atomSpins()

Returns
The atomic spin polarizations.

Return type
list of floats.

contour()

Returns
The contour method to be used.

Return type
OzakiContour | *SemiCircleContour*

couplingMatrix()

Returns
The Heisenberg exchange coupling matrix corresponding the model $H_{ij} = J_{ij}\hat{e}_i \cdot \hat{e}_j$. The shape of the array is [*n_translations*, *n_atoms*, *n_atoms*], where *n_translations* is the number of translations, given by the product of the repetitions, and *n_atoms* is the number of atoms in the configuration.

Return type
PhysicalQuantity of type energy.

curieTemperature()

Calculates the Curie temperature based on the mean field approximation, $T_c k_B = 2J_0/3$, where $J_0 = \sum_i J_{0i}$.

Returns
The calculated Curie temperature.

Return type
PhysicalQuantity of type temperature.

curieTemperatureRPA(*q_grid=None*, *r_max=None*)

Calculates the Curie temperature based on the random phase approximation (RPA), $\frac{1}{T_c k_B} = \frac{6\mu_B}{M}\sum_q \frac{1}{E(q)}$, where where $E(q)$ is the magnon energy, and M is the magnetization.

Parameters

- **q_grid** (*MonkhorstPackGrid*) – The q-points used for the RPA calculation. **Default:** MonkhorstPackGrid(8,8,8)
- **r_max** (*PhysicalQuantity* of type length.) – Maximum radius to include in the sum. **Default:** Include all calculated interactions.

Returns
The calculated Curie temperature.

Return type
PhysicalQuantity of type temperature.

distances()

Returns
The distance between atoms corresponding to the coupling matrices. The shape of the array is [*n_translations*, *n_atoms*, *n_atoms*], where *n_translations* is the number of translations, given by the product of the repetitions, and *n_atoms* is the number of atoms in the configuration.

Return type
PhysicalQuantity of type length.

exchangeInteractionRange()

Returns
The interaction range of the configuration in the A, B, and C directions.

Return type
list of int.

exchangeStiffness(*r_max=None*, *damping_factor=None*, *magnetic_tolerance=None*, *atom_indices=None*)

Calculate the exchange stiffness constant as $A_{ex} = \frac{2\mu_B}{3M}\sum_i J_{0i}\cdot R_{0i}^2 \cdot e^{-d\cdot R_{0i}}$, where M is the magnetization and d is a damping factor. If a damping factor is used, the actual exchange stiffness should be extrapolated to zero damping factor, as described in DOI:10.1103/PhysRevB.64.174402.

Parameters

- **r_max** (*PhysicalQuantity* of type length.) – Maximum radius to include in the sum. **Default:** Include all calculated radii.
- **damping_factor** (*PhysicalQuantity* of type inverse length.) – Damping factor to be added to the sum. **Default:** 0*Ang**-1.
- **magnetic_tolerance** (*float.*) – Only include contribution from atoms with a magnetic moment larger than the magnetic tolerance. **Default:** 0.1
- **atom_indices** (*list of int*) – Include contributions to the exchange stiffness from atoms in atom_indices and their interaction with all the others. **Default:** All atom_indices defined upon construction

Returns
The calculated exchange stiffness.

Return type
PhysicalQuantity of type energy * distance^2.

extrapolatedExchangeStiffness(*damping_factor_range=None*, *r_max=None*, *polynomial_order=None*, *magnetic_tolerance=None*, *atom_indices=None*)

Calculate the extrapolated exchange stiffness constant as $\lim_{\eta\to 0} \frac{2\mu_B}{(3M} \sum_i J_{0i} R_{0i}^2 e^{-\eta R_{0i}}$, where M is the magnetization and η is a damping factor. If a damping factor is used, the actual exchange stiffness should be extrapolated to zero damping factor, as described in DOI:10.1103/PhysRevB.64.174402.

Parameters

- **damping_factor_range** – List of damping factors used to extrapolate to zero damping_factor. The list must contain five or more entities. **Default:** A range of damping factors ranging from *0.1/d_0* to *1/d_0*, where *d_0* is the smallest nearest neighbor distance.
- **r_max** (*PhysicalQuantity* of type length.) – Maximum radius to include in the sum. **Default:** Include all calculated radii.
- **polynomial_order** (*int.*) – Order of the polynomial used to extrapolate to zero damping. **Default:** 5
- **magnetic_tolerance** (*float.*) – Only include contribution from atoms with a magnetic moment larger than the magnetic tolerance. **Default:** 0.1
- **atom_indices** (*list of int*) – Include contributions to the exchange stiffness from atoms in atom_indices and their interaction with all the others. **Default:** All atom_indices defined uppon construction

Returns
The calculated exchange stiffness extrapolated to zero damping factor, a list of exchange stiffness values for each damping factor, the list of damping factors used, and the polynomial fit.

Return type
PhysicalQuantity of type energy * distance^2, *PhysicalQuantity* of type energy * distance^2, *PhysicalQuantity* of type inverse length, numpy.polyfit

fourierTransformCouplings(*qpoint*, *r_max=None*, *damping_factor=None*)

Calculate the Fourier transform of the J_ij matrix.

Parameters

- **qpoint** (*list of three floats.*) – Fractional q-point
- **r_max** (*PhysicalQuantity* of type inverse length.) – Maximum radius to include in the sum. **Default:** Include all calculated interactions.
- **damping_factor** – Exponential damping factor. A factor of exp(-damping_factor*R) is multiplied to the fourier transform. **Default:** Zero.

Returns
The fourier transformed matrix.

Return type
PhysicalQuantity of type energy.

kpoints()

Returns
The k-points with which to calculate the Heisenberg exchange constants.

Return type
class:*~.MonkhorstPackGrid* | *RegularKpointGrid*

magnonBandstructure(*qpoints=None*, *points_per_segment=None*, *route=None*, *r_max=None*, *magnetic_tolerance=None*)

Parameters

- **q_path** (*list of list of floats*) – List of fractional q-points. This option is mutually exclusive to `q_route` and `points_per_segment`. The shape of the list is (number of q-points, 3), e.g. `[[0.0, 0.0, 0.0], [0.25, 0.0, 0.0], [0.5, 0.0, 0.0]]`
- **route** (*list of str*) – The route to take through the Brillouin zone as a list of symmetry points of the unit cell, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `q_path`.
- **points_per_segment** (*int*) – The number of points per segment of the route. This option is mutually exclusive to `q_path`. **Default:** 20
- **r_max** (*PhysicalQuantity* of type length.) – Maximum interaction radius to include in the calculation. **Default:** Include all calculated interactions.
- **magnetic_tolerance** (*float.*) – Only include contribution from atoms with a magnetic moment larger than the magnetic tolerance. **Default:** 0.1

Returns
A bandstructure object which can be saved and visualized.

Return type
PhononBandstructure

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

scaledCouplingMatrix()

Returns the coupling matrix elements scaled with the atomic spin polarizations. This corresponds to the matrix elements from the model $H_{ij} = J_{ij}\hat{S}_i \cdot \hat{S}_j$.

Returns
The coupling matrix elements. The shape of the array is [*n_translations*, *n_atoms*, *n_atoms*], where *n_translations* is the number of translations, given by the product of the repetitions, and *n_atoms* is the number of atoms in the configuration.

Return type
PhysicalQuantity of type energy

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spinComponentMatrices()

Returns

Returns spin-component A-matrices as defined in e.g. Computer Physics Communications 264 (2021) 107938, eq. (14).

The shape of the array is [4, 4, *n_translations*, *n_atoms*, *n_atoms*], where the order of the first two indices is 0, x, y, z components, and where *n_translations* is the number of translations, given by the product of the repetitions, and *n_atoms* is the number of atoms in the configuration.

The A-matrices are not calculate for MoleculeConfigurations.

Return type
PhysicalQuantity of type energy.

translations()

Returns
The list of translations to be used.

Return type
list of lists of int.

uniqueCouplingMatrixElementsAndDistances(*atom_index=None*, *tolerance=None*)

Parameters

- **atom_index** (*int*) – The index of the atom for which to get the distances and coupling matrix elements.
- **tolerance** (*PhysicalQuantity* of type energy) – Energy tolerance for when two matrix elements are considered equal. **Default:** 0.01 meV

Returns
Returns lists of distances and coupling matrix elements with unique elements as well as the multiplicity of the unique distances.

Return type
PhysicalQuantity of type length, *PhysicalQuantity* of type energy, numpy.array of ints.

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Calculate Heisenberg exchange coupling for Fe (BCC):

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = BodyCenteredCubic(2.8665*Angstrom)

# Define elements
elements = [Iron]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#--------------------------------------
# Exchange-Correlation
#--------------------------------------
exchange_correlation = SGGA.PBE

k_point_sampling = KpointDensity(
    density_a=7.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=120.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('Fe-BCC.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Heisenberg Exchange
# ------------------------------------------------------------
kpoints = KpointDensity(
    density_a=12.0*Angstrom,
    )

heisenberg_exchange = HeisenbergExchange(
    configuration=bulk_configuration,
    kpoints=kpoints,
    atom_indices=All,
    exchange_interaction_range=(9, 9, 9),
    number_of_contour_points=30,
    integral_lower_bound=Automatic,
    precalculate_eigenstates=False,
    )
nlsave('Fe-BCC.hdf5', heisenberg_exchange)
```

heisenberg_exchange.py

Notes

The Heisenberg exchange Hamiltonian is given by[1],[2]

$$H = -\sum_{i \neq j} J_{ij} \hat{e}_i \hat{e}_j \tag{4.1}$$

where $\hat{e}_i$ is the normalized local spin vector on atom i and J_{ij} is the Heisenberg exchange coupling constant.

We calculate the exchange coupling matrix elements $J_{i\mathbf{0},j\mathbf{R}}$ between a atom i in the central unit cell labeled $\mathbf{0}$ and atom j in a different unit cell displaced from the central one with a lattice vector $\mathbf{R}$. We use a real-space Greens function implementation[3],[4] of the original Liechtenstein-Katsnelson-Antropov-Gubanov (LKAG) formula[Page 3039, 1]:

$$J_{0i,\mathbf{R}j} = -\frac{1}{4\pi} \int_{-\infty}^{E_F} d\epsilon \mathrm{Im}\mathrm{Tr}\left[\Delta_i G_{i,j}(\epsilon, \mathbf{R}) \Delta_j G_{i,j}(\epsilon, -\mathbf{R})\right] \tag{4.2}$$

where i and j label atomic indices within a unit cell, $\mathbf{R}$ is a lattice vector, and where $\Delta_i = H_i^{\uparrow} - H_i^{\downarrow}$ is the on-site difference between the up and down part of the Hamiltonian matrix. The real-space Greens functions are calculated as

$$G(\epsilon, \mathbf{R}) = \sum_{\mathbf{k}_i} w_{\mathbf{k}_i} G(\mathbf{k}_i, \epsilon) e^{-i\mathbf{k}_i \cdot \mathbf{R}} \tag{4.3}$$

where $w_{\mathbf{k}_i}$ is the k-point weight and with the reciprocal space Greens function given by:

$$G(\mathbf{k}, \epsilon) = (\epsilon \mathbf{S}(\mathbf{k}) - \mathbf{H}(\mathbf{k}))^{-1}$$

The energy integral in (4.2) is calculated using complex contour integration.

Curie temperature

The Curie temperature, T_C, of a ferromagnet is the temperature at which the average magnetic moment becomes zero due to random fluctuations of the local magnetic moment directions. T_C can be estimated from the Heisenberg model in different ways. Within the mean field approximation (MFA) one have[2]

$$T_C^{MFA} = \frac{2}{3k_B} \sum_{j\mathbf{R} \neq i\mathbf{0}} J_{i\mathbf{0}j\mathbf{R}} \tag{4.4}$$

The MFA Curie temperature is calculated as

```
curie_temperature_mfa = heisenberg_exchange.curieTemperature()
```

The MFA formula typically overestimates the Curie temperature when compared to experiments. A different approach is based on the random phase approximation (RPA)[2]:

$$\frac{1}{k_B T_C^{RPA}} = \frac{6\mu_B}{M} \frac{1}{N_q} \sum_{\mathbf{q}} \frac{1}{E(\mathbf{q})} \tag{4.5}$$

[1] A.I. Liechtenstein, M.I. Katsnelson, V.P. Antropov, and V.A. Gubanov. Local spin density functional approach to the theory of exchange interactions in ferromagnetic metals and alloys. *Journal of Magnetism and Magnetic Materials*, 67(1):65–74, 1987. URL: http://www.sciencedirect.com/science/article/pii/0304885387907219, doi:https://doi.org/10.1016/0304-8853(87)90721-9.

[2] M.ăPajda, J.ăKudrnovský, I.ăTurek, V.ăDrchal, and P.ăBruno. Ab-initio calculations of exchange interactions, spin-wave stiffness constants, and Curie temperatures of Fe, Co, and Ni. *Phys. Rev. B*, 64:174402, Oct 2001. URL: https://link.aps.org/doi/10.1103/PhysRevB.64.174402, doi:10.1103/PhysRevB.64.174402.

[3] Asako Terasawa, Munehisa Matsumoto, Taisuke Ozaki, and Yoshihiro Gohda. Efficient Algorithm Based on Liechtenstein Method for Computing Exchange Coupling Constants Using Localized Basis Set. *Journal of the Physical Society of Japan*, 88(11):114706, 2019. URL: https://doi.org/10.7566/JPSJ.88.114706, doi:10.7566/JPSJ.88.114706.

[4] XuăHe, Nicole Helbig, MatthieuăJ. Verstraete, and Eric Bousquet. TB2J: A python package for computing magnetic interaction parameters. *Computer Physics Communications*, 264:107938, 2021. URL: https://link.aps.org/doi/10.1103/PhysRevB.64.174402, doi:10.1103/PhysRevB.64.174402.

where $E(\mathbf{q})$ is the spin wave energy at q-point $\mathbf{q}$ and M is the magnetic moment per atom and μ_B is the Bohr magneton. The RPA Curie temperature is calculated as

```
q_grid = MonkhorstPackGrid(10,10,10)
curie_temperature_rpa = heisenberg_exchange.curieTemperatureRPA(q_grid=q_grid)
```

Since an average over the Brillouin zone is performed, the RPA calculation needs to be converged with respect to the number of q-points.

Exchange stiffness

The exchange stiffness describes the energy of a long wavelength spin wave excitation. In particular, if the energy as function of wave number q is written as

$$E(q) = E_0 + A_{ex}q^2 + \ldots$$

the exchange stiffness constant, A_{ex} is seen to be given by the curvature of the $E(q)$ vs. q curve. A_{ex} can be calculated from the Heisenberg exchange parameters as[2]

$$A_{ex} = \frac{2\mu_B}{3M}\sum_{\mathbf{R}}\sum_{ij} J_{i\mathbf{0},j\mathbf{R}} r^2_{i\mathbf{0},j\mathbf{R}} \tag{4.6}$$

where $r_{i\mathbf{0},j\mathbf{R}}$ is the distance between atom i in cell $\mathbf{0}$ and atom j in cell $\mathbf{R}$. This can be calculated as

```
exchange_stiffness = heisenberg_exchange.exchangeStiffness()
```

In practice, (4.6) can be difficult to converge with respect to the sum over lattice vectors $\mathbf{R}$. This is particularly pronounced in the case of iron where the exchange parameters are very long ranged. In this case it is possible to dampen the oscillations and extrapolate the result to zero damping as:

$$\begin{aligned} A_{ex} &= \lim_{\eta\to 0} A_{ex}(\eta) \\ A_{ex}(\eta) &= \frac{2\mu_B}{3M}\sum_{\mathbf{R}}\sum_{ij} J_{i\mathbf{0},j\mathbf{R}} r^2_{i\mathbf{0},j\mathbf{R}} e^{-\eta r_{i\mathbf{0},j\mathbf{R}}} \end{aligned} \tag{4.7}$$

The extrapolation to $\eta \to 0$ can be calculated as

```
extrapolated_exchange_stiffness, exchange_stiffness_vec, damping_factor_range, p_fit \
    = heisenberg_exchange.extrapolatedExchangeStiffness()
```

where the final extrapolated result is given by *extrapolated_exchange_stiffness*. The other returned quantities are a vector of calculated exchange stiffness values obtained for the damping factors given by *damping_factor_range*. Finally the polynomial fit is also returned.

Spin-scaling

The Heisenberg exchange Hamiltonian is sometimes written as

$$H = -\sum_{i\neq j} \tilde{J}_{ij}\hat{S}_i\hat{S}_j$$

where $\hat{S}_i$ is the local spin vector on atom i and J_{ij} is the Heisenberg exchange coupling constant. Following[Page 3039, 1 ,5], in QuantumATK we calculate the coupling constants for the Hamiltonian

$$H = -\sum_{i \neq j} J_{ij} \hat{e}_i \hat{e}_j$$

where the spin operators are substituted by unit vectors $\hat{e}_i$ pointing in the direction of the ith site magnetization.

Both the coupling constants J_{ij} and $\tilde{J}_{ij}$ are available in a *HeisenbergExchange* object and are simply relates as

$$\tilde{J}_{ij} = \frac{J_{ij}}{S_i \cdot S_j}$$

where S_i is calculated from the *MullikenPopulation* on each atom as $S_i = (M_i^{up} - M_i^{down})/2$ where M_i^{up} is the Mulliken population for up-electrons on atom i.

The coupling constants J_{ij} are obtained as:

```
coupling_matrix = heisenberg_exchange.couplingMatrix()
```

and the coupling constants $\tilde{J}_{ij}$ are obtained as:

```
scaled_coupling_matrix = heisenberg_exchange.scaledCouplingMatrix()
```

HookFunctions

class HookFunctions(*pre_step_hooks=None*, *post_step_hooks=None*, *measurement_hooks=None*)

Class to store and manage all hook functions for a dynamics calculation (MD or TS-FBMC).

Parameters

- **pre_step_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.
- **post_step_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.
- **measurement_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called at the end of each MD step after all constraints have been applied. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and can be accessed using the measurement method. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.

[5] MyungăJoon Han, Taisuke Ozaki, and Jaejun Yu. Electronic structure, magnetic interactions, and the role of ligands in $\mathrm Mn_n(n=4,12)$ single-molecule magnets. *Phys. Rev. B*, 70:184421, 2004. URL: https://link.aps.org/doi/10.1103/PhysRevB.70.184421, doi:10.1103/PhysRevB.70.184421.

addMeasurementHooks(*hook_functions*)

Add a measurement hook function or functions.

Parameters
hook_functions (*callable | list of callable*) – A user-defined function or a list of functions which will be called at the end of each MD step after all constraints have been applied. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and can be accessed using the measurement method. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order.

addPostStepHooks(*hook_functions*)

Add a post step hook function or functions.

Parameters
hook_functions (*callable | list of callable*) – A user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order.

addPreStepHooks(*hook_functions*)

Add a pre-step hook function or functions.

Parameters
hook_functions (*callable | list of callable*) – A user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order.

allHooks()

Returns
All hook functions separated by hook types.

Return type
dict

anyMeasurementHooksRequireStress()

Check if any of the measurement hooks requires stress.

Returns
True, if any measurement hook requires stress.

Return type
bool

callAllHookFunctions(*hook_type*, *step*, *time*, *configuration*, *local_forces*, *stress*, *trajectory*, *md_quantities=None*)

Run through all hook functions, gather their required arguments and call them in the given order.

Parameters

- **hook_type** (*str*) – The hook type.
- **step** (*int*) – The current MD step number.
- **time** (*PhysicalQuantity* of type time) – The current simulation time.

- **configuration** (DistributedConfiguration | *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The current configuration.
- **local_forces** (*PhysicalQuantity* of type energy/length) – The local forces vector. For serial simulation this will be the same as the global vector.
- **stress** (*PhysicalQuantity* of type energy/length**3 | None) – The stress tensor or None if no stress calculation is carried out.
- **trajectory** (MDTrajectory) – The trajectory calculated so far.
- **md_quantities** (*MDQuantities*) – Optional, the MDQuantities object. **Default:** None.

Returns
Measurement values.

Return type
dict

hasMeasurementHooks()

Returns
True, if there are measurement hooks in the store.

Return type
bool

hasPostStepHooks()

Returns
True, if there are post-step hooks in the store.

Return type
bool

hasPreStepHooks()

Returns
True, if there are pre-step hooks in the store.

Return type
bool

measurementHooks()

Returns
The measurement hooks.

Return type
list of HookFunctionCall

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Create NLInfo.

Returns
The nlinfo.

Return type
dict

`postStepHooks()`

Returns
The post-step hooks.

Return type
list of HookFunctionCall

`preStepHooks()`

Returns
The pre-step hooks.

Return type
list of HookFunctionCall

`removeMeasurementHook(`*index*`)`

Remove a measurement hook function.

Parameters
index (*int*) – The index of the hook function to remove.

Returns
The removed hook function.

Return type
HookFunctionCall | None

`removePostStepHook(`*index*`)`

Remove a post-step hook function.

Parameters
index (*int*) – The index of the hook function to remove.

Returns
The removed hook function.

Return type
HookFunctionCall | None

`removePreStepHook(`*index*`)`

Remove a pre-step hook function.

Parameters
index (*int*) – The index of the hook function to remove.

Returns
The removed hook function.

Return type
HookFunctionCall | None

`setHookFunctions(`*pre_step_hooks=None*, *post_step_hooks=None*, *measurement_hooks=None*`)`

Set hook functions.

Note! This replaces any currently set hook functions.

Parameters

- **pre_step_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.
- **post_step_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.
- **measurement_hooks** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called at the end of each MD step after all constraints have been applied. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and can be accessed using the measurement method. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters

metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up a HookFunctions object with a StrainConfigurationHook as a pre-step hook, a custom hook function as a post-step hook and a measurement hook to measure the uniaxial stress and strain in all three Cartesian directions:

```
# -*- coding: utf-8 -*-
from .hook_functions_configuration import silver

# %% HookFunctions

# Pre-step hook functions
strain_configuration_hook = StrainConfigurationHook(
    strain_direction='xx', strain_rate=5.0 * 1 / ps, call_interval=10, strain_interval=1
)


# Post-step hook functions
def custom_hook(step, time, configuration, forces):
    call_interval = 10
    if step % call_interval != 0:
        return

    print(forces)
```

(continues on next page)

(continued from previous page)

```
# Measurement hooks
md_measurement = MDMeasurement(
    measurements=[
        'strain_xx',
        'strain_yy',
        'strain_zz',
        'stress_xx',
        'stress_yy',
        'stress_zz',
    ],
    call_interval=10
)


pre_step_hooks = [
    strain_configuration_hook,
]
post_step_hooks = [
    custom_hook,
]
measurement_hooks = [
    md_measurement,
]
hook_functions = HookFunctions(
    pre_step_hooks=pre_step_hooks,
    post_step_hooks=post_step_hooks,
    measurement_hooks=measurement_hooks,
)

nlsave('HookFunctions_results.hdf5', hook_functions)


# %% MolecularDynamics

method = NVEVelocityVerlet(
    initial_velocity=MaxwellBoltzmannDistribution(
        temperature=300.0 * Kelvin,
        remove_center_of_mass_momentum=False,
        random_seed=None,
        enforce_temperature=True,
    ),
    time_step=1.0 * fs,
)

md_trajectory = MolecularDynamics(
    configuration=silver,
    constraints=[],
    trajectory_filename='HookFunctions_results.hdf5',
    steps=5000,
    log_interval=100,
```

(continues on next page)

(continued from previous page)

```
    method=method,
    hook_functions=hook_functions,
)
last_image = md_trajectory.lastImage()


# %% TimeStampedForceBiasMonteCarlo

monte_carlo_method = ForceBiasMonteCarlo(
    temperature=300.0 * Kelvin,
    max_atom_displacement=0.1 * Angstrom,
    heating_rate=0.0 * Kelvin,
    random_seed=None,
    max_random_attempts=500,
)

tfmc_trajectory = TimeStampedForceBiasMonteCarlo(
    configuration=last_image,
    constraints=[],
    trajectory_filename='HookFunctions_results.hdf5',
    steps=5000,
    log_interval=100,
    method=monte_carlo_method,
    hook_functions=hook_functions,
)
last_image_1 = tfmc_trajectory.lastImage()
```

`hook_functions_configuration.py` `hook_functions.py`

For sake of clarity, the configuration and calculator are defined in a separate file. In the example, both a *MolecularDynamics* and a *TimeStampedForceBiasMonteCarlo* are instantiated to illustrate how the HookFunctions is passed to them respectively.

General

The HookFunctions class can be used to organize sets of pre-, post- and measurement hook functions, that can then be passed to a *MolecularDynamics* or *TimeStampedForceBiasMonteCarlo* function to be used in a dynamics simulation.

The pre-step hook is invoked immediately before each forces calculation during the MD loop, whereas the post-step hook is invoked immediately after the forces calculation. The measurement hook is invoked at the end of each MD step (see also the reference manual notes *MolecularDynamics*).

Each category of hook functions can consist of a list of functions. For instance, the pre-step hook can consist of a StrainConfigurationHook and two CustomHooks. The functions within each category are then called in the order they are given in the list.

A HookFunctions object can be saved to disk, and so can be created once and used/reused in different dynamics simulations.

The HookFunctions object is also supported in NanoLab, in the form of a Workflow Builder Block with the icon . It can be found under the **Optimization and Dynamics** category. The block requires an input configuration and provides a HookFunctions object.

Available hook functions

The following classes are available for use in scripting, though not all of them may be available in NanoLab:

- *ForceCappedEquilibration*
- *MDMeasurement*
- *NonEquilibriumHeatExchange*
- *NonEquilibriumMomentumExchange*
- *PlumedMetadynamics*
- *SteeredMolecularDynamics*
- *StrainConfigurationHook*

Custom hook functions

In principle, any callable object can be used as a custom hook function. Python `generator` objects are not allowed as hook functions. Examples of valid hook functions are:

Function

```
def customHookFunction(step, time, configuration):
    call_interval = 10
    if step % call_interval != 0:
        return

    ...

hook_functions = HookFunctions(pre_step_hooks=[customHookFunction])
```

For functions, the call interval can be included as part of the function body, if the hook should not be called at each step.

Callable class

```
class CustomHookCallable:
    def __call__(self, step, time, configuration):
        ...

    def callInterval(self):
        return 10

hook_functions = HookFunctions(pre_step_hooks=[CustomHookCallable])
```

For callable classes, the call interval can be provided by defining a `callInterval` method on the class that returns an integer, if the hook should not be called at each step.

Class method

```
class CustomHookClass:
    def hookMethod(self, step, time, configuration):
        ...
```

(continues on next page)

(continued from previous page)

```
    def callInterval(self):
        return 10

hook_functions = HookFunctions(pre_step_hooks=[CustomHookClass.hookMethod])
```

For class methods, there can also be a `callInterval` method on the class that returns the call interval as an integer.

Warning: It is not recommended to use class methods as these cannot be reconstructed when a saved HookFunctions object is read from disk. They can, however, be safely used if the custom hook class and HookFunctions are defined within the same script. If possible, opt to re-write the custom hook class with a `__call__` method instead.

Note! Any modification to the configuration, forces, or stress, must be made in-place, e.g:

```
def customHookFunction(step, time, configuration, forces):
    # Add a small constant forces to the forces vector of the first atom.
    forces[0] += [0.1, 0.1, 0.1] * eV / Ang
```

Because the pre-step hook is invoked immediately before each forces calculation during the MD loop, any modification to the forces or stress, e.g. adding an external force, should be made in a post-step hook, which is invoked after the forces calculation. If a modification is done in a pre-step hook, it would be overwritten by the calculated forces, and have no effect.

In contrast, any modification to the configuration, e.g. straining the cell, should be made in a pre-step-hook, so that the following forces calculation will be carried out on the modified configuration.

Input arguments

A custom hook function must take at least the following arguments:

- **step**
- **time**
- **configuration**

It may optionally take the following arguments:

- forces
- local_forces (for MPI-distributed MD simulations)
- stress
- trajectory

If it is to be used as a measurement hook, it may also optionally take the following arguments:

- temperature
- pressure
- potential_energy
- kinetic_energy
- md_quantities

`md_quantities` contains all of the other measurement types, and additionally has the methods `reservoirTemperature`, `reservoirPressure` and `timeStep`.

Any arguments not listed above (excluding `self`, `cls` for callable classes and class methods) will raise an exception when the HookFunctions object is instantiated.

Return value

In general, a hook function does not return a value. For pre- and post-step hooks any return value is simply neglected.

For measurement hooks, a `dict` can be returned, for which each entry in the dictionary is then stored as a separate measurement on the *MDTrajectory*, and would be automatically available as a plot when visualized in the MovieTool. In this way, custom measurement quantities can be introduced in a dynamics simulation.

Notes

- The keyword arguments `pre_step_hook` and `post_step_hook` for *MolecularDynamics* and a *TimeStampedForceBiasMonteCarlo* and `measurement_hook` for *MolecularDynamics* are also available for convenience of use. However, if a `hook_functions` is given, these keyword arguments will be ignored.
- A HookFunctions object must be saved explicitly. The *MolecularDynamics* and *TimeStampedForceBiasMonteCarlo* functions do not save the hook functions they use, irrespective of which keyword(s) were used to pass the functions.

HotbitDirectory

class HotbitDirectory(*path*)

Constructor for the Hotbit directory parser.

Parameters
path (*str*) – The path of the directory to read files from.

fillingMethod(*element*)

Parameters
element (*PeriodicTableElement*) – The element for which to query the filling method.

Returns
The method used for setting up the initial occupation for the given element.

Return type
`SphericalSymmetric | Anisotropic`

onsiteSpinOrbitSplit(*element*)

Query the onsite spin-orbit split energies for a given element.

Parameters
element (*PeriodicTableElement*) – The element for which to query the spin-orbit split energies.

Returns
The onsite spin-orbit split for this element.

Return type
PhysicalQuantity of type energy

onsiteSpinSplit(*element*)

Query the onsite spin-split energies for a given element.

Parameters
element (*PeriodicTableElement*) – The element for which to query the spin splits.

Returns
The onsite spin-split energies for this element as a square matrix of size *n* where *n* is the number of subshells of the given element.

Return type
PhysicalQuantity of type energy

pairPotentials(*configuration*)

Query function for getting the pair potentials.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration defining which elements to load.

Returns
The pair-potentials.

Return type
list of *PairPotential*

setFillingMethod(*element*, *filling_method*)

Set the method used for setting up the initial occupation for the given element.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the filling method.
- **filling_method** (`SphericalSymmetric` | `Anisotropic`) – The method used for setting up the initial occupation.

setOnsiteSpinOrbitSplit(*element*, *split*)

Set the onsite spin-orbit split energies for a given element in the configuration.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the spin-orbit splits.
- **split** (*PhysicalQuantity* of type energy) – The onsite spin-split energies for this element as a square matrix of size *n* where *n* is the number of subshells of the given element.

setOnsiteSpinSplit(*element*, *split*)

Set the onsite spin split energies for a given element in the configuration.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the spin splits.
- **split** (*PhysicalQuantity* of type energy) – The on-site spin-split energies for each subshell pair of the element.

setVacuumLevel(*element*, *vacuum_level*)

Set the energy shift of the vacuum level for a given element.

Parameters

- **element** (*PeriodicTableElement*) – The element for which to set the energy shift of the vacuum level.
- **vacuum_level** (*PhysicalQuantity* of type energy) – The energy shift of the vacuum level.

table(*configuration*)

Query function for getting the *SlaterKosterTable* representation of the parameter files in the directory.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration defining which elements to load.

Returns
A *SlaterKosterTable* instance representing the parameter files in the directory.

Return type
SlaterKosterTable

vacuumLevel(*element*)

Parameters
element (*PeriodicTableElement*) – The element for which to query the energy shift of the vacuum level.

Returns
The energy shift of the vacuum level for the given element.

Return type
PhysicalQuantity of type energy

Usage Examples

Set up a self-consistent tight-binding calculation using parameters in the format developed by the Hotbit consortium.

```
# Set up a graphene crystal
bulk_configuration = BulkConfiguration(
    bravais_lattice=Hexagonal(2.4612*Angstrom, 6.709*Angstrom),
    elements=[Carbon, Carbon],
    cartesian_coordinates=[[ 0.        ,  0.        ,  0.        ],
                           [ 1.2306    ,  0.71050145,  0.        ]]*Angstrom
    )

# setup slater-koster calculator with hotbit parameters

basis_set = HotbitDirectory("hotbit/standard")

pair_potentials = HotbitDirectory("hotbit/standard")

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(4, 4, 1) )

calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    pair_potentials=pair_potentials,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
bulk_configuration.setCalculator(calculator)

initial_velocity = MaxwellBoltzmannDistribution(
        temperature=1000.0*Kelvin
    )
```

(continues on next page)

(continued from previous page)

```
method = NVEVelocityVerlet(
        time_step=1.0*femtoSecond,
        initial_velocity=initial_velocity
    )

molecular_dynamics = MolecularDynamics(
        bulk_configuration,
        constraints=[],
        trajectory_filename='trajectory.nc',
        steps=50,
        log_interval=1,
        method=method
    )
```

Notes

- Details about the Hotbit parameters can be found at the website of the Hotbit consortium.
- To perform a spin-polarized calculation with the DFTB parameters you must use the atomic spin-polarization splitting W from the built-in database (see *Spin polarization*).

HuckelBasisParameters

class HuckelBasisParameters(*element*, *orbitals*, *occupations=None*, *filling_method=None*, *ionization_potential=None*, *onsite_hartree_shift=None*, *onsite_spin_split=None*, *onsite_spin_orbit_split=None*, *number_of_valence_electrons=None*, *wolfsberg_helmholtz_constant=None*, *vacuum_level=None*)

Class for representing the parameters that determine the properties of a Huckel basis.

Parameters

- **element** (*PeriodicTableElement*) – The element associated with the basis set.
- **orbitals** (*SlaterOrbital* | list of *SlaterOrbital*) – The basis orbitals. One should be given for each subshell.
- **occupations** (*list of float*) – The initial occupation of each orbital (subshell) as a list of positive floats with one entry for each subshell. Either this, or `number_of_valence_electrons` must be given.
- **filling_method** (`SphericalSymmetric` | `Anisotropic`) – The method used for setting up the initial occupation. **Default:** `SphericalSymmetric`
- **ionization_potential** (*PhysicalQuantity* of type energy) – The ionization potential with one entry for each (subshell). **Default:** Must be given.
- **onsite_hartree_shift** (*PhysicalQuantity* of type energy) – The on-site Hartree shift with one entry for each (subshell). **Default:** Must be given.
- **onsite_spin_split** (*PhysicalQuantity* of type energy) – The on-site spin-split values of each orbital (subshell) pair. This should be a square matrix of size n where n is the number of orbitals (subshells). **Default:** No splitting

- **onsite_spin_orbit_split** (*PhysicalQuantity* of type energy) – The on-site spin-orbit with one entry for each orbital (subshell). **Default:** No splitting
- **number_of_valence_electrons** (*int*) – The number of valence electrons used to determine the neutral state. Either this, or `occupations` must be given.
- **wolfsberg_helmholtz_constant** (*float*) – The Wolfsberg-Helmholtz constant. Must be positive. **Default:** `1.75`
- **vacuum_level** (*PhysicalQuantity* of type energy) – The energy shift of the vacuum level. **Default:** `0.0 * Hartree`

angularMomenta()

Returns
The angular momentum of each orbital (subshell) in the basis set.

Return type
list of int

element()

Returns
The element of the basis set.

Return type
PeriodicTableElement

fillingMethod()

Returns
The method used for setting up the initial occupation.

Return type
`SphericalSymmetric | Anisotropic`

ionizationPotential()

Returns
The ionization potential of each orbital (subshell) in the basis set.

Return type
PhysicalQuantity of type energy

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfValenceElectrons()

Returns
The sum of the occupations.

Return type
float

occupations()

Returns
The occupations associated with the orbitals.

Return type
list of float

`onsiteHartreeShift()`

Returns
The on-site Hartree shift of each orbital (subshell) in the basis set.

Return type
PhysicalQuantity of type energy

`onsiteSpinOrbitSplit()`

Returns
The on-site spin-orbit split of each orbital (subshell) in the basis set.

Return type
PhysicalQuantity of type energy

`onsiteSpinSplit()`

Returns
The on-site spin-split values of each orbital (subshell) pair. This should be a square matrix of size n where n is the number of orbitals (subshells).

Return type
PhysicalQuantity of type energy

`orbitals()`

Returns
The orbitals of the basis set.

Return type
list of *SlaterOrbital*

`uniqueString()`

Return a unique string representing the state of the object.

`vacuumLevel()`

Returns
The energy shift of the vacuum level.

Return type
PhysicalQuantity of type energy

`wolfsbergHelmholtzConstant()`

Returns
The Wolfsberg-Helmholtz constant.

Return type
float

Usage Examples

Define an Extended Huckel Basis for Carbon

```
carbon_2s = SlaterOrbital(
   principal_quantum_number=2,
   angular_momentum=0,
   slater_coefficients=[ 2.0249*1/Bohr ],
   weights=[ 0.76422]
   )

carbon_2p = SlaterOrbital(
   principal_quantum_number=2,
   angular_momentum=1,
   slater_coefficients=[ 1.62412*1/Bohr , 2.17687*1/Bohr ],
   weights=[ 0.27152 , 0.73886 ]
   )

carbon_3d = SlaterOrbital(
   principal_quantum_number=3,
   angular_momentum=2,
   slater_coefficients=[ 1.1944*1/Bohr ],
   weights=[ 0.49066]
   )

CarbonBasis = HuckelBasisParameters(
   element=Carbon,
   orbitals=[ carbon_2s , carbon_2p , carbon_3d ],
   ionization_potential=[ -19.88924*eV , -13.08001*eV , -2.04759*eV ],
   onsite_hartree_shift=ATK_U(Carbon, ['2s','2p','3d']),
   onsite_spin_split=ATK_W(Carbon, ['2s','2p','3d']),
   number_of_valence_electrons=4,
   wolfsberg_helmholtz_constant=2.8,
   vacuum_level=0.0*eV,
   )
```

Notes

Tables with the available parameter sets can be found in *Extended Hückel basis sets*.

The `onsite_hartree_shift` can be obtained from the *ATK_U* database.

The `onsite_spin_split` can be obtained from the *ATK_W* database.

The overlap matrix,

$$
\begin{aligned}
S_{ij} &= \delta_{ij}, \\
&\text{if } \mathbf{R}_i = \mathbf{R}_j \\
&= \int_V \phi_i(\mathbf{r} - \mathbf{R}_i)\phi_j(\mathbf{r} - \mathbf{R}_j)\mathrm{d}\mathbf{r} \\
&\text{if } \mathbf{R}_i \neq \mathbf{R}_j
\end{aligned}
$$

is the central object in the extended Huckel model, where ϕ is of the type *SlaterOrbital*. The one-electron Hamiltonian

is defined by

$$
\begin{aligned}
H_{ij} &= E_i + V_H(\mathbf{R}_i) - E^{\mathrm{VAC}}, \\
&\text{if } i = j \\
&\quad = \frac{1}{4}(\beta_i + \beta_j)(E_i + E_j)S_{ij} + \frac{1}{2}(V_H(\mathbf{R}_i) + V_H(\mathbf{R}_j) - 2E^{\mathrm{VAC}})S_{ij} \\
&\text{if } i \neq j
\end{aligned}
$$

where the parameters E_i is the `ionization_potential`, β the `wolfsberg_helmholtz_constant`, and the `vacuum_level`, E^{VAC}, is used to shift the position of the vacuum level.

V_H is the Hartree potential, which is obtained by solving Poissons equation for the charge density

$$\delta n(\mathbf{r}) = \sum_{\mu} \delta m_{\mu} (\frac{\alpha_{\mu}}{\pi})^{\frac{3}{2}} e^{-\alpha_{\mu}|\mathbf{r}-\mathbf{R}_{\mu}|^2},$$

where $\delta m_{\mu} = m_{\mu} - Z_{\mu}$ is the total charge of atom μ, i.e. the sum of the Mulliken population m_{μ} and the ionic charge $-Z_{\mu}$.

The width of the Gaussian, $\alpha = \pi U^2/4e^4$, is specified through the `onsite_hartree_shift`, U.

Table 4.19: HuckelBasisParameters

Symbol	*HuckelBasisParameters*
E_i	`ionization_potential`
β	`wolfsberg_helmholtz_constant`
U	`onsite_hartree_shift`
E^{VAC}	`vacuum_level`

HuckelCalculator

class HuckelCalculator(*basis_set=None*, *pair_potentials=None*, *charge=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *poisson_solver=None*, *weighting_scheme=None*, *checkpoint_handler=None*, *spin_polarization=None*, *fixed_spin_moment=None*, *dynamical_matrix_parameters=None*, *algorithm_parameters=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*)

Class for representing calculations using the extended Huckel model for configurations of the type *MoleculeConfiguration* and *BulkConfiguration*.

Parameters

- **basis_set** (list of *HuckelBasisParameters*) – An object describing the basis set used for the Extended-Huckel calculation. **Default:** HoffmannHuckelParameters.All
- **pair_potentials** (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations. **Default:** No pair potential
- **charge** (*float*) – The charge of the system; a charge of -1 corresponds to one extra electron. **Default:** 0.0
- **numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10.0*Hartree,
```

(continues on next page)

(continued from previous page)

```
    k_point_sampling=MonkhorstPackGrid(1, 1, 1),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1.0e-6,
    interaction_max_range=10.0*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(300.0*Kelvin))
```

- **iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self- consistent Huckel calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`.
- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourierSolver* for a *BulkConfiguration* without any metallic or dielectric `SpatialRegion`.
- **weighting_scheme** (*HoffmannWeighting* | *WolfsbergWeighting*) – The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian. **Default:** *WolfsbergWeighting*
- **checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.
- **spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`
- **fixed_spin_moment** (float | `False`) – Total spin moment (per unit cell) to use, defined as $\Delta N = N\_N$, where N_up and N_down are the number of electrons in the Up and Down spin channels, respectively. When specified the spin moment will be fixed at the given value by introducing separate Fermi levels for the Up and Down spin channels. Can only be specified when doing a calculation with polarized spin. If set to `False` the spin moment will not be fixed - a single Fermi level is used. **Default:** `False`
- **dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.
- **algorithm_parameters** (*AlgorithmParameters*) – The *AlgorithmParameters* used for calculating the density matrix. **Default:**

```
AlgorithmParameters(
    density_matrix_method=DiagonalizationSolver(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    store_energy_density_matrix=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    use_symmetries=False)
```

- **hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.
- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:**

```
ParallelParameters(
    processes_per_neb_image=None,
    processes_per_individual=None,
    processes_per_bias_point=None,
    processes_per_saddle_search=1)
```

algorithmParameters()

Returns
The algorithm parameters.

Return type
AlgorithmParameters

basisSet()

Returns
The basis set associated with the Hamiltonian parametrization.

Return type
BasisSet

charge()

Returns
The charge of the system.

Return type
float

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

dynamicalMatrixParameters()
This method is deprecated.

fixedSpinMoment()
Get the fixed spin moment.

Returns
The fixed spin moment or `False` if the spin moment is not held fixed.

Return type
float | `False`

hamiltonianDerivativesParameters()
This method is deprecated.

hamiltonianParametrization()

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
`SemiEmpiricalHamiltonianParametrization`

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfSpins()

Returns
The number of spins.

Return type
int

numericalAccuracyParameters()

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setCheckpointHandler(*checkpoint_handler*)

Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

setWeightingScheme(*weighting_scheme*)

Set the weighting scheme.

Parameters
weighting_scheme (*HoffmannWeighting* | *WolfsbergWeighting*) – The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible. @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

weightingScheme()

Returns
The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian.

Return type
HoffmannWeighting | *WolfsbergWeighting*

Attention: The *HuckelCalculator* is being deprecated. Use the *SemiEmpiricalCalculator* with the *HuckelHamiltonianParametrization* instead.

Usage Examples

Perform a self-consistent Hückel calculation by setting the *IterationControlParameters*

```
iteration_control_parameters = IterationControlParameters(
    damping_factor=0.05,
    number_of_history_steps=12,
    )

calculator = HuckelCalculator(
    iteration_control_parameters=iteration_control_parameters,
    )
```

Restart a Hückel calculation using the self-consistent state from a previous calculation

```
# Read in the BulkConfiguration with the old SCF state
old_calculation=nlread("filename.nc",BulkConfiguration)[0]

# Define the BulkConfiguration with the same number of atoms and the same elements
new_calculation=BulkConfiguration(...)

# Define the calculator
old_calculator = old_calculation.calculator()

# make a clone of the old calculator
new_calculator = old_calculator()

# Attach the calculator and use the old initial state
new_calculation.setCalculator(new_calculator,initial_state=old_calculation)
```

Calculate the spin-polarized band structure of iron. Note that spin-polarized calculations must be self-consistent.

```
# Set up iron in the BCC configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=BodyCenteredCubic(2.8665*Angstrom),
    elements=[Iron],
    cartesian_coordinates=[[ 0.,  0.,  0.]]*Angstrom
    )

# Setup spin-polarized calculation
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(4, 4, 4) )

calculator = HuckelCalculator(
    basis_set=CerdaHuckelParameters.Iron_bcc_Basis,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=IterationControlParameters(),
    spin_polarization=True,
    )
bulk_configuration.setCalculator(calculator)

# Calculate the bandstructure
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'H', 'P', 'G', 'N', 'P', 'N', 'H'],
    )
nlsave('fe_huckel.nc', bandstructure)
```

Notes

For the details of the extended-Hückel model, see the chapter on *Semi Empirical*.

Default is that the Hückel calculation is non-self-consistent. To make it self-consistent you must set the *IterationControlParameters*.

Note that most parameters are fitted for non-self-consistent situations, and to apply the parameters for self-consistent situations the `vacuum_level` of each element must be shifted to cancel the additional onsite term in the reference structure where the parameters where fitted.

HuckelHamiltonianParametrization

class HuckelHamiltonianParametrization(*basis_set=None*, *weighting_scheme=None*)

A class representing the hamiltonian parametrization for a Huckel type model

Parameters

- **basis_set** (list of *HuckelBasisParameters*) – An object describing the basis set used for the Extended-Huckel calculation. **Default:** `HoffmannHuckelParameters.All`
- **weighting_scheme** (*HoffmannWeighting* | *WolfsbergWeighting*) – The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian. **Default:** *WolfsbergWeighting*

basisSet()

Returns
The basis set.

Return type
list of *HuckelBasisParameters*

pairPotentials()

Return the pair potentials. If no pair potentials are available, it returns `None`.

Returns
The pair potentials.

Return type
PairPotential

usingOrthogonalBasis()

Routine for determining if the Hamiltonian parametrization is using an orthogonal basis set. Always False for Huckel.

weightingScheme()

Returns
The weighting scheme used for calculating off-site matrix elements of the Huckel Hamiltonian.

Return type
HoffmannWeighting | *WolfsbergWeighting*

Usage Examples

The following example demonstrates some of the different basis sets and weighting schemes available for a Huckel-type Hamiltonian parametrization.

```
# Create a Huckel Hamiltonian parametrization with the default
# basis set and the Hoffmann weighting scheme.
hamiltonian_parametrization = HuckelHamiltonianParametrization(
    weighting_scheme=HoffmannWeighting)

# Default weighting and a custom basis set: in this case
# the Hoffmann Carbon basis and Cerda Carbon graphite basis.
basis_set = [
    HoffmannHuckelParameters.Carbon_Basis,
    CerdaHuckelParameters.Carbon_graphite_Basis
]
hamiltonian_parametrization = HuckelHamiltonianParametrization(
    basis_set=basis_set)

# Custom basis set: Hoffmann Hydrogen with a custom vacuum level
# of 1.0 eV and Muller Silicon with a custom vacuum level of 0.0 eV.
basis_set = [
    HoffmannHuckelParameters.Hydrogen_Basis(vacuum_level=1.0 * eV),
    MullerHuckelParameters.Silicon_Basis(vacuum_level=0.0 * eV)
]
hamiltonian_parametrization = HuckelHamiltonianParametrization(
    basis_set=basis_set)
```

Notes

For the details of the weighting schemes used for the extended-Hückel model, see the chapter on *Weighting schemes*. Tables with the available basis sets can be found here: *Extended Hückel basis sets*.

See also *Parameters for the extended Hückel method* for a more general description of the parameters in the extended-Hückel model.

HydrogenOrbital

class HydrogenOrbital(*principal_quantum_number*, *angular_momentum*, *radial_cutoff_radius*, *confinement_start_radius*, *charge=None*, *confinement_strength=None*, *confinement_power=None*, *radial_step_size=None*)

Class for representing an atomic orbital of a Hydrogen atom.

Parameters

- **principal_quantum_number** (*positive int*) – The all-electron principal quantum number (n) of the orbital.
- **angular_momentum** (*non-negative int*) – The angular momentum quantum number (l) of the orbital.
- **radial_cutoff_radius** (*PhysicalQuantity* of type length) – The distance from the core where the basis orbital is zero (compact support radius).
- **confinement_start_radius** (*PhysicalQuantity* of type length) – Radial distance to where the confinement potential starts. The `confinement_start_radius` must be less than or equal to the `radial_cutoff_radius`.

- **charge** (*float*) – Charge of the Hydrogen atom when generating the basis function. **Default:** `1.0`
- **confinement_strength** (*PhysicalQuantity* of type energy) – The confinement strength of the potential confining the Hydrogen orbital. **Default:** `20 * Hartree`
- **confinement_power** (*int*) – The confinement power for the potential. **Default:** `1`
- **radial_step_size** (*PhysicalQuantity* of type length) – The radial step size determining the distance between grid points on the linear radial grid. **Default:** `0.01 * Bohr`

angularMomentum()

Returns
The angular momentum.

Return type
int

charge()

Returns
The charge used for generating this *HydrogenOrbital* instance.

Return type
float

confinementPower()

Returns
The power of the confinement potential.

Return type
int

confinementStartRadius()

Returns
The radius where the confinement potential starts.

Return type
PhysicalQuantity of type length

confinementStrength()

Returns
The strength of the confinement potential.

Return type
PhysicalQuantity of type energy

nlinfo()

Returns
The nlinfo.

Return type
dict

principalQuantumNumber()

Returns
The principal quantum number.

Return type
int

`radialCutoffRadius()`

Returns
The radial cutoff radius.

Return type
PhysicalQuantity of type length

`radialStepSize()`

Returns
The radial grid spacing.

Return type
PhysicalQuantity of type length

`uniqueString()`

Return a unique string representing the state of the object.

HyperdynamicsBondDistortion

`class HyperdynamicsBondDistortion(`*first_atom_tag*, *second_atom_tag*, *bond_minimum=None*, *bond_maximum=None*, *fuzz_factor=None*`)`

Class for calculating bond distortions used in a hyperdynamics simulation.

Parameters

- **first_atom_tag** (*str*) – Tag giving the first set of atoms that can possibly form a bond.
- **second_atom_tag** (*str*) – Tag giving the second set of atoms that can possibly form a bond.
- **bond_minimum** (*PhysicalQuantity* of type length) – The minimum length of the bond.
- **bond_maximum** (*PhysicalQuantity* of type length) – The maximum length of the bond.
- **fuzz_factor** (*float*) – The fuzz factor used in detecting bonds in the configuration. **Default:** 1.1

`bondMaximum()`

Returns
The maximum bond length.

Return type
PhysicalQuantity of type length

`bondMinimum()`

Returns
The minimum bond length.

Return type
PhysicalQuantity of type length

`firstAtomTag()`

Returns
The first atom tag.

Return type
str

`fuzzFactor()`

Returns
The fuzz factor.

Return type
float

`numberOfVariables()`

Returns
The number of indexed geometric coordinates being accelerated

Return type
int

`secondAtomTag()`

Returns
The second atom tag.

Return type
str

`uniqueString()`
Return a unique string representing the state of the object.

Usage Examples

General

Notes

The *HyperdynamicsBondDistortion* implements a distortion based on bonds between atoms. This distortion can be used in either a *StaticHyperdynamics* or *CollectiveVariableHyperdynamics* simulation.

The atoms whose bonds make up the distortion is given using the `first_atom_tag` and `second_atom_tag` arguments. These take the tags of two sets of atoms between which the relevant bonds are formed. These can be the same set of atoms or different sets. These sets should include all atoms of each type that can possibly have the types of bonds to be boosted. Whether or not a bias potential is applied between two atoms depends on whether or not a bond exists in the reference configuration. If the atoms are far enough apart that no bond exists between them, then they will not be included in the distortion bonds. By giving sets of atoms where bonds can possibly be formed it allows the bias to be reset after each reaction and for newly formed bonds to be included in the distortion.

The detection of the bonds is controlled with the `fuzz_factor` keyword argument. Atoms must be closer than the sum of the covalent radii times the fuzz factor to be considered bonded. The default value is 1.1, which is the general bond finding default.

The bond length at the equilibrium structure and transition state are also given with the `bond_minimum` and `bond_maximum` keyword arguments respectively. A bond minimum is always required. The bond maximum is optional, and can be given as `None` in cases where the reaction does not have a well defined transition state.

In the case where a transition state bond length is given, the distortion χ can be given as:

$$\chi = \begin{cases} 0 & r \leq r_{min} \\ \frac{r - r_{min}}{r_{max} - r_{min}} & r_{min} < r < r_{max} \\ 1 & r \geq r_{max} \end{cases}$$

In cases where no maximum is given the distortion is given as:

$$\chi = \begin{cases} 0 & r \leq r_{min} \\ \frac{r - r_{min}}{r_{min}} & r > r_{min} \end{cases}$$

In the first case the distortion is limited to between 0 and 1. In the second case the distortion has no upper limit. In this case a distortion cutoff should be used in the hyperdynamics simulation to ensure that the resulting global distortion is between 0 and 1.

HyperdynamicsPositionDistortion

class HyperdynamicsPositionDistortion(*atom_tag*, *maximum_radius=None*)

Class for calculating position distortions used in a hyperdynamics simulation.

Parameters

- **atom_tag** (*str*) – The tag of the atoms to distort from their positions.
- **maximum_radius** (*PhysicalQuantity* of type length) – The distance from the equilibrium position to the transition state.

atomTag()

Returns
The tag of the atoms that have a positional acceleration.

Return type
str

maximumRadius()

Returns
The maximum radius of the position distortion.

Return type
PhysicalQuantity of type length

numberOfVariables()

Returns
The number of variables that are being accelerated.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

General

Notes

The *HyperdynamicsPositionDistortion* implements a distortion based on atomic position. This distortion can be used in either a *StaticHyperdynamics* or *CollectiveVariableHyperdynamics* simulation.

The atoms whose positions are used to define the distortions are given with a configuration tag using the `atom_tag` argument. A maximum radius of the distance between the atomic position and the transition state is also given with the `max_radius` argument. When the distortion is reset the current atomic position is recorded. The distortion χ is then given as:

$$\chi = \begin{cases} \frac{|\mathbf{r}-\mathbf{r}_0|}{r_{max}} & |\mathbf{r}-\mathbf{r}_0| \leq r_{max} \\ 1 & |\mathbf{r}-\mathbf{r}_0| > r_{max} \end{cases}$$

Here $\mathbf{r}$ is the current atomic position, $\mathbf{r}_0$ is the reference atomic position and r_{max} is the maximum distance between the atomic position and the transition state. The distortion ranges from 0 to 1 and is limited to 1 once the atom moves outside the maximum radius.

This type of distortion is most useful for simulations in crystalline materials. Here the crystal lattice gives the atomic position a well defined reference point. In amorphous materials it is likely more efficient to use distortions which do not rely on a well defined reference point, such as the *HyperdynamicsBondDistortion*.

HypersphereDirection

class HypersphereDirection(*num_directions=None*, *order=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Random'>*, *magnitude=None*, *optimization_iterations=None*)

Class used to generate random directions as points on the surface of a hypersphere. The points are then optimized to be as far away from each other as possible.

Parameters

- **num_directions** (*int*) – The number of directions to generate. **Default:** 500
- **order** (*NLFlag*) – The order in which the directions are fetched. The parameter can be one of Random, RoundRobin or Furthest. Random will return the vectors in a random order. When using Furthest, each returned vector is chosen such that it has the largest sum of distances to the already returned vectors. RoundRobin will return the vectors in a round-robin fashion.
- **magnitude** (*PhysicalQuantity of type length or tuple.*) – The magnitude of the returned directions. If this is a scalar, all the directions will have the same magnitude. If it is a tuple, the magnitude will be randomly chosen between the two values. **Default:** 1 * Angstrom
- **optimization_iterations** (*int*) – The number of iterations to optimize the directions. **Default:** 10

getMagnitude()

Get the magnitude of the directions.

Returns
The magnitude of the directions.

Return type
PhysicalQuantity of type length

getNextDirection()

Get the next direction.

Returns
The next random direction.

Return type
numpy.ndarray

initialize(*num_dimensions*)

Initialize the generator with the number of dimensions.

Parameters
num_dimensions (*int*) – The number of dimensions of the generated vectors.

isInitialized()

Check if the generator is initialized.

Returns
True if the generator is initialized.

Return type
bool

reset()

Reset the direction generator. This means cleaning the list of used vectors and resetting the current direction.

Notes

HypersphereDirection is used to generate initial directions for the different saddle search methods used in *AdaptiveKineticMonteCarlo*. The *HypersphereDirection* method first generates a set of random points on a hypersphere. The distances between the points are then maximized using a simple gradient descent algorithm[1].

After instantiating a new *HypersphereDirection* object, the *initialize()* method should be called with the desired number of vector components an argument. This method will generate the random points on the hypersphere and store them on the object. The order in which the optimized vectors are returned can be controlled by the `order` argument.

HysteresisLoop

class HysteresisLoop(*temperature*, *magnetic_field_strengths*, *magnetic_field_direction*, *total_magnetic_moments*, *max_total_magnetic_moment*)

An object containing the data from a spin dynamics simulation of a hysteresis loop.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the simulation was run.
- **magnetic_field_strengths** (*PhysicalQuantity* of type magnetic field) – The applied magnetic field at each step of the simulation.
- **magnetic_field_direction** (*numpy.ndarray*) – The direction of the applied magnetic field.
- **total_magnetic_moments** (*PhysicalQuantity* of type energy / magnetic field) – The total magnetic moment at each step of the simulation.

[1] ManuelăPlasencia Gutiérrez, Carlos Argáez, and Hannes Jónsson. Improved minimum mode following method for finding first order saddle points. *Journal of Chemical Theory and Computation*, 13(1):125–134, 2017. URL: https://pubs.acs.org/doi/abs/10.1021/acs.jctc.5b01216, doi:10.1021/acs.jctc.5b01216.

- **max_total_magnetic_moment** (*PhysicalQuantity* of type energy / magnetic field) – The maximum total magnetic moment.

coercivity()

Get the coercivity, which is the field one has to apply in the opposite direction of the initial magnetization it takes to demagnetize the system.

Returns
The coerciticity

Return type
PhysicalQuantity of type magnetic field

magneticFieldDirection()

Get the direction of the applied magnetic field. :return: The direction of the applied magnetic field. :rtype: numpy.ndarray

magneticFieldStrengths()

Get the applied magnetic field strengths. :return: The applied magnetic fields. :rtype: *PhysicalQuantity* of type magnetic field.

maxTotalMagneticMoment()

Get the maximum total magnetic moments. :return: The maximum total magnetic moments. :rtype: *PhysicalQuantity* of type energy / magnetic field

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the data.

Return type
dict

nlprint(*stream*)

Print report with the data and results to a given stream.

Parameters
stream (*stream*) – The stream that the report should be written to.

projectedMagneticMoments()

Get the total magnetic moments projected on the direction of the applied magnetic field. :return: The projected total magnetic moments. :rtype: *PhysicalQuantity* of type energy / magnetic field

relativeMagnetizations()

Calculate the relative magnetizations. :return: The relative magnetizations. :retype: ndarray

remanence()

Get the remanence, i.e. the magnetic moment when there is no applied magnetic field.

Returns
The remanence.

Return type
PhysicalQuantity of type energy / magnetic field

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`temperature`()

Get the temperature at which the simulation was run. :return: The temperature. :rttype: *PhysicalQuantity* of type temperature.

`totalMagneticMoments`()

Get the total magnetic moments. :return: The total magnetic moments. :rtype: *PhysicalQuantity* of type energy / magnetic field

`uniqueString`()

Return a unique string representing the state of the object.

IVCharacteristics

`class IVCharacteristics`(*configuration*, *filename*, *object_id*, *gate_regions=None*, *gate_source_voltages=None*, *drain_source_voltages=None*, *gate_potential_alignment=None*, *energies=None*, *kpoints=None*, *kpoints_weights=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*, *device_transmission_method=None*, *enforce_zero_transmission_in_band_gap=None*, *source_electrode=None*, *max_initial_state_configurations=None*)

Constructor for the IVCharacteristics object.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which to perform the study. The configuration must include at least one metallic region acting as the gate.
- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **gate_regions** (*int | list of int*) – An index or list of indices of the metallic regions present in the configuration for which to apply the gate voltage. **Default:** All metallic regions present in the configuration.
- **gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of the gate-source voltages for which the current will be calculated the next time the object is updated. It is assumed that the source electrode is grounded, and that the gate-source voltage is defined as `V_gs = V_g - V_s`. Therefore, in each transmission calculation, the source electrode voltage will be set to zero and the gate regions voltage to `V_gs`. **Default:** `[0.0] * Volt`
- **drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of the drain-source voltages for which the current will be calculated the next time the object is updated. It is assumed that the source electrode is grounded, and that the drain-source voltage is defined as `V_ds = V_d - V_s`. Therefore, in each transmission calculation, the source electrode voltage will be set to zero and the left electrode voltage to `V_ds`. **Default:** `[0.5] * Volt`

- **gate_potential_alignment** (*GatePotentialAlignment*) – A parameter indicating the difference between the work function of the gate electrodes and the work function of a reference electrode, plus an optional energy shift. Such difference is used to determine the reference potential for `V_gs=0`, effectively shifting an `I-Vgs` characteristics. **Default:** `GatePotentialAlignment(reference_electrode=`Left`, shift=0.0*eV, use_intrinsic_chemical_potential=False)`

- **energies** (list of *PhysicalQuantity* of type energy) – A list of the energies for which the *TransmissionSpectrum* at each voltage should be calculated. **Default:** Energy range that covers the bias window, plus $30k_BT$.

- **kpoints** (*MonkhorstPackGrid* | *RegularKpointGrid* | *AdaptiveGrid* | list (size n_kpoints) of list (size 3) of float) – The k-points for which the *TransmissionSpectrum* at each voltage should be calculated. Note that the k-points must be in the same xy-plane. Set `nc=1` for MonkhorstPackGrid and RegularKpointGrid. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

- **kpoints_weights** (*list (size n_kpoints) of float*) – The weight of each k-point for which the *TransmissionSpectrum* at each voltage should be calculated. **Default:** The weights corresponding to the MonkhorstPackGrid/RegularKpointGrid, or equal weights if the k-points are specified as a list.

- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The SelfEnergyCalculator to be used for the *TransmissionSpectrum* calculation at each voltage. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`

- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`) – The choice for the zero of energy to be used for the *TransmissionSpectrum* calculation at each voltage. **Default:** `AverageFermiLevel`

- **infinitesimal** (*PhysicalQuantity* of type energy) – Small positive energy, used to move the *TransmissionSpectrum* calculation at each voltage away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6 * eV`

- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of the calculations, each to be stored in a separate file. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'iv_characteristics_'`

- **number_of_processes_per_task** (*int*) – The number of processes that will be used to execute each task. If this value is greater than or equal to the total number of available processes, each single task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. **Default:** All available processes execute each task collaboratively.

- **device_transmission_method** (*GreensFunction* | *SparseGreensFunction*.) – The method employed in the *TransmissionSpectrum* calculation at each voltage. Note that *GreensFunction* is only supported for a maximum of 2 (1) `processes_per_contour_point` for Unpolarized, Polarized (Noncollinear, SpinOrbit) calculations. **Default:** `GreensFunction(processes_per_contour_point=1)`.

- **enforce_zero_transmission_in_band_gap** (*bool*) – Flag which specifies whether the transmission values are enforced to zero inside the band gap. **Default:** True

- **source_electrode** (`Left` | `Right`) – Specify which electrode should be set as Source. **Default:** `Left`

- **max_initial_state_configurations** – Maximum number of previous converged configurations used to determine the initial state for a new bias point. If `1` is specified, the configuration closest in bias to the new bias point is used as initial state. If an integer `n > 1` is specified, then the `n` configurations closest in bias are used to extrapolate the initial state for the new bias point. It must be a positive integer. **Default:** `3`

- **max_initial_state_configurations** – int

addAnalysis(*gate_source_voltages*, *drain_source_voltages*, *analysis_type*, *analysis_arguments=None*)

Add an analysis object to the IVCharacteristics study for given gate-source and drain-source voltage pair.

This method can be invoked several times; in this case, multiple analysis objects will be added to the same voltage pairs.

Parameters

- **gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of gate-source voltages for which the analysis should be appended. Each voltage in the list needs to be already present.
- **drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of drain-source voltages for which the analysis should be appended. Each voltage in the list needs to be already present.
- **analysis_type** (An `Analysis` type supported by IVCharacteristics.) – The type of the analysis object to be calculated.
- **analysis_arguments** (A dictionary of valid arguments for an analysis object of the type specified in `analysis_type`.) – Dictionary of optional arguments to be passed to the analysis object. The configuration arguments must not be present. **Default:** An empty dictionary, except for ProjectedLocalDensityOfStates or DeviceDensityOfStates analysis, where the default `energies`, `kpoints`, `self_energy_calculator`, `energy_zero_parameter`, and `infinitesimal` are take from the study object itself.

addProjectedLocalDensityOfStates(*gate_source_voltages*, *drain_source_voltages*, *energies=None*)

Add a ProjectedLocalDensityOfStates analysis object to the IVCharacteristics study for a given gate-source and drain-source voltage pair. The input paramenters are taken from the IVCharacteristics study.

Parameters

- **gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of gate-source voltages for which the analysis should be appended. Each voltage in the list needs to be already present.
- **drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of drain-source voltages for which the analysis should be appended. Each voltage in the list needs to be already present.
- **energies** (list of *PhysicalQuantity* of type energy) – The energies for which the density of states should be calculated. **Default:** The value given in the constructor of the study object.

addVoltages(*gate_source_voltages=None*, *drain_source_voltages=None*)

Add a list of gate-source and/or drain-source voltage points to the IVCharacteristics study. These will be calculated the next time the object is updated.

Parameters

- **gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – The list of gate-source voltages to add.
- **drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – The list of drain-source voltages to add.

calculatedVoltages()

Returns
The list of gate-source and drain-source voltage pairs for which a transmission spectrum has been calculated.

Return type
list of *PhysicalQuantity* of type voltage

configuration(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the updated configuration for the given gate-source and drain-source voltages.

This result will only be available after the calculation ran successfully.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The configuration associated with this voltage pair. If not available, returns None.

Return type
DeviceConfiguration | None

configurationCalculator(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the updated configurations calculator for the given gate-source and drain-source voltages.

This result will only be available after the calculation ran successfully.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The calculator associated with this voltage pair. If not available, returns None.

Return type
DeviceLCAOCalculator | *DeviceSemiEmpiricalCalculator* None

current(*gate_source_voltage*, *drain_source_voltage*, *electrode_temperatures=None*)

The current calculated from the *TransmissionSpectrum* analysis. The sign of the current is adjusted to be consistent with the choice of source electrode.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.
- **electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated current for the required gate-source and drain-source voltages. If not available, returns None.

Return type
PhysicalQuantity of type current | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

deviceTransmissionMethod()

Returns
The method used for the *TransmissionSpectrum* calculation at each voltage.

Return type
GreensFunction | *SparseGreensFunction*

drainInducedBarrierLowering(*threshold_current*, *high_drain_source_voltage*, *low_drain_source_voltage*, *min_subthreshold_current=None*, *max_subthreshold_current=None*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*)

Calculate drain induced barrier lowering (DIBL). The threshold voltage is defined as the gate-source voltage for which the current has a user-defined value, evaluated at a high and a low source-drain voltage value.

The DIBL is calculated as the unitless ratio

$DIBL = \frac{V_{TH}^{low} - V_{TH}^{high}}{V_{DS}^{high} - V_{DS}^{low}}$

In order to obtain meaningful results, a current in the subthreshold regime must be specified.

Parameters

•**threshold_current** (*PhysicalQuantity* of type current) – The current for which the device is considered to be at the gate-source threshold.

•**high_drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage at the working point (i.e., the supply voltage).

•**low_drain_source_voltage** (*PhysicalQuantity* of type voltage) – The low drain voltage used as reference. It needs to be a finite value close to 0.

•**min_subthreshold_current** (*PhysicalQuantity* of type current) – The minimum value of current used to extract the subthreshold regime model. **Default:** threshold_current / 10

•**max_subthreshold_current** (*PhysicalQuantity* of type current) – The minimum value of current used to extract the subthreshold regime model. **Default:** threshold_current * 10

•**range_min** (*PhysicalQuantity* of type voltage) – The minimum gate-source voltage used for the processing. This parameter can be used to select the nmos-like or pmos-like section of the IV curve in bipolar devices. **Default:** No restriction.

•**range_max** (*bool*) – The maximum gate-source voltage in the required range. This parameter can be used to select the nmos-like or pmos-like section of the IV curve in bipolar devices. **Default:** No restriction.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated drain induced barrier lowering.

Return type
float

drainSourceSaturationVoltage(*gate_source_voltage*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the drain-source saturation voltage for a given gate-source voltage. It is assumed that a transition to saturation regime occurs in the interval specified by range_min and range_max. The value is estimated by fitting the data to a Schichman-Hodges model (SPICE Mosfet model 1).

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage, for which the drain-source saturation voltage will be calculated.

•**range_min** (*PhysicalQuantity* of type voltage) – The minimum drain-source voltage in the required range. **Default:** No restriction.

•**range_max** (*bool*) – The maximum drain-source voltage in the required range. **Default:** No restriction.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

- **electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated drain-source saturation voltage, fitted in the required range.

Return type
PhysicalQuantity of type voltage

drainSourceVoltages()

Returns
The list of all calculated and not calculated drain-source voltages. Any drain-source voltage which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of *PhysicalQuantity* of type voltage

energies()

Returns
The list of energies used for the *TransmissionSpectrum* calculation at each voltage.

Return type
list of *PhysicalQuantity* of type energy

energyZeroParameter()

Returns
The choice for the zero of energy used for the *TransmissionSpectrum* calculation at each voltage.

Return type
AverageFermiLevel | AbsoluteEnergy

enforceZeroTransmissionInBandGap()

Returns
Whether the transmission values are enforced to zero inside the band gap.

Return type
bool

filename()

Returns
The filename where the study object is stored.

Return type
str

gatePotentialAlignment()

Returns
The object describing the alignment of the gate electrode potential.

Return type
class:~.GatePotentialAlignment:

gateReferencePotential()

Returns
The absolute shift in gate voltages introduced by a gate potential alignment, with respect to the internal default. If the constructor parameter *gate_potential_alignment* is defined with a non-default *reference_electrode* then additional tasks

have to be executed. In that case the absolute shift in gate voltages is returned when the information is available, and *None* is returned otherwise.

Return type
PhysicalQuantity of type voltage | None

gateRegions()

Returns
The list of indices of the metallic regions present in the configuration which the gate voltage is applied to.

Return type
list of int

gateSourceVoltages()

Returns
The list of all calculated and not calculated gate-source voltages. Any gate-source voltage which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of *PhysicalQuantity* of type voltage

infinitesimal()

Returns
The small positive energy used to move the *TransmissionSpectrum* calculation at each voltage away from the real axis for the case of a recursion-style self-energy calculator.

Return type
PhysicalQuantity of type energy

isConverged(*gate_source_voltage*, *drain_source_voltage*)

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
Whether the calculator for the required gate-source and drain-source voltages has been converged. If not available, returns None.

Return type
bool | None

kpoints()

Returns
The k-points used for the *TransmissionSpectrum* calculation at each voltage.

Return type
MonkhorstPackGrid | *RegularKpointGrid* | *AdaptiveGrid* | list (size n_kpoints) of list (size 3) of float

kpointsWeights()

Returns
The weight of each k-point used for the *TransmissionSpectrum* calculation at each voltage.

Return type
list (size n_kpoints) of float

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

maxInitialStateConfigurations()

Returns
The maximum number of converged configurations used to determine the initial state for a new bias point.

Return type
int

nlinfo()

Returns
Structured information about the Study.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

onOffRatio(*drain_source_voltage*, *off_gate_source_voltage*, *on_gate_source_voltage*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the on/off ratio for the available data points at a given drain-source voltage.

Parameters

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage which determines the gate-source voltage/current data set from which the on/off ratio will be calculated. **Default:** The first drain-source voltage in the IVCharacteristics study.

•**off_gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage corresponding to the off state.

•**on_gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage corresponding to the on state.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated on/off ratio in the required range.

Return type
float

removeAnalysis(*gate_source_voltages*, *drain_source_voltages*, *analysis_type*)

Remove an analysis object from the IVCharacteristics study for given gate-source and drain-source voltage pairs.

Note that if multiple analysis objects of the same type exist, they will all be removed.

Parameters

•**gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of gate-source voltages for which the analysis object should be removed.

•**drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of drain-source voltages for which the analysis object should be removed.

•**analysis_type** (An `Analysis` type supported by IVCharacteristics.) – The type of the analysis object to be removed.

removeDrainSourceVoltage(*drain_source_voltage*)

Remove a drain-source voltage from the IVCharacteristics study. All analysis objects corresponding to the given drain-source voltage will be removed.

Parameters
drain_source_voltage (*PhysicalQuantity* of type voltage) – The drain-source voltage which should be removed from the IVCharacteristics study.

removeGateSourceVoltage(*gate_source_voltage*)

Remove a gate-source voltage from the IVCharacteristics study. All analysis objects corresponding to the given gate-source voltage will be removed.

Parameters
gate_source_voltage (*PhysicalQuantity* of type voltage) – The gate-source voltage which should be removed from the IVCharacteristics study.

results(*gate_source_voltage*, *drain_source_voltage*, *result_types=None*)

Retrieve all results of the specified types for the given gate-source and drain-source voltages.

Only the results of calculations that ran successfully are available.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

•**result_types** (*list*) – A list of types which should be included in the returned quantity. For example, result_types=[DeviceConfiguration] will only return instances of DeviceConfiguration. If not specified, all results are returned.

Returns
The available results from calculations associated with this voltage pair. If none available, it returns an empty list.

Return type
list

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

selfEnergyCalculator()

Returns
The SelfEnergyCalculator used for the *TransmissionSpectrum* calculation at each voltage.

Return type
DirectSelfEnergy | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*

setCalculator(*gate_source_voltage*, *drain_source_voltage*, *calculator*, *initial_state=None*, *initial_spin=None*)

Set a new calculator and initial state for an existing gate-source and drain-source voltage pair.

The next call to update() will recalculate the configuration, transmission spectrum and any additional Analysis objects for this gate-source and drain-source voltage pair. If a configuration with an identical calculator had already converged, no calculation will be performed.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

•**calculator** (*DeviceLCAOCalculator* | *DeviceSemiEmpiricalCalculator*) – The new calculator to be set on the configuration.

•**initial_state** (*DeviceConfiguration*) – The initial state to be used for the configuration. **Default:** The initial state is automatically chosen from the already run tasks.

•**initial_spin** (*InitialSpin* | None) – An initial spin to be applied in combination with the initial state. It can only be defined if an initial state is also defined. Alternatively, a global initial spin to be used during the first configuration update can be set directly on the configuration passed to the class constructor.

setTransmissionSpectrumArguments(*gate_source_voltage*, *drain_source_voltage*, *transmission_arguments=None*)

Set new arguments for the TransmissionSpectrum object for an existing gate-source and drain-source voltage pair.

The next call to update() will recalculate the transmission spectrum for this gate-source and drain-source voltage pair.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

•**transmission_arguments** (*A dictionary of valid arguments for a TransmissionSpectrum object.*) – Dictionary of arguments to be passed to the TransmissionSpectrum object. The configuration argument must not be present. **Default:** An empty dictionary.

`sourceElectrode()`

Return the side of the source electrode, specified as `Left` or `Right`. The quantity is used to convert from absolute electrode voltages to drain-source and gate-source voltages.

`subthresholdSlope(`*drain_source_voltage=None*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*`)`

Calculate the subthreshold slope for the available data points at a given drain-source voltage.

Note that by default the entire range is used for the fitting, which might give an unphysical result. It is strongly recommended for the user to define a physical range in which to perform the fitting.

Parameters

- **`drain_source_voltage`** (*PhysicalQuantity* of type voltage) – The drain-source voltage which determines the gate-source voltage/current data set from which the subthreshold slope will be calculated.
- **`range_min`** (*PhysicalQuantity* of type voltage) – The minimum gate-source voltage in the required range. **Default:** No restriction.
- **`range_max`** (*bool*) – The maximum gate-source voltage in the required range. **Default:** No restriction.
- **`converged_only`** – Whether to only include data points which have converged. **Default:** True
- **`electrode_temperatures`** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated subthreshold slope, fitted in the required range. Units are mV/dec.

Return type
float

`transconductance(`*drain_source_voltage*, *gate_source_voltage*, *converged_only=None*, *electrode_temperatures=None*`)`

Calculate the transconductance, defined as:

$g_m = \partial I_D / \partial V_{GS}$

at a given drain-source voltage, where I_D is the drain current and V_{GS} the gate-source voltage. The transconductance is evaluated around the gate-source voltage specified in input. The derivative is calculated by finite difference, using the available currents.

Parameters

- **`drain_source_voltage`** (*PhysicalQuantity* of type voltage) – The drain-source voltage at which the transconductance is calculated,
- **`gate_source_voltage`** (*PhysicalQuantity* of type voltage) – The gate-source voltage at which the transconductance is evaluated.
- **`converged_only`** – Whether to only include data points which have converged. **Default:** True
- **`electrode_temperatures`** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The value of transconductance at the given gate-source and drain-source voltage.

Return type
PhysicalQuantity of type current / voltage.

transmissionSpectrum(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the TransmissionSpectrum for the updated configuration for the given gate-source and drain-source voltages.

This result will only be available after the transmission spectrum calculation ran successfully.

Additional TransmissionSpectrum analysis objects added with addAnalysis will be ignored.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The transmission spectrum associated with this voltage pair. If not available, returns None.

Return type
TransmissionSpectrum | None

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study.

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *IVCharacteristics* object can be used to perform a detailed study on a device. The object is initialized by specifying the *DeviceConfiguration* to study, a list of gate-source and drain-source voltages over which to scan, and the parameters needed to perform a *TransmissionSpectrum* calculation for each voltage.

The *IVCharacteristics* object supports configurations with an arbitrary number of gates, defined as metallic regions in the *DeviceConfiguration*. During the calculation the gate voltage is set equally on the gate regions specified by the `gate_regions` parameter; other metallic regions present in the configuration will not be affected. It is also possible to use *IVCharacteristics* for a *DeviceConfiguration* without any gate. In this case, `gate_regions` and `gate_source_voltages` can be left unspecified. A dummy value `gate_source_voltages=0.0*Volt` is then assumed.

After performing the calculation, the results can be retrieved by calling a number of methods:

- `calculatedVoltages`
- `configuration`
- `configurationCalculator`
- `current`
- `isConverged`
- `results`
- `transmissionSpectrum`

There are also a number of post-processing methods available, which will perform their analysis on the available results:

- `onOffRatio`
- `subthresholdSlope`

•drainSourceSaturationVoltage

•drainInducedBarrierLowering

•transconductance

The user is free to add gate-source and drain-source voltage points to the study by calling addVoltages. It is also possible to specify additional *Analysis* objects to be calculated on the configuration alongside the *TransmissionSpectrum* for each voltage point (or only selected points); this can be done by calling addAnalysis. The results for these additional Analysis calculations can be retrieved by calling the results method.

Finally, existing gate voltage points can be modified by calling a number of methods:

•setCalculator

•setTransmissionSpectrumArguments

Calling update after one of these methods will recalculate any point which is no longer up to date, including any additional *Analysis* objects present.

Tip: See the tutotial *Electrical characteristics of devices using the IVCharacteristics study object* for an example of usage of *IVCharacteristics* study on a realistic device and a demonstration of the graphical user interface.

See also *GatePotentialAlignment*.

Usage Example

This example shows how to perform a simple study of a gated carbon nanotube system and display a summary of results.

First, we define the device system with metallic regions to act as the gate. This is given in the following file (for clarity, we split this from the *IVCharacteristics* study itself):

iv_characteristics_device_setup.py

We can now proceed with the study. Note that this example is only intended to give an overview of the capabilities of *IVCharacteristics*.

```
# Setup the device configuration.
device_configuration = nlread('iv_characteristics_device_setup.py',␣
↪DeviceConfiguration)[0]

# ------------------------------------------------------------
# IVCharacteristics study
# ------------------------------------------------------------
# Initialize the IV characteristics study with 11 gate-source
# and a single drain-source voltage points.
iv_characteristics = IVCharacteristics(
    configuration=device_configuration,
    filename='ivcharacteristics.hdf5',
    object_id='graphene_ribbon',
    gate_regions=[0, 1],
    gate_source_voltages=numpy.linspace(-1, 1, 11)*Volt,
    drain_source_voltages=0.1*Volt,
    log_filename_prefix='ivcharacteristics_',
    )
```

(continues on next page)

(continued from previous page)

```
# Run update to calculate the initial points.
iv_characteristics.update()

# Print the iv_characteristics report.
nlprint(iv_characteristics)

# Add a new gate-source voltage point.
iv_characteristics.addVoltages(gate_source_voltages=1.5 * Volt)

# Call update again to calculate the new point.
iv_characteristics.update()

# Calculate the on/off ratio at Vds=0.1*Volt and print it out.
on_off_ratio = iv_characteristics.onOffRatio(
    0.1 * Volt,
    -1.0 * Volt,
    1.0 * Volt)
print("On/off ratio:", on_off_ratio)

# Add an additional DeviceDensityOfStates calculation to two of the gate voltage
# points. We also specify custom arguments for this object.
analysis_arguments = {'energies': numpy.arange(-1.0, 1.0, 0.01) * eV}
iv_characteristics.addAnalysis(
    [0.2, 0.4] * Volt,
    0.1 * Volt,
    DeviceDensityOfStates,
    analysis_arguments
    )

# Update again.
iv_characteristics.update()

# Get the calculated objects for a particular voltage value.
gate_source_voltage = 0.2 * Volt
drain_source_voltage = 0.1 * Volt
configuration = iv_characteristics.configuration(
    gate_source_voltage,
    drain_source_voltage)
calculator = iv_characteristics.configurationCalculator(
    gate_source_voltage,
    drain_source_voltage)
transmission_spectrum = iv_characteristics.transmissionSpectrum(
    gate_source_voltage,
    drain_source_voltage)
device_dos = iv_characteristics.results(
    gate_source_voltage,
    drain_source_voltage,
    [DeviceDensityOfStates])[0]

# Set a new initial state for this gate voltage value.
restart_gate_voltage = -0.2 * Volt
restart_configuration = iv_characteristics.configuration(
```

(continues on next page)

(continued from previous page)

```
    restart_gate_voltage,
    drain_source_voltage)
iv_characteristics.setCalculator(
    gate_source_voltage,
    drain_source_voltage,
    calculator,
    restart_configuration)

# Also set some new arguments for the transmission spectrum.
new_arguments = {'energies': numpy.linspace(-2, 2, 21)*eV}
iv_characteristics.setTransmissionSpectrumArguments(
    gate_source_voltage,
    drain_source_voltage,
    new_arguments)

# Running update will calculate this gate voltage point again.
iv_characteristics.update()

# Print again the iv_characteristics report.
nlprint(iv_characteristics)
```

`iv_characteristics.py`

Here is the resulting output from the last `nlprint` command:

```
+-------------------------------------------------------------------------------+
| IVCharacteristics Report                                                      |
+-------------------------------------------------------------------------------+
|       Gate-source      Drain-source      Drain-source                         |
|        voltage (V)       voltage (V)       current (A)                       |
+-------------------------------------------------------------------------------+
|          -1.000000          0.100000      1.432517e-13                       |
|          -0.800000          0.100000      7.487856e-13                       |
|          -0.600000          0.100000      7.141447e-12                       |
|          -0.400000          0.100000      1.520051e-10                       |
|          -0.200000          0.100000      7.014625e-09                       |
|           0.000000          0.100000      1.632069e-07                       |
|           0.200000          0.100000      5.286407e-07                       |
|           0.400000          0.100000      1.376396e-07                       |
|           0.600000          0.100000      1.431255e-07                       |
|           0.800000          0.100000      1.121649e-07                       |
|           1.000000          0.100000      1.175979e-07                       |
|           1.500000          0.100000      9.040193e-08                       |
+-------------------------------------------------------------------------------+
```

Gate Potential Alignment

The parameter `gate_potential_alignment` allows the definition of a reference for the gate-source potential which can rigidly shift V_{gs} - I_{ds} characteristics and take into account, for example, for different gate metal work functions.

In the following sections it is explained how the gate potential is applied in relation to the left and right electrode boundary conditions, and how `gate_potential_alignment` modifies the absolute gate potential.

Definition of the Gate potential with Dirichlet boundary conditions

The solutions for the bulk left and right electrodes, evaluated at the beginning of a device calculation, provide the energy and electrostatic potential references.

During the device calculation, the electronic structures of the electrodes are shifted to align the whole system to a common energy reference and properly include any built-in potential.

In order to ensure consistency between the electrostatic potential of the device and the common energy reference, the Dirichlet boundary conditions on the left surface of the simulation grid at equilibrium ($V_{ds} = 0$) is set as:

$$V(\mathbf{r}) = V_{bulk,left}(\mathbf{r}), \mathbf{r} \in \Omega_{left}$$

where $V_{bulk,left}(\mathbf{r})$ is the electrode bulk potential, $V_{applied,left}$ the voltage applied on the left contact and Ω_{left} the left boundary surface.

Similarly, on the right surface we have:

$$V(\mathbf{r}) = V_{bulk,right}(\mathbf{r}) + V_{built-in}, \mathbf{r} \in \Omega_{right}$$

where $-|e|V_{built-in} = \mu_{bulk,right} - \mu_{bulk,left}$ is the built-in potential due to the difference of chemical potential in the isolated electrodes at equilibrium.

When we apply a potential on a gate, the constraint $V(\mathbf{r}) = \hat{V}_g$ is imposed on all points within the gate region. For $V_{bulk,left}(\mathbf{r}) = 0$, or in general for electrodes with $V_{bulk,left}(\mathbf{r}) \neq 0$ where the bulk potential converge to a uniform zero potential in the vacuum, a zero gate potential will be aligned with the vacuum level of the electrode. We can interpret it as having a gate metal workfunction equal to the left electrode workfunction.

For electrodes with non-zero bulk potential and floating boundary conditions (i.e., a combination of Periodic and Neumann boundary conditions in all directions), the electrostatic potential is shifted such that its total average is zero. As a consequence, the vacuum potential is usually close to zero, however deviations up to tens of eV can arise and translate in an effective shift of the gate reference. The vacuum level of an electrode, hence the amount of this shift, can always be determined by inspecting the electrostatic potential as shown in the tutorial *InAs p-i-n junction*.

The parameter `gate_potential_alignment` allows one to modify the alignment between gate and electrode potentials without modifying the nominal V_{gs}. This corresponds to setting the metal gate work function, in the limit of zero electrode vacuum potential.

Note: The initial reference for the gate potential is not affected by the source-drain voltage. IV curves calculated with different settings of `gate_potential_alignment` are equivalent, apart from a rigid shift in the V_{gs} axis.

Definition of the Gate potential with Neumann boundary conditions

When Neumann boundary conditions are applied, the potential on the left and right face of the simulation box is not directly imposed. However, all the arguments in the section *Definition of the Gate potential with Dirichlet boundary conditions* are still valid. In fact, the correct solution will converge to the electrostatic potential of the electrode at the boundary at zero bias, same as if Dirichlet boundary conditions had been imposed.

Gate Potential Alignment option

The effective gate voltage set during an *IVCharacteristics* is given by:

$$\hat{V}_g = V_{gs} + V_{ref}$$

where V_{ref} is determined through the `gate_potential_alignment` parameter as:

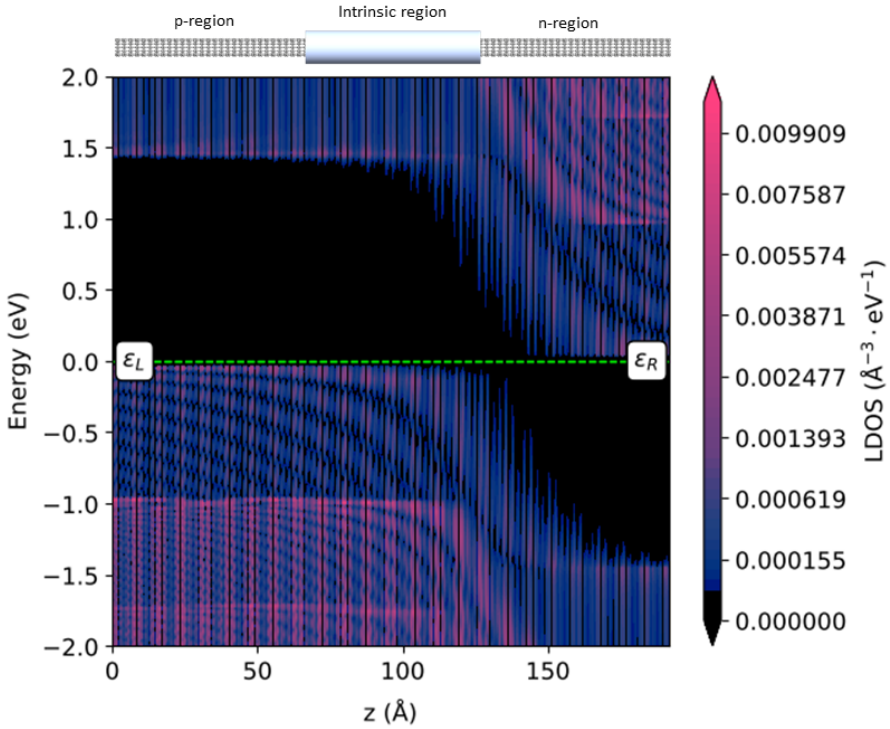
$$-|e|V_{ref} = (\phi_{ref} - \phi_{left} + \Delta\phi)$$

ϕ_{left}, ϕ_{ref} are the work functions of the isolated left electrode and an isolated bulk reference system, which can be set to the left or right electrode through the parameter `reference_electrode`. Additionally, `use_intrinsic_chemical_potential` allows to remove any doping from the reference electrode bulk configuration when calculating the work function difference. $\Delta\phi$ is the work function difference between the gate metal and the isolated reference bulk system. By default, $\phi_{ref} = \phi_{left}$ and $\Delta\phi = 0$.

The value of V_{ref} can be queried using the method `gateReferencePotential()`.

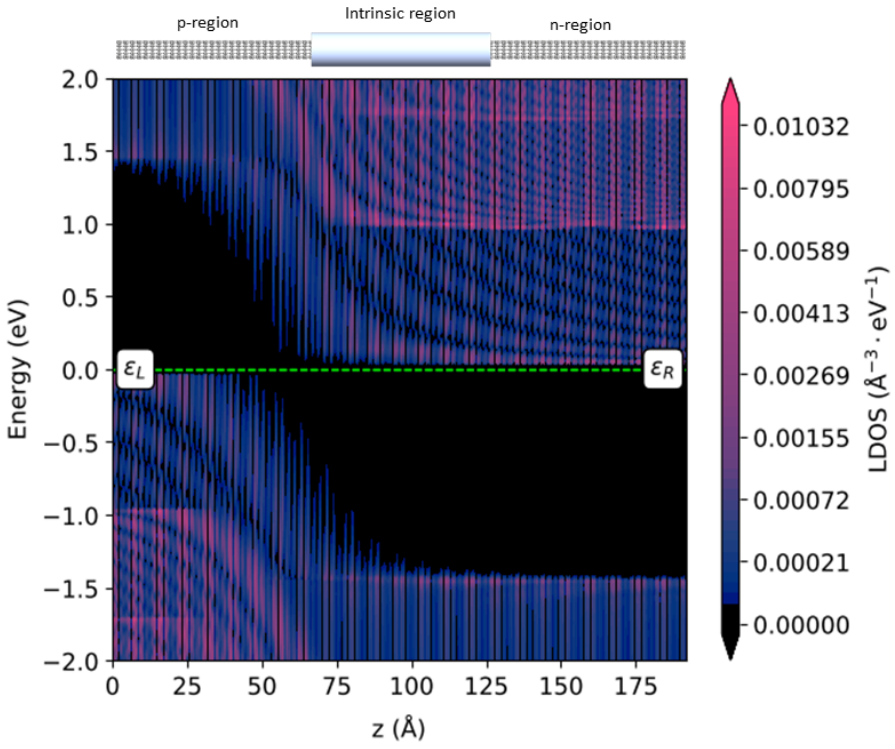
Example

The effect of `gate_potential_alignment` can be better understood by looking at the *ProjectedLocalDensityOfStates* for an example p-i-n junction. The figure below shows such result for a carbon nanotube with a gate in the central region, calculated with a semi-empirical calculator. You can download the script used to generate this picture: `cnt-gaa-pin.py`



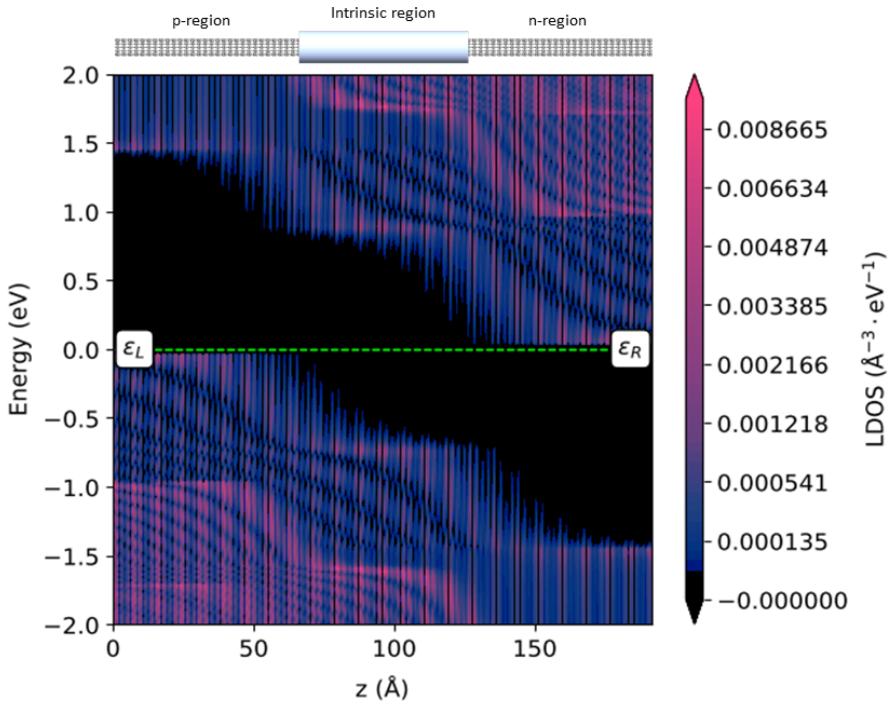
It can be seen that the conduction and valence band edges in the intrinsic region are aligned with the band edges in the left region. This is consistent with the fact that the gate potential is aligned by default with the left electrode potential.

If we want to modify the band alignment at equilibrium, we can do so by using `gate_potential_alignment`. The figure below is the *ProjectedLocalDensityOfStates* obtained from an *IVCharacteristics* with `gate_potential_alignment=GatePotentialAlignment(reference_electrode=Right)`. We can see that the band edges in the central region are now aligned with the right electrode.



If the chemical potentials of left and right electrode are already known, the same result can be obtained by setting directly their difference, e.g. `gate_potential_alignment=GatePotentialAlignment(work_function_difference=-1.4*eV)`.

The keyword `use_intrinsic_chemical_potential` can be used to set the intrinsic systems as a reference, that is, an intrinsic bulk system identical to the chosen electrode except for the absence of doping. The figure below shows the *ProjectedLocalDensityOfStates* obtained for *IVCharacteristics* with `gate_potential_alignment=GatePotentialAlignment(use_intrinsic_chemical_potential=True)`.



The parameter `use_intrinsic_chemical_potential` can also be used in combination with `work_function_difference`. This can be useful if the user wants to simulate a metal gate for which the difference between the intrinsic electrode material and the metal work function is known.

As an example, assuming that we want to describe a device with doped bulk Silicon electrodes and a gate metal with work function difference:

$$\phi_{gate} - \phi_{intrinsic} = \Delta\phi$$

where $\phi_{intrinsic}$ is the work function of intrinsic Silicon. The user can then set the parameters `use_intrinsic_chemical_potential=True` and `work_function_difference` equal to $\Delta\phi$.

IVCurve

class IVCurve(*configuration*, *biases=None*, *energies=None*, *kpoints=None*, *kpoints_weights=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *selfconsistent_configurations_filename_prefix=None*, *log_filename_prefix=None*)

Constructor for the IVCurve object.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which the IV curve should be calculated.
- **biases** (*PhysicalQuantity* of type Volt.) – A list of the bias voltages for which the current should be calculated. **Default:** `numpy.linspace(0, 2, 11)*Volt`
- **energies** (*PhysicalQuantity* of type energy.) – A list of the energies for which the IV curve transmission spectra should be calculated. **Default:** numpy.linspace(-2, 2, 101) * eV

- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | *AdaptiveGrid* | sequence of sequence (size 3) of float) – The k-points for which the IV Curve transmission spectra should be calculated. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

- **kpoints_weights** (*sequence of float*) – The weight of each k-point. **Default:** The weights corresponding to the MonkhorstPackGrid, or a list of [1.0, 1.0,] if k-points are specified as floats.

- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The SelfEnergyCalculator to be used for the transmission spectra. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`

- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`

- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the transmission calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

- **selfconsistent_configurations_filename_prefix** (*string | NoCheckpointHandler (no file is saved)*) – Prefix for the filenames where the individual SCF evaluations at different bias values are stored. The filenames are formed by appending a number and the default file extension (.hdf5) to this string. **Default:** `"ivcurve_selfconsistent_configuration_"`

- **log_filename_prefix** (*string | LogToStdOut*) – Prefix for the filenames where the logging output for each bias-value calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. **Default:** `"ivcurve_"`

biases()

Returns
The bias voltages for which the current is calculated.

Return type
PhysicalQuantity of type Volt.

currents()

Returns
The calculated currents at the specified bias voltages.

Return type
PhysicalQuantity of type Ampere.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

transmissionSpectra()

Returns
The calculated transmission spectra used for the current evaluations.

Return type
list of *TransmissionSpectrum*.

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Calculate the *IVCurve* for a given *DeviceConfiguration*:

```
# ------------------------------------------------------------
# Two-probe Configuration
# ------------------------------------------------------------

# ------------------------------------------------------------
# Left Electrode
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 9.0]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
left_electrode_elements = [Lithium, Lithium, Lithium]

# Define coordinates
left_electrode_coordinates = [[ 2.5,  2.5,  1.5],
                              [ 2.5,  2.5,  4.5],
                              [ 2.5,  2.5,  7.5]]*Angstrom

# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )

# ------------------------------------------------------------
# Right Electrode
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.0, 0.0, 0.0]*Angstrom
```

(continues on next page)

(continued from previous page)

```
vector_b = [0.0, 5.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 9.0]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
right_electrode_elements = [Lithium, Lithium, Lithium]

# Define coordinates
right_electrode_coordinates = [[ 2.5,  2.5,  1.5],
                               [ 2.5,  2.5,  4.5],
                               [ 2.5,  2.5,  7.5]]*Angstrom

# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )

# -----------------------------------------------------------
# Central Region
# -----------------------------------------------------------

# Set up lattice
vector_a = [5.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 22.0]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
central_region_elements = [Lithium, Lithium, Lithium, Hydrogen, Hydrogen, Lithium,␣
↪Lithium,
                           Lithium]

# Define coordinates
central_region_coordinates = [[  2.5,   2.5,   1.5],
                              [  2.5,   2.5,   4.5],
                              [  2.5,   2.5,   7.5],
                              [  2.5,   2.5,  10.5],
                              [  2.5,   2.5,  11.5],
                              [  2.5,   2.5,  14.5],
                              [  2.5,   2.5,  17.5],
                              [  2.5,   2.5,  20.5]]*Angstrom

# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )

device_configuration = DeviceConfiguration(
```

(continues on next page)

(continued from previous page)

```
    central_region,
    [left_electrode, right_electrode]
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Numerical Accuracy Settings
#---------------------------------------
left_electrode_k_point_sampling = MonkhorstPackGrid(
    na=1,
    nb=1,
    nc=100,
    )
left_electrode_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=left_electrode_k_point_sampling,
    )

right_electrode_k_point_sampling = MonkhorstPackGrid(
    na=1,
    nb=1,
    nc=100,
    )
right_electrode_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=right_electrode_k_point_sampling,
    )

#---------------------------------------
# Poisson Solver Settings
#---------------------------------------
left_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

right_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

#---------------------------------------
# Electrode Calculators
#---------------------------------------
left_electrode_calculator = HuckelCalculator(
    numerical_accuracy_parameters=left_electrode_numerical_accuracy_parameters,
    poisson_solver=left_electrode_poisson_solver,
    )

right_electrode_calculator = HuckelCalculator(
```

(continues on next page)

(continued from previous page)

```
    numerical_accuracy_parameters=right_electrode_numerical_accuracy_parameters,
    poisson_solver=right_electrode_poisson_solver,
    )

#----------------------------------------
# Device Calculator
#----------------------------------------
calculator = DeviceHuckelCalculator(
    electrode_calculators=
        [left_electrode_calculator, right_electrode_calculator],
    )

device_configuration.setCalculator(calculator)
nlprint(device_configuration)
device_configuration.update()
nlsave('ivcurve.nc', device_configuration)

# ------------------------------------------------------------
# IV Curve
# ------------------------------------------------------------
biases = [0.000000, 0.200000, 0.400000, 0.600000, 0.800000, 1.000000,
          1.200000, 1.400000, 1.600000, 1.800000, 2.000000]*Volt

iv_curve = IVCurve(
    configuration=device_configuration,
    biases=biases,
    energies=numpy.linspace(-2,2,101)*eV,
    kpoints=MonkhorstPackGrid(1,1),
    self_energy_calculator=RecursionSelfEnergy(),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    selfconsistent_configurations_filename_prefix="ivcurve_selfconsistent_configuration_
↪",
    log_filename_prefix="ivcurve_"
    )
nlsave('ivcurve.nc', iv_curve)
nlprint(iv_curve)
```

ivcurve.py

IdealGasThermoChemistry

class IdealGasThermoChemistry(*configuration*, *dynamical_matrix=None*)

IdealGasThermoChemistry making it easy to extract Gibbs free energy and entropy.

Parameters

- **configuration** (MoleculeConfiguration|BulkConfiguration) – The configuration.
- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix used to calculate the vibrational modes.

electronicEntropy(*spin*)

The electronic contribution to the entropy.

Parameters
spin (*integer or half-integer*) – The total spin.

Returns
The electronic entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

energy()

Returns
The total energy.

Return type
PhysicalQuantity of type energy

enthalpy(*temperature*, *include_translations=True*, *include_rotations=True*, *include_vibrations=True*)

The enthalpy, `H`, as a function of temperature.

The enthalpy is here defined by `H = E_{ZPE} + E_{electrons} + int_0^T dT C_p (T)`.

Parameters

- **temperature** (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.
- **include_translations** (*bool*) – `True`, if translational contributions to the enthalpy should be included. **Default:** `True`.
- **include_rotations** (*bool*) – `True`, if rotational contributions to the enthalpy should be included. **Default:** `True`.
- **include_vibrations** (*bool*) – `True`, if vibrational contributions to the enthalpy should be included. **Default:** `True`.

Returns
The enthalpy, `H(T)`.

Return type
PhysicalQuantity with unit energy

entropy(*temperature*, *pressure=PhysicalQuantity(1.0, bar)*, *include_translations=True*, *include_rotations=True*, *include_vibrations=True*)

The entropy.

Parameters

- **temperature** (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.
- **pressure** (*PhysicalQuantity* with unit pressure) – The (optional) gas pressure.
- **include_translations** (*bool*) – `True`, if translational contributions to the entropy should be included. **Default:** `True`.
- **include_rotations** (*bool*) – `True`, if rotational contributions to the entropy should be included. **Default:** `True`.
- **include_vibrations** (*bool*) – `True`, if vibrational contributions to the entropy should be included. **Default:** `True`.

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

freeEnergy(*temperature*, *pressure=PhysicalQuantity(1.0, bar)*, *include_translations=True*, *include_rotations=True*, *include_vibrations=True*)

The Gibbs free energy including the total energy contribution.

Parameters

- **temperature** (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.
- **pressure** (*PhysicalQuantity* with unit pressure) – The pressure. **Default:** `1 * bar`.
- **include_translations** (*bool*) – `True`, if translational contributions to the free energy should be included. **Default:** `True`.
- **include_rotations** (*bool*) – `True`, if rotational contributions to the free energy should be included. **Default:** `True`.
- **include_vibrations** (*bool*) – `True`, if vibrational contributions to the free energy should be included. **Default:** `True`.

Returns
The Gibbs free energy.

Return type
PhysicalQuantity with wiht unit energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

pressureEntropy(*pressure_0*, *pressure*)

The pressure contribution to the entropy.

Parameters

- **pressure_0** (*PhysicalQuantity* with of type pressure) – The reference pressure.
- **pressure** (*PhysicalQuantity* with of type pressure) – The gas pressure.

Returns
The pressure contribution to the entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

rotationalEntropy(*temperature*)

The rotational contribution to the entropy.

Parameters
temperature (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.

Returns
The rotational entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

translationalEntropy(*temperature*, *pressure_0=PhysicalQuantity(1.0, bar)*)

The translational contribution to the entropy.

Parameters

- **temperature** (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.
- **pressure_0** (*PhysicalQuantity* with unit pressure) – The reference gas pressure. **Default:** `1 * bar`.

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

uniqueString()

Return a unique string representing the state of the object.

vibrationalEntropy(*temperature*)

The vibrational contribution to the entropy.

Parameters
temperature (*PhysicalQuantity* with unit temperature) – The temperature or an array of temperatures.

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

zeroPointEnergy()

The zero point energy.

Returns
The zero point energy.

Return type
PhysicalQuantity with unit energy

Notes

IdealGasThermoChemistry implements the calculations of the Gibbs free energy in the ideal gas limit, as given by the standard derivation of ideal gas thermodynamic properties as done in e.g. C.J. Cramer, Essentials of Computational Chemistry.

The Gibbs free energy is given as a function of temperature, T,

$$G(T) = H(T) - TS$$

where $H(T)$ is the enthalpy, while S is the entropy.

In the ideal gas limit, the total enthalpy can be defined by

$$H(T) = E_{\text{electronic}} + E_{\text{ZPE}} + \int_0^T C_p \, \mathrm{d}T$$

where $E_{\mathrm{electronic}}$ is the electronic contribution to the energy (also sometimes called the total energy) and E_{ZPE} is the zero point energy. The heat capacity at constant pressure, C_p, can be divided up into components,

$$C_p = k_{\mathrm{B}} + C_{V,\mathrm{trans}} + C_{V,\mathrm{vib}} + C_{V,\mathrm{rot}} + C_{V,\mathrm{elec}},$$

where the translational contribution for an ideal gas is given by, $C_{V,\mathrm{trans}} = 3k_{\mathrm{B}}/2$. The integral over the vibrational contribution to the heat capacity, is given by,

$$\int_0^T C_{V,\mathrm{vib}}\,\mathrm{d}T = \sum_i^{\mathrm{dof}} \frac{\hbar\omega_i}{e^{\hbar\omega_i/k_{\mathrm{B}}T} - 1}$$

The sum runs over all vibrational degrees of freedom each with an ω_i vibrational frequency. Note that the number of degrees of freedom also depends on the molecule geometry, with monatomic molecules containing no internal vibrational degrees of freedom, linear molecule having $3N - 5$ degrees of freedom and nonlinear molecules having $3N - 6$ degrees of freedom (with N being the number of atoms).

The rotational contribution depends on the geometry of the molecule:

$$C_{V,\mathrm{rot}} = k_{\mathrm{B}} \begin{cases} 0 & \text{, for monatomic molecules;} \\ 1 & \text{, for linear molecules;} \\ \frac{3}{2} & \text{, for nonlinear molecules.} \end{cases}$$

Finally, the electronic contribution to the heat capacity can be assumed to be zero.

The ideal gas entropy can be calculated as a function of temperature and pressure,

$$\begin{aligned} S(T,P) &= S(T,P_0) - k_{\mathrm{B}} \ln \frac{P}{P_0} \\ &= S_{\mathrm{trans}} + S_{\mathrm{rot}} + S_{\mathrm{vib}} + S_{\mathrm{elec}} - k_{\mathrm{B}} \ln \frac{P}{P_0} \end{aligned}$$

While the last term (also referred to as the pressure entropy) depends on the reference pressure, P_0, this term is cancelled by an equal and opposite term in the translational contribution to the entropy as given by (including contributions from the Stirling approximations),

$$S_{\mathrm{trans}} = k_{\mathrm{B}} \ln \left(\left[\frac{2\pi M k_{\mathrm{B}} T}{h^2} \right]^{3/2} \frac{k_{\mathrm{B}} T}{P_0} \right) + \frac{5}{2} k_{\mathrm{B}}$$

Here M is the total mass of the molecule. The rotational contribution depends again on the molecule geometry,

$$S_{\mathrm{rot}} = k_{\mathrm{B}} \begin{cases} 0 & \text{, for monatomic molecules;} \\ \ln\left(\frac{8\pi^2 I k_{\mathrm{B}} T}{\sigma h^2}\right) + 1 & \text{, for linear molecules;} \\ \ln\left(\frac{\sqrt{\pi I_{\mathrm{A}} I_{\mathrm{B}} I_{\mathrm{C}}}}{\sigma} \left[\frac{8\pi^2 k_{\mathrm{B}} T}{h^2}\right]^{3/2}\right) + \frac{3}{2} & \text{, for nonlinear molecules.} \end{cases}$$

Here I_{A}, I_{B} and I_{C} are the principle moment of inertia, while I is the total moment of inertia. The number, σ, is the symmetry number for the molecule.

Finally the vibrational contribution is,

$$S_{\mathrm{vib}} = k_{\mathrm{B}} \sum_i^{\mathrm{dof}} \left[\frac{\hbar\omega_i}{k_{\mathrm{B}} T (e^{\hbar\omega_i/k_{\mathrm{B}}T} - 1)} - \ln\left(1 - e^{-\hbar\omega_i/k_{\mathrm{B}}T}\right) \right].$$

If the molecule has a non-zero total spin, one can also include that by adding the contribution,

$$S_{\mathrm{elec}} = k_{\mathrm{B}} \ln(2 \cdot \mathrm{spin} + 1).$$

ImageDependentPairPotential

class ImageDependentPairPotential(*spring_constant=None*, *constraints=None*, *break_symmetry=None*, *exponent=None*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *memory_size=None*)

Use the Image Dependent Pair Potential method for interpolating images.

Parameters

- **spring_constant** (*PhysicalQuantity* of type force per length) – The spring constant used between the images. **Default:** 5.0*eV/Ang**2.
- **constraints** (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained. **Default:** []
- **break_symmetry** (*bool*) – If the symmetry should be broken. This can be necessary for interpolation to work and to prevent unphysically close atoms if the endpoints are symmetry equivalent.
- **exponent** (*float*) – The exponent used in the 1/distance weight function. The larger the exponent the more short bonds are emphasized. **Default:** 4.0.
- **max_forces** (*PhysicalQuantity* of type eV/Ang) – The maximum forces when the optimization should stop. **Default:** 0.02*eV/Ang.
- **max_steps** (*int*) – The maximum number of optimization steps to take after adding an image. **Default:** 200.
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.
- **memory_size** (*int*) – The maximum number of previous steps that will be stored in memory in order to approximate the Hessian matrix. **Default:** 9

breakSymmetry()

Returns
True, if the symmetry should be broken.

Return type
bool

constraints()

Returns
The constrained atoms.

Return type
list of type int

exponent()

Returns
The exponent used in the 1/distance weight function. The larger the exponent the more short bonds are emphasized.

Return type
float

maxForces()

Returns
The maximum forces when the optimization should stop.

Return type
PhysicalQuantity of type eV/Ang

maxStepLength()

Returns
The maximum step length the optimizer may take.

Return type
PhysicalQuantity of type length

maxSteps()

Returns
The maximum number of optimization steps to take after adding an image.

Return type
int

memorySize()

Returns
The memory size which is use for the LBFGS method.

Return type
int

setConstraints(*constraints*)

Set the constraints.

Parameters
constraints (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained.

springConstant()

Returns
The spring constant used between the images.

Return type
PhysicalQuantity of type force per length

tags()

Returns
The tags used for defining the constraints.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

Notes

The *ImageDependentPairPotential*[1] (IDPP) class is a robust interpolation algorithm for generating initial images for a *NudgedElasticBand* (NEB) in Cartesian coordinates. Compared to linear interpolation, it can reduce the computational effort needed for optimizing the NEB path due to finding a initial guess closer to the minimum path, in particular if rotations are involved. Furthermore the IDPP can improve convergence of electronic structure calculations by avoiding atoms getting unphysically close.

[1] S.ăSmidstrup, A.ăPedersen, K.ăStokbro, and H.ăJónsson. Improved initial guess for minimum energy path calculations. *J. Chem. Phys.*, 140:214106, 2014. doi:10.1063/1.4878664.

The pairwise distances d_{ij} of atom pairs i, j are scaled linearly between the the initial (α) and final (β) images. This defines the target values for the pairwise distances at the image with index κ,

$$d_{ij}^{\kappa} = d_{ij}^{\alpha} + \kappa(d_{ij}^{\beta} - d_{ij}^{\alpha})/p,$$

with p being the total number of images.

Since there are more distances than independent degrees of freedom, the interpolated atomic coordinates cannot satisfy all distance constraints rigorously. Therefore the images in the IDPP-path are optimized using an objective function S_{κ}^{IDPP} that attains its minimum when the pair distances in an image are as close to the target values as possible:

$$S_{\kappa}^{IDPP} = \sum_{i}^{N}\sum_{j>i}^{N} \omega(d_{ij}) \left(d_{ij}^{\kappa} - \sqrt{\sum_{\sigma}(r_{i,\sigma} - r_{j,\sigma})^2} \right)^2$$

where $\sigma = x, y, z$ and $\omega = 1/d^{exponent}$ is a weight function whose exponent can be used to penalize short bonds.

ImageStressCorrection

class ImageStressCorrection(*bulk_modulus*)

A container for parameters for a strain energy correction based on the image stress correction for calculations of a charged point defect in a supercell of bulk material.

Parameters
bulk_modulus (*PhysicalQuantity* of type pressure.) – The bulk modulus of the host material. **Default:** None.

bulkModulus()

Returns
The bulk modulus of the host material.

Return type
PhysicalQuantity of type pressure.

uniqueString()

Return a unique string representing the state of the object.

For a description of the strain energy correction model, see *Finite-size scaling and correction terms* and references therein.

ImproperCosineTorsionPotential

class ImproperCosineTorsionPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *k*, *n*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type. This is the central atom when calculating the angle.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.
- **k** (*PhysicalQuantity of type energy*) – Potential parameter.
- **n** (*int*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

ImproperTorsionPotential

class ImproperTorsionPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *k*, *psi0*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type. This is the central atom when calculating the angle.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.
- **k** (*PhysicalQuantity of type energy / angle**2*) – Potential parameter.
- **psi0** (*PhysicalQuantity of type angle*) – Potential parameter (the equilibrium angle)

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a ImproperTorsionPotential for an ethene molecule.

```
# Set up improper dihedral indices for two SP2-carbon atoms.
improper_dihedral_indices = [[1, 0, 2, 3], [3, 4, 5, 1]]
molecule_configuration.setImproperDihedralIndices(improper_dihedral_indices)

# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethene_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=-0.2*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)

# Add improper torsion potential around SP2-carbon atoms.
improper_torsion_potential = ImproperTorsionPotential(
    particleType1='C',
    particleType2='H',
    particleType3='H',
    particleType4='C',
    k=0.1*eV/Radians**2,
    psi0=0.0*Radians,
)
potential_set.addPotential(improper_torsion_potential)
# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the ImproperTorsionPotential block of the script is shown. The full script can be found found in the file `ethene_improper_torsion_potential.py`.

Notes

•Improper torsion (also called improper dihedral) potentials are typically used in biomolecular force fields to model the out-of-plane torsion of a planar group of atoms, e.g. around an SP2-carbon atom in a benzene molecule. Improper torsion potentials work similar to regular torsion potentials (e.g. *CosineTorsionPotential*), however, the quadruple of atoms *(a,b,c,d)* is typically not connected by a sequence of bonds, as in regular torsion potentials.

•As they cannot be inferred from the bonded topology, improper dihedrals have to be defined separately. They can be set on a configuration via `configuration.setImproperDihedralIndices(improper_dihedral_indices)` where `improper_dihedral_indices` is a list or array of all improper dihedral quadruples that should be used for the configuration. Each entry in this list must be a sequence of 4 atom indices *(a,b,c,d)*.

•The improper torsion angle is defined by taking one of the atoms around the center as a common atom. So for instance if *a* is the central atom and *b* is taken as the common atom, the improper torsion angle ψ is then calculated as the angle between the two planes *(a,b,c)* and *(a,b,d)*. This is the equivalent of calculating the torsion angle of atoms *(c,a,b,d)*.

•Given the 4 atom indices *(a,b,c,d)* there are 3 possible improper torsions that can be defined by taking different atoms as the common atom. The ImproperTorsionPotential calculates the potential using all 3 definitions of the improper torsion angle. This means that only one potential needs to be added for each type of central planar atom.

•The improper torsion potential has a simple quadratic form where the energy V is given as:

$$V(\psi) = k\left(\psi - \psi_0\right)^2$$

Here ψ is the improper torsion angle, ψ_0 is the equilibrium improper torsion angle and k is the stiffness of the improper torsion.

InelasticIVCharacteristics

class InelasticIVCharacteristics(*configuration*, *filename*, *object_id*, *gate_regions=None*, *gate_source_voltages=None*, *drain_source_voltages=None*, *gate_potential_alignment=None*, *energies=None*, *kpoints=None*, *kpoints_weights=None*, *electron_phonon_self_energies=None*, *loa_order=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*, *source_electrode=None*, *max_initial_state_configurations=None*)

Constructor for the InelasticIVCharacteristics Study object.

InelasticIVCharacteristics can be used to calculate IV curves including electron-phonon interactions using the `OneShotSelfConsistentApproximation` object.

Parameters

•**configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which to perform the study. The configuration must include at least one metallic region acting as the gate.

•**filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.

•**object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.

•**gate_regions** (*int | list of int*) – An index or list of indices of the metallic regions present in the configuration for which to apply the gate voltage. **Default:** All metallic regions present in the configuration.

•**gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of the gate-source voltages for which the current will be calculated the next time the object is updated. It is assumed that the source electrode is grounded, and that the gate-source voltage is defined as `V_gs = V_g - V_s`. Therefore, in each transmission calculation, the source electrode voltage will be set to zero and the gate regions voltage to `V_gs`. **Default:** `[0.0] * Volt`

•**drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – A list of the drain-source voltages for which the current will be calculated the next time the object is updated. It is assumed that the source electrode is grounded, and that the drain-source voltage is defined as `V_ds = V_d - V_s`. Therefore, in each transmission calculation, the source electrode voltage will be set to zero and the left electrode voltage to `V_ds`. **Default:** `[0.5] * Volt`

•**gate_potential_alignment** (*GatePotentialAlignment*) – A parameter indicating the difference between the work function of the gate electrodes and the work function of a reference electrode, plus an optional energy shift. Such difference is used to determine the reference potential for `V_gs=0`, effectively shifting an `I-Vgs` characteristics. **Default:** `GatePotentialAlignment(reference_electrode=`Left`, shift=0.0*eV, use_intrinsic_chemical_potential=False)`

•**electron_phonon_self_energies** (list of instances of *AcousticDeformationPotentialSelfEnergy* | *OpticalDeformationPotentialSelfEnergy*) – A list of models used to determine the Electron-Phonon scattering self energy. If None is specified, no electron-phonon self energy is included.

•**loa_order** (*int*) – The order of the generalized lowest-order approximation (LOA) used in the *OneShotSelfConsistentBornApproximation* to calculate the interacting current. Must be a positive integer. **Default:** 1

•**energies** (list of *PhysicalQuantity* of type energy) – A list of the energies for which the *OneShotSelfConsistentBornApproximation* at each voltage should be calculated. **Default:** Energy range that covers the bias window, plus $30k_BT$.

•**kpoints** (*MonkhorstPackGrid* | *RegularKpointGrid* | *AdaptiveGrid* | list (size n_kpoints) of list (size 3) of float) – The k-points for which the *OneShotSelfConsistentBornApproximation* at each voltage should be calculated. Note that the k-points must be in the same xy-plane. Set `nc=1` for MonkhorstPackGrid and RegularKpointGrid. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

•**kpoints_weights** (*list (size n_kpoints) of float*) – The weight of each k-point for which the *OneShotSelfConsistentBornApproximation* at each voltage should be calculated. **Default:** The weights corresponding to the MonkhorstPackGrid/RegularKpointGrid, or equal weights if the k-points are specified as a list.

•**log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of the calculations, each to be stored in a separate file. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'iv_characteristics_'`

•**number_of_processes_per_task** (int | `AllProcesses` | *ProcessesPerNode* `) – The number of processes that will be used to execute each task. If this value is greater than or equal to the total number of available processes, each single task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. **Default:** `AllProcesses`, all available processes execute each task collaboratively.

•**source_electrode** (`Left` | `Right`) – Specify which electrode should be set as Source. **Default:** `Left`

•**max_initial_state_configurations** – Maximum number of previous converged configurations used to determine the initial state for a new bias point. If `1` is specified, the configuration closest in bias to the new bias point is used as initial state. If an integer `n > 1` is specified, then the `n` configurations closest in bias are used to extrapolate the initial state for the new bias point. It must be a positive integer. **Default:** `2`

•**max_initial_state_configurations** – int

addVoltages(*gate_source_voltages=None*, *drain_source_voltages=None*)

Add a list of gate-source and/or drain-source voltage points to the IVCharacteristics study. These will be calculated the next time the object is updated.

Parameters

•**gate_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – The list of gate-source voltages to add.

•**drain_source_voltages** (*PhysicalQuantity* of type voltage | sequence of *PhysicalQuantity* of type voltage) – The list of drain-source voltages to add.

calculatedVoltages()

Returns
The list of gate-source and drain-source voltage pairs for which a transmission spectrum has been calculated.

Return type
list of *PhysicalQuantity* of type voltage

configuration(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the updated configuration for the given gate-source and drain-source voltages.

This result will only be available after the calculation ran successfully.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The configuration associated with this voltage pair. If not available, returns None.

Return type
DeviceConfiguration | None

configurationCalculator(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the updated configurations calculator for the given gate-source and drain-source voltages.

This result will only be available after the calculation ran successfully.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The calculator associated with this voltage pair. If not available, returns None.

Return type
DeviceLCAOCalculator | *DeviceSemiEmpiricalCalculator* None

current(*gate_source_voltage*, *drain_source_voltage*)

The current calculated from the *OneShotSelfConsistentBornApproximation* analysis. The sign of the current is adjusted to be consistent with the choice of source electrode.

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The calculated current for the required gate-source and drain-source voltages. If not available, returns None.

Return type
PhysicalQuantity of type current | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

drainInducedBarrierLowering(*threshold_current*, *high_drain_source_voltage*, *low_drain_source_voltage*, *min_subthreshold_current=None*, *max_subthreshold_current=None*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*)

Calculate drain induced barrier lowering (DIBL). The threshold voltage is defined as the gate-source voltage for which the current has a user-defined value, evaluated at a high and a low source-drain voltage value.

The DIBL is calculated as the unitless ratio

$DIBL = \frac{V_{TH}^{low} - V_{TH}^{high}}{V_{DS}^{high} - V_{DS}^{low}}$

In order to obtain meaningful results, a current in the subthreshold regime must be specified.

Parameters

•**threshold_current** (*PhysicalQuantity* of type current) – The current for which the device is considered to be at the gate-source threshold.

•**high_drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage at the working point (i.e., the supply voltage).

•**low_drain_source_voltage** (*PhysicalQuantity* of type voltage) – The low drain voltage used as reference. It needs to be a finite value close to 0.

•**min_subthreshold_current** (*PhysicalQuantity* of type current) – The minimum value of current used to extract the subthreshold regime model. **Default:** threshold_current / 10

•**max_subthreshold_current** (*PhysicalQuantity* of type current) – The minimum value of current used to extract the subthreshold regime model. **Default:** threshold_current * 10

•**range_min** (*PhysicalQuantity* of type voltage) – The minimum gate-source voltage used for the processing. This parameter can be used to select the nmos-like or pmos-like section of the IV curve in bipolar devices. **Default:** No restriction.

•**range_max** (*bool*) – The maximum gate-source voltage in the required range. This parameter can be used to select the nmos-like or pmos-like section of the IV curve in bipolar devices. **Default:** No restriction.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated drain induced barrier lowering.

Return type
float

drainSourceSaturationVoltage(*gate_source_voltage*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the drain-source saturation voltage for a given gate-source voltage. It is assumed that a transition to saturation regime occurs in the interval specified by range_min and range_max. The value is estimated by fitting the data to a Schichman-Hodges model (SPICE Mosfet model 1).

Parameters

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage, for which the drain-source saturation voltage will be calculated.

•**range_min** (*PhysicalQuantity* of type voltage) – The minimum drain-source voltage in the required range. **Default:** No restriction.

•**range_max** (*bool*) – The maximum drain-source voltage in the required range. **Default:** No restriction.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated drain-source saturation voltage, fitted in the required range.

Return type
PhysicalQuantity of type voltage

drainSourceVoltages()

Returns
The list of all calculated and not calculated drain-source voltages. Any drain-source voltage which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of *PhysicalQuantity* of type voltage

`electronPhononSelfEnergies()`

Returns
The objects describing the electron-phonon self energy models.

Return type
sequence of *AcousticDeformationPotentialSelfEnergy* | *OpticalDeformationPotentialSelfEnergy*

`energies()`

Returns
The list of energies used for the *OneShotSelfConsistentBornApproximation* calculation at each voltage.

Return type
list of *PhysicalQuantity* of type energy

`filename()`

Returns
The filename where the study object is stored.

Return type
str

`gatePotentialAlignment()`

Returns
The object describing the alignment of the gate electrode potential.

Return type
class:~.GatePotentialAlignment:

`gateReferencePotential()`

Returns
The absolute shift in gate voltages introduced by a gate potential alignment, with respect to the internal default. If the constructor parameter *gate_potential_alignment* is defined with a non-default *reference_electrode* then additional tasks have to be executed. In that case the absolute shift in gate voltages is returned when the information is available, and *None* is returned otherwise.

Return type
PhysicalQuantity of type voltage | None

`gateRegions()`

Returns
The list of indices of the metallic regions present in the configuration which the gate voltage is applied to.

Return type
list of int

`gateSourceVoltages()`

Returns
The list of all calculated and not calculated gate-source voltages. Any gate-source voltage which has not yet been calculated will be calculated the next time the object is updated.

Return type
list of *PhysicalQuantity* of type voltage

isConverged(*gate_source_voltage*, *drain_source_voltage*)

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
Whether the calculator for the required gate-source and drain-source voltages has been converged. If not available, returns None.

Return type
bool | None

kpoints()

Returns
The k-points used for the *OneShotSelfConsistentBornApproximation* calculation at each voltage.

Return type
MonkhorstPackGrid | *RegularKpointGrid* | *AdaptiveGrid* | list (size n_kpoints) of list (size 3) of float

kpointsWeights()

Returns
The weight of each k-point used for the *OneShotSelfConsistentBornApproximation* calculation at each voltage.

Return type
list (size n_kpoints) of float

loaOrder()

Returns
The order of the LOA.

Return type
int

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

maxInitialStateConfigurations()

Returns
The maximum number of converged configurations used to determine the initial state for a new bias point.

Return type
int

nlinfo()

Returns
Structured information about the Study.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

onOffRatio(*drain_source_voltage*, *off_gate_source_voltage*, *on_gate_source_voltage*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the on/off ratio for the available data points at a given drain-source voltage.

Parameters

- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage which determines the gate-source voltage/current data set from which the on/off ratio will be calculated. **Default:** The first drain-source voltage in the IVCharacteristics study.
- **off_gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage corresponding to the off state.
- **on_gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage corresponding to the on state.
- **converged_only** – Whether to only include data points which have converged. **Default:** True
- **electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated on/off ratio in the required range.

Return type
float

oneShotSelfConsistentBornApproximation(*gate_source_voltage*, *drain_source_voltage*)

Retrieve the OneShotSelfConsistentBornApproximation for the updated configuration for the given gate-source and drain-source voltages.

This result will only be available after the one shot scba calculation ran successfully.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.

Returns
The one shot SCBA object associated with this voltage pair. If not available, returns None.

Return type
OneShotSelfConsistentBornApproximation | None

removeDrainSourceVoltage(*drain_source_voltage*)

Remove a drain-source voltage from the IVCharacteristics study. All analysis objects corresponding to the given drain-source voltage will be removed.

Parameters
drain_source_voltage (*PhysicalQuantity* of type voltage) – The drain-source voltage which should be removed from the IVCharacteristics study.

removeGateSourceVoltage(*gate_source_voltage*)

Remove a gate-source voltage from the IVCharacteristics study. All analysis objects corresponding to the given gate-source voltage will be removed.

Parameters
gate_source_voltage (*PhysicalQuantity* of type voltage) – The gate-source voltage which should be removed from the IVCharacteristics study.

results(*gate_source_voltage*, *drain_source_voltage*, *result_types=None*)

Retrieve all results of the specified types for the given gate-source and drain-source voltages.

Only the results of calculations that ran successfully are available.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.
- **result_types** (*list*) – A list of types which should be included in the returned quantity. For example, result_types=[DeviceConfiguration] will only return instances of DeviceConfiguration. If not specified, all results are returned.

Returns
The available results from calculations associated with this voltage pair. If none available, it returns an empty list.

Return type
list

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

setCalculator(*gate_source_voltage*, *drain_source_voltage*, *calculator*, *initial_state=None*, *initial_spin=None*)

Set a new calculator and initial state for an existing gate-source and drain-source voltage pair.

The next call to update() will recalculate the configuration, transmission spectrum and any additional Analysis objects for this gate-source and drain-source voltage pair. If a configuration with an identical calculator had already converged, no calculation will be performed.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.
- **calculator** (*DeviceLCAOCalculator* | *DeviceSemiEmpiricalCalculator*) – The new calculator to be set on the configuration.
- **initial_state** (*DeviceConfiguration*) – The initial state to be used for the configuration. **Default:** The initial state is automatically chosen from the already run tasks.
- **initial_spin** (*InitialSpin* | None) – An initial spin to be applied in combination with the initial state. It can only be defined if an initial state is also defined. Alternatively, a global initial spin to be used during the first configuration update can be set directly on the configuration passed to the class constructor.

setOneShotSelfConsistentBornApproximationArguments(*gate_source_voltage*, *drain_source_voltage*, *one_shot_scba_arguments=None*)

Set new arguments for the OneShotSelfConsistentBornApproximation object for an existing gate-source and drain-source voltage pair.

The next call to update() will recalculate the transmission spectrum for this gate-source and drain-source voltage pair.

Parameters

- **gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage.
- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage.
- **one_shot_scba_arguments** (*A dictionary of valid arguments for a OneShotSelfConsistentBornApproximation object.*) – Dictionary of arguments to be passed to the OneShotSelfConsistentBornApproximation object. The configuration argument must not be present. **Default:** An empty dictionary.

sourceElectrode()

Return the side of the source electrode, specified as `Left` or `Right`. The quantity is used to convert from absolute electrode voltages to drain-source and gate-source voltages.

subthresholdSlope(*drain_source_voltage=None*, *range_min=None*, *range_max=None*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the subthreshold slope for the available data points at a given drain-source voltage.

Note that by default the entire range is used for the fitting, which might give an unphysical result. It is strongly recommended for the user to define a physical range in which to perform the fitting.

Parameters

- **drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage which determines the gate-source voltage/current data set from which the subthreshold slope will be calculated.
- **range_min** (*PhysicalQuantity* of type voltage) – The minimum gate-source voltage in the required range. **Default:** No restriction.
- **range_max** (*bool*) – The maximum gate-source voltage in the required range. **Default:** No restriction.
- **converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The calculated subthreshold slope, fitted in the required range. Units are mV/dec.

Return type
float

transconductance(*drain_source_voltage*, *gate_source_voltage*, *converged_only=None*, *electrode_temperatures=None*)

Calculate the transconductance, defined as:

$g_m = \partial I_D / \partial V_{GS}$

at a given drain-source voltage, where I_D is the drain current and V_{GS} the gate-source voltage. The transconductance is evaluated around the gate-source voltage specified in input. The derivative is calculated by finite difference, using the available currents.

Parameters

•**drain_source_voltage** (*PhysicalQuantity* of type voltage) – The drain-source voltage at which the transconductance is calculated,

•**gate_source_voltage** (*PhysicalQuantity* of type voltage) – The gate-source voltage at which the transconductance is evaluated.

•**converged_only** – Whether to only include data points which have converged. **Default:** True

•**electrode_temperatures** (list (size 2) of *PhysicalQuantity* of type temperature) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

Returns
The value of transconductance at the given gate-source and drain-source voltage.

Return type
PhysicalQuantity of type current / voltage.

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study.

InelasticTransmissionSpectrum

class InelasticTransmissionSpectrum(*configuration*, *dynamical_matrix=None*, *hamiltonian_derivatives=None*, *bulk_dynamical_matrix=None*, *bulk_hamiltonian_derivatives=None*, *energies=None*, *kpoints=None*, *qpoints=None*, *method=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *phonon_modes=None*, *phonon_energy_minimum=None*, *phonon_energy_maximum=None*, *phonon_energy_intervals=None*, *fermis_golden_rule_only=None*, *spectral_representation=None*, *electrode_extensions=None*, *store_coupling_matrices=None*, *reuse_tolerance=None*, *voltage_fraction_left=None*)

Class for calculating the inelastic transmission spectrum of a device configuration using the eXtended Lowest Order Expanssion (XLOE) method or Lowest Order Expanssion (LOE) method.

Parameters

- **configuration** (*DeviceConfiguration*) – The DeviceConfiguration for which the inelastic transmission should be calculated.
- **dynamical_matrix** (*DynamicalMatrix*) – A DynamicalMatrix object constructed for the same DeviceConfiguration as this object. This option is mutually exclusive to `bulk_dynamical_matrix`.
- **hamiltonian_derivatives** (*HamiltonianDerivatives*) – A HamiltonianDerivatives object constructed for the same DeviceConfiguration and calculator as this object. This option is mutually exclusive to `bulk_hamiltonian_derivatives`.
- **bulk_dynamical_matrix** (*DynamicalMatrix*) – A DynamicalMatrix object constructed for the same DeviceConfiguration as this object. This option can only be used if the central region of the device configuration `configuration` can be obtained as `N` repetitions of the bulk configuration used to calculate the `bulk_dynamical_matrix`. This option is mutually exclusive to `dynamical_matrix`.
- **bulk_hamiltonian_derivatives** (*HamiltonianDerivatives*) – A HamiltonianDerivatives object constructed for a bulk configuration. This option can only be used if the central region of the device configuration `configuration` can be obtained as `N` repetitions of the bulk configuration used to calculate the `bulk_hamiltonian_derivatives`. This option is mutually exclusive to `hamiltonian_derivatives`.
- **energies** (A *PhysicalQuantity* list with an energy unit.) – The energy for which the inelastic transmission function is calculated. The energy is calculated relative to the Fermi energy. **Default:** `[0.0]*eV`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – The k-points for the incoming electrons. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. **Default:** `[[0.0, 0.0, 0.0]]`
- **qpoints** – The transverse q-points for which the phonons should be calculated. This can either be given as a q-point grid or a list of fractional q-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. Outgoing electrons are scatered to k + q. **Default:** `[[0.0, 0.0, 0.0]]`
- **method** (`XLOE` | `LOE`) – The method used for calculating the inelastic transmissions. **Default:** `XLOE`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* with energy unit.) – Small energy, used to move the self-energy calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`
- **phonon_modes** (*A list of non-negative integers.*) – List with indices of phonon modes to consider. **Default:** `range(3*number_of_atoms)`. Include all modes.
- **phonon_energy_minimum** (*PhysicalQuantity* with energy unit.) – A lower limit for phonons to be considered. **Default:** `0.0*eV`
- **phonon_energy_maximum** (*PhysicalQuantity* with energy unit.) – An upper limit for phonons to be considered. This could be the bias voltage, if configuration is calculated at finite bias voltages. **Default:** `1.0*eV`
- **phonon_energy_intervals** (A *PhysicalQuantity* list with an energy unit.) – A list of energies. In each interval, between energy points i and i+1, all phonon modes will be summed to an effective phonon mode with an average energy of all the modes. Specifying this variable will overwrite the actual phonon modes. If the phonon_energy_intervals list has N energy points, there will effectively be N-1 phonon modes in the calculation for each (k,q) point. **Default:** `None`
- **fermis_golden_rule_only** (*bool*) – If True, only the terms $Tr(MA_LMA_R)$ are included in the inelastic transmissions. This is well-suited for e.g. p-i-n like structures. If true, then spectral_representation must be set to True (default). **Default:** `False`

•**spectral_representation** (*bool*) – If True, a spectral representation will be used. Calculations will typically be much faster. **Default:** `True`

•**electrode_extensions** (*list of two non-negative integers larger then 1 or [0,0]*) – The left and right electrode will be extended by an integer number of full electrode sizes into the central region. The electrode extensions are treated as non-interacting. The electrode extensions must be short enough such that there is no direct overlap between the left and right electrode extensions. **Default:** `[0,0]`

•**store_coupling_matrices** – Boolean to control if the electron-phonon coupling matrices should be precalculated and stored (True) or calculated on the fly (False). The option False saves memory, but is computationally slower. **Default:** `True`

•**voltage_fraction_left** (*float in range [0;1] with no unit.*) – Voltage drop asymmetry used in XLOE. Voltage fraction at left electrode used in the energy shift in the Greens functions $G(E \pm \hbar\omega \cdot f_L)$ where f_L is the `voltage_fraction_left`. 0.5 corresponds to a symmetric energy shift, while 1.0 means that all the energy shift is in the left chemical potential and 0.0 means that only the right electrode potential is shifted. **Default:** `0.5`

elasticConductance(*bias=None*, *temperature=None*, *fermi_shift=None*, *k_indices=None*, *spin=None*)

Returns the elastic contribution to the differential conductance.

Parameters

•**bias** (*PhysicalQuantity* of type voltage.) – The voltages for which the conductance should be calculated. **Default:** `numpy.linspace(0, 0.1, 100) * Volt`

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature for which the conductance should be calculated. **Default:** `300 * Kelvin`

•**fermi_shift** (*PhysicalQuantity* of type energy.) – Shift of Fermi level, such that the conductance is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0 * eV`

•**k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0, 1, 2, , number_of_kpoints-1]

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin flag. **Default:** `Spin.Up`

Returns
The elastic conductance for each bias point.

Return type
PhysicalQuantity of type Siemens.

elasticCurrent(*bias=None*, *temperature=None*, *fermi_shift=None*, *k_indices=None*, *spin=None*)

Returns the elastic contribution to the current

Parameters

•**bias** (*PhysicalQuantity* of type voltage.) – The voltages for which the current should be calculated. **Default:** `numpy.linspace(0,0.1,100)*Volt`

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature for which the current should be calculated. **Default:** `300*Kelvin`

•**fermi_shift** (*PhysicalQuantity* of type energy.) – Shift of Fermi level, such that the current is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0*eV`

•**k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0, 1, 2, , number_of_kpoints-1]

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin flag. **Default:** `Spin.Up`

Returns
The elastic current for each bias point.

Return type
PhysicalQuantity of type Ampere.

`elasticTransmission()`

Returns
The elastic transmission of shape (*N*, *K*, *S*) where *N* is the number of energies, *K* is the number of k-points, and *S* is the number of spins.

Return type
numpy.ndarray.

`electrodeFermiLevels()`

Returns
The electrode Fermi levels in absolute energies.

Return type
PhysicalQuantity of type energy.

`energies()`

Returns
The energies at which the inelastic transmission spectrum is calculated.

Return type
PhysicalQuantity of energy unit.

`energyZero()`

Returns
The energy zero used for the energy scale in this inelastic transmission spectrum.

Return type
PhysicalQuantity of type energy.

`energyZeroParameter()`

Returns
The energy zero parameter used for the energy scale in this inelastic transmission spectrum.

Return type
`AbsoluteEnergy | AverageFermiLevel`

`evaluate()`

Returns

Return a tuple with *(energies, T0, Tsym_pos, Tsy,_neg, Tasym_pos, Tasym_neg)*, where

energies : is the list of energies given in the constructor

T0 : is the elastic transmission spectrum

Tsym_pos : is the symmetric contribution to the inelastic transmission at positive bias

Tsym_neg : is the symmetric contribution to the inelastic transmission at negative bias

Tasym_pos : is the asymmetric contribution to the inelastic transmission at positive bias

Tasym_neg : is the asymmetric contribution to the inelastic transmission at negative bias

Return type
tuple

inelasticConductance(*bias=None*, *temperature=None*, *modes=None*, *fermi_shift=None*, *k_indices=None*, *q_indices=None*, *spin=None*, *tolerance=None*)

Returns the inelastic contribution to the differential conductance.

Parameters

•**bias** (*PhysicalQuantity* of type voltage.) – The voltages for which the conductance should be calculated. **Default:** `numpy.linspace(0, 0.1, 100) * Volt`

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature for which the conductance should be calculated. **Default:** `300 * Kelvin`

•**modes** (*A list of non-negative integers.*) – If specified, the inelastic conductance is only calculated for these modes. **Default:** All the modes, i.e. [0, 1, 2, , N-1], where N is the number of degrees of freedom.

•**fermi_shift** (*PhysicalQuantity* of type energy.) – Shift of Fermi level, such that the conductance is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0 * eV`

•**k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0, 1, 2, , number_of_kpoints-1]

•**q_indices** (*list.*) – A list of non-negative integers with the indices for the q-points to include. **Default:** All the q-points, i.e. [0, 1, 2, , number_of_qpoints-1]

•**spin** (`Spin.Up` | `Spin.Down`) – The spin flag. **Default:** `Spin.Up`

•**tolerance** (*float.*) – If specified, only phonon modes with an inelastic transmission larger than tolerance*max(inelasticTransmission) will be included. **Default:** 0.0 (all modes will be included).

Returns
The inelastic conductance for each bias point.

Return type
PhysicalQuantity of type Siemens.

inelasticCurrent(*bias=None*, *temperature=None*, *modes=None*, *fermi_shift=None*, *k_indices=None*, *q_indices=None*, *spin=None*, *tolerance=None*)

Returns the inelastic contribution to the current.

Parameters

•**bias** (*PhysicalQuantity* with voltage unit.) – The voltages for which the current should be calculated. **Default:** `numpy.linspace(0,0.1,100)*Volt`

•**temperature** (*PhysicalQuantity* with temperature unit.) – The temperature for which the current should be calculated. **Default:** `300*Kelvin`

•**modes** (*A list of non-negative integers.*) – If specified, the inelastic current is only calculated for these modes. **Default:** All the modes, i.e. [0,1,2,,N-1], where N is the number of degrees of freedom.

•**fermi_shift** (*PhysicalQuantity* with energy unit.) – Shift of Fermi level, such that the current is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0*eV`

•**k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0,1,2,,number_of_kpoints-1]

•**q_indices** (*list.*) – A list of non-negative integers with the indices for the q-points to include. **Default:** All the q-points, i.e. [0,1,2,,number_of_qpoints-1]

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin flag. **Default:** `Spin.Up`

•**tolerance** (*float.*) – If specified, only phonon modes with an inelastic transmission larger than tolerance*max(inelasticTransmission) will be included. **Default:** 0.0 (all modes will be included).

Returns
The inelastic current for each bias point.

Return type
PhysicalQuantity of type Ampere.

inelasticCurrentIntegral(*bias=None*, *temperature=None*, *modes=None*, *fermi_shift=None*, *k_indices=None*, *q_indices=None*, *spin=None*, *tolerance=None*, *interpolation_factor=None*)

Returns the inelastic contribution to the current calculate by integrating symmetric transmissions using the current expression from J. Appl. Phys. 109, 124503 (2011).

Parameters

•**bias** (*PhysicalQuantity* of type voltage.) – The voltages for which the current should be calculated. **Default:** `numpy.linspace(0, 0.1, 100)*Volt`

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature for which the transmission spectrum should be calculated. **Default:** `300 * Kelvin`

•**modes** (*A list of non-negative integers.*) – If specified, the inelastic current is only calculated for these modes. **Default:** All the modes, i.e. [0, 1, 2, , N-1], where N is the number of degrees of freedom.

•**fermi_shift** (*PhysicalQuantity* of type energy.) – Shift of Fermi level, such that the current is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0 * eV`

•**k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0, 1, 2, , number_of_kpoints-1]

•**q_indices** (*list.*) – A list of non-negative integers with the indices for the q-points to include. **Default:** All the q-points, i.e. [0, 1, 2, , number_of_qpoints-1]

•**spin** (Spin.Up | Spin.Down | `Spin.All`) – The spin flag. **Default:** `Spin.Up`

•**tolerance** (*float.*) – If specified, only phonon modes with an inelastic transmission larger than tolerance*max(inelasticTransmission) will be included. **Default:** 0.0 (all modes will be included).

•**interpolation_factor** (*int*) – Integer determining interpolation refinement of the energies and transmissions. **Default:** 1

Returns
The inelastic current for each bias point.

Return type
PhysicalQuantity of type Ampere.

inelasticElectronTunnelingSpectroscopy(*bias=None*, *temperature=None*, *modes=None*, *fermi_shift=None*, *k_indices=None*, *q_indices=None*, *spin=None*, *tolerance=None*)

Returns the inelastic tunneling spectrocopy signal (IETS), d^2I/dV^2

Parameters

•**bias** (*PhysicalQuantity* of type voltage.) – The voltages for which the IETS should be calculated. **Default:** `numpy.linspace(0, 0.1, 100) * Volt`

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature for which the IETS should be calculated. **Default:** `300 * Kelvin`

•**modes** (*A list of non-negative integers.*) – If specified, the IETS is only calculated for these modes. **Default:** All the modes, i.e. [0, 1, 2, , N-1], where N is the number of degrees of freedom.

•**fermi_shift** (*PhysicalQuantity* with energy unit.) – Shift of Fermi level, such that the IETS is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** `0.0 * eV`

- **k_indices** (*list.*) – A list of non-negative integers with the indices for the k-points to include. **Default:** All the k-points, i.e. [0, 1, 2, , number_of_kpoints-1]
- **q_indices** (*list.*) – A list of non-negative integers with the indices for the q-points to include. **Default:** All the q-points, i.e. [0, 1, 2, , number_of_qpoints-1]
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin flag. **Default:** `Spin.Up`
- **tolerance** (*float.*) – If specified, only phonon modes with an inelastic transmission larger than tolerance*max(inelasticTransmission) will be included. **Default:** 0.0 (all modes will be included).

Returns
The inelastic tunneling spectrocopy signal for each bias point.

Return type
PhysicalQuantity of type Siemens/Volt.

`inelasticTransmissionAsymmetricNegativeBias`(*mode=None*)

Return the asymmetric inelastic transmission for each mode for negative bias voltages.

Parameters
mode (int | `All`) – The phonon mode for which to return the inelastic transmission. If not provided, the result for all modes will be returned. **Default:** `All`

Returns
The asymmetric negative inelastic transmission. If mode=All, the shape of the returned array is (number_of_degrees_of_freedom, number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins). If mode=integer, the shape is (number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins).

Return type
numpy.ndarray.

`inelasticTransmissionAsymmetricPositiveBias`(*mode=None*)

Return the asymmetric inelastic transmission for each mode for positive bias voltages.

Parameters
mode (int | `All`) – The phonon mode for which to return the inelastic transmission. If not provided, the result for all modes will be returned. **Default:** `All`

Returns
The asymmetric positive inelastic transmission. If mode=All, the shape of the returned array is (number_of_degrees_of_freedom, number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins). If mode=integer, the shape is (number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins).

Return type
numpy.ndarray.

`inelasticTransmissionSymmetricNegativeBias`(*mode=None*)

Return the symmetric inelastic transmission for each mode for negative bias voltages.

Parameters
mode (int | `All`) – The phonon mode for which to return the inelastic transmission. If not provided, the result for all modes will be returned. **Default:** `All`

Returns
The symmetric negative inelastic transmission. If mode=All, the shape of the returned array is (number_of_degrees_of_freedom, number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins). If mode=integer, the shape is (number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins).

Return type
numpy.ndarray.

inelasticTransmissionSymmetricPositiveBias(*mode=None*)

Return the symmetric inelastic transmission for each mode for positive bias voltages.

Parameters
mode (int | All) – The phonon mode for which to return the inelastic transmission. If not provided, the result for all modes will be returned. **Default:** All

Returns
The symmetric positive inelastic transmission. If mode=All, the shape of the returned array is (number_of_degrees_of_freedom, number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins). If mode=integer, the shape is (number_of_energies, number_of_kpoints, number_of_qpoints, number_of_spins).

Return type
numpy.ndarray.

infinitesimal()

Returns
The infinitesimal used in self-energy calculations.

Return type
PhysicalQuantity of type energy.

interactionRegion()

Returns
The atom indices for the interaction region.

Return type
list of ints

kpoints()

Returns
The k-points.

Return type
list of floats with shape (K, 3), where K is the number of k-points.

maximumPhononEnergy()

Returns
Return the maximally allowed phonon energy.

Return type
PhysicalQuantity of type energy.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Returns
The method used for the calculation.

Return type
XLOE | LOE

minimumPhononEnergy()

Returns
The minimally allowed phonon energy.

Return type
PhysicalQuantity of type energy.

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

phononEnergies()

Returns
The phonon energies for all the modes.

Return type
PhysicalQuantity of energy unit.

phononModes()

Returns
The phonon mode indices.

Return type
list of ints

qpoints()

Returns
The q-points.

Return type
list of floats with shape (Q, 3), where Q is the number of q-points.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate a *InelasticTransmissionSpectrum*. Notice that the *DeviceConfiguration*, the *DynamicalMatrix* and the *HamiltonianDerivatives* are read from a file `filename.nc` from a separate calculation.

```
# Example of inelastic transmission spectrum

# Load device configuration
device_configuration = nlread('filename.nc', DeviceConfiguration)[0]

# Load dynamical matrix
```

(continues on next page)

(continued from previous page)

```
dynamical_matrix = nlread('filename.nc', DynamicalMatrix)[0]

# Load hamiltonian derivatives
hamiltonian_derivatives = nlread('filename.nc', HamiltonianDerivatives)[0]

# ------------------------------------------------------------
# Inelastic transmission spectrum
# ------------------------------------------------------------
inelastic_transmission_spectrum = InelasticTransmissionSpectrum(
    configuration=device_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    energies=numpy.linspace(0, 0, 1)*eV,
    kpoints=MonkhorstPackGrid(1, 1),
    qpoints=MonkhorstPackGrid(1, 1),
    method=XLOE,
    phonon_modes=All,
    )

# ------------------------------------------------------------
# Post processing of data
# ------------------------------------------------------------

# Setup bias voltages
bias = numpy.linspace(-0.1, 0.1, 1000)*Volt

# Set temperature
temp = 10*Kelvin

# Calculate the inelastic current
I = inelastic_transmission_spectrum.inelasticCurrent(bias=bias, temperature=temp)

# Calculate the inelastic conductance
G = inelastic_transmission_spectrum.inelasticConductance(bias=bias, temperature=temp)
```

Large Device Calculations

In order to calculate the *InelasticTransmissionSpectrum* for a large device with thousands of atoms, it is possible to apply certain approximations that will speed up the calculations significantly.

Summing up phonon modes in energy intervals

The first option is to specify `phonon_energy_intervals` instead of `phonon_modes=All` like:

```
inelastic_transmission_spectrum = InelasticTransmissionSpectrum(
    configuration=device_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    phonon_energy_intervals=numpy.linspace(0, 0.1, 10)*eV,
    )
```

For a device with, say, 1000 atoms in the central region, there will be 3000 phonon modes. However, by specifying `phonon_energy_intervals` the phonon modes will be summed in each interval to form new effective phonon modes. The number of intervals required will depend on the actual system, but typical values between 10 and 50 will often suffice, thus dramatically reducing the number of required calculations.

Note: When analyzing the results of an *InelasticTransmissionSpectrum* calculated with the `phonon_energy_intervals` parameter, the number of phonon modes one can query for will be the number of intervals used in the calculation.

In the above example, the intervals will be `[0, 0.01]*eV, [0.01, 0.02]*eV, ...`. The ith effective phonon mode corresponding to the energy interval $[\epsilon_i^0, \epsilon_i^1]$ will then be defined as (see also notes below)

$$\tilde{\mathbf{v}}^i = \sum_{\lambda \,|\, \epsilon_i^0 < \hbar\omega_\lambda < \epsilon_i^1} \mathbf{u}^\lambda \sqrt{\frac{\hbar}{2M_I\omega_\lambda}}$$

Much fewer calculations then need to be performed in *InelasticTransmissionSpectrum* and the calculation time will be reduced very significantly for large devices.

Note: The use of `phonon_energy_intervals` is an approximation. It is intented to be used in large device calculations, where the primary focus is on the total current. It should not be used for, e.g., a molecular junction, where interest is on the inelastic signal from individual phonons.

Using bulk DynamicalMatrix and HamiltonianDerivatives

Many device calculations are performed for a structure which is translationally invariant in the C-direction, apart from doping profiles and electrostatic regions. This might be the case for a field effect transistor with a nin-juction or a pn-junction where the electrodes and the central region are composed of the same material.

In that case, it is possible to calculate the *DynamicalMatrix* and *HamiltonianDerivatives* for the smallest repeatable unit cell, provided that the central region atomic structure can be represented as:

```
central_region = bulk_configuration.repeat(nc=nc)
```

where `nc` is the number of times the small bulk configuration should be repeated in order to form the central region.

The calculation of the *DynamicalMatrix* and *HamiltonianDerivatives* for the small `bulk_configuration` will be much faster than the corresponding calculations for the full device configuration.

The *InelasticTransmissionSpectrum* can then be calculated as follows:

```
# Calculate the bulk hamiltonian derivatives for the small system.
bulk_hamiltonian_derivatives = HamiltonianDerivatives(bulk_configuration)

# Calculate the bulk dynamical matrix for the small system.
bulk_dynamical_matrix = DynamicalMatrix(bulk_configuration)

# Calculate the inelastic transmission spectrum using bulk dynamical matrix
# and bulk hamiltonian derivatives.
inelastic_transmission_spectrum = InelasticTransmissionSpectrum(
    configuration=device_configuration,
    bulk_dynamical_matrix=bulk_dynamical_matrix,
```

(continues on next page)

(continued from previous page)

```
    bulk_hamiltonian_derivatives=bulk_hamiltonian_derivatives,
    phonon_energy_intervals=numpy.linspace(0, 0.1, 10)*eV,
    )
```

By using both the `phonon_energy_intervals`, `bulk_dynamical_matrix` and `bulk_hamiltonian_derivatives` it is feasible to perform *InelasticTransmissionSpectrum* calculations for devices with several thousand of atoms.

Note: Using the `bulk_dynamical_matrix` and `bulk_hamiltonian_derivatives` relies on the assumption that the dynamical matrix and the derivative of the Hamiltonian are translational invariant, although the Hamiltonian itself is not. In, e.g., a pn-junction the Hamiltonian is not translational invariant due to the doping profile. However, it might be a reasonable approximation to assume that the derivative is approximately translational invariant thus justifying the use of `bulk_hamiltonian_derivatives`.

It is not possible to use `bulk_dynamical_matrix` and `bulk_hamiltonian_derivatives` for device configurations containing interfaces between different materials.

Notes

In order to calculate the *InelasticTransmissionSpectrum* three main ingredients are needed:

- A *DeviceConfiguration* to get the Hamiltonian of the system.
- A *DynamicalMatrix* object for calculating the phonon modes.
- A *HamiltonianDerivatives* object for calculating the electron-phonon coupling matrix, M^{λ}, for a particular phonon mode, λ.

The device configuration used as input to *InelasticTransmissionSpectrum* must be exactly the same as used for the *HamiltonianDerivatives* and *DynamicalMatrix* objects. As explained above, for device configurations with a translationally invariant central region, it is possible to calculate the *HamiltonianDerivatives* and *DynamicalMatrix* objects for bulk configurations, which can be repeated to form the central region of the device.

From the dynamical matrix D we can obtain the phonon frequencies, ω_{λ}, and eigenvectors, $\mathbf{u}^{\lambda}$,

$$D\mathbf{u}^{\lambda} = \omega_{\lambda}^{2}\mathbf{u}^{\lambda}.$$

The electron-phonon coupling matrix for a given phonon mode, λ, is obtained as:

$$M_{ij}^{\lambda} = \sum_{I\nu}\langle i|\frac{\partial\hat{H}}{\partial R_{I\nu}}|j\rangle\mathbf{v}_{I\nu},$$

where

$$\mathbf{v}_{I\nu} = \mathbf{u}_{I\nu}^{\lambda}\sqrt{\frac{\hbar}{2M_{I}\omega_{\lambda}}}$$

and where the sum runs over atom indices I and Cartesian directions $\nu = (x, y, z)$. The derivative of the Hamiltonian is what is being calculated in the *HamiltonianDerivatives* object.

The theory behind the *InelasticTransmissionSpectrum* can be found in Refs.[1],[2]. In short, the starting point is the

[1] J.-T. Lu, R.ăB. Christensen, G.ăFoti, T.ăFrederiksen, T.ăGunst, and M.ăBrandbyge. Efficient calculation of inelastic vibration signals in electron transport: Beyond the wide-band approximation. *Phys. Rev. B*, 89:081405, Feb 2014. doi:10.1103/PhysRevB.89.081405.

[2] T.ăFrederiksen, M.ăPaulsson, M.ăBrandbyge, and A.-P. Jauho. Inelastic transport theory from first principles: Methodology and application to nanoscale devices. *Phys. Rev. B*, 75:205413, May 2007. doi:10.1103/PhysRevB.75.205413.

Meir-Wingreen current formula:

$$I(V) = \frac{2e}{h}\int_{-\infty}^{\infty} d\epsilon \mathrm{Tr}\left[\Sigma_L^{<}(\epsilon)G^{<}(\epsilon) - \Sigma_L^{<}G^{<}(\epsilon)\right],$$

where $G^{</>}$ are the lesser/greater Greens functions for the central region (including electron-phonon self-energies) and $\Sigma_L^{</>}$ are the lesser/greater self-energies due to coupling to the left electrode. The electron-phonon coupling self-energies are described in the self-consistent Born approximation (SCBA) assuming free (uncoupled to electrons and uncoupled to electrodes) phonon Greens functions. The phonon spectral function is thus a series of delta-peaks at $\pm\hbar\omega_\lambda$. We further assume that the phonon occupations are given by the Bose-Einstein distribution. Currently, it is thus not possible to include heating effects.

The SCBA electron-phonon self-energies are computationally demanding as they require an integral over energy, e.g.[Page 3127, 2].

$$\Sigma_{ph,\lambda}^{<,>}(\epsilon) = i\int_{-\infty}^{\infty}\frac{d\epsilon'}{2\pi}M^\lambda D_0^{<,>}(\lambda,\epsilon-\epsilon')G^{<,>}(\epsilon')M^\lambda$$

The idea behind the Lowest Order Expanssion (LOE) and eXtended LOE (XLOE) is to circumvent these integrals by assuming that the spectral function (or density of states) in the central region varies slowly on an energy scale given by the phonon energies, $\hbar\omega$. By expanding the Meir-Wingreen current formula to the second order in M (this is the lowest order) one can derive the LOE current expression:

$$I^{\mathrm{LOE}}(V,T) = I_0(V) + \sum_\lambda I_\lambda^{sym}(V,T,\langle n_\lambda\rangle)\mathcal{T}_\lambda^{sym} + \sum_\lambda I_\lambda^{asym}(V,T)\mathcal{T}_\lambda^{asym}$$

where $I_0(V)$ is the non-interacting or elastic current that can be obtained from the normal transmission function. I_λ^{sym} and I_λ^{asym} are universal current functions given by

$$I_\lambda^{sym}(V,T,\langle n_\lambda\rangle) = \frac{e}{\pi\hbar}\left(2eV\langle n_\lambda\rangle + \frac{\hbar\omega_\lambda - eV}{e^{(\hbar\omega_\lambda - eV)k_BT}-1} - \frac{\hbar\omega_\lambda + eV}{e^{(\hbar\omega_\lambda + eV)k_BT}-1}\right)$$

and

$$I_\lambda^{asym} = \frac{e}{\hbar}\int_{-\infty}^{\infty}\frac{d\epsilon}{2\pi}\left[n_F(\epsilon) - n_F(\epsilon - eV)\right]\times\mathcal{H}_{\epsilon'}\{n_F(\epsilon'+\hbar\omega_\lambda) - n_F(\epsilon'-\hbar\omega_\lambda)\}(\epsilon),$$

where $\mathcal{H}_{\epsilon'}$ is the Hilbert transform and the bias is defined via $eV = \mu_R - \mu_L$. The inelastic transmission functions $\mathcal{T}_\lambda^{sym}$ and $\mathcal{T}_\lambda^{asym}$ are given by

$$\mathcal{T}_\lambda^{sym} = \mathrm{Tr}\left[G^r\Gamma_L G^a\{M^\lambda A_R M^\lambda + \frac{i}{2}(\Gamma_R G^a M^\lambda A M^\lambda - H.c.)\}\right],$$

$$\mathcal{T}_\lambda^{asym} = \mathrm{Tr}\left[G^a\Gamma_L G^r\{\Gamma_R G^a M^\lambda (A_R - A_L)M^\lambda + H.c\}\right],$$

where $H.c.$ denotes hermetian conjugate and where the spectral functions are

$$A_{L,R} = G^r\Gamma_{L,R}G^a,$$

and

$$A = i(G^r - G^a) = A_L + A_R.$$

In the above equations, all the Greens function are non-interacting, i.e. they do not include the electron-phonon self-energies. When using LOE, all Greens function, self-enegies, and spectral functions are evaluated at the same energy. In XLOE, certain Greens functions and spectral functions will be evaluated at energies $\epsilon_\pm = \epsilon \pm \hbar\omega_\lambda/2$. This means that many more calculations are needed for the XLOE calculations which therefore takes more time.

LOE or XLOE

The LOE is originally developed for studying molecular junctions where a molecule is placed between metallic electrodes. If there are no molecular states close to the Fermi energy, the LOE will be sufficient to use. However, if there are states close to the Fermi energy, the XLOE will be a more correct description as it allows some variation of the density of states. XLOE should also be used if transport occurs close to semi-conductor band edges.

In XLOE one distinguishes between phonon emission and phonon absorption processes, as illustrated in Fig. 4.24. The inelastic transmission functions for the emission process, corresponding to a positive bias can be obtained as:

```
t_sym_pos = inelastic_transmission_spectrum.inelasticTransmissionSymmetricPositiveBias()
t_asym_pos = inelastic_transmission_spectrum.
↪inelasticTransmissionAsymmetricPositiveBias()
```

and for phonon absorption, corresponding to a negative bias situation as:

```
t_sym_neg = inelastic_transmission_spectrum.inelasticTransmissionSymmetricNegativeBias()
t_asym_neg = inelastic_transmission_spectrum.
↪inelasticTransmissionAsymmetricNegativeBias()
```

How many energy points

The *InelasticTransmissionSpectrum* can be called with a user-defined list of energies (default is $E = 0\,eV$, relative to the Fermi energy). If the non-interacting transmission is finite at the Fermi level (e.g. in a molecular junction with metallic leads) it will often be sufficint to calculate the inelastic transmisssion spectrum only at the Fermi energy and use the LOE current formula, which is called by:

```
bias = numpy.linspace(-0.1,0.1,100)*Volt
current = inelastic_transmission_spectrum.inelasticCurrent(bias=bias)
```

The conductance dI/dV and inelastic electron tunneling spectra (IETS) d^2I/dV^2 can likewise be obtained from the LOE current expression as:

```
bias = numpy.linspace(-0.1,0.1,100)*Volt
conductance = inelastic_transmission_spectrum.inelasticConductance(bias=bias)
iets = inelastic_transmission_spectrum.inelasticElectronTunnelingSpectroscopy(bias=bias)
```

In situations where there are no electrode states at the Fermi energy, like in a low- or un-doped semi-conductor, the direct use of the LOE and XLOE expressions will result in zero inelastic current, when evaluated at the Fermi energy. In that case, the inelastic transmission spectra must be calculated at a range of energies (like the normal *TransmissionSpectrum*). The current can then be calculated as:

```
bias = numpy.linspace(-0.1,0.1,100)*Volt
current = inelastic_transmission_spectrum.inelasticCurrentIntegral(bias=bias)
```

In this case the current is calculated in a way, following Ref.[3]:

$$I(V) = \sum_{\lambda} \int_{-\infty}^{\infty} d\epsilon \left[\mathcal{T}_{\lambda}^{sym,pos}(\epsilon) F_{pos}^{\lambda}(\epsilon) + \mathcal{T}_{\lambda}^{sym,neg}(\epsilon) F_{neg}^{\lambda}(\epsilon) \right],$$

where the energy- and mode depended prefactors are

$$F_{pos}^{\lambda} = n_F(\mu_L)[1 - n_F(\mu_R - \hbar\omega_\lambda)][n_B(\hbar\omega_\lambda) + 1] - n_F(\mu_R + \hbar\omega_\lambda)[1 - n_F(\mu_L)]n_B(\hbar\omega_\lambda),$$

[3] W.ăVandenberghe, B.ăSoree, W.ăMagnus, and M.ăV. Fischetti. Generalized phonon-assisted Zener tunneling in indirect semiconductors with non-uniform electric fields: A rigorous approach. *J. Appl. Phys.*, 2011. doi:10.1063/1.3595672.

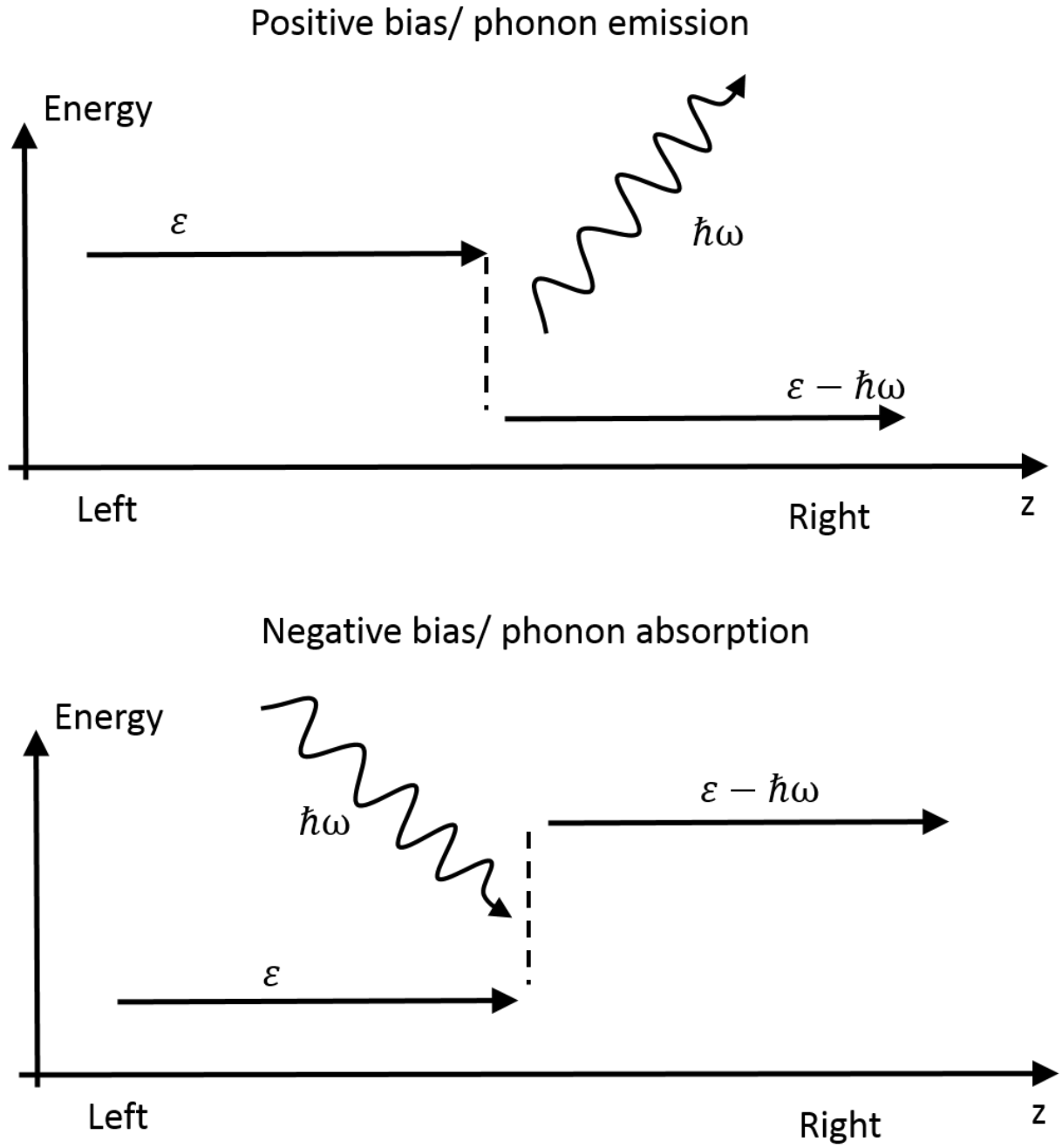


Fig. 4.24: In XLOE phonon emission and absorption processes are distinguished. The upper panel illustrates a phonon emission process, where an electron incident from the left electrode emits a phonon and exit the right electrode with a lower energy. At low temperatures, this process is only possible at positive bias voltages, hence the name. The lower panel illustrates a phonon absorption process where the electron exits with a higher energy in the right electrode than in the left.

and

$$F^{\lambda}_{neg} = n_F(\mu_L)[1 - n_F(\mu_R + \hbar\omega_\lambda)]n_B(\hbar\omega_\lambda) - n_F(\mu_R - \hbar\omega_\lambda)[1 - n_F(\mu_L)][n_B(\hbar\omega_\lambda) + 1].$$

n_F and n_B are the Fermi-Dirac and Bose-Einstein distribution functions respectively.

InfraredSpectrum

class InfraredSpectrum(*md_trajectory*, *start_time=None*, *end_time=None*, *time_resolution=None*, *wavenumber_limit=None*, *temperature=None*, *path_length=None*, *smoothing_factor=None*, *info_panel=None*)

Class for calculating the infrared vibrational spectrum from an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory*) – The MDtrajectory from which the infrared is calculated. If present the dipole measurement is used, otherwise information is taken from each image.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots or measurements in the MD trajectory that are included in the analysis.
- **wavenumber_limit** (`PhysicalQuantity` of type inverse length) – The maximum wavenumber displayed in the spectrum. **Default:** `5000 * cm^-1`.
- **temperature** (`PhysicalQuantity` of type temperature) – The temperature used in calculating absorption intensity. **Default:** `300 * Kelvin`.
- **path_length** (`PhysicalQuantity` of type length) – The length of sample the light passes through. Longer path lengths decreases the amount of light transmitted **Default:** `0.001 * cm`.
- **smoothing_factor** (*int*) – Apply smoothing to the spectrum by averaging n points. **Default:** 1.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.

absorption()

Returns
The infrared absorption for each wavenumber in the spectrum.

Return type
ndarray

absorptionCoefficient()

Returns
The absorption coefficient for each wavenumber in the infrared spectrum.

Return type
`PhysicalQuantity` of type inverse length

frequencies()

Returns
The frequency of each datapoint in the infrared spectrum.

Return type
`PhysicalQuantity` of type inverse time

transmission()

Returns
The infrared transmission for each wavenumber in the spectrum.

Return type
ndarray

wavenumbers()

Returns
The wavenumber of each datapoint in the infrared spectrum.

Return type
PhysicalQuantity of type inverse length

Usage Examples

Load an *MDTrajectory* containing a simulation of a liquid electrolyte, and calculate its infrared (IR) vibrational spectrum. The IR transmission through a sample with a path length of 100 micron is then plotted.

```
md_trajectory = nlread('electrolyte_trajectory.hdf5')[-1]

# Create the IR Spectrum
ir_spectrum = InfraredSpectrum(
    md_trajectory,
    wavenumber_limit=4000*cm**-1,
    temperature=300*Kelvin,
    path_length=0.01*cm,
    smoothing_factor=1,
)

# Create a model.
model = Plot.PlotModel(x_unit=cm**-1)
model.framing().setTitle('Electrolyte IR Spectra')
model.xAxis().setLabel('Wavenumber')
model.yAxis().setLabel('Transmission')
model.legend().setVisible(False)

# Create and add line.
line = Plot.Line(ir_spectrum.wavenumbers(), ir_spectrum.transmission())
line.setLabel('Spectrum')
line.setColor('blue')
model.addItem(line)

# Set x-axis limits.
model.setLimits('x', 4000*cm**-1, 500*cm**-1)

# Show the plot for interactive editing.
Plot.show(model)
```

infrared_spectrum.py

Notes

From a molecular dynamics trajectory, the infrared vibrational spectrum can be calculated by obtaining the Fourier transform of the system dipole autocorrelation function[1][2]. The adsorption coefficient $\alpha(\omega)$ is given as:

$$\alpha(\omega) = \frac{\omega c \tanh\left(0.5\beta\hbar\omega\right)}{3\epsilon_0 \hbar n(\omega) V} \int_0^\infty \langle \mathbf{M}(0) \cdot \mathbf{M}(t) \rangle \cos(\omega t) dt$$

Here ω is the vibrational frequency, β is the Boltzmann factor $1/kT$, $n(\omega)$ is the frequency dependent refractive index and V is the configuration volume. Note that in calculating the infrared spectrum the refractive index is assumed to be constant in the infrared frequency range. The total system dipole $\mathbf{M}$ is also given as:

$$\mathbf{M} = \sum_{i=0}^{N} q_i \mathbf{r}_i$$

Here q_i and $\mathbf{r}_i$ are the atomic charge and position respectively, with the sum being taken over all atoms in the system. Finally the transmission spectrum $T(\omega)$ and absorption spectrum $A(\omega)$ can be estimated with the relationship

$$T(\omega) = 1 - A(\omega) = \exp(-\alpha(\omega) l)$$

where l is a path length that represents the thickness of the sample.

In order to estimate the change in dipole moment during the simulation required to calculate the adsorption spectrum, partial charges must be set on the input configuration. Partial atomic charges can be added to a configuration using the `setPartialCharges` method. These charges are used to calculate the dipole in each image. Obtaining reliable vibrational spectra also requires saving the dipole moment at high frequency. As only the total system dipole is required, this is most efficiently done by saving the dipole as a measurement using the `dipole_vector` molecular dynamics measurement. Where appropriate dipole measurement data is contained within the input trajectory, this will be used in preference to calculating the dipole from each image. This allows using high-frequency dipole data with a minimal memory and computational overhead.

It should also be noted that since the analysis is based on the assigned partial charges, the relative size of different peaks may be dependent on the charge assignment. Of particular importance is the magnitude of the dipole moment change during each vibrational mode. Vibrations that do not result in a change in dipole moment do not result in peaks in the infrared spectrum.

The infrared spectrum analysis contains some optional arguments. The argument `wavenumber_limit` sets the largest wavenumber considered in the vibrational spectrum analysis. Similarly the argument `temperature` sets the temperature used in calculating the absorption coefficient, and is commonly set to the approximate temperature of the molecular dynamics simulation. The amount of infrared radiation absorbed or transmitted is controlled with the `path_length` variable. This can be used to enhance or reduce the size of peaks in the absorption spectrum to highlight important features. Finally the argument `smoothing_factor` controls the resolution of the final spectral data. Increasing the smoothing factor down-samples the resulting spectrum, averaging each value over the given number of points. This can help to reduce noise in the calculated spectrum.

[1] Matej Praprotnik and Duanka Janeiè. Molecular dynamics integration and molecular vibrational theory. III. The infrared spectrum of water. *J. Chem. Phys.*, 122:174103, 2005. URL: https://aip.scitation.org/doi/10.1063/1.1884609, doi:10.1063/1.1884609.

[2] Bertrand Guillot. A molecular dynamics study of the far infrared spectrum of liquid water. *J. Chem. Phys.*, 95:1543, 1991. URL: https://aip.scitation.org/doi/10.1063/1.461069, doi:10.1063/1.461069.

InitialSpin

class InitialSpin(*scaled_spins=None*)

Class for representing an initial spin configuration.

The initial scaled spin for each atom should be given as a number between -1 and 1 corresponding to the net spin-polarization in fractions of the atomic polarization according to Hunds rule.

This should be either:

- A sequence of floating point numbers, whose position in the list corresponds to the atom index in the configuration
- A list of tuples each of an atom index and corresponding scaled spin value.
- A list of tuples each of a *PeriodicTableElement* object and a scaled spin value.
- A combination of the latter two.

For non-collinear spin systems the tuple has four numbers, atom index, scaled spin, theta, phi, where the latter two are spherical coordinates as *PhysicalQuantity* object of type Degree or Radians.

Parameters
scaled_spins (*list*) – The initial scaled spins for each atom. **Default:** `1.0 for each atom.`.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

There are two basic ways to specify the *InitialSpin* of the atoms.

1. Using a simple list. In this case, `scaled_spins` is a list containing the initial scaled spin for each atom, i.e. the number of entries in the list must be equal to the number of atoms in the system (in the central region for *DeviceConfiguration*). Example: Set initial spin 1, -0.5, 1 on the first three atoms, 0 for the rest (the configuration contains 50 atoms).

```
bulk_configuration.setCalculator(
    calculator=LCAOCalculator(),
    initial_spin=InitialSpin(scaled_spins=[1.,-0.5,1.]+[0.,]*47)
    )
```

2. Identify atoms, or groups of atoms, and set their initial spin individually. In this case, `scaled_spins` is a list of tuples, each with two elements. The first entry in the tuple can be an integer, to select an individual atom, or an element (like Carbon), to select all atoms of a certain element. The second entry is the initial scaled spin to set for the selected atoms.

   Example: The previous example may be specified the following way:

```
bulk_configuration.setCalculator(
    calculator=LCAOCalculator(),
    initial_spin=InitialSpin(scaled_spins=[(0,1.),(1,-0.5),(2,1.)])
    )
```

   Set the spin 1 on Iron atoms, except for atoms with index 3 and 7.

```
bulk_configuration.setCalculator(
     calculator = LCAOCalculator(),
     initial_spin = InitialSpin(scaled_spins=[(Iron,1.),(3,-1), (7,-1)])
     )
```

•Later entries in the list override earlier ones.

•The initial scaled spin is set to 1.0 for atoms for which another value is not specified.

•The sign (positive or negative) of the specified initial spin determines whether the system has a net surplus of spin-up or spin-down electrons.

For noncollinear spin you may specify the spin direction in physical spherical coordinates (r, θ, ϕ), where θ is the angle with the z axis, and ϕ the polar angle in the x-y plane relative to the x-axis. The collinear case is $\theta = 0$ rad or $\theta = \pi$ rad.

```
bulk_configuration.setCalculator(
    calculator = LCAOCalculator(),
    initial_spin = InitialSpin(scaled_spins=[
        (Iron, 1.0, 0.5*numpy.pi*Radians, 0.4*numpy.pi*Radians),
        (3,    0.8, 270*Degrees         , 216*Degrees)])
    )
```

Also see *RandomSpin* for an alternative way to set the initial spin.

Notes

The simplest way to obtain a spin-polarized calculation is to use *InitialSpin* , i.e. without any parameters, which means that the initial scaled spin is set to 1.0 for all atoms. This is usually the safest way to obtain good convergence in spin-polarized in a *MoleculeConfiguration* and a *BulkConfiguration*. In a spin-polarized *DeviceConfiguration*, however, a more careful analysis is often needed in order to obtain reliable convergence. This will be discussed below.

The initial scaled spin for each atom specifies the degree of polarization for unpaired electrons (either spin-up or spin-down) within a given valence electron shell. To understand this properly, it is best to consider a specific example.

The initial spins for the different atoms in a configuration are independent of each other, so without loss of generality we can limit the discussion to one atom. Let us introduce the symbol α to denote the initial scaled spin for that atom.

For definiteness, let us look at bulk Fe, which has a total of 8 valence electrons ($4s^2 3d^6$).

How should these electrons be distributed with respect to spin in a spin-polarized calculation? In the simplest picture, we start by filling all the available spin-up states, and when they have been filled we add the remaining electrons into spin-down states. The resulting occupations are shown in the following *table* (note that the 4s orbitals, both up and down, are filled before the 3d orbitals).

Table 4.20: Iron in the maximum spin configuration, i.e. $\alpha = 1.0$.

	4s	3d
Spin up	↑	↑↑↑↑↑
Spin down	↓	↓

This configuration corresponds to an initial scaled spin of $\alpha = 1.0$ (or -1, if we reverse the roles of spin-up and spin-down). For obvious reasons, we will refer to this configuration as the **maximum spin configuration**.

Let us define some symbols for reference. Let $n_{i,\sigma}$ be the total population of shell i (4s and 3d, for iron) in the maximum spin configuration, where σ denotes the spin. We also define the maximum spin polarization for each shell,

$$\Delta N_i = n_{i,\uparrow} - n_{i,\downarrow},$$

which thus is 4 for 3d in the case of iron. For 4s the maximum spin polarization is zero, which means that this shell cannot be initially polarized in the calculations. A spin polarization of the 4s shell can still occur as a result of the

self-consistent calculation, but it requires that the total population of the 4s shell is lower than two, and this cannot be used as an initial condition, since the total population of each shell,

$$N_i = \sum_\sigma n_{i,\sigma},$$

is fixed and constant in the process of assigning the initial populations.

Now, the initial scaled spin α determines the initial spin occupations $o_{i,\sigma}$ of the states in shell i through the following expression:

$$\alpha_i = \frac{o_{i,\uparrow} - o_{i,\downarrow}}{\Delta N_i}.$$

As mentioned, the total number of valence electrons in each shell N_i must always be conserved, and this poses a constraint on the initial spin configurations that can be constructed in this way. It can easily be verified, that $\alpha_i = 1$ gives exactly $o_{i,\sigma} = n_{i,\sigma}$.

Let us see what happens with iron if we set $\alpha = 0.5$. As mentioned above, only the 3d shell can be polarized initially, and we find

$$o_{3\mathrm{d},\uparrow} - o_{3\mathrm{d},\downarrow} = \alpha\ \Delta N_{3\mathrm{d}} = 0.5 \times 4 = 2.$$

Combined with the constraint $N_{3\mathrm{d}} = o_{3\mathrm{d},\uparrow} + o_{3\mathrm{d},\downarrow} = 6$, we find

$$o_{3\mathrm{d},\uparrow} = 4, \qquad o_{3\mathrm{d},\downarrow} = 2.$$

These populations are equally distributed among the five 3d states for each respective spin.

As a further example, had we instead specified $\alpha = 0.75$ we would have obtained a spin-polarized configuration with 4.5 spin-up and 1.5 spin-down electrons on average per unit-cell in the 3d shell.

Now, the parameter `initial_spin` only affects the initial spin polarization. The converged spin-polarization could very well be different. Let us again study iron as an example.

The script below calculates and prints the converged Mulliken population of a spin-polarized bulk bcc iron system. Since we do not know what value to use for the initial scaled spin, we use 1.0.

```
bulk_configuration = BulkConfiguration(
    bravais_lattice=BodyCenteredCubic(2.8665*Angstrom),
    elements=[Iron],
    cartesian_coordinates=[[ 0.,  0.,  0.]]*Angstrom
    )

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(10, 10, 10),
    )

calculator = LCAOCalculator(
    exchange_correlation=LSDA.PZ,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

initial_spin = InitialSpin(scaled_spins=[1.0])

bulk_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
```

(continues on next page)

(continued from previous page)

```
)

mulliken_population = MullikenPopulation(bulk_configuration)
nlprint(mulliken_population)
```

iron_spin.py

If you run the script, it will report the following occupations:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| Mulliken Population Report                                                    |
|                                                                               |
| ----------------------------------------------------------------------------- |
|                          |                                                    |
| Element   Total  Shell   | Orbitals                                           |
|                          |                                                    |
|                          |     xy     zy  zz-rr     zx  xx-yy                 |
|   0  Fe   5.108  0.704   |  0.140  0.140  0.142  0.140  0.142                 |
|           2.892  0.497   |  0.111  0.111  0.081  0.111  0.081                 |
|                          |      s                                             |
|                 -0.025   | -0.025                                             |
|                 -0.032   | -0.032                                             |
|                          |     xy     zy  zz-rr     zx  xx-yy                 |
|                  3.641   |  0.694  0.694  0.779  0.694  0.779                 |
|                  1.530   |  0.370  0.370  0.210  0.370  0.210                 |
|                          |      s                                             |
|                  0.317   |  0.317                                             |
|                  0.335   |  0.335                                             |
|                          |      y      z      x                               |
|                  0.471   |  0.157  0.157  0.157                               |
|                  0.562   |  0.187  0.187  0.187                               |
+-------------------------------------------------------------------------------+
```

The number show the *MullikenPopulation* of each orbital, for each spin (for each shell, the spin-up populations are listed on the first line, spin-down on the second one).

If you kept an eye on the calculation, you would have noticed that it converged in 18 iterations. The question is now, could we have improved the convergence rate by choosing a different initial spin polarization?

The converged total *MullikenPopulation* (on average per unit cell) for the spin-up and spin-down states are 5.11 and 2.89, respectively. Now these numbers also include the 4s and some 5p electrons (since a DoubleZetaPolarized *BasisSet* was used), but the detailed output shows that the spin polarization of these shells is small. Hence, as a reasonable approximation we can use $o_{3d,\uparrow} - o_{3d,\downarrow} = 5.11 - 2.89 = 2.22$.

Thus, the converged result corresponds to a value of $\alpha = 2.22/\Delta N_{3d} = 0.56$. It can therefore be expected that if we had used this value in the calculation from the beginning, it would have converged faster. This is indeed the case; rerunning the script with the new value of `scaled_spins=[0.56]`, the self-consistent iteration reaches convergence in 10 steps, i.e. we have almost gained a factor 2 in speed!

Note: Note how the resulting populations in the two calculations are identical; we are only setting the *initial* spin; the self-consistent calculation will find the proper solution anyway, provided the initial value is not too far off. In bulk systems this is, as mentioned above, not very critical, but for transport calculations it can sometimes be very hard to reach convergence unless a good initial value is provided. And, under any circumstance, with a good initial we can

save a lot of calculation time.

Finally, let us show how to determine the value that should be assigned to α in order to generate a given, initial, spin-configuration.

Suppose we wanted to perform a spin-polarized calculation for manganese (7 valence electrons, ($4s^2$ $3d^5$)) with an initial spin occupation as the one shown in the following *table*.

Table 4.21: Manganese in an electronic configuration with a majority of spin-down electrons.

	4s	3d
Spin up	↑	↑↑
Spin down	↓	↓↓↓

We can see that $o_{3d,\uparrow} - o_{3d,\downarrow} = -1$, as we have one unpaired spin-down electron. The number of unpaired electrons in the maximum spin configuration for manganese is 5 (all 3d electrons in the spin-up state), and thus we need to set $\alpha = -0.2$ to obtain the desired initial configuration.

InterfaceBuilder

class InterfaceBuilder(*configuration_0*, *configuration_1*, *shortest_surface_lattice_vector=None*, *longest_surface_lattice_vector=None*, *minimum_surface_lattice_vector_angle=None*, *max_surface_area=None*, *grid_density=None*, *strain_max=None*, *angle_min=None*, *angle_max=None*, *angle_delta=None*, *layer_tolerance0=None*, *layer_tolerance1=None*)

This class generates interfaces from two surface configurations. These configurations must be properly cleaved surfaces as generated by the `cleave` method on *BulkConfiguration*. This means that the lattice vectors are in a standardized form: a vector along the x-axis, a-b vector in the x-y plane, and the c vector along the z-axis.

Parameters

- **configuration_0** (list of type *BulkConfiguration*) – A configuration (or list of configurations) for the left half of the interface.
- **configuration_1** (list of type *BulkConfiguration*) – A configuration (or list of configurations) for the right half of the interface.
- **shortest_surface_lattice_vector** (*PhysicalQuantity* of type length) – The minimum length of any surface vector. **Default:** 0.0 * Angstrom.
- **longest_surface_lattice_vector** (*PhysicalQuantity* of type length) – The maximum length of either surface vector. **Default:** 1e10 * Angstrom.
- **minimum_surface_lattice_vector_angle** (*PhysicalQuantity* of type angle) – The smallest allowed angle between the surface lattice vectors. **Default:** 0.0 * Degrees.
- **max_surface_area** (*PhysicalQuantity* of type length**2) – The maximum surface area.
- **grid_density** (*2-tuple of type int*) – The number of times that the a and b lattice vectors of the first configuration should be repeated during the lattice match search. Larger values consider more possible repetitions.
- **strain_max** (*float*) – The largest engineering strain that will be considered a valid match. The default value of 1 corresponds to a 100% deformation. **Default:** 1.0.
- **angle_min** (*PhysicalQuantity* of type angle) – The minimum rotational angle of the second lattice with repect to the first. **Default:** 0.0 * Degrees.

- **angle_max** (*PhysicalQuantity* of type angle) – The maximum rotational angle of the second lattice with repect to the first. **Default:** 180.0 * Degrees.
- **angle_delta** (*PhysicalQuantity* of type angle) – The resolution of the rotational angles of the second lattice. Smaller values lead to a higher resolution search of rotational values. **Default:** 4.0 * Degrees.
- **layer_tolerance0** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.
- **layer_tolerance1** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.

bestMatch(*strain_method=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.StrainSecond'>*)

Parameters
strain_method (*StrainFirst | StrainSecond | StrainBoth*) – Specifies which layer should be strained. If StrainFirst is given, then the first layer will be strained and the second layer will have no strain, while the opposite is true if StrainSecond is chosen. If StrainBoth is given, then each layer is strained to an intermediate lattice.

Returns
The match with the highest score. High scoring structures balance system size and strain. If no matches were found *None* is returned.

Return type
InterfaceMatch | None

matches(*strain_method=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.StrainSecond'>*)

Parameters
strain_method (*StrainFirst | StrainSecond | StrainBoth*) – Specifies which layer should be strained. If StrainFirst is given, then the first layer will be strained and the second layer will have no strain, while the opposite is true if StrainSecond is chosen. If StrainBoth is given, then each layer is strained to an intermediate lattice.

Returns
A list of all lattice matches sorted by mean absolute strain.

Return type
list of type *InterfaceMatch*

nlprint(*stream=None*)

Print out table of matches.

Parameters
stream (*python stream*) – The stream the data should be printed to. **Default:** `NLPrintLogger()`

Usage Examples

Create an interface using Si (100) and Au (111) surfaces.

```
# Cleave each bulk configuration. This is required before using the interface builder.
si_100 = bulk_si.cleave(1, 0, 0)
au_111 = bulk_au.cleave(1, 1, 1)

# Setup interface builder. The configurations should be cleaved first.
interface_builder = InterfaceBuilder(si_100, au_111)

# Get all of the interface matches, specifying that the second surface (gold) should be␣
```

(continues on next page)

(continued from previous page)

```
↪strained.
# The matches are sorted from lowest to highest mean absolute strain.
matches = interface_builder.matches(strain_method=StrainSecond)

# Loop over all matches and print out information and save each to a hdf5 file.
print('{:>20s} {:>10s}'.format('number of atoms', 'strain (%)'))
for match in matches:
    # Get the number of atoms and strain from the match.
    number_of_atoms = match.numberOfAtoms()
    mean_absolute_strain = 100.0 * match.meanAbsoluteStrain()

    # Print out information about the match.
    print('{:20d} {:10.3f}'.format(number_of_atoms, mean_absolute_strain))

    # Make the interface configuration and save it to a file.
    interface_configuration = match.makeInterface()
    nlsave('matches.hdf5', interface_configuration)

# Scatter plot of number of atoms vs strain.
import pylab
pylab.scatter(
    [100 * match.meanAbsoluteStrain() for match in matches],
    [match.numberOfAtoms() for match in matches],
)
pylab.xlabel('Mean Absolute Strain (%)')
pylab.ylabel('Number of Atoms')
pylab.xscale('log')
pylab.savefig('size_strain_plot.png')
```

The script will loop over all possible interface matches and print the number of atoms in the structure and the mean absolute strain as well as generate the following plot:

The script can be downloaded: `si_au_interface.py`

Alternatively by using the `bestMatch` method an interface structure can be generated in one line of code like so:

```
interface_configuration = InterfaceBuilder(
   bulk_1.cleave(1, 0, 0),
   bulk_2.cleave(1, 1, 1)
).bestMatch().makeInterface()
```

Interface Builder

The **Interface Builder** of QuantumATK allows the construction of complex interfaces by automatically matching the 2D unit cells of two different surfaces and compare the strain. The input is two crystal surfaces which can be created from bulk systems using the **Surface (Cleave)** tool. A specified number of atomic layers can be added for each material, an atomic layer being defined as the atoms that lie with the same distance to the AB-plane of the unit cell.

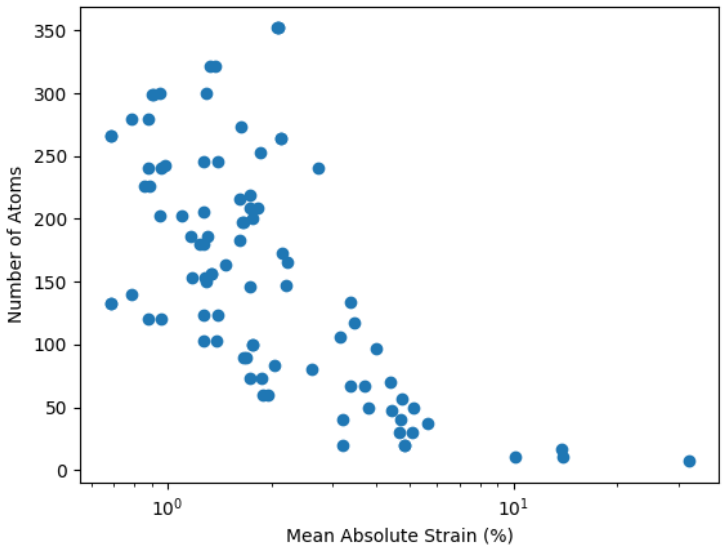


Fig. 4.25: The number of atoms as a function of mean absolute strain.

Select Surface Cells

The **Select Surface Cells** button is used to view the results of the lattice matching. This opens a window where the strain of all the found matches is plotted against the total number of atoms in the interface unit cell, see Fig. 4.27. The selected unit cell is the red dot, one can select alternative matches by clicking the blue dots.

The procedure for matching the two lattices is as follows;

•The unit cell vectors of surface *a* (a_1, a_2) and *b* (b_1, b_2) are extracted.

•The *b* cell is rotated by an angle θ, (c_1, c_2) in Fig. 4.28. This is done for a set of angles $\theta \in [\theta_{min}; \theta_{max}]$ with an default increment of $\delta_\theta = 4$ degrees. For each of the rotations;

–All possible lattice vectors, v, where (c_1, c_2) has been repeated a maximum of (n^{max}, m^{max}) times is found, v_1 or v_2 in Fig. 4.28.

$$v = i \cdot c_1 + j \cdot c_2, \qquad i \in [-n^{max}; n^{max}],\ j \in [0, m^{max}].$$

–We express v in the lattice of cell *a* by creating a linear transformation,

$$U = \begin{pmatrix} a_{1,x} & a_{2,x} \\ a_{1,y} & a_{2,y} \end{pmatrix}, \qquad v = Us,$$

where $s = U^{-1}v$ describes how v is expressed in the unit cell vectors a_1 and a_2.

–The vector, v, needs to be truncated to the grid of *a*. This is done by rounding the elements of s to integers, s', and calculating

$$u = Us'.$$

u (u_1 or u_2 in Fig. 4.28) corresponds to the lattice vector of cell *a* that v will be matched to.

–All possible cells, (v_1, v_2), of *b* is created by iterating through all the found lattice vectors v two by two.

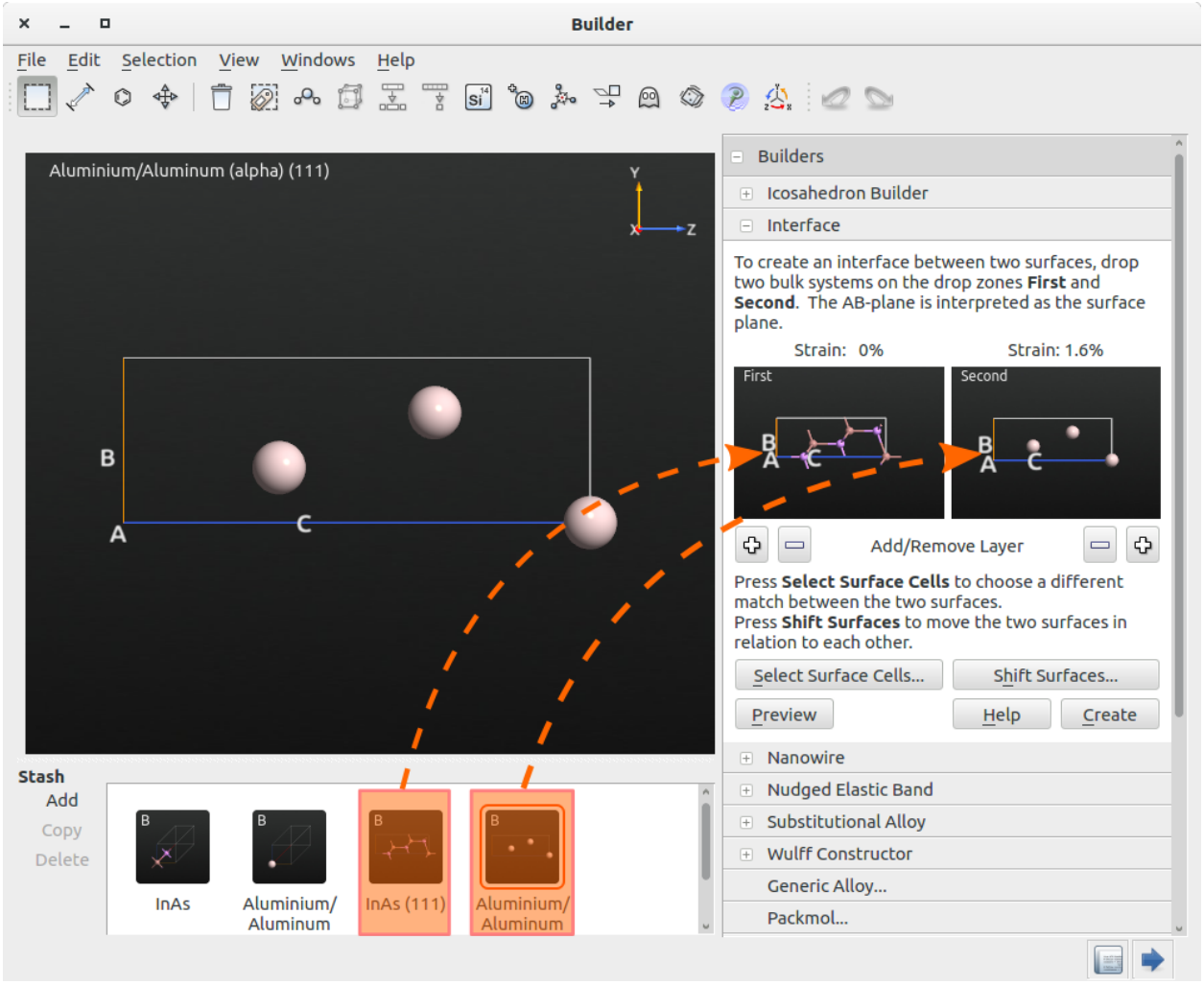


Fig. 4.26: The interface builder used on InAs(111) and Al(111).

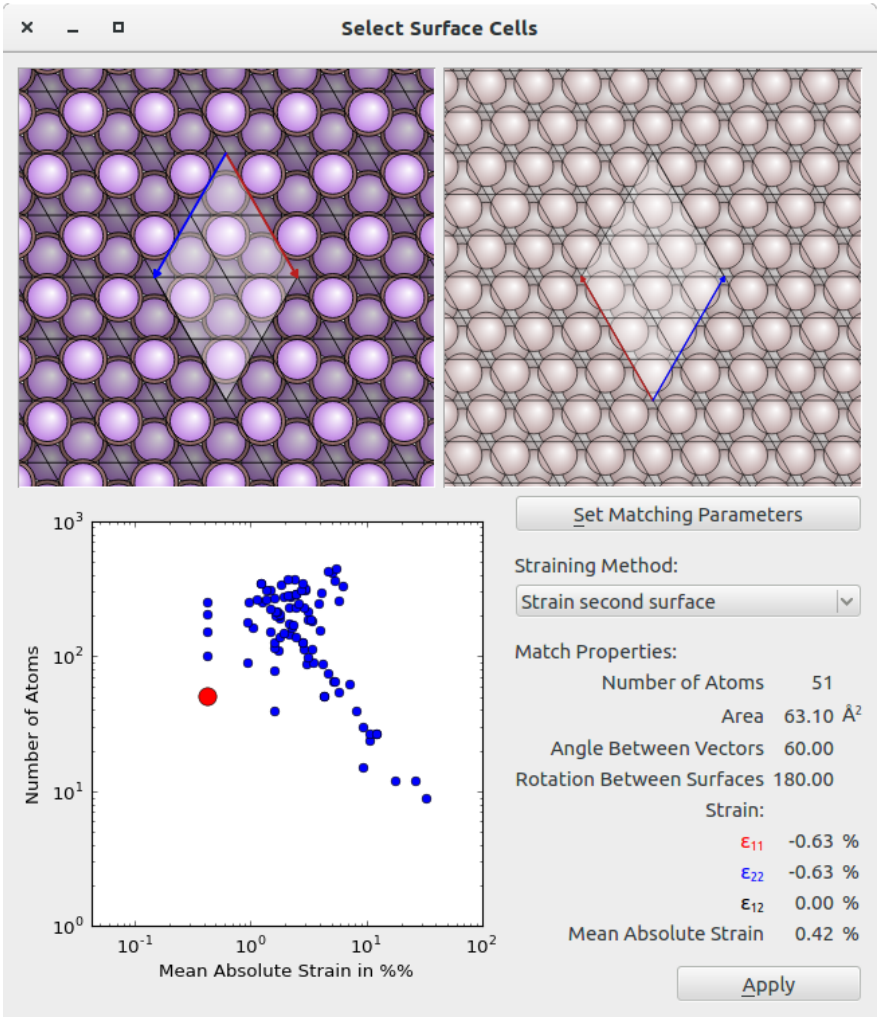


Fig. 4.27: The Select Surface Cells window.

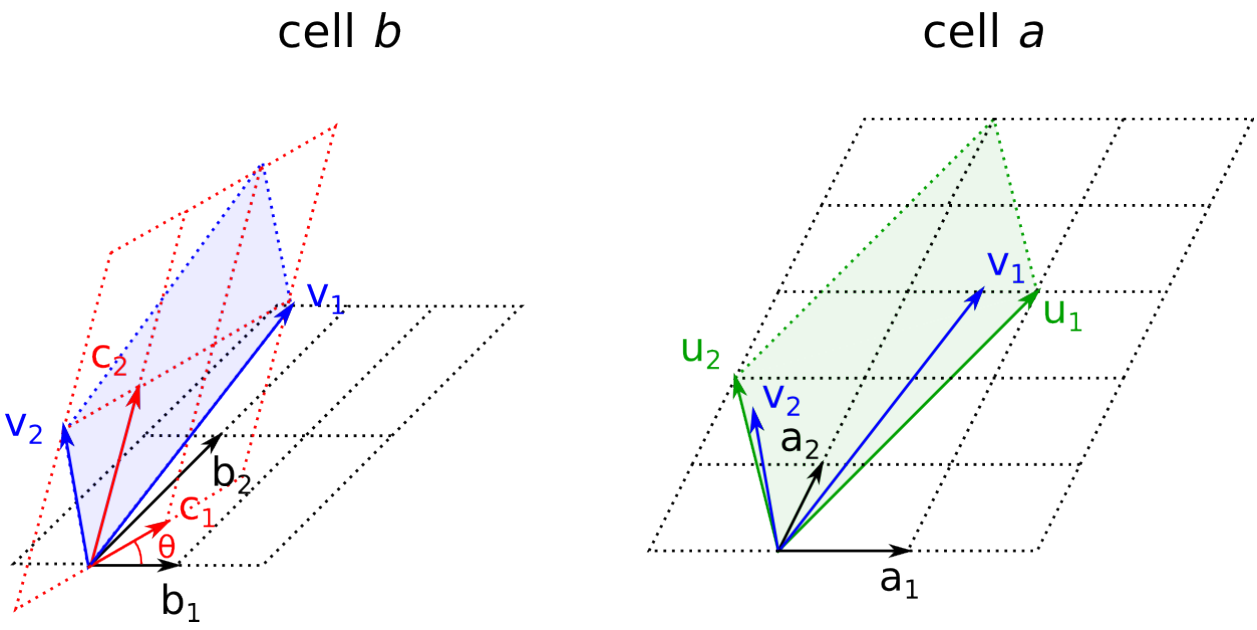


Fig. 4.28: The matching of two surface lattices. (a_1, a_2) and (b_1, b_2) are the unit cell vectors of the two surfaces, and (c_1, c_2) is the rotated cell of the b surface. v_1 and v_2 are a linear combination of c_1 and c_2. u_1 and u_2 are v_1 and v_2, respectively, truncated to the grid of cell a.

–The strain tensor from the (v_1, v_2) cell to the corresponding (u_1, u_2) cell is calculated. This can be done using the following three straining methods; only straining (v_1, v_2), only straining (u_1, u_2), or straining both equally. The method can by chosen in the **Select Surface Cells** window. For the first straining method, we get

$$\varepsilon_{11} = \left|\frac{v_{1,x}}{u_{1,x}}\right| - 1.$$
$$\varepsilon_{22} = \left|\frac{v_{2,y}}{u_{2,y}}\right| - 1.$$
$$\varepsilon_{12} = \frac{1}{2}\frac{v_{2,x} - \frac{v_{1,x}}{u_{1,x}}u_{2,x}}{u_{2,y}}.$$

Equivalent formulas are used for the second straining method. If both surfaces should be strained equally, an intermediate cell is created between the (v_1, v_2) and (u_1, u_2) cell. This cell in constructed s.t. the strain tensor from the first lattice to the intermediate one is exactly minus the stress tensor from the second to the intermediate one. From the stress tensor, we find the mean absolute stress ε^{av},

$$\varepsilon^{av} = \frac{\varepsilon_{11} + \varepsilon_{22} + \varepsilon_{12}}{3}.$$

Lattices with a mean absolute stress outside the specified limits $\varepsilon^{av} \in [\varepsilon_{min}; \varepsilon_{max}]$ are discarded and strains within a specified $tolerance$ are considered equal.

–The total number of atoms in the corresponding interface is found by considering the area of the cell and the lattice match is ranked according to the following score, S;

$$S_\phi = \sum_\alpha 1 - 7\mathrm{e}^{-\frac{(\phi-\alpha)^2}{15}},$$
$$\alpha \in [15, 30, 45, 60, 90, 120, 150]\,\text{degrees},$$
$$S = \mathrm{e}^{-\varepsilon^{av} - \frac{A}{10} - S_\phi},$$

where ϕ is the angle between the two vectors of the cell, and A is the area of the cell. This score favours the cells where the angle between the vectors are close to any of the angles in α, the mean absolute strain is low and the area is small. This is based on the assumption that a small unit cell will be energetically favourable and that a low strain will minimize the amount of defects. The score is a subjective number, created to have a default guess for the interface, and is therefore not meant as a general way of predicting the most physically sensible interface.

- The lattice match with the best score will be the default choice of the builder.

The matching parameters can be changed using the **Set Matching Parameters** button in the **Select Surface Cells** window, see Fig. 4.29. The default parameters are listed in the table below.

Table 4.22: Default matching parameters.

n^{max}	6
m^{max}	6
θ_{min}	0 degrees
θ_{max}	180 degrees
δ_θ	4 degrees
ε_{min}	-100 %
ε_{max}	100 %
$tolerance$	1^{-6}

Shift Surfaces

When a suitable lattice match has been found, the **Shift Surfaces** button can be used to shift one surface w.r.t. to the other. This can be done either manually or by minimizing the energy of the system. Using the second option, the geometry is optimized using a classical potential from the QuantumATK Classical package and with the constrains that each material acts as a rigid body. The optimization method is LBFGS and the optimization procedure ends when the inter-atomic forces are below 0.02 eV/Å. The calculated displacement vector is shown in the window together with the total energy of the system, if the **Calculate energy** checkbox is checked. The displacement is automatically applied to the interface. It should be noted, that his procedure only finds local minima of which there could be several. To find the global minima one should optimize the structure using several different starting points determined by the symmetry.

Notes

- The input structures must be surfaces. In QuantumATK this means that the lattice vectors are prepared according to the following convention: the a-vector is along the x-axis, the b vector is in the a-b plane, the c vector is along the z-axis. The easiest way to prepare such a structure is to use the `cleave` method of *`BulkConfiguration`*.
- The `matches` and `bestMatch` methods return a list of *`InterfaceMatch`* objects.
- The `bestMatch` method returns a match that balances the total number of atoms in the system with the absolute mean strain. It can be used to find a good match without having to filter through all possible matches.

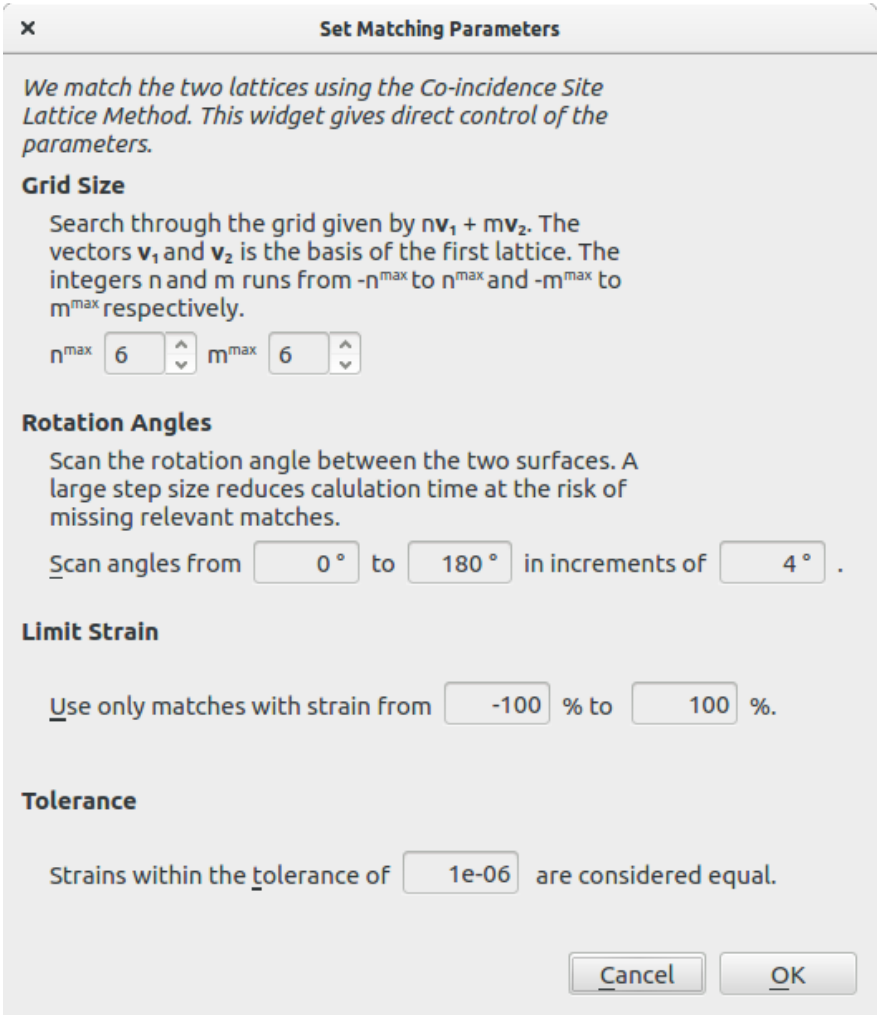


Fig. 4.29: The Set Matching Parameters window.

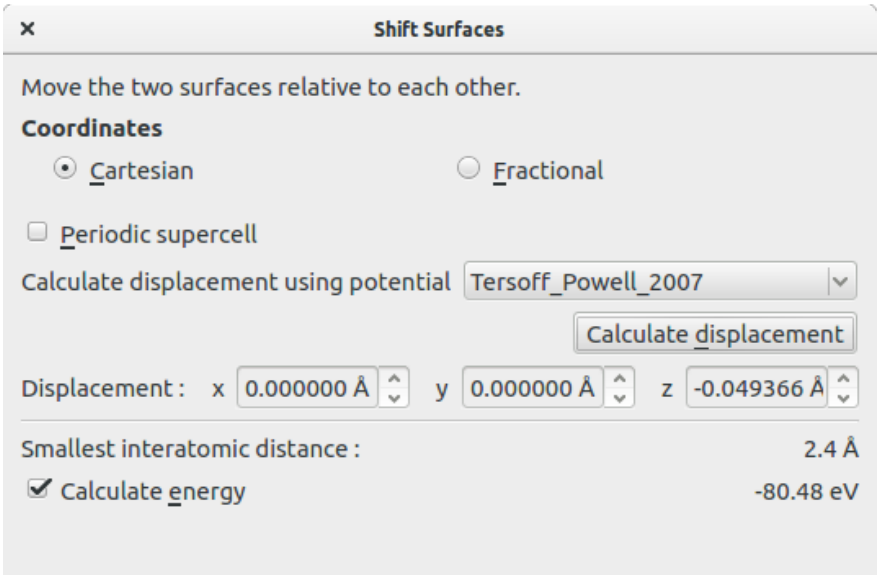


Fig. 4.30: The Shift Surfaces window.

InterfaceDefectAnalysis

class InterfaceDefectAnalysis(*defects*, *projected_dos=None*)

Create the interface defects analysis object.

Parameters

- **defects** (List or *Table* of *ChargedPointDefectConfiguration*) – A sequence of defects at an interface,
- **projected_dos** (*ProjectedDensityOfStates*) – A *ProjectedDensityOfStates* analysis object that can be used to plot the local density along the interface direction.

availableStableTransitions(*defect_filter=None*, *finite_size_correction=True*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Get the stable charge state transitions in the collection of defects.

Parameters

- **defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.
- **finite_size_correction** (*bool*) – Whether to include the finite size correction..
- **include_vibrations** (*bool*) – Include vibrations, if available, in the report output.
- **temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
The total stable transitions that are in the defects

Return type
List of 2-tuple

availableTotalTransitions()

Returns
The total stable and unstable transitions that are in the defects

Return type
List of 2-tuple

availableUnstableTransitions(*defect_filter=None*, *finite_size_correction=True*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Get the unstable charge state transitions in the collection of defects.

Parameters

- **defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.
- **finite_size_correction** (*bool*) – Whether to include the finite size correction.
- **include_vibrations** (*bool*) – Include vibrations, if available, in the report output.
- **temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
The total unstable transitions that are in the defects

Return type
List of 2-tuple

bandGap()

Returns
The band gap in the pristine material

Return type
PhysicalQuantity of type energy

charges()

Returns
The calculated charge states for the defects.

Return type
List of int

defectConfiguration(*index*, *charge*)

Return the defect configuration for the given index and charge state

Parameters

- **index** (*int*) – The index of the defect in the collection.
- **charge** (*int*) – The charge state of the defect

Returns
The configuration of the defect

Return type
BulkConfiguration

defectPositions(*defect_filter*)

Get a list of the defect positions in the C coordinate.

Parameters
defect_filter (*Defect class or tuple of defect classes*) – Filter the results to only include certain types of defects.

Returns
The list of the defect positions of the given types

Return type
PhysicalQuantity of type length

defectTypes()

Returns
A list of the types of defects that are contained within the data set

Return type
list of defect class types

fermiLevel()

Returns
The position of the fermi level in reference to the valence band.

Return type
PhysicalQuantity of type energy

formationEnergies(*charge*, *defect_filter=None*, *element_in_pristine=None*, *finite_size_correction=True*, *electron_chemical_potential=PhysicalQuantity(0.0, eV)*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Calculate the formation energies for a range of defects at a specific charge state. If vibrations are included this energy corresponds to the Helmholtz free energy of formation. If vibrations are not included this energy is the internal electronic energy of formation.

Parameters

- **charge** (*int*) – The charge state of the defect.
- **defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.
- **element_in_pristine** (PeriodicTableElement | All | None) – The element whose chemical potential value is taken from the pristine material. Can be None (elemental reference) All (thermodynamic reference) or an element in the pristine material (element poor reference).
- **finite_size_correction** (*bool*) – Whether to include the finite size correction.
- **electron_chemical_potential** (*PhysicalQuantity* of type energy) – The chemical potential at the defect.
- **include_vibrations** (*bool*) – Include vibrations, if available, in the report output.
- **temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
Tuple of the C-axis position and the formation free energy of the defects

Return type
Tuple of *PhysicalQuantity* of type length and *PhysicalQuantity* of type energy

numberOfDefects()

Returns
The number of defects stored in the object.

Return type
int

positions(*defect_filter=None*)

Return the positions of the defects along the C axis, averaged over charge states.

Parameters
defect_filter (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.

Returns
The defect interface position

Return type
PhysicalQuantity of type length

pristine()

Returns
The pristine configuration common to the defects.

Return type
BulkdConfiguration

projectedDos()

Returns
The projected density of states for the pristine material, if given.

Return type
ProjectedDensityOfState | None

repetitions()

Returns
The repetition of the pristine in the defect structure

Return type
3-tuple of int.

stableTransitions(*transition*, *defect_filter=None*, *finite_size_correction=True*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Calculate the energies of a charge state transitions across the defects.

Parameters

- **transition** (*2-tuple of integer charge states*) – The transition to get the energy of
- **defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.
- **finite_size_correction** (*bool*) – Whether to include the finite size correction.
- **include_vibrations** (*bool*) – Include vibrations, if available, in the report output.
- **temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
Tuple of the C-axis position and the trap level of the defects

Return type
Tuple of *PhysicalQuantity* of type length and *PhysicalQuantity* of type energy

totalTransitions(*transition*, *defect_filter=None*, *finite_size_correction=True*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Calculate the energies of a charge state transitions across the defects, both stable and unstable.

Parameters

- **transition** (*2-tuple of integer charge states*) – The transition to get the energy of

•**defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.

•**finite_size_correction** (*bool*) – Whether to include the finite size correction.

•**include_vibrations** (*bool*) – Include vibrations, if available, in the report output.

•**temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
Tuple of the C-axis position and the trap level of the defects

Return type
Tuple of *PhysicalQuantity* of type length and *PhysicalQuantity* of type energy

uniqueString()

Return a unique string representing the state of the object.

unstableTransitions(*transition*, *defect_filter=None*, *finite_size_correction=True*, *include_vibrations=True*, *temperature=PhysicalQuantity(300.0, K)*)

Calculate the energies of an unstable charge state transitions across the defects.

Parameters

•**transition** (*2-tuple of integer charge states*) – The transition to get the energy of

•**defect_filter** (*Defect class or tuple of defect classes.*) – Filter the results to only include certain types of defects.

•**finite_size_correction** (*bool*) – Whether to include the finite size correction.

•**include_vibrations** (*bool*) – Include vibrations, if available, in the report output.

•**temperature** (*PhysicalQuantity* of type temperature) – Temperature to calculate the formation energy at.

Returns
Tuple of the C-axis position and the trap level of the defects

Return type
Tuple of *PhysicalQuantity* of type length and *PhysicalQuantity* of type energy

Notes

The *InterfaceDefectAnalysis* object enables simpler analysis of defects across an interface. Using this class the defect positions along the interface, formation energies and electron trap levels can be calculated. This object can also be opened in the *Interface Defect Analyzer* to show these properties.

Defects are given as *ChargedPointDefectConfiguration* objects in either a list or a *Table*. The interface in these calculations must be along the C direction of the unit cell. Defects must all be compatible, meaning they have the same pristine configuration, supercell repetitions, charge states, and whether vibrations or finite size corrections were calculated for the defect. Optionally a *ProjectedDensityOfStates* with site projections of the pristine material can also be given. This can be combined with the defect information to show the electron trap levels in reference to the local density of states.

InterfaceFiniteSizeCorrectionParameters

class InterfaceFiniteSizeCorrectionParameters(*dielectric_profile=None*, *gaussian_width=None*, *gaussian_position=None*, *optimize_width=None*, *optimize_position=None*, *averaging_interval=None*, *electrostatic_self_energy_method=None*)

Specify parameters for the Freysoldt and Neugebauer formation energy correction for defects in slabs and interfaces.

Parameters

- **dielectric_profile** (*list of tuples*) – The dielectric constant profile along the interface direction (C). It is assumed a pieceweise constant profile, where each constant interval is defined by its initial point (as fractional coordinates), final point and the dielectric constant. **Default:** [(0., 1., 1.)]
- **gaussian_width** (*PhysicalQuantity* of type length | None) – The width of the Gaussian used to model the charge of the defect.
- **gaussian_position** (*float | None*) – The position of the Gaussian used to model the defect charge along C, in fractional coordinates.
- **optimize_width** (*bool*) – Whether or not to optimize the width of the Gaussian used to model the defect charge.
- **optimize_position** (*bool*) – Whether or not to optimize the position of the Gaussian used to model the defect charge.
- **averaging_interval** (*tuple*) – The interval used for averaging the electrostatic potential along C far from the defect, specified as the fractional distance of the start and end point from the defect (in the positive direction of the lattice vector C). **Default:** (0.3, 0.7)
- **electrostatic_self_energy_method** (*SupercellExtrapolation* or *CylindricalSpectralExpansion*) – The method used to calculate the isolated charge electrostatic self energy, for the periodic charge correction.

averagingInterval()

Returns
The interval used for averaging the electrostatic potential.

Return type
tuple

dielectricProfile()

Returns
The dielectric profile used in the electrostatic correction.

Return type
float

electrostaticSelfEnergyMethod()

Returns
The method used to calculate the isolated charge electrostatic self energy.

Return type
SupercellExtrapolation or *CylindricalSpectralExpansion*

gaussianPosition()

Returns
The position of the Gaussian used to model the defect charge, as fractional distance along C.

Return type
float

`gaussianWidth()`

Returns
The width of the Gaussian used to model the charge of the defect.

Return type
PhysicalQuantity of type length | None

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`optimizePosition()`

Returns
Whether or not to optimize the position of the Gaussian used to model the defect charge.

Return type
bool

`optimizeWidth()`

Returns
Whether or not to optimize the position of the Gaussian used to model the defect charge.

Return type
bool

`uniqueString()`

Return a unique string representing the state of the object.

InterfaceMatch

`class InterfaceMatch(`*surface_configuration_0*, *surface_configuration_1*, *surface_indices*, *layer_tolerance0*, *layer_tolerance1*, *strain_method*`)`

This class represents an interface match as produced by the *InterfaceBuilder*. It can be used to generate an interface structure or queried to determine the properties (e.g. number of atoms or strain) of the match.

Parameters

- **surface_configuration_0** (*BulkConfiguration*) – The bulk configuration representing the first (left) surface.
- **surface_configuration_1** (*BulkConfiguration*) – The bulk configuration representing the second (right) surface.
- **surface_indices** (*sequence of type int*) – The 8 indices representing how the surface lattice vectors for each configuration should be repeated.
- **layer_tolerance0** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.
- **layer_tolerance1** (*int*) – Layers are detected by grouping atoms by their z-coordinates. Increase this tolerance to combine nearby off-lattice atoms into the same layer. Range: [-10, 4]. **Default:** -10.

•**strain_method** (*StrainFirst | StrainSecond | StrainBoth*) – The strain method that should be used.

configurations()

Returns
The two surface configurations used to make the interface.

Return type
list of type *BulkConfiguration*

layeredSurfaces()

Returns
The two surface configurations with custom layers from makeInterface.

Return type
list of type *BulkConfiguration*

makeInterface(*displacement_vector=None*, *layers=None*, *use_periodic_supercell=None*, *quick_optimize=None*, ***kwargs*)

Create an interface with the given number of layers on each side of the interface, optionally with a displacement between the two sides.

Parameters

•**displacement_vector** (*PhysicalQuantity* of type length | None) – This vector is added to the coordinates of the second surface configuration. The meaning of the displacement depends on the choice of `displacement_vector_origin`. **Default:** no displacement.

•**layers** (*sequence | None*) – The start and end layer for both surface configurations. None means the start and end layer are unchanged. **Default:** None.

•**use_periodic_supercell** (*bool | None*) – Ensure that the cell remains periodic when shifting in the C-direction. **Default:** False.

•**quick_optimize** (*bool*) – If True, perform a quick optimization of the interface after its construction to ensure that atoms are adequately spaced from one another at the interface and cell boundaries. The operation is performed by treating the left and right interfaces as rigid bodies and optimizing a generic potential based on covalent radii between the two surfaces. Note that the displacement vector is applied on top of this optimization. **Default:** True.

•**displacement_vector_origin** (*Automatic | None*) – **DEPRECATED!** *Use ``quick_optimize=True`` for ``Automatic`` and ``quick_optimize=False`` for None.* The origin of the displacement vector. The default value represents an arbitrary origin based on the relative position of the atoms in each cell. If `Automatic` is given then the origin (i.e. an all zero displacement) is calculated by treating the left and right interfaces as rigid bodies and optimizing a generic potential based on covalent radii between the two surfaces. **Default:** None.

Returns
The interface configuration. The first layer atoms are tagged as Left Interface and the second layer is tagged as Right Interface.

Return type
BulkConfiguration

meanAbsoluteStrain()

Returns
The average of the absolute values of the three independent components of the engineering strain tensor.

Return type
float

numberOfAtoms()

Returns
The number of atoms in the configuration.

Return type
int

score()

Returns
A subjective score of the quality of the interface match. Structures that are both small and low in mean absolute strain score higher.

Return type
float

strainMethod()

Returns
The strain method.

Return type
StrainFirst | StrainSecond | StrainBoth

strainTensor()

Returns
The 2x2 engineering strain tensor that for the strained surface configuration (as determined by the strain_method).

Return type
ndarray

surfaceIndices()

Returns
The surface indices for this match.

Return type
ndarray of type int

Usage Examples

This object is returned by the `matches` and `bestMatch` methods of *InterfaceBuilder*. See its reference manual entry for more details on how to create interfaces.

Interfaces

class Interfaces(*configurations=None*)

Class to set up configurations with multiple interfaces by: 1. Adding layer configurations 2. Calculating lattice matches 3. Creating an Interfaces configuration.

Layers can be cleaved, resized (in C in terms of atomic planes) and displaced prior to creating the interfaces configuration.

Parameters
configurations (list of *BulkConfiguration*) – The list of configurations to start with. The order of the list will also be the order of the layers.

addLayer(*configuration*)

Add a new configuration as a layer.

Parameters
configuration (*BulkConfiguration*) – The configuration to add.

atomicPlanes(*layer_index*)

The number of atomic planes for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The number of atomic planes.

Return type
int

static availableHKMGConfigurations()

Returns
The list of available HKMG configuration keys.

Return type
list

static availableMRAMConfigurations()

Returns
The list of available MRAM configuration keys.

Return type
list

bestMatch()

Returns
The best match.

Return type
InterfacesMatch

block(*layer_index*)

The block configuration for the given index. The block has a standardized unit cell and, if there is a match selected, will have the surface vectors of the selected match.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The block configuration.

Return type
BulkConfiguration

calculateMatches(*fixed_strain_layer_index=0*)

Calculate new matches.

Parameters
fixed_strain_layer_index (*int | StrainAll*) – The index of the layer for which strain should be fixed.

Returns
The newly calculated matches.

Return type
list of InterfacesMatch

`clearAllLayers()`

Remove all layers.

`cleavePlane(layer_index)`

The cleave plane for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The cleave plane.

Return type
3-tuple of int

`static createAnnealingParameters(annealing_temperature=PhysicalQuantity(600.0, K), annealing_steps=20000, cooling_steps=10000, random_seed=None)`

Setup annealing parameter used for optimizing interfaces.

Parameters

- **annealing_temperature** (PhysicalQuantity of type temperature) – The target temperature.
- **annealing_steps** (*int*) – The number of annealing steps.
- **cooling_steps** (*int*) – The number of cooling steps.
- **random_seed** (*int | None*) – The random seed.

Returns
The annealing parameters.

Return type
list

`displacement(layer_index)`

The displacement vector for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The displacement vector.

Return type
PhysicalQuantity

`static hKMGConfiguration(name)`

Get one of the available HKMG configurations.

Parameters
name (*str*) – The name of the configuration. Should be one of the names in Interfaces.availableHKMGConfigurations()

Returns
The configuration.

Return type
BulkConfiguration

inputConfiguration(*layer_index*)

Get the input configuration for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The input configuration.

Return type
BulkConfiguration

layerLength(*layer_index*)

The layer length for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The layer length.

Return type
PhysicalQuantity of type length

static mRAMConfiguration(*name*)

Get one of the available MRAM configurations.

Parameters
name (*str*) – The name of the configuration. Should be one of the names in Interfaces.availableMRAMConfigurations()

Returns
The configuration.

Return type
BulkConfiguration

makeConfiguration(*match_index=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *optimize=False*)

Make the configuration for the given match.

Parameters

- **match_index** (*InterfacesMatch | Automatic*) – The index of the match to use. If Automatic is given, the best match will be used.
- **optimize** (*bool*) – If True, the final configuration will be geometry-optimized.

Returns
The Interfaces configuration.

Return type
BulkConfiguration

matchIndex()

Returns
The index of the selected match.

Return type
int

matchParameters()

Returns
The currently set match parameters.

Return type
dict

matches()

Returns
The current matches.

Return type
list of InterfacesMatch

maxStrain()

Parameters
max_strain (*float*) – The maximum mean absolute strain, as a fraction in the.

maxSurfaceArea()

:param max_surface_area:The maximum surface area. :type max_surface_area:PhysicalQuantity of type area

maxSurfaceVectorLength()

Parameters
max_surface_vector_length (*PhysicalQuantity of type length*) – The maximum surface vector length.

minSurfaceVectorAngle()

Parameters
min_surface_vector_angle (*PhysicalQuantity of type angle*) – The minimum surface vector angle.

minSurfaceVectorLength()

Parameters
min_surface_vector_length (*PhysicalQuantity of type length*) – The minimum surface vector length.

modifyLayer(*layer_index*, *cleave_plane=None*, *plane_indices=None*, *tolerance=None*, *strip_vacuum=None*, *displacement=None*)

Convenience method to set up a layer.

Parameters

- layer_index (*int*) – The index of the layer to change.
- cleave_plane (*3-tuple of int*) – The cleave plane to set.
- plane_indices (*2-tuple of int*) – The plane indices to set.
- tolerance (*int*) – The logarithmic tolerance to use for detecting atomic planes. Must be in the range [-10, 4].
- strip_vacuum (*bool*) – If True, strip the top and bottom vacuum. Put 1 Å back at the top.
- displacement (*PhysicalQuantity of type length*) – The displacement vector to set, containing the displacements in A, B and C directions.

numberOfLayers()

Returns
The number of layers currently in the store.

Return type
int

numberOfMatches()

Returns
The current number of matches.

Return type
int

planeIndices(*layer_index*)

The plane indices for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The plane indices.

Return type
2-tuple of int

removeLayer(*layer_index*)

Remove the layer with the given index.

Parameters
layer_index (*int*) – The index of the layer to change.

setCleavePlane(*layer_index*, *cleave_plane*)

Set a new cleave plane to the layer with the given index.

Parameters

- **layer_index** (*int*) – The index of the layer to change.
- **cleave_plane** (*3-tuple of int*) – The cleave plane to set.

setDisplacement(*layer_index*, *displacement*)

Set a new displacement for the layer with the given index.

Parameters

- **layer_index** (*int*) – The index of the layer to change.
- **displacement** (*PhysicalQuantity of type length*) – The displacement vector to set, containing the displacements in A, B and C directions.

setMatchParameters(*max_strain=None*, *max_surface_area=None*, *min_surface_vector_length=None*, *max_surface_vector_length=None*, *min_surface_vector_angle=None*)

Convenience method to set all match parameters at once.

Parameters
max_strain (*float*) – The maximum mean absolute strain to include in the lattice match, as a fraction in the range [0.0, 1.0].

:param max_surface_area:The maximum surface area to include in the lattice match. :type max_surface_area:PhysicalQuantity of type area

Parameters

- **min_surface_vector_length** (*PhysicalQuantity of type length*) – The minimum surface vector length to include in the lattice match.

- **max_surface_vector_length** (*PhysicalQuantity of type length*) – The maximum surface vector length to include in the lattice match.
- **min_surface_vector_angle** (*PhysicalQuantity of type angle*) – The minimum surface vector angle to include in the lattice match.

setMaxStrain(*max_strain*)

Set a new The maximum mean absolute strain to include in the lattice match, as a fraction in the range [0.0, 1.0].

Parameters
max_strain (*float*) – The maximum mean absolute strain to include in the lattice match, as a fraction in the.

setMaxSurfaceArea(*max_surface_area*)

Set a new mum surface area to include in the lattice match.

:param max_surface_area:The maximum surface area. :type max_surface_area:PhysicalQuantity of type area

setMaxSurfaceVectorLength(*max_surface_vector_length*)

Set a new maximum surface vector length to include in the lattice match.

Parameters
max_surface_vector_length (*PhysicalQuantity of type length*) – The maximum surface vector length.

setMinSurfaceVectorAngle(*min_surface_vector_angle*)

Set a new minimum surface vector angle to include in the lattice match.

Parameters
min_surface_vector_angle (*PhysicalQuantity of type angle*) – The minimum surface vector angle.

setMinSurfaceVectorLength(*min_surface_vector_length*)

Set a new minimum surface vector length to include in the lattice match.

Parameters
min_surface_vector_length (*PhysicalQuantity of type length*) – The minimum surface vector length.

setPlaneIndices(*layer_index*, *plane_indices*, *tolerance=-10*)

Set new plane indices for the layer with the given index.

Parameters

- **layer_index** (*int*) – The index of the layer to change.
- **plane_indices** (*2-tuple of int*) – The plane indices to set.
- **tolerance** (*int*) – The logarithmic tolerance to use for detecting atomic planes. Must be in the range [-10, 4].

setStripVacuum(*layer_index*, *strip_vacuum*)

Set the strip vacuum flag to a layer.

Parameters

- **layer_index** (*int*) – The index of the layer to change.
- **strip_vacuum** (*bool*) – If True, strip the top and bottom vacuum. Put 1 Å back at the top.

strainData(*strain_type*, *as_percentages=False*)

Get formatted match data in terms of the maximum or average mean absolute strains vs. the number of atoms.

Parameters

- **strain_type** (*Interfaces.MAX_MEAN_ABSOLUTE_STRAIN | Interface.AVG_MEAN_ABSOLUTE_STRAIN*) – The type of data to get.

- **as_percentages** (*bool*) – If True, the maximum or average mean absolute strains are given in percent ([0.0, 100.0]) instead of a fraction ([0.0, 1.0]).

Returns
The formatted match data.

Return type
2-tuple of list

stripVacuum(*layer_index*)
The strip vacuum flag.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The strip vacuum flag.

Return type
bool

title(*layer_index*)
The title of the given layer.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The layer title.

Return type
str

tolerance(*layer_index*)
The tolerance for detecting atomic planes for the given index.

Parameters
layer_index (*int*) – The index of the layer.

Returns
The tolerance for detecting atomic planes.

Return type
int

Usage Examples

Create a configuration with three interfaces using layers of Si (001), SiO2 (quartz, 001) and Au (111).

```
from QuantumATK import *
from ExampleConfigurations import configurations

# Create Interfaces instance.
interfaces = Interfaces(configurations=configurations)

# Set up Layer 0.
interfaces.modifyLayer(
    layer_index=0,
    cleave_plane=(0, 0, 1),
```

(continues on next page)

(continued from previous page)

```
    plane_indices=(0, 3),
    tolerance=-10,
    displacement=(0.0, 0.0, 0.0) * Angstrom,
)

# Set up Layer 1.
interfaces.modifyLayer(
    layer_index=1,
    cleave_plane=(0, 0, 1),
    plane_indices=(0, 8),
    tolerance=-10,
    displacement=(0.0, 0.0, 0.4) * Angstrom,
)

# Set up Layer 2.
interfaces.modifyLayer(
    layer_index=2,
    cleave_plane=(1, 1, 1),
    plane_indices=(0, 2),
    tolerance=-10,
    displacement=(0.0, 0.0, 0.0) * Angstrom,
)

# Set up the match parameters.
interfaces.setMatchParameters(
    max_strain=0.08,
    max_surface_area=300.0 * Angstrom**2,
    min_surface_vector_length=1.0 * Angstrom,
    max_surface_vector_length=20.0 * Angstrom,
    min_surface_vector_angle=30.0 * Degrees,
)

# Generate matches.
interfaces.calculateMatches(
    fixed_strain_layer_index=0
)

# Create the interfaces configuration.
configuration = interfaces.makeConfiguration(
    match_index=12,
    strip_vacuum=False,
    optimize=False
)
```

`si_sio2_au_example.py` `ExampleConfigurations.py`

The Interfaces class is first instantiated with the input configurations. Next, each layer is modified and the match parameters set. Specifically, the SiO2 layer is displaced by 0.4 Å in the C direction and the Au layer is cleaved to expose the (1, 1, 1) surface. This is followed by a call to calculateMatches, with the Si layer (layer index 0) having its strain fixed to 0. This method loops over all possible interface matches within the limits set by the match parameters. Finally, makeConfiguration is called with a given match_index to produce the final interfaces stack configuration (see figure).

This configuration also contains strain information in its metatext, which can be viewed in the preview pane of the Data

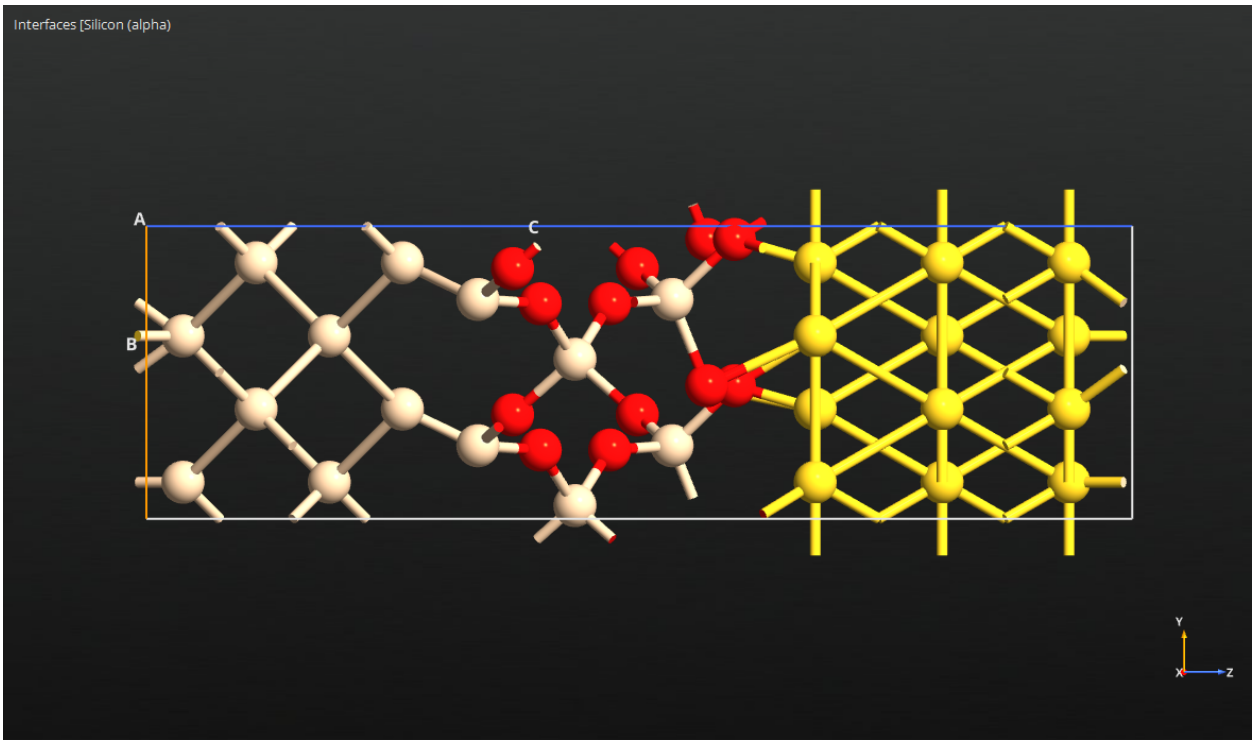


Fig. 4.31: The resulting interfaces configuration.

View or in script by using

```
configuration.metatext()
```

General

The **Interfaces** class of QuantumATK allows the construction of multilayered configurations by automatically matching the 2D unit cells of an arbitrary number of layers and comparing the strain. The input configurations should be crystalline, but are not required to be. All input configurations will be cleaved to (001) unless another cleave plane is given, and made into slabs before being added to the multilayered configuration.

Notes

- The layers are automatically adjusted to standardized unit cells when creating interfaces. In QuantumATK this means that the lattice vectors are prepared according to the following convention: the a-vector is along the x-axis, the b vector is in the a-b plane, the c vector is along the z-axis.
- The parameter `plane_indices` can be used to expose different cut-planes at the interface of a given layer and to set the thickness of the layer.
- Using `match_index=Automatic` in the input parameters returns a match that balances the total number of atoms in the system with the mean absolute strain. It can be used to find a good match without having to filter through all possible matches.
- Using `quick_optimize` requires a valid QATK-ForceField license. This option only performs a generic pre-optimization to avoid too close atoms.

- When setting the `strip_vacuum`, `displacements` and `quick_optimize` options, the operations will be performed in that order. The layers are first stripped of vacuum along the C-direction before the interfaces configuration is produced. Atoms are then shifted based on any given (per-layer) displacements using layer index tags. Finally, quick-optimization is performed on the displaced configuration, which means the displacements may change. Essentially, if you use both displacements and quick-optimization, the displacements can be seen as setting the initial conditions for the optimization.

Interstitial

class Interstitial(*element*, *cartesian_coordinates=None*, *fractional_coordinates=None*)

Defines an interstitial defect. Used as part of a *ChargedPointDefect* study.

Parameters

- **element** (*PeriodicTableElement*) – The element to add at the given position.
- **cartesian_coordinates** (sequence (size 3) of *PhysicalQuantity* of type length) – The position at which to insert the interstitial atom, given in absolute coordinates. This option is mutually exclusive to `fractional_coordinates`. **Default:** None
- **fractional_coordinates** (*sequence (size 3) of float*) – The position at which to insert the interstitial atom, given in fractional coordinates of the bulk unit cell. This option is mutually exclusive to `cartesian_coordinates`. **Default:** None

cartesianCoordinates()

Returns
The coordinates of the interstitial defect as Cartesian coordinates if they have been specified, otherwise returns None.

Return type
sequence (size 3) of *PhysicalQuantity* of type length | None

element()

Returns
The element used in the interstitial defect.

Return type
PeriodicTableElement

fractionalCoordinates()

Returns
The coordinates of the interstitial defect as fractional coordinates if they have been specified, otherwise returns None.

Return type
`numpy.array` (size 3) of float | None

generateDefectConfiguration(*reference_bulk_configuration*, *supercell_repetitions*, *use_ghost=None*)

Generate the supercell with the embedded defect.

Parameters

- **reference_bulk_configuration** (*BulkConfiguration*) – The bulk unit cell configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
The supercell configuration with the embedded defect.

Return type
BulkConfiguration

uniqueString()

Return a unique string representing the state of the object.

Notes

The *Interstitial* object defines an interstitial defect in a material.

An *Interstitial* can be added to a *NamedPointDefect* to define a defect for use in a *ChargedPointDefectConfiguration* calculation.

See *Defining the point defect* for a description of how to use this object as part of a *ChargedPointDefect* study.

InterstitialGenerator

class InterstitialGenerator(*host_configuration*, *interstitial_element*, *interstitial_types=None*, *symmetry_tolerance=None*, *group_distance=None*)

A class which describes the possible interstitial defects for a given host material.

Parameters

- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **interstitial_element** (*PeriodicTableElement*) – The element to add as interstitial defect.
- **interstitial_types** (*VoronoiVertices | VoronoiFaces | VoronoiRidges | sequence of VoronoiVertices, VoronoiFaces, VoronoiRidges*) – How the interstitials in the material should be automatically created. **Default:** VoronoiVertices.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.
- **group_distance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining groups of defects. **Default:** 0.1 * Angstrom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters
top_n_configurations (*int*) – Number of most distinct configurations.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByLatticeSpecies(*element_list*)

This method is not implemented

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

•**point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.

•**keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`

•**select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** `1`

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)

Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)

Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

groupDistance()

Returns
The tolerance to be used when determining groups of defects.

Return type
PhysicalQuantity of type length

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

interstitialElement()

Returns
The element to add as interstitial defect.

Return type
PeriodicTableElement

interstitialTypes()

Returns
How the interstitials in the material should be automatically created.

Return type
VoronoiVertices | VoronoiFaces | VoronoiRidges

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

symmetryTolerance()

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

uniqueDefects()

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

uniqueString()

Return a unique string representing the state of the object.

Notes

The *InterstitialGenerator* object generates the symmetrically unique interstitial defects for a given host material. Possible sites for adding interstitial atoms are determined by calculating the Voronoi volume of each atom. Interstitial atoms can then be added at the vertices, the faces, or between the vertices of the Voronoi volume. Using symmetry analysis, these points are then reduced to a symetrically unique set of points. The *InterstitialGenerator* object is derived from the *BaseDefectGenerator* object, and thus contains the common methods from that object.

To create a *InterstitialGenerator* object, a specific host configuration must be given along with the interstitial element. Optionally, the kinds of Voronoi positions can also be specified. This is done using the argument `interstitial_types``argument which can take one or a list of the ``VoronoiFaces`, `VoronoiVertices` or `VoronoiRidges` keywords. Once the generator is created filters can be added so that only only a desired subset of defects can be generated.

The unique defects can be returned with the `uniqueDefects` method as *NamedPointDefect* objects, which can be used as input for a *ChargedPointDefectConfiguration* object. The defects and generators can also be given to a *Defects* object to create the basic defect configurations.

IonicConductivity

class IonicConductivity(*md_trajectory*, *start_time=None*, *end_time=None*, *atom_selection=None*, *anisotropy=None*, *ignore_measurement_data=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the IonicConductivity object.

Parameters

- **md_trajectory** (*MDTrajectory* | `AtomicConfiguration`) – The MDTrajectory from which the ionic conductivity is calculated.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **atom_selection** (*PeriodicTableElement* | str | list of ints | list of *PeriodicTableElement* or str) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **anisotropy** (*list of type int | int | None*) – The list of Cartesian directions (x=0, y=1, z=2) to calculate the anisotropic ionic conductivity in or a single Cartesian direction. By default an isotropic calculation is performed. **Default:** None.
- **ignore_measurement_data** (*bool*) – Whether or not to use stored measurement data. **Default:** False.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.

chargeCurrentAutocorrelation()

Returns
The autocorrelation of the charge current.

Return type
`PhysicalQuantity` of type squared charge velocity.

data()

Returns
The ionic conductivity values.

Return type
PhysicalQuantity of type conductance per time.

times()

Returns
Return the times of the Green-Kubo integral.

Return type
PhysicalQuantity of type time.

Usage Examples

Calculate the ionic conductivity of lithium atoms in a battery electrolyte.

```
# Load in an equilibrated initial configuration with calculator
bulk_configuration = nlread('Battery_Electrolyte.hdf5')[-1]

# Calculate the trajectory, including charge current measurements
method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=340*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=ConfigurationVelocities(remove_center_of_mass_momentum=True),
)

constraints = [FixCenterOfMass()]
measurement_names = ['charge_current', 'charge_current_components']

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Battery_Electrolyte_Trajectory.hdf5',
    steps=10000000,
    trajectory_interval=10000,
    measurement_hook=MDMeasurement(measurement_names, call_interval=10),
    method=method
)

# Calculate the ionic conductivity
self_diffusion = IonicConductivity(
    md_trajectory,
    atom_selection=Lithium
)

# Get the times in ps and the conductivity values in Siemens/Meter.
```

(continues on next page)

(continued from previous page)

```
times        = self_diffusion.times()
conductivity = self_diffusion.data()

# Plot the diffusion coefficient calculated with different time limits
model = Plot.PlotModel(ps, Siemens/Meter)
model.title().setText('Ionic Conductivity of Lithium Electrolyte')
model.xAxis().setLabel('Time')
model.yAxis().setLabel('Conductivity')
model.legend().setVisible(False)

# Add diffusion integral
line = Plot.Line(times, conductivity)
line.setColor('blue')

# Add average
average = Plot.Average()
line.addItem(average)
max_time = times[-1]
average.setBounds(max_time*0.1, max_time*0.9)
average.setColor('red')

# Add line to the plot
model.addItem(line)

# Set limits and show the plot
model.setLimits()
Plot.show(model)
```

ionic_conductivity.py

Notes

The *IonicConductivity* σ is calculated from the ionic current auto-correlation function using the Green-Kubo expression:

$$\sigma = \frac{1}{3k_BTV}\int_0^\infty d\tau \langle \mathbf{j}_i(t)\mathbf{j}_i(t+\tau)\rangle$$

Here the ensemble average $< \ldots >$ averages over time origins t. The ionic current $\mathbf{j}(t)$ is given as the sum of the individual charge velocities:

$$\mathbf{j}(t) = \sum_i^N q_i \mathbf{v}_i(t)$$

The atomic charges are taken from the atomic partial charges assigned to each atom. These need to be set in the *MDTrajectory* used for the analysis.

When calculating quantities using Green-Kubo expressions, it is important that the trajectory both contains a short enough time step to be able to determine the decay in the auto-correlation function, as well as a long enough time scale to encompass the auto-correlation time length and provide enough time origins for accurate sampling. When calculating the ionic conductivity, ideally as the charge current auto-correlation decreases to zero, the above integral should converge to a static value at longer time scales. Practically however, longer times in the auto-correlation function corresponds to fewer time origins in the ensemble average. This increases the sampling error and can lead to non-convergence in the charge current auto-correlation functions and subsequently the ionic conductivity.

As in each time point only a single sum of charge currents is required, it is possible to save this data in an *MDTrajectory* as a measurement. This can be used to increase the frequency of data without significantly increasing the size of the trajectory. To add this measurement during a molecular dynamics simulation the measurements `charge_current` and `charge_current_component` can be used. The first saves just the total charge current to the *MDTrajectory*, while the second saves individual charge currents for each element and tag in the configuration. This is useful for calculating the ionic conductivity of different components of the system. When the *MDTrajectory* is loaded by the *IonicConductivity* analysis it looks for these two measurements, and if present the ionic conductivity is calculated from these. Setting the argument `ignore_measurement_data` to `True` forces the analysis to be performed from just the configurations contained in the trajectory.

The *IonicConductivity* analysis method returns the calculated ionic conductivity using different time limits. The optimal value of the ionic conductivity is usually taken in the region where the auto-correlation has decayed sufficiently for the integral to converge, and also there are enough time origins for accurate sampling. The auto-correlation time scale can vary significantly depending on the potential acting on the atoms.

By default, all elements are taken into account, but a specified selection can be given as well. The `atom_selection` parameter accepts an element, a tag name, or a list of atom indices to select which charge currents are included in the calculation. This can be useful for when only certain atoms should be included in the analysis, such as when there are constrained atoms in the system that excluded. It is also possible to select to only calculate ionic conductivity in specific Cartesian directions using the `anisotropy` argument. This can be useful in cases such as calculating the ionic conductivity of atoms between two surfaces.

IrregularTimeMeanSquareDisplacement

class IrregularTimeMeanSquareDisplacement(*trajectories*, *time_measurement=None*, *position_measurement=None*, *maximum_time=None*, *atom_selection=None*, *anisotropy=None*, *resolution=None*, *prune_data=None*, *verbose=None*)

Calculate the mean square displacement of a group of atomic positions. It is assumed that the times are unevenly spaced, and thus this algorithm is suited to being used with hypertime or kinetic Monte Carlo trajectories.

Parameters

- **trajectories** (MDTrajectory | *sequence of* MDTrajectory) – The trajectories to be analyzed
- **time_measurement** (*str*) – The key used for the simulation time in measurements. If `None` is given the normal molecular dynamics time will be used.
- **position_measurement** (*str*) – The key used for the atom positions in measurements. If `None` is given the trajectory atomic positions will be used.
- **maximum_time** (*str*) – The maximum time to measure the mean square displacement over. If `None` is given the average length of the input trajectories is used.
- **atom_selection** (`None` | PeriodicTableElement | str | sequence of PeriodicTableElement or str) – List of either elements or tags used to select atoms from the trajectory. This can only be used when using the configurations in the trajectory.
- **anisotropy** (int | sequence of int | `None`) – The Cartesian directions (x=0, y=1, z=2) to calculate the anisotropic MSD. **Default:** `None`.
- **resolution** (*int*) – The number of divisions in the mean square displacement histogram.
- **prune_data** (*int*) – Use every nth timestep for the analysis. A value of 1 uses all data.
- **verbose** (*PhysicalQuantity* of type length) – Print the progress of the analysis as it runs.

dataCount()

Returns
The number of data points used in each time bin.

Return type
Sequence of int

dataIndices()

Returns
The indices of the times that are used in calculating the mean square displacement. Times in which no displacements are found are removed from the final histogram.

Return type
ndarray

displacementStandardDeviation()

Returns
The standard deviation of the displacement histogram.

Return type
PhysicalQuantity of type square length

displacementStandardError()

Returns
The standard error of the displacement histogram.

Return type
PhysicalQuantity of type square length

displacementVariance()

Returns
The variance of the displacement histogram.

Return type
PhysicalQuantity of type quartic length

static fromData(*times*, *positions*, *maximum_time=None*, *atom_selection=None*, *anisotropy=None*, *resolution=None*, *prune_data=None*, *verbose=None*)

Create a IrregularTimeMeanSquareDisplacement object from time and position data.

Parameters

- **times** (*PhysicalQuantity* of type time) – The times of each state.
- **positions** (*PhysicalQuantity* of type distance) – The atomic positions of each state.
- **maximum_time** (*str*) – The maximum time to measure the mean square displacement over. IfNone is given the average length of the input trajectories is used.
- **atom_selection** (*None | PeriodicTableElement | str | sequence of PeriodicTableElement or str*) – List of either elements or tags used to select atoms from the trajectory. This can only be used when using the configurations in the trajectory.
- **anisotropy** (*int | sequence of int | None*) – The Cartesian directions (x=0, y=1, z=2) to calculate the anisotropic MSD. **Default:** None.
- **resolution** (*int*) – The number of divisions in the mean square displacement histogram.
- **prune_data** (*int*) – Use every nth timestep for the analysis. A value of 1 uses all data.

•**verbose** (*bool*) – Print the progress of the analysis as it runs.

Returns
An IrregularTimeMeanSquareDisplacement object.

Return type
IrregularTimeMeanSquareDisplacement

meanSquareDisplacement()

Returns
The mean square displacement histogram.

Return type
PhysicalQuantity of type square length

times()

Returns
The times at which the mean square displacement histogram was calculated.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

In this example the mean square displacement (MSD) is calculated for a trajectory from a hyperdynamics simulation. Such a trajectory might come from the diffusion example in *CollectiveVariableHyperdynamics* documentation. Here the `Hypertime` measurement is used for the time data and the `tracked_atoms` measurement for the position data. The data is also pruned so that only every second data point is used.

```
msd = IrregularTimeMeanSquareDisplacement(
    trajectories=trajectory,
    time_measurement='Hypertime',
    position_measurement='tracked_atoms',
    prune_data=2,
    verbose=True,
)
```

Once the MSD is calculated it can be plotted and a line fitted to calculate the diffusion coefficient. This is 1/6th the gradient of the MSD.

```
# Create the plot model
model = Plot.PlotModel(ps, Angstrom**2)
model.xAxis().setLabel('Time')
model.yAxis().setLabel('Mean Square Displacement')
model.legend().setVisible(True)

# Add the MSD data and a linear fit to the plot
line = Plot.Line(msd.times(), msd.meanSquareDisplacement())
line.setLabel('MSD Data')
line.setColor('mediumpurple')
fit = Plot.PolynomialFit(order=1)
```

(continues on next page)

(continued from previous page)

```
line.addItem(fit)
model.addItem(line)

# Set the range of the fit so that it can be easily adjusted
max_time = msd.times()[-1]
fit.setBounds(0.1 * max_time, 0.9 * max_time)

# Show the plot
model.setLimits()
Plot.show(model)
```

Notes

The *IrregularTimeMeanSquareDisplacement* class implements the calculation of the mean square displacement (MSD) of atoms in cases where the time steps between atomic positions are not evenly spaced. These kinds of trajectories can be generated by simulations such as hyperdynamics or *KineticMonteCarlo*. In hyperdynamics the elapsed time between frames depends on the applied bias potential. Similarly in kinetic Monte Carlo the elapsed time between images depends on the rates to other states from the current state.

In order to calculated the MSD the *IrregularTimeMeanSquareDisplacement* selects a period of time to calculate the MSD over, and then divides that up into a number of segments. A histogram of atomic square displacements is then constructed by binning each square displacement into the nearest time segment. With a regularly spaced trajectory this can be done exactly by using the time step size as the time segment length. With irregular time steps each step must be rounded into the nearest time segment. From the histogram the average square displacement is calculated at each time segment. Additionally the variance, standard deviation and standard error of the mean are also calculated. These give a measure of the statistical uncertainty in the MSD calculation. These do not account for systematic errors, due largely to insufficient sampling of the of the system.

To calculate the MSD one or more *MDTrajectory* objects must be given. If multiple trajectories are given then the MSD is averaged over all the trajectories. The atomic time and position data can be taken either directly from the trajectory or from measurements contained within the trajectory. The keyword arguments `time_measurement` and `position_measurement` can be used to specify the measurements for the time and position respectively. If not given then the data is taken from the trajectory directly. Regardless of the source, the time and position data must have the same number of entries. If the position data is taken from the trajectory then the atoms used can be selected either by tags or element type. This is done with the keyword argument `atom_selection`. Calculating the MSD in specific cartesian directions is also possible using the `anisotropy` keyword argument. If this is not given then the radial MSD is calculated.

A number of options control how the histogram is constructed. The total histgram time can be selected using the `maximum_time` keyword argument. If not given then the average length of the trajectories is used. Setting this explicitly can be useful in cases where the MSD is calculated from multiple trajectories with substantially different lengths. The number of segments is likewise given with the `resolution` keyword argument. Formally calculating the MSD scales as $O(N^2)$, where N is the number of data points. This can take a significant amount of time if a very large number of data points are used. The `prune_data` keyword allows giving the spacing between data points. Giving a value of 1 will use all data points, while a value of 2 will only use every second data point. Finally, additional information tracking the progress of the calculation can be printed using the `verbose` keyword argument. This is useful in estimating the time required to calculate the MSD when using large trajectories or many steps.

IsotropicFiniteSizeCorrectionParameters

class IsotropicFiniteSizeCorrectionParameters(*dielectric_constant=None*, *gaussian_width=None*, *optimize_width=None*, *optimize_position=None*, *min_cartesian_averaging_distance=None*, *max_fractional_averaging_distance=None*, *averaging_direction=None*, *bulk_modulus=None*)

Specify parameters for the Freysoldt, Neugebauer and Van de Walle Bulk formation energy correction for defects.

Parameters

- **dielectric_constant** (float | *OpticalSpectrum*) – The dielectric constant of the pristine material. Can also be given as an OpticalSpectrum analysis object from which the dielectric constant is calculated. **Default:** 1.0
- **gaussian_width** (*PhysicalQuantity* of type length | None) – The width of the Gaussian used to model the charge of the defect. Specifying None allows the width of the defect to be optimized. **Default:** None
- **optimize_width** (*bool*) – Whether or not to optimize the width of the Gaussian used to model the defect charge. **Default:** True
- **optimize_position** (*bool*) – Whether or not to optimize the position of the Gaussian used to model the defect charge. **Default:** False
- **min_cartesian_averaging_distance** (*PhysicalQuantity* of type length) – The distance from the defect, in Cartesian coordinates at which averaging of the defect potential begins. If, in fractional coordinates, this distance is greater than the maximum distance given in fractional coordinate, the maximum fractional distance is used.
- **max_fractional_averaging_distance** (*float*) – The maximum distance, in fractional coordinates, at which that the averaging of the long-range defect position begins. If this value is greater, in Cartesian coordinates then the minimum Cartesian averaging distance, this length is used.
- **averaging_direction** (*int | None*) – The integer index (0, 1, 2) of the direction in which the potential is varied. If None is given the potential is determined in the longest dimension of the material. **Default:** None

averagingDirection()

Returns
The integer index (0, 1, 2) of the direction in which the potential is varied.

Return type
int | None

bulkModulus()

Returns
The bulk modulus used in the finite size stress correction.

Return type
float | None

dielectricConstant()

Returns
The dielectric constant used in the electrostatic correction.

Return type
float

gaussianWidth()

Returns
The width of the Gaussian used to model the charge of the defect.

Return type
PhysicalQuantity of type length | None

`maxFractionalAveragingDistance()`

Returns
The maximum distance, in fractional coordinates where the averaging of the long-range defect potential begins.

Return type
float

`minCartesianAveragingDistance()`

Returns
The distance from the defect, in Cartesian coordinates that averaging begins at. If this distance is greater than the maxFractionalAveragingDistance, when converted to fractional coordinates, the maxFractionalAveragingDistance is used.

Return type
float | *PhysicalQuantity* of type length.

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`optimizePosition()`

Returns
Whether or not to optimize the position of the Gaussian used to model the defect charge.

Return type
bool

`optimizeWidth()`

Returns
Whether or not to optimize the position of the Gaussian used to model the defect charge.

Return type
bool

`uniqueString()`
Return a unique string representing the state of the object.

Notes

The *IsotropicFiniteSizeCorrectionParameters* object contains the parameters for the finite size charge correction, as suggested by Freysoldt, Neugebauer and Van De Walle[1][2]. This corrects for the artificial interactions of a charged defect with itself due to periodic boundary conditions. The method assumes that both the charge is localized to a point in the defect configuration, and also that the defect environment is isotropic. A model Gaussian charge distribution at the defect is fitted to the difference between the pristine and defect potentials. This charge is then used to correct both

[1] Christoph Freysoldt, Jörg Neugebauer, and ChrisăG. Van De Walle. Fully Ab Initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, 2009. doi:10.1103/PhysRevLett.102.016402.

[2] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and ChrisăG. Van De Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86(1):253–305, 2014. doi:10.1103/RevModPhys.86.253.

the long and short range energy of the additional charge. Optionally an additional correction for the stress created by the defect can also be added.

A default *IsotropicFiniteSizeCorrectionParameters* can be created without any parameters. For the charge correction, the dielectric constant of the pristine material is required. This can be specified using the `dielectric_constant` arguement. This can be either a float or a :ref:`OpticalSpectrum_c` analysis object from which the dielectric constant can be extracted. Likewise, for the stress correction, a value for the pristine bulk modulus is required. This can be either specified directly or given as an :ref:`ElasticConstants_c` analysis calculation using the `bulk_modulus` argument.

In fitting the Gaussian charge distribution the potential is projected along one axis of the unit cell. By default this is the longest direction, but can also be set using the `averaging_direction` argument. This can be useful in cases such as interfaces, where one cell dimension is not isotropic. The distance away from defect where the potential averaging begins can also be set. The minimum distance is set in Cartesian coordinates using the argument `min_cartesian_averaging_distance`. The maximum distance can likewise be set using the argument `max_fractional_averaging_distance`. These two arguments allow the averaging range to be set a reasonable distance from the defect without being too close. These settings also allow the same correction to be used with defects of different supercell sizes.

The width of the Gaussian charge distribution can be directly set using the `gaussian_width` argument. The correcting charge distribution can also be optimized by setting the `optimize_width` argument to `True`. Similarly, optimizing the position of the defect along the projection direction is selected by setting the `optimize_position` argument to `True`.

IterationControlParameters

class IterationControlParameters(*tolerance=None*, *max_steps=None*, *algorithm=None*, *damping_factor=None*, *number_of_history_steps=None*, *start_mixing_after_step=None*, *mixing_variable=None*, *preconditioner=None*, *linear_dependence_threshold=None*, *exx_energy_tolerance=None*, *max_exx_updates=None*, *non_convergence_behavior=None*, *enable_scf_stop_file=None*)

Class for representing the parameters relevant for SCF iteration control within self-consistent electronic structure models.

Some parameter defaults are specific for each calculator, see *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *LCAOCalculator*, *PlaneWaveCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*, or *DeviceLCAOCalculator*.

Parameters

- **tolerance** (*float*) – The accuracy which the SCF loop must reach before termination. Should be positive. **Default:** Specific for each calculator. When doing hybrid DFT calculations, this will also be the tolerance used for the outer SCF loop that updates the exact exchange operator.
- **max_steps** (*int*) – The maximum number of iteration steps the SCF loop should execute. Should be non-negative. **Default:** `100`
- **algorithm** (*PulayMixer* | *AndersonMixer*) – The mixing strategy employed on the mixing variable in the SCF loop. **Default:** A default *PulayMixer* instance
- **damping_factor** (float | `AdaptiveDampingFactor`) – The fraction of the output value of the mixing parameter which is mixed into the next step. Should be at least 0 and smaller than 1. For bulk LCAO calculations, it is also possible to provide an instance of `AdaptiveDampingFactor`. In this case, the damping factor will be changed to an optimal value for the system during the SCF loop. The optimal value is calculated from the systems band gap. Using an adaptive damping factor can improve the SCF convergence especially for semi-conductor and insulator systems. **Default:** `0.1`
- **number_of_history_steps** (*int*) – The total number of history steps to take into account for the mixing algorithm. Should be positive. **Default:** `min(20, max_steps)`

- **start_mixing_after_step** (*int*) – The total number of steps to wait before the mixing algorithm is used. Should be non-negative. **Default:** `0`
- **mixing_variable** (`HamiltonianVariable`) – The mixing variable used in the SCF loop. **Default:** A default `HamiltonianVariable` instance
- **preconditioner** (`Preconditioner.Off` | *Kerker*) – The preconditioner used in the SCF loop. If `damping_factor` is an instance of `AdaptiveDampingFactor`, *Kerker* preconditioner is not supported. **Default:** `Preconditioner.Off`
- **linear_dependence_threshold** (*float*) – Pulay steps which have a linear dependence lower than the threshold are discarded. Should be non-negative. **Default:** `0.0`
- **exx_energy_tolerance** (*Not used.*) – **Deprecated:** from v2019.12, see `tolerance`.
- **max_exx_updates** (*int*) – The maximum number of updates of the exact exchange operator that will be performed. If this number is reached, the SCF loop will be terminated whether it is converged or not. Should be an integer larger than 0. This parameter is only used in hybrid DFT calculations. **Default:** `50`
- **non_convergence_behavior** (`StopCalculation` | `ContinueCalculation`) – Determines the behavior when the SCF loop does not reach the desired tolerance. **Default:** `ContinueCalculation()`
- **enable_scf_stop_file** (*bool*) – Determines whether to enable a file for stopping the SCF loop. If `True`, creation of the stop file will stop the SCF at the next step. The name of the file for this SCF calculation will be shown in the log output. The file must be created in the current working directory. **Default:** `True`

algorithm()

Returns
The mixing algorithm stored by the *IterationControlParameters* object.

Return type
PulayMixer | *AndersonMixer*

dampingFactor()

Returns
The damping factory stored by the *IterationControlParameters* object.

Return type
float | `AdaptiveDampingFactor`

enableScfStopFile()

Returns
Determines whether to enable a file for stopping the SCF loop.

Return type
bool

exxEnergyTolerance()

Deprecated: from v2019.12

linearDependenceThreshold()

Returns
The linear dependence threshold value stored by the *IterationControlParameters* object.

Return type
float

maxExxUpdates()

Returns
The maximum number of exact exchange operator updates.

Return type
int

maxSteps()

Returns
The maximum steps value stored by the *IterationControlParameters* object.

Return type
int

mixingVariable()

Returns
The mixing variable stored by the *IterationControlParameters* object.

Return type
HamiltonianVariable

nlinfo()

Returns
The nlinfo.

Return type
dict

nonConvergenceBehavior()

Returns
Determines the behavior when the SCF loop does not reach the desired tolerance.

Return type
StopCalculation | ContinueCalculation

numberOfHistorySteps()

Returns
The number of history steps value stored by the *IterationControlParameters* object.

Return type
int

Note: The value returned is bounded above by the maximum number of steps, and below by 1.

---

preconditioner()

Returns
The preconditioner stored by the *IterationControlParameters* object.

Return type
Preconditioner.Off | *Kerker*

startMixingAfterStep()

Returns
The step to start mixing after value stored by the *IterationControlParameters* object.

Return type
int

`tolerance()`

Returns
The tolerance value stored by the *IterationControlParameters* object.

Return type
float

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Setup iteration control parameters for difficult convergence case

```
iteration_control_parameters = IterationControlParameters(
    damping_factor=0.05,
    number_of_history_steps=12,
    )

calculator = HuckelCalculator(
    iteration_control_parameters=iteration_control_parameters,
    )
```

Setup iteration control parameters with a preconditioner for a large system that is difficult to converge

```
iteration_control_parameters = IterationControlParameters(
    damping_factor=0.2,
    preconditioner=Kerker(0.02*Hartree, 0.5*Hartree, 0.01),
    mixing_variable=HamiltonianVariable,
    number_of_history_steps=12,
    )

calculator = LCAOCalculator(
    iteration_control_parameters=iteration_control_parameters,
    )
```

Notes

Mixing algorithm

The available SCF mixing algorithms are the distributed and non-distributed version of the *PulayMixer*, and the *AndersonMixer*. The mixing algorithm can be set using the `algorithm` argument to *IterationControlParameters*.

Preconditioner

It is possible to add a preconditioner within the SCF loop. For `HamiltonianVariable` mixing the preconditioner will damp the variations in the low Fourier components of the Hartree potential. Currently a *Kerker* style preconditioner is available when *PulayMixer* is selected as mixing algorithm.

Tolerance and accuracy

A few important notes on the keyword argument `tolerance`.

- The `tolerance` in QuantumATK is not relative, but absolute. For the total energy and the band-structure energy the internal unit is Hartree, thus for these quantities the tolerance gives the accuracy as measured in Hartree. For iteration control of a matrix (e.g., the Hamiltonian or the density matrix) it is maximum value of the difference between the corresponding matrix elements of the old and new matrix. This means that the tolerance should be interpreted as given in Hartree. The `tolerance` parameter itself should, however, be given without any unit. The `tolerance` parameter is the maximally tolerated difference between an element of the Hamiltonian in two successive self-consistent iterations.
- In order to achieve a better `accuracy` than the one corresponding to the choice of default values, it is often a good idea to decrease the tolerance by one or two orders of magnitude. Often, but not always, this will only add a few extra iterations to the calculation, since the Pulay mixing algorithm usually performs very well once it approaches equilibrium. If only a fast rough estimation of the results is needed the accuracy can be lowered by increasing the tolerance value.

IterativeDiagonalizationSolver

class IterativeDiagonalizationSolver(*bands_around_fermi_level=None*, *target_fermi_level=None*, *subspace_dimension_factor=None*, *interior_eigenvalue_strategy=None*, *max_iterations=None*)

Constructor for the iterative subspace diagonalizaton solver. This solver uses a Krylov-Schur algorithm as implemented in SLEPc and it allows to calculate interior eigenvalues around a target value.

Parameters

- **bands_around_fermi_level** (*Positive int*) – The number of bands around the Fermi level per principal spin channel. **Default:** `12`
- **target_fermi_level** (*PhysicalQuantity* of type energy | list of 2 *PhysicalQuantity* of type energy. **Default:** `None`) – The energy around which the eigenvalues are searched. Typically, you want to use the corresponding intrinsic bulk Fermi level. A user-specified target fermi level can allow in certain cases to avoid an expensive update on the configuration. For Polarized calculations, a value per spin channel can be specified. **Default:** The fermi level from the updated configuration.
- **subspace_dimension_factor** (*float*) – The subspace dimension used for the iterative solver, as a multiple of *bands_around_fermi_level*. A subspace larger than *bands_around_fermi_level* can be required to find degenerate eigenvalues or increase convergence for the lowest eigenvalues. It must be at least `1`. **Default:** `4.0`
- **interior_eigenvalue_strategy** (`HarmonicExtraction` | `ShiftAndInvert`) – Flag to select the strategy used for targeting interior eigenvalues. `ShiftAndInvert` instructs the solver to perform a shift-and-invert spectral transformation to enhance convergence of Eigenvalues closest to the target Fermi level. `HarmonicExtraction` is an extraction technique which allows to avoid an explicit shift-and-invert operation, hence being significantly more efficient. **Default:** `ShiftAndInvert`
- **max_iterations** (*int*) – The max number of iterations of the iterative diagonalization solver. It must be greater than 0. **Default:** `1000`

bandsAroundFermiLevel()

Returns
Specified number of bands that was determined around the Fermi level.

Return type
Non-negative int

interiorEigenvalueStrategy()

Returns
The interior eigenvalue strategy.

Return type
HarmonicExtraction | ShiftAndInvert

maxIterations()

Returns
The max number of iterations.

Return type
Positive int

subspaceDimensionFactor()

Returns
The subspace dimension factor.

Return type
float

targetFermiLevel()

Returns
The target/requested Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

Note: When dealing with larger systems using an orthogonal basis set, setting interior_eigenvalue_strategy to HarmonicExtraction might be a significantly faster and less memory-demanding option. However, HarmonicExtraction can be less robust compared to ShiftAndInvert.

KDotPExpansion1D

class KDotPExpansion1D(*updates_per_segment=None*, *number_of_additional_bands=None*)

Class for representing parameters for a k.p expansion along a path in k-space (Bandstructure).

Parameters

- **updates_per_segment** (*int*) – The number of exact calculations performed for each bandstructure segment. Must be larger than one. **Default:** 3
- **number_of_additional_bands** (*int*) – The number of additional bands included in the exact updates. Must be a non-negative integer. **Default:** 10

numberOfAdditionalBands()

Returns
The number of exact updates per bandstructure segment.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

updatesPerSegment()

Returns
The number of exact updates per bandstructure segment.

Return type
int

KDotPExpansion3D

class KDotPExpansion3D(*k_point_sampling=None*, *number_of_additional_bands=0*, *use_correction_term=None*)

Class for representing parameters for a k.p expansion from a 3D k-grid. Used in bandstructure and density of states.

Parameters

- **k_point_sampling** (sequence (size 3) of int | *MonkhorstPackGrid*) – The k-point grid supporting the interpolation. For the irreducible k-points of this grid the exact eigensolutions are generated. Must be a fully 3D grid, meaning that there are more than 1 k-point in each direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **number_of_additional_bands** (*int*) – The number of additional bands included in the exact updates. The expansion becomes more accurate the more bands are included. Must be a non-negative integer. The default uses no additional bands, i.e. it just uses the bands of the SCF calculation. **Default:** 0
- **use_correction_term** (*bool*) – If set to `True`, the k.p Hamiltonian includes the correction term discussed in Enabling accurate first-principle calculations of electronic properties with a corrected k ů p scheme by Kristian Berland and Clas Persson. **Default:** True

kPointSampling()

Returns
The k-point grid to use.

Return type
`MonkhorstPackgrid`

numberOfAdditionalBands()

Returns
The number of exact updates per bandstructure segment.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

useCorrectionTerm()

Returns
The flag indicating whether to use the 3D correction term.

Return type
bool

useGroundStateQuantities()

Returns
The flag indicating whether the ground state grid, bands and wave functions are to be used.

Return type
bool

Kerker

class Kerker(*energy_q0=None*, *energy_qmax=None*, *maximum_damping=None*)
Class for representing Kerker style preconditioner

Parameters

- **energy_q0** (*PhysicalQuantity* of type energy) – The Kerker `q0` parameter, given in energy units. i.e *0.5 * q0 * q0`*. **Default:** `0.01 * Hartree`
- **energy_qmax** (*PhysicalQuantity* of type energy) – The maximum wave number which is preconditioned, given in energy units, i.e `0.5 * qmax * qmax`. NOTE: For Plane wave calculation this is ignored. **Default:** `0.25 * Hartree`
- **maximum_damping** (*float*) – The damping of the `q=0` component. This should be between 0 and 1. **Default:** `0.01`

energyQ0()

Returns
The Kerker `q0` parameter, given in energy units. i.e `0.5 * q0 * q0`.

Return type
PhysicalQuantity of type energy

energyQmax()

Returns
The maximum wave number which is preconditioned, given in energy units, i.e `0.5 * qmax * qmax`.

Return type
PhysicalQuantity of type energy

maximumDamping()

Returns
The damping of the `q=0` component.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Use a *Kerker* preconditioner.

```
iteration_control_parameters = IterationControlParameters(
preconditioner=Kerker(energy_q0=0.01*Hartree, energy_qmax=0.25*Hartree, maximum_
↪damping=0.01),
)
```

Notes

•The *Kerker* preconditioner damps Fourier components of the electron density according to the scheme

$$\begin{cases} f \times \frac{q^2 + \frac{f_0}{f} q_0^2}{q^2 + q_0^2} & \text{if } q \leq q_{max} \\ f & \text{if } q > q_{max}. \end{cases}$$

In this equation, q_{max} , q_0 , and f_0 are specified through the *Kerker* object parameters `energy_qmax`, `energy_q0`, and `maximum_damping`. The parameter `f` is specified through the *IterationControlParameters* variable `damping_factor`.

•The default parameters are appropriate for `HamiltonianVariable` mixing.

KineticEnergyDistribution

class KineticEnergyDistribution(*md_trajectory*, *start_time=None*, *end_time=None*, *maximum_energy=None*, *resolution=None*, *atom_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the kinetic energy distribution of an MD simulation.

Parameters

•**md_trajectory** (*MDTrajectory* | `AtomicConfiguration`) – The MDTrajectory or configuration to calculate the kinetic energy distribution for.

•**start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`

•**end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** the last time frame

•**maximum_energy** (*PhysicalQuantity* of type energy) – Upper limit on energy values considered for analysis. **Default:** The maximum kinetic energy of the simulation.

•**resolution** (*PhysicalQuantity* of type energy) – The bin size, which determines the resolution of the histogram. **Default:** `0.01 * eV`

•**atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** all elements

•**time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the kinetic energy distribution histogram.

energies()

Return the energies associated with the histogram.

Usage Examples

Load an *MDTrajectory* and calculate the *KineticEnergyDistribution* considering all aluminum atoms:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

ke_distribution = KineticEnergyDistribution(md_trajectory,
                                            start_time=10000.0*fs,
                                            end_time=50000.0*fs,
                                            maximum_energy=1.0*eV,
                                            resolution=0.005*eV,
                                            atom_selection=Aluminum)

# Get the energies and the histogram of the kinetic energy distribution.
energies  = ke_distribution.energies().inUnitsOf(eV)
histogram = ke_distribution.data()

# Plot the data using pylab.
import pylab

pylab.bar(energies, histogram,
          label='Kinetic energies of aluminum',
          width=0.005)
pylab.xlabel('Kinetic energy (eV)')
pylab.ylabel('Histogram')
pylab.legend()

pylab.show()
```

kinetic_energy_distribution.py

Notes

By default, all elements are taken into account.

The `atom_selection` parameter accepts an element, a tag name, or a list of indices to select atoms for the velocity distribution. This can be useful, e.g. in the presence of constraints as constrained atoms should be excluded in this analysis.

KineticMonteCarlo

class KineticMonteCarlo(*rate_matrix*, *initial_state=0*, *superbasin_threshold=None*, *rng=None*)

Construct a kinetic Monte Carlo simulation.

Parameters

- **rate_matrix** (*PhysicalQuantity* of type frequency) – A right handed (rows sum to 1) rate matrix. This must be a square matrix.
- **initial_state** (*int*) – The initial state of the KMC simulation. This corresponds to a row index in the `rate_matrix`. **Default:** `0`
- **superbasin_threshold** (*int*) – The number of steps between two states before they are merged together and treated with the coarse graining algorithm. A value of `None` disables coarse graining. **Default:** `None`
- **rng** (*numpy.random.mtrand.RandomState*) – The random number generator used to generate the random number stream. If None, a new random number generator stream will be made with a seed obtained from the operating systems entropy pool. **Default:** `None`

findSuperbasin(*state*)

Find the superbasin that contains the state.

Parameters
state (*int*) – The state to find the containing superbasin of.

Returns
Superbasin if the state is in a superbasin, otherwise None.

Return type
Superbasin | None

getTransitionCount(*state_1*, *state_2*)

Parameters

- **state_1** (*int*) – The first of the two states.
- **state_2** (*int*) – The second of the two states.

Returns
The transition count between two states.

Return type
int

mergeStates(*state_1*, *state_2*)

Merge two states together. If either or both states belong to a superbasin, the superbasins are merged instead.

Parameters

- **state_1** (*int*) – The first of the two states to merge.
- **state_2** (*int*) – The second of the two states to merge.

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print out the trajectory of the KMC simulation.

Parameters
stream (*A stream that supports strings being written to using 'write'.*) – The stream to write the trajectory to. **Default:** sys.stdout

numberOfSteps()

Returns
The total number of steps taken in this KMC simulation.

Return type
int

setRateMatrix(*rate_matrix*)

Set a new rate matrix.

Parameters
rate_matrix (*PhysicalQuantity* of type frequency) – A right-handed (rows sum to 1) rate matrix. This must be a square matrix.

setSuperbasinThreshold(*superbasin_threshold*)

Sets the superbasin threshold.

Parameters
superbasin_threshold (*int*) – The number of steps between two states before they are merged together and treated with the coarse graining algorithm. A value of None disables coarse graining. **Default:** None

state()

Returns
The current state of the Markov chain.

Return type
int

step()

Take a single KMC step.

Returns
The escape time from the current state.

Return type
PhysicalQuantity of type frequency

superbasinThreshold()

Returns
Return the superbasin threshold.

Return type
int

superbasins()

Returns
The list of superbasins.

Return type
list of type Superbasin

time()

Returns
The total time that has elapsed during the KMC simulation.

Return type
PhysicalQuantity of type time

times()

Returns
The times associated with each state in the trajectory.

Return type
PhysicalQuantity of type time

trajectory()

Returns
The KMC trajectory. This is a list of integer state numbers (row index in the rate matrix).

Return type
list of type int

uniqueString()

Return a unique string representing the state of the object.

updateTransitionCount(*state_1*, *state_2*)

Update the transition count between two states. This method also merges states together when the transition count equals the superbasin threshold parameter.

Parameters

- **state_1** (*int*) – The first of the two states.
- **state_2** (*int*) – The second of the two states.

Usage Examples

Setup a simple KMC simulation with three states and print the results.

```
# Define a rate matrix. The ij entries describe the reaction rate from state i to state␣
↪j.
rate_matrix = [
    [ 0.0, 1e9, 0.0 ],
    [ 1e8, 0.0, 1e7 ],
    [ 0.0, 1e6, 0.0 ],
]/Second

kmc = KineticMonteCarlo(
    rate_matrix,
    initial_state=0,
    )

for i in range(1000):
    kmc.step()

nlprint(kmc)
```

kmc.py

Notes

KineticMonteCarlo objects are normally passed to the *AdaptiveKineticMonteCarlo* class.

This class can also be used to setup a KMC simulation manually. The only input that is required is the matrix of reaction rates. No configurations are used. It is up to the user to give meaning to each of states in the KMC trajectory.

The treatment of superbasins using the Monte Carlo with absorbing Markov chains (MCAMC) algorithm is described in refs[1,2].

KpointDensity

class KpointDensity(*density_a=None*, *density_b=None*, *density_c=None*, *symmetries=None*, *force_timereversal=None*, *shift_to_gamma=None*)

Parameters

- **density_a** (*PhysicalQuantity* of unit length) – k-point density along the A direction. A value of 0 ensures that a single grid point will always be used for this direction (i.e., Γ). **Default:** `0.0 Angstrom`
- **density_b** (*PhysicalQuantity* of unit length) – k-point density along the B direction. A value of 0 ensures that a single grid point will always be used for this direction (i.e., Γ). **Default:** The same as `density_a`.
- **density_c** (*PhysicalQuantity* of unit length) – k-point density along the C direction. A value of 0 ensures that a single grid point will always be used for this direction (i.e., Γ). **Default:** The same as `density_a`.
- **symmetries** (*sequence of tuple of (array (size 3, 3), array (size 3))*) – A list of user-defined symmetries used for the reduction of the k-points in integration. **Default:** The relevant symmetries will be determined from the structure.
- **force_timereversal** (*bool*) – Always enforce time-reversal symmetry. **Default:** `True` (`False` with spin-orbit or non-collinear spin)
- **shift_to_gamma** (*bool | sequence (size 3) of bool*) – Shift grid so that Γ (0.0, 0.0, 0.0) is included along all or certain directions. **Default:** *True* `

densityA()

Returns
k-point density along the A direction.

Return type
PhysicalQuantity of unit length

densityB()

Returns
k-point density along the B direction.

Return type
PhysicalQuantity of unit length

densityC()

Returns
k-point density along the C direction.

[1] M.ăA. Novotny. Monte Carlo Algorithms with Absorbing Markov Chains: Fast Local Algorithms for Slow Dynamics. *Phys. Rev. Lett.*, 74:1–5, Jan 1995. doi:10.1103/PhysRevLett.74.1.

[2] S.ăT. Chill, M.ăWelborn, R.ăTerrell, L.ăZhang, J.-C. Berthet, A.ăPedersen, H.ăJónsson, and G.ăHenkelman. EON: Software for long time simulations of atomic scale systems. *Modelling Simul. Mater. Sci. Eng.*, 22(5):055002, 2014. doi:10.1088/0965-0393/22/5/055002.

Return type
PhysicalQuantity of unit length

`forceTimereversal()`

Returns
Whether time-reversal symmetry is always enforced.

Return type
bool

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`shiftToGamma()`

Returns
Whether the k-points are shifted so that Γ is included along each direction.

Return type
list (size 3) of bool

`symmetries()`

Returns
The symmetries used for the reduction of the k-points in integration.

Return type
list of tuple of (array (size 3, 3), array (size 3))

`uniqueString()`

Return a unique string representing the state of the object.

KrylovSelfEnergy

`class KrylovSelfEnergy(`*save_self_energies=None*, *lambda_min=None*, *sparse_threshold=None*, *storage_strategy=None*`)`

Self-energy calculator based on the Krylov method. The Krylov scheme uses an iterative approach for getting a subset of all the electrode modes.

Parameters

- **storage_strategy** (*SaveInMemory* | *StoreOnDisk* | *NoStorage*) – The way self energies are stored between iterations. **Default:** *SaveInMemory()*
- **lambda_min** (*float > 0*) – The smallest logarithmic decrement of a mode to take into account. **Default:** `0.1`
- **sparse_threshold** (*float > 0*) – Self energies (in Hartree) smaller than this value treated as 0 when storing as sparse matrix. **Default:** `1e-12`
- **save_self_energies** – **Deprecated:** from v2017.0, use `storage_strategy` instead.

`lambdaMin()`

Returns
The smallest logarithmic decrement of a mode

Return type
float

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`saveSelfEnergies()`

Deprecated: from v2017.0

`sparseThreshold()`

Returns
Values treated as 0.

Return type
float

`storageStrategy()`

Returns
The way self energies are stored between iterations.

Return type
SaveInMemory | *StoreOnDisk* | *NoStorage*

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Define that the self energy on the real contour is calculated with the Krylov method. Only modes with a decay slower than 0.05 are included.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_real=KrylovSelfEnergy(lambda_min=0.05),
)
```

Examples on how to use the `storage_strategy` parameter can be found in the *Usage Examples* of *RecursionSelfEnergy*.

Notes

KrylovSelfEnergy uses the iterative Krylov subspace method by H. H. Sørensen *et. al.*[1],[2] for calculating the self energy matrix. The `lambda_min` parameter determines the size of the Krylov subspace. In the limit `lambda_min` goes to zero, the full space will be included.

[1] H.ăH.ăB. Sorensen, P.ăC. Hansen, D.ăE. Petersen, S.ăSkelboe, and K.ăStokbro. Krylov subspace method for evaluating the self-energy matrices in electron transport calculations. *Phys. Rev. B*, 77:155301, Apr 2008. doi:10.1103/PhysRevB.77.155301.

[2] H.ăH.ăB. Sorensen, P.ăC. Hansen, D.ăE. Petersen, S.ăSkelboe, and K.ăStokbro. Efficient wave-function matching approach for quantum transport calculations. *Phys. Rev. B*, 79:205322, May 2009. doi:10.1103/PhysRevB.79.205322.

LBFGS

class LBFGS(*memory_size=20*, *finite_difference=PhysicalQuantity(0.01, Ang)*, *line_search=True*)

Constructor for the L-BFGS optimizer method.

Parameters

- **memory_size** (*int*) – The maximum number of previous steps that will be stored in memory in order to approximate the Hessian matrix. **Default:** 20
- **finite_difference** (*PhysicalQuantity* with length units) – The finite different step length used to approximate the curvature along the initial descent direction. **Default:** 0.01*Angstrom
- **line_search** (*bool*) – Controls if a line search is used. **Default:** True if the objective function supports line searches

scfInitialGuess()

Returns
The SCF initial guess used for the LBFGS optimizer; always the previous configuration.

Return type
UsePreviousConfiguration

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Optimize the geometry of a water molecule using the LBFGS algorithm.

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,  -1.70000000e-05,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Define a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# Perform optimization using the LBFGS algorithm.
OptimizeGeometry(molecule_configuration, optimizer_method=LBFGS())
```

lbfgs.py

Notes

LBFGS is the recommended optimizer to use in QuantumATK. It should have superior performance to the FIRE optimizer for nearly every optimization problem.

While there is not an official steepest descent optimizer in QuantumATK, the LBFGS optimizer can be used as a steepest descent optimizer if the `memory_size` is set to zero.

Algorithm Details

The value of the objective function is not explicitly minimized (unless a line search is used), instead, the optimizer minimizes the forces in the system to find a stationary point. This is important because it allows for geometry optimizations, minimizations of nudged elastic bands (using the projected forces), and saddle point optimization (using min-mode projected forces) to be performed from one general optimizer.

When the forces are minimized directly (e.g. in a nudged elastic band optimization), it is not possible to use a line-search algorithm to determine the step size. Instead the curvature along the current step direction is estimated and the step length is chosen to minimize a one-dimensional quadratic system.

Without the aid of a line-search, special care must be taken to ensure that the approximate Hessian remains positive definite (all positive eigenvalues) or the optimization will not converge to a minimum. If a step is taken that would include a negative curvature into the memory, the entire history is reset and a gradient descent step is taken along the force.

The memory is also reset in the case that a step size larger than the maximum step length is proposed. This criterion indicates that the current coordinates are far away from a stationary point. Due to this, the algorithm behaves as the gradient descent method in the non-quadratic region of the potential energy surface (i.e. far from stationary points). This is the ideal behavior because attempting to minimize the approximate quadratic problem when the underlying true Hessian matrix is rapidly changing gives poor results.

Some objective functions support line searches. When possible, a back-tracking line search will be used to ensure reduction in the function value at each step. The step size typically results in a reduction in the function value. So it is only when a bad step is taken (i.e. one that increases the function value) that the back-tracking line search is used. There is a hard maximum on the number of steps the line search algorithm will take to prevent getting stuck. In the case that a decrease in function value can not be found the line search will give up and take a steepest descent step and reset the L-BFGS memory.

Details about this algorithm can be found in[1].

LCAOCalculator

class LCAOCalculator(*basis_set=None*, *exchange_correlation=None*, *charge=None*, *fixed_spin_moment=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *poisson_solver=None*, *checkpoint_handler=None*, *dynamical_matrix_parameters=None*, *correction_extension=None*, *algorithm_parameters=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*, *solvation_parameters=None*, *use_atom_pair_matrix=None*)

Class for representing calculations using the ATK-DFT numerical orbital model for configurations of the type MoleculeConfiguration and BulkConfiguration.

Parameters

[1] D.ăC. Liu and J.ăNocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989. doi:10.1007/BF01589116.

- **basis_set** (list | tuple | *BasisSet* | PAWBasisSet) – An object describing the basis set used for the LCAO calculation. This keyword can be given in three ways.

1. As a single *BasisSet* | PAWBasisSet object.
2. As a tuple (string, BasisSet). The string must match a tag on the atom in order to be applied.
3. As a sequence of entries matching either of the above formats

  Default: BasisGGAPseudoDojo.Medium

- **exchange_correlation** (*ExchangeCorrelation*) – The exchange correlation functional for this calculation. **Default:** GGA.PBE
- **charge** (*float*) – The charge of the system, a charge of -1 corresponds to one extra electron. **Default:** 0.0
- **fixed_spin_moment** (float | False) – Total spin moment (per unit cell) to use, defined as $\Delta N = N_\uparrow - N_\downarrow$, where $N_\uparrow$ and $N_\downarrow$ are the number of electrons in the Up and Down spin channels, respectively. When specified the spin moment will be fixed at the given value by introducing separate Fermi levels for the Up and Down spin channels. Can only be specified when doing a calculation with polarized spin. If set to False the spin moment will not be fixed - a single Fermi level is used. **Default:** False
- **numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The numerical accuracy parameters used for the DFT calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=75.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 1),
    radial_step_size=0.001*Bohr,
    density_cutoff=1.0e-6,
    interaction_max_range=20.0*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(1000.0*Kelvin))
```

  Note that the density mesh cutoff will be altered depending on the elements present in the configuration.

- **iteration_control_parameters** (*IterationControlParameters* | NonSelfconsistent) – The iteration control parameters used for the DFT calculation. For non-self-consistent calculations set this parameter to NonSelfconsistent. **Default:**

```
IterationControlParameters(
    tolerance=1.0e-4,
    max_steps=100,
    algorithm=PulayMixer(restart_strategy=AdaptiveHistoryRestart()),
    damping_factor=0.1,
    number_of_history_steps=20,
    start_mixing_after_step=0,
    mixing_variable=HamiltonianVariable,
    linear_dependence_threshold=0.0,
    preconditioner=Preconditioner.Off)
```

- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourierSolver* for a *BulkConfiguration* without any metallic or dielectric SpatialRegion.
- **checkpoint_handler** (*CheckpointHandler*) – The CheckpointHandler used for specifying the save-file and the time interval between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.

•**dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.

•**correction_extension** (*GrimmeDFTD2* | *GrimmeDFTD3*) – The correction extension to used, when calculating energy, forces and stress. **Default:** None.

•**algorithm_parameters** (*AlgorithmParameters*) – The algorithm parameters used for calculating the density matrix. **Default:**

```
AlgorithmParameters(
    density_predictor=False,
    density_matrix_method=DiagonalizationSolver(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    store_energy_density_matrix=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    use_symmetries=False)
```

•**hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.

•**parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:**

```
ParallelParameters(
    processes_per_neb_image=None,
    processes_per_individual=None,
    processes_per_bias_point=None,
    processes_per_saddle_search=1)
```

•**solvation_parameters** (None | *CosmoSolvationParameters*) – Optional parameters for a COSMO solvation model. If this is specified, the COSMO solvation model contribution will be included in the SCF loop.

•**use_atom_pair_matrix** (*bool*) – If this flag is set to true, the density matrix/hamiltonian and overlap matrices will be stored in an atom-pair matrix format. **Default:** `False`

algorithmParameters()

Returns
The algorithm parameters.

Return type
AlgorithmParameters.

basisSet()

Returns
The basis set.

Return type
list

charge()

Returns
The charge of the system.

Return type
float

checkpointHandler()

Returns
The checkpoint handler.

Return type
CheckpointHandler

correctionExtension()

Returns
The correction extension or None if not set.

Return type
GrimmeDFTD2 | *GrimmeDFTD3* | None

dynamicalMatrixParameters()

Return the DynamicalMatrixParameters. Deprecated.

Returns
None since the class DynamicalMatrixParameters is deprecated.

Return type
None

exchangeCorrelation()

Returns
The exchange-correlation.

Return type
ExchangeCorrelation

fixedSpinMoment()

Get the fixed spin moment.

Returns
The fixed spin moment or `False` if the spin moment is not held fixed.

Return type
float | `False`

hamiltonianDerivativesParameters()

Return the HamiltonianDerivativesParameters. Deprecated.

Returns
None since the class HamiltonianDerivativesParameters is deprecated.

Return type
None

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The iteration control parameters.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numericalAccuracyParameters()

Returns
The numerical accuracy parameters.

Return type
NumericalAccuracyParameters

parallelParameters()

Returns
The parallel parameters object.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver set on the calculator.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setBasisSet(*basis_set*)

Set the basis set.

Parameters
basis_set (*list*) – The basis set to use. The basis set is given as a sequence of *BasisSet* objects for each element in the configuration.

setCheckpointHandler(*checkpoint_handler*)

Set the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The checkpoint handler to use.

setCorrectionExtension(*correction_extension*)

Set the iteration correction extension.

Parameters
correction_extension (*GrimmeDFTD2* | *GrimmeDFTD3* | None) – The correction extension to use.

setExchangeCorrelation(*exchange_correlation*)

Set the exchange-correlation.

Parameters
exchange_correlation (*ExchangeCorrelation*) – The exchange-correlation to use.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The iteration control parameters to use.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The numerical accuracy parameters to use.

setParallelParameters(*parallel_parameters*)

Method for setting the parallel parameters.

Parameters
parallel_parameters (*ParallelParameters* | None) – The parallel parameters to set. If None a default version of the parameters is used.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver to set on the calculator.

setSolvationParameters(*solvation_parameters*)

Set the solvation parameters.

Parameters
solvation_parameters (*CosmoSolvationParameters*) – The solvent parameters to use.

solvationParameters()

Returns
The solvation parameters.

Return type
CosmoSolvationParameters

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible. @private

upgradeADMMBasisSet(*configuration*)

Make sure the ADMM basis set is compatible with the regular DFT basis set, if set.

useAtomPairMatrix()

Returns
If true, the density matrix/hamiltonian and overlap matrices will be stored in an atom-pair matrix format.

Return type
bool

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Usage Examples

Define an *LCAOCalculator* with user defined basis set:

```
basis_set = [
    LDABasis.Hydrogen_DoubleZetaPolarized,
    LDABasis.Oxygen_DoubleZetaPolarized,
    LDABasis.Nitrogen_DoubleZeta,
    LDABasis.Carbon_SingleZeta,
]
calculator = LCAOCalculator(basis_set=basis_set)
```

Restart an LCAO calculation using the self-consistent state from a previous calculation:

```
# Read in the BulkConfiguration with the old scf state
old_calculation=nlread("filename.nc",BulkConfiguration)[0]

# Define the BulkConfiguration with similar number of atoms
new_calculation=BulkConfiguration(...)

# extract the old calculator
old_calculator = old_calculation.calculator()

# make a clone of the old calculator
new_calculator = old_calculator()

# Attach the calculator and use the old initial state
new_calculation.setCalculator(new_calculator, initial_state=old_calculation)
```

LanczosSaddleSearch

class LanczosSaddleSearch(*constraints=None*, *initial_direction_generator=None*, *index_selector=None*, *adaptive_displacement=False*, *active_sphere=None*, *min_move_distance=None*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *optimizer_method=None*, *lanczos_relative_error=None*, *lanczos_maximum_iterations=None*, *finite_difference=None*, *finite_difference_method=None*, *reorthogonalize=None*)

Class for performing a saddle search using the Lanczos algorithm.

Parameters

- **constraints** (*list of ints*) – The atoms that should be constrained during the process search. **Default:** []
- **initial_direction_generator** (*DirectionGenerator*) – The generator for the initial displacements. **Default:** RandomDirection
- **index_selector** (*callable*) – A callable to select the indices to displace. Takes as input the current configuration and returns the indices to displace.
- **adaptive_displacement** (*bool*) – When using adaptive displacement the initial atom displacement magnitude will be set dynamically. The chosen magnitude will be 0.8 multiplied with the average distance between reactant and saddle point. This will overewrite the value set in by the initial_direction_generator.
- **active_sphere** (*PhysicalQuantity* of type length) – The radius of the sphere that contains all atoms that have moved.
- **min_move_distance** (*PhysicalQuantity* of type length) – The minimum distance that the selected indices must have moved between reactant and saddle. All saddle configurations that do not meet this criteria will be discarded.
- **max_forces** (*PhysicalQuantity* of type force) – The maximum forces allowed in the orthogonal optimization. **Default:** 0.01*eV/Angstrom
- **max_steps** (*int*) – The maximum number of uphill steps and orthogonal steps. **Default:** 200
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length for the uphill steps and orthogonal steps. **Default:** 0.1*Angstrom
- **optimizer_method** (list of OptimizerMethod) – The optimizer method to use. **Default:** LBFGS
- **lanczos_relative_error** (*float*) – The relative error for the Lanczos algorithm. **Default:** 0.0001
- **lanczos_maximum_iterations** (*int*) – The maximum number of iterations for the Lanczos algorithm. **Default:** 20
- **finite_difference** (*PhysicalQuantity* of type length) – The step size for the finite difference estimation of the second derivatives. **Default:** 0.01*Angstrom
- **finite_difference_method** (*Forward | Central*) – The type of finite difference used to estimate the second derivatives. **Default:** Forward

reset()

Reset the state of the saddle search.

run(*reactant_configuration*)

Run the saddle search algorithm.

Parameters
reactant_configuration (*AtomicConfiguration*) – The configuration to start the search from.

Returns
The data from the saddle search.

Return type
SaddleSearchData

setRandomSeed(*seed*)

Set the random seed for the initial direction generator.

Parameters
seed (*int*) – The random seed.

Notes

LanczosSaddleSearch is one of the three saddle search methods available for use in *AdaptiveKineticMonteCarlo*. The *LanczosSaddleSearch* method uses the Lanczos algorithm to determine the minimum mode towards the saddle point.

LanczosSaddleSearch requires an initial direction which is used to initiate the saddle search, controlled by the `initial_direction_generator` argument. The direction generator should be a callable that returns a vector. This callable will be used to generate a configuration from which the saddle search is started. There are two pre-made direction generators available. The *RandomDirection* generates a random direction, while the *HypersphereDirection* generates a radial vector on a hypersphere of given radius. Each time a new saddle search is initiated, a new initial direction is generated.

In addition to the initial direction, one can specify which atom(s) to displace. This is done using the `index_selector` argument. This argument can be either a list of indices or a (picklable) callable.

When a saddle point has been found, an object of type `SaddleSearchData`. This object, most importantly, contains the found saddle point as well as the lowest eigenvector, of the Hessian matrix at the saddle point.

Langevin

class Langevin(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *heating_rate=None*, *friction=None*, *random_seed=None*)

NVT Langevin integrator class which implements the Langevin thermostat.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1.0 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. The temperature can be given as a single temperature value for the entire system, or as a list of 2-tuples of str and *PhysicalQuantity* of type temperature, applying local thermostats to the tagged groups of atoms. E.g. `[('group1', 280.0 * Kelvin), ('group2', 320.0 * Kelvin)]`. **Default:** `300.0 * Kelvin`
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `0.0 * Kelvin s**-1`
- **friction** (*PhysicalQuantity* of type 1/time) – The friction constant in the Langevin equation. **Default:** `0.01 * fs**-1`
- **random_seed** (*int*) – The seed for the random generator providing the stochastic forces. Must be between 0 and 2**32. **Default:** The default random seed

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the Langevin thermostat:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up MD method
method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    friction=0.01*femtoSecond**-1,
    heating_rate=0*Kelvin/picoSecond,
    initial_velocity=None
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
    method=method
)
```

Notes

•The Langevin thermostat can be used to simulate a canonical NVT-ensemble, by integrating the Langevin equation

$$m_i \frac{d^2\mathbf{r_i}}{dt^2} = -m_i\gamma\mathbf{v_i} + \mathbf{F_i} + \mathbf{\Gamma_i}\,,$$

which adds a friction term with a friction constant γ and a stochastic random force Γ to Newtons equations of motion. QuantumATK implements the algorithm proposed by Goga et al[1].

•In contrast to other NVT methods, such as *NVTNoseHoover* or *NVTBerendsen*, the dynamics of the system can be altered significantly by the Langevin thermostat, and the evolution of the system is no longer deterministic. It should therefore primarily be used to sample configurations from the canonical ensemble, rather than to calculate dynamical properties.

•You can specify one or more thermostats acting on tagged sub-groups of atoms of the configuration, by giving a list of `(tag-name, temperature)`-tuples instead of a single, global `reservoir_temperature` value.

•If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

•Setting the `reservoir_temperature` to 0K, will switch off the stochastic force, resulting in *damped dynamics*, which can be used to coarsely relax a configuration.

LennardJonesMNPotential

class LennardJonesMNPotential(*particleType1*, *particleType2*, *A*, *B*, *m*, *n*, *r_cut=None*, *ASpecial=None*, *BSpecial=None*, *bonded_mode=None*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**A** (*PhysicalQuantity of the length**m * energy*) – Potential parameter.

•**B** (*PhysicalQuantity of the length**n * energy*) – Potential parameter.

•**m** (*float*) – Potential parameter.

•**n** (*float*) – Potential parameter.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

•**ASpecial** (*PhysicalQuantity of type length**m * energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**BSpecial** (*PhysicalQuantity of type length**n * energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**bonded_mode** – Either LennardJonesMNPotential.evaluateAll or LennardJonesMNPotential.evaluate34.

If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

If evaluate34 is chosen, interactions between particles that are connected by bonds with a path length of at most two (i.e. are connected by a bond or are bonded to a common atom) are omitted. Interactions between particles that are connected

[1] N.ăGoga, A.ăJ. Rzepiela, A.ăH. deăVries, S.ăJ. Marrink, and H.ăJ.ăC. Berendsen. Efficient Algorithms for Langevin and DPD Dynamics. *Journal of Chemical Theory and Computation*, 8(10):3637–3649, 2012. PMID: 26593009. doi:10.1021/ct3000876.

by a bonded path of length three use the sigmaSpecial and epsilonSpecial parameters, while all other interactions use the normal sigma and epsilon parameters.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for argon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Argon crystal
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.316, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.316, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.316]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Argon, Argon, Argon, Argon]

# Define coordinates
fractional_coordinates = [[ 0.0,  0.0,  0.0],
                          [ 0.5,  0.5,  0.0],
                          [ 0.5,  0.0,  0.5],
                          [ 0.0,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# Create a new potential set for argon
potentialSet = TremoloXPotentialSet(name='Argon_COMPASS')

# Add the particle type, containing the Lennard-Jones parameters, to the potential set
potentialSet.addParticleType(ParticleType(symbol='Ar', mass=39.948*atomic_mass_unit))

# Add the Lennard-Jones potential to the potential set
potentialSet.addPotential(LennardJonesMNPotential('Ar', 'Ar',
                                                  A=3456.895*eV*Angstrom**9.0,
                                                  B=88.773*eV*Angstrom**6.0,
                                                  m=9.0,
                                                  n=6.0,
                                                  r_cut=9.0*Angstrom))

# Create a TremoloX calculator from the potential set
calculator=TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)
```

The Lennard Jones parameters of this example are taken from the **COMPASS** force field.

Notes

The *LennardJonesMNPotential* potential typically describes a combination of van der Waals and repulsive interactions.

To describe the interaction between two particles i and j, this potential uses the following potential function:

$$V_{ij}(r) = \frac{A_{ij}}{r^m} - \frac{B_{ij}}{r^n} .$$

This form permits a more flexible definition of a Lennard-Jones potential. You have to manually specify the interaction parameters between all pairs of particle types, and therefore you can use any set of combination rules you want.

Lennard-Jones parameters can be taken from various force fields, such as the **Universal Force Field (UFF)**, the **Generalized AMBER Force Field (GAFF)**, the **Transferable Potentials for Phase Equilibria (TraPPE)** force field, or, as in the above example, the **COMPASS** force field.

The best choice depends on whether the respective force field has been designed for the desired context. Therefore, one should always carefully read the force field publication before using a new potential parameter set and ideally compare some test results against experiments or suitable quantum mechanical calculations.

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected via less than 4 bonds. If `LennardJonesPotential.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `LennardJonesPotential.evaluate34` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one or two consecutive bonds. For atoms separated by exactly three bonds (i.e. atoms 1 and 4 in a bonded sequence), the *special* Lennard-Jones parameters (respectively the combined `epsilon14` and `sigma14` of the *ParticleType*) are used instead.

LennardJonesPotential

class LennardJonesPotential(*particleType1*, *particleType2*, *r_cut=None*, *sigma=None*, *epsilon=None*, *sigmaSpecial=None*, *epsilonSpecial=None*, *bonded_mode=None*, *longRangeCorrection=None*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

•**sigma** (*PhysicalQuantity of type length*) – Potential parameter.

•**epsilon** (*PhysicalQuantity of type energy*) – Potential parameter.

•**sigmaSpecial** (*PhysicalQuantity of type length*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**epsilonSpecial** (*PhysicalQuantity of type energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**bonded_mode** – Either LennardJonesPotential.evaluateAll or LennardJonesPotential.evaluate34.

If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

If evaluate34 is chosen, interactions between particles that are connected by bonds with a path length of at most two (i.e. are connected by a bond or are bonded to a common atom) are omitted. Interactions between particles that are connected by a bonded path of length three use the sigmaSpecial and epsilonSpecial parameters, while all other interactions use the normal sigma and epsilon parameters.

•**longRangeCorrection** (*bool*) – Flag to switch long range correction tail for energy and pressure on/off.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for argon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Argon crystal
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.316, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.316, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.316]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Argon, Argon, Argon, Argon]

# Define coordinates
fractional_coordinates = [[ 0.0,  0.0,  0.0],
                          [ 0.5,  0.5,  0.0],
                          [ 0.5,  0.0,  0.5],
                          [ 0.0,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )


# Create a new potential set for argon
potentialSet = TremoloXPotentialSet(name='Argon_TraPPE')

# Add the particle type, containing the Lennard-Jones parameters, to the potential set
potentialSet.addParticleType(ParticleType(symbol='Ar', mass=39.948*atomic_mass_unit,␣
↪sigma=3.39*Angstrom, epsilon=0.01*eV))

# Add the Lennard-Jones potential to the potential set
potentialSet.addPotential(LennardJonesPotential('Ar', 'Ar', r_cut=9.0*Angstrom))

# Create a TremoloX calculator from the potential set
calculator=TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)
```

The Lennard Jones parameters of this example are taken from the **TraPPE** force field.

Notes

The *LennardJonesPotential* typically describes a combination of van der Waals and repulsive interactions.

To describe the interaction between two particles i and j, the Lennard-Jones potential uses the following potential function:

$$V_{ij}(r) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right].$$

If not specified explicitly in the potential, the Lennard-Jones parameters σ and ϵ are taken from the *ParticleType* the potential acts on. The parameters between different types are calculated based on the Lorentz-Berthelot mixing rules

$$\epsilon_{ij} = \sqrt{\epsilon_i \cdot \epsilon_j}$$

and

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}.$$

Lennard-Jones parameters can be taken from various force fields, such as the **Universal Force Field (UFF)**, the **Generalized AMBER Force Field (GAFF)**, the **COMPASS** force field, or, as in the above example, the **Transferable Potentials for Phase Equilibria (TraPPE)** force field.

Note that some force fields, such as e.g. AMBER, use the slightly different formula $V(r) = \epsilon[(\sigma/r)^{12} - 2(\sigma/r)^{6}]$ to describe Lennard-Jones interactions. In order to use parameters defined for that formula in TremoloXs Lennard-Jones potentials, the σ values taken from these force fields have to be multiplied with $2^{-1/6}$.

The best choice of parameters depends on whether the respective force field has been designed for the desired context. Therefore, one should always carefully read the force field publication before using a new potential parameter set and ideally compare some test results against experiments or suitable quantum mechanical calculations.

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected via less than 4 bonds. If `LennardJonesPotential.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `LennardJonesPotential.evaluate34` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one or two consecutive bonds. For atoms separated by exactly three bonds (i.e. atoms 1 and 4 in a bonded sequence), the *special* Lennard-Jones parameters (respectively the combined `epsilon14` and `sigma14` of the *ParticleType*) are used instead.

LennardJonesShiftPotential

class LennardJonesShiftPotential(*particleType1*, *particleType2*, *r_cut=None*, *r_i=None*, *sigma=None*, *epsilon=None*, *delta=None*, *sigmaSpecial=None*, *epsilonSpecial=None*, *deltaSpecial=None*, *bonded_mode=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius of this potential. Below this distance the potential is capped at a constant force.
- **sigma** (*PhysicalQuantity of type length*) – Potential parameter.

•**epsilon** (*PhysicalQuantity of type energy*) – Potential parameter.

•**delta** (*float*) – Potential parameter.

•**sigmaSpecial** (*PhysicalQuantity of type length*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**epsilonSpecial** (*PhysicalQuantity of type energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**deltaSpecial** (*float*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.

•**bonded_mode** – Either LennardJonesPotential.evaluateAll or LennardJonesPotential.evaluate34.

If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

If evaluate34 is chosen, interactions between particles that are connected by bonds with a path length of at most two (i.e. are connected by a bond or are bonded to a common atom) are omitted. Interactions between particles that are connected by a bonded path of length three use the sigmaSpecial and epsilonSpecial parameters, while all other interactions use the normal sigma and epsilon parameters.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

The shifted Lennard-Jones potential differs from the original LJ-potential by a constant shift δ that is added to the pairwise energies. Moreover, the potential can be modified at short distances $r < r_{inner}$, so that the force is capped at a constant value instead of diverging towards large repulsive forces. This can be used to slowly remove initial overlaps between atoms in MD simulations (see also *ForceCappedEquilibration*). Its functional form was taken from[1] and is

$$U_{ij}(r_{ij}) = \begin{cases} U_{ij}^{shift}(r_{ij}), & r_{ij} > r_{ij}^{inner} \\ (r_{ij} - r_{ij}^{inner}) \frac{\partial U_{ij}^{shift}(r_{ij}^{inner})}{\partial r} + U_{ij}^{shift}(r_{ij}^{inner}), & r_{ij} \leq r_{ij}^{inner} \end{cases}$$

and

$$U_{ij}^{shift}(r_{ij}) = 4 \cdot \epsilon \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} + \delta_{ij} \right]$$

[1] Rolf Auhl, Ralf Everaers, GaryăS. Grest, Kurt Kremer, and StevenăJ. Plimpton. Equilibration of long chain polymer melts in computer simulations. *J. Chem. Phys.*, 119(24):12718–12728, 2003. URL: https://doi.org/10.1063/1.1628670, doi:10.1063/1.1628670.

LennardJonesSpline2Potential

class LennardJonesSpline2Potential(*particleType1=None*, *particleType2=None*, *r_b=None*, *r_e=None*, *r_m=None*, *r_s=None*, *sigma=None*, *epsilon=None*, *sigmaSpecial=None*, *epsilonSpecial=None*, *bonded_mode=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **r_b** (*PhysicalQuantity of type length*) – The outer cutoff radius. Must be larger than r_e.
- **r_e** (*PhysicalQuantity of type length*) – The third cutoff radius. Must be larger than r_m.
- **r_m** (*PhysicalQuantity of type length*) – The second cutoff radius. Must be larger than r_s.
- **r_s** (*PhysicalQuantity of type length*) – The first cutoff radius.
- **sigma** (*PhysicalQuantity of type length*) – Potential parameter.
- **epsilon** (*PhysicalQuantity of type energy*) – Potential parameter.
- **sigmaSpecial** (*PhysicalQuantity of type length*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.
- **epsilonSpecial** (*PhysicalQuantity of type energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.
- **bonded_mode** – Either LennardJonesSpline2Potential.evaluateAll or LennardJonesSpline2Potential.evaluate34.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate34 is chosen, interactions between particles that are connected by bonds with a path length of at most two (i.e. are connected by a bond or are bonded to a common atom) are omitted. Interactions between particles that are connected by a bonded path of length three use the sigmaSpecial and epsilonSpecial parameters, while all other interactions use the normal sigma and epsilon parameters.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

In contrast to the *LennardJonesSplinePotential*, the *LennardJonesSpline2Potential* is zero outside an interval $[r_s, r_b]$, and yields the original Lennard-Jones energy

$$V_{ij}(r) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right].$$

only within an interval $[r_m, r_e]$. Cubic splines are use to ensure a smooth transition to zero over the intervals $[r_s, r_m]$, as well as $[r_e, r_b]$.

If not specified explicitly in the potential, the Lennard-Jones parameters σ and ϵ are taken from the *ParticleType* the potential acts on. The parameters between different types are calculated based on the Lorentz-Berthelot mixing rules

$$\epsilon_{ij} = \sqrt{\epsilon_i \cdot \epsilon_j}$$

and

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}.$$

When using *bonded force fields* the `bonded_mode` parameter can be used to modify how this potential acts between atoms that are connected via less than 4 bonds. If `LennardJonesPotential.evaluateAll` (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If `LennardJonesPotential.evaluate34` (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one or two consecutive bonds. For atoms separated by exactly three bonds (i.e. atoms 1 and 4 in a bonded sequence), the *special* Lennard-Jones parameters (respectively the combined `epsilon14` and `sigma14` of the *ParticleType*) are used instead.

LennardJonesSplinePotential

class LennardJonesSplinePotential(*particleType1*, *particleType2*, *r_cut=None*, *r_i=None*, *sigma=None*, *epsilon=None*, *sigmaSpecial=None*, *epsilonSpecial=None*, *bonded_mode=None*, *longRangeCorrection=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **sigma** (*PhysicalQuantity of type length*) – Potential parameter.
- **epsilon** (*PhysicalQuantity of type energy*) – Potential parameter.
- **sigmaSpecial** (*PhysicalQuantity of type length*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.
- **epsilonSpecial** (*PhysicalQuantity of type energy*) – Potential parameter. This parameter is only used when bonded_mode is set to evaluate34.
- **bonded_mode** – Either LennardJonesSplinePotential.evaluateAll or LennardJonesSplinePotential.evaluate34.

If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

If evaluate34 is chosen, interactions between particles that are connected by bonds with a path length of at most two (i.e. are connected by a bond or are bonded to a common atom) are omitted. Interactions between particles that are connected by a bonded path of length three use the sigmaSpecial and epsilonSpecial parameters, while all other interactions use the normal sigma and epsilon parameters.

- **longRangeCorrection** (*bool*) – Flag to switch long range correction tail for energy and pressure on/off.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for argon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Argon crystal
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.316, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.316, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.316]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Argon, Argon, Argon, Argon]

# Define coordinates
fractional_coordinates = [[ 0.0,  0.0,  0.0],
                          [ 0.5,  0.5,  0.0],
                          [ 0.5,  0.0,  0.5],
                          [ 0.0,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )


# Create a new potential set for argon
potentialSet = TremoloXPotentialSet(name='Argon_TraPPE')

# Add the particle type, containing the Lennard-Jones parameters, to the potential set
potentialSet.addParticleType(ParticleType(symbol='Ar', mass=39.948*atomic_mass_unit,␣
↪sigma=3.39*Angstrom, epsilon=0.01*eV))

# Add the Lennard-Jones potential to the potential set
potentialSet.addPotential(LennardJonesSplinePotential('Ar', 'Ar', r_cut=9.0*Angstrom, r_
↪i=7.0*Angstrom))

# Create a TremoloX calculator from the potential set
calculator=TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)
```

The Lennard Jones parameters of this example are taken from the **TraPPE** force field.

Notes

The *LennardJonesSplinePotential* typically describes a combination of van der Waals and repulsive interactions.

To describe the interaction between two particles i and j, the Lennard-Jones potential uses the following potential function:

$$V_{ij}(r) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right].$$

If not specified explicitly in the potential, the Lennard-Jones parameters σ and ϵ are taken from the *ParticleType* the potential acts on. The parameters between different types are calculated based on the Lorentz-Berthelot mixing rules

$$\epsilon_{ij} = \sqrt{\epsilon_i \cdot \epsilon_j}$$

and

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}.$$

Lennard-Jones parameters can be taken from various force fields, such as the **Universal Force Field (UFF)**, the **Generalized AMBER Force Field (GAFF)**, the **COMPASS** force field, or, as in the above example, the **Transferable Potentials for Phase Equilibria (TraPPE)** force field.

Note that some force fields, such as e.g. AMBER, use the slightly different formula $V(r) = \epsilon[(\sigma/r)^{12} - 2(\sigma/r)^{6}]$ to describe Lennard-Jones interactions. In order to use parameters defined for that formula in TremoloXs Lennard-Jones potentials, the σ values taken from these force fields have to be multiplied with $2^{-1/6}$.

The best choice of parameters depends on whether the respective force field has been designed for the desired context. Therefore, one should always carefully read the force field publication before using a new potential parameter set and ideally compare some test results against experiments or suitable quantum mechanical calculations.

The ensure a smooth transition to zero at the cutoff-distance, a spline-based smoothing function S is invoked as $U(r) = U^{LJ}(r) \cdot S(r)$ which switches the potential to zero over the interval $[r_i, r_{cut}]$.

When using *bonded force fields* the bonded_mode parameter can be used to modify how this potential acts between atoms that are connected via less than 4 bonds. If LennardJonesPotential.evaluateAll (or mode_bondless) is chosen, the potential acts between all selected atoms independent of the bonds between them. If LennardJonesPotential.evaluate34 (or mode_14) is chosen, the potential is switched off for all atoms that are connected via one or two consecutive bonds. For atoms separated by exactly three bonds (i.e. atoms 1 and 4 in a bonded sequence), the *special* Lennard-Jones parameters (respectively the combined epsilon14 and sigma14 of the *ParticleType*) are used instead.

LinearInterpolation

class LinearInterpolation(*constraints=None*)

Use linear interpolation for generating images.

Parameters
constraints (*list of type int*) – Unused constraint parameter to make the interface similar to the other interpolation algorithms.

constraints()

Returns
The constrained atoms.

Return type
list of type int

setConstraints(*constraints*)

Set the constraints.

Parameters
constraints (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained.

tags()

Returns
The tags used for defining the constraints.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

LinkedObject

class LinkedObject(*filename*, *object_id*, *, *class_type*, *instance=None*)

Class for linking to an object in a hdf5 file.

Parameters

- **filename** (*str*) – The filename.
- **object_id** (*str*) – The object id.
- **class_type** (*class*) – Only link to objects of this type.
- **instance** (*object*) – The optional instance.

`classType()`

Returns
The class type.

Return type
class

`copy()`

Returns
A copy (without any cached instance).

Return type
The copy

`static fallback()`

The fallback handler for the entire cargo chain. Only applies to the top level object.

Returns
The optional fallback function.

Return type
callable

`filename()`

Returns
The filename.

Return type
str

`instance()`

Returns
[description].

Return type
[type]

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`objectId()`

Returns
The object_id.

Return type
str

`static registry()`

Retrieve the current registry. This is used e.g. when calling nlsave() or nlread() on the CargoSerializable itself.

Returns
The registry.

Return type
Registry

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

classmethod upgrade(*data*, *entries*, *data_version*)

Legacy support:

Upgrade the LinkedObject from the version read from the disk to the current LinkedObject.

Parameters

- **data** (*dict*) – The data.
- **data_version** (*int*) – The version number from serialized input (from the disk).

classmethod version()

Returns
The version of the LinkedObject, used for legacy support issues.

Return type
int

LiquidEquilibrium

class LiquidEquilibrium(*solvent_components_a*, *solvent_components_b*, *mole_fractions=None*, *temperature=None*, *parameters=None*)

Calculate the thermodynamics of mixing of two different liquids.

Parameters

- **solvent_components_a** (*CosmoRealSolvent* | *MoleculeConfiguration* | *CosmoRSMixture*) – The components of the first solvent in the mixture.
- **solvent_components_b** (*CosmoRealSolvent* | *MoleculeConfiguration* | *CosmoRSMixture*) – The components of the second solvent in the mixture.
- **mole_fractions** (*Sequence of float*) – The mole fractions of the first solvent in the mixture. **Default:** (0, 0.25, 0.5, 0.75, 1)
- **temperature** (*PhysicalQuantity* of type temperature) – The mixture temperature. **Default:** `298 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO real solvent parameters

activities()

Returns
The list of the activity of each species. This is indexed by the order of the species in the mix going from solvent A to B, and then by the mixture mole fractions.

Return type
ndarray of floats size (number_of_species x number_of_mole_fractionss)

activityCoefficients()

Returns
The list of activity coefficients. This is indexed by the order of the species in the mix going from solvent A to B, and then by the mixture mole fractions.

Return type
ndarray of floats size (number_of_species x number_of_mole_fractionss)

excessEnergy()

Returns
The excess energy at each specified mole fraction.

Return type
Sequence of *PhysicalQuantity* of type energy

freeEnergyOfMixing()

Returns
The mixing free energy at each specified mole fraction.

Return type
Sequence of *PhysicalQuantity* of type energy

moleFractions()

Returns
The list of mole fractions at which the equilibrium is calculated.

Return type
list of floats

numberOfMoleFractions()

Returns
The number of points at which the mixing is calculated.

Return type
int

numberOfUniqueSpecies()

Returns
The number of unique species in the mixture.

Return type
int

parameters()

Returns
The COSMO-RS parameters

Return type
CosmoRSParameters

solventComponentsA()

Returns
The components of the first solvent in the mixture.

Return type
CosmoRSMixture

solventComponentsB()

Returns
The components of the second solvent in the mixture.

Return type
CosmoRSMixture

temperature()

Returns
The mixture temperature.

Return type
PhysicalQuantity of type temperature

uniqueSpecies()

Returns
A list of the unique species in the mixture.

Return type
Sequence of *CosmoRSMixture*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the thermodynamic properties of mixing solvent water with a mixture of methanol and acetonitrile.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
methanol = database.exportSpecies('methanol')
acetonitrile = database.exportSpecies('acetonitrile')

# Create the mixtures.
solvent_a = CosmoRSMixture([(water, 1)])
solvent_b = CosmoRSMixture([(methanol, 0.25), (acetonitrile, 0.75)])

# Define the mole fractions of solvent_a.
mole_fractions = (0, 0.5, 1)

# Create the liquid equilibrium object.
liquid_equilibrium = LiquidEquilibrium(
    solvent_a,
    solvent_b,
    mole_fractions=mole_fractions,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Get the excess Gibbs free energy.
excess_energy = liquid_equilibrium.excessEnergy()
for x, energy in zip(mole_fractions, excess_energy):
```

(continues on next page)

(continued from previous page)

```
    nlprint(f'The excess energy at x = {x} is {energy}')

# Get the Gibbs free of mixing.
energy_of_mixing = liquid_equilibrium.freeEnergyOfMixing()
for x, energy in zip(mole_fractions, energy_of_mixing):
    nlprint(f'The energy of mixing at x = {x} is {energy}')

# Get the activity coefficients of water.
activity_coeffs = liquid_equilibrium.activityCoefficients()[0]
for x, activity_coeff in zip(mole_fractions, activity_coeffs):
    nlprint(f'The activity coefficient of water at x = {x} is {activity_coeff}')

# Get the activity of water.
activities = liquid_equilibrium.activities()[0]
for x, activity in zip(mole_fractions, activities):
    nlprint(f'The activity of water at x = {x} is {activity}')
```

`lle_example.py`

Notes

The `LiquidEquilibrum` object allows calculation of a number of liquid-liquid equilibrium (LLE) properties. First is the excess Gibbs free energy G^E which is given as:

$$G^E = \sum_i x_i(\mu_i^{solv} - \mu_i^{liq})$$

The excess Gibbs free energy estimates the change in free energy resulting from the different molecular interactions in the liquid and solvated states. Here i represents the species in the mixed solution. μ_i^{solv} is the solvation chemical potential of species i. μ_i^{liq} is the solvation chemical potential of the pure species. x_i is the mole fraction. G^E can be retrieved from the object using the `excessEnergy` method.

Second is the free energy of mixing

$$\Delta G_{mix} = G^E + RT \sum_i x_i \ln\ x_i$$

R is the ideal gas constant and T the temperature of the system. Note that the second term originates from the ideal Gibbs free energy of mixing, which is added to the excess Gibbs free energy. ΔG_{mix} can be retrieved from the object using the `freeEnergyOfMixing` method.

Third is the activity coefficient

$$\gamma_i = \exp\left(\frac{\mu_i^{solv} - \mu_i^{liq}}{RT}\right)$$

Here γ_i respresents the acitivity coefficient of species i in the solution. The activity coefficient is a quantity that encompasses the deviation of species from ideality. γ can be retrieved from the object using the `activityCoefficients` method. The result is in dimensions of number of components in the mixed solution and number of mole fraction steps.

Fourth is the activity which closely relates to the acitivity coefficient

$$a_i = \gamma_i x_i$$

a_i may be considered as the *effective* mole fraction of a species i in solution. a_i can be retrieved from the object using the `activities` method. Likewise, the result is in dimensions of number of components in the mixed solution and number of mole fraction steps.

LiquidThermodynamicIntegrationHook

class LiquidThermodynamicIntegrationHook(*configuration*, *sigma*, *scaling*, *total_steps*, *cutoff=None*, *temperature=None*, *equilibration_steps=None*, *lambda_min=None*, *measurement_call_interval=None*)

Post step hook for calculating the free energy of liquid using an Uhlenbeck-Ford (UF) reference state.

Parameters

- **configuration** (*DistributedConfiguration*) – The current configuration.
- **sigma** (*PhysicalQuantity* of type length.) – The length scale in the UF model. This is similar to the length scale in a Lennard-Jones potential model.
- **scaling** (*int*) – The unitless scaling parameter in the UF model. Must be one of 1, 25, 50, 75 or 100.
- **total_steps** (*int*) – The total number of simulation steps.
- **cutoff** (*PhysicalQuantity* of type length.) – The cutoff of the UF model
- **temperature** (*PhysicalQuantity* of type temperature.) – The temperature of the system
- **equilibration_steps** (*int*) – The number of steps to equilibrate at the reference state.
- **lambda_min** (*float*) – Minimum value of the switching parameter used during the simulation.
- **measurement_call_interval** (*int*) – The interval between calculating measurement data

callInterval()

Returns
The call interval of this hook function.

Return type
int

reference()

Returns the reference system.

Returns
The reference system

Return type
BaseReference

referenceFreeEnergy()

Returns the free energy of the reference system.

Returns
The enegy of the reference system

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

LocalBandstructure

class LocalBandstructure(*configuration*, *symmetry_label=None*, *kpoint_cartesian=None*, *kpoint_fractional=None*, *directions_cartesian=None*, *directions_fractional=None*, *bands_below_fermi_level=None*, *bands_above_fermi_level=None*, *band_index=None*, *points_per_segment=None*, *effective_mass_method=None*, *stencil_order=None*, *delta=None*)

Analysis class for representing the local properties of the bandstructure for specified k-point, bands, and directions.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the local bandstructure.
- **symmetry_label** (*str*) – The k-point (as a symmetry point) for which to calculate the local bandstructure. This options is mutually exclusive with `kpoint_cartesian` and `kpoint_fractional`. **Default:** `'G'` (the Gamma-point `(0.0, 0.0, 0.0)`)
- **kpoint_cartesian** (*PhysicalQuantity* of type inverse length) – The k-point (in Cartesian coordinates) for which to calculate the local bandstructure. This option is mutually exclusive with `symmetry_label` and `kpoint_fractional`. This should be given as a length 3 *PhysicalQuantity*. **Default:** The Gamma-point `(0.0, 0.0, 0.0) * Angstrom**-1`
- **kpoint_fractional** (*list of float*) – The k-point (in fractional coordinates) for which to calculate the local bandstructure as three floats. This option is mutually exclusive with `kpoint_cartesian` and `symmetry_label`. **Default:** The Gamma-point `[0.0, 0.0, 0.0]`
- **directions_cartesian** (*PhysicalQuantity* of type inverse length) – Direction vectors along which to calculate the local bandstructure. This should be given as a *PhysicalQuantity* of shape (*N*, 3) where *N* is the number of directions. This option is mutually exclusive with `directions_fractional`, but either option should be specified. **Note:** The bandstructure is calculated in the extent of the direction vectors so their length has significance for the range of the parabolic and non-parabolic fits.
- **directions_fractional** (*list of lists of floats*) – Direction vectors along which to calculate the local bandstructure. This should be given as a list of *N* lists where *N* is the number of directions and each list contains 3 floats. This option is mutually exclusive with `directions_cartesian`, but either option should be specified. **Note:** The bandstructure is calculated in the extent of the direction vectors so their length has significance for the range of the parabolic and non-parabolic fits.
- **bands_below_fermi_level** (*int*) – The number of bands below the Fermi level per principal spin channel to include. This option is mutually exclusive with `band_index`. If both `bands_below_fermi_level` and `bands_above_fermi_level` are given, at least one must be non-zero. This value should be non-negative. **Default:** 1
- **bands_above_fermi_level** (*int*) – The number of bands above the Fermi level per principal spin channel to include. This option is mutually exclusive with `band_index`. If both `bands_below_fermi_level` and `bands_above_fermi_level` are given, at least one must be non- zero. This value should be non-negative. **Default:** 1
- **band_index** (*int*) – The band index for which to calculate the local bandstructure. This option is mutually exclusive with `bands_below_fermi_level` and `bands_above_fermi_level`. This value must be non-negative.
- **points_per_segment** (*int*) – The number of k-points that is calculated from the given k-point or symmetry point along the extent of the given direction vectors. These k-points along the segments are available for the regression of the non-parabolicity parameter. This value must be larger than or equal to 2. **Default:** `20`
- **effective_mass_method** (`Analytical | Numerical`) – The method used when calculating the effective masses. **Default:** `Numerical`
- **stencil_order** (*int*) – The number of points to use in the second order derivative stencil. This value should be larger than or equal to 2. **Default:** `5`

•**delta** (*PhysicalQuantity* of type inverse length) – The spacing between the k-points used for the stencil. This value must be positive. **Default:** `0.001 * Angstrom**-1`

cartesianDirections()

Returns
The cartesian directions as an (*N*, 3) array where *N* is the number of directions.

Return type
PhysicalQuantity of type inverse length

effectiveMassMethod()

Returns
The method used to calculate the effective masses.

Return type
`Analytical | Numerical`

energies()

Returns

The energies at the k-point as a *PhysicalQuantity* of shape (*directions*, *bands*, *spins*, 2) where *directions* is the number of directions, *bands* is the number of bands, and *spins* is the number of spins.

Note: `self.energies()[:, :, :, 0]` contains energies exactly at the given k-point whereas for `self.energies()[:, :, :, 1]` the evaluations are done at the nearest extrema for each band.

Return type
PhysicalQuantity of type energy

evaluate(*spin=None*, *band=None*, *direction=None*, *points_in_regression=None*, *extremum_correction=None*)

Evaluate the bandstructure properties at the k-point of the object.

Parameters

•**spin** (`Spin.Up | Spin.Down | Spin.All`) – The spin for which to return the effective masses and the non-parabolicity parameters. For `Noncollinear` calculations only `Spin.All` is accepted. **Default:** `Spin.All`

•**band** (*int*) – The band for which to return the effective masses and the non-parabolicity parameters. Must be non-negative. **Default:** All bands that were specified when the *LocalBandstructure* was created.

•**direction** (*int*) – The direction for which to return the effective masses and the non-parabolicity parameters. Must be non-negative. **Default:** All directions that were specified when the *LocalBandstructure* was created.

•**points_in_regression** (*int*) – The number of k-points in the fit of the non-parabolicity parameter. Must be larger than or equal to 2 and smaller than or equal to `points_per_segment`. **Default:** `2`

•**extremum_correction** (`ExtremumCorrection.No | ExtremumCorrection.Nearest | ExtremumCorrection.All`) – The extremum correction method of the k-point to evaluate for. `ExtremumCorrection.No`: There is no extremum correction of the user specified k-point. `ExtremumCorrection.Nearest`: The user specified k-point is corrected to the nearest extremum in each band for the given segment. `ExtremumCorrection.All`: The returned *PhysicalQuantity* arrays of the effective masses and non-parabolicity parameters contains the results of `ExtremumCorrection.No` (in index 0) and `ExtremumCorrection.Nearest` (in index 1) in the last axis. **Default:** `ExtremumCorrection.Nearest`

Returns

A tuple of two *PhysicalQuantity* arrays. Both have shape (*directions*, *bands*, *spins*, 1) or (*directions*, *bands*, *spins*, 2) depending on the `extremum_correction` flag. Here *directions* is the number of directions, *bands* is the number of bands, and *spins* is the number of spins.

The first *PhysicalQuantity* array is of type mass and contains effective masses. The second is of type inverse energy and contains non-parabolicity parameters.

Return type
tuple of *PhysicalQuantity* arrays

fitBands(*spin=None*, *band=None*, *direction=None*, *points_in_regression=None*, *extremum_correction=None*)

Method for calculating the parabolic and non-parabolic fits to the bandstructure.

Parameters

- **spin** (`Spin.Up | Spin.Down | Spin.All`) – The spin for which to calculate the effective mass and the non- parabolicity parameter. For `Noncollinear` calculations only Spin.All is accepted. For this method Spin.All is not accepted for `Polarized` calculations since only one spin species can be queried at a time. **Default:** `Spin.All` (`Spin.Up` for `Polarized` calculations)
- **band** (*int*) – The band index for which to calculate the effective mass and the non-parabolicity parameter. Must be non-negative. **Default:** `0`
- **direction** (*int*) – The direction index for which to calculate the effective mass and the non-parabolicity parameter. Must be non-negative. **Default:** `0`
- **points_in_regression** (*int*) – The number of k-points for which to fit the non-parabolicity parameter. Must be larger than or equal to 2 and smaller than or equal to points_per_segment **Default:** `2`
- **extremum_correction** (`ExtremumCorrection.No | ExtremumCorrection.Nearest`) – The extremum correction method of the k-point to evaluate for. `ExtremumCorrection.No`: There is no extremum correction of the user specified k-point. `ExtremumCorrection.Nearest`: The user specified k-point is corrected to the nearest extremum. **Default:** `ExtremumCorrection.Nearest`

Returns

A tuple of five quantities.

1. The norm of the distances from the k-point where the subsequent quantities are evaluated as a *PhysicalQuantity* of type inverse length.
2. The calculated band as a *PhysicalQuantity* of type energy.
3. The effective-mass (parabolic) band as a *PhysicalQuantity* of type energy.
4. The non-parabolic band as a *PhysicalQuantity* of type energy.
5. The deviation of the non-parabolic band from the calculated band as a *PhysicalQuantity* of type energy.

Return type
tuple

kpoint()

Returns
The k-point in fractional coordinates.

Return type
`numpy.array`

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the effective mass and non-parabolicity parameter of silicon for a k-point, 3 directions, and one band below the Fermi level:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(7, 7, 7),
    density_mesh_cutoff=75.0*Hartree,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
```

(continues on next page)

(continued from previous page)

```
bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------------
# Local bandstructure
# ------------------------------------------------------------------
directions_cartesian = [[0.0, 0.0, 1.0],
                        [1.0, 1.0, 0.0],
                        [1.0, 1.0, 1.0]]*Angstrom**-1

local_bandstructure = LocalBandstructure(
    configuration=bulk_configuration,
    directions_cartesian=directions_cartesian,
    kpoint_fractional=[0.5, 0.5, 0.5],
    bands_below_fermi_level=1,
    bands_above_fermi_level=0,
    stencil_order=5,
    delta=0.001*Angstrom**-1,
    points_per_segment=20
    )

nlprint(local_bandstructure)

# Evaluation of the local bandstructure parameters.
effective_mass, nonparabolicity_parameter = local_bandstructure.evaluate(
    spin=Spin.Up,
    band=0,
    direction=2
    )

eff_mass = effective_mass[0,0,0].inUnitsOf(electron_mass)
nonpara = nonparabolicity_parameter[0,0,0].inUnitsOf(eV**-1)

print('Effective mass: %.2f me' % (eff_mass))
print('Non-parabolicity parameter: %.4f 1/eV' % (nonpara))
```

local_bandstructure.py

Notes

•The non-parabolic dispersion model is given by

$$E(\mathbf{k})\left(1+\alpha E(\mathbf{k})\right)=\frac{\hbar^2|\mathbf{k}|^2}{2m^*},$$

where α and m^* are the non-parabolicity parameter and the effective mass respectively. The non-parabolic band dispersion is explicitly expressed as

$$E(\mathbf{k})=\frac{-1+\sqrt{1+\frac{2\alpha\hbar^2}{m^*}|\mathbf{k}|^2}}{2\alpha}.$$

- The non-parabolicity parameter is fitted for a number of points along the length and direction of the given direction vector.
- The behavior of `bands_above_fermi_level` and `bands_below_fermi_level` with regard to the spin type *unpolarized*, *polarized*, and *noncollinear* is described in *Notes* of the *EffectiveMass* analysis object.

LocalDensityOfStates

class LocalDensityOfStates(*configuration*, *kpoints=None*, *energies=None*, *energy_zero_parameter=None*, *band_indices=None*, *spectrum_method=None*, *density_mesh_cutoff=None*)

Class for calculating the partial electron density.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the partial electron density.
- **kpoints** – The k-points for which to perform the calculation. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the projected density of states. The energies are relative to the zero of energy specified in `energy_zero_parameter`. **Default:** `numpy.arange(-1.0, 1.0, 0.01) * eV`
- **energy_zero_parameter** (`FermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `FermiLevel`
- **band_indices** (*list of int*) – The band indices of the Bloch states to include in the partial density. **Default:** All occupied bands
- **spectrum_method** (`GaussianBroadening`) – The method to use for calculating the partial electron density. **Default:** `GaussianBroadening(0.1*eV)`
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

bandIndices()

Returns
The band indicies to include in the partial electron density.

Return type
list of int.

energies()

Returns
The energies.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Sum` | `Spin.Up` | `Spin.Down` | `Spin.X` | `Spin.Y` | `Spin.Z`) – The spin component to project on. **Default:** `Spin.All`

Returns
The values for each energy at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-3} * energy $^{-1}$

fermiLevel(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. Must be either `Spin.Up`, `Spin.Down`, or `Spin.All`. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** `Spin.Up`

Returns
The Fermi level in absolute energy.

Return type
`PhysicalQuantity` of type energy

kpoints()

Returns
The kpoints given as input.

Return type
MonkhorstPackGrid | *RegularKpointGrid* | numpy.ndarray

localDensityOfStateAtEnergyIndex(*energy_index=None*)

Return the local density of states for a particular energy index.

Parameters
enegy_index (*int*) – The index of the energy for which to return the local density of states.

Returns
A GridValues object with the local dos at a particular energy.

Return type
GridValues

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spectrumMethod()

Returns
The method used for calculating the partial electron density.

Return type
GaussianBroadening

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Analyze the local density of states of a silver (111) surface. This structure is characterized by a surface state around the Γ-point. In the example, the kpoint sampling is specified with a *RegularKpointGrid* in a kpoint region around the Γ-point.

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [2.88903, 0.0, 0.0]*Angstrom
vector_b = [-1.44451, 2.50197, 0.0]*Angstrom
vector_c = [0.0, 0.0, 40.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver]

# Define coordinates
fractional_coordinates = [[ 0.749999669084,  0.5           ,  0.175654000711],
                          [ 0.749999669084,  0.5           ,  0.352570000323],
                          [ 0.749999669084,  0.5           ,  0.529485999935],
                          [ 0.749999669084,  0.5           ,  0.706401999547],
                          [ 0.416666424719,  0.833333341397,  0.234626000582],
                          [ 0.416666424719,  0.833333341397,  0.411542000194],
                          [ 0.416666424719,  0.833333341397,  0.588457999806],
```

(continues on next page)

(continued from previous page)

```
                            [ 0.416666424719,  0.833333341397,  0.765373999418],
                            [ 0.083334237113,  0.166666658603,  0.293598000453],
                            [ 0.083334237113,  0.166666658603,  0.470514000065],
                            [ 0.083334237113,  0.166666658603,  0.647429999677],
                            [ 0.083334237113,  0.166666658603,  0.824345999289]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = LDA.PZ

k_point_sampling = MonkhorstPackGrid(
    na=11,
    nb=11,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=45.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('local-density-of-states.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Local Density Of States
# ------------------------------------------------------------
kpoints = RegularKpointGrid(
    ka_range=[-0.05, 0.05],
    kb_range=[-0.05, 0.05],
    kc_range=[0.0, 0.0],
    na=5,
    nb=5,
    nc=1,
    )
```

(continues on next page)

(continued from previous page)

```
local_density_of_states = LocalDensityOfStates(
    configuration=bulk_configuration,
    kpoints=kpoints,
    energies=numpy.linspace(-1, 1, 101)*eV,
    band_indices=All,
    energy_zero_parameter=FermiLevel,
    spectrum_method=GaussianBroadening(0.1*eV),
    )
nlsave('local-density-of-states.hdf5', local_density_of_states)
```

local_density_of_states.py

Notes

The *LocalDensityOfStates* can be calculated by specifying the

1. `energies`. The local density of states will be calculated at the specified energies.
2. `band_indices`. Only states from the specified band indices will contribute to the partial electron density. By default, all bands are taken into account and the selection is determined by the energy range.
3. `kpoints`. The partial electron density is calculated from the eigenstates at the specified k-points.
4. `spectrum_method`. The method used to integrate the density of states. Currently only `GaussianBroadening` is supported. Each electronic state is assigned a gaussian weight determined by the eigenvalue and gaussian broadening.

The local density of states is defined as:

$$\begin{aligned} D(E, \mathbf{r}) &= \sum_i \sum_{\mathbf{k}} w_{\mathbf{k}} |\psi_{i,\mathbf{k}}(\mathbf{r})|^2 \delta(E - \epsilon_{i,\mathbf{k}}), \\ &\approx \sum_i \sum_{\mathbf{k}} w_{\mathbf{k}} |\psi_{i,\mathbf{k}}(\mathbf{r})|^2 w(E, \epsilon_{i,\mathbf{k}}), \end{aligned}$$

where the sum over band indices i includes only the specified bands (by default all bands). The weights $w_{\mathbf{k}}$ and $w(E, \epsilon_{i,\mathbf{k}})$ are the k-point weights and the Gaussian weights, respectively. The energy dependent Gaussian weight is computed with a normalized Gaussian centered at $\epsilon_{i,\mathbf{k}}$ with standard deviation σ over the specified energy range, i.e.

$$w(\epsilon_{i,\mathbf{k}}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(E - \epsilon_{i,\mathbf{k}})^2}{2\sigma^2}\right)$$

Tersoff-Hamann approximation of simulated STM

The *LocalDensityOfStates* can be used to simulate scanning tunneling microscopy (STM) images within the Tersoff-Hamann approximation[1] assuming that the tip wave function has s-orbital character. When simulating an STM image, the `energy` can be interpreted as the bias difference applied between the tip and the surface and the *LocalDensityOfStates* represents the STM dI/dV measurement.

Notice that the Tersoff-Hamann approximation is the simplest approach to simulated STM images and it is not always applicable. Also bare in mind that far away from a surface the partial density will be zero for an LCAO calculation since the basis orbitals have a finite range.

[1] J.ăTersoff and D.ăR. Hamann. Theory of the scanning tunneling microscopy. *Phys. Rev. B*, 31:805, 1985. doi:10.1103/PhysRevB.31.805.

LocalDeviceDensityOfStates

class LocalDeviceDensityOfStates(*configuration*, *energy=None*, *kpoints=None*, *kpoints_weights=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *density_mesh_cutoff=None*)

Constructor for the Local Device Density of States (LDDOS) class. This class inherits from the *GridValues* class.

Parameters

•**configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The one or two-probe configuration with attached calculator for which the LDDOS. should be calculated.

•**energy** (*PhysicalQuantity* of type energy.) – The energy for which the LDDOS should be calculated. **Default:** `0.0*eV`

•**kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | sequence of sequence (size 3) of float) – The k-points for which the LDDOS should be calculated. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

•**kpoints_weights** (*sequence of float*) – The weight of each k-point for which the transmission spectrum should be calculated. This should only be specified if kpoints is given as a list of k-points. **Default:** The weights corresponding to the MonkhorstPackGrid, or a list of [1.0, 1.0,] if k-points are specified as floats.

•**contributions** (`Left` | `Right` | `All`) – The density contributions to include in the LDDOS. **Default:** `All`

•**self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used for the LDDOS. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`

•**energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`.) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`

•**infinitesimal** (*PhysicalQuantity* of type energy.) – Small energy, used to move the density of states calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

•**projection_type** (*str*) –

The type of projection to perform. Should be either

–sum for the sum over the plane spanned by the two other axes.

–average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

contributions()

Returns
The contributions used in the LDDOS calculation.

Return type
Left | Right | All

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy^{-1} Œ length^{-4}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

energy()

Returns
The energy used in this LDDOS.

Return type
PhysicalQuantity of type energy.

energyZero()

Returns
The energy zero used for the energy scale in this LDDOS.

Return type
PhysicalQuantity of type energy.

energyZeroParameter()

Returns
The specified choice for the energy zero.

Return type
AverageFermiLevel | AbsoluteEnergy

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The value at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type energy^{-1} Œ length^{-3}

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the local device density of states is calculated for, always Spin.All.

Return type
Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the *LocalDeviceDensityOfStates* for a *DeviceConfiguration* and save the result for visualization with QuantumATK:

```
ldos = LocalDeviceDensityOfStates(device_configuration, 0.0*eV, contributions=All)
nlsave('ldos.nc',ldos)
```

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

The routine utilizes the symmetries of the configuration to reduce the computational load for k-point averaging.

The local density of states (LDOS) $D(E, \mathbf{r})$ for a device is computed via the spectral density matrix $\rho(E) = \rho^L(E) + \rho^R(E)$ (cf. *The non-equilibrium Greens function (NEGF) method*) where L/R denotes the contribution from the left/right electrode (see the keyword `contributions` above).

For *DFT: LCAO* calculators the LDDOS is evaluated as

$$D(E, \mathbf{r}) = \sum_{ij} \rho_{ij}(E)\phi_i(\mathbf{r})\phi_j(\mathbf{r})$$

where we note that the basis set orbitals, $\phi_i(\mathbf{r})$, are real functions in QuantumATK through the use of spherical harmonics.

For *Semi Empirical* calculators the LDDOS is projected on real space through Gaussian functions as

$$D(E,\mathbf{r}) = \sum_i \rho_{ij} S_{ij} \left(\frac{\alpha_i}{\pi}\right)^{\frac{3}{2}} e^{-\alpha_i |\mathbf{r}-\mathbf{R}_i|^2}$$

to ensure consistency with the definition of the electron density (see *Electron density*).

The LDOS is related to the *DeviceDensityOfStates* (DDOS) $D(E)$ via an integral over all space.

If the function is integrated over both space and energy with the Fermi function,

$$\int D(E,\mathbf{r}) f\left(\frac{E-\mu}{k_B T}\right) dE d\mathbf{r} = \sum_{ij} D_{ij} S_{ij} = N,$$

it will give the total number of electrons N.

LocalStress

class LocalStress(*configuration*)

Class for calculating the local stress.

Parameters
configuration (*BulkConfiguration*) – The configuration for which the local stress should be calculated.

evaluate(*atom_index=None*)

Return the local stress of the selected atom. Each stress tensor is given by 6 energies: $(\sigma_i^{xx}, \sigma_i^{yy}, \sigma_i^{zz}, \sigma_i^{yz}, \sigma_i^{xz}, \sigma_i^{xy})$ where i is `atom_index` and σ_i^{pair} is the corresponding stress tensor.

Parameters
atom_index (*int*) – Index of atom to return. **Default:** Local stress of all atoms.

Returns
The requested stress tensor as a length 6 array if `atom_index` is not `None`, or all stress tensors as an (*N*, 6) array if `atom_index` is `None`.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *LocalStress* of a SiO2 quartz cell using a classical Tersoff potential:

```
# ------------------------------------------------------------
# Set up a SiO2 alpha-quartz cell.
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.00000 ],
                          [ 0.0000,  0.4697,  0.66667 ],
                          [ 0.5303,  0.5303,  0.33333 ],
                          [ 0.4135,  0.2669,  0.1191  ],
                          [ 0.2669,  0.4135,  0.547567],
                          [ 0.7331,  0.1466,  0.785767],
                          [ 0.5865,  0.8534,  0.214233],
                          [ 0.8534,  0.5865,  0.452433],
                          [ 0.1466,  0.7331,  0.8809  ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = Tersoff_SiO_2007()
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Local stress
# ------------------------------------------------------------
local_stress = LocalStress(bulk_configuration)
```

(continues on next page)

(continued from previous page)

```
nlsave('Quartz_ls.nc', local_stress)
nlprint(local_stress)
```

local_stress.py

Notes

•The *LocalStress* provides a measure how much each atom contributes to the overall stress tensor of the system. It does not represent a physical stress, in a strict sense, as the point particles do not have a finite volume. Therefore, in contrast to the *Stress*, the local stress is not given in units of energy per volume but in energy units.

•The *LocalStress* is calculated by analytically decomposing the total stress tensor into local contributions of the individual atoms of the system. Therefore, the local stress can only be calculated for classical potentials with the *TremoloX-Calculator*.

•The local stress contributions of each atom sum up to yield the total stress tensor of the system

$$\sigma = \frac{1}{V}\sum_i \sigma_i \,.$$

•The local stress of each atom is composed of additive components arising from the pair-, triple-, and quadruple-interactions in the system.

For pairwise interactions $v_{ij}(\mathbf{r}_{ij})$ the local stress of atom i is calculated as

$$\sigma_i^{\text{pair}} = \sum_{(i,j)\in\mathcal{P}} -\frac{1}{2}\mathbf{F}_i^{ij}\mathbf{r}_{ij}^T + \sum_{(j,i)\in\mathcal{P}} \frac{1}{2}\mathbf{F}_i^{ji}\mathbf{r}_{ij}^T \,,$$

where $\mathbf{F}_k^{ij} = -\frac{d}{d\mathbf{r}_k}v_{ij}$ is the force on atom k resulting form the interaction between atoms i and j. The sum runs over all interactions pairs $\mathcal{P}$.

For triple and quadruple interactions the stress contributions are accordingly

$$\sigma_i^{\text{triple}} = -\frac{1}{2}\sum_{(i,j,k)\in\mathcal{T}} \left(\mathbf{F}_j^{ijk}\mathbf{r}_{ij}^T + \mathbf{F}_k^{ijk}\mathbf{r}_{ik}^T\right) + \frac{1}{2}\sum_{(j,i,k)\in\mathcal{T}} \mathbf{F}_i^{jik}\mathbf{r}_{ij}^T + \frac{1}{2}\sum_{(j,k,i)\in\mathcal{T}} \mathbf{F}_i^{jki}\mathbf{r}_{ij}^T$$

and

$$\begin{aligned}\sigma_i^{\text{quadruple}} =< & \tfrac{1}{2}\textstyle\sum_{(i,j,k,m)\in\mathcal{Q}} \mathbf{F}_i^{ijkm}\mathbf{r}_{ij}^T + \tfrac{1}{2}\sum_{(j,i,k,m)\in\mathcal{Q}} \left(-\mathbf{F}_j^{jikm}\mathbf{r}_{ij}^T + (\mathbf{F}_j^{jikm} + \mathbf{F}_i^{jikm})\mathbf{r}_{ik}^T\right) \\ & < \\ < & + \tfrac{1}{2}\textstyle\sum_{(j,k,i,m)\in\mathcal{Q}} \left(-(\mathbf{F}_j^{jkim} + \mathbf{F}_k^{jkim})\mathbf{r}_{ik}^T - \mathbf{F}_m^{jkim}\mathbf{r}_{im}^T\right) + \tfrac{1}{2}\sum_{(j,k,m,i)\in\mathcal{Q}} \mathbf{F}_i^{jkmi}\mathbf{r}_{im}^T\end{aligned}$$

Here, $\mathcal{T}$ and $\mathcal{Q}$ denote the sets of all interaction-triples and -quadruples, and $\mathbf{F}_k^{\text{triple,quadruple}} = -\frac{d}{d\mathbf{r}_k}v^{\text{triple,quadruple}}$ is the force on particle k arising from a triple- or quadruple-potential v.

The extension to manybody potentials, such as *EAMsetflPotential* is carried out in a similar way.

•In the current implementation, the long-range contributions in the *CoulombSPME* potential are not included in the local stress as they are evaluated in reciprocal space.

•The local stress of a configuration can be visualized by opening the saved local stress object in the *Viewer*. To visualize the local stress tensor of each atom i, you can select between the local pressure representation, i.e.

$$p_i = -\frac{1}{3}(\sigma_i^{xx} + \sigma_i^{yy} + \sigma_i^{zz}) \,,$$

or the Frobenius norm of the tensor

$$norm(\sigma_i) = \left[\sum_{\alpha,\beta=\{x,y,z\}} (\sigma_i^{\alpha\beta})^2 \right]^{1/2} .$$

By selecting the `User defined` option, you can choose which components should be included in the Frobenius norm.

LocalStructure

class LocalStructure(*configuration*, *cutoff_radius_structure=None*, *cutoff_radius_coordination=None*, *centrosymmetry_neighbors=None*)

Class for calculating the local structure parameters.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration for which the structure analysis should be calculated.
- **cutoff_radius_structure** (*PhysicalQuantity* of type length) – The cutoff radius used for the local structure analysis. **Default:** `8.0 Angstrom`
- **cutoff_radius_coordination** (*PhysicalQuantity* of type length) – The cutoff radius used to calculate the coordination number. Cannot be larger than cutoff_radius. **Default:** `4.0 Angstrom`
- **centrosymmetry_neighbors** (*even int*) – The number of neighbors for each atom to be included in the centro-symmetry calculation (e.g. 12 for face centered cubic, 8 for body centered cubic). Must be between 8 and 20. **Default:** `12` (face centered cubic).

centrosymmetryParameters(*atom_index=None*)

Return the list of centrosymmetry parameters for all atoms or a selected atom.

Parameters
atom_index (*int*) – Index of the atom to return the centrosymmetry parameters for. **Default:** Centrosymmetry parameters of all atoms.

Returns
Either an array of all centrosymmetry parameters if `atom_index` is `None`, or the parameter for the specified atom.

Return type
`numpy.array` | float

coordinationNumbers(*atom_index=None*)

Return the list of coordination numbers for all atoms or a selected atom.

Parameters
atom_index (*int*) – Index of the atom to return the coordination number for. **Default:** Coordination numbers of all atoms.

Returns
Either an array of all coordination numbers if `atom_index` is `None`, or the coordination number for the specified atom.

Return type
`numpy.array` | float

evaluate(*atom_index=None*)

Return the list of local structure indices for all atoms or a selected atom.

- 0: Undefined
- 1: FCC

•2: BCC

•3: HCP

•4: Icosahedral

•5: Diamond-like

Parameters
atom_index (*int*) – Index of the atom to return the local crystal structure for. **Default:** Local structure of all atoms.

Returns
Either all local structure indices as an array if `atom_index` is `None`, or the structure index for the specified atom

Return type
`numpy.array` | float

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Analyze the *LocalStructure* of fcc copper:

```
# ------------------------------------------------------------
# Local structure
# ------------------------------------------------------------
local_structure = LocalStructure(bulk_configuration,
                                 cutoff_radius_structure=8.0*Angstrom,
                                 cutoff_radius_coordination=3.5*Angstrom,
                                 centrosymmetry_neighbors=12)

nlsave('ls_copper.nc', local_structure)
nlprint(local_structure)

# Extract the local structure index of the first atom
```

(continues on next page)

(continued from previous page)

```
ls_atom1 = local_structure.evaluate(atom_index=0)

# Extract the centrosymmetry parameter of the first atom
cs_atom1 = local_structure.centrosymmetryParameters(atom_index=0)

# Extract the coordination number of the first atom
cn_atom1 = local_structure.coordinationNumbers(atom_index=0)
```

local_structure.py

Notes

- The *LocalStructure* object assigns a crystal structure type to each atom based on its local environment, as described in[1]. The range of the local atomic environment is specified by the `cutoff_radius_structure` argument. The assigned crystal structures, however, are not particularly sensitive to the exact choice of this value, and the default setting commonly yields correct results. The results can be visualized by the corresponding plugin of the 3D Viewer.
- Furthermore, the *LocalStructure* object calculates the centrosymmetry parameter, as well as the coordination number for each atom.
- The centrosymmetry parameter quantifies the symmetry of the local environment around each atomic site, as described in[2]. When the atom represents a perfect center of symmetry, a zero value is returned, while deviations from a centrosymmetric arrangement yield values larger than zero. This parameter can, for instance, be used to detect defects in the crystal lattice. For meaningful results, it is important to specify the correct number of nearest neighbors of the considered crystal structure (e.g. 8 for bcc, and 12 for fcc) via the `centrosymmetry_neighbors` argument. Note that diamond-like structures can not be analyzed, since they are not centrosymmetric.
- The coordination number returns the number of neighbors of each atom within a given cutoff range, as specified by the `cutoff_radius_coordination` argument. Unlike the crystal structure indices, the resulting coordination numbers are rather sensitive to the choice of the cutoff and the default setting may not provide the best results for all materials.

[1] A.ăStukowski. Structure identification methods for atomistic simulations of crystalline materials. *Modelling and Simulation in Materials Science and Engineering*, 20(4):45021, 2012. URL: http://stacks.iop.org/0965-0393/20/i=4/a=045021.

[2] C.ăL. Kelchner, S.ăJ. Plimpton, and J.ăC. Hamilton. Dislocation nucleation and defect structure during surface indentation. *Phys. Rev. B*, 58:11085–11088, Nov 1998. doi:10.1103/PhysRevB.58.11085.

MACEFittingParameters

class MACEFittingParameters(*experiment_name*, *training_data_filepath=None*, *testing_data_filepath=None*, *isolated_atom_energies=None*, *validation_fraction=None*, *batch_size=None*, *validation_batch_size=None*, *max_number_of_epochs=None*, *patience=None*, *device=None*, *distributed_training=None*, *random_seed=None*, *keep_checkpoints=None*, *number_of_workers=None*, *default_dtype=None*, *number_of_channels=None*, *max_l_equivariance=None*, *distance_cutoff=None*, *energy_key=None*, *forces_key=None*, *stress_key=None*, *energy_weight=None*, *forces_weight=None*, *stress_weight=None*, *loss_function_type=None*, *compute_stress=None*, *evaluation_interval=None*, *error_measure=None*, *model_type=None*, *number_of_interactions=None*, *correlation_order=None*, *max_ell=None*, *number_of_radial_basis_functions=None*, *scaling_type=None*, *learning_rate=None*, *weight_decay=None*, *restart_from_last_checkpoint=None*, *use_exponential_moving_average=None*, *exponential_moving_average_decay=None*, *scheduler_patience=None*, *gradient_clipping_threshold=None*, *save_for_cpu=None*, *log_directory=None*, *results_directory=None*, *checkpoint_directory=None*, *foundation_model_path=None*, *use_multiheads_finetuning=None*, *pretrained_head_train_file=None*, *number_of_samples_from_pretrained_head=None*, *mp_data_path=None*, *mp_descriptors_path=None*, *additional_parameters=None*)

A parameter class for passing settings to training MACE models[1][2]. The class acts as a QuantumATK interface to the *mace-torch* package functionality. The parameters are used for setting up from-scratch fitting or fine-tuning of MACE models to custom data. The default parameters mostly follow the default settings in the open source *mace-torch* package with some exceptions for increasing the ease of use. Advanced settings not covered by the parameters available via this class can be set using the `additional_parameters` dictionary via keys corresponding to advanced open source MACE arguments.

Parameters

•**experiment_name** (*str*) – The name of the experiment.

•**training_data_filepath** (*str*) – The path to the training data. **Default:** train.xyz

•**testing_data_filepath** (*str*) – The path to the testing data. **Default:** None

•**isolated_atom_energies** (dict of `Element`: *PhysicalQuantity* of type energy) – The energy of an isolated atom for each species. The dictionary should contain the species as keys and the energy as values. **Default:** {}

•**validation_fraction** (*float*) – The fraction of the training data to use for validation. **Default:** 0.1

•**batch_size** (*int*) – The batch size for training. **Default:** 8

•**validation_batch_size** (*int*) – The batch size for validation. **Default:** 8

•**max_number_of_epochs** (*int*) – The maximum number of epochs to train for. **Default:** 200

•**patience** (*int*) – The number of epochs to wait before early stopping. **Default:** 40

•**device** (`Automatic` | *MACEParameterOptions.DEVICE* option) – The device to train on. **Default:** `Automatic`

•**distributed_training** (*bool*) – Whether to use distributed (multi GPU) training. **Default:** False

[1] Ilyes Batatia, DavidăPeter Kovacs, Gregor N.ăC. Simm, Christoph Ortner, and Gabor Csanyi. MACE: higher order equivariant message passing neural networks for fast and accurate force fields. In AliceăH. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*. 2022. URL: https://openreview.net/forum?id=YPpSngE-ZU.

[2] Ilyes Batatia, Simon Batzner, DávidăPéter Kovács, Albert Musaelian, Gregor N.ăC. Simm, Ralf Drautz, Christoph Ortner, Boris Kozinsky, and Gábor Csányi. The design space of e(3)-equivariant atom-centered interatomic potentials. 2022. arXiv:2205.06643, doi:10.48550/arXiv.2205.06643.

•**random_seed** (*int*) – The random seed used for splitting the provided training data into validation and training data sets. **Default:** 123

•**keep_checkpoints** (*bool*) – Whether to keep checkpoints. **Default:** True

•**number_of_workers** (*int*) – The number of workers for data loading. **Default:** 0

•**default_dtype** (*MACEParameterOptions.DTYPE* option of type str) – The default torch data type. **Default:** MACEParameterOptions.DTYPE.FLOAT64

•**number_of_channels** (*int*) – The number of embedding channels. **Default:** 128

•**max_l_equivariance** (*int*) – The max L equivariance of the message. **Default:** 1

•**distance_cutoff** (*PhysicalQuantity* of type energy) – The radial cutoff distance for each node. **Default:** 5.0*Angstrom

•**energy_key** (*str*) – The key for the energy in the dataset. **Default:** REF_energy

•**forces_key** (*str*) – The key for the forces in the dataset. **Default:** REF_forces

•**stress_key** (*str*) – The key for the stress in the dataset. **Default:** REF_stress

•**energy_weight** (*float*) – The weight of the energy loss. **Default:** 1

•**forces_weight** (*float*) – The weight of the forces loss. **Default:** 100

•**stress_weight** (*float*) – The weight of the stress loss. **Default:** 1

•**loss_function_type** (*MACEParameterOptions.LOSS_TYPE* option of type str) – The loss function type to use. **Default:** MACEParameterOptions.LOSS_TYPE.WEIGHTED

•**compute_stress** (*bool*) – Whether to compute the stress. **Default:** False

•**evaluation_interval** (*int*) – The epoch interval at which evaluations should occur. **Default:** 1

•**error_measure** (*MACEParameterOptions.ERROR_MEASURE* option of type str) – Type of error table produced at the end of the training. **Default:** MACEParameterOptions.ERROR_MEASURE.PER_ATOM_RMSE

•**model_type** (*MACEParameterOptions.MODEL_TYPE* option of type str) – The model type to train. **Default:** MACEParameterOptions.MODEL_TYPE.MACE

•**number_of_interactions** (*int*) – The number of interactions. **Default:** 2

•**correlation_order** (*int*) – The correlation order at each layer. **Default:** 3

•**max_ell** (*int*) – The highest ell of spherical harmonics. **Default:** 3

•**number_of_radial_basis_functions** (*int*) – The number of radial basis functions. **Default:** 8

•**scaling_type** (*MACEParameterOptions.SCALING_TYPE* option of type str) – The type of scaling to the output. **Default:** MACEParameterOptions.SCALING_TYPE.RMS_FORCES_SCALING

•**learning_rate** (*float*) – The learning rate of optimizer. **Default:** 0.005

•**weight_decay** (*float*) – The weight decay (L2 penalty). **Default:** 5e-7

•**restart_from_last_checkpoint** (*bool*) – Whether to restart training from last saved checkpoint. In order to ensure easy restartability/resubmittability from the job tool, this setting has to be activated from the first submission. **Default:** True

•**use_exponential_moving_average** (*bool*) – Whether to use exponential moving average. **Default:** False

•**exponential_moving_average_decay** (*float*) – The decay rate of the exponential moving average. **Default:** 0.99

•**scheduler_patience** (*int*) – The patience of the scheduler. **Default:** 5

- **gradient_clipping_threshold** (*float*) – The gradient clipping threshold for clipping high parameter values during training. **Default:** 10
- **save_for_cpu** (*bool*) – Whether to save a model loadable on CPU. **Default:** True
- **log_directory** (*str*) – The directory to save logs. **Default:** None
- **results_directory** (*str*) – The directory to save results. **Default:** None
- **checkpoint_directory** (*str*) – The directory to save checkpoints. **Default:** None
- **foundation_model_path** (*str*) – The path to the foundation MACE model to finetune. **Default:** None
- **use_multiheads_finetuning** (*bool*) – Whether to use multiheads finetuning. **Default:** False
- **pretrained_head_train_file** (*str*) – The path to the pretrained head train file for multiheads finetuning. **Default:** None
- **number_of_samples_from_pretrained_head** (*int*) – The number of samples from the pretrained head for multiheads finetuning. **Default:** 10000
- **mp_data_path** (*str*) – The path to the MP data for finetuning an MP foundation model. **Default:**
- **mp_descriptors_path** (*str*) – The path to the MP descriptors for finetuning an MP foundation model. **Default:**
- **additional_parameters** (*dict*) – Additional parameters for the MACE model. Here parameters that have not been included in the parameter class, but that are available in the *mace-torch* package, can be set as dictionary entries. Keys must match *mace-torch* input parameter names exactly and values must be in the right format as expected by the open source package. **Default:** {}

additionalParameters()

Returns
Additional parameters for the MACE model.

Return type
dict

batchSize()

Returns
The batch size for training.

Return type
int

checkpointDirectory()

Returns
The directory to save checkpoints.

Return type
str

computeStress()

Returns
Whether to compute the stress.

Return type
bool

`correlationOrder()`

Returns
The correlation order at each layer.

Return type
int

`defaultDtype()`

Returns
The default torch data type.

Return type
str

`device()`

Returns
The device to train on.

Return type
str

`distanceCutoff()`

Returns
The radial cutoff distance for each node.

Return type
PhysicalQuantity of type energy

`distributedTraining()`

Returns
Whether to use distributed (multi GPU) training.

Return type
bool

`energyKey()`

Returns
The key for the energy in the dataset.

Return type
str

`energyWeight()`

Returns
The weight of the energy loss.

Return type
float

`errorMeasure()`

Returns
Type of error table produced at the end of the training.

Return type
str

evaluationInterval()

Returns
The epoch interval at which evaluations should occur.

Return type
int

experimentName()

Returns
The name of the experiment.

Return type
str

exponentialMovingAverageDecay()

Returns
The decay rate of the exponential moving average.

Return type
float

forcesKey()

Returns
The key for the forces in the dataset.

Return type
str

forcesWeight()

Returns
The weight of the forces loss.

Return type
float

foundationModelPath()

Returns
The path to the foundation MACE model to be finetuned.

Return type
str

gradientClippingThreshold()

Returns
The gradient clipping value.

Return type
float

isolatedAtomEnergies()

Returns
The energy of an isolated atom for each species.

Return type
dict of {Element: *PhysicalQuantity* of type energy}

keepCheckpoints()

Returns
Whether to keep checkpoints.

Return type
bool

learningRate()

Returns
The learning rate of optimizer.

Return type
float

logDirectory()

Returns
The directory to save logs.

Return type
str

lossFunctionType()

Returns
The loss function to use.

Return type
str

maxEll()

Returns
The highest ell of spherical harmonics.

Return type
int

maxLEquivariance()

Returns
The max L equivariance of the message.

Return type
int

maxNumberOfEpochs()

Returns
The maximum number of epochs to train for.

Return type
int

modelType()

Returns
The model type to train.

Return type
str

mpDataPath()

Returns
The path to the MP data for finetuning an MP foundation model.

Return type
str

mpDescriptorsPath()

Returns
The path to the MP descriptors for finetuning an MP foundation model.

Return type
str

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfChannels()

Returns
The number of embedding channels.

Return type
int

numberOfInteractions()

Returns
The number of interactions.

Return type
int

numberOfRadialBasisFunctions()

Returns
The number of radial basis functions.

Return type
int

numberOfSamplesFromPretrainedHead()

Returns
The number of samples from the pretrained head to use for Multihead Replay Finetuning.

Return type
int

numberOfWorkers()

Returns
The number of workers for data loading.

Return type
int

patience()

Returns
The number of epochs to wait before early stopping.

Return type
int

pretrainedHeadTrainFile()

Returns
The path to the pretrained head train file.

Return type
str

randomSeed()

Returns
The random seed.

Return type
int

restartFromLastCheckpoint()

Returns
Whether to restart training from last saved checkpoint.

Return type
bool

resultsDirectory()

Returns
The directory to save results.

Return type
str

saveForCpu()

Returns
Whether to save the model in CPU.

Return type
bool

scalingType()

Returns
The type of scaling to the output.

Return type
str

schedulerPatience()

Returns
The patience of the scheduler.

Return type
int

stressKey()

Returns
The key for the stress in the dataset.

Return type
str

stressWeight()

Returns
The weight of the stress loss.

Return type
float

testingDataFilepath()

Returns
The path to the testing data.

Return type
str

trainingDataFilepath()

Returns
The path to the training data.

Return type
str

uniqueString()

Return a unique string representing the state of the object.

useExponentialMovingAverage()

Returns
Whether to use exponential moving average.

Return type
bool

useMultiheadsFinetuning()

Returns
Whether to use multiheads finetuning.

Return type
bool

validationBatchSize()

Returns
The batch size for validation.

Return type
int

validationFraction()

Returns
The fraction of the training data to use for validation.

Return type
float

weightDecay()

Returns
The weight decay (L2 penalty).

Return type
float

Usage Examples

The MACEFittingParameters class is used for choosing settings for training MACE Machine Learned Force Field (MLFF) models[Page 3245, 1][Page 3245, 2] using the *MachineLearnedForceFieldTrainer*. Once trained, the model is automatically converted into a QATK-compatible format, and a *TremoloXCalculator* can be set up using the trained model.

There are three main usage scenarios of the MACEFittingParameters class.

1. Training a MACE model from scratch.

   This is the most straightforward way to train a MACE model where a model is trained only on data provided by the user. The model size and hyperparameters are fully customizable allowing users to define and utilize their own optimal tradeoff between desired model accuracy and performance. Shown in the example below is a simple training script that trains a MACE model from scratch highlighting some of the most important parameters.

```
# Setup MACE parameters for training from scratch
mace_fp = MACEFittingParameters(
    # Name of the model - has to be set
    experiment_name='mace_experiment1',

    # Most important model size parameters (affects accuracy and speed)
    max_l_equivariance=0,
    number_of_channels=64,
    distance_cutoff=4.0*Ang,

    # Weights for the different parts of the loss term
    energy_weight=1.0,
    forces_weight=100.0,
    stress_weight=1.0,
    loss_function_type=MACEParameterOptions.LOSS_TYPE.UNIVERSAL,

    # Most relevant other parameters regarding training setup
    validation_fraction=0.2, # Ratio of training data used for validation
    max_number_of_epochs=200, # Number of epochs to train for
    patience=50, # Number of epochs without any improvement that will cause model␣
↪training to finalize
    batch_size=4, # Generally, higher means quicker training, but too high batch_sizes␣
↪can cause GPU memory errors
    validation_batch_size=4, # Generally, higher means quicker training, but too high␣
↪batch_sizes can cause GPU memory errors
    random_seed=123, # Can be used for experimenting with the influence on different␣
↪data splits
    default_dtype=MACEParameterOptions.DTYPE.FLOAT64, # Can be adjusted for shifting␣
```

(continues on next page)

(continued from previous page)

```
↪tradeoff between model accuracy and training speed
    device=Automatic, # Device can be set explicitly but Automatic is recommended to␣
↪automatically use the GPU if available

    # Include stress if desired and present in the training data
    compute_stress=True,
)

# Setup ML model training object
mlfft = MachineLearnedForceFieldTrainer(
    fitting_parameters=mace_fp,
    training_sets=training_set,
    calculator=calculator
)

# Run the training
mlfft.train()
```

2.Training with Naive Finetuning.

This is the simplest way to finetune a MACE model. The model is trained on a new dataset with model weights and hyperparameters initializing from the values of the pretrained model. Any hyperparameter settings related to the neural network architecture provided in the finetuning MACEFittingParameters object will be overwritten. This ensures that the model size and architecture remain consistent with the pretrained model.

```

# Setup MACE parameters for training with Naive Finetuning
mace_fp = MACEFittingParameters(
    # Name of the model - has to be set
    experiment_name='mace_experiment3',

    # Weights for the different parts of the loss term and type pf loss
    energy_weight=1.0,
    forces_weight=100.0,
    stress_weight=1.0,
    loss_function_type=MACEParameterOptions.LOSS_TYPE.UNIVERSAL,

    # Most relevant parameters regarding training setup
    validation_fraction=0.2, # Ratio of training data used for validation
    max_number_of_epochs=30, # Number of epochs to train for - keep low for finetuning
    batch_size=4, # Generally, higher means quicker training, but too high batch_sizes␣
↪can cause GPU memory errors
    validation_batch_size=4, # Generally, higher means quicker training, but too high␣
↪batch_sizes can cause GPU memory errors
    random_seed=123, # Can be used for experimenting with the influence on different␣
↪data splits
    default_dtype=MACEParameterOptions.DTYPE.FLOAT64, # Can be adjusted for shifting␣
↪tradeoff between model accuracy and training speed
    device=Automatic, # Device can be set explicitly but Automatic is recommended to␣
↪automatically use the GPU if available

    # When using naive finetuning on a custom model (or on a MP foundation model)
```

(continues on next page)

(continued from previous page)

```
    foundation_model_path='/path/to/model/data/on/run/machine/my_previous_mace_model.
↪model',
)

# Setup ML model training object
mlfft = MachineLearnedForceFieldTrainer(
    fitting_parameters=mace_fp,
    training_sets=training_set,
    calculator=calculator
)

# Run the training
mlfft.train()
```

3.Training with Multihead Replay Finetuning.

This is the most advanced way to finetune a MACE model. It is also the recommended approach for finetuning any materials project foundation model[3][4] as this usually leads to more robust and stable models. The model is trained on both a new dataset as well as (some of) the data used for training the original model in separate model heads. The model heads have different readout weights but share the remaining model weights to be optimized during the fitting. This ensures that fitting takes both data sets into account, learning from new data while retaining the knowledge of the original model as best as possible. The model weights and the hyperparameters initialize from the values of the pretrained model. +Any hyperparameter settings related to the neural network architecture provided in the finetuning +MACEFittingParameters object will be overwritten. This ensures that the model size and architecture +remain consistent with the pretrained model.

```

# Setup MACE parameters for training with Multihead Replay Finetuning
mace_fp = MACEFittingParameters(
    # Name of the model - has to be set
    experiment_name='mace_experiment3',

    # Weights for the different parts of the loss term and type of loss
    energy_weight=1.0,
    forces_weight=100.0,
    stress_weight=1.0,
    loss_function_type=MACEParameterOptions.LOSS_TYPE.UNIVERSAL,

    # Most relevant parameters regarding training setup
    validation_fraction=0.2, # Ratio of training data used for validation
    max_number_of_epochs=20, # Number of epochs to train for - keep low for finetuning
    batch_size=4, # Generally, higher means quicker training, but too high batch_sizes␣
↪can cause GPU memory errors
    validation_batch_size=4, # Generally, higher means quicker training, but too high␣
↪batch_sizes can cause GPU memory errors
    random_seed=123, # Can be used for experimenting with the influence on different␣
↪data splits
    default_dtype=MACEParameterOptions.DTYPE.FLOAT64, # Can be adjusted for shifting␣
↪tradeoff between model accuracy and speed
```

(continues on next page)

[3] Ilyes Batatia, Philipp Benner, Yuan Chiang, AlinăM Elena, DávidăP Kovács, Janosh Riebesell, XavierăR Advincula, Mark Asta, Matthew Avaylon, WilliamăJ Baldwin, and others. A foundation model for atomistic materials chemistry. *arXiv preprint arXiv:2401.00096*, 2023.

[4] Mace-foundation models. URL: https://github.com/ACEsuit/mace-foundations.

(continued from previous page)

```
    device=Automatic, # Device can be set explicitly but Automatic is recommended to␣
↪automatically use the GPU if available

    # When using multihead replay finetuning on MP foundation models
    use_multiheads_finetuning=True,
    foundation_model_path='/path/to/model/data/on/run/machine/mace_agnesi_small.model',
    number_of_samples_from_pretrained_head=15000, # Number of samples from the original␣
↪model training data to use for the multihead replay finetuning
    pretrained_head_train_file='mp', # Use 'mp' for MP foundation type model. Use path to␣
↪xyz file with training data for other custom models
    mp_data_path='/path/to/model/data/on/run/machine/mp_traj_combined.xyz',
    mp_descriptors_path='/path/to/model/data/on/run/machine/descriptors.npy',
)

# Setup ML model training object
mlfft = MachineLearnedForceFieldTrainer(
    fitting_parameters=mace_fp,
    training_sets=training_set,
    calculator=calculator
)

# Run the training
mlfft.train()
```

The example scripts, including a full training setup with *MachineLearnedForceFieldTrainer* and *TremoloXCalculator* for retrieving the trained models as calculators, are available for downloading:

1. `mace_example1.py`
2. `mace_example2.py`
3. `mace_example3.py`

Notes

The MACEFittingParameters class provides an interface for training MACE MLFFs[Page 3245, 1][Page 3245, 2], utilizing the open source *mace-torch* package developed by the authors. The framework provides a multitude of possible settings aiding the creation of accurate and performant MLFFs.

In order to use any MACE training functionality, it is assumed that at least one *TrainingSet* is available with precomputed energies, forces, and, if desired, stress. The calculator used for calculating the `TrainingSet` data is similarly assumed to be either attached to the TrainingSet or to be defined in the `calculator` parameter of the *MachineLearnedForceFieldTrainer* class - unless isolated atom energies are passed explicitly in the MACEFittingParameters. **Importantly**, the calculator should use spin polarized calculations in order to create accurate isolated atom energy reference values. If previous MACE model training data is stored in XYZ files, these files can be explicitly passed into the MACEFittingParameters object using the `training_data_filepath` and `testing_data_filepath` arguments. However, this requires the isolation energies to be explicitly provided through the `isolated_atom_energies` arguments. In QuantumATK, it is recommended to supply TrainingSets to the *MachineLearnedForceFieldTrainer* class for a smooth training experience.

The MACEFittingParameters class is a simplified version of the complete set of MACE training parameters available in the open source package with more self-explanatory parameter namings and a reduced number of total parameters

for simplifying general user interaction. MACE training parameters that have been left out of the MACEFittingParameters class, but that are available in the open source training framework, can, however, still be passed into the MACEFittingParameters object by supplying them as key-value pairs in a python dictionary to be inserted via the `additional_parameters` parameter.

Most of the parameters have default values corresponding to the defaults in the open source package, but some have been adapted in the `MACEFittingParameters` class for ease of use and more appropriate settings for most QuantumATK users. All used default values are documented in the parameter list above.

The `mp_descriptors_path` and `mp_data_path` parameters have been introduced in order to avoid automatic downloading of the MACE descriptors and training data files containing Materials Project data for the assembly of the training set used for training MP foundation models. In order to Multihead Replay Finetuning with MP foundation models, these parameters therefore have to be set with local paths to locations where the MACE descriptors and training data files are stored on the cluster/location where the training will be performed - see the Multihead Replay Finetuning example script above. These files therefore have to be downloaded/transferred to the compute file system from the MACE-foundations repository[Page 3256, 4] in order to make finetuning from MP foundation models possible. Here, it is also possible to check which models are currently recommended to start finetuning from.

The MP foundation models[Page 3256, 3], their underlying training data[5], and (in the case of MP foundation models) their descriptors file, all of which are required for performing Multihead Replay Finetuning, are available via the mace-foundations repo of ACESuit[Page 3256, 4]. **It should be noted**, that the training data was calculated with VASP calculators and that total compatibility with QuantumATK-native calculators is not guaranteed. This is therefore a prime use case for Multihead Replay Finetuning, which supports heads using different DFT definitions. The total energies may differ, but derivative quantities such as forces and stress for similar levels of theory show signs of good agreement (for same level of theory). This means that in terms of absolute energies, pretrained models may have to somewhat relearn different energy scales, but the inter-atomic relationships learned by a foundation model are still a strong foundation for continued learning although the level/type of theory is updated. In practice, the different model heads will have separate weights for the readout layer, but otherwise share the remaining model weights.

After training a model via the general training flow outlined above, the resulting model can be used directly with the *TremoloXCalculator*, as evident in the downloadable example scripts, and is loadable via the `MachineLearnedForceFieldCalculator` block in the Workflow Builder. While the main use case within QuantumATK utilizes the `.qatkpt` output file, the regular MACE `.model` output file will remain available after training.

GPU acceleration of MACE training using CUDA is supported. The default way to activate GPU acceleration is by checking the Enable proprietary GPU acceleration option in the **Job Manager** or by using the `atkpython_gpu` command instead of `atkpython`, as described in the *technical notes*.

To utilize multiple GPUs for training, enable the `distributed_training` parameter and ensure that the training script runs with a single process per GPU.

MACEParameterOptions

class MACEParameterOptions

Container class for MACE parameters with choosable options.

class DEVICE

Device options for MACE training:

- MACEParameterOptions.DEVICE.AUTOMATIC
- MACEParameterOptions.DEVICE.CPU
- MACEParameterOptions.DEVICE.GPU

[5] Bowen Deng, Peichen Zhong, KyuJung Jun, Janosh Riebesell, Kevin Han, ChristopherăJ Bartel, and Gerbrand Ceder. Chgnet as a pretrained universal neural network potential for charge-informed atomistic modelling. *Nature Machine Intelligence*, 5(9):1031–1041, 2023.

class DTYPE

Data type options for MACE training:

•MACEParameterOptions.DTYPE.FLOAT32

•MACEParameterOptions.DTYPE.FLOAT64

class ERROR_MEASURE

Error measure options for model evaluation for MACE training:

•MACEParameterOptions.ERROR_MEASURE.PER_ATOM_RMSE

•MACEParameterOptions.ERROR_MEASURE.TOTAL_RMSE

•MACEParameterOptions.ERROR_MEASURE.PER_ATOM_RMSE_STRESS_VIRIALS

•MACEParameterOptions.ERROR_MEASURE.PER_ATOM_MAE_STRESS_VIRIALS

•MACEParameterOptions.ERROR_MEASURE.PER_ATOM_MAE

•MACEParameterOptions.ERROR_MEASURE.TOTAL_MAE

•MACEParameterOptions.ERROR_MEASURE.DIPOLE_RMSE

•MACEParameterOptions.ERROR_MEASURE.DIPOLE_MAE

•MACEParameterOptions.ERROR_MEASURE.ENERGY_DIPOLE_RMSE

class LOSS_TYPE

Loss function options for MACE training:

•MACEParameterOptions.LOSS_TYPE.ENERGY_FORCES

•MACEParameterOptions.LOSS_TYPE.WEIGHTED

•MACEParameterOptions.LOSS_TYPE.FORCES_ONLY

•MACEParameterOptions.LOSS_TYPE.VIRIALS

•MACEParameterOptions.LOSS_TYPE.STRESS

•MACEParameterOptions.LOSS_TYPE.DIPOLE

•MACEParameterOptions.LOSS_TYPE.HUBER

•MACEParameterOptions.LOSS_TYPE.UNIVERSAL

•MACEParameterOptions.LOSS_TYPE.ENERGY_FORCES_DIPOLE

class MODEL_TYPE

Model type options for MACE training:

•MACEParameterOptions.MODEL_TYPE.BOTNET

•MACEParameterOptions.MODEL_TYPE.MACE

•MACEParameterOptions.MODEL_TYPE.SCALE_SHIFT_MACE

•MACEParameterOptions.MODEL_TYPE.SCALE_SHIFT_BOTNET

•MACEParameterOptions.MODEL_TYPE.ATOMIC_DIPOLES_MACE

•MACEParameterOptions.MODEL_TYPE.ENERGY_DIPOLES_MACE

class SCALING_TYPE

Scaling type options for MACE training:

•MACEParameterOptions.SCALING_TYPE.STD_SCALING

•MACEParameterOptions.SCALING_TYPE.RMS_FORCES_SCALING

•MACEParameterOptions.SCALING_TYPE.NO_SCALING

MDMeasurement

class MDMeasurement(*measurements*, *call_interval=None*)

A measurement hook used to save common MD scalar quantities. Should be used as measurement_hook in MolecularDynamics.

Parameters

•**measurements** (*list of type str*) – A list of measurements to save. Valid measurement names include:

–kinetic_energy

–potential_energy

–volume

–density

–temperature

–pressure

–stress

–strain

–dipole

–charge_current

The measurements, pressure, stress, and strain represent the average of the diagonal components of these tensors. Individual tensor components can be obtained by appending an underscore and two Cartesian directions. For example, the pressure along the xy direction is represented with pressure_xy and the strain along the x direction is strain_xx. If the full tensor is desired, then _tensor can be appended. For example strain_tensor or pressure_tensor. The same applies for vectors, with components referenced by x, y and z. The complete vector is referenced by the postfix _vector

•**call_interval** (*int*) – The interval at which the hook functions call method should be invoked during an MD simulation. **Default:** 1 (called at every step).

callInterval()

Returns
The call interval of this hook function.

Return type
int

measurementNames()

Returns
The names of all measured properties.

Type
list of type str

measurements()

Returns
The measurements.

Return type
list of str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run a 6 ps NVT MD simulation, with a strain rate of 0.25% per ps, while measuring the stress and strain at every MD step. These measurements are then used to plot a stress-strain curve.

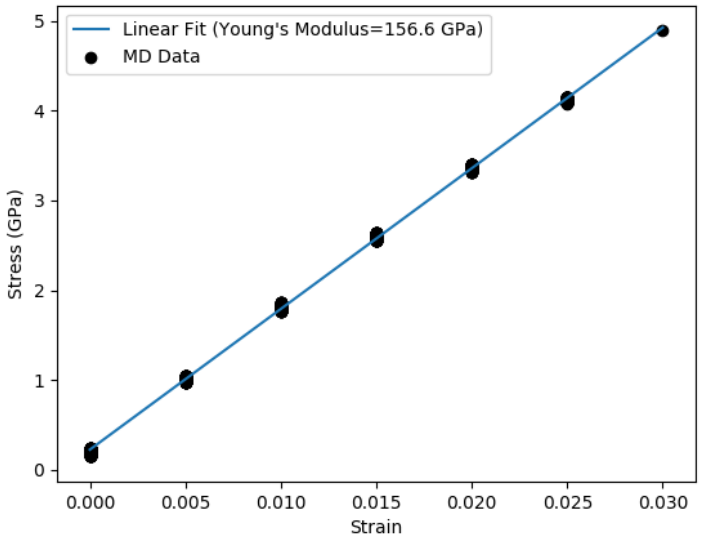
```
# Define a strain configuration hook.
strain_hook = StrainConfigurationHook(
    strain_direction='x',
    strain_rate=0.005 / ps,
    strain_interval=1000,
)

measurement_names = ['stress_xx', 'strain_xx']

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='stress_strain.hdf5',
    steps=6000,
    log_interval=1000,
    pre_step_hook=strain_hook,
    measurement_hook=MDMeasurement(measurement_names, call_interval=1),
    method=method,
)

bulk_configuration = md_trajectory.lastImage()

# Get stress strain values.
times, strains = md_trajectory.measurement('strain_xx')
times, stresses = md_trajectory.measurement('stress_xx')

# Perform linear fit.
polynomial_coefficients = numpy.polyfit(strains, stresses.inUnitsOf(GPa), 1)
youngs_modulus = polynomial_coefficients[0] * GPa

# Evaluate polynomial to plot best fit line.
stresses_fit = numpy.polyval(polynomial_coefficients, strains)

# Plot the fit.
import pylab
pylab.plot(
    strains,
```

(continues on next page)

(continued from previous page)

```
    stresses_fit,
    label="Linear Fit (Young's Modulus=%.1f GPa)" % youngs_modulus,
)

# Plot MD data.
pylab.scatter(
    strains,
    stresses.inUnitsOf(GPa),
    label='MD Data',
    color='k',
)

# Setup plot.
pylab.xlabel('Strain')
pylab.ylabel('Stress (GPa)')
pylab.legend()

# Save to file.
pylab.savefig('stress_strain.png')
```

The resulting plot will look like:



The full script can be downloaded: `stress_strain.py`

Notes

- All measurements are stored on the *MDTrajectory* object and can be accessed using the `measurement` method. This method returns a tuple of the MD times associated with the measurement and the measurements itself.
- Custom measurements can be defined using the `measurement_hook` argument of *MolecularDynamics*. For additional details see *MolecularDynamics notes*.
- Measurements can be visualized using the MovieTool with the MDTrajectory.
- When a MDTrajectory is read on the LabFloor all measurements will be displayed. Custom plots can be created using the Data Plot plugin.

MDSaddleSearch

class MDSaddleSearch(*temperature*, *saddle_search_parameters=None*, *constraints=None*, *neb_log_filename_prefix=None*, *active_learning=None*)

Saddle search using high temperature molecular dynamics followed by a NEB optimization.

Parameters

- **temperature** (A *PhysicalQuantity* with temperature units) – The temperature for the MD search.
- **saddle_search_parameters** (SaddleSearchParameters) – The parameters for the saddle search.
- **constraints** (*list of ints*) – The atoms that should be constrained during the process search. **Default:** []
- **neb_log_filename_prefix** (*str | None*) – The log filename prefix to use during NEB optimization. None means to use stdout. **Default:** None
- **active_learning** (ActiveLearningSimulation | *None*) – The active learning object, if specified, otherwise None.

eigenvector()

Returns
The eigenvector as a (nx3) numpy array

Return type
numpy.array

eigenvectorGuess()

Returns
The initial guess of the eigenvector as a (nx3) numpy array.

Return type
numpy.array

escapeTime()

Returns
The estimated escape time.

Return type
PhysicalQuantity of type time

finalConfiguration()

Returns
The final configuration.

Return type
AtomicConfiguration

`initialNudgedElasticBand()`

Returns
The initial guess of the NEB.

Return type
NudgedElasticBand

`mdTrajectories()`

Returns
The MDTrajectories

Return type
list

`optimizedNudgedElasticBand()`

Returns
The optimized NEB.

Return type
NudgedElasticBand

`reset()`

Reset the direction generator.

`run(`*configuration*`)`

Run the MD Saddle Search calculation.

`saddleConfiguration()`

Returns
The saddle configuration.

Return type
AtomicConfiguration

`saddleConfigurationGuess()`

Returns
The initial guess of the saddle configuration.

Return type
AtomicConfiguration

`setRandomSeed(`*seed*`)`

Set the random seed for the initial direction generator.

Parameters
seed (*int*) – The random seed.

`setRedundantSearchFunction(`*is_search_redundant*`)`

Parameters
is_search_redundant (*callable that takes a configuration and energy and returns the state id for a configuration or returns None if the id is not known*) – A function to check if the product state has already been found before. This allows for early termination. If it is set to None, then this check will not be performed. **Default:** None

Notes

MDSaddleSearch is one of the three saddle search methods available for use in *AdaptiveKineticMonteCarlo*. It uses high temperature molecular dynamics (MD) simulations to locate saddle points[1]. The high temperature ensures that processes with a wide range of barriers are sampled. During the simulation, the current configuration is periodically minimized. If the minimized structure is different from the initial configuration, a reaction has occurred. If no reaction has occurred, the simulation is continued. If a reaction has occurred, the saddle point is located by running a *NudgedElasticBand* (NEB) optimization. The converged NEB is then used as a starting point for a minimum-mode following saddle optimization algorithm.

If a previously discovered minimum is found the saddle search terminates early.

MDTrajectory

class MDTrajectory(*path=None*, *group_name=None*, *overwrite=None*)

Constructor for the MDTrajectory that contains the configurations from a *MolecularDynamics()* run.

Parameters

- **path** (*str | None*) – The path to the HDF5 file. If the path is None then all data will be stored in memory and no file will be written. **Default:** None
- **group_name** (*str | None*) – The name of the HDF5 group to store the data in. If the value is None then a new group will be automatically made. **Default:** None
- **overwrite** (*bool*) – If not False, any existing hdf5 group named group_name will be overwritten. **Default:** False

calculator()

Returns
The calculator if one is available, otherwise None.

Return type
Calculator | None

cells(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The cell vectors of the images or None if the trajectory length is zero or the configuration is a MoleculeConfiguration.

Return type
PhysicalQuantity of type length | None

concatenate(**trajectories*, *interval=1*, *path=None*, *object_id=None*)

Combine several MDTrajectory objects into a new MDTrajectory.

Parameters

- **trajectories** (*list of MDTrajectory objects*) – List of MDTrajectory objects to concatenate to this trajectory. The trajectories must be continuous, i.e. the last images of the previous trajectory must match the first image of the next trajectory.
- **interval** (*int*) – Only keep every *interval* steps.

[1] SamuelăT. Chill and Graeme Henkelman. Molecular dynamics saddle search adaptive kinetic Monte Carlo. *The Journal of Chemical Physics*, 140:214110, 2014. doi:10.1063/1.4880721.

•**path** (*str | None*) – The path to the HDF5 file. If the path is None then all data will be stored in memory and no file will be written. **Default:** None

•**object_id** – The object id to use. If the value is None then an id will be automatically made. **Default:** None

Returns
The concatenated MDTrajectory.

Return type
MDTrajectory

coordinates(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The Cartesian coordinates of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type length | None

elements()

Returns
The elements of the images as a list of length n or None if the trajectory length is zero.

Return type
list of type *PeriodicTableElement*

forces(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The forces of the images. If the trajectory length is zero or the forces were not written during the MolecularDynamics run then None is returned.

Return type
PhysicalQuantity of type energy/length | None

image(*image_index*, *set_calculator=True*, *set_velocities=True*, *recycle_configuration=False*)

Parameters

•**image_index** (*int*) – The index of the configuration that should be extracted.

•**set_calculator** (*bool*) – If True, the calculator is set on the returned image. This can be disabled to increase the performance when iterating over images, because the overhead of copying and setting up a new calculator can be large. **Default:** True

•**set_velocities** (*bool*) – If True, the velocities are set on the returned image. This can be disabled to increase performance. **Default:** True

•**recycle_configuration** (*bool*) – If True, the same configuration instance (with updated coordinates/cell) is returned each time image is called. This can increase performance, when iterating through images, but only allows for one configuration to exist at a time. **Default:** False

Returns
A copy of the configuration with the image_index.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

kineticEnergies(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The kinetic energies of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type energy | None

lastImage()

Returns
The last image in the trajectory with the calculator attached.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

lastStep()

Returns
The step number of the last step.

Return type
int

lastTime()

Returns
The time associated with the last step.

Return type
PhysicalQuantity of type time

length()

Returns
The number of images in the trajectory.

Return type
int

logInterval()
Deprecated: The method logInterval will be deprecated. Please use trajectoryInterval instead.

Returns
The interval at which snapshots are saved.

Return type
int

masses()

Returns
The masses of the atoms with shape (num_atoms,) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type mass

measurement(*name*, *idx=Ellipsis*)

Parameters

- **name** (*str*) – The name of the measurement.
- **idx** (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
A 2-tuple of the time and the measurement.

Return type
tuple

measurementNames()

Returns
The names of the measurements.

Return type
list of type str

metatext()

Returns
The metatext of the object or None if no metatext is set.

Return type
str | None

nlinfo()

Returns
The trajectory information.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)
Write a text description of the trajectory to the stream.

Parameters
stream (*file-like object*) – The stream to write to. **Default:** `sys.stdout`

numberOfAtoms()

Returns
The number of atoms in the system.

Return type
int

potentialEnergies(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The potential energies of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type energy | None

pressureTensors(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The pressure tensors of the images as a vector with shape (m, 3, 3) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type pressure | None

pressures(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The pressure values of the images as a vector with shape (m) or None if the trajectory length is zero.

Return type
PhysicalQuantity of type pressure | None

prune(*interval*, *path=None*, *object_id=None*)

Copy this MDTrajectory and reduce the number of images.

Parameters

- **interval** (*int*) – Only keep every *interval* steps.
- **path** (*str | None*) – The path to the HDF5 file. If the path is None then all data will be stored in memory and no file will be written. **Default:** None
- **object_id** – The object id to use. If the value is None then an id will be automatically made. **Default:** None

Returns
The pruned MDTrajectory.

Return type
MDTrajectory

reservoirPressures(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The reservoir pressure of the MD barostat. If the trajectory length is zero or this trajectory was not generated from a NPT simulation then None is returned.

Return type
PhysicalQuantity of type temperature

reservoirTemperatures(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The reservoir temperature of the MD thermostat. If the trajectory length is zero or this trajectory was generated from a NVE simulation then None is returned.

Return type
PhysicalQuantity of type temperature

setMetatext(*text*)

Set a given metatext string on the object.

Parameters
text (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

steps(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The MD step numbers of the images or None the trajectory length is zero.

Return type
array of type int | None

stresses(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The stresses of the images. If the trajectory length is zero or the stresses were not written during the MolecularDynamics run then None is returned. None is also returned if the configuration is a MoleculeConfiguration.

Return type
PhysicalQuantity of type pressure | None

temperatures(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The instantaneous kinetic temperatures of the images. If the trajectory length is zero or the temperatures were not written during the MolecularDynamics run then None is returned.

Return type
PhysicalQuantity of type temperature

timeInterval()

Returns
The time interval between two images.

Return type
PhysicalQuantity of type time

timeStep()

Returns
The MD time step of the simulation.

Return type
PhysicalQuantity of type time

times(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The simulation times of the images or None the trajectory length is zero.

Return type
PhysicalQuantity of type time | None

trajectoryInterval()

Returns
The interval at which snapshots are saved.

Return type
int

uniqueElements()

Returns
The unique elements of the images or None if the trajectory length is zero.

Return type
list of type *PeriodicTableElement*

velocities(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** All indices

Returns
The velocities of the images. If the trajectory length is zero or the velocities were not written during the MolecularDynamics run then None is returned.

Return type
PhysicalQuantity of type length/time

volumes(*idx=Ellipsis*)

Parameters
idx (*int | slice*) – The index or slice to return. **Default:** Ellipsis

Returns
The volumes of the unit cells. If this was a MD simulation of molecules, then None will be returned.

Return type
PhysicalQuantity of type length**3 | None

writeToXYZ(*filename*)

Make a xyz dump of the whole trajectory.

Parameters
filename (*str*) – The name of the file to write to.

Usage Examples

Perform four Molecular Dynamics simulations with different time steps and save their trajectories in the file `trajectory.nc`.

```
# Define a benzene ring
cartesian_coordinates = [[ 0.0     ,  1.399795, 0.0],
                         [ 1.212253,  0.69976 , 0.0],
                         [ 1.212253, -0.69976 , 0.0],
                         [ 0.0     , -1.399795, 0.0],
                         [-1.212253, -0.69976 , 0.0],
```

(continues on next page)

(continued from previous page)

```
                          [-1.212253,  0.69976 , 0.0],
                          [ 0.0     ,  2.500678, 0.0],
                          [ 2.165671,  1.250363, 0.0],
                          [ 2.165671, -1.250363, 0.0],
                          [ 0.0     , -2.500678, 0.0],
                          [-2.165671, -1.250363, 0.0],
                          [-2.165671,  1.250363, 0.0]]*Angstrom

molecule_configuration = MoleculeConfiguration(
    elements=[Carbon,]*6+[Hydrogen,]*6,
    cartesian_coordinates=cartesian_coordinates
    )

# Attach a calculator
molecule_configuration.setCalculator(BrennerCalculator())

initial_velocity = MaxwellBoltzmannDistribution(
        temperature=300.0*Kelvin
    )

length_of_simulation = 20*femtoSecond

# Generate trajectories for different time steps
for time_step in [0.1, 0.5, 1, 2]*femtoSecond:

    method = NVEVelocityVerlet(
        time_step=time_step,
        initial_velocity=initial_velocity
        )

    # Generate trajectory
    molecular_dynamics = MolecularDynamics(
        molecule_configuration,
        trajectory_filename='trajectory.nc',
        steps=int(length_of_simulation/time_step),
        log_interval=max(1,int(2*femtoSecond/time_step)),
        method=method
        )
```

mdrun.py

Plot the total energy as function of time for the trajectories in the file `trajectory.nc`.

```
import pylab

# Read in a list with the trajectories
trajectories = nlread('trajectory.nc', MDTrajectory)

pylab.figure()
for trajectory in trajectories:
    # Get the time axis of the trajectory
    t = trajectory.times().inUnitsOf(femtoSecond)
```

(continues on next page)

(continued from previous page)

```
    # Calculate the time step
    time_step = t[-1]/trajectory.steps()[-1]

    # Get the energies
    epot = trajectory.potentialEnergies().inUnitsOf(eV)
    ekin = trajectory.kineticEnergies().inUnitsOf(eV)

    # Plot the data
    pylab.plot(t,epot+ekin, label='timestep=' + str(time_step)+' fs')
    pylab.xlabel('time (fs)')
    pylab.ylabel('total energy (eV)')
    pylab.legend()

pylab.show()
```

mdplot.py

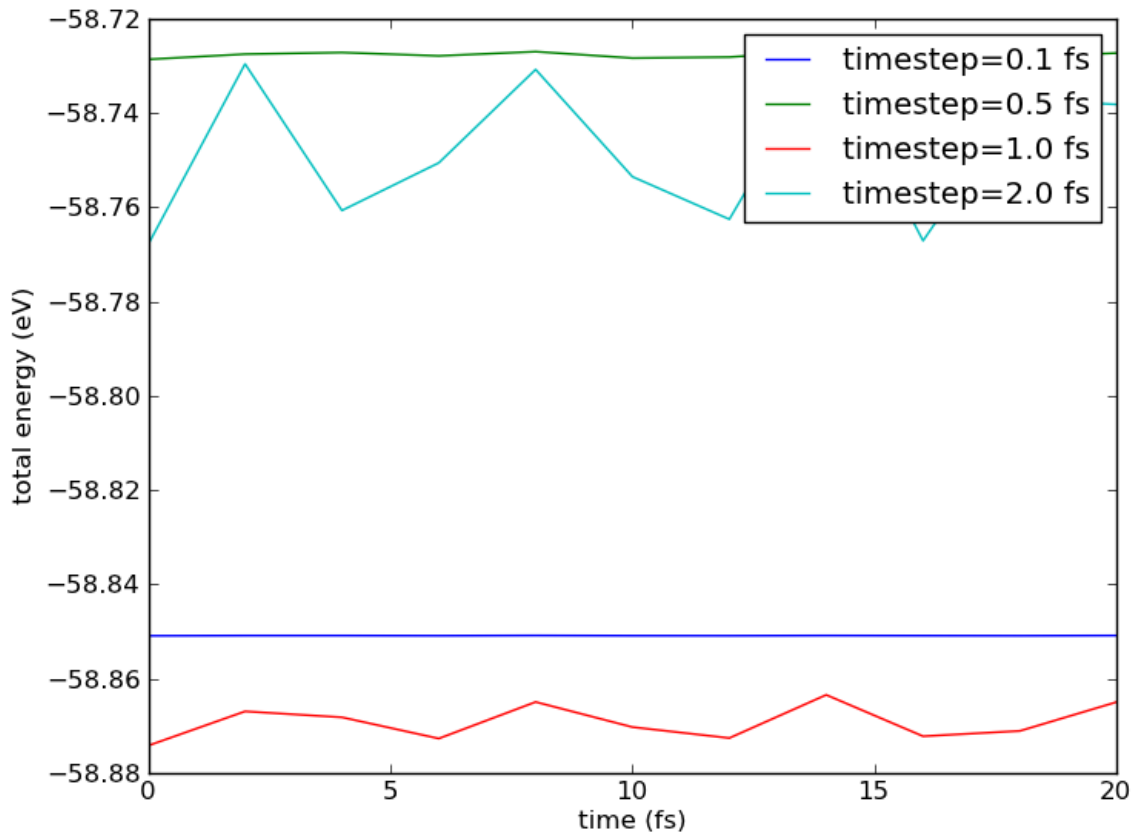


Fig. 4.32: The total energy as function of the simulation time in MD runs with different time steps.

MTPErrorPredictionParameters

class MTPErrorPredictionParameters(*predict_energy_error=None*, *predict_forces_error=None*, *check_interval=None*, *write_atomic_error_estimates=None*, *stop_simulation_energy_threshold=None*, *stop_simulation_force_threshold=None*)

Parameters for specifying MTP error prediction MD simulation settings.

Parameters

- **predict_energy_error** (*bool*) – Whether the energy error should be predicted. **Default:** True
- **predict_forces_error** (*bool*) – Whether the forces error should be predicted. **Default:** True
- **check_interval** (*int*) – The interval at which prediction errors are checked and saved. If atomic error estimates are used, this should be the same as the trajectory interval in the MolecularDynamicsErrorPrediction simulation. **Default:** 100
- **write_atomic_error_estimates** (*bool*) – Whether atomic error estimates should be given for forces. **Default:** True
- **stop_simulation_energy_threshold** (*PhysicalQuantity of type energy | None*) – The energy threshold value at which the MDErrorPrediction simulation is stopped. **Default:** None
- **stop_simulation_force_threshold** (*PhysicalQuantity of type energy / length | None*) – The force threshold value at which the MDErrorPrediction simulation is stopped. **Default:** None

checkInterval()

Returns
The interval at which prediction errors are checked and saved.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

predictEnergyError()

Returns
Whether energy error is predicted.

Return type
bool

predictForcesError()

Returns
Whether forces error is predicted.

Return type
bool

stopSimulationEnergyThreshold()

Returns
The energy threshold value at which the MDErrorPrediction simulation is stopped.

Return type
PhysicalQuantity of type energy | None

`stopSimulationForceThreshold()`

Returns
The force threshold value at which the MDErrorPrediction simulation is stopped.

Return type
PhysicalQuantity of type energy / length | None

`uniqueString()`

Return a unique string representing the state of the object.

`writeAtomicErrorEstimates()`

Returns
Whether atomic error estimates should be given for forces.

Return type
bool

Notes

An example showing how *MTPErrorPredictionParameters* can be utilized for error prediction is shown in the *MolecularDynamicsErrorPredictionHook* class documentation.

MTPPotential

`class MTPPotential(`*file*, *suppress_intercept=False*, *group_name=''*`)`

Constructor of the potential

Parameters

- **file** (*str*) – The name of a file containing the data for a fitted moment tensor potential.
- **suppress_intercept** (*Bool*) – Suppress the intercept of the linear model. This will cause the energy to deviate from the reference by some system-independent additive constant, but will ensure that the energy of the empty system is 0.
- **group_name** (*str*) – Optional name of a hdf5-group in file containing the data for a fitted moment tensor potential, if the data is not in the root-group of file

`classmethod getAllParameterNames()`

Return the names of all used parameters as a list.

`getAllParameters()`

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

`static getDefaults()`

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

`getParameter(`*parameterName*`)`

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Create a calculator for use in calculations on SiC. The parameters for the MTP are contained in the file `SiC_MTP_Parameters.mtp`.

```
# Create the TremoloX potential set
potential_set = TremoloXPotentialSet(name='SiC MTP Potential')

# Add the silicon particle type to the potential set
potential_set.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    atomicNumber=14,
))

# Add the carbon particle type to the potential set
potential_set.addParticleType(ParticleType(
    symbol='C',
    mass=12.0107*atomic_mass_unit,
    charge=None,
    atomicNumber=6,
))

# Add the MTP potential parameters contained in the SiC_MTP_Parameters.mtp file.
mtp_potential = MTPPotential(
    file='SiC_MTP_Parameters.mtp',
    suppress_intercept=False,
    group_name='',
)
potential_set.addPotential(mtp_potential)

# Create the Forcefield calculator and add it to the configuration.
calculator = TremoloXCalculator(parameters=potential_set)
```

mtp_calculator_example.py

Notes

This class implements the moment tensor potential (MTP) framework[1]. Moment tensor potentials are a class of systematically improvable machine learning (ML) potentials. They require a parameter file, which has been trained on a suitable set of ab-initio calculations of configurations containing representative atomic environments.

A Forcefield calculator that uses an MTP can be created using a potential set that contains a *MTPPotential* object. When added to a potential set the *MTPPotential* class takes as an argument the file name of the file that contains the parameters for the MTP. These files typically have the extention `.mtp`. Parameter files can be created using MTP training tools such as the *MomentTensorPotentialTraining* class. In addition to the MTP parameters, the potential set must also include a particle type definition for each element contained within the MTP.

The above example shows how to create a Forcefield calculator for calculations on SiC. Here the MTP contains parameters for both Si and C in the file `SiC_MTP_Parameters.mtp` file.

MachineLearnedForceFieldTrainer

class MachineLearnedForceFieldTrainer(*fitting_parameters*, *training_sets=None*, *calculator=None*, *train_test_split=None*, *random_seed=None*)

Class for training a machine learned force field.

Parameters

- **fitting_parameters** (BaseMLFFFittingParameters) – The parameters for the training.
- **training_sets** (*TrainingSet* | *Table* | sequence of [*TrainingSet*] | None) – The list of training sets to use for training. **Default:** None
- **calculator** (*Calculator* | None) – The calculator to use for calculating the constant terms if applicable for the model. If None, the calculator of the training set is used. **Default:** None
- **train_test_split** (*float*) – The fraction of the training set to use for training. The rest is used for testing. Must be a float between 0 and 1. If set to 1, the entire training set is used for training. **Default:** 0.9
- **random_seed** (*int*) – The random seed used for splitting the data into training and testing data. **Default:** Generated automatically.

train()

Train the machine learned force field.

Usage Examples

In order to train a MACE model using the `MachineLearnedForceFieldTrainer`, the general approach is to set up a fitting parameters object for the model type to train, to load in the training data, and to load/configure an appropriate calculator for calculating isolated atom energies if required. Additionally, non-default `train_test_split` and `random_seed` values for controlling the way the input data is split can be created. These objects and values are passed to the `MachineLearnedForceFieldTrainer` class which will start the training process by simply calling the `train()` method.

The example below shows how a MACE model is trained using the `MachineLearnedForceFieldTrainer` class.

[1] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

```
# TrainingSet with precomputed Energy Forces (and Stress if present and desired) data.␣
↪This can be
# one or more TrainingSets
training_set = nlread('My_TrainingSet_Data.hdf5', TrainingSet)

# Fetch the calculator from the training set. If not present, it should be retrieved from
# elsewhere/set up analogously to calculate isolated atom energies
calculator = training_set.calculator()

# Setup model specific fitting parameters object for the training
mace_fp = MACEFittingParameters(
    # Name of the model - has to be set
    experiment_name='mace_experiment',
    # Other parameters can be set as desired
    # number_of_channels=32,
    # ...
)

# Setup ML model training object
mlfft = MachineLearnedForceFieldTrainer(
    fitting_parameters=mace_fp,
    training_sets=training_set,
    calculator=calculator,
    # Optional parameters can be set as desired
    train_test_split=0.8,
    random_seed=1234,
)

# Run the training
mlfft.train()
```

This example script is available for download: `mlfft_example.py`

Notes

- The `MachineLearnedForceFieldTrainer` class is used to train machine-learned force fields (MLFFs) in QuantumATK. The intention of it is to offer a model agnostic trainer object that can train various kinds of models, where the type of the `fitting_parameter` object will determine how the training is conducted. It can train MLFF models that have FittingParameters objects that fulfil certain setup requirements. This does not yet include the *MomentTensorPotentialFittingParameters*.
- The `MachineLearnedForceFieldTrainer` class is designed to be used with only the *TrainingSet* class for training data input. That means that training data has to be converted into that format either by using the conversion/export as utility in the Data View in the GUI or by directly converting/wrapping other data storage types into TrainingSet objects.
- The `MachineLearnedForceFieldTrainer` class is similarly designed to only be used for the actual training step of a MLFF model. Gathering training data is not part of the class and has to be done separately before starting the training. Since the training process can take different training various training paths when multiple types of models are going to be supported, the trainer object also does not support a general model evaluation step. The evaluation of the trained model therefore has to be done separately after training is concluded and a MLFF model file has been saved.

MaddenDispersivePotential

class MaddenDispersivePotential(*particleType1*, *particleType2*, *b*, *C6*, *C8*, *kmax6=6*, *kmax8=8*, *c6=1.0*, *c8=1.0*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **b** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **c6** (*float*) – Potential parameter.
- **c8** (*float*) – Potential parameter.
- **C6** (*PhysicalQuantity of type energy * length**6*) – Potential parameter.
- **C8** (*PhysicalQuantity of type energy * length**8*) – Potential parameter.
- **kmax6** (*int*) – Potential parameter.
- **kmax8** (*int*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Madden-Potential potential for an $MgSiO_3$ perovskite structure by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
```

(continues on next page)

(continued from previous page)

```
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.25  ],
                          [ 0.0144,  0.9436,  0.75  ],
                          [ 0.4856,  0.4436,  0.75  ],
                          [ 0.5144,  0.5564,  0.25  ],
                          [ 0.    ,  0.5   ,  0.    ],
                          [ 0.    ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.    ],
                          [ 0.1015,  0.4673,  0.25  ],
                          [ 0.8985,  0.5327,  0.75  ],
                          [ 0.6015,  0.0327,  0.75  ],
                          [ 0.3985,  0.9673,  0.25  ],
                          [ 0.6962,  0.2983,  0.0524],
                          [ 0.6962,  0.2983,  0.4476],
                          [ 0.3038,  0.7017,  0.5524],
                          [ 0.1962,  0.2017,  0.5524],
                          [ 0.8038,  0.7983,  0.4476],
                          [ 0.8038,  0.7983,  0.0524],
                          [ 0.1962,  0.2017,  0.9476],
                          [ 0.3038,  0.7017,  0.9476]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potential_set = TremoloXPotentialSet(name = 'Jahn_AlCaMgOSi_2007')

# Define and add the general particle types.
potential_set.addParticleType(
    ParticleType(
        symbol='Si',
        mass=28.0855*atomic_mass_unit,
        charge=4.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
```

(continues on next page)

(continued from previous page)

```
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='Mg',
        mass=24.305*atomic_mass_unit,
        charge=2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='O',
        mass=15.9994 * atomic_mass_unit,
        charge=-2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)

# Define and add the Madden-Ewald-option.
option = MaddenEwaldOption(kappa=0.5, r_cut=7.5*Angstrom)
potential_set.addOption(option)

# Define and add the dispersive potential.
potential = MaddenDispersivePotential(
    particleType1='Mg',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='Si',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
potential = MaddenDispersivePotential(
    particleType1='O',
    particleType2='O',
    b=2.718371*1/Ang,
    C6=26.51346*eV*Ang**6,
    C8=142.77637*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)

# Define and add the Madden particle type for oxygen.
potential = MaddenParticle(
    particleType='O',
    dipolePolarization=True,
    quadrupolePolarization=True,
    alpha=8.7671*elementary_charge**2*Bohr**2 / Hartree,
    B=0*elementary_charge**3*Bohr**4 / Hartree**2,
    C=11.5124*elementary_charge**2*Bohr**4 / Hartree ,
    dipoleDistortion=True,
    quadrupoleDistortion=True,
    D=0.49566*Hartree,
    beta=1.2325*Bohr**-1,
    zeta=0.89219*Bohr**-1,
    eta=4.3646*Bohr**-1,
    gamma=1*Hartree
)
potential_set.addPotential(potential)

# Define and add the repulsive potential.
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Mg',
    A=1127.6125*eV,
    a=3.1346777*1/Ang,
    B=1615675.9*eV,
    b=7.3914748*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Si',
    A=1177.627*eV,
    a=2.9135798*1/Ang,
    B=1196266.8*eV,
    b=7.5233777*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
```

(continues on next page)

(continued from previous page)

```
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='O',
    A=0.0*eV,
    a=0.0*1/Ang,
    B=0.0*eV,
    b=0.0*1/Ang,
    C=14530.879*eV,
    c=5.03763186568*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Define and add the polarization damping potential.
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Mg',
    bD=4.1853654*1/Ang,
    cD=2.828,
    kmaxD=4,
    bQ=3.6471714*1/Ang,
    cQ=1.3317,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Si',
    bD=4.015668*1/Ang,
    cD=1.5933,
    kmaxD=4,
    bQ=3.6974382*1/Ang,
    cQ=1.0592,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Set the Coulomb solver.
potential_set.setCoulombSolver(
  CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2)
)

# Set up the TremoloX-calculator with the constructed Madden-potential.
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

(continues on next page)

(continued from previous page)

```
# Optimize the structure and the cell.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.01*eV/Ang,
        max_stress=0.001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
        )
nlprint(bulk_configuration)
```

See also ref.maddencommon.notes.

MaddenEwaldOption

class MaddenEwaldOption(*kappa*, *r_cut*)

Constructor of the option.

Parameters

- **kappa** (*PhysicalQuantity of type length**-1*) – Splitting parameter of the Ewald summation.
- **r_cut** (*PhysicalQuantity of type length*) – Real-space cutoff of the Ewald summation.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Madden-Potential potential for an $MgSiO_3$ perovskite structure by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.25  ],
                          [ 0.0144,  0.9436,  0.75  ],
                          [ 0.4856,  0.4436,  0.75  ],
                          [ 0.5144,  0.5564,  0.25  ],
                          [ 0.    ,  0.5   ,  0.    ],
                          [ 0.    ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.    ],
                          [ 0.1015,  0.4673,  0.25  ],
                          [ 0.8985,  0.5327,  0.75  ],
                          [ 0.6015,  0.0327,  0.75  ],
                          [ 0.3985,  0.9673,  0.25  ],
                          [ 0.6962,  0.2983,  0.0524],
                          [ 0.6962,  0.2983,  0.4476],
                          [ 0.3038,  0.7017,  0.5524],
                          [ 0.1962,  0.2017,  0.5524],
                          [ 0.8038,  0.7983,  0.4476],
                          [ 0.8038,  0.7983,  0.0524],
                          [ 0.1962,  0.2017,  0.9476],
                          [ 0.3038,  0.7017,  0.9476]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potential_set = TremoloXPotentialSet(name = 'Jahn_AlCaMgOSi_2007')

# Define and add the general particle types.
```

(continues on next page)

(continued from previous page)

```
potential_set.addParticleType(
    ParticleType(
        symbol='Si',
        mass=28.0855*atomic_mass_unit,
        charge=4.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='Mg',
        mass=24.305*atomic_mass_unit,
        charge=2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='O',
        mass=15.9994 * atomic_mass_unit,
        charge=-2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)

# Define and add the Madden-Ewald-option.
option = MaddenEwaldOption(kappa=0.5, r_cut=7.5*Angstrom)
potential_set.addOption(option)

# Define and add the dispersive potential.
potential = MaddenDispersivePotential(
    particleType1='Mg',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='Si',
```

(continues on next page)

(continued from previous page)

```
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='O',
    particleType2='O',
    b=2.718371*1/Ang,
    C6=26.51346*eV*Ang**6,
    C8=142.77637*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)

# Define and add the Madden particle type for oxygen.
potential = MaddenParticle(
    particleType='O',
    dipolePolarization=True,
    quadrupolePolarization=True,
    alpha=8.7671*elementary_charge**2*Bohr**2 / Hartree,
    B=0*elementary_charge**3*Bohr**4 / Hartree**2,
    C=11.5124*elementary_charge**2*Bohr**4 / Hartree ,
    dipoleDistortion=True,
    quadrupoleDistortion=True,
    D=0.49566*Hartree,
    beta=1.2325*Bohr**-1,
    zeta=0.89219*Bohr**-1,
    eta=4.3646*Bohr**-1,
    gamma=1*Hartree
)
potential_set.addPotential(potential)

# Define and add the repulsive potential.
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Mg',
    A=1127.6125*eV,
    a=3.1346777*1/Ang,
    B=1615675.9*eV,
    b=7.3914748*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Si',
    A=1177.627*eV,
    a=2.9135798*1/Ang,
    B=1196266.8*eV,
    b=7.5233777*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='O',
    A=0.0*eV,
    a=0.0*1/Ang,
    B=0.0*eV,
    b=0.0*1/Ang,
    C=14530.879*eV,
    c=5.03763186568*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Define and add the polarization damping potential.
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Mg',
    bD=4.1853654*1/Ang,
    cD=2.828,
    kmaxD=4,
    bQ=3.6471714*1/Ang,
    cQ=1.3317,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Si',
    bD=4.015668*1/Ang,
    cD=1.5933,
    kmaxD=4,
    bQ=3.6974382*1/Ang,
    cQ=1.0592,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Set the Coulomb solver.
```

(continues on next page)

(continued from previous page)

```
potential_set.setCoulombSolver(
  CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2)
)

# Set up the TremoloX-calculator with the constructed Madden-potential.
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Optimize the structure and the cell.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.01*eV/Ang,
        max_stress=0.001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
        )
nlprint(bulk_configuration)
```

See also ref.maddencommon.notes.

MaddenOptimizationOption

class MaddenOptimizationOption(*repMaxIterations=None*, *repFTol=None*, *repGradTol=None*, *polMaxIterations=None*, *polFTol=None*, *polGradTol=None*)

Constructor of the option.

Parameters

- **repMaxIterations** (*int*) – The maximal number of iterations that are allowed during the optimization of the repulsive potential part.
- **repFTol** (*float*) – Stop the optimization of the repulsive part of the Madden potential if the relative change in the objective function value is smaller than this value. If set to < 0, this stopping criterion is disabled.
- **repGradTol** (*float*) – Stop the optimization of the repulsive part of the Madden potential if the 2-norm of the gradient of the objective function is smaller than this value. If set to < 0, this stopping criterion is disabled.
- **polMaxIterations** (*int*) – The maximal number of iterations that are allowed during the optimization of the polarization potential part.
- **polFTol** (*float*) – Stop the optimization of the polarization part of the Madden potential if the relative change in the objective function value is smaller than this value. If set to < 0, this stopping criterion is disabled.
- **polGradTol** (*float*) – Stop the optimization of the polarization part of the Madden potential if the 2-norm of the gradient of the objective function is smaller than this value. If set to < 0, this stopping criterion is disabled.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a Madden potential and add an optimization option to stop the internal optimization of the polarization potential after 20 iterations.

```
# Set up a Madden potential.
potential_set = Tangney_OSi_2002()

# Set up an optimization option to stop the optimization after 20 iterations.
optimization_option = MaddenOptimizationOption(
    polMaxIterations=20,
)
potential_set.addOption(optimization_option)
```

Notes

This option can be used to specify the details of the internal optimization in a Madden potential. It can be added to a *TremoloXPotentialSet* via the `addOption()` method.

MaddenParticle

class MaddenParticle(*particleType*, *dipolePolarization*, *quadrupolePolarization*, *alpha*, *B*, *C*, *dipoleDistortion*, *quadrupoleDistortion*, *D*, *beta*, *zeta*, *eta*, *gamma*)

Constructor of the potential.

Parameters

- **particleType** (*ParticleType or ParticleIdentifier*) – Identifier of the particle type.
- **dipolePolarization** (*bool*) – Enable/disable dipole polarization (induced dipoles).
- **quadrupolePolarization** (*bool*) – Enable/disable quadrupole polarization (induced quadrupoles).
- **alpha** (*PhysicalQuantity of type charge**2 * length**2 / energy*) – Potential parameter for the electrostatic self-energy term.
- **B** (*PhysicalQuantity of type charge**3 * length**4 / energy**2*) – Potential parameter for the electrostatic self-energy term.

•**C** (*PhysicalQuantity of type charge**2 * length**4 / energy*) – Potential parameter for the electrostatic self-energy term.

•**dipoleDistortion** (*bool*) – Enable/disable dipolar ion distortion.

•**quadrupoleDistortion** (*bool*) – Enable/disable quadrupole ion distortion.

•**D** (*PhysicalQuantity of type energy*) – Potential parameter for the distortion self-energy term.

•**beta** (*PhysicalQuantity of type length**-1*) – Potential parameter for the distortion self-energy term.

•**zeta** (*PhysicalQuantity of type length**-1*) – Potential parameter for the distortion self-energy term.

•**eta** (*PhysicalQuantity of type length**-1*) – Potential parameter for the distortion self-energy term.

•**gamma** (*PhysicalQuantity of type energy*) – Potential parameter for the distortion self-energy term.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Madden-Potential potential for an $MgSiO_3$ perovskite structure by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.25  ],
                          [ 0.0144,  0.9436,  0.75  ],
```

(continues on next page)

(continued from previous page)

```
                            [ 0.4856,  0.4436,  0.75  ],
                            [ 0.5144,  0.5564,  0.25  ],
                            [ 0.    ,  0.5   ,  0.    ],
                            [ 0.    ,  0.5   ,  0.5   ],
                            [ 0.5   ,  0.    ,  0.5   ],
                            [ 0.5   ,  0.    ,  0.    ],
                            [ 0.1015,  0.4673,  0.25  ],
                            [ 0.8985,  0.5327,  0.75  ],
                            [ 0.6015,  0.0327,  0.75  ],
                            [ 0.3985,  0.9673,  0.25  ],
                            [ 0.6962,  0.2983,  0.0524],
                            [ 0.6962,  0.2983,  0.4476],
                            [ 0.3038,  0.7017,  0.5524],
                            [ 0.1962,  0.2017,  0.5524],
                            [ 0.8038,  0.7983,  0.4476],
                            [ 0.8038,  0.7983,  0.0524],
                            [ 0.1962,  0.2017,  0.9476],
                            [ 0.3038,  0.7017,  0.9476]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potential_set = TremoloXPotentialSet(name = 'Jahn_AlCaMgOSi_2007')

# Define and add the general particle types.
potential_set.addParticleType(
    ParticleType(
        symbol='Si',
        mass=28.0855*atomic_mass_unit,
        charge=4.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='Mg',
        mass=24.305*atomic_mass_unit,
        charge=2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
```

(continues on next page)

(continued from previous page)

```
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='O',
        mass=15.9994 * atomic_mass_unit,
        charge=-2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)

# Define and add the Madden-Ewald-option.
option = MaddenEwaldOption(kappa=0.5, r_cut=7.5*Angstrom)
potential_set.addOption(option)

# Define and add the dispersive potential.
potential = MaddenDispersivePotential(
    particleType1='Mg',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='Si',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='O',
    particleType2='O',
    b=2.718371*1/Ang,
    C6=26.51346*eV*Ang**6,
    C8=142.77637*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
```

(continues on next page)

(continued from previous page)

```
potential_set.addPotential(potential)

# Define and add the Madden particle type for oxygen.
potential = MaddenParticle(
    particleType='O',
    dipolePolarization=True,
    quadrupolePolarization=True,
    alpha=8.7671*elementary_charge**2*Bohr**2 / Hartree,
    B=0*elementary_charge**3*Bohr**4 / Hartree**2,
    C=11.5124*elementary_charge**2*Bohr**4 / Hartree ,
    dipoleDistortion=True,
    quadrupoleDistortion=True,
    D=0.49566*Hartree,
    beta=1.2325*Bohr**-1,
    zeta=0.89219*Bohr**-1,
    eta=4.3646*Bohr**-1,
    gamma=1*Hartree
)
potential_set.addPotential(potential)

# Define and add the repulsive potential.
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Mg',
    A=1127.6125*eV,
    a=3.1346777*1/Ang,
    B=1615675.9*eV,
    b=7.3914748*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Si',
    A=1177.627*eV,
    a=2.9135798*1/Ang,
    B=1196266.8*eV,
    b=7.5233777*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='O',
    A=0.0*eV,
    a=0.0*1/Ang,
    B=0.0*eV,
    b=0.0*1/Ang,
```

(continues on next page)

(continued from previous page)

```
    C=14530.879*eV,
    c=5.03763186568*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Define and add the polarization damping potential.
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Mg',
    bD=4.1853654*1/Ang,
    cD=2.828,
    kmaxD=4,
    bQ=3.6471714*1/Ang,
    cQ=1.3317,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Si',
    bD=4.015668*1/Ang,
    cD=1.5933,
    kmaxD=4,
    bQ=3.6974382*1/Ang,
    cQ=1.0592,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Set the Coulomb solver.
potential_set.setCoulombSolver(
  CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2)
)

# Set up the TremoloX-calculator with the constructed Madden-potential.
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Optimize the structure and the cell.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.01*eV/Ang,
        max_stress=0.001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
```

(continues on next page)

(continued from previous page)

```
        )
nlprint(bulk_configuration)
```

See also ref.maddencommon.notes.

MaddenPolarizationDampingPotential

class MaddenPolarizationDampingPotential(*particleType1*, *particleType2*, *bD*, *cD*, *kmaxD*, *bQ*, *cQ*, *kmaxQ*, *r_cut*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type. Dipolar or quadrupolar polarization must be activated for this particle type by adding a corresponding MaddenParticle object to the potential set.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **bD** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **cD** (*float*) – Potential parameter.
- **kmaxD** (*int*) – Potential parameter.
- **bQ** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **cQ** (*float*) – Potential parameter.
- **kmaxQ** (*int*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Madden-Potential potential for an $MgSiO_3$ perovskite structure by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.25  ],
                          [ 0.0144,  0.9436,  0.75  ],
                          [ 0.4856,  0.4436,  0.75  ],
                          [ 0.5144,  0.5564,  0.25  ],
                          [ 0.    ,  0.5   ,  0.    ],
                          [ 0.    ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.    ],
                          [ 0.1015,  0.4673,  0.25  ],
                          [ 0.8985,  0.5327,  0.75  ],
                          [ 0.6015,  0.0327,  0.75  ],
                          [ 0.3985,  0.9673,  0.25  ],
                          [ 0.6962,  0.2983,  0.0524],
                          [ 0.6962,  0.2983,  0.4476],
                          [ 0.3038,  0.7017,  0.5524],
                          [ 0.1962,  0.2017,  0.5524],
                          [ 0.8038,  0.7983,  0.4476],
                          [ 0.8038,  0.7983,  0.0524],
                          [ 0.1962,  0.2017,  0.9476],
                          [ 0.3038,  0.7017,  0.9476]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potential_set = TremoloXPotentialSet(name = 'Jahn_AlCaMgOSi_2007')

# Define and add the general particle types.
```

(continues on next page)

(continued from previous page)

```
potential_set.addParticleType(
    ParticleType(
        symbol='Si',
        mass=28.0855*atomic_mass_unit,
        charge=4.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='Mg',
        mass=24.305*atomic_mass_unit,
        charge=2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='O',
        mass=15.9994 * atomic_mass_unit,
        charge=-2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)

# Define and add the Madden-Ewald-option.
option = MaddenEwaldOption(kappa=0.5, r_cut=7.5*Angstrom)
potential_set.addOption(option)

# Define and add the dispersive potential.
potential = MaddenDispersivePotential(
    particleType1='Mg',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='Si',
```

(continues on next page)

(continued from previous page)

```
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='O',
    particleType2='O',
    b=2.718371*1/Ang,
    C6=26.51346*eV*Ang**6,
    C8=142.77637*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)

# Define and add the Madden particle type for oxygen.
potential = MaddenParticle(
    particleType='O',
    dipolePolarization=True,
    quadrupolePolarization=True,
    alpha=8.7671*elementary_charge**2*Bohr**2 / Hartree,
    B=0*elementary_charge**3*Bohr**4 / Hartree**2,
    C=11.5124*elementary_charge**2*Bohr**4 / Hartree ,
    dipoleDistortion=True,
    quadrupoleDistortion=True,
    D=0.49566*Hartree,
    beta=1.2325*Bohr**-1,
    zeta=0.89219*Bohr**-1,
    eta=4.3646*Bohr**-1,
    gamma=1*Hartree
)
potential_set.addPotential(potential)

# Define and add the repulsive potential.
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Mg',
    A=1127.6125*eV,
    a=3.1346777*1/Ang,
    B=1615675.9*eV,
    b=7.3914748*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
```

(continues on next page)

(continued from previous page)

```
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Si',
    A=1177.627*eV,
    a=2.9135798*1/Ang,
    B=1196266.8*eV,
    b=7.5233777*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='O',
    A=0.0*eV,
    a=0.0*1/Ang,
    B=0.0*eV,
    b=0.0*1/Ang,
    C=14530.879*eV,
    c=5.03763186568*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Define and add the polarization damping potential.
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Mg',
    bD=4.1853654*1/Ang,
    cD=2.828,
    kmaxD=4,
    bQ=3.6471714*1/Ang,
    cQ=1.3317,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Si',
    bD=4.015668*1/Ang,
    cD=1.5933,
    kmaxD=4,
    bQ=3.6974382*1/Ang,
    cQ=1.0592,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Set the Coulomb solver.
```

(continues on next page)

(continued from previous page)

```
potential_set.setCoulombSolver(
  CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2)
)

# Set up the TremoloX-calculator with the constructed Madden-potential.
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Optimize the structure and the cell.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.01*eV/Ang,
        max_stress=0.001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
        )
nlprint(bulk_configuration)
```

See also ref.maddencommon.notes.

MaddenRepulsivePotential

class MaddenRepulsivePotential(*particleType1*, *particleType2*, *A*, *a*, *B*, *b*, *C*, *c*, *r_cut*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type. Dipolar or quadrupolar ion distortion must be activated for this particle type by adding a corresponding MaddenParticle object to the potential set.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **a** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **B** (*PhysicalQuantity of type energy*) – Potential parameter.
- **b** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **C** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Madden-Potential potential for an $MgSiO_3$ perovskite structure by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.25  ],
                          [ 0.0144,  0.9436,  0.75  ],
                          [ 0.4856,  0.4436,  0.75  ],
                          [ 0.5144,  0.5564,  0.25  ],
                          [ 0.    ,  0.5   ,  0.    ],
                          [ 0.    ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.5   ],
                          [ 0.5   ,  0.    ,  0.    ],
                          [ 0.1015,  0.4673,  0.25  ],
                          [ 0.8985,  0.5327,  0.75  ],
                          [ 0.6015,  0.0327,  0.75  ],
                          [ 0.3985,  0.9673,  0.25  ],
                          [ 0.6962,  0.2983,  0.0524],
                          [ 0.6962,  0.2983,  0.4476],
                          [ 0.3038,  0.7017,  0.5524],
                          [ 0.1962,  0.2017,  0.5524],
                          [ 0.8038,  0.7983,  0.4476],
                          [ 0.8038,  0.7983,  0.0524],
                          [ 0.1962,  0.2017,  0.9476],
                          [ 0.3038,  0.7017,  0.9476]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potential_set = TremoloXPotentialSet(name = 'Jahn_AlCaMgOSi_2007')

# Define and add the general particle types.
potential_set.addParticleType(
    ParticleType(
        symbol='Si',
        mass=28.0855*atomic_mass_unit,
        charge=4.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='Mg',
        mass=24.305*atomic_mass_unit,
        charge=2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)
potential_set.addParticleType(
    ParticleType(
        symbol='O',
        mass=15.9994 * atomic_mass_unit,
        charge=-2.0,
        sigma=None,
        sigma14=None,
        epsilon=None,
        epsilon14=None
    )
)

# Define and add the Madden-Ewald-option.
option = MaddenEwaldOption(kappa=0.5, r_cut=7.5*Angstrom)
potential_set.addOption(option)
```

(continues on next page)

(continued from previous page)

```
# Define and add the dispersive potential.
potential = MaddenDispersivePotential(
    particleType1='Mg',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='Si',
    particleType2='O',
    b=4.1681689*1/Ang,
    C6=1.3021902*eV*Ang**6,
    C8=4.2341478*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenDispersivePotential(
    particleType1='O',
    particleType2='O',
    b=2.718371*1/Ang,
    C6=26.51346*eV*Ang**6,
    C8=142.77637*eV*Ang**8,
    kmax6=6,
    kmax8=8,
    r_cut=12.5*Angstrom
)
potential_set.addPotential(potential)

# Define and add the Madden particle type for oxygen.
potential = MaddenParticle(
    particleType='O',
    dipolePolarization=True,
    quadrupolePolarization=True,
    alpha=8.7671*elementary_charge**2*Bohr**2 / Hartree,
    B=0*elementary_charge**3*Bohr**4 / Hartree**2,
    C=11.5124*elementary_charge**2*Bohr**4 / Hartree ,
    dipoleDistortion=True,
    quadrupoleDistortion=True,
    D=0.49566*Hartree,
    beta=1.2325*Bohr**-1,
    zeta=0.89219*Bohr**-1,
    eta=4.3646*Bohr**-1,
    gamma=1*Hartree
)
```

(continues on next page)

(continued from previous page)

```
potential_set.addPotential(potential)

# Define and add the repulsive potential.
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Mg',
    A=1127.6125*eV,
    a=3.1346777*1/Ang,
    B=1615675.9*eV,
    b=7.3914748*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='Si',
    A=1177.627*eV,
    a=2.9135798*1/Ang,
    B=1196266.8*eV,
    b=7.5233777*1/Ang,
    C=170982.73*eV,
    c=8.0190528*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)
potential = MaddenRepulsivePotential(
    particleType1='O',
    particleType2='O',
    A=0.0*eV,
    a=0.0*1/Ang,
    B=0.0*eV,
    b=0.0*1/Ang,
    C=14530.879*eV,
    c=5.03763186568*1/Ang,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Define and add the polarization damping potential.
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Mg',
    bD=4.1853654*1/Ang,
    cD=2.828,
    kmaxD=4,
    bQ=3.6471714*1/Ang,
    cQ=1.3317,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
```

(continues on next page)

(continued from previous page)

```
potential_set.addPotential(potential)
potential = MaddenPolarizationDampingPotential(
    particleType1='O',
    particleType2='Si',
    bD=4.015668*1/Ang,
    cD=1.5933,
    kmaxD=4,
    bQ=3.6974382*1/Ang,
    cQ=1.0592,
    kmaxQ=6,
    r_cut=6.0*Angstrom
)
potential_set.addPotential(potential)

# Set the Coulomb solver.
potential_set.setCoulombSolver(
  CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2)
)

# Set up the TremoloX-calculator with the constructed Madden-potential.
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Optimize the structure and the cell.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.01*eV/Ang,
        max_stress=0.001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        trajectory_filename=None,
        optimizer_method=LBFGS(),
        )
nlprint(bulk_configuration)
```

See also ref.maddencommon.notes.

MagneticAnisotropy

class MagneticAnisotropy(*temperatures*, *theta_angles*, *phi_angles*, *relative_magnetic_moments*, *max_total_magnetic_moment*, *total_torques*)

An object storing magnetic moments and spin torques at a range of temperatures.

Parameters

- **temperatures** – The list of temperatures.
- **theta_angles** – The list of theta angles.
- **phi_angles** – The list of phi angles.

- **relative_magnetic_moments** – The relative magnetic moment for each temperature, theta and phi angle. Thus it should be an array of shape (N_temp, N_theta, N_phi, 3), where N_temp is the number of temperatures, N_theta is the number of theta angles and N_phi is the number of phi angles.
- **total_torques** – The total magnetic torque on the system for each temperature, phi and theta angle. Must be an array of the same shape as relative_magnetic_moments.

freeEnergy(*use_phi_angle=True*)

Parameters
use_phi_angle (*bool*) – Determines what angle should be used for the integration. If *True* the phi angles will be used; otherwise the theta angles.

Returns
The free energy obtained by integrating the torque over angles. The shape is (*n_t*, *n_a*), where *n_t* is the number of temperatures, and *n_a* is the number of angle points.

Return type
PhysicalQuantity of type energy

magneticAnisotropyEnergy(*use_phi_angle=True*)

Parameters
use_phi_angle (*bool*) – Determines what angle should be used for the integration. If *True* the phi angles will be used; otherwise the theta angles.

Returns
The maximum difference of the free energy for each temperature

Return type
PhysicalQuantity of type energy

maxTotalMagneticMoment()

Get the maximum total magnetic moment.

Returns
The maximum total magnetic moment.

Return type
PhysicalQuantity of type energy/magnetic field

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the data.

Return type
dict

nlprint(*stream*)

Print report with the data and results to a given stream.

Parameters
stream (*stream*) – The stream that the report should be written to.

phiAngles()

Get the list of phi angles.

Returns
The phi angles.

relativeMagneticMoments()

Calculate the relative magnetizations.

Returns
The relative magnetizations. The relative magnetizations. Will be an array of shape (*N_temp*, *N_theta*, *N_phi*, 3).

Retype
ndarray

relativeMagnetizations()

Calculate the relative magnetizations.

Returns
The relative magnetizations. Will be an array of shape (*N_temp*, *N_theta*, *N_phi*).

Retype
ndarray

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

temperatures()

Get the list of temperatures.

Returns
The temperatures.

thetaAngles()

Get the list of theta angles.

Returns
The theta angles.

totalMagneticMoments()

Get the total magnetic moments.

Returns
The list of magnetic moments.

Return type
PhysicalQuantity with units of energy/magnetic field

totalTorques()

Get the total magnetic torques on the system.

Returns
The total magnetic torques. Will be an array of shape (*N_temp*, *N_theta*, *N_phi*, 3).

Return type
PhysicalQuantity with units of energy

uniqueString()

Return a unique string representing the state of the object.

MagneticAnisotropyEnergy

class MagneticAnisotropyEnergy(*configuration*, *filename*, *object_id*, *theta_angles=None*, *phi_angles=None*, *kpoints=None*, *projections=None*, *energy_minimum=None*, *energy_maximum=None*, *bands_per_electron=None*, *initial_spin=None*, *initial_state=None*, *store_k_point_resolved_data=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*)

Constructor for the MagneticAnisotropyEnergy object.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the projected density of states. The configuration must have an LCAOCalculator attached with a pseudopotential that supports Spin-Orbit coupling.
- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **theta_angles** (*PhysicalQuantity* of type angle.`) – A list of theta angles specifying the spin rotation for the magnetic anisotropy calculations. **Default:** `[0.0, 90.0]*Degrees`
- **phi_angles** – A list of phi angles specifying the spin rotation for the magnetic anisotropy calculations. **Default:** `0.0*Degrees`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points for which to calculate the projected density of states. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **projections** (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for the calculating the weights. **Default:** `ProjectOnSites`.
- **energy_minimum** (*PhysicalQuantity* of type energy.) – The smallest energy to consider, relative to the Fermi energy. **Default:** `-1000*eV`
- **energy_maximum** (*PhysicalQuantity* of type energy.) – The highest energy to consider, relative to the Fermi energy. **Default:** `1.0*eV`
- **bands_per_electron** (*float*) – The number of bands per electron to include in the projection calculations. The number must be 1.0 or larger. **Default:** `1.1`
- **initial_spin** – The initial spins for the polarized configuration. **Default:** Maximum spin polarization.
- **initial_state** (*BulkConfiguration* | *MoleculeConfiguration* | None) – The initial state object to be used for the polarized configuration. The initial_state must have a calculator with either polarized or unpolarized exchange-correlation. **Default:** No initial state.
- **store_k_point_resolved_data** (*bool*) – Whether or not the energies and projections should be stored k-point resolved. **Default:** False for LCAOCalculator, True for PlaneWaveCalculator.
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of magnetic anisotropy energy calculations. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'magneticanisotropyenergy_'`
- **number_of_processes_per_task** (int | *ProcessesPerNode* | `AllProcesses`) – The number of processes that will be used to execute each task. If this value is greater than or equal to the total number of available processes, each single

task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. **Default:** `AllProcesses`, all available processes execute each task collaboratively.

addAngles(*theta_angles=None*, *phi_angles=None*)

Add a list of theta- and/or phi angles to the MagneticAnisotropyEnergy study. These will be calculated the next time the object is updated.

Parameters

- **theta_angles** (*PhysicalQuantity* of type angle) – A list with theta angles specifying the spin rotation for the magnetic anisotropy calculations.
- **phi_angles** – A list with phi angles specifying the spin rotation for the magnetic anisotropy calculations.

addProjections(*projections=None*)

Add a list of projection or a projection generator to the MagneticAnisotropyEnergy study. These will be calculated the next time the object is updated.

Parameters
projections (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for the calculating the weights.

bandEnergies(*theta*, *phi*)

Method to get the calculated band energies for each projection, band and k-point. This method only returns data when `store_k_point_resolved_data=True`, which is not default for LCAOCalculator. When `store_k_point_resolved_data=False` the method will return None.

Parameters

- **theta** (*PhysicalQuantity* of type angle.) – The theta angle to get the results for.
- **phi** (*PhysicalQuantity* of type angle.) – The phi angle to get the results for.

Returns
The band energies for the updated spin orbit configuration for specified theta and phi angles. The energies array has dimension (number_number_of_kpoints, number_of_bands).

Return type
PhysicalQuantity of type energy

bandEnergyVsAngles()

Calculates the summed band energies for all the angles.

Returns
The total magnetic anisotropy energy as function of theta and phi angles. The shape of the array is (n-theta, n-phi), where n-theta is the number of theta angles and n-phi is the number of phi angles.

Return type
PhysicalQuantity of type energy.

bandsPerElectron()

Returns
The number of bands per electron to include in the projection calculations.

Return type
float

calculatedPhiAngles()

Retrieve the list of calculated phi angles.

Returns
The list of calculated phi angles.

Return type
PhysicalQuantity of type angle.

calculatedThetaAngles()

Retrieve the list of calculated theta angles.

Returns
The list of calculated theta angles.

Return type
PhysicalQuantity of type angle.

calculator()

Returns
The calculator attached to the configuration.

Return type
LCAOCalculator

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

energyMaximum()

Returns
The highest energy to consider, relative to the Fermi energy.

Return type
PhysicalQuantity of type energy.

energyMinimum()

Returns
The smallest energy to consider, relative to the Fermi energy.

Return type
PhysicalQuantity of type energy.

filename()

Returns
The filename where the study object is stored.

Return type
str

initialSpin()

Returns
The initial spin used for the polarized calculation.

Return type
InitialSpin

initialState()

Returns
The initial state used for the polarized calculation.

Return type
BulkConfiguration | None

kpoints()

Returns
The k-point sampling used for calculating the magnetic anisotropy energy.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

magneticAnisotropyEnergy(*theta_0=None*, *phi_0=None*, *theta_1=None*, *phi_1=None*, *energy_minimum=None*, *energy_maximum=None*)

The magnetic anisotropy energy is defined as *MAE = E(theta_1, phi_1) - E(theta_0, phi_0)*.

Parameters

- **theta_0** (*PhysicalQuantity* of type angle.) – One of the theta angles to check. **Default:** The first theta angle.
- **theta_1** (*PhysicalQuantity* of type angle.) – One of the theta angles to check. **Default:** The last theta angle.
- **phi_0** (*PhysicalQuantity* of type angle.) – One of the phi angles to check. **Default:** The first phi angle.
- **phi_1** (*PhysicalQuantity* of type angle.) – One of the phi angles to check. **Default:** The last phi angle.
- **energy_minimum** (*PhysicalQuantity* of type energy.) – The smallest energy to consider, relative to the Fermi energy.
- **energy_maximum** (*PhysicalQuantity* of type energy.) – The highest energy to consider, relative to the Fermi energy.

Returns
The total MAE.

Return type
PhysicalQuantity of type energy.

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

phiAngles()

Returns
The phi angles used for the calculated magnetic anisotropy energy.

Return type
PhysicalQuantity of type angle.

projectionWeights(*theta*, *phi*)

Method to get the calculated projection weights for each projection, band and k-point. This method only returns data when `store_k_point_resolved_data=True`, which is not default for LCAOCalculator. When `store_k_point_resolved_data=False` the method will return None.

Parameters

- **theta** (*PhysicalQuantity* of type angle.) – The theta angle to get the results for.
- **phi** (*PhysicalQuantity* of type angle.) – The phi angle to get the results for.

Returns
Retrieve the projection weights for the updated spin orbit configuration for specified theta and phi angles. The weights array has dimensions (number_of_projections, number_number_of_kpoints, number_of_bands).

Return type
PhysicalQuantity of type energy

projections()

Returns
The projections used for the calculated magnetic anisotropy energy.

Return type
list of *Projection* | *Projection*

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

siteAndOrbitalProjectedMAE(*theta_0=None*, *theta_1=None*, *phi_0=None*, *phi_1=None*, *axis='z'*)

Function for getting the site and shell projected magnetic anisotropy. The magnetic anisotropy energy is defined as *MAE = E(theta_1, phi_1) - E(theta_0, phi_0)*. The site- and orbital-projected energies are plotted along the specified axis.

Parameters

- **theta_0** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The smallest calculated theta angle.
- **theta_1** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The largest calculated theta angle.
- **phi_0** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The smallest calculated phi angle.
- **phi_1** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The largest calculated phi angle.
- **axis** (*str.*) – The axis to plot the site projections along. This should either be x, y, or z.

Returns
Two dicts with coordinates and site- and orbital-resolved MAE. The first dict contains for each element the coordinates for atoms of this type. The second dict contains for each element a list of arrays with dimension (number of atoms, 2*l + 1) for each index l in the list.

Return type
dict, dict.

siteAndShellProjectedMAE(*theta_0=None*, *theta_1=None*, *phi_0=None*, *phi_1=None*, *axis='z'*)

Function for getting the site and shell projected magnetic anisotropy. The magnetic anisotropy energy is defined as *MAE = E(theta_1, phi_1) - E(theta_0, phi_0)*. The site- and shell-projected energies are plotted along the specified axis.

Parameters

- **theta_0** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The smallest calculated theta angle.
- **theta_1** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The largest calculated theta angle.
- **phi_0** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The smallest calculated phi angle.
- **phi_1** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The largest calculated phi angle.
- **axis** (*str.*) – The axis to plot the site projections along. This should either be x, y, or z.

Returns
Two dicts with coordinates and site- and shell-resolved MAE. The first dict contains for each element the coordinates for atoms of this type. The second dict contains for each element an array with dimension (number of atoms, number of shells).

Return type
dict, dict.

siteProjectedMAE(*theta_0=None*, *theta_1=None*, *phi_0=None*, *phi_1=None*, *axis='z'*)

Function for getting the site projected magnetic anisotropy. The magnetic anisotropy energy is defined as *MAE = E(theta_1, phi_1) - E(theta_0, phi_0)*.

Parameters

- **theta_0** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The smallest calculated theta angle.

- **theta_1** (*PhysicalQuantity* of type angle.) – One of the theta angles at which to evaluate the energies. **Default:** The largest calculated theta angle.
- **phi_0** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The smallest calculated phi angle.
- **phi_1** (*PhysicalQuantity* of type angle.) – One of the phi angles at which to evaluate the energies. **Default:** The largest calculated phi angle.
- **axis** (*str.*) – The axis to plot the site projections along. This should either be x, y, or z.

Returns
x- and y-data as coordinates and energies. If no site projections are present the return will be None, None

Return type
PhysicalQuantity of type length, *PhysicalQuantity* of type energy.

storeKPointResolvedData()

Returns
Boolean controlling if energies and projections should be stored k-point resolved.

Return type
bool

thetaAngles()

Returns
The theta angles used for the calculated magnetic anisotropy energy.

Return type
PhysicalQuantity of type angle.

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

updatedPolarizedConfiguration()

Retrieve the updated polarized configuration

Returns
The optimized device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

updatedSpinOrbitConfiguration(*theta*, *phi*)

Retrieve the updated spin orbit configuration for specified theta and phi angles.

Returns
The optimized device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *MagneticAnisotropyEnergy* object can be used to perform a detailed study of the magnetic anisotropy energy, which is defined as the energy difference between two spin orientations:

$$MAE = E(\theta_1, \phi_1) - E(\theta_0, \phi_0),$$

where the spin orientation is described by the two spherical angles θ and ϕ.

Force theorem

In the *MagneticAnisotropyEnergy* study object we calculate the MAE using the force theorem, which allows you to evaluate the energy difference using non-self consistent band energies[1]:

$$MAE = \sum_i f_i(\theta_1, \phi_1)\epsilon_i(\theta_1, \phi_1) - \sum_i f_i(\theta_0, \phi_0)\epsilon_i(\theta_0, \phi_0),$$

where $f_i(\theta, \phi)$ is the occupation factor for band i (including both band and k-point index) with the spin orientation (θ, ϕ) and $\epsilon_i(\theta, \phi)$ is the corresponding band energy.

The work implemented in the study object is the following

1. Perform a self-consistent polarized calculation.
2. For each spin orientation:
   i. Make a non-self consistent spin-orbit calculation using the polarized electron density and effective potential, rotated in the desired direction.
   ii. Using this configuration calculate and sum the band energies according to the formula above.
   iii. For each band energy (band index and k-point) we potentially also calculate the projection weight using the specified projection scheme. This step allows for a detailed site- and/or orbital compositional analysis.

The use of the force theorem (FT) has been validated in several papers, e.g.[2],[3] against self-consistent calculations of total energies. Using the FT has several advantages over the self-consistent approach:

- It is significantly faster since the self-consistent solution of the spin-orbit calculation is computationally significantly more demanding than the polarized one.
- Often it is harder to achieve good convergence for a spin-orbit calculation than for a polarized one.
- Using the FT approach allows for projection analysis since each band energy has an associated eigenvector that can be projected onto atomic sites or individual orbitals (see below). The projection analysis is often helpful in understanding the physics behind the calculated MAE. An example could be the difference between surface/interface atoms and bulk atoms in the contribution to the MAE in a metallic slab[4].

[1] G.ăH.ăO. Daalderop, P.ăJ. Kelly, and M.ăF.ăH. Schuurmans. First-principles calculation of the magnetocrystalline anisotropy energy of iron, cobalt, and nickel. *Phys. Rev. B*, 41:11919, 1990. doi:10.1103/PhysRevB.41.11919.

[2] Piotr Blonski and Jurgen Hafner. Density-functional theory of the magnetic anisotropy of nanostructures: an assessment of different approximations. *Journal of Physics: Condensed Matter*, 21(42):426001, 2009. URL: http://stacks.iop.org/0953-8984/21/i=42/a=426001.

[3] M.ăBlanco-Rey, J.I. Cerda, and A.ăArnau. Validity of perturbative methods to treat the spin–orbit interaction: application to magnetocrystalline anisotropy. *New J. Phys.*, 21:073054, 2019. URL: https://iopscience.iop.org/article/10.1088/1367-2630/ab3060.

[4] K.ăBairagi, A.ăBellec, V.ăRepain, C.ăChacon, Y.ăGirard, Y.ăGarreau, J.ăLagoute, S.ăRousset, R.ăBreitwieser, Yu-Cheng Hu, YenăCheng Chao, WoeiăWu Pai, D.ăLi, A.ăSmogunov, and C.ăBarreteau. Tuning the Magnetic Anisotropy at a Molecule-Metal Interface. *Phys. Rev. Lett.*, 114:247203, 2015. URL: https://link.aps.org/doi/10.1103/PhysRevLett.114.247203.

Note: The calculator must contain a basis/pseudopotential that supports spin-orbit calculations.

Usage Example

This example script shows how to perform a magnetic anisotropy energy calculation of CoPt in the $L0_1$ phase

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleTetragonal(3.81*Angstrom, 3.68846*Angstrom)

# Define elements
elements = [Cobalt, Platinum, Platinum, Cobalt]

# Define coordinates
fractional_coordinates = [[ 0.            ,  0.            ,  0.            ],
                          [ 0.            ,  0.5           ,  0.5           ],
                          [ 0.5           ,  0.            ,  0.5           ],
                          [ 0.5           ,  0.5           ,  0.            ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
    BasisGGAPseudoDojoSO.Cobalt_Medium,
    BasisGGAPseudoDojoSO.Platinum_Medium,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = SOGGA.PBE

k_point_sampling = MonkhorstPackGrid(
    na=25,
```

(continues on next page)

(continued from previous page)

```
    nb=25,
    nc=25,
    force_timereversal=False,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=155.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=ColdSmearing(0.05*eV),
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
nlsave('CoPt_MAE.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Magnetic Anisotropy Energy
# ------------------------------------------------------------
# Kpoint sampling
kpoint_grid = MonkhorstPackGrid(
    na=35,
    nb=35,
    nc=35,
    )

# File name.
filename = 'CoPt_MAE.hdf5'

magnetic_anisotropy_energy = MagneticAnisotropyEnergy(
    configuration=bulk_configuration,
    filename=filename,
    object_id='magnetic_anisotropy_energy',
    theta_angles=numpy.linspace(0, 90, 2)*Degrees,
    phi_angles=numpy.linspace(0, 0, 1)*Degrees,
    kpoints=kpoint_grid,
    projections=NoProjection,
    energy_minimum=-1000*eV,
    energy_maximum=1*eV,
    bands_per_electron=1.1,
    log_filename_prefix='magnetic_anisotropy_energy_',
    number_of_processes_per_task=None,
)

magnetic_anisotropy_energy.update()
```

magnetic_anisotropy_energy.py

Note: In order to get accurate results, a high k-point sampling is needed on the calculator and an even higher sampling on the *MagneticAnisotropyEnergy* study object. Convergence with respect to k-point sampling should be carefully checked.

Results

The total MAE can be obtained from the method `magneticAnisotropyEnergy` which returns the total MAE for specified angles and projections. By default:

```
mae = magnetic_anisotropy_energy.magneticAnisotropyEnergy()
```

will return the MAE calculated from the first and last angle pairs defined in the construction of the study object.

It is further possible the access the summed band energies (i.e. not band energy difference) from the method `summedBandEnergies`.

In order to inspect the dependence on the angles it is possible, using the method `projectedBandEnergyVsAngles`, to obtain the projected band energies for all the considered angles.

Adding angles

After having updated a *MagneticAnisotropyEnergy* study object, it is possible perform additional calculations for different angles using the `addAngles` method:

```
magnetic_anisotropy_energy.addAngles(phi_angles=[22.5, 45.0]*Degrees)
magnetic_anisotropy_energy.update()
```

By doing so one avoids to redo the self-consistent polarized calculation since this is already stored on the study.

Adding projections

After having updated a *MagneticAnisotropyEnergy* study object, it is also possible perform additional calculations for different projections using the `addProjections` method:

```
magnetic_anisotropy_energy.addProjections(ProjectOnShellsBySite)
magnetic_anisotropy_energy.update()
```

for projecting on individual shells $s, p, d,$ etc. on each atom, or:

```
magnetic_anisotropy_energy.addProjections(ProjectOnOrbitalsBySite)
magnetic_anisotropy_energy.update()
```

for projecting on each orbital $s, p_x, p_y, p_z,$ etc. on each atom.

Calculating without projections

If one is only interested in the total MAE and not concerned about the atom or orbital projections it is possible, and often significantly faster, to set the projections to NoProjections like:

```
projections=NoProjection
```

MarkovChain

class MarkovChain(*configuration*, *configuration_energy*, *events=None*, *energy_tolerance=None*, *distance_tolerance=None*, *filename=None*, *object_id=None*)

Construct a markov chain from a list of *HTSTEvent* objects.

Parameters

- **configuration** (BulkConfiguration) – The configuration for the first state of the Markov chain.
- **configuration_energy** (*PhysicalQuantity* of type energy) – The energy of the configuration.
- **events** (list of type *HTSTEvent*) – A list of HTSTEvent objects used to construct the Markov chain. By default an empty Markov chain will be made. **Default:** None
- **distance_tolerance** (*PhysicalQuantity* of type length) – This is the distance tolerance used to compare geometries. **Default:** 0.1*Angstrom
- **energy_tolerance** (*PhysicalQuantity* of type energy) – This energy tolerance is used to determine if two structures are different based upon energy alone. Configurations that are closer than this tolerance undergo a full geometry comparison. **Default:** 0.02*eV

addConnection(*connection*, *event*)

Add a new connection to the Markov chain. This is a potentially dangerous method to call since it intentionally bypasses the geometry comparison routines.

Parameters

- **connection** (*a pair of ints*) – A tuple of integers representing the reactant and product state ids respectively.
- **event** (HTSTEvent) – The event to add.

addEvent(*event*)

Add an event to the Markov chain.

Parameters
event (HTSTEvent) – A event to be added to the Markov chain.

Returns
The id of the product state

Return type
int

addStateTime(*state_id*, *dt*)

Increase the amount of MD time simulation in a state.

Parameters

- **state_id** (*int*) – The id of the state to add time to.
- **dt** (*A PhysicalQuantity with time units*) – The amount of time to add.

getSaddleConfiguration(*initial_state_id*, *final_state_id*)

Parameters

- **initial_state_id** (*int*) – The id of the initial state.
- **final_state_id** (*int*) – The id of the final state.

Returns
The saddle configuration between two states.

Return type
BulkConfiguration

getSaddleEnergy(*initial_state_id*, *final_state_id*)

Parameters

- **initial_state_id** (*int*) – The id of the initial state.
- **final_state_id** (*int*) – The id of the final state.

Returns
The energy of the saddle point.

Return type
PhysicalQuantity of type energy

getState(*state_id*)

Parameters
state_id (*int*) – The state id.

Returns
The requested state.

Return type
MarkovState

getStateConfidence(*state_id*, *kmc_temperature*)

Return the confidence of the requested state. The confidence represents an estimate of the probability that a KMC step chooses the correct event. For example, a confidence of 0.99 means that there is a 1% chance that the KMC simulation picked the wrong event in a single step.

Parameters

- **state_id** (*int*) – The id of the state.
- **kmc_temperature** (*PhysicalQuantity* of type temperature) – The temperature of the KMC simulation.

Returns
The confidence.

Return type
float

getStateConfiguration(*state_id*)

Parameters
state_id (*int*) – The id of the state.

Returns
The configuration associated with a state.

Return type
BulkConfiguration

`getStateEnergy`(*state_id*)

Parameters
state_id (*int*) – The id of the state.

Returns
The energy of a state.

Return type
PhysicalQuantity of type energy

`getStateEvents`(*state_id*)

Return the HTSTEvents that connect to state_id.

Parameters
state_id (*int*) – The state_id of the state.

Returns
A dictionary that maps (reactant_id, product_id) pairs to HTSTEvents.

Return type
OrderedDict

`getStateTime`(*state_id*)

Parameters
state_id (*int*) – The id of the state.

Returns
The total amount of MD time simulation in the state.

Return type
PhysicalQuantity of type time

`getStates`()

Returns
A list of the states in the Markov chain.

Return type
list of type MarkovState

`incrementObservationCount`(*initial_state_id*, *final_state_id*)

Increment the number of times that a transition from the initial state to the final state has occurred.

Parameters

- **initial_state_id** (*int*) – The initial state.
- **final_state_id** (*int*) – The final state.

`isConnected`(*state_id_1*, *state_id_2*)

Determine if two states are connected by an event.

Parameters

- **state_id_1** (*int*) – The id of the first state.
- **state_id_2** (*int*) – The id of the second state.

Returns
True if connected otherwise False.

Return type
bool

metatext()

Returns
The metatext of the object or None if no metatext is set.

Return type
str | None

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print out the Markov chain in DOT graph format.

Parameters
stream (*file-like object*) – The stream to write the DOT graph to. **Default:** sys.stdout

observationCount(*initial_state_id*, *final_state_id*)

Parameters

- **initial_state_id** (*int*) – The initial state.
- **final_state_id** (*int*) – The final state.

Returns
Return the number of times that a transition from the initial state to the final state has occurred.

Return type
int

rateMatrix(*temperature*)

Calculate the right handed (rows sum to the total escape rate for a state) rate matrix at a given temperature.

Parameters
temperature (*PhysicalQuantity* of type temperature) – The temperature to calculate the rates at.

Returns
The rate matrix.

Return type
PhysicalQuantity of type frequency

setMetatext(*text*)

Set a given metatext string on the object.

Parameters
text (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

Notes

MarkovChain objects are normally passed to the *AdaptiveKineticMonteCarlo* class.

MaxwellBoltzmannDistribution

class MaxwellBoltzmannDistribution(*temperature=None*, *remove_center_of_mass_momentum=None*, *random_seed=None*, *enforce_temperature=None*)

Maxwell-Boltzmann velocity distribution for particles in a *MolecularDynamics()* simulation.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature parameter of the Maxwell-Boltzmann distribution. **Default:** 300.0*Kelvin
- **remove_center_of_mass_momentum** (*bool*) – If True, the center-of-mass momentum should be removed, otherwise it is kept.
- **random_seed** (*int*) – The seed used for the random number generator. The seed should be between 0 and 2**31-1.
- **enforce_temperature** (*bool*) – Rescale the velocities of the system to ensure the instantaneous temperature is exactly the requested temperature. **Default:** True

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Perform a molecular dynamics run on a water molecule, using the default settings, but having applied a Maxwell-Boltzmann distribution of velocities to all atoms:

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[0.0,  -1.70000000e-05,   1.20198000e-01],
                         [0.0,   7.59572000e-01,  -4.86714000e-01],
                         [0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Define a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# Setup a velocity distribution-defining class.
velocities = MaxwellBoltzmannDistribution()

# Setup the MD method.
md_method = NVEVelocityVerlet(initial_velocity=velocities)

# Perform MD
md_configuration = MolecularDynamics(molecule_configuration,
                                     method=md_method,
                                     steps=10)
```

(continues on next page)

(continued from previous page)

```
# Save the final configuration
nlsave('maxwellboltzmann.nc', md_configuration)
```

maxwellboltzmann.py

MeamElementPotential

class MeamElementPotential(*particleType*, *latticeType*, *nearestNeighbors*, *alpha*, *beta*, *referenceDistance*, *referenceEnergy*, *scalingFactor*, *weightingFactors*, *rho*, *gamma*, *attrac=0.0*, *repuls=0.0*, *nn2=False*, *zbl=True*)

Constructor of the potential.

Parameters

- **particleType** (*ParticleType or ParticleIdentifier*) – Identifier of the particle type.
- **latticeType** (*str*) – Type of the reference structure.
- **nearestNeighbors** (*int*) – Number of nearest neighbors in the reference structure.
- **alpha** (*float*) – Exponential decay factor for pair potential.
- **beta** (*sequence of float*) – Exponential decay factors for the background electron densities. Must be of length four.
- **referenceDistance** (*PhysicalQuantity of type length*) – The nearest neighbor distance in the reference structure.
- **referenceEnergy** (*PhysicalQuantity of type energy*) – The cohesive energy.
- **scalingFactor** (*float*) – Scaling Factor in embedding function.
- **weightingFactors** (*sequence of float*) – Weighting factors in the reference configuration. Must be of length three.
- **rho** (*float*) – Element-dependent density scaling.
- **gamma** (*int*) – Choice for gamma function.
- **attrac** (*float*) – The attractive factor in the pair potential.
- **repuls** (*float*) – The repulsive factor in the pair potential.
- **nn2** (*bool*) – Triggers second nearest neighbor formulation for symmetric pair.
- **zbl** (*bool*) – Toggles zbl blending for small distances.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a MEAM potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.  ],
                          [ 0.75,  0.75,  0.25],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.75,  0.25,  0.75],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = TremoloXPotentialSet(name='MEAM_Si_2007')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
```

(continues on next page)

(continued from previous page)

```
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[],
))


_option = MeamGlobalOption(
    delr=0.1*Angstrom,
    erose=2,
    wf_mixing=2,
    r_cut=4.5*Angstrom,
    augment_1st=0.0,
    embedding_negative=False,
    density_scaling=False,
)
potentialSet.addOption(_option)
_potential = MeamElementPotential(
    particleType=ParticleIdentifier('Si', []),
    latticeType='dia',
    nearestNeighbors=4,
    alpha=4.89890486934,
    beta=[3.55, 2.5, 0.0, 7.5, ],
    referenceDistance=2.35*Angstrom,
    referenceEnergy=4.63*eV,
    scalingFactor=0.58,
    weightingFactors=[1.8, 5.25, -2.61, ],
    rho=1.0,
    gamma=3,
    attrac=0.0,
    repuls=0.0,
    nn2=True,
    zbl=False,
)
potentialSet.addPotential(_potential)
_potential = MeamScreeningPotential(
    particleType1=ParticleIdentifier('Si', []),
    particleType2=ParticleIdentifier('Si', []),
    particleType3=ParticleIdentifier('Si', []),
    Cmin=1.41,
    Cmax=2.8,
)
potentialSet.addPotential(_potential)

calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

See also ref.meamcommon.notes.

MeamGlobalOption

class MeamGlobalOption(*delr=None*, *erose=None*, *wf_mixing=None*, *augment_1st=None*, *r_cut=None*, *embedding_negative=None*, *density_scaling=None*, *use_spline=None*)

Constructor of this option.

Parameters

- **delr** (*PhysicalQuantity of type length*) – Smoothing delta for screening function.
- **erose** (*int*) – Choses the form of the rose energy function.
- **wf_mixing** (*int*) – Mixing rule for the weighting factors.
- **augment_1st** (*bool*) – Flag whether the first weighting factor will be augmented.
- **r_cut** (*PhysicalQuantity of type length*) – Global cutoff radius.
- **embedding_negative** (*bool*) – Flag to enable linear embedding for negative densities.
- **density_scaling** (*bool*) – Flag to toggle dynamo style background density.
- **use_spline** (*bool*) – If set to False, MEAM will not precompute pair interactions. In most cases, this will make the calculation slower, except for some potential fitting applications.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a MEAM potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
```

(continues on next page)

(continued from previous page)

```
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.  ],
                          [ 0.75,  0.75,  0.25],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.75,  0.25,  0.75],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = TremoloXPotentialSet(name='MEAM_Si_2007')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[],
))


_option = MeamGlobalOption(
    delr=0.1*Angstrom,
    erose=2,
    wf_mixing=2,
    r_cut=4.5*Angstrom,
    augment_1st=0.0,
    embedding_negative=False,
    density_scaling=False,
)
potentialSet.addOption(_option)
_potential = MeamElementPotential(
    particleType=ParticleIdentifier('Si', []),
    latticeType='dia',
    nearestNeighbors=4,
    alpha=4.89890486934,
```

(continues on next page)

(continued from previous page)

```
    beta=[3.55, 2.5, 0.0, 7.5, ],
    referenceDistance=2.35*Angstrom,
    referenceEnergy=4.63*eV,
    scalingFactor=0.58,
    weightingFactors=[1.8, 5.25, -2.61, ],
    rho=1.0,
    gamma=3,
    attrac=0.0,
    repuls=0.0,
    nn2=True,
    zbl=False,
)
potentialSet.addPotential(_potential)
_potential = MeamScreeningPotential(
    particleType1=ParticleIdentifier('Si', []),
    particleType2=ParticleIdentifier('Si', []),
    particleType3=ParticleIdentifier('Si', []),
    Cmin=1.41,
    Cmax=2.8,
)
potentialSet.addPotential(_potential)

calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

See also ref.meamcommon.notes.

MeamPairPotential

class MeamPairPotential(*particleType1*, *particleType2*, *latticeType*, *nearestNeighbors*, *alpha*, *referenceDistance*, *referenceEnergy*, *attrac=0.0*, *repuls=0.0*, *nn2=False*, *zbl=True*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **latticeType** (*str*) – Type of the reference structure.
- **nearestNeighbors** (*int*) – Number of nearest neighbors in the reference structure.
- **alpha** (*float*) – Exponential decay factor for pair potential.
- **referenceDistance** (*PhysicalQuantity of type length*) – The equilibrium nearest neighbor distance..
- **referenceEnergy** (*PhysicalQuantity of type energy*) – The cohesive energy.
- **attrac** (*float*) – The attractive factor in the pair potential.
- **repuls** (*float*) – The repulsive factor in the pair potential.
- **nn2** (*bool*) – Triggers second nearest neighbor formulation for symmetric pair.

•**zbl** (*bool*) – Toggles zbl blending for small distances.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

This potential does not really have useful defaults, so Ill just make some up for testing only

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a MEAM potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.  ],
                          [ 0.75,  0.75,  0.25],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.75,  0.25,  0.75],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = TremoloXPotentialSet(name='MEAM_Si_2007')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[],
))


_option = MeamGlobalOption(
    delr=0.1*Angstrom,
    erose=2,
    wf_mixing=2,
    r_cut=4.5*Angstrom,
    augment_1st=0.0,
    embedding_negative=False,
    density_scaling=False,
)
potentialSet.addOption(_option)
_potential = MeamElementPotential(
    particleType=ParticleIdentifier('Si', []),
    latticeType='dia',
    nearestNeighbors=4,
    alpha=4.89890486934,
    beta=[3.55, 2.5, 0.0, 7.5, ],
    referenceDistance=2.35*Angstrom,
    referenceEnergy=4.63*eV,
    scalingFactor=0.58,
    weightingFactors=[1.8, 5.25, -2.61, ],
    rho=1.0,
    gamma=3,
    attrac=0.0,
    repuls=0.0,
    nn2=True,
    zbl=False,
)
potentialSet.addPotential(_potential)
_potential = MeamScreeningPotential(
    particleType1=ParticleIdentifier('Si', []),
    particleType2=ParticleIdentifier('Si', []),
    particleType3=ParticleIdentifier('Si', []),
    Cmin=1.41,
    Cmax=2.8,
)
```

(continues on next page)

(continued from previous page)

```
potentialSet.addPotential(_potential)

calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

See also ref.meamcommon.notes.

MeamScreeningPotential

class MeamScreeningPotential(*particleType1*, *particleType2*, *particleType3*, *Cmin=2.0*, *Cmax=2.8*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type.
- **Cmin** (*Float*) – The lower border for screening.
- **Cmax** (*Float*) – The upper border for screening.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a MEAM potential for a SiO2 quartz crystal by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.  ],
                          [ 0.75,  0.75,  0.25],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.75,  0.25,  0.75],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = TremoloXPotentialSet(name='MEAM_Si_2007')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[],
))


_option = MeamGlobalOption(
    delr=0.1*Angstrom,
    erose=2,
    wf_mixing=2,
```

(continues on next page)

(continued from previous page)

```
    r_cut=4.5*Angstrom,
    augment_1st=0.0,
    embedding_negative=False,
    density_scaling=False,
)
potentialSet.addOption(_option)
_potential = MeamElementPotential(
    particleType=ParticleIdentifier('Si', []),
    latticeType='dia',
    nearestNeighbors=4,
    alpha=4.89890486934,
    beta=[3.55, 2.5, 0.0, 7.5, ],
    referenceDistance=2.35*Angstrom,
    referenceEnergy=4.63*eV,
    scalingFactor=0.58,
    weightingFactors=[1.8, 5.25, -2.61, ],
    rho=1.0,
    gamma=3,
    attrac=0.0,
    repuls=0.0,
    nn2=True,
    zbl=False,
)
potentialSet.addPotential(_potential)
_potential = MeamScreeningPotential(
    particleType1=ParticleIdentifier('Si', []),
    particleType2=ParticleIdentifier('Si', []),
    particleType3=ParticleIdentifier('Si', []),
    Cmin=1.41,
    Cmax=2.8,
)
potentialSet.addPotential(_potential)

calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

See also ref.meamcommon.notes.

MeanSquareDisplacement

class MeanSquareDisplacement(*md_trajectory*, *start_time=None*, *end_time=None*, *atom_selection=None*, *anisotropy=None*, *use_com=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the MeanSquareDisplacement object.

Parameters

- **md_trajectory** (*MDTrajectory* | `AtomicConfiguration`) – The MDTrajectory or configuration to calculate the mean-square-displacement for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`.

- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** all atoms.
- **anisotropy** (*list of type int | int | None*) – The list of Cartesian directions (x=0, y=1, z=2) to calculate the anisotropic mean square displacement in or a single Cartesian direction. By default an isotropic calculation is performed. **Default:** None.
- **use_com** (*bool*) – Whether or not displacement is calculated from molecular center of mass. **Default:** False.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.

data()

Return the mean-square-displacement values.

times()

Return the time values.

Usage Examples

Load an *MDTrajectory*, and calculate the mean-square-displacement (MSD) of all aluminum atoms. Estimate the diffusion coefficient from the slope of the MSD-curve, according to $MSD(t) = 6Dt$:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

msd = MeanSquareDisplacement(md_trajectory, atom_selection=Aluminum)

# Get the times in ps and the MSD values in Ang**2.
t        = msd.times().inUnitsOf(ps)
msd_data = msd.data().inUnitsOf(Angstrom**2)

# Plot the data using pylab.
import pylab

pylab.plot(t, msd_data, label='MSD of aluminum')
pylab.xlabel('t (ps)')
pylab.ylabel('MSD(t) (Ang**2)')
pylab.legend()

pylab.show()

# Fit the slope of the MSD to estimate the diffusion coefficient.
# If you discover non-linear behavior at small times, discard this initial part in the␣
↪fit.
a = numpy.polyfit(t[5:], msd_data[5:], deg=1)

# Calculate the diffusion coefficient in Ang**2/ps.
diffusion_coefficient = a[0]/6.0
```

mean_square_displacement.py

Notes

The *MeanSquareDisplacement* is calculated as:

$$MSD(t) = \left\langle [\mathbf{r}(t) - \mathbf{r}(0)]^2 \right\rangle .$$

In practice, the average $< ... >$ runs over all selected atoms i in the trajectory, and an additional average over simulation time is carried out to improve the statistical sampling. That means for a given time difference t all image pairs that are separated by t are taken into account in the average, as

$$MSD(t) = \frac{1}{(t_{max} - t)N_{atoms}} \sum_{i=1}^{N_{atoms}} \sum_{t'=0}^{t_{max}-t} [\mathbf{r}_i(t' + t) - \mathbf{r}_i(t')]^2 .$$

Note, that this requires a system which is equilibrated, i.e. its macroscopic properties do not change during the simulation.

By default, all elements are taken into account, but a specified selection can be given as well. The `atom_selection` parameter accepts an element, a tag name, or a list of indices to select atoms for the velocity distribution. This can be useful, e.g. in the presence of constraints as constrained atoms should be excluded in this analysis.

MethfesselPaxton

class MethfesselPaxton(*broadening*, *order=None*)

Parameters

- **broadening** (*PhysicalQuantity* of type energy or temperature) – The broadening of the distribution.
- **order** (*int*) – The order of the distribution. Must be an integer >= 0. **Default:** 1

broadening()

Returns
The broadening.

Return type
PhysicalQuantity of type energy

order()

Returns
The order.

Return type
int

Usage Examples

Use the Methfessel-Paxton occupation function with a broadening of 0.1 eV on an LCAOCalculator:

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    occupation_method=MethfesselPaxton(0.1*eV))

calculator = LCAOCalculator(numerical_accuracy_parameters=numerical_accuracy_parameters)
```

Notes

Note: For comparison of different occupation methods and suggestions for which one to choose, see *Occupation Methods*.

In the smearing scheme by Methfessel and Paxton[1] one replaces the delta function in the density of states by its finite expansion in Hermite polynomials

$$\tilde{\delta}_N(x) = \sum_{n=0}^{N} A_n H_{2n}(x) e^{-x^2},$$

where $A_n = \frac{(-1)^n}{n! 4^n \sqrt{\pi}}$ and $H_n(x)$ is the Hermite polynomial of order n. The 0 order approximation is simply a Gaussian distribution.

The integer occupation numbers are therefore replaced by fractional occupations given by the distribution

$$f_N(x) = f_0(x) + \sum_{n=1}^{N} A_n H_{2n-1}(x) e^{-x^2},$$

with $x = \frac{\epsilon - \mu}{\sigma}$.

In the Methfessel-Paxton smearing scheme the generalized entropy is

$$S_N = \frac{1}{2} A_n \sum_i H_{2n}(x_i) e^{-x_i^2}$$

where $x_i = \frac{\epsilon_i - \mu}{\sigma}$ and i runs over all states.

The benefit of the Methfessel-Paxton smearing function is that for orders $N > 0$ the total free energy has no linear or quadratic dependence on the broadening. This means that total energies and forces are close to their zero broadening value even for large values of the broadening parameter. In fact, it is often not necessary to extrapolate the energy to zero broadening. However, an estimate of the zero broadening correction is given by the expression

$$\Delta E_{\sigma \to 0}(\sigma) = \frac{N+1}{N+2} \sigma S.$$

While it is possible in **ATK** to use any order of the Methfessel-Paxton occupation function, there is typically no benefit of going to higher orders than $N = 1$, which is also the default.

Note: The order $N = 0$ Methfessel-Paxton occupation function is identical to the one in the Gaussian smearing scheme. This does therefore not have the favorable cancellation of the second order dependence of the broadening on the free energy.

[1] M.ăMethfessel and A.ăT. Paxton. High-precision sampling for Brillouin-zone integration in metals. *Phys. Rev. B*, 40(6):3616–3621, aug 1989. doi:10.1103/PhysRevB.40.3616.

MillerIndices

class MillerIndices(*h=1*, *k=0*, *l=0*)

Class for representing a triplet of Miller indices.

Parameters

- **h** (*int*) – The first Miller index.
- **k** (*int*) – The second Miller index.
- **l** (*int*) – The third Miller index.

h()

Returns
The first index.

Return type
int

hkl()

Return the indices.

Returns
The indices

Return type
array of ints

k()

Returns
The second index.

Return type
int

l()

Returns
The third index.

Return type
int

Mobility

class Mobility(*configuration*, *electron_phonon_coupling=None*, *kpoints=None*, *method=None*, *temperature=None*, *phonon_modes=None*, *electron_bands=None*, *fermi_shift=None*, *energies=None*, *inverse_relaxation_time=None*, *mobility_object=None*, *integration_method=None*, *refinement=None*, *relaxation_time_approximation=None*, *calculate_hall_coefficients=None*, *use_kpoint_symmetry=None*, *energy_broadening=None*)

Analysis class for calculating the mobility for a bulk configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration for which to calculate the mobility.
- **electron_phonon_coupling** (*ElectronPhononCoupling*) – An electron-phonon coupling object used for calculating mobilities. When `method=Full`, `electron_phonon_coupling` must be provided.

•**method** (`Full | Isotropic.`) – Keyword specifying if the results should be calculated with the full electron-phonon coupling including angular dependence (`Full`) or with an isotropic scattering rate (`Isotropic`). When `method=Full`, `electron_phonon_coupling` must be provided. When `method=Isotropic`, `kpoints` must be provided together with `energies` and `inverse_relaxation_time`.

•**kpoints** – List of fractional k-points. When `method=Isotropic`, `kpoints` must be provided. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Bose and Fermi distributions. **Default:** `300 * Kelvin`

•**phonon_modes** (list of ints | `All`) – Phonon modes to include as a list of indices or `All`. **Default:** `All` (Include all phonon modes available in the `electron_phonon_coupling`)

•**electron_bands** (list of int | `All`) – The band indices of the Bloch states to include. **Default:** `All` (Include all bands available in the `electron_phonon_coupling`)

•**fermi_shift** (*PhysicalQuantity* of type energy) – The Fermi shift. **Default:** `0.0 * eV`

•**energies** (*PhysicalQuantity* of type energy) – List of energies for which to define the scattering rate. This parameter is only used, if `method=Isotropic`. **Default:** `numpy.linspace(-0.5,0.5,100)*eV`

•**inverse_relaxation_time** (*PhysicalQuantity* of type energy) – List of scattering rates corresponding to the provided energies, or a constant scattering rate. If a list is provided the length must be the same as the `energies`. If a `mobility_object` is also provided, the `inverse_relaxation_time` will be added to the scattering rate obtained from the `mobility_object`. This parameter is only used if `method=Isotropic`. **Default:** `0.0/Second`

•**mobility_object** (*Mobility*) – A mobility object calculated with the `Full` method. This can be used to calculate the scattering rate. If a `inverse_relaxation_time` is also provided, the `inverse_relaxation_time` will be added to the scattering rate obtained from the `mobility_object`.

•**integration_method** (`GaussianBroadening() | TetrahedronMethod`) – The method to use for calculating the q-integral. **Default:** `GaussianBroadening(3e-2 * eV)`

•**refinement** (*Positive int*) – The number of times the q-grid is refined in each direction. **Default:** `1` (no interpolation)

•**calculate_hall_coefficients** (*bool*) – Boolean to determine if the Hall coefficients should be calculated or not. **Default:** `True`

•**use_kpoint_symmetry** (*bool*) – Boolean to determine if symmetries are used to reduce k-point sampling. Only used for `method=Isotropic`, `method=Full` uses electron phonon coupling. **Default:** `True`.

•**energy_broadening** (*PhysicalQuantity* of type energy) – The width of the Gaussian function approximating the delta-function. **Default:** `3e-2 * eV` **Deprecated:** from v2018.0

•**relaxation_time_approximation** (*not used*) – **Deprecated:** from v2017.0, should no longer be used.

calculateThermalVelocity(*contribution=None*, *direction=None*, *fermi_shift=None*, *temperature=None*)

Calculates the thermal velocity of electrons and holes.

Parameters

•**contribution** (`Hole | Electron | All`) – The contribution to the thermal velocity. This can be either `All`, `Hole` or `Electron`. If `All`, results for `Hole` and `Electron` will be returned. **Default:** `All`

•**direction** (*list or numpy.array of length 3.*) – The direction of the velocity in Cartesian coordinates. **Default:** [1, 0, 0]

•**fermi_shift** (*PhysicalQuantity* of type energy) – Parameter to change the Fermi energy in the calculation of the thermal velocity.

•**temperature** (`PhysicalQuantity` of type temperature) – The temperature for the thermal smearing of the Fermi distribution. **Default:** The temperature set on this Mobility object.

Returns
(v, n, band_edges), where v is the calculated thermal velocity, n is the carrier concentration, and band_edges is the energy difference between the Fermi level (including the specified fermi shift) and carrier band edge (conduction band for electrons and valence band edge for holes.) If contribution=``All`` results are returned for Hole, Electron contribution as the first and second index.

Return type
PhysicalQuantity of unit length per time, *PhysicalQuantity* of unit inverse volume, *PhysicalQuantity* of unit energy.

`carrierDensities`(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the carrier density. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated carrier density. If contribution is `Hole` or `Electron`, the hole or electron carrier density is returned. If contribution is `All`, a list is returned with the calculated hole- and electron carrier densities as the first and second element. In case of `Sum`, the total carrier density is returned.

Return type
PhysicalQuantity with the unit `Meter**-3`

`conductivity`(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the conductivity. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated conductivity. If contribution is `Hole` or `Electron`, the hole or electron conductivity is returned. If contribution is `All`, a list is returned with the calculated hole- and electron conductivities as the first and second element. In case of `Sum`, the total conductivity is returned.

Return type
PhysicalQuantity with the unit `Siemens / Meter`

`conductivityTensor`(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the conductivity tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated conductivity tensor. If contribution is `Hole` or `Electron`, the hole or electron conductivity tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron conductivity tensors as the first and second element. In case of `Sum`, the total conductivity tensor is returned.

Return type
PhysicalQuantity with the unit `Siemens / Meter`

`conductivityTensorConstantMeanFreePath`(*mean_free_path*, *contribution=None*)

Parameters

- **mean_free_path** (*PhysicalQuantity* of unit length) – The constant mean-free-path.
- **contribution** (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the conductivity tensor obtained from a constant mean-free-path. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated conductivity tensor assuming a constant mean-free-path. If contribution is `Hole` or `Electron`, the hole

or electron conductivity tensor is returned. If contribution is All, a list is returned with the calculated hole- and electron conductivity tensors as the first and second element. In case of Sum, the total conductivity tensor is returned.

Return type
PhysicalQuantity with the unit Meter / Siemens

conductivityTensorConstantRelaxationTime(*relaxation_time*, *contribution=None*)

Parameters

- **relaxation_time** (*PhysicalQuantity* of unit time) – The constant relaxation_time.
- **contribution** (All | Sum | Hole | Electron) – The contribution to the conductivity tensor obtained from a constant relaxation time. This can be either All, Sum, Hole, or Electron. **Default:** All

Returns
The calculated conductivity tensor assuming a constant relaxation time. If contribution is Hole or Electron, the hole or electron conductivity tensor is returned. If contribution is All, a list is returned with the calculated hole- and electron conductivity tensors as the first and second element. In case of Sum, the total conductivity tensor is returned. second element. In case of Sum, the total conductivity tensor is returned.

Return type
PhysicalQuantity with the unit Meter / Siemens

conductivityTensorOverConstantMeanFreePath(*contribution=None*)

Parameters
contribution (All | Sum | Hole | Electron) – The contribution to the conductivity tensor/MFP, where MFP is the constant mean-free-path. This can be either All, Sum, Hole, or Electron. **Default:** All

Returns
The calculated conductivity tensor/MFP. If contribution is Hole or Electron, the hole or electron conductivity tensor/MFP is returned. If contribution is All, a list is returned with the calculated hole- and electron conductivity tensors/MFP as the first and second element. In case of Sum, the total conductivity tensor is returned.

Return type
PhysicalQuantity with the unit Siemens / Meter^2

conductivityTensorOverConstantRelaxationTime(*contribution=None*)

Parameters
contribution (All | Sum | Hole | Electron) – The contribution to the conductivity tensor/tau, where tau is the constant relaxation time. This can be either All, Sum, Hole, or Electron. **Default:** All

Returns
The calculated conductivity tensor/tau. If contribution is Hole or Electron, the hole or electron conductivity tensor/tau is returned. If contribution is All, a list is returned with the calculated hole- and electron conductivity tensors/tau as the first and second element. In case of Sum, the total conductivity tensor is returned.

Return type
PhysicalQuantity with the unit Siemens / (Meter * Second)

eigenvaluesK()

Returns
The electron Bloch state energies for each k-point. The shape is (spin, k-points, electron bands).

Return type
PhysicalQuantity of type energy

electronBands()

Returns
The band indices of the Bloch states included in the electron-phonon coupling calculation.

Return type
list of non-negative ints

energies()

Returns
The user inputted energies list with shape (n_energies).

Return type
PhysicalQuantity of type energy.

energyBroadening()

Returns
The width of the Gaussian function approximating the delta-function, if the integration method is GaussianBroadening. If the TetrahedronMethod is used, it returns 0.0*eV.

Return type
PhysicalQuantity of type energy

energyZero()

The energy zero is equal to the Fermi level. For fixed spin moment calculations it is the average of the Fermi level for spin up and spin down.

Returns
The energy zero.

Return type
PhysicalQuantity of type energy

evaluate()

Returns
The calculated hole- and electron mobilities as the first and second element, respectively.

Return type
PhysicalQuantity with the unit Meter**2 / (Volt * Second)

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.Up

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

fermiShift()

Returns
The Fermi shift.

Return type
PhysicalQuantity of type energy

firstMomentTensor(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the first moment tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated first moment tensor. If contribution is `Hole` or `Electron`, the hole or electron first moment tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron first moment tensors as the first and second element. In case of `Sum`, the total first moment tensor is returned.

Return type
PhysicalQuantity with the unit `Volt * Siemens / (Meter * Kelvin)`

generateEnergyDependentInverseLifeTime(*energies*, *energy_broadening=None*, *mode_resolved=None*)

Generator method for the inverse life time as a function of energy.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – The energies to calculate the scattering rate for, given as an energy list.
- **energy_broadening** (*PhysicalQuantity* of type energy) – The energy broadening of delta-functions. **Default:** `3.0e-2*eV`
- **mode_resolved** (*bool.*) – Boolean determining if the phonon mode resolved data should be returned together with the mode summed result. **Default:** False.

Returns
The inverse life time. The shape is (n_energies,), where n_energies is the number of energies. If `mode_resolved` is `True`, the phonon-mode resolved inverse life time will be returned as well with shape (n_modes, n_energies)

Return type
PhysicalQuantity of type inverse time.

generateEnergyDependentInverseMeanFreePath(*energies*, *energy_broadening=None*, *use_life_time=None*, *mode_resolved=None*)

Generator method for the inverse mean free path in the k-grid and as a function of energy.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – The energies to calculate the scattering rate for, given as an energy list.
- **energy_broadening** (*PhysicalQuantity* of type energy) – The energy broadening of delta-functions. **Default:** `3.0e-2*eV`
- **use_life_time** (*bool*) – Boolean determining if the inverse life time should be used for calculating the mean free path. If `False`, the inverse relaxation time will be used. **Default:** False.
- **mode_resolved** (*bool*) – Boolean determining if the phonon mode resolved data should be returned together with the mode summed result. **Default:** False.

Returns
The inverse mean free path. The shape is (n_energies,), where n_energies is the number of energies. If `mode_resolved` is `True`, the phonon-mode resolved inverse relaxation time will be returned as well with shape (n_modes, n_energies)

Return type
PhysicalQuantity of type inverse length.

generateEnergyDependentInverseRelaxationTime(*energies*, *energy_broadening=None*, *mode_resolved=None*)

Generator method for the inverse relaxation time as a function of energy.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – The energies to calculate the scattering rate for, given as an energy list.
- **energy_broadening** (*PhysicalQuantity* of type energy) – The energy broadening of delta-functions. **Default:** `3.0e-2*eV`
- **mode_resolved** (*bool.*) – Boolean determining if the phonon mode resolved data should be returned together with the mode summed result. **Default:** False.

Returns
The inverse relaxation time. The shape is (n_energies,), where n_energies is the number of energies. If `mode_resolved` is `True`, the phonon-mode resolved inverse relaxation time will be returned as well with shape (n_modes, n_energies)

Return type
PhysicalQuantity of type inverse time.

hallCoefficientTensor()

Returns
The calculated hole- and electron Hall-coefficient tensors as the first and second element of the first entry, respectively. The shape is (carrier type, induced electric field direction, applied charge current density direction, applied magnetic field direction).

Return type
PhysicalQuantity with the unit `Meter**3 / Coulomb`

hallConductivityTensor()

Returns
The calculated hole- and electron Hall-conductivity tensors as the first and second element of the first entry, respectively. The shape is (carrier type, charge current density direction, electric field direction, magnetic field direction).

Return type
PhysicalQuantity with the unit `Meter / Coulomb * Siemens**2`

integrationMethod()

:returns:The method to use for calculating the q-integral. :rtype `GaussianBroadening()` | `TetrahedronMethod`

inverseLifeTime()

Returns
The calculated inverse life time with the shape (spin, phonon modes, k-points, electron bands).

Return type
PhysicalQuantity of type inverse time

inverseMeanFreePath(*use_life_time=None*)

Calculates the inverse mean free path due to electron-phonon coupling.

Parameters
use_life_time (*bool.*) – Boolean determining if the inverse life time should be used for calculating the mean free path. If `False`, the inverse relaxation time will be used. **Default:** False.

Returns
A phonon mode, k-point, and electron band resolved mean free time.

Return type
PhysicalQuantity of type inverse time of shape (phonon modes, k-points, electron bands).

inverseRelaxationTime()

Returns
The calculated inverse relaxation time with the shape (spin, phonon modes, k-points, electron bands).

Return type
PhysicalQuantity of type inverse time

`inverseRelaxationTimeEnergyDependent()`

Returns
The calculated or user inputted inverse relaxation time with the shape (n_energies).

Return type
PhysicalQuantity of type inverse time

`kpoints()`

Returns
The fractional k-points. The shape is (k-points, 3).

Return type
list of lists of floats

`kpointsWeights()`

Returns
The unreduced fractional k-points. The shape is (k-points).

Return type
list of lists of floats

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`method()`

Returns
The method used for the calculation. Either `Full` or `Isotropic`.

Return type
NLFlag

`mobility(`*contribution=None*`)`

Parameters
`contribution` (`All` | `Sum` | `Hole` | `Electron`.) – The contribution to the mobility. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated mobility. If contribution is `Hole` or `Electron`, the hole or electron mobility is returned. If contribution is `All`, a list is returned with the calculated hole- and electron mobilities as the first and second element. In case of `Sum`, the total mobility is returned. Notice that the summed mobility is not necessarily equal to the sum of Hole and Electron contributions.

Return type
PhysicalQuantity with the unit `Meter**2 / (Volt * Second)`

`mobilityTensor(`*contribution=None*`)`

Parameters
`contribution` (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the mobility tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated mobility tensors. If contribution is `Hole` or `Electron`, the hole or electron mobility tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron mobility tensors as the first and second element. In case of `Sum`, the total mobility tensor is returned. Notice that the summed mobility tensor is not necessarily equal to the sum of Hole and Electron contributions.

Return type
PhysicalQuantity with the unit `Meter**2 / (Volt * Second)`

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

phononEnergies()

Returns
The phonon energies. The shape is (phonon modes in electron-phonon coupling, q-points).

Return type
PhysicalQuantity of type energy

phononModes()

Returns
The phonon modes included in the mobility calculation.

Return type
list of non-negative ints

qpoints()

Returns
The unreduced fractional q-points. The shape is (q-points, 3).

Return type
list of lists of floats

refinement()

Returns
The number of times the q-grid is refined in each direction.

Return type
Positive float

resistivityTensor(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the resistivity tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated resistivity tensor. If contribution is `Hole` or `Electron`, the hole or electron resistivity tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron resistivity tensors as the first and second element. In case of `Sum`, the total resistivity tensor is returned.

Return type
PhysicalQuantity with the unit `Meter / Siemens`

resistivityTensorConstantMeanFreePath(*mean_free_path*, *contribution=None*)

Parameters

- **mean_free_path** (*PhysicalQuantity* of unit length.) – The constant mean-free-path.
- **contribution** (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the resistivity tensor obtained from a constant mean-free-path. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated resistivity tensor assuming a constant mean-free-path. If contribution is `Hole` or `Electron`, the hole or electron resistivity tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron resistivity tensors as the first and second element. In case of `Sum`, the total resistivity tensor is returned.

Return type
PhysicalQuantity with the unit `Meter / Siemens`

resistivityTensorConstantRelaxationTime(*relaxation_time*, *contribution=None*)

Parameters

- **relaxation_time** (*PhysicalQuantity* of unit time) – The constant relaxation_time.
- **contribution** (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the resistivity tensor obtained from a constant relaxation time. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated resistivity tensor assuming a constant relaxation time. If contribution is `Hole` or `Electron`, the hole or electron resistivity tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron resistivity tensors as the first and second element. In case of `Sum`, the total resistivity tensor is returned. second element. In case of `Sum`, the total resistivity tensor is returned.

Return type
PhysicalQuantity with the unit `Meter / Siemens`

resistivityTensorTimesConstantMeanFreePath(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the resistivity tensor*MFP, where MFP is the constant mean-free-path. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated resistivity tensor*MFP. If contribution is `Hole` or `Electron`, the hole or electron resistivity tensor*MFP is returned. If contribution is `All`, a list is returned with the calculated hole- and electron resistivity tensors*MFP as the first and second element. In case of `Sum`, the total resistivity tensor*MFP is returned.

Return type
PhysicalQuantity with the unit `Meter^2 / Siemens`

resistivityTensorTimesConstantRelaxationTime(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the resistivity tensor*tau, where tau is the constant relaxation time. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated resistivity tensor*tau. If contribution is `Hole` or `Electron`, the hole or electron resistivity tensor*tau is returned. If contribution is `All`, a list is returned with the calculated hole- and electron resistivity tensors*tau as the first and second element. In case of `Sum`, the total resistivity tensor*tau is returned.

Return type
PhysicalQuantity with the unit `Meter * Second / Siemens`

seebeckCoefficient(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the Seebeck coefficient. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated Seebeck coefficient. If contribution is `Hole` or `Electron`, the hole or electron Seebeck coefficient is returned. If contribution is `All`, a list is returned with the calculated hole- and electron Seebeck coefficient as the first and second element. In case of `Sum`, the total Seebeck coefficient is returned.

Return type
PhysicalQuantity with the unit `Volt / Kelvin`

seebeckCoefficientTensor(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the Seebeck coefficient tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated Seebeck coefficient tensor. If contribution is `Hole` or `Electron`, the hole or electron Seebeck coefficient tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron Seebeck coefficient tensor as the first and second element. In case of `Sum`, the total Seebeck coefficient tensor is returned.

Return type
PhysicalQuantity with the unit `Volt / Kelvin`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

temperature()

Returns
The temperature for the thermal smearing of the Bose and Fermi distributions.

Return type
PhysicalQuantity of type temperature

thermalConductivity(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the electronic thermal conductivity. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated electronic thermal conductivity. If contribution is `Hole` or `Electron`, the hole or electron thermal conductivity is returned. If contribution is `All`, a list is returned with the calculated hole- and electron thermal conductivities as the first and second element. In case of `Sum`, the total thermal conductivity is returned.

Return type
PhysicalQuantity with the unit `J / (Second * Meter * Kelvin)`

thermalConductivityTensor(*contribution=None*)

Parameters
contribution (`All` | `Sum` | `Hole` | `Electron`) – The contribution to the electronic thermal conductivity tensor. This can be either `All`, `Sum`, `Hole`, or `Electron`. **Default:** `All`

Returns
The calculated electronic thermal conductivity tensor. If contribution is `Hole` or `Electron`, the hole or electron thermal conductivity tensor is returned. If contribution is `All`, a list is returned with the calculated hole- and electron thermal conductivity tensors as the first and second element. In case of `Sum`, the total thermal conductivity tensor is returned.

Return type
PhysicalQuantity with the unit `J / (Second * Meter * Kelvin)`

uniqueString()

Return a unique string representing the state of the object.

updateTransportCoefficients(*energies=None*, *inverse_relaxation_time=None*, *mobility_object=None*, *temperature=None*, *fermi_shift=None*, *energy_broadening=None*)

Function for updating the transport coefficients (mobility, conductivity, resistivity, Seebeck coefficient, Hall conductivity). This function can only be used if method=Isotropic.

Note that by calling this function you will change variables on the object it-self.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – List of energies for which to define the scattering rate.
- **inverse_relaxation_time** (*PhysicalQuantity* of type energy) – List of scattering rates corresponding to the provided energies, or a constant scattering rate. If a list is provided the length must be the same as the `energies`. If a `mobility_object` is also provided, the `inverse_relaxation_time` will be added to the scattering rate obtained from the `mobility_object`.
- **mobility_object** (*Mobility*) – A mobility object calculated with the `Full` method. This can be used to calculate the scattering rate. If a `inverse_relaxation_time` is also provided, the `inverse_relaxation_time` will be added to the scattering rate obtained from the `mobility_object`.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature to use.
- **fermi_shift** (*PhysicalQuantity* of type energy) – The Fermi shift.
- **energy_broadening** (*PhysicalQuantity* of type energy.) – The energy broadening of delta-functions to use

Returns
Dictionary with all the updated transport coefficients, or list of dictionaries if the input temperature and/or fermi_shift was a list (also returns the combined index list of fermi_shifts, temperature in this case).

Return type
dict.

useKpointSymmetry()

Returns
Whether the used k-points should be reduced by symmetries.

Return type
bool

velocitiesK()

Returns
The electron Bloch velocities for each k-point. The shape is (spin, k-points, electron bands, velocity components).

Return type
PhysicalQuantity with the unit `Meter / Second`

Usage Examples

Calculate the mobility of graphene including the full angular dependence of the scattering rates and with the electron-phonon coupling and relaxation time calculated from first-principles:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)

elements = [Carbon, Carbon]

fractional_coordinates = [[0.0           , 0.0           , 0.5],
                          [0.333333333333, 0.666666666667, 0.5]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# Hamiltonian derivatives
# ------------------------------------------------------------
hamiltonian_derivatives = HamiltonianDerivatives(
    bulk_configuration,
    repetitions=(5, 5, 1),
    )

# ------------------------------------------------------------
# Dynamical matrix
# ------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    repetitions=(5, 5, 1),
    max_interaction_range=10*Angstrom,
    )

# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Electron Phonon Coupling
# ------------------------------------------------------------
k_a = numpy.linspace(0.323333, 0.343333, 11)
k_b = numpy.linspace(0.323333, 0.343333, 11)
k_c = numpy.linspace(0, 0, 1)
kpoints = [[a, b, c] for a in k_a for b in k_b for c in k_c]

q_a = numpy.linspace(-0.02, 0.02, 11)
q_b = numpy.linspace(-0.02, 0.02, 11)
q_c = numpy.linspace(0, 0, 1)
qpoints = [[a, b, c] for a in q_a for b in q_b for c in q_c]

electron_phonon_coupling = ElectronPhononCoupling(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    hamiltonian_derivatives=hamiltonian_derivatives,
    kpoints_fractional=kpoints,
    qpoints_fractional=qpoints,
    electron_bands=[3, 4],
    phonon_modes=All,
    energy_tolerance=0.03*eV,
    initial_state_energy_range=[-100,100]*eV,
    store_dense_coupling_matrices=False,
    )

# ------------------------------------------------------------
# Mobility
# ------------------------------------------------------------
mobility = Mobility(
    configuration=bulk_configuration,
    method=Full,
    electron_phonon_coupling=electron_phonon_coupling,
    )

nlsave('graphene_method_full.nc', mobility)
```

`mobility.py`

Note that the above script only shows the principal work flow for calculating the mobility. For converged calculations, one will often need to use more k-points and q-points in the *ElectronPhononCoupling* object - see the paper[1] for details. In order to perform a mobility calculation using the `Full` method, one needs to input an *ElectronPhononCoupling* object. While this will result in accurate calculations including the full angular dependece of the scattering rates and relaxation time, it is also a numerically challenging task because the *ElectronPhononCoupling* object needs to be calculated with many k- and q-points and in addition one needs the *DynamicalMatrix* and *HamiltonianDerivatives* objects.

An alternative approach to mobility calculations is available in QuantumATK. By specifying the method keyword to `Isotropic` and providing an inverse relaxation time like:

```
mobility = Mobility(
    bulk_configuration,
```

(continues on next page)

[1] T.ăGunst, T.ăMarkussen, K.ăStokbro, and M.ăBrandbyge. First-principles method for electron-phonon coupling and electron mobility: Applications to two-dimensional materials. *Phys. Rev. B*, 93:035414, Jan 2016. doi:10.1103/PhysRevB.93.035414.

(continued from previous page)

```
    method=Isotropic,
    kpoints=MonkhorstPackGrid(10, 10, 10),
    energies=numpy.linspace(-0.5, 0.5, 100)*eV,
    inverse_relaxation_time=1e10*Second**-1)
```

it is possible to perform much faster calculations, since it is not required to have an *ElectronPhononCoupling*. Using the `Isotropic` method the calculation of both the *DynamicalMatrix* and *HamiltonianDerivatives* objects can also be avoided.

Using the `Isotropic` method will require knowledge of the inverse relaxation time (or scattering rate). This can either be specified as a constant `inverse_relaxation_time=1e10 * Second**-1` or as an array with the same length as the provided energies `inverse_relaxation_time=numpy.linspace(0, 1e12, 100) * Second**-1`. Knowledge of the inverse relaxation time can be obtained from the literature or it can be approximated from a previous Mobility calculation using the `Full` method, i.e. calculated from an *ElectronPhononCoupling* object. This can be obtained as:

```
# Load a mobility object calculated with the Full method.
mobility = nlread('graphene_method_full.nc', Mobility)[0]

# Define an energy range.
energies=numpy.linspace(-0.5, 0.5, 100)*eV

# Calculate the inverse relaxation time.
inverse_relaxation_time = mobility.
↪generateEnergyDependentInverseRelaxationTime(energies=energies)
```

The obtained inverse relaxation time can be input directly to a new Mobility calculation with the `Isotropic` method. Alternatively, the old Mobility object can be provided as input to a new Mobility calculation:

```
# Load a mobility object calculated with the Full method.
mobility_full = nlread('graphene_method_full.nc', Mobility)[0]

# Perform and new mobility calculation with the Isotropic method.
mobility = Mobility(
    bulk_configuration,
    method=Isotropic,
    energies=numpy.linspace(-0.5,0.5,100)*eV,
    mobility_object=mobility_full)
```

If both the `inverse_relaxation_time` and `mobility_object` are provided, the resulting inverse relaxation time will be the sum of the provided and the one generated from the `mobility_object`.

A full script for a Mobility calculation using the `Isotropic` method is provided below using the constant inverse relaxation time approximation:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)

elements = [Carbon, Carbon]

fractional_coordinates = [[0.0           , 0.0           , 0.5],
                          [0.333333333333, 0.666666666667, 0.5]]
```

(continues on next page)

(continued from previous page)

```
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ----------------------------------------------------------------
# Calculator
# ----------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()


# ----------------------------------------------------------------
# Mobility
# ----------------------------------------------------------------
kpoint_grid = RegularKpointGrid(
    ka_range=[0.32, 0.34],
    kb_range=[0.32, 0.34],
    kc_range=[0.0, 0.0],
    na=15,
    nb=15,
    nc=1,
    )

mobility_constant_relaxation_time = Mobility(
    configuration=bulk_configuration,
    method=Isotropic,
    energies=numpy.linspace(-0.05, 0.05, 100)*eV,
    inverse_relaxation_time=1e12*Second**-1,
    electron_bands=[3,4],
    kpoints=kpoint_grid,
    temperature=300*Kelvin,
    )

nlsave('graphene_constant_relaxation_time.nc', mobility_constant_relaxation_time)
```

mobility_isotropic_scattering_rate.py

The script below shows an example of how to calculate the mobility with an energy dependent inverse relaxation time:

```
# ----------------------------------------------------------------
# Bulk configuration
# ----------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)

elements = [Carbon, Carbon]

fractional_coordinates = [[0.0           , 0.0           , 0.5],
                          [0.333333333333, 0.666666666667, 0.5]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 1),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()


# ------------------------------------------------------------
# Mobility
# ------------------------------------------------------------
kpoint_grid = RegularKpointGrid(
    ka_range=[0.32, 0.34],
    kb_range=[0.32, 0.34],
    kc_range=[0.0, 0.0],
    na=15,
    nb=15,
    nc=1,
    )

# Setup energy list.
energies = numpy.linspace(-0.05, 0.05, 100)*eV,

# Setup inverse relaxation time as an array with the same length as the energies.
inverse_relaxation_time = abs(numpy.linspace(-1e12, 1e+12, 100))*Second**-1

mobility_linear_relaxation_time = Mobility(
    configuration=bulk_configuration,
    method=Isotropic,
    energies=energies,
    inverse_relaxation_time=inverse_relaxation_time,
    electron_bands=[3,4],
```

(continues on next page)

(continued from previous page)

```
    kpoints=kpoint_grid,
    temperature=300*Kelvin,
    )

nlsave('graphene_linear_relaxation_time.nc', mobility_linear_relaxation_time)
```

mobility_isotropic_scattering_rate2.py

Notes

The mobility module in QuantumATK enables you to calculate the phonon-limited mobility using the semi-classical Boltzmann transport equation (BTE). There are two principle methods to use:

- In order to calculate the mobility from first-principles including the full angular dependence of the electron-phonon scattering, you must use the `Full` method. In that case you need a *BulkConfiguration* and a *ElectronPhononCoupling* object. See Ref.[Page 3352, 1] to get a description of the background theory and results obtained with the Mobility object.
- Assuming an isotropic inverse relaxation time, one can avoid integrations over k,q-space and instead integrate over energy. This leads to much faster mobility calculations since the *ElectronPhononCoupling* object is no longer needed. Instead you need to provide an inverse relaxation time to the Mobility calculation. The implementation in QuantumATK of the isotropic scattering rate method essentially follows BoltzTrap, Ref.[2]. Major differences to the description in Ref.[2] are:

(i) QuantumATK allows for an energy dependent inverse relaxation time.

(ii) The band velocities in QuantumATK are calculated using first order pertubation theory and are thus exact and without any problems at band crossings.

(iii) The energy integrations are performed using the tetrahedron method[3].

ModelChargeCorrection

class ModelChargeCorrection(*gaussian_width=None*, *optimize_width=None*, *averaging_distance_from_defect=None*)

A container for parameters for a Gaussian model charge correction for calculations of a charged point defect in a supercell of bulk material.

Parameters

- **gaussian_width** (*PhysicalQuantity* of type length.) – The width of the Gaussian model charge. If None, the correction is skipped as long as `optimize_width` is `False`. Otherwise the width is optimized internally. **Default:** None.
- **optimize_width** (*bool.*) – Whether the Gaussian charge width should be internally optimized to minimize the potential difference standard deviation in the averaging region. The parameter cannot be set to True if gaussian_width is set. **Default:** True if gaussian_width is None, False otherwise.
- **averaging_distance_from_defect** (*float | PHYSICALQUANTITY| of type length*) – The distance from the defect center beyond which the average potential is calculated. The correction scheme assumes that in this averaging region the electrostatic potential is approximately equal to that given by a Gaussian charge distribution at the defect center. The averaging distance can either be given in fractional coordinates relative to the longest unit cell vector

[2] GeorgăK.H. Madsen and DavidăJ. Singh. BoltzTraP. A code for calculating band-structure dependent quantities. *Computer Physics Communications*, 175(1):67–71, 2006. doi:http://dx.doi.org/10.1016/j.cpc.2006.03.007.

[3] P.ăE. Blochl, O.ăJepsen, and O.ăK. Andersen. Improved tetrahedron method for Brillouin-zone integrations. *Phys. Rev. B*, 49:16223–16233, Jun 1994. doi:10.1103/PhysRevB.49.16223.

of the supercell, or as a fixed value in Cartesian coordinates. The value cannot exceed half of the supercell length, i.e., in fractional coordinates it must be between 0 and 0.5. **Default:** `5.0 * Angstrom`

averagingDistanceFromDefect()

Returns
The parameter used to define the range of the potential for averaging.

Return type
float | *PhysicalQuantity* of type length

gaussianWidth()

Returns
The width of the Gaussian model charge.

Return type
PhysicalQuantity of type length

optimizeWidth()

Returns
Whether the width is obtained by optimization.

Return type
bool

uniqueString()
Return a unique string representing the state of the object.

ModifiedStiwe2Potential

class ModifiedStiwe2Potential(*particleType1*, *particleType2*, *a*, *b*, *c*, *d*, *e*, *f*, *alpha*, *beta*, *gamma*, *r_i*, *r_cut=None*)
Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of type energy*) – Potential parameter.
- **b** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c** (*PhysicalQuantity of type energy*) – Potential parameter.
- **d** (*PhysicalQuantity of type energy / length**5*) – Potential parameter.
- **e** (*PhysicalQuantity of type energy / length**4*) – Potential parameter.
- **f** (*PhysicalQuantity of type energy / length**3*) – Potential parameter.
- **alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **beta** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **gamma** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – Inner cutoff radius
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for selenium by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
lattice = Hexagonal(4.35517*Angstrom, 4.94945*Angstrom)

# Define elements
elements = [Selenium, Selenium, Selenium]

# Define coordinates
fractional_coordinates = [[0.217, -0.   , 0.166666666667],
                          [0.0  ,  0.217, 0.833333333333],
                          [0.783,  0.783, 0.5           ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='Oligschleger_Se_1996')
potentialSet.addParticleType(ParticleType(
    symbol='Se',
    mass=78.96*atomic_mass_unit,
    atomicNumber=34
))
```

(continues on next page)

(continued from previous page)

```
potential = ModifiedStiwe2Potential(
    particleType1='Se',
    particleType2='Se',
    a=29827.6728838*eV,
    b=0.861355523673*eV,
    c=-53.3454232425*eV,
    d=0.36561835*eV/Angstrom**5,
    e=1.5708619*eV/Angstrom**4,
    f=2.1702274*eV/Angstrom**3,
    alpha=-4.5617982906*1/Angstrom,
    beta=-4.44415371795e-05*1/Angstrom,
    gamma=-1.06468623932*1/Angstrom,
    r_i=2.80029917726*Angstrom,
    r_cut=4.14794315632*Angstrom
)
potentialSet.addPotential(potential)
potential = ModifiedStiwe3Potential(
    particleType1='Se',
    particleType2='Se',
    particleType3='Se',
    a=28.3766542324*eV,
    b=-8.33395695839*eV,
    c=-22.2984447633*eV,
    d=0.044142311935*eV/Angstrom**5,
    e=0.0429066410484*eV/Angstrom**4,
    f=-0.126000984941*eV/Angstrom**3,
    g=10.7308750893*1/eV,
    h=3.72061089286*1/eV,
    alpha=-0.271976653846*1/Angstrom,
    beta=-0.893447393162*1/Angstrom,
    gamma=-0.211685952991*1/Angstrom,
    omega=95.3688,
    r_i=2.80029917726*Angstrom,
    r_cut=4.1129394166*Angstrom
)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *ModifiedStiwe2Potential* and *ModifiedStiwe3Potential* classes implement a modified version of the original Stillinger-Weber potentials *Stiwe2Potential* and *Stiwe3Potential*[1], as described in[2].

ModifiedStiwe3Potential

class ModifiedStiwe3Potential(*particleType1*, *particleType2*, *particleType3*, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *alpha*, *beta*, *gamma*, *omega*, *r_i*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of type energy*) – Potential parameter.
- **b** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c** (*PhysicalQuantity of type energy*) – Potential parameter.
- **d** (*PhysicalQuantity of type energy / length**5*) – Potential parameter.
- **e** (*PhysicalQuantity of type energy / length**4*) – Potential parameter.
- **f** (*PhysicalQuantity of type energy / length**3*) – Potential parameter.
- **g** (*PhysicalQuantity of type energy**-1*) – Potential parameter.
- **h** (*PhysicalQuantity of type energy**-1*) – Potential parameter.
- **alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **beta** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **gamma** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **omega** (*Physical Quantity of type angle*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – Inner cutoff radius
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

[2] C.ăOligschleger, R.ăO. Jones, S.ăM. Reimann, and H.ăR. Schober. Model interatomic potential for simulations in selenium. *Phys. Rev. B*, 53:6165–6173, 1996. URL: http://link.aps.org/doi/10.1103/PhysRevB.53.6165, doi:10.1103/PhysRevB.53.6165.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for selenium by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
lattice = Hexagonal(4.35517*Angstrom, 4.94945*Angstrom)

# Define elements
elements = [Selenium, Selenium, Selenium]

# Define coordinates
fractional_coordinates = [[0.217, -0.   , 0.166666666667],
                          [0.0  ,  0.217, 0.833333333333],
                          [0.783,  0.783, 0.5           ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='Oligschleger_Se_1996')
potentialSet.addParticleType(ParticleType(
    symbol='Se',
    mass=78.96*atomic_mass_unit,
    atomicNumber=34
))

potential = ModifiedStiwe2Potential(
    particleType1='Se',
    particleType2='Se',
    a=29827.6728838*eV,
    b=0.861355523673*eV,
    c=-53.3454232425*eV,
    d=0.36561835*eV/Angstrom**5,
    e=1.5708619*eV/Angstrom**4,
    f=2.1702274*eV/Angstrom**3,
```

(continues on next page)

(continued from previous page)

```
    alpha=-4.5617982906*1/Angstrom,
    beta=-4.44415371795e-05*1/Angstrom,
    gamma=-1.06468623932*1/Angstrom,
    r_i=2.80029917726*Angstrom,
    r_cut=4.14794315632*Angstrom
)
potentialSet.addPotential(potential)
potential = ModifiedStiwe3Potential(
    particleType1='Se',
    particleType2='Se',
    particleType3='Se',
    a=28.3766542324*eV,
    b=-8.33395695839*eV,
    c=-22.2984447633*eV,
    d=0.044142311935*eV/Angstrom**5,
    e=0.0429066410484*eV/Angstrom**4,
    f=-0.126000984941*eV/Angstrom**3,
    g=10.7308750893*1/eV,
    h=3.72061089286*1/eV,
    alpha=-0.271976653846*1/Angstrom,
    beta=-0.893447393162*1/Angstrom,
    gamma=-0.211685952991*1/Angstrom,
    omega=95.3688,
    r_i=2.80029917726*Angstrom,
    r_cut=4.1129394166*Angstrom
)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *ModifiedStiwe2Potential* and *ModifiedStiwe3Potential* classes implement a modified version of the original Stillinger-Weber potentials *Stiwe2Potential* and *Stiwe3Potential*[1], as described in[2].

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

[2] C.ăOligschleger, R.ăO. Jones, S.ăM. Reimann, and H.ăR. Schober. Model interatomic potential for simulations in selenium. *Phys. Rev. B*, 53:6165–6173, 1996. URL: http://link.aps.org/doi/10.1103/PhysRevB.53.6165, doi:10.1103/PhysRevB.53.6165.

MolecularConfigurationsParameters

class MolecularConfigurationsParameters(*reference_configurations*, *sample_size=None*, *monte_carlo_calculator=None*, *monte_carlo_temperature=None*, *max_rotated_bonds=None*, *overlap_ratio=None*, *max_displacement=None*, *sample_cis_trans=None*, *generate_bulk_configurations=None*, *random_seed=None*, *log_filename_prefix=None*, *data_tag=None*)

Class for storing parameters for generating a set of training configurations using a combination of sampling different torsion angles and atomic displacements.

Parameters

- **reference_configurations** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | sequence of configurations) – One or more reference configurations to be used to generate the training set.
- **sample_size** (*int*) – The total number of configurations to return. **Default:** 20
- **monte_carlo_calculator** (*Calculator* | None) – The calculator to use for the molecular dynamics simulations. Giving None disables Monte Carlo structure selection. **Default:** None.
- **monte_carlo_temperature** (PhysicalQuantity of type temperature) – Temperature used in the Monte Carlo selection. **Default:** 300K.
- **max_rotated_bonds** (*int*) – The maximum number of bonds rotated in each new configuration. **Default:** Rotate all available bonds in each configuration.
- **overlap_ratio** (*float*) – The acceptable VDW radius overlap between two atoms. **Default:** 0.7.
- **max_displacement** (PhysicalQuantity of type length) – The maximum displacement in the atomic random displacement. **Default:** 0.1 Angstrom.
- **sample_cis_trans** (*bool*) – Whether or not cis and trans orientations of SP2 bonds are sampled. **Default:** True.
- **generate_bulk_configurations** (*bool*) – Whether or not the returned dataset contains bulk configurations. **Default:** False.
- **random_seed** (*int*) – The random seed used for generating the displacements. **Default:** Generated automatically.
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting.

dataTag()

Returns
The selection tag added to the data in the training set.

Return type
str

generateBulkConfigurations()

Returns
Whether or not the returned configurations are Bulk or Molecules.

Return type
bool

logFilenameIdentifier()

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

logFilenamePrefix()

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | LogToStdOut | None

maxDisplacement()

Returns
The maximum random displacement for each atom.

Return type
PhysicalQuantity

maxRotatedBonds()

Returns
The maximum number of bonds rotated in each trial configuration.

Return type
int

monteCarloCalculator()

Returns
The calculator used to calculate Monte Carlo energies.

Return type
Calculator | None

monteCarloTemperature()

Returns
The temperature used in the Monte Carlo selection

Return type
PhysicalQuantity of type temperature

overlapRatio()

Returns
The minimum acceptable VDW radii overlap ratio.

Return type
float

randomSeed()

Returns
The random seed used in the random number generator.

Return type
int | None

referenceConfigurations()

Returns
The list of reference configurations to be used to generate the training set.

Return type
list of [*MoleculeConfiguration* | *BulkConfiguration*]

sampleCisTrans()

Returns
Whether or not cis and trans orientations of SP2 bonds are sampled.

Return type
bool

sampleSize()

Returns
The number of training configurations for each combination of list parameters.

Return type
int

strainTensors()

Returns
The list of strains to apply to each configuration (None for no strain).

Return type
list of [numpy.ndarray | None]

supercellRepetitionsList()

Returns
The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
list of tuple (size 3) of int

systemSizes()

Parameters
system_sizes (*int | sequence of int*) – Minimum number of atoms in the system. If None then supercell_repetitions_list will be used to do the repetition of the cell. If set, supercell_repetitions_list will be ignored. **Default:** None

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Setup of a training set of C_2F_4 and C_2F_6 using *MolecularConfigurationsParameters*.

```
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Fluoroethane configuration
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Define elements
elements = [Carbon, Carbon, Fluorine, Fluorine, Fluorine, Fluorine, Fluorine, Fluorine]

# Define coordinates
cartesian_coordinates = [[-0.003263937084, -0.            ,  0.            ],
                         [ 1.516263937084, -0.            , -0.            ],
                         [-0.469715381813,  0.            , -1.301600235398],
                         [-0.469715381813, -1.127218869426,  0.650800117699],
                         [-0.469715381813,  1.127218869426,  0.650800117699],
                         [ 1.982715381813,  1.127218869426, -0.650800117699],
                         [ 1.982715381813,  0.            ,  1.301600235398],
                         [ 1.982715381813, -1.127218869426, -0.650800117699]]*Angstrom

# Set up configuration
C2F6_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)
C2F6_configuration.findBonds()

# ------------------------------------------------------------
# Fluoroethene configuration
# ------------------------------------------------------------
# Define elements
elements = [Carbon, Carbon, Fluorine, Fluorine, Fluorine, Fluorine]

# Define coordinates
cartesian_coordinates = [[ 0.089971642147,  0.            , -0.            ],
                         [ 1.423028357853,  0.            ,  0.            ],
                         [-0.594324695679,  0.            , -1.173915650421],
                         [-0.594324695679,  0.            ,  1.173915650421],
                         [ 2.107324695679,  0.            ,  1.173915650421],
                         [ 2.107324695679,  0.            , -1.173915650421]]*Angstrom

# Set up configuration
C2F4_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)
C2F4_configuration.findBonds()

# ------------------------------------------------------------
# Specify molecular training set
# ------------------------------------------------------------
training_set = MolecularConfigurationsParameters(
    [C2F6_configuration, C2F4_configuration],
    sample_size=100,
    generate_bulk_configurations=True
)

# ------------------------------------------------------------
# Specify fitting parameters
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
nl_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=False,
)

fitting_parameters = MomentTensorPotentialFittingParameters(
    inner_cutoff_radii=1.0*Angstrom,
    tapering_cutoff_radii=1.1*Angstrom,
    outer_cutoff_radii=5*Angstrom,
    mtp_filename='Carbon_Fluorine_MTP.mtp',
    non_linear_coefficients_parameters=nl_parameters,
)

# ------------------------------------------------------------
# Generate the MTP
# ------------------------------------------------------------
mtp = MomentTensorPotentialTraining(
    'Carbon_Fluorine_MTP.hdf5',
    'Carbon_Fluorine_MTP',
    [training_set],
    calculator=LCAOCalculator(),
    fitting_parameters_list=[fitting_parameters],
    train_test_split=0.9,
)
mtp.update()
```

`Molecule_MTP.py`

This script generates 100 configurations each of C_2F_4 and C_2F_6. The energy, force and stress of each configuration is then calculated using density functional theory. This data is then used to fit an MTP potential with settings specified with *MomentTensorPotentialFittingParameters*. The *MomentTensorPotentialTraining* study object containing the data and fitting statistics is in the file `Carbon_Fluorine_MTP.hdf5`. The fitted MTP potential is in the file `Carbon_Fluorine_MTP.mtp`. A calculator using this MTP potential can be returned from the *MomentTensorPotentialTraining* object using the method `momentTensorPotentialCalculators`.

Notes

The *MolecularConfigurationsParameters* generates a trajectory of configurations that can be used to train a moment tensor potential. Starting with one or more configurations as input, new configurations are generated by rotating available bond torsions and adding random displacements to the atoms. Bonds that are part of cyclic structures, such as the carbon-carbon bond in cyclohexane are not freely rotated so as to not break the ring structure. Double and triple bonds are also not freely rotated because of their typically high rotational barriers. Double bonds are rotated over a smaller range centered on the planar geometry.

To avoid larger molecules being twisted onto itself while sampling the torsions, distances between non-bonded atoms are checked for each configuration. Configurations where inter-atomic distances are shorter than the factor of the van der Waals radius given by the parameter `overlap_ratio` are rejected. An additional Monte Carlo selection can also be performed on configurations with rotated torsions. A calculator for evaluating the configuration energy and a Monte Carlo selection temperature must be specified. This is done using the arguments `monte_carlo_calculator` and `monte_carlo_temperature` respectively. If a calculator is not given then then Monte Carlo selection is not performed.

The number of bonds rotated in each trial structure can be set using the argument `max_rotated_bonds`. Limiting the number of bonds rotated at each step can make it easier to find non-overlapping structures when fitting an MTP for a polymer or other large molecule. For configurations with double bonds, both cis and trans orientations of the bond are sampled by default. Setting the argument `sample_cis_trans` disables this sampling and turns off bond rotation for double bonds.

In some cases, such as fitting an MTP for a surface process simulations molecular and bulk training sets may be combined together. In these cases it is more convenient for the molecular training sets to also contain bulk configurations. Setting the argument `generate_bulk_configurations` to `True` generates a training set of molecules that are placed in periodic boundary conditions with a large buffer.

After defining the *MolecularConfigurationsParameters* it can be used with *MomentTensorPotentialTraining* for fitting an MTP potential.

MolecularDynamicsErrorPredictionHook

class MolecularDynamicsErrorPredictionHook(*mtp_error_prediction_parameters=None*)

Measurement hook for molecular dynamics simulations using MomentTensorPotentials with error prediction via query-by-committee models.

Parameters
mtp_error_prediction_parameters (*MTPErrorPredictionParameters*) – The parameters that specify the details of the error prediction for the MomentTensorPotential MD Error Prediction Hook Function.

callInterval()

Returns
The call interval of this hook function.

Return type
int

measurementNames()

Returns
The names of all measured properties.

Type
list of type str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up a molecular dynamics simulation with error prediction for the energy and forces values during the simulation of a cobalt disicilide structure using a pretrained MTP containing committee potentials.

```
# Set up configuration
cosi2 = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# Set tags for fixed atoms
cosi2.addTags('fixed', [0, 4, 12, 16])


# Load the pretrained MTP containing committee potentials and set up a calculator
potentialSet = QuantumATK_MTP_CoSi_2022_12()
calculator = TremoloXCalculator(parameters=potentialSet)


# Set Calculator
cosi2.setCalculator(calculator)
nlsave('CoSi2_MD_ErrorPrediction.hdf5', cosi2)


# The MolecularDynamicsErrorPredictionHook takes an MTPErrorPredictionParameters object␣
↪as input.
mtp_error_prediction_parameters = MTPErrorPredictionParameters(
    predict_energy_error=True,
    predict_forces_error=True,
    check_interval=1,
    write_atomic_error_estimates=True,
    stop_simulation_energy_threshold=None,
    stop_simulation_force_threshold=None
)

# MolecularDynamicsErrorPredictionHook
molecular_dynamics_error_prediction_hook = MolecularDynamicsErrorPredictionHook(
    mtp_error_prediction_parameters=mtp_error_prediction_parameters
)

# HookFunctions
pre_step_hooks = []
post_step_hooks = []
measurement_hooks = [
    molecular_dynamics_error_prediction_hook,
]
hook_functions = HookFunctions(
    pre_step_hooks=pre_step_hooks,
    post_step_hooks=post_step_hooks,
    measurement_hooks=measurement_hooks,
)
nlsave('CoSi2_MD_ErrorPrediction.hdf5', hook_functions)

# Constraints
fix_atom_indices_0 = cosi2.indicesFromTags(['fixed'])
constraints = [FixStrain(True, True, True), FixAtomConstraints(fix_atom_indices_0)]

# Set up MolecularDynamics with the error prediction hook
md_trajectory = MolecularDynamics(
    configuration=cosi2,
    constraints=constraints,
    log_interval=1,
```

(continues on next page)

(continued from previous page)

```
    trajectory_filename='CoSi2_MD_ErrorPrediction.hdf5',
    hook_functions=hook_functions,
    trajectory_object_id='md',
)
```

The script is available for download: `CoSi2_MD_ErrorPrediction.py`.

Notes

The *MolecularDynamicsErrorPredictionHook* class can be used to estimate errors of the energy and forces values determined via molecular dynamics simulations performed with MTPs containing committee models. The hook function uses a *query-by-committee* approach for estimating molecular dynamics energy and force errors which is consistent with the *ActiveLearningSimulation* extrapolation grade. The ensemble standard deviation of the forces prediction is used as a measure of extrapolation, which can be interpreted as a prediction of the forces error in the configuration[1].

MTPs with committee models can either be found in many of the pretrained MTPs or trained customly. The supported pretrained MTPs all contain 6 committee models and includes the phrasing *Supports MTP error prediction* in their descriptions. A custom MTP can be trained by setting a non-zero *committee_size* parameter in *MomentTensorPotentialTraining*. Training custom MTPs with committee models is not the default.

There are a number of options that can be set via the *MTPErrorPredictionParameters* class which influences the hook behaviour.

The error prediction results are stored in the MD trajectory output. In the GUI, the system-wide error predictions are, if computed, readily available in the MDTrajectory Analyzer under the names *Energy error prediction* and *Forces error prediction*. If present, the atomic force error predictions are available by opening the MDTrajectory with the Viewer and coloring the atoms via the *Atomic forces error predictions* option.

MolecularDynamicsSnapshotsParameters

class MolecularDynamicsSnapshotsParameters(*reference_configurations*, *supercell_repetitions_list=None*, *strain_tensors=None*, *calculator=None*, *sample_size=None*, *steps=None*, *method=None*, *log_filename_prefix=None*, *data_tag=None*)

Class for storing parameters for generating a set of training configurations using snapshots from a molecular dynamics simulation.

Parameters

- **reference_configurations** (*MoleculeConfiguration* | *BulkConfiguration* | sequence of [*MoleculeConfiguration* | *BulkConfiguration*]) – One or more reference configurations to be used to generate the training set. For bulk configurations, this is the unit cell from which a supercell can be defined and additional strain can be applied.
- **supercell_repetitions_list** (*sequence (size 3) of int | sequence of sequence (size 3) of int*) – The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. Cannot be specified if there are molecules in the list of reference configurations. **Default:** No repetitions.
- **strain_tensors** (*sequence (size 3) of sequence (size 3) of float | sequence of sequence (size 3) of sequence (size 3) of float*) – The list of strains to apply to each configuration. Cannot be specified if there are molecules in the list of reference configurations. **Default:** No strain.

[1] Christoph Schran, Krystof Brezina, and Ondrej Marsalek. Committee neural network potentials control generalization errors and enable active learning. *The Journal of Chemical Physics*, 153(10):104105, 2020.

- **calculator** (*Calculator*) – The calculator to use for the molecular dynamics simulations. **Default:** The same calculator used to fit the moment tensor potential.
- **sample_size** (*int*) – The number of training configurations to generate for each combination of reference configuration, supercell, and strain. **Default:** `20`
- **steps** (*int*) – The total number of molecular dynamics steps to take for each combination of reference configuration, supercell, and strain. **Default:** *10 * sample_size*
- **method** (`BaseMDmethod`) – The molecular dynamics method to use. **Default:** `Langevin(initial_velocity=MaxwellBoltzmannDistribution())`
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting.

calculator()

Returns
The calculator to use for the molecular dynamics simulation, or None for the same calculator used to fit the moment tensor potential.

Return type
Calculator | None

dataTag()

Returns
The selection tag added to the data in the training set.

Return type
str

logFilenameIdentifier()

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

logFilenamePrefix()

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | `LogToStdOut` | None

method()

Returns
The molecular dynamics method to use.

Return type
`BaseMDmethod`

referenceConfigurations()

Returns
The list of reference configurations to be used to generate the training set.

Return type
list of [*MoleculeConfiguration* | *BulkConfiguration*]

sampleSize()

Returns
The number of training configurations for each combination of list parameters.

Return type
int

steps()

Returns
The total number of molecular dynamics steps to take.

Return type
int

strainTensors()

Returns
The list of strains to apply to each configuration (None for no strain).

Return type
list of [numpy.ndarray | None]

supercellRepetitionsList()

Returns
The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
list of tuple (size 3) of int

systemSizes()

Parameters
system_sizes (*int | sequence of int*) – Minimum number of atoms in the system. If None then supercell_repetitions_list will be used to do the repetition of the cell. If set, supercell_repetitions_list will be ignored. **Default:** None

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Run a MD simulation of a quartz crystal via *MolecularDynamicsSnapshotsParameters* using a fast Tersoff force field for 50000 steps, and recalculate energy, forces, and stress for 50 equally spaced snapshots with the reference DFT calculator.

```
# Set up lattice
lattice = Hexagonal(4.966693338812276*Angstrom, 5.4647751809821905*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
fractional_coordinates = [[ 0.469217958328,  0.              ,  0.              ],
                          [ 0.              ,  0.469217958328,  0.666666666667],
                          [ 0.530782041672,  0.530782041672,  0.333333333333],
                          [ 0.409091887977,  0.269718793416,  0.115058361235],
                          [ 0.269718793416,  0.409091887977,  0.551608638765],
                          [ 0.730281206584,  0.139373094561,  0.781725361235],
                          [ 0.590908112023,  0.860626905439,  0.218274638765],
                          [ 0.860626905439,  0.590908112023,  0.448391361235],
                          [ 0.139373094561,  0.730281206584,  0.884941638765]]

# Set up reference configuration of quartz
reference_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates,
)

# Define the molecular dynamics method
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=1500.0*Kelvin,
    remove_center_of_mass_momentum=True,
    random_seed=None,
    enforce_temperature=True,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=1500*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

# Define the molecular dynamics calculator
potentialSet = Tersoff_SiO_2007()
calculator = TremoloXCalculator(parameters=potentialSet)

# Define the MD snapshots parameters:
# Run 50000 MD steps and take 50 equally spaced samples for the final training set.
training_sets = MolecularDynamicsSnapshotsParameters(
    reference_configuration,
    calculator=calculator,
    sample_size=50,
    steps=50000,
    method=method,
    log_filename_prefix=None,
)

# Set up MTP training and run the training data generation and calculation.
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_study.hdf5',
```

(continues on next page)

(continued from previous page)

```
    object_id='training',
    training_sets=training_sets,
    calculator=LCAOCalculator(),
    calculate_stress=True,
)
moment_tensor_potential_training.update()
```

MolecularDynamicsSnapshotsParameters_example.py

Notes

This class provides a convenient and automatic way to generate and use snapshots from MD simulations as training data. There are two ways this class can be used:

- Use the final reference calculator to run the MD. In this case the configurations are just taken from the MD and no re-calculation of energy, forces, and stress is needed.
- Use a fast and more approximate calculator (e.g. force field, semi-empirical, or fast DFT) for the MD simulation, and recalculate energy, forces, and stress only for the selected snapshots with the more accurate reference DFT calculator.

Since the snapshots from MD simulations are typically very correlated and similar to the previous images, using all available images from an MD simulation usually does not provide much additional information to the training of a ML force field. Therefore, one often has to run long MD to obtain a range of uncorrelated and different configurations.

Fortunately highly accurate DFT method are not normally required to obtain physically meaningful configurations for use as training data. A empirical potential with more approximate descriptions of the interactions is often sufficient for generating training configurations.

This procedure of using a more approximate FF or DFT method to quickly generate different relevant training configurations, and then using the final reference calculator for the training data of selected uncorrelated snapshots can be very efficient. It can especially work well when fitting MTPs for amorphous materials or interfaces, where sampling the range of possible configurations is a little more complicated.

If needed, additional relevant training data can be generated in a second step using the *ActiveLearningSimulation* method to refine the total training data set.

It is also possible to use different supercell sizes or apply different strain tensors to the reference configuration. In this case a separate MD simulation will be run for each combination the given values, each producing `sample_size` final training configurations.

This class always generates an MD trajectory. If one wants to use the trajectory from an existing MD simulation as training data, one should use the *TrainingSet* class instead.

MolecularEnergySpectrum

class MolecularEnergySpectrum(*configuration=None*, *energy_zero_parameter=None*, *projection_list=None*)

Class for calculating the molecular energy spectrum for a configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration the molecular energy spectrum should be calculated for.
- **energy_zero_parameter** (`FermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `FermiLevel`

- **projection_list** (*ProjectionList*) – Specifies the orbitals within the configuration to include in the calculation. **Default:** `ProjectionList(All)` - All atoms included.

degeneracy()

Returns
The spin degeneracy for the spectrum. That is, 2 for Unpolarized spin type, else 1.

Return type
float

determineHomoEnergy(*spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.All'>*)

Determine the HOMO energy for the molecule for the given spin channel. For polarized calculations with fixed spin momentum, the average of 2 Fermi levels is used.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin for which the HOMO energy should be returned. For `Spin.All`, the returned energy is the highest of all spin channels. **Default:** `Spin.All`

Returns
The molecular energy of HOMO.

Return type
PhysicalQuantity of type energy

determineHomoLumoGap(*spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.All'>*)

Determine the HOMO-LUMO gap for the molecule for the given spin channel. For polarized calculations with fixed spin momentum, the average of 2 Fermi levels is used.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin for which the LUMO energy should be returned. For `Spin.All`, the returned energy is the lowest of all spin channels. **Default:** `Spin.All`

Returns
The HOMO-LUMO gap energy.

Return type
PhysicalQuantity of type energy

determineLumoEnergy(*spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.All'>*)

Determine the LUMO energy for the molecule for the given spin channel. For polarized calculations with fixed spin momentum, the average of 2 Fermi levels is used.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin for which the LUMO energy should be returned. For `Spin.All`, the returned energy is the lowest of all spin channels. **Default:** `Spin.All`

Returns
The molecular energy of LUMO.

Return type
PhysicalQuantity of type energy

energyZero()

Return the energy zero used for the energy scale in this molecular energy spectrum.

Returns
The energy zero used for the energy scale in this molecular energy spectrum.

Return type
PhysicalQuantity or type energy

`evaluate`(*spin=None*)

Return the molecular energy spectrum.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin for which the spectrum should be returned. For noncollinear and spin-orbit calculations, this must be `Spin.All`. **Default:** `Spin.Up` for unpolarized calculations, else `Spin.All`

Returns
The molecular energy spectrum as a *PhysicalQuantity* with one entry for each molecular eigenstate.

Return type
PhysicalQuantity of type energy

`fermiLevel`(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. Must be either `Spin.Up`, `Spin.Down`, or `Spin.All`. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** `Spin.Up`

Returns
The Fermi level in absolute energy.

Return type
`PhysicalQuantity` of type energy

`fermiTemperature`()

Return the Fermi temperature used in this molecular energy spectrum.

Returns
The Fermi temperature.

Return type
PhysicalQuantity or type temperature

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`occupation`()

Method for calculating the occupation for these eigenstates.

Returns
The occupation for each orbital.

Return type
list of float

`projectionList`()

Return the *ProjectionList* object used in this molecular energy spectrum.

Returns
The *ProjectionList* object.

Return type
ProjectionList

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the molecular energy spectrum of an ammonia molecule and print out the energy levels and occupations:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates= [[ 0.      ,  0.      ,  0.124001],
                            [ 0.      ,  0.941173, -0.289336],
                            [ 0.81508 , -0.470587, -0.289336],
                            [-0.81508 , -0.470587, -0.289336]]*Angstrom )

# Define the calculator
calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the molecular energy spectrum
molecular_energy_spectrum = MolecularEnergySpectrum(
    configuration=molecule_configuration,
    energy_zero_parameter=AbsoluteEnergy,
    )
nlsave('molecular.hdf5', molecular_energy_spectrum)

# Extract the energies and occupations
energy = molecular_energy_spectrum.evaluate()
occ = molecular_energy_spectrum.occupation()

# ... and print them out
print("level   energy(eV)    occupation")
for i in numpy.arange(len(energy)):
    print('  %d  %12.4f  %12.4f  ' % (i, energy[i].inUnitsOf(eV),occ[0][i]))
```

nh3.py

Notes

•To export the data of a *MolecularEnergySpectrum*, use the method *nlprint*.

•The *MolecularEnergySpectrum* can be visualized with QuantumATK when it is saved with *nlsave*.

MolecularOrderParameter

class MolecularOrderParameter(*md_trajectory*, *polymer_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *gui_worker=None*)

Analyzer for the rotational and translational order in a molecular structure.

Parameters

•**md_trajectory** (*MDTrajectory* | *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The MDTrajectory or AtomicConfiguration that contains the polymers.

•**polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used. Tags can also be set automatically with the flag Automatic

•**start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.

•**end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.

•**time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

polymerTags()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

rotationalOrder()

Returns
The rotational order parameter for each image in the given trajectory.

Return type
ndarray

times()

Returns
Times for each image used in the analysis.

Return type
PhysicalQuantity of type time

translationalOrder()

Returns
The translational order parameter for each image in the given trajectory.

Return type
ndarray

Usage Examples

Calculate the translational and rotational order of a polystyrene melt during a molecular dynamics simulation starting from a non-equilibrated structure.

```
from NL.GUI.Graphics.Plotter.API import *

# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('Polystyrene.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Polystyrene_Trajectory.hdf5',
    steps=100000,
    log_interval=10,
    trajectory_interval=1000,
    method=method
)

# ------------------------------------------------------------
# Calculate Order Parameters
# ------------------------------------------------------------
analyzer = MolecularOrderParameter(md_trajectory)
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
model = PlotModel(ps)
model.axis('x').setLabel('Time')
model.axis('y').setLabel('Order Parameter')
model.legend().setVisible(True)
colors = ['red', 'blue']

line = Line(analyzer.times(), analyzer.rotationalOrder())
line.setLabel('Rotation')
line.setColor('red')
model.addItem(line)

line = Line(analyzer.times(), analyzer.translationalOrder())
line.setLabel('Translation')
line.setColor('blue')
model.addItem(line)

model.setLimits('x', 0*ps, 100*ps)
model.setLimits('y', 0, 1)

save(model, 'MolecularOrderParameter_Plot.png')
```

`MolecularOrderParameter_Example.py` `Polystyrene.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate the molecular order for the polymer system. The resulting order is shown in the plot below.

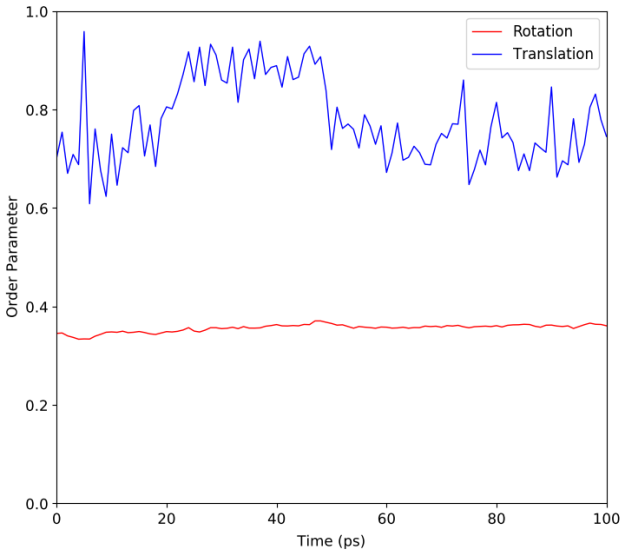


Fig. 4.33: Molecular rotational and translational parameters for polystyrene structure.

Notes

This class enables the calculation of the relative rotational and translational order of either molecules or polymer chains from a *MDTrajectory* or *BulkConfiguration*.

Molecules in a bulk system can be both translationally or rotationally ordered. Translational order relates to the ability for molecules to form regularly spaced layers along a particular direction. Likewise rotational order relates to the extent that molecules in the bulk structure are aligned together[1]. To calculate the ordering of a molecular system, the vector representing each molecule is first found by diagonalizing the inertial tensor $\mathbf{I}$, which is defined as

$$I_{ab} = \sum_{j=1}^{N} m_j \left(r_j^2 \delta_{ab} - r_{ja} r_{jb} \right)$$

Here $\mathbf{r}_j$ is the distance from the molecule center of mass to the jth atom and m_j is the mass of that atom. The long axis of the molecule that defines its direction is the eigenvector associated with the smallest eigenvalue. With the direction of each molecule defined, an overall ordering direction can be found similarly by diagonalizing the second-rank ordering tensor $\mathbf{Q}$, which is given as:

$$Q_{ab} = \frac{1}{N} \sum_{j=1} N \left(\frac{3}{2} u_{ja} u_{jb} - \frac{1}{2} \delta_{ab} \right)$$

Here $\mathbf{u}_j$ is a unit vector representing the direction of the jth molecule. The nematic direction is then the eigenvector associated with the largest eigenvalue of the ordering tensor. Given the unit vectors $\mathbf{u}_j$ and $\mathbf{n}$ representing the direction of each molecule and the overall ordering direction respectively, the rotational ordering ω is given as:

$$\omega = \frac{3}{2} \left\langle (\mathbf{u}_j \cdot \mathbf{n})^2 \right\rangle - \frac{1}{2}$$

The translational ordering τ can also be given as

$$\tau = \left\langle \exp\left(\frac{2\pi i}{d} \right) \mathbf{u}_j \cdot \mathbf{n} \right\rangle_j$$

Here d represents the interlayer spacing. The interlayer spacing is found by determining the spacing that maximizes the translational order.

Both rotational and translational ordering parameters lie in the range 0 to 1 for completely disordered materials and fully ordered materials, respectively.

To calculate the molecular order a *MolecularOrderParameter* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration*. The specific images in the *MDTrajectory* to be analyzed can be selected using the `start_time`, `end_time` and `time_resolution` arguments. The molecules analyzed in the configuration can be specified with the `polymer_tags` argument. This takes a list of tags of polymer molecules in the configuration. By default polymers tagged with `POLYMER_MOLECULE_#` are analyzed, where `#` is a numerical index. For molecular systems it is also possible to give the flag `Automatic` for the `polymer_tags` argument. This specifies that the molecular bond graph, rather than explicit tags, are to be used to determine which atoms belong to each molecule.

Once the `MolecularOrderParmeter` object is created and the values of each of the descriptors has been calculated, the values can be queried using the remaining functions. The method `rotationalOrder` returns the numpy array of the rotational order parameters for each image in the trajectory. Likewise the `translationalOrder` method returns a numpy array with the translational ordering values.

[1] Mineăllk Caper and Emine Cebe. Molecular dynamics study of the odd-even effect in some 4-n-alkyl-4-cyanobiphenyls. *Physical Review E*, 73:061711, 2006. URL: https://journals.aps.org/pre/abstract/10.1103/PhysRevE.73.061711, doi:10.1103/PhysRevE.73.061711.

MolecularReactionPath

class MolecularReactionPath(*reaction_path*, *filename*, *object_id*, *calculator=None*, *fuzz_factor=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*, *optimize_geometry_parameters=None*, *dynamical_matrix_parameters=None*)

Class for estimating the reactant, transition and product states of a NudgedElasticBand. The product state can only be bimolecular.

Parameters

- **reaction_path** (*NudgedElasticBand*) – The reaction path to investigate the best transition state estimate on. The reaction path can only contain MoleculeConfigurations.
- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **calculator** (Calculator | None) – A calculator to attach. **Default:** Use the calculator on the reaction path.
- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond. **Default:** 1.2
- **log_filename_prefix** (str | LogToStdOut) – Filename prefix for the logging output of magnetic anisotropy energy calculations. If LogToStdOut, all logging will instead be sent to standard output. **Default:** 'molecular_reaction_path_'.
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters for the geometry optimization.
- **dynamical_matrix_parameters** (DynamicalMatrixParameters) – The parameters for the DynamicalMatrix.

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

dynamicalMatrixParameters()

Returns
The parameters for the dynamical matrix.

Return type
DynamicalMatrixParameters

filename()

Returns
The filename where the study object is stored.

Return type
str

firstProductMolecule()

Returns
The first reaction product state.

Return type
IdealGasThermoChemistry

fuzzFactor()

Returns
The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.

Return type
float

hasTightBarrier()

Returns
None if the study has not been updated, True if a tight barrier was found.

Return type
None | bool

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlinfo()

Returns
Structured information about the DynamicalMatrix.

Return type
dict

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometryParameters()

Returns
The parameters for the geometry optimization.

Return type
OptimizeGeometryParameters

products()

Returns
The product states.

Return type
list (size 2) of *IdealGasThermoChemistry*

reactant()

Returns
The reactant state.

Return type
IdealGasThermoChemistry

reactionPath()

Returns
The reaction path.

Return type
NudgedElasticBand

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

secondProductMolecule()

Returns
The second reaction product state.

Return type
IdealGasThermoChemistry

transitionState()

Returns
The transition state at the given temperature.

Return type
IdealGasThermoChemistry

uniqueString()
Return a unique string representing the state of the object.

update()
Perform the update

MoleculeConfiguration

class MoleculeConfiguration(*elements=None*, *cartesian_coordinates=None*, *xyz_format=None*, *ghost_atoms=None*, *padding_length=None*, *velocities=None*, *tag_data=None*, *fast_init=False*, *bonds=None*, *dielectric_regions=None*, *metallic_regions=None*, *partial_charges=None*, *improper_dihedral_indices=None*)

A molecule configuration consisting elements and coordinates.

Parameters

•**elements** (array of *PeriodicTableElement*) – The elements that should be in the configuration.

•**cartesian_coordinates** (*PhysicalQuantity* of type length) – The Cartesian coordinates of the atoms in the configuration.

•**xyz_format** (*str*) – A XYZ formatted string representing the configuration. The parameter is mutually exclusive to `elements` and `cartesian_coordinates`.

•**ghost_atoms** (*list of ints | None*) – Indices of atoms that should be treated as ghost. **Default:** No ghost atoms.

•**padding_length** (*PhysicalQuantity* of type length) – Length of additional vacuum put around the molecule configuration when inscribed in a calculation box. **Default:** 0.0*Angstrom

•**tag_data** (*dict*) – Mapping between tags and associated indices. **Default:** No tags

•**velocities** (*PhysicalQuantity* of type velocity | None) – The velocities to set on the configuration. Has the dimensionality nx3. **Default:** None

•**fast_init** (*bool*) – Skip checking types and do not copy the arguments (i.e. only store references to them). This is an advanced option that should only be used internally. **Default:** False

•**bonds** (*array with shape (n,2) or (n,5) | None*) – An array with two atom indices for each bond along with the vector which periodic images this bond connects to.

•**dielectric_regions** (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

•**metallic_regions** (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

•**partial_charges** (*PhysicalQuantity of type charge | None*) – A PhysicalQuantity array of the atomic partial charge for each atom.

•**improper_dihedral_indices** (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

addBonds(*bond_list*)

Add bonds.

Parameters
bond_list (*numpy.ndarray*) – The list of bonds to add.

addTags(*tags*, *indices=None*)

Add a set of tags to atoms matching a collection of indices.

Parameters

•**tags** (*list | str*) – The list of tags to add to matching atoms.

•**indices** (*list | int | None*) – The list of indices to match atoms against. **Default:** All indices.

atomicMasses()

Returns
The masses of the atoms in the configuration.

Return type
PhysicalQuantity of type mass

`atomicNumbers()`

Returns
The list of atomic numbers associated with the elements.

Return type
list of ints

`bonds()`

Get the list of bond connections that can be used for bonded potentials in ATK-ForceField.

The connectivity indices refer to the wrapped configuration.

Returns
An array with the the two atom indices for each bond along with the vector which periodic images this bond connects.

Return type
array

`calculator()`

Returns
The calculator attached to the configuration, i.e. the calculator that will be used for both simulation and analysis.

Return type
Calculator

`cartesianCoordinates()`

Returns
The Cartesian coordinates of the atoms as an nx3 array.

Return type
PhysicalQuantity of type length

`coarseGrainDescriptors()`

Returns
The list of either elements or coarse grain particles for each element. Isotopes are returned as just their base element, UnitedAtoms and Particle are given as their specific type.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

`copy()`

Returns
A copy of the current configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

`copyAndDeleteAtoms(`*indices*`)`

Create a new configuration by deleting some atoms from this configuration.

Parameters
indices (*list of int*) – The indices of the atoms to delete.

Returns
The configuration with some atoms deleted.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndMerge(*other*)

Create a new configuration by merging this configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The other configuration.

Returns
The merged configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndShiftAtoms(*displacement*, *indices=None*)

Create a new configuration with some atoms translated.

Parameters

- **displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.
- **indices** (*list of int*) – The indices to shift. **Default:** All.

Returns
The configuration with the translation applied.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

deleteAtoms(*indices*)

Delete atoms from the configuration by specifying their indices.

Parameters
indices (*array of int*) – The indices to delete.

deleteBonds(*bond_list=None*, *pair_selection=None*)

Delete bonds connected to atomic indices.

Parameters

- **bond_list** (*A two-dimensional sequence*) – The pairs of bonds indices.
- **pair_selection** (*list | None*) – Specifies two groups between which bonds are delete. Selectable groups are elements, index lists, tag names, or None (all atoms).

dielectricRegions()

Returns
The dielectric regions in the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

elements()

Returns
The elements in configuration.

Return type
list of *PeriodicTableElement*

externalPotential()

Returns
The external potential present on the configuration.

Return type
AtomicShift | *AtomicCompensationCharge* | None

findBonds(*fuzz_factor=1.1*, *pair_selection=None*)

Find bonds in the configuration according to the combined covalent radii of the element pairs, multiplied with a fuzz factor. Optionally, find bonds only between two specified sub-groups of atoms. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.
- **pair_selection** (list(2) of type *PeriodicTableElement*, list of int, or str) – Specifies two groups between which bonds are detected. Selectable groups are elements, index lists, tag names, or None (all atoms). By default bonds between all atoms in the configuration are taken into account.

fixedSpinDirections()

Returns
The fixed spin directions for the configuration.

Return type
FixedSpin | None

generateShifts()

The method is defined only to support the same interface for all derived classes.

No shifts by default.

Returns
The array of shifts.

Return type
PhysicalQuantity of type length

ghostAtoms()

Returns
The list of ghost atoms.

Return type
list of ints

hillFormula()

Returns
The Hill formula for the molecule.

Return type
str

improperDihedralIndices()

Returns
The list of atom indices for each improper dihedral or None if no improper dihedrals are defined. Improper dihedrals are mainly used in bonded force fields.

Return type
numpy array | None

indices()

Returns
The indices of the atoms.

Return type
list of int

indicesFromIsotopes(*isotopes*)

Parameters
isotopes (*list of type PeriodicTableElement or Isotope*) – The isotopes to select.

Returns
The indices of the selected isotopes.

Return type
list of type int

indicesFromTags(*tags=None*)

List the indices associated with a given collection of tags.

Parameters
tags (*list | str*) – A list of tags for which all matching indices should be extracted.

Returns
The list of indices corresponding to the specified tag name(s).

Return type
list of ints

magneticMoments()

Returns
The magnetic moments of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type Bohr magneton.

merge(*other*)

Merge configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – A different configuration.

metallicRegions()

Returns
The metallic regions for the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`

Returns
The configuration information.

Return type
dict

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AtomicConfiguration object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`.

`numberOfAtoms()`

Returns
The total number of atoms in the configuration.

Return type
int

`paddingLength()`

Returns
The padding length.

Return type
PhysicalQuantity of type length

`partialCharges(indices=None)`

Get the list of partial atomic charges that can be used for representing electrostatic interactions in ATK-ForceField.

Parameters
indices (*list | int | None*) – The indices for which to return the partial charges. **Default:** All indices.

Returns
A PhysicalQuantity array of the atomic partial charge for each atom.

Return type
PhysicalQuantity of type charge | None

`particleDescriptors()`

Returns
The list of particle descriptors for each atom. This should return the exact same list as was given in the constructor argument elements.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

`static periodicBoundaries()`

Returns
The periodic boundary conditions of the configuration.

Return type
list

removeTags(*tags=None*, *indices=None*, *purge=False*)

Remove a set of tags from atoms matching a collection of indices.

Parameters

•**tags** (*list | str*) – The list of tags to add to matching atoms. **Default:** All tags.

•**indices** (*list | int*) – The list of indices to match atoms against. **Default:** All indices.

•**purge** (*bool*) – When removing tags from the configuration, delete the tag completely when not associated with any atoms anymore. **Default:** False.

scalePartialCharges(*scale_factor*, *indices=None*)

Scale the partial charges with a given scale factor. These partial charges are used with Forcefield calculators.

Parameters

•**scale_factor** (*float*) – The factor for scaling charges.

•**indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

setBonds(*bond_list*, *skip_checks=False*)

Set the bonds on the configuration. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

•**bond_list** (*list(n, 2) | None.*) – A list which contains for each bond the indices of the two connected atoms.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False

setCalculator(*calculator*, *initial_state=None*, *initial_spin=None*)

Attach a *Calculator* to the configuration which will be used in calculations involving the configuration.

Parameters

•**calculator** (*Calculator* | None) – The calculator object that should be attached to the configuration.

•**initial_state** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* with a calculator | None) – The initial state to be used for this configuration. **Default:** No initial state.

•**initial_spin** (*InitialSpin* | None) – The initial InitialSpin object to be used for this configuration. **Default:** `InitialSpin()`, except when used with a calculator of type *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*. In this case `InitialSpin([0., 0., ...])` is used, i.e. the scaled spins for each atom are set to 0.

setCartesianCoordinates(*cartesian_coordinates*, *indices=None*, *skip_checks=False*)

Set the Cartesian coordinates of the atoms.

Parameters

•**cartesian_coordinates** (*PhysicalQuantity* of type length) – The new coordinates of the atoms in each image.

•**indices** (*list*) – The indices of the atoms to set the positions of. **Default:** All indices.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setDielectricRegions(*dielectric_regions*)

Set the dielectric regions for the configuration.

Parameters
dielectric_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

setExternalPotential(*external_potential*)

Set an external potential on the configuration that will be used in calculations involving the configuration.

Parameters
external_potential (*AtomicShift* | *AtomicCompensationCharge*) – The external potential to apply.

setImproperDihedralIndices(*improper_dihedral_indices*)

Set the list of atom indices for each improper dihedral in bonded force fields.

Parameters
improper_dihedral_indices (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

setMagneticField(*magnetic_field*)

Set local magnetic field. The spins will be forced to point in the directions given by the magnetic_field object. The magnetic field can be defined for each atom. This only has an effect for Noncollinear or Spinorbit calculations.

Parameters
magnetic_field (*FixedSpin*) – The magnetic field to be applied.

setMagneticMoments(*magnetic_moments=None*, *skip_checks=False*)

Function to set magnetic moments on the configuration.

Parameters

- **magnetic_moments** (*PhysicalQuantity* of type Bohr magneton.) – The magnetic_moments to set on the configuration. Has the dimensionality nx3. **Default:** None.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setMetallicRegions(*metallic_regions*)

Set the metallic regions for the configuration.

Parameters
metallic_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setPartialCharges(*charge_list*, *indices=None*, *skip_checks=False*, *update_calculator=True*)

Set the partial charges on the configuration. The partial charges are used primarily to model electrostatic interactions in the TremoloX-calculator.

Parameters

- **charge_list** (*PhysicalQuantity of type charge | None*) – A list of atomic partial charges which contains a charge for each atom.
- **indices** (*list | int | None*) – The indices for which to set the partial charges. **Default:** All indices.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.
- **update_calculator** (*bool*) – Whether to update and attached Forcefield calculator with the new charges. **Default:** True.

setVelocities(*velocities=None*, *skip_checks=False*)

Function to set velocities on the configuration.

Parameters

•**velocities** (*PhysicalQuantity* of type velocity | None) – The velocities to set on the configuration. Has the dimensionality nx3. **Default:** None.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

shiftAtoms(*displacement*, *indices=None*, *skip_checks=False*)

Translate some atoms.

Parameters

•**displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.

•**indices** (*list of int*) – The indices to shift. **Default:** All.

•**skip_checks** (*bool*) – True, if all consistency checks should be skipped.

shiftPartialCharges(*total_charge*, *indices=None*)

Shift the partial charges so that their sum is the given total charge value. These partial charges are used with Forcefield calculators.

Parameters

•**total_charge** (*PhysicalQuantity of type charge*) – The new total charge.

•**indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

symbols()

Returns
The element symbols of the configuration.

Return type
list of str

tags(*indices=None*)

List the tags associated with a given collection of indices. The list returned is the set union of tags associated with the given indices. If no collection of indices is provided, then all tags on the configuration are returned.

Parameters
indices (*list | int*) – The indices to check. **Default:** All indices.

Returns
The set union of tags present on the provided indices.

Return type
set

uniqueElements(*ordered=False*)

Parameters
ordered (*bool*) – If the elements should be returned in ascending order by atomic number.

Returns
The unique elements contained in the configuration.

Return type
list of *PeriodicTableElement*

uniqueString()

Return a unique string representing the state of the object.

update(*force_restart=False*)

A self-consistent solution is generated, using the currently set calculator.

Parameters
force_restart (*bool*) – Force the self-consistent calculation to restart. **Default:** False.

velocities()

Returns
The velocities of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type velocity

Usage Examples

Define the geometry of and construct a water molecule:

```
# Set up elements and positions
elm = [ Oxygen, Hydrogen, Hydrogen ]
pos = [[ 0.000, 0.000, 0.0],
       [ 0.757, 0.586, 0.0],
       [-0.757, 0.586, 0.0]]*Angstrom

# Add them to a configuration
h2o = MoleculeConfiguration(elm, pos)
```

molecule_configuration.py

Alternative way of specification:

```
# Set up elements and positions using xyz format
h2o = MoleculeConfiguration(xyz_format=
    """3

       O   0.000 0.000 0.0
       H   0.757 0.586 0.0
       H  -0.757 0.586 0.0""")
```

molecule_configuration_xyz.py

Notes

ATK recognizes four types of atomic geometries:

•Molecules (*MoleculeConfiguration*)

•Bulk (*BulkConfiguration*)

•One Probe (*SurfaceConfiguration*)

•Two Probe (*DeviceConfiguration*)

A molecule is considered to be a system isolated in space. The QuantumATK calculators will place the molecule within an unit cell, in order to obtain a real space grid for describing the effective potential of the system.

It is possible to specify vacuum basis sets (ghost atoms) by adding an additional atom and include the index of that atom in the `ghost_atom` list. In this case the valence basis set of the atom is included in the orbital list, but there will be no atomic potential at the site.

MolierePotential

class MolierePotential(*particleType1*, *particleType2*, *c1*, *c2*, *c3*, *c4*, *d1*, *d2*, *d3*, *d4*, *f*, *Zi*, *Zj*, *s*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **c1** (*float*) – Potential parameter.
- **c2** (*float*) – Potential parameter.
- **c3** (*float*) – Potential parameter.
- **c4** (*float*) – Potential parameter.
- **d1** (*float*) – Potential parameter.
- **d2** (*float*) – Potential parameter.
- **d3** (*float*) – Potential parameter.
- **d4** (*float*) – Potential parameter.
- **f** (*PhysicalQuantity of type length*) – Potential parameter.
- **Zi** (*PhysicalQuantity of type charge*) – Potential parameter.
- **Zj** (*PhysicalQuantity of type charge*) – Potential parameter.
- **s** (*PhysicalQuantity of type energy*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Moliere-type potential between silicon and argon. *TremoloXPotentialSet*.

```
potentialSet = TremoloXPotentialSet(name = 'Moliere_ArSi')
potentialSet.addParticleType(ParticleType.fromElement(Silicon))
potentialSet.addParticleType(ParticleType.fromElement(Argon))

_potential = MolierePotential(
        particleType1=ParticleIdentifier('Si', []),
        particleType2=ParticleIdentifier('Ar', []),
        f=0.09734*Angstrom,
        c1=0.35,
        c2=0.55,
        c3=0.1,
        c4=0.0,
        d1=0.3,
        d2=1.2,
        d3=6.0,
        d4=0.0,
        Zi=14.0*elementary_charge,
        Zj=18.0*elementary_charge,
        s=0.0*eV,
        r_i=5.0*Angstrom,
        r_cut=7.5*Angstrom,
)
potentialSet.addPotential(_potential)
calculator = TremoloXCalculator(parameters=potentialSet)
```

moliere.py

Notes

This potential implements the Moliere potential[1], which models the short-range screened Coulomb repulsion between two nuclei. It can be used in combination with a conventional potential, such as Tersoff, which is usually not hard enough at short distances during high energy impacts. It is also suitable for modeling a purely repulsive interaction, such as between nobel gas atoms or ions. The potential follows the form

$$V_{ij}^{Moliere} = s_{ij} + \frac{Z_i Z_j}{4\pi\epsilon_0 r_{ij}} \sum_{m=1}^{4} c_{ij}^{(m)} \exp\left(-\frac{d_{ij}^{(m)} r_{ij}}{f_{ij}}\right).$$

where Z_i and Z_j are the nuclear charges of the elements involved.

[1] Gert Moliere. Theorie der streuung schneller geladener teilchen i. einzelstreuung am abgeschirmten coulomb-feld. *Zeitschrift für Naturforschung A*, 2(3):133–145, 1947.

Typically used values for $c^{(m)}$ are (0.35, 0.55, 0.1, 0) and for $d^{(m)}$ (0.3, 1.2, 6.0, 0). The parameter f_{ij} is often taken as $0.83(9\pi^2/128)^{1/3}a_{Bohr}(Z_i^{0.5}+Z_j^{0.5})^{-2/3}$ (see e.g.[2]). In this case the nucleus charges $Z_{i,j}$ should be taken as unitless.

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

MomentTensorPotentialFittingParameters

class MomentTensorPotentialFittingParameters(*basis_size=None*, *inner_cutoff_radii=None*, *tapering_cutoff_radii=None*, *outer_cutoff_radii=None*, *mtp_filename=None*, *cutoff_function=None*, *load_energy=None*, *load_forces=None*, *load_stress=None*, *ridge_regression_regularization=None*, *ridge_regression_cross_validation=None*, *constant_terms=None*, *weights=None*, *forces_cap=None*, *non_linear_coefficients_parameters=None*, *data_tags=None*, *use_element_specific_coefficients=None*, *solver=None*, *use_gpu_acceleration=None*)

Class for storing parameters for fitting a moment tensor potential.

Parameters

- **basis_size** (*PredefinedBasisSmall | PredefinedBasisBig | int | str*) – The basis set size to use. If a filename is given the basis is read for this file. **Default:** `PredefinedBasisSmall`
- **inner_cutoff_radii** (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The inner cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `0.5 * Angstrom`
- **tapering_cutoff_radii** (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The tapering cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `0.7 * Angstrom`
- **outer_cutoff_radii** (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The outer cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `5.0 * Angstrom`
- **mtp_filename** (*str*) – The filename for the fitted MTP parameter set. An exception is raised if the filename already exists. **Default:** The parameters will be written to a file MTP_parameters.mtp, prefixed by the index of the fit in the list.
- **cutoff_function** (*CutoffFunctionChebyshevOriginal | CutoffFunctionChebyshevExpansion*) – The cutoff function to use. **Default:** `CutoffFunctionChebyshevOriginal`

[2] CameronăF. Abrams and DavidăB. Graves. Molecular dynamics simulations of si etching by energetic cf3+. *Journal of Applied Physics*, 86(11):5938–5948, December 1999. URL: https://doi.org/10.1063/1.371637, arXiv:https://pubs.aip.org/aip/jap/article-pdf/86/11/5938/18980143/5938_1_online.pdf, doi:10.1063/1.371637.

- **load_energy** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the energy during fitting. **Default:** LoadQuantityWhereAvailable
- **load_forces** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the forces during fitting. **Default:** LoadQuantityWhereAvailable
- **load_stress** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the stress during fitting. **Default:** LoadQuantityWhereAvailable
- **ridge_regression_regularization** (*float*) – The regularization strength for the ridge regression model. **Default:** 1.0e-3
- **ridge_regression_cross_validation** (*int*) – The number of folds for enabling cross-validation in the ridge regression model. **Default:** No cross-validation.
- **constant_terms** (dict of {Element: *PhysicalQuantity* of type energy}) – The energy of an isolated atom for each species. **Default:** Calculated with the fitting calculator. by *MomentTensorPotentialTraining*.
- **weights** (Sequence of *PhysicalQuantity* of type (1/energy, length/energy, length**3/energy)) – The weights used for weighting energy, forces and stress [E, F, S] of every training configuration. If the quantity is not used for fitting or absent from the data, the corresponding weight is ignored. **Default:** No weighting.
- **forces_cap** (*PhysicalQuantity* of type energy / length) – Configurations with max. force magnitudes on an atom larger than this value will be discarded from the training data. **Default:** 100.0 * eV / Angstrom
- **non_linear_coefficients_parameters** (*NonLinearCoefficientsParameters*:) – The non-linear coefficients and parameters related to their optimization. **Default:** NonLinearCoefficientsParameters()
- **data_tags** (*None | str | list*) – A single tag or a list of tags identifying the training sets to be used in the fitting. None results in all available training sets being included. **Default:** None
- **use_element_specific_coefficients** (*bool*) – True, if element-specific coefficients should be used, otherwise False. **Default:** True
- **solver** (*SVDSolver | ProjectionSolver*) – The linear solver algorithm to determine the linear coefficients. **Default:** ProjectionSolver
- **use_gpu_acceleration** (*bool*) – Switch GPU acceleration using CUDA on or off. This accelerates the singular value decomposition (SVD or Projection) solver when determining the linear coefficients. This requires running the training on a CUDA-enabled GPU, otherwise it has no effect. **Default:** False.

basisSize()

Returns
The basis set size to use or the filename to read the basis from.

Return type
PredefinedBasisSmall | PredefinedBasisBig | int | str

constantTerms()

Returns
The energy of an isolated atom for each species, or None if it will be calculated with the fitting calculator by *MomentTensorPotentialTraining*.

Return type
dict of {Element: *PhysicalQuantity* of type energy} | None

cutoffFunction()

Returns
The cutoff function to use.

Return type
CutoffFunctionChebyshev

`dataTags()`

Returns
The tags identifying which datasets are used in the fitting. None includes all available datasets

Return type
None | list

`forcesCap()`

Returns
The forces cap.

Return type
PHYSICALQUANTITY| of type energy / length.

`innerCutoffRadii()`

Returns
The inner cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

Return type
PhysicalQuantity of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}

`loadEnergy()`

Returns
Whether to use the energy during fitting.

Return type
LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable

`loadForces()`

Returns
Whether to use the forces during fitting.

Return type
LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable

`loadStress()`

Returns
Whether to use the stress during fitting.

Return type
LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable

`mtpFilename()`

Returns
The filename for the fitted MTP parameter set, or None if it will be set by *MomentTensorPotentialTraining*.

Return type
str | None

`nlinfo()`

Returns
The nlinfo.

Return type
dict

nonLinearCoefficientsParameters()

Returns
The non-linear coefficients and parameters related to their optimization.

Return type
NonLinearCoefficientsParameters | *NonLinearParticleSwarmOptimizationParameters*

outerCutoffRadii()

Returns
The outer cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

Return type
PhysicalQuantity of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}

ridgeRegressionCrossValidation()

Returns
The number of folds for enabling cross-validation in the ridge regression model, of None for no cross-validation.

Return type
int | None

ridgeRegressionRegularization()

Returns
The regularization strength for the ridge regression model.

Return type
float

setBasisSize(*basis_size*)

Method for setting the basis size.

Parameters
basis_size (*PredefinedBasisSmall | PredefinedBasisBig | int | str*) – The basis set size to use. If a filename is given the basis is read for this file.

setConstantTerms(*constant_terms*)

Method for setting the constant terms.

Parameters
constant_terms (dict of {`Element`: *PhysicalQuantity* of type energy}) – The energy of an isolated atom for each species.

setCutoffFunction(*cutoff_function*)

Method for setting the cutoff function.

Parameters
cutoff_function (*CutoffFunctionChebyshevOriginal | CutoffFunctionChebyshevExpansion*) – The cutoff function to use.

setDataTags(*data_tags*)

Method for setting data tags.

Parameters
data_tags (*None | str | list*) – A single tag or a list of tags identifying the training sets to be used in the fitting. None results in all available training sets being included.

setForcesCap(*forces_cap*)

Method for setting the forces cap.

Parameters
forces_cap (*PhysicalQuantity* of type energy / length) – Configurations with max. force magnitudes on an atom larger than this value will be discarded from the training data.

setInnerCutoffRadii(*inner_cutoff_radii*)

Method for setting the inner cutoff radii.

Parameters
inner_cutoff_radii (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The inner cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

setLoadEnergy(*load_energy*)

Method for setting load_energy.

Parameters
load_energy (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the energy during fitting.

setLoadForces(*load_forces*)

Method for setting load_forces.

Parameters
load_forces (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the forces during fitting.

setLoadStress(*load_stress*)

Method for setting load_stress.

Parameters
load_stress (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the stress during fitting.

setMtpFilename(*mtp_filename*)

Method for setting the MTP filename.

Parameters
mtp_filename (*str*) – The filename for the fitted MTP parameter set.

setNonLinearCoefficientsParameters(*non_linear_coefficients_parameters*)

Method for setting non_linear_coefficients_parameters.

Parameters
non_linear_coefficients_parameters (*NonLinearCoefficientsParameters* | *NonLinearParticleSwarmOptimizationParameters*) – The non-linear coefficients and parameters related to their optimization.

setOuterCutoffRadii(*outer_cutoff_radii*)

Method for setting the outer cutoff radii.

Parameters
outer_cutoff_radii (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The outer cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

setRidgeRegressionCrossValidation(*ridge_regression_cross_validation*)

Method for setting ridge_regression_cross_validation.

Parameters
ridge_regression_cross_validation (*int*) – The number of folds for enabling cross-validation in the ridge regression model.

setRidgeRegressionRegularization(*ridge_regression_regularization*)

Method for setting ridge_regression_regularization.

Parameters
ridge_regression_regularization (*float*) – The regularization strength for the ridge regression model.

setSolver(*solver*)

Returns
[description].

Return type
[type]

setTaperingCutoffRadii(*tapering_cutoff_radii*)

Method for setting the tapering cutoff radii.

Parameters
tapering_cutoff_radii (*PhysicalQuantity* of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The tapering cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

setUseElementSpecificCoefficients(*use_element_specific_coefficients*)

Set the flag to use element-specific coefficients or not.

Parameters
use_element_specific_coefficients (*bool*) – If True, element-specific coefficients will be used.

setUseGpuAcceleration(*use_gpu_acceleration*)

Switch GPU acceleration using CUDA on or off. This accelerates the singular value decomposition (SVD) solver when determining the linear coefficients. This requires running the training on a CUDA-enabled GPU, otherwise it has no effect.

Parameters
use_gpu_acceleration (*bool*) – The flag to switch GPU acceleration on or off.

setWeights(*weights*)

Method for setting the weights.

Parameters
weights (Sequence of *PhysicalQuantity* of type (1/energy, length/energy, length**3/energy)) – The weights used for weighting energy, forces and stress [E, F, S] of every training configuration. If the quantity is not used for fitting or absent from the data, the corresponding weight is ignored.

solver()

Returns
[description].

Return type
[type]

taperingCutoffRadii()

Returns
The tapering cutoff radius for each element pair. If a single value is given, this will be used for all pairs.

Return type
PhysicalQuantity of type length | dict of {tuple (size 2) of `Element`: *PhysicalQuantity* of type length}

uniqueString()

Return a unique string representing the state of the object.

useElementSpecificCoefficients()

Returns
True, element-specific coefficients are used, otherwise False.

Return type
bool

useGpuAcceleration()

Returns
The flag to enable GPU acceleration using CUDA . This accelerates the singular value decomposition (SVD) solver when determining the linear coefficients. This requires running the training on a CUDA-enabled GPU, otherwise it has no effect.

Return type
bool

weights()

Returns
The weights used for weighting energy, forces and stress [E, F, S] of every training configuration.

Return type
Sequence of size 3

Usage Examples

Setup of *MomentTensorPotentialFittingParameters*. Note that if optimization of the *NonLinearCoefficientsParameters* is required it has to be specified here.

```
# Define non-linear coefficients and enable optimization.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
   perform_optimization=True,
   initial_coefficients=Random,
)

# MTP filename.
mtp_filename = 'mtp_parameters.mtp'

# Define parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
   basis_size=2000,
   outer_cutoff_radii=4.0*Angstrom,
   mtp_filename=mtp_filename,
   non_linear_coefficients_parameters=non_linear_coefficients_parameters,
   use_element_specific_coefficients=True,
)
```

Global optimization of the non-linear coefficients via particle-swarm-optimization (PSO) can be set up using the *NonLinearParticleSwarmOptimizationParameters*:

```
# Define non-linear particle-swarm-optimization.
non_linear_coefficients_parameters = NonLinearParticleSwarmOptimizationParameters(
     population_size=36,
     number_of_generations_per_iteration=20,
)

# Define parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
   basis_size=2000,
   outer_cutoff_radii=4.0*Angstrom,
   mtp_filename='mtp_parameters.mtp',
   non_linear_coefficients_parameters=non_linear_coefficients_parameters,
   use_element_specific_coefficients=True,
)
```

Use element-wise inner, tapering, and outer cutoff radii:

```
inner_cutoff_radii = {
    (Oxygen, Oxygen): 1.0*Ang,
    (Oxygen, Hydrogen): 0.5*Ang,
    (Hydrogen, Hydrogen): 0.5*Ang,
}
tapering_cutoff_radii = {
    (Oxygen, Oxygen): 1.1*Ang,
    (Oxygen, Hydrogen): 0.6*Ang,
    (Hydrogen, Hydrogen): 0.6*Ang,
}
outer_cutoff_radii = {
    (Oxygen, Oxygen): 4.0*Ang,
    (Oxygen, Hydrogen): 4.0*Ang,
    (Hydrogen, Hydrogen): 4.0*Ang,
}

# MTP filename.
mtp_filename = 'mtp_parameters.mtp'

# Define parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
   basis_size=PredefinedBasisSmall,
   inner_cutoff_radii=inner_cutoff_radii,
   tapering_cutoff_radii=tapering_cutoff_radii,
   outer_cutoff_radii=outer_cutoff_radii,
   mtp_filename=mtp_filename,
   use_element_specific_coefficients=True,
)
```

Notes

MomentTensorPotentialFittingParameters serves as an input argument for *MomentTensorPotentialTraining*.

This class specifies the hyper-parameters for the MTP training.

The most important parameters to specify and vary would be the MTP basis, the outer cutoff radii, as well as the *NonLinearCoefficientsParameters* (local optimization) or *NonLinearParticleSwarmOptimizationParameters* (global optimization), which specify the optimization of the element-pair dependent coefficients in the radial functions.

The effect of the basis size is generally that if the number of MTP basis functions is increased the training error is reduced, which improves the accuracy. However, a trade-off is that the extrapolation of the MTP, i.e. the prediction capability for configurations that are further outside the range of the training data, becomes worse. In practice this often means that some simulations, especially of amorphous materials or at high temperatures can become more unstable. If the number of basis is reduced the effect is the opposite: The simulation are typically more stable but the accuracy decreases.

It is recommended to use either the `PredefineBasisSmall`, or basis sizes between 400 and 800 functions for general purpose applications. If the MTP requires very high accuracy (e.g. for accurate phonon calculations of crystalline materials) one can increase the number of basis functions even more to e.g. around 2000 to 5000. In this case one has to be more careful with MD simulations at high temperature.

If the training data contains multiple elements, better accuracy can be achieved by using element-specific coefficients in the MTP model. This can be enabled by setting `use_element_specific_coefficients=True`. Using the option will result in slightly longer training and simulation times, but the accuracy gain can be significant, in particular for training data with many different elements, layered materials, or systems, where it is difficult to achieve a low enough training error with the normal MTP version.

The outer cutoff radii of the atomic environment can be chosen as global or as element-pair specific, and they should be chosen to at least include the first shell of neighbors for each element pair. The fit can often be improved by including more neighbor shells in the environments, up to the point where at larger radii the atomic environment becomes too complex to be described accurately by the finite number of MTP basis functions. Therefore, it is recommended to run several fits with different cutoff radii, and use the one that results in both a low error and stable simulations. Typical outer cutoff radii are between 3 and 6 Ang.

The inner cutoff radii determine the inner limit of the atomic neighborhood. Note, that the default value for the inner cutoff radii is set to 0.5 Ang, which works well for most elements, in that it allows to focus the radial functions on the important part of the interaction range.

The tapering cutoff radii determine the range below which the radial functions are smoothly tapered to zero and they should set slightly higher than the inner cutoff radii.

To optimize the hyper-parameters one can set up a list of *MomentTensorPotentialFittingParameters* with different parameters and choose the parameter set which results in the lowest training and test error, as reported by the `nlprint()` output of the *MomentTensorPotentialTraining* object.

Two different kinds of radial function can be chosen: The original polynomial functions used in[1], which are in this implementation orthonormalized via Chebyshev poynomials (default and recommended option), and the Chebyshev expansion as explained in[2]. These are specified by setting the parameter `cutoff_function` to `CutoffFunctionChebyshevOriginal` respectively `CutoffFunctionChebyshevExpansion`.

Optionally, one may specify the isolated atomic energies which are involved in calculating the constant terms in the MTP fitting. If not specified, these energies will be calculated automatically using the same reference calculator as the

[1] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

[2] Konstantin Gubaev, EvgenyăV. Podryabinkin, GusăL.W. Hart, and AlexanderăV. Shapeev. Accelerating high-throughput searches for new alloys with active learning of interatomic potentials. *Comput. Mater. Sci.*, 156:148–156, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0927025618306372, doi:https://doi.org/10.1016/j.commatsci.2018.09.031.

rest of the training set. If the reference calculator is a DFT calculator it will always use the spin-polarized version of the calculator, keeping all other parameters the same.

The fitting of the linear coefficients can be accelerated by running on a GPU. This is particularly beneficial when the non-linear optimization is switched off via setting `perform_optimization=False` in the *NonLinearCoefficientsParameters*. To enable GPU acceleration, the flag `use_gpu_acceleration` needs to be set to `True`. To make use of this feature, the calculation needs to be run on a GPU that supports CUDA-11.8, otherwise it will fall back to the default CPU mode.

MomentTensorPotentialTraining

class MomentTensorPotentialTraining(*filename*, *object_id*, *training_sets*, *calculator=None*, *correction_calculator=None*, *calculate_stress=None*, *train_on_energy_only=None*, *ignore_non_converged_configurations=None*, *fitting_parameters_list=None*, *train_test_split=None*, *random_seed=None*, *scaled_spins=None*, *number_of_processes_per_task=None*, *log_filename_prefix=None*, *configurations_per_task=None*, *committee_size=None*, *committee_mtp_filename_prefix=None*)

Study for training and testing a moment tensor potential.

Parameters

- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **training_sets** (*RandomDisplacementsParameters* | *CrystalInterfaceTrainingParameters* | *AlloyTrainingParameters* | MolecularDynamicsSnapshotsParameters | *TrainingSet* | sequence of [*RandomDisplacementsParameters* | *CrystalInterfaceTrainingParameters* | *AlloyTrainingParameters* | MolecularDynamicsSnapshotsParameters] | *TrainingSet* | *Table*) – The list of training sets to generate as part of the complete dataset.
- **calculator** (*Calculator*) – The calculator to fit the moment tensor potential to. **Default:** `LCAOCalculator()`.
- **correction_calculator** (*Calculator*) – The calculator to correct energy force and stress data with. The MTP is fit to the reference calculator minus the contribution of the correction calculator. **Default:** `None`.
- **calculate_stress** (*bool*) – Whether the stress will be calculated and added to the output. Only applies to bulk configurations. **Default:** `True`.
- **train_on_energy_only** (*bool*) – Flag to state whether the MTP should be trained solely on energy or not. **Default:** `False`.
- **ignore_non_converged_configurations** (*bool*) – Flag to state whether or not non-converged configurations are ignored in fitting. **Default:** `True`.
- **fitting_parameters_list** (*MomentTensorPotentialFittingParameters* | sequence of *MomentTensorPotentialFittingParameters* | *Table*) – The list of parameter sets for fitting the moment tensor potential. A separate moment tensor potential will be fitted for each parameter set in the list. **Default:** No fitting is performed.
- **train_test_split** (*float*) – The proportion of the complete dataset to use for training; the rest will be used for testing. Must be in the range (0, 1]. **Default:** `0.8`.
- **random_seed** (*int*) – The random seed used for generating the train/test split. **Default:** Generated automatically.

- **scaled_spins** (A list of tuples each of a *PeriodicTableElement* object and a scaled spin value. For non-collinear spin systems the tuple has four numbers, atom index, scaled spin, theta, phi, where the latter two are spherical coordinates as *PhysicalQuantity* object of type Degree or Radians) – The initial scaled spins for each type of element. **Default:** None.
- **number_of_processes_per_task** (int | None | ProcessesPerNode) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively. **Default:** None.
- **log_filename_prefix** (str | LogToStdOut) – General filename prefix for the logging output of the study tasks. If LogToStdOut, all logging will instead be sent to standard output. **Default:** 'mtp_training_'.
- **configurations_per_task** (*int*) – The number of configurations packed into each bundle task. For most applications, the default value is recommended to ensure a good balance between the performances of parallelization, restarting, and disc reading during task execution. This parameter is ignored for training sets of type CrystalInterface. **Default:** Determined automatically with a linear bundle scaling based on the average number of atoms per configuration.
- **committee_size** (*int | None*) – The number of committee models to train and store for query-by-committee MTP MD uncertainty predictions. If None, no committee model will be trained and added to the main MTP model. **Default:** None
- **committee_mtp_filename_prefix** (*str | None*) – Sets the prefix for the name of the mtp file containing the combined MTP parameters for the trained base MTP and the committee MTPs. If no prefix is set, the combined parameters are saved to the mtp_filename set in MomentTensorPotentialFittingParameters. **Default:** None

availableDataTags()

Returns
The available tags that have been set to select different datasets.

Return type
list

bestMTPParameters()

Ranks the fits and returns the best one.

Returns
The best fit

Return type
MTPParameters or None

calculateStress()

Returns
Whether the stress will be calculated and added to the output. Only applies to bulk configurations.

Return type
bool

calculator()

Returns
The calculator to fit the moment tensor potential to.

Return type
Calculator

committeeMTPFilenamePrefix()

Returns
The prefix set on the mtp filename that the base mtp and committee mtp models are saved to.

Return type
str | None

committeeSize()

Returns
The number of committee models specified by the user, or None if assigned automatically.

Return type
int | None

configurationsPerTask()

Returns
The number of configurations per task specified by the user, or None if assigned automatically.

Return type
int | None

continueTrainingSets(*log_filename_prefix=None*, *recalculate_training_data=None*)

Return a list with 2 items: The first is a training set that includes all the converged configurations, the second is a training set with all the non-converged configurations.

Parameters

- **log_filename_prefix** (*str*) – Prefix for the log file for completed and ignored data.
- **recalculate_training_data** (*bool*) – Whether or not the energy, forces and stresses are recalculated for the converged configurations. Note: The *TrainingSet* containing the non-converged configurations is always recalculated. **Default:** False

Returns
List of training sets containing available converged and non-converged configurations.

Return type
list

correctionCalculator()

Returns
The calculator to correct energy force and stress data with.

Return type
Calculator

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

filename()

Returns
The filename where the study object is stored.

Return type
str

fittingParametersList()

Returns
The list of parameter sets for fitting the moment tensor potential, or None if no fitting is performed.

Return type
list of *MomentTensorPotentialFittingParameters* | None

fittingReport(*stream*, *data_tags=None*, *sort_best_fit=False*)

Print a string containing an ASCII table summarizing the available results for the study.

Parameters

- **stream** (*file-like*) – The stream to write to. This should be an object that supports strings being written to it using a `write` method.
- **data_tags** (*str | list | None*) – List of tags indicating the datasets to calculate the statistics for.
- **sort_best_fit** (*bool*) – Print the fits sorted by quality. **Default:** `False`

ignoreNonConvergedConfigurations()

Returns
Whether or not configurations are ignored if the calculation of energy does not converge. Only applies to calculators that use an SCF method.

Return type
bool

ignoredConfigurations()

Return the configurations that are ignored because the calculation of energy, force and stress did not converge.

Returns
The training configurations that did not converge.

Return type
ConfigurationDataContainer | None

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

momentTensorPotentialCalculators(*fit_index=None*, *include_correction=False*)

The output moment tensor potential calculators, generated with the given fitting parameter sets. This result will only be available after the study has been updated.

Parameters

- **fit_index** (*int*) – The index of the fitting parameter set.
- **include_correction** (*bool*) – Whether or not the returned MTP includes the correction calculator. **Default:** False.

Returns
The fitted moment tensor potential calculators. If not available, returns None.

Return type
TremoloXCalculator | list of *TremoloXCalculator* | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

nonLinearCoefficients()

The output non-linear coefficients for the moment tensor potential fitting, generated with the given fitting parameter sets. If the optimization of the coefficients is switched off, they will be equivalent to the input coefficients. This result will only be available after the study has been updated.

Returns
The optimized non-linear coefficients. If not available, returns None.

Return type
dict | None

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

randomSeed()

Returns
The random seed used for generating the train/test split, or None if there is no split.

Return type
int | None

rankFits(*data_tags=None*, *weights=None*, *statistical_measure=None*)

Ranks the fits based on some (to be specified) quality measure.

Parameters

- **data_tags** (*str | list | None*) – List of tags indicating the datasets to calculate the statistics for.
- **weights** (*list(2) of lists(3) of float*) – Weights to be used for the energy, forces, stress and training, testing errors. The weights are always normalized. [training[energy, forces, stress], testing[energy, forces, stress]]. **Default:** `[[1, 1, 1], [1, 1, 1]]`.

- **statistical_measure** (*RMSE | MAE | R2Score*) – The statistical measure used as quality metric. RMSE, MAE, R2Score. **Default:** R2Score.

Returns
Two lists: 1. An ordered list of fit indices ranked by the quality measure (best first). 2. A list of the quality measures in the original fit order.

Return type
list(2) of lists

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

scaledSpins()

Returns
The scaled spins of the atoms.

Return type
list.

statisticsData(*fit_indices=None*, *data_tags=None*, *plot_cohesive_energy=False*)

Gets the training statistics data from the placeholder task, or generates it if it is not available.

Returns
The training statistics data.

trainOnEnergyOnly()

Returns
Whether the training should be based solely on energy (True) or on energy, forces and stress (False).

Return type
bool

trainTestSplit()

Returns
The proportion of the complete dataset to use for training; the rest will be used for testing.

Return type
float

trainingSets()

Returns
The list of training sets to use or generate based on the reference configurations.

Return type
list of [*RandomDisplacementsParameters* | *CrystalInterfaceTrainingParameters* | *AlloyTrainingParameters* | *MolecularConfigurationsParameters* *MolecularDynamicsSnapshotsParameters* | *TrainingSet*]

trainingTestingDatasets(*train_test_split=None*, *data_tags=None*, *efs_only=False*)

The complete output training and testing datasets with the calculated E/F/S data. This result will only be available after the study has been updated.

Parameters

- **train_test_split** (*float*) – The proportion of the complete dataset to use for training; the rest will be used for testing. Must be in the range (0, 1]. **Default:** The value given, when constructing the object.
- **data_tags** (*str | list | None*) – One or more tags used to identify which configurations is returned. **Default:** None, which returns all available configurations.
- **efs_only** (*bool*) – Flag to disable reading of validation and correction data. This significantly speeds up construction of the trajectory.

Returns
The complete training and testing datasets. If not available, returns None.

Return type
tuple (size 2) of *Trajectory* | None

uniqueString()
Return a unique string representing the state of the object.

update()
Run the calculations for the study object and ensure that the MTP parameters are written to the given filenames.

writeMTPFiles(*fit_index=None*, *mtp_filename=None*)
Write the MTP parameters to file, Works only if the mtp file content is available from the fitting task.

Parameters

- **fit_index** (*int | None*) – Write only the content for this fit index. If None, write the files for all fits.
- **mtp_filename** (*str | None*) – Write the content to this filename. If None, the filename specified in the corresponding fitting parameters is used.

Usage Examples

Setup of *MomentTensorPotentialTraining* of quartz.

```
# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697        ,  0.            ,  0.            ],
                          [ 0.            ,  0.4697        ,  0.666666666667],
                          [ 0.5303        ,  0.5303        ,  0.333333333333],
                          [ 0.4135        ,  0.2669        ,  0.1191        ],
                          [ 0.2669        ,  0.4135        ,  0.547567      ],
                          [ 0.7331        ,  0.1466        ,  0.785767      ],
                          [ 0.5865        ,  0.8534        ,  0.214233      ],
                          [ 0.8534        ,  0.5865        ,  0.452433      ],
                          [ 0.1466        ,  0.7331        ,  0.8809        ]]

# Set up configuration
reference_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
)

# Define calculator for E/F/S data calculations.
calculator = LCAOCalculator()

# Set up non-linear coefficients with optimization.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
)

# Set up parameters to use in the MTP fitting.
fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=1000,
    outer_cutoff_radii=3.0*Angstrom,
    mtp_filename='mtp_study.mtp',
    non_linear_coefficients_parameters=non_linear_coefficients_parameters,
)

# In this specific example, the default displacement protocol for crystals is used
training_sets = crystalTrainingRandomDisplacements(
    reference_configuration,
    supercell_repetitions_list=[(2, 2, 2), (3, 3, 3)],
    sample_size_per_stage=10,
)

# Set up MTP training.
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_study',
    object_id='training',
    training_sets=training_sets,
    calculator=calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters,
)
moment_tensor_potential_training.update()
nlprint(moment_tensor_potential_training)

# The MTP calculator can now be extracted from the MomentTensorPotentialTraining object.
mtp_calculator = moment_tensor_potential_training.momentTensorPotentialCalculators()[0]
```

MomentTensorPotentialTraining_example1.py

Several MomentTensorPotentialTraining objects with pre-calculated training data can be loaded and then passed into another MomentTensorPotentialTraining object which does the fitting:

```
training_sets = []

# Load another MomentTensorPotentialTraining object with precalculated data.
mtp_training_data_input = nlread('mtp_training_data.hdf5',␣
↪MomentTensorPotentialTraining)[0]
training_sets.append(
    TrainingSet(mtp_training_data_input, recalculate_training_data=False)
```

(continues on next page)

(continued from previous page)

```
)

# Load another Trajectory object with precalculated data.
trajectory_training_data_input = nlread('trajectory_training_data.hdf5', Trajectory)[0]
training_sets.append(
    TrainingSet(trajectory_training_data_input, recalculate_training_data=False)
)

# Set up MTP training.
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_training.hdf5',
    object_id='training',
    training_sets=training_sets,
    calculator=LCAOCalculator(),
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters,
)
moment_tensor_potential_training.update()
nlprint(moment_tensor_potential_training)
```

MomentTensorPotentialTraining_example2.py

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

This class implements the moment tensor potential (MTP) framework[1]. This class combines 3 different stages of the MTP training workflow:

- The generation and calculation of training data
- The actual training of the MTP parameters
- The validation of the trained MTP

To generate and calculate training data different possibilities are provided:

- *RandomDisplacementsParameters* or *crystalTrainingRandomDisplacements*: A series of random atomic displacements and strain to sample to phase space around an equilibrium configuration.
- *MolecularDynamicsSnapshotsParameters*: Snapshots from molecular dynamics simulations using either the final reference calculator, or a force field or fast high-throughput DFT calculator. In the latter case energies, forces, and stress are recalculated using the reference calculator for a subset of the snapshots from the MD trajectory.
- *TrainingSet*: A set of available training configurations with or without pre-calculated reference energy, forces, and stress. If no reference data is provided, it will automatically be recalculated using the given reference calculator.
- *CrystalInterfaceTrainingParameters*: Used to generate training configurations by building interfaces from two crystalline bulk materials.
- *MolecularConfigurationsParameters*: Class for generating a set of training configurations using a combination of sampling different torsion angles and atomic displacements.

[1] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

•*AlloyTrainingParameters*: Class for generating a set of alloy training configurations.

The calculation of reference data in the *MomentTensorPotentialTraining* can be parallelized efficiently over different process groups via the keyword `number_of_processes_per_task`.

A *MomentTensorPotentialTraining* with calculated reference data can also be used as input to other *MomentTensorPotentialTraining* objects by passing it via a *TrainingSet* object. This can be used to separate the calculation of training data and fitting steps. It also allows easily combining many different pre-calculated training data sets into a single fit. This can become more practical for complex materials where the training data contains many different separate components.

The second (optional) stage is the training of the MTP parameters. Here, the *MomentTensorPotentialFittingParameters* object can be passed to specify the MTP hyper-parameters that should be used for the fit. To optimize these parameters a list of these objects with different parameters can be given. A fit will be carried out for each of these to identify the best combination of parameters from the training and test errors reported via `nlprint()`.

The `nlprint()` report prints a summary of all training energy, forces, and stress root-mean-square errors with respect to the reference data for all fitting parameters. An independent test error calculated for configurations which have been taken out of the training data is also reported. The ratio of configurations used for training respectively testing is specified by the `train_test_split` parameters. The two sets are split randomly from the entire training data.

The training and test errors both give indications of the quality of the potential and possible ways to improve the fit. If the training and test errors are both similarly high, this is an indication that the model is not complex enough to properly represent the training set. In this case the potential may be improved by increasing the basis set size. If the test error is significantly larger than the training error, this is an indication that the training set is too small, and that the potential is being over-fitted. In this case the potential may be improved by generating more training data, either by specifying new training sets or by performing active learning.

The list with the final calculators with the trained MTP parameters, one calculator for each fit, can be queried via `momentTensorPotentialCalculators()`. Alternatively, the MTP calculator may be imported as an `.mtp` file using the *MTPPotential* class.

For the parallelization of the *MomentTensorPotentialTraining* it is recommended to use many MPI processes, as both the DFT calculations and the MTP fitting benefit more from MPI parallelization than from threading.

MonkhorstPackGrid

class MonkhorstPackGrid(*na=None*, *nb=None*, *nc=None*, *symmetries=None*, *force_timereversal=None*, *k_point_shift=None*, *shift_to_gamma=None*)

Parameters

•**na** (*Positive int*) – Number of k-points in A direction. **Default:** 1

•**nb** (*Positive int*) – Number of k-points in B direction. **Default:** 1

•**nc** (*Positive int*) – Number of k-points in C direction. **Default:** 1

•**symmetries** (*list(n_symmetries) of (array(3, 3), array(3)).*) – A list of user-defined symmetries used for the reduction of the k-points in integration. **Default:** Empty list, which means ATK will determine the relevant symmetries.

•**force_timereversal** (*bool*) – Always enforce time reversal symmetry. **Default:** True (False with spin-orbit or noncollinear spin)

•**k_point_shift** (*list(3) of float*) – Optional k-point shift in fractional coordinates. This option is mutually exclusive to `shift_to_gamma`. **Default:** Determined internally depending on the value of `shift_to_gamma`.

- **shift_to_gamma** (*list(3) of bool | bool*) – Shift grid so that Γ (0.0, 0.0, 0.0) is included along all or certain directions. This option is mutually exclusive to `k_point_shift`. **Default:** True (Not active if `k_point_shift` is specified)

`allKpoints()`

Returns
All the k-points of the *MonkhorstPackGrid* before symmetry reduction.

Return type
ndarray(N,3) of float where N is the number of k-points.

`allKpointsWeights()`

Returns
The weight of all k-points before symmetry reduction.

Return type
ndarray(n) for float where n is the number of k-points before symmetry reduction.

`disableTimereversal()`

After construction of the *MonkhorstPackGrid*, the time reversal symmetry can be disabled using this function.

`foldoutMapVector()`

Query method for getting the map between the symmetry reduced k-points and unreduced k-points.

Returns
A list with indices matching the k-points in the symmetry reduced set and in the unreduced set.

Return type
ndarray(N) with integer values between 0 and M-1. N is the number of k-points in the unreduced set, M is the number of k-points in the reduced set.

`forceTimereversal()`

Returns
Whatever force_timereversal is enabled or disabled.

Return type
bool

`kPointShift()`

Returns
The k_point_shift which the k-points are shifted with.

Return type
list(3) of float

`kpoints()`

Returns
The symmetry reduced kpoints.

Return type
ndarray(N,3) of float where N is the number of symmetry reduced k-points.

`kpointsWeights()`

Returns
The weights of the symmetry reduced k-points.

Return type
ndarray(n) for floats, where n is the number of unreduced number of k-points.

na()

Returns
The number of k-points along the kA-direction.

Return type
int

nb()

Returns
The number of k-points along the kB-direction.

Return type
int

nc()

Returns
The number of k-points along the kC-direction.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

shiftToGamma()

Returns
Whatever the k-points are shifted to the Γ.

Return type
bool

symmetries()

Returns
The symmetries to apply for reducing the k-points.

Return type
list(n_symmetries) of (array(3, 3), array(3)).

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a 2x5x3 Monkhorst-Pack k-point grid. Calculate the irreducible k-points using the time reversal symmetry and mirror symmetry in the A direction. Print out the relation between the full k-point set and the irreducible set:

```
# Make a  MonkhorstPackGrid with mirror symmetry in A direction and time reversal␣
↪symmetry
symmetries = [( numpy.array([[-1,0,0], [0,1,0], [0,0,1]]), numpy.zeros(3))]
mpg = MonkhorstPackGrid(2, 5, 3, symmetries=symmetries)

# Get the kpoints
irreducible_kpoints = mpg.kpoints()
irreducible_weights = mpg.kpointsWeights()

all_kpoints = mpg.allKpoints()

# Print out the irreducible kpoints
print(len(irreducible_kpoints), ' irreducible kpoints')
for kp, w in zip(irreducible_kpoints, irreducible_weights):
    print(kp, w*len(all_kpoints))

# Get the map from the full kpoints to the irreducible part
fold_out_map = mpg.foldoutMapVector()

# Print out the relation between the full and irreducible set
print(len(all_kpoints), '  kpoints')
for i, kp in enumerate(all_kpoints):
    print(kp, ' -> ', irreducible_kpoints[fold_out_map[i]])
```

monkhorstpackgrid.py

Notes

Make a uniform k-point grid according to procedure of Monkhorst and Pack[1],

$$\mathbf{k}_{prs} = u_p\,\mathbf{b}_1 + u_r\,\mathbf{b}_2 + u_s\,\mathbf{b}_3,$$

$$u_r = (2r - q - 1)/2q \qquad (r = 1, 2, 3, ..., q),$$

where $\mathbf{k}_{prs}$ are the k-vectors corresponding to the special points of the k-grid, $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$ are the reciprocal lattice vectors, u_r is the sequence of numbers, and q^3 is the total number of the k-points used to sample the Brillouin zone.

- The original Monkhorst-Pack procedure assumes that odd k-point grids include the Γ point while even k-point grids avoid the Γ point.
- It is possible and often advantageous to shift the grid by Δk. For example, one may need to include the Γ point for cubic and hexagonal lattices.
- When the time-reversal symmetry of the Schrödinger equation is preserved, it is possible to take advantage of the inversion symmetry to reduce the number of k-points used for sampling the Brillouin zone. This is done by default in QuantumATK.

[1] H.ăJ. Monkhorst and J.ăD. Pack. Special points for Brillouin-zone integrations. *Phys. Rev. B*, 13:5188–5192, Jun 1976. doi:10.1103/PhysRevB.13.5188.

•If the time-reversal symmetry of the Schrödinger equation is broken, e.g., by spin-orbit interaction, the inversion symmetry cannot be adopted to reduce the number of k-points. The time-reversal symmetry can be turned off, by using the `force_timereversal` keyword.

MorseBondPotential

class MorseBondPotential(*particleType1*, *particleType2*, *k*, *E*, *r_0=None*)

Constructor of the potential.

Parameters

•**particleType1** (*ParticleType or ParticleIdentifier*) – Identifier of the first particle type.

•**particleType2** (*ParticleType or ParticleIdentifier*) – Identifier of the second particle type.

•**k** (*PhysicalQuantity of length**-1*) – Potential parameter.

•**E** (*PhysicalQuantity of type energy*) – Potential parameter.

•**r_0** (*PhysicalQuantity of type lenth*) – Potential parameter (equilibrium distance).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a bonded potential with a MorseBondPotential for an ethane molecule.

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Ethane_bonded')

# Add particle types for Carbon and Hydrogen.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=-0.3*elementary_charge)
)
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.1*elementary_charge)
)
```

(continues on next page)

(continued from previous page)

```
# Set up a new bond potential for C-C bonds and add it to the potential set.
bond_potential = MorseBondPotential(
    particleType1='C',
    particleType2='C',
    k=3.8*Ang**-1,
    E=1.0*eV,
    r_0=1.526*Ang,
)
potential_set.addPotential(bond_potential)

# Set up a new bond potential for C-H bonds and add it to the potential set.
bond_potential = MorseBondPotential(
    particleType1='C',
    particleType2='H',
    k=3.8*Ang**-1,
    E=1.0*eV,
    r_0=1.090*Ang,
)
potential_set.addPotential(bond_potential)

# Create a new TremoloXCalculator with this potential.
calculator = TremoloXCalculator(parameters=potential_set)
```

Here, only the MorseBondPotential block of the script is shown. The full script can be found found in the file `ethane_morse_bond_potential.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- This bond potential is calculated as

$$V(r) = E_0 \left[(1 - \exp(-k(r - r_0)))^2 - 1 \right]$$

where r is the distance between the atoms connected by this bond and r_0 is the equilibrium distance.

MorsePotential

class MorsePotential(*particleType1*, *particleType2*, *r_0*, *k*, *E_0*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**r_0** (*PhysicalQuantity of type length*) – Potential parameter.

•**k** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**E_0** (*PhysicalQuantity of type energy*) – Potential parameter.

•**r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.

•**r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O',
                                         r_0=2.1*Angstrom,
                                         k=2.0067*1/Ang,
                                         E_0=0.340554*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=1.0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O',
                                         r_0=3.618701*Angstrom,
                                         k=1.379316*1/Ang,
                                         E_0=0.042395*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=22.0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

(continues on next page)

(continued from previous page)

```
bulk_configuration.update()
```

Notes

The *MorsePotential* defines a potential of the form

$$V_{ij}(r) = E_{ij}\left[\left(1 - e^{-k(r-r_0)}\right)^2 - 1\right].$$

It is often used to describe interatomic bonds, as the resulting potential well is typically steeper than the one produced by a *LennardJonesPotential*.

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

MullikenPopulation

class MullikenPopulation(*configuration*)

Class for calculating the Mulliken population for a configuration.

Parameters
configuration (*DeviceConfiguration* | *BulkConfiguration* | *MoleculeConfiguration*.) – The Configuration which the Mulliken population should be calculated for.

atomicAngles()

Returns
The atomic polarization angles as a list of (θ, ϕ) tuples representing spherical coordinates.

Return type
list of tuple

atomicCharge()

Calculate the charge of each atom.

Returns
The charge of each atom in a length n array.

Return type
PhysicalQuantity of type charge

atoms(*spin=None*)

Return an array with the Mulliken population of each atom.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The mulliken population is calculated for this spin. **Default:** Spin.All

bond(*atom_i*, *atom_j*)

Return a float which is the Mulliken population projected onto the bond between atom_i and atom_j.

Parameters

- **atom_i** (*int*) – Index of the first atom in the bond projection. atom_i should satisfy 0 <= atom_i < number_of_atoms.
- **atom_j** (*int*) – Index of the second atom in the bond projection. atom_j should satisfy 0 <= atom_j < number_of_atoms.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

orbitals(*spin=None*)

Calculate the Mulliken population of all orbitals.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The mulliken population is calculated for this spin. **Default:** Spin.All

Returns
One array for each atom. Each array holds the Mulliken population of each orbital of that atom.

Return type
list of numpy.array

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the Mulliken population of an ammonia molecule and print projections of orbitals and bonds:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.,       0.,        0.124001],
                           [ 0.,       0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]] * Angstrom)

# Define the calculator
calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the mulliken population
mulliken_population = MullikenPopulation(
    configuration=molecule_configuration)

nlsave('mulliken.hdf5', mulliken_population)

# print all occupations
nlprint(mulliken_population)

# print partial occupations of N
print('N ', mulliken_population.atoms(spin=Spin.Sum)[0])
print(' | ', mulliken_population.orbitals(spin=Spin.Sum)[0])
print('')

# print partial occupations of first H
print('H ', mulliken_population.atoms(spin=Spin.Sum)[1])
print(' | ', mulliken_population.orbitals(spin=Spin.Sum)[1])
print('')

# print Mulliken population of N-H bond
print('N-H bond ', mulliken_population.bond(0, 1))
```

mulliken_nh3.py

Notes

The total number of electrons, N, is given by

$$N = \sum_{ij} D_{ij} S_{ij},$$

where D is the density matrix, S the overlap matrix, and the sum is over all orbitals in the system.

The Mulliken population is defined by different partitions of this sum into orbitals, atoms and bonds.

- **Mulliken population of orbitals**

The Mulliken Population of orbitals, M_i , is defined by restricting one of the sum indexes to the orbital, i.e.

$$M_i = \sum_{j} D_{ij} S_{ji}.$$

•**Mulliken population of atoms**
The Mulliken Population of atoms, M_μ , is defined by adding up all the orbital contributions on atom number μ

$$M_\mu = \sum_{i\in\mu}\sum_j D_{ij}S_{ji}.$$

•**Mulliken population of bonds**
The Mulliken Population of bonds, $M_{\mu\nu}$, is defined by restricting the sum indexes to the orbitals on atom number μ and ν , i.e.

$$M_{\mu\nu} = (2-\delta_{\mu\nu})\sum_{i\in\mu}\sum_{j\in\nu} D_{ij}S_{ji}.$$

Note the factor two, which ensures that

$$N = \sum_{\mu\geq\nu} M_{\mu\nu}.$$

Noncollinear spin

For noncollinear systems, the Mulliken population of atoms is a four component spin tensor:

$$M_\mu^{\sigma\sigma'} = \sum_{i\in\mu}\sum_j D_{ij}^{\sigma\sigma'}S_{ji}.$$

The Mulliken Population of atoms can be diagonalized to give a local spin direction. This direction is reported by the *nlprint* command for noncollinear systems.

The *nlprint* report for noncollinear spin also shows the orbital Mulliken populations in the local spin direction.

MultigridSolver

class MultigridSolver(*boundary_conditions=None*, *solvent_dielectric_constant=None*)

The multigrid-based Poisson solver.

Parameters

•**boundary_conditions** (list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*) – A list of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:**

–*MoleculeConfiguration*: `[[MultipoleBoundaryCondition()] * 2] * 3`

–*BulkConfiguration*: `[[PeriodicBoundaryCondition()] * 2] * 3`

–*SurfaceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition(), NeumannBoundaryCondition()]]
```

–*DeviceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition()] * 2]
```

- **solvent_dielectric_constant** (*float*) – The dielectric constant of the media surrounding the configuration. Must be a number larger than 0. **Default:** `1.0`

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

nlinfo()

Returns
The nlinfo.

Return type
dict

solventDielectricConstant()

Returns
The solvent dielectric constant.

Return type
float

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Define a *MultigridSolver* with specified boundary conditions on the 6 faces:

```
poisson_solver = MultigridSolver(
    boundary_conditions=[
        [DirichletBoundaryCondition(), NeumannBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Define a *MultigridSolver* with specified boundary conditions in the A, B, and C-direction, but similar boundary conditions on opposite faces.:

```
poisson_solver = MultigridSolver(
    boundary_conditions=[
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)
```

(continues on next page)

(continued from previous page)

```
calculator = LCAOCalculator(
    poisson_solver=poisson_solver,
)
```

Notes

•By setting the `solvent_dielectric_constant` it is possible to perform calculations of solvents. In this case the volume of the configuration is defined by inscribing each atom in a sphere of size given by the van der Waals radius of the element. Inside the volume of the configuration the dielectric constant is 1, outside the volume of the configuration the dielectric constant is equal to the value of `solvent_dielectric_constant`.

•See also *Poisson solvers*.

MultipoleBoundaryCondition

class MultipoleBoundaryCondition

A Multipole type boundary condition.

NPTBerendsen

class NPTBerendsen(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *thermostat_timescale=None*, *reservoir_pressure=None*, *barostat_timescale=None*, *compressibility=None*, *coupling_mask=None*, *heating_rate=None*, *compression_rate=None*)

NPT Berendsen integrator class which implements the Berendsen barostat.

Parameters

•**initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*

•**time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1.0 * fs`

•**reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. **Default:** `300.0 * Kelvin`

•**thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen temperature coupling. **Default:** `100.0 * fs`

•**reservoir_pressure** (*PhysicalQuantity* of type pressure) – The reservoir pressure in the simulation. The pressure can be given either as a scalar value, as a vector of length 3 denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. A scalar value will result in isotropic pressure coupling, whereas for all other representations, each pressure component will be coupled to the barostat independently. **Default:** `1.0*bar`

•**barostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen pressure coupling. **Default:** `1000.0 * fs`

•**compressibility** (*PhysicalQuantity* of type pressure**-1) – The estimated compressibility of the system relating volume changes to pressures changes. **Default:** `1.0e-4*bar**-1`

- **coupling_mask** (*array of bools*) – The mask determining which elements of the stress tensor, the barostat should couple to. Can be given as a vector of length 3, denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. By default the barostat couples only to the diagonal elements. **Default:** True for diagonal elements, False for shear components.
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `None`
- **compression_rate** (*PhysicalQuantity* of type pressure/time | None) – The compression rate of the target pressure. **Default:** `0.0 * bar/fs`

isotropicCoupling()

Returns
Whether the barostat applies isotropic or anisotropic pressure coupling.

Return type
bool

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the Berendsen barostat with isotropic pressure coupling:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )
```

(continues on next page)

(continued from previous page)

```
# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up a Berendsen barostat with isotropic pressure coupling.
method = NPTBerendsen(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    reservoir_pressure=1.0*bar,
    barostat_timescale=500.0*femtoSecond,
    compressibility=1.0e-4*bar**-1,
    heating_rate=0*Kelvin/picoSecond,
    compression_rate=0.0*bar/picoSecond,
    initial_velocity=None
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
    method=method
)
```

nptberendsen.py

Apply anisotropic pressure coupling only to the zz-direction of the pressure tensor.

```
method = NPTBerendsen(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    reservoir_pressure=[0.0, 0.0, 10.0]*GPa,
    barostat_timescale=500.0*femtoSecond,
    compressibility=1.0e-4*bar**-1,
    heating_rate=0*Kelvin/picoSecond,
    coupling_mask=[False, False, True, False, False, False],
    compression_rate=0.0*bar/picoSecond,
    initial_velocity=None
)
```

nptberendsen2.py

Notes

- The Berendsen barostat can be used to approximately simulate a canonical NPT-ensemble, as described in Ref.[1].
- If a scalar `reservoir_pressure` value is given, isotropic pressure coupling is applied, i.e. all cell vectors are rescaled by the same factor, and the isotropic pressure $P = 1/3(P_{xx} + P_{yy} + P_{zz})$ is used.

  If a pressure vector (of length 3 or 6, in Voigt notation), or a 3x3-tensor is given as `reservoir_pressure`, anisotropic pressure coupling is applied, meaning that each degree of freedom of the cell vectors will be rescaled independently.

  You can enable/disable the coupling to selected components of the pressure tensor via the parameter `coupling_matrix`. This parameter only has an effect with anisotropic pressure coupling.

  By default, the barostat couples only to the hydrostatic pressure, i.e. the diagonal elements of the pressure tensor.
- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

  If a non-zero `compression_rate` is specified, the reservoir pressure will be changed linearly during the simulation, according to the specified compression rate.
- The `compressibility` parameter is used to determine the relationship between volume and pressure changes, when the simulation cell is rescaled, which affects how well the observed barostat time scale agrees with the specified *barostat_timescale* parameter. However, if not set to extreme and unphysical values, the `compressibility` parameter should not influence the result of the simulation.

NPTBernettiBussi

class NPTBernettiBussi(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *thermostat_timescale=None*, *reservoir_pressure=None*, *barostat_timescale=None*, *compressibility=None*, *coupling_mask=None*, *heating_rate=None*, *compression_rate=None*, *random_seed=None*)

NPT integrator class which implements the stochastic velocity rescaling (Bussi-Donadio-Parrinello) thermostat in combination with the stochastic cell rescaling (Bernetti-Bussi) barostat. It correctly samples the isobaric-isothermic ensemble.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1.0 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. **Default:** `300.0 * Kelvin`
- **thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen temperature coupling. **Default:** `100.0 * fs`
- **reservoir_pressure** (*PhysicalQuantity* of type pressure) – The reservoir pressure in the simulation. The pressure can be given either as a scalar value, as a vector of length 3 denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. A scalar value will result in isotropic pressure coupling, whereas for all other representations, each pressure component will be coupled to the barostat independently. **Default:** `1.0*bar`
- **barostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen pressure coupling. **Default:** `1000.0 * fs`

[1] H.ăJ.ăC. Berendsen, J.ăP.ăM. Postma, W.ăF. van Gunsteren, A.ăDiNola, and J.ăR. Haak. Molecular dynamics with coupling to an external bath. *J. Chem. Phys.*, 81(8):3684–3690, 1984. doi:http://dx.doi.org/10.1063/1.448118.

- **compressibility** (*PhysicalQuantity* of type pressure**-1) – The estimated compressibility of the system relating volume changes to pressures changes. **Default:** `1.0e-4*bar**-1`
- **coupling_mask** (*array of bools*) – The mask determining which elements of the stress tensor, the barostat should couple to. Can be given as a vector of length 3, denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. By default the barostat couples only to the diagonal elements. **Default:** True for diagonal elements, False for shear components.
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `None`
- **compression_rate** (*PhysicalQuantity* of type pressure/time | None) – The compression rate of the target pressure. **Default:** `0.0 * bar/fs`
- **random_seed** (*int*) – The seed for the random generator providing the stochastic part of the velocity scaling. Must be between 0 and 2**32. **Default:** The default random seed

isotropicCoupling()

Returns
Whether the barostat applies isotropic or anisotropic pressure coupling.

Return type
bool

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si in the isothermal-isobaric ensemble using the NPTBernettiBussi method:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up MD method
method = NPTBernettiBussi(
    time_step=1*femtoSecond,
    initial_velocity=MaxwellBoltzmannDistribution(temperature=300*Kelvin),
    # Thermostat settings
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    # Barostat settings
    reservoir_pressure=1.0*bar,
    barostat_timescale=1000.0*femtoSecond,
    compressibility=1.0e-4*bar**-1,
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.hdf5',
    steps=50,
    log_interval=10,
    method=method
)
```

`nptbernettibussi.py`

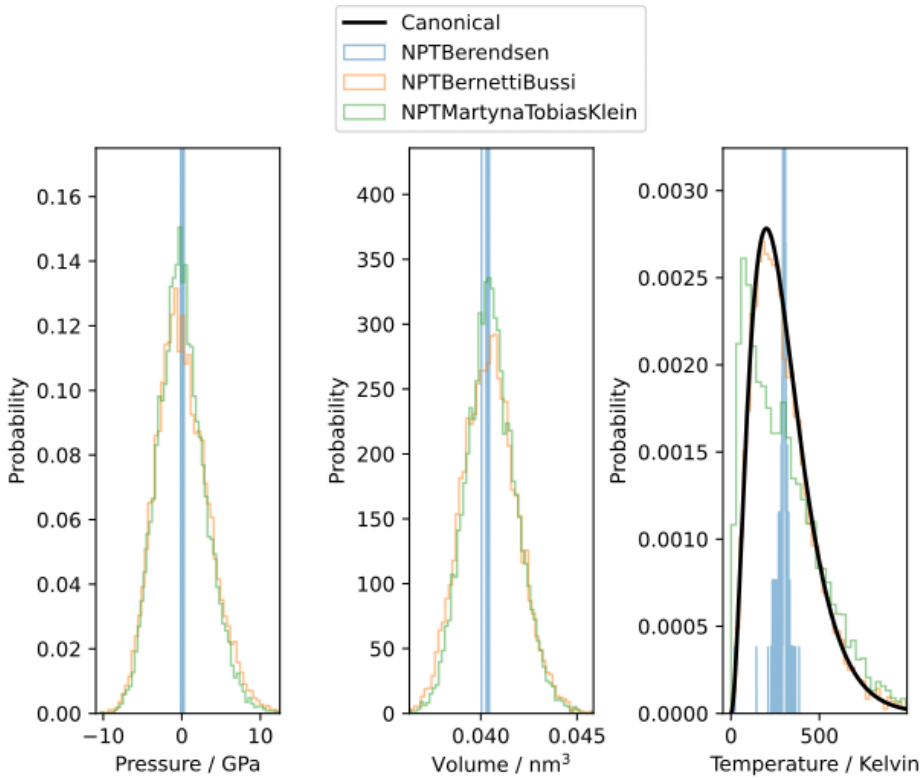
The Bernetti-Bussi thermostat combines the stochastic velocity rescaling thermostat[1] for temperature controll with the stochastic cell rescaling barostat[2] for pressure control. It correctly produces the volume and temperature distributions of the NPT ensemble.

The script below compares the (instantaneous) pressure, volume and temperature distributions generated by three different barostats: Both the Martyna-Tobias-Klein (MTK) barostat and the Bernetti-Bussi barostat give the correct broad volume and huge pressure fluctuations, while the Berendsen barostat produces a very narrow distribution.

Since the center of mass is constrained, the system has only 3 degrees of freedom. The deterministic MTK method has difficulties in reaching the canonical distribution for the instantaneous temperature on such a small system.

[1] Giovanni Bussi, Davide Donadio, and Michele Parrinello. Canonical sampling through velocity rescaling. *Journal of Chemical Physics*, 126(1):014101, 2007. doi:10.1063/1.2408420.

[2] Mattia Bernetti and Giovanni Bussi. Pressure control using stochastic cell rescaling. *The Journal of chemical physics*, 153 11:114107, 2020.



`compare_PVT_distributions.py`

Notes

- The Bernetti-Bussi barostat[Page 3433, 2,3] is a stochastic variant of the Berendsen barostat. However, unlike the Berendsen barostat it correctly samples the isothermal-isobaric ensemble. The lattice vectors of the cell and the positions are scaled by a random number, which is chosen such as to achieve the correct volume fluctuations.
- If a scalar `reservoir_pressure` value is given, isotropic pressure coupling is applied, i.e. all cell vectors are rescaled by the same factor, and the isotropic pressure $P = 1/3(P_{xx} + P_{yy} + P_{zz})$ is used.

  If a pressure vector (of length 3 or 6, in Voigt notation), or a 3x3-tensor is given as `reservoir_pressure`, anisotropic pressure coupling is applied, meaning that each degree of freedom of the cell vectors will be rescaled independently.

  The coupling to selected components of the pressure tensor can be enabled/disabled via the parameter `coupling_matrix`. This parameter only has an effect with anisotropic pressure coupling.

  By default, the barostat couples only to the hydrostatic pressure, i.e. the diagonal elements of the pressure tensor.
- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

  If a non-zero `compression_rate` is specified, the reservoir pressure will be changed linearly during the simulation, according to the specified compression rate.

[3] VittorioăDel Tatto, Paolo Raiteri, Mattia Bernetti, and Giovanni Bussi. Molecular Dynamics of Solids at Constant Pressure and Stress Using Anisotropic Stochastic Cell Rescaling. *Applied Sciences*, 2022.

NPTMartynaTobiasKlein

class NPTMartynaTobiasKlein(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *thermostat_timescale=None*, *reservoir_pressure=None*, *barostat_timescale=None*, *coupling_mask=None*, *heating_rate=None*, *compression_rate=None*, *chain_length=None*)

The NPTMartynaTobiasKlein integrator class which implements the barostat proposed by Martyna, Tobias, and Klein.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. **Default:** `300.0 * Kelvin`
- **thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for MTK temperature coupling. **Default:** `100.0 * fs`
- **reservoir_pressure** (*PhysicalQuantity* of type pressure) – The reservoir pressure in the simulation. The pressure can be given either as a scalar value, as a vector of length 3 denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. A scalar value will result in isotropic pressure coupling, whereas for all other representations, each pressure component will be coupled to the barostat independently. **Default:** `1.0 * bar`
- **barostat_timescale** (*PhysicalQuantity* of type time) – The time constant for MTK pressure coupling. **Default:** `1000.0 * fs`
- **coupling_mask** (*array of bools*) – The mask determining which elements of the stress tensor, the barostat should couple to. Can be given as a vector of length 3, denoting the hydrostatic components, as a vector of length 6 in Voigt notation, or as a 3x3 tensor. By default the barostat couples only to the diagonal elements. **Default:** `True` for diagonal elements, `False` for shear components.
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `None`
- **compression_rate** (*PhysicalQuantity* of type pressure/time | None) – The compression rate of the target pressure. **Default:** `0.0 * bar / fs`
- **chain_length** (*int*) – The number of subsequent thermostats. **Default:** `3`

isotropicCoupling()

Returns
Whether the barostat applies isotropic or anisotropic pressure coupling.

Return type
bool

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()
Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the MartynaTobiasKlein barostat with isotropic pressure coupling:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up a MTK-barostat with isotropic pressure coupling.
method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    reservoir_pressure=1.0*bar,
    barostat_timescale=500.0*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    compression_rate=0.0*bar/picoSecond,
    chain_length=3,
    initial_velocity=None
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
```

(continues on next page)

(continued from previous page)

```
    method=method
)
```

Apply anisotropic pressure coupling only to the zz-direction of the pressure tensor.

```
method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    reservoir_pressure=[0.0, 0.0, 10.0]*GPa,
    barostat_timescale=500.0*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    coupling_mask=[False, False, True, False, False, False],
    compression_rate=0.0*bar/picoSecond,
    chain_length=3,
    initial_velocity=None
)
```

Notes

- The MartynaTobiasKlein barostat can be used to simulate a canonical NPT-ensemble. Similar to the *NVTNoseHoover* algorithm, chains of subsequent thermostats and barostats are used to suppress large temperature and pressure oscillations. The equations of motion are taken from Ref.[1], whereas the integration algorithm is implemented as described in Ref[2].

- If a scalar `reservoir_pressure` value is given, isotropic pressure coupling is applied, i.e. all cell vectors are rescaled by the same factor, and the isotropic pressure $P = 1/3(P_{xx} + P_{yy} + P_{zz})$ is used.

  If a pressure vector (of length 3 or 6, in Voigt notation), or a 3x3-tensor is given as `reservoir_pressure`, anisotropic pressure coupling is applied, meaning that each degree of freedom of the cell vectors will be rescaled independently.

  You can enable/disable the coupling to selected components of the pressure tensor via the parameter `coupling_matrix`. This parameter only has an effect with anisotropic pressure coupling.

  By default, the barostat couples only to the hydrostatic pressure, i.e. the diagonal elements of the pressure tensor.

- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

  If a non-zero `compression_rate` is specified, the reservoir pressure will be changed linearly during the simulation, according to the specified compression rate.

[1] GlennăJ. Martyna, DouglasăJ. Tobias, and MichaelăL. Klein. Constant pressure molecular dynamics algorithms. *J. Chem. Phys.*, 101(5):4177–4189, 1994. doi:http://dx.doi.org/10.1063/1.467468.

[2] MarkăE Tuckerman, José Alejandre, Roberto López-Rendón, AndreaăL Jochim, and GlennăJ Martyna. A Liouville-operator derived measure-preserving integrator for molecular dynamics simulations in the isothermal–isobaric ensemble. *J. Phys. A: Mathematical and General*, 39(19):5629, 2006. URL: http://stacks.iop.org/0305-4470/39/i=19/a=S18.

NRLHamiltonianParametrization

class NRLHamiltonianParametrization(*parameters*)

Class representing the Hamiltonian parametrization for the Navy Research Laboratory Tight-Binding model.

Parameters
parameters (NRLTightBindingParameters) – An object describing the parameters used for the in the NRL model.

basisSet()

Returns the Slater-Koster basis set used to construct the environmental independent component of the Hamiltonian and Overlap.

Returns
The Slater-Koster basis set.

Return type
SlaterKosterTable

pairPotentials()

Return the pair potentials. If no pair potentials are available, it returns None.

Returns
The pair potentials.

Return type
PairPotential

parameters()

Returns
A dictionary with all the numerical parameters.

Return type
dict

usingOrthogonalBasis()

Routine for determining if the Hamiltonian parametrization is using an orthogonal basis set.

Returns
True if the basis is orthogonal, False otherwise.

Return type
bool

Usage Examples

The following example demonstrates some of the different parameters available for a NRL Hamiltonian parametrization.

```
# Create a NRL Hamiltonian parametrization with the sp
# parameters for Silicon.
hamiltonian_parametrization = NRLHamiltonianParametrization(
    parameters=NRLParameters.Si_sp)

# NRL parametrization with the parameters for Carbon.
hamiltonian_parametrization = NRLHamiltonianParametrization(
    parameters=NRLParameters.C)
```

(continues on next page)

(continued from previous page)

```
# Set up a semi-empirical calculator with the NRL parametrization.
calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization)
```

Notes

The Naval Research Laboratory tight-binding (NRL-TB) method uses distance- and environment-dependent Slater-Koster parameters to provide transferability between different atomic and molecular structures. The NRL parametrization is thoroughly described in[1]. The Slater-Koster model, which the NRL parametrization is based on, is described in *Background information*.

The available NRL parameter sets in QuantumATK can be found in the following table: *NRL style parameters for electronic structure and total energy calculations.*

NVEVelocityVerlet

class NVEVelocityVerlet(*initial_velocity=None*, *time_step=None*, *max_force_threshold=None*)

Constructor for the NVE Velocity-Verlet integrator class.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1.0*fs`
- **max_force_threshold** (*PhysicalQuantity* of type energy/length) – The maximum allowed force to run MD simulation with single time step. If maximum force in an MD simulation is larger than the threshold force, multiple time steps MD simulation is performed. **Default:** None

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

[1] D.ăA. Papaconstantopoulos and M.ăJ. Mehl. The Slater-Koster tight-binding method: A computationally efficient and accurate approach. *J. Phys. Condens. Matter*, 15(10):R413–R440, 2003. doi:10.1088/0953-8984/15/10/201.

Usage Examples

Perform a micro-canonical molecular dynamics run of 10 steps on a water molecule, using the NVE Verlet Velocity MD method:

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[0.0,  -1.70000000e-05,   1.20198000e-01],
                         [0.0,   7.59572000e-01,  -4.86714000e-01],
                         [0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Define a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# Perform MD
md_configuration = MolecularDynamics(molecule_configuration,
                                     method=NVEVelocityVerlet(),
                                     steps=10)

# Save the final configuration
nlsave('velocityverlet.nc', md_configuration)
```

nvevelocityverlet.py

Notes

Uses the Verlet algorithm to perform a micro-canonical, classical molecular-dynamics simulation, i.e. the energy of the system is constant during the simulation.

NVTBerendsen

class NVTBerendsen(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *heating_rate=None*, *thermostat_timescale=None*)

The NVT Berendsen integrator class which implements the Berendsen thermostat.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. The temperature can be given as a single temperature value for the entire system, or as a list of 2-tuples of str and

PhysicalQuantity of type temperature, applying local thermostats to the tagged groups of atoms. E.g. `[('group1', 280.0 * Kelvin), ('group2', 320.0 * Kelvin)]`. **Default:** `300.0 * Kelvin`

- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `0.0 * Kelvin / fs`
- **thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen temperature coupling. **Default:** `100.0 * fs`

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the Berendsen thermostat:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up MD method
method = NVTBerendsen(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
```

(continues on next page)

(continued from previous page)

```
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    initial_velocity=None
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
    method=method
)
```

nvtberendsen.py

Apply two thermostats to tagged groups of atoms of the configuration:

```
# Set up MD method with two thermostats on tagged groups of atoms.
method = NVTBerendsen(
    time_step=1*femtoSecond,
    reservoir_temperature=[('region1', 300*Kelvin), ('region2', 600*Kelvin)],
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    initial_velocity=None
)
```

nvtberendsen2.py

Notes

- The Berendsen thermostat can be used to approximately simulate a canonical NVT-ensemble, as described in Ref.[1].
- You can specify one or more thermostats acting on tagged sub-groups of atoms of the configuration, by giving a list of `(tag-name, temperature)`-tuples instead of a single, global `reservoir_temperature` value.
- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

NVTBussiDonadioParrinello

class NVTBussiDonadioParrinello(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *heating_rate=None*, *thermostat_timescale=None*, *random_seed=None*)

The NVT Bussi-Donadio-Parrinelllo thermostat.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*

[1] H.ăJ.ăC. Berendsen, J.ăP.ăM. Postma, W.ăF. van Gunsteren, A.ăDiNola, and J.ăR. Haak. Molecular dynamics with coupling to an external bath. *J. Chem. Phys.*, 81(8):3684–3690, 1984. doi:http://dx.doi.org/10.1063/1.448118.

- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. The temperature can be given as a single temperature value for the entire system, or as a list of 2-tuples of str and *PhysicalQuantity* of type temperature, applying local thermostats to the tagged groups of atoms. E.g. `[('group1', 280.0 * Kelvin), ('group2', 320.0 * Kelvin)]`. **Default:** `300.0 * Kelvin`
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `0.0 * Kelvin / fs`
- **thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for temperature coupling. **Default:** `100.0 * fs`
- **random_seed** (*int*) – The seed for the random generator providing the stochastic part of the velocity scaling. Must be between 0 and 2**32. **Default:** The default random seed

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the Bussi-Donadio-Parrinello thermostat:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
```

(continues on next page)

(continued from previous page)

```
bulk_configuration.setCalculator(calculator)

# Set up MD method
method = NVTBussiDonadioParrinello(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    initial_velocity=None
)

# Run MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
    method=method
)
```

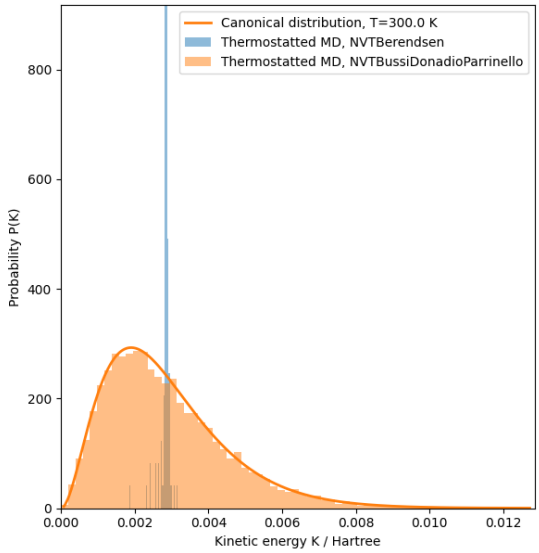
nvtbussidonadioparrinello.py

The script below demonstrates that the Bussi-Donadio-Parrinello produces a NVT ensemble, while the Berendsen thermostat fails to do so. The histogram of the kinetic energies from a long molecular dynamics trajectory is compared with the expected canonical distribution for the kinetic energy.

The analytical expression for the canonical equilibrium distribution of the kinetic energy K at temperature T is given by

$$P(K)dK = \frac{\beta^{N_f/2}}{\Gamma(N_f/2)} K^{N_f/2-1} \exp(-\beta K) dK$$

where $\beta = 1/(k_B T)$ and N_f is the number of degrees of freedom.




```
import matplotlib.pyplot as plt

# Increase the number of MD steps to converge the histograms.
STEPS = 5000

plt.xlabel("Kinetic energy K / Hartree")
plt.ylabel("Probability P(K)")

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Run MD simulations with each thermostat.
for thermostat_class in NVTBerendsen, NVTBussiDonadioParrinello:
    # Set up MD method
    method = thermostat_class(
        time_step=1*femtoSecond,
        reservoir_temperature=300*Kelvin,
        thermostat_timescale=100*femtoSecond,
        heating_rate=0*Kelvin/picoSecond,
        initial_velocity=ZeroVelocities(),
    )

    # Run long MD simulation
    md_trajectory = MolecularDynamics(
```

(continues on next page)

(continued from previous page)

```
        bulk_configuration,
        constraints=[],
        trajectory_filename='trajectory.hdf5',
        steps=STEPS,
        log_interval=50,
        method=method
    )

    # Choose color and label for thermostat
    color = next(plt.gca()._get_lines.prop_cycler)["color"]
    label = thermostat_class.__name__

    # Kinetic energy distribution from samples in MD trajectory.
    temperatures = numpy.unique(md_trajectory.reservoirTemperatures())
    assert len(temperatures) == 1, 'Reservoir temperature should not change during MD␣
↪run.'
    temperature = temperatures[0]
    beta = (1.0/(boltzmann_constant * temperature)).inUnitsOf(Hartree**(-1))
    ekin = md_trajectory.kineticEnergies().inUnitsOf(Hartree)

    # Plot histogram of kinetic energies
    plt.hist(
        ekin,
        bins='auto', alpha=0.5,
        color=color, stacked=True, density=True, label=f'Thermostatted MD, {label}'
    )

# Find number of degrees of freedom
n = bulk_configuration.numberOfAtoms()
Nf = 3*n

# To avoid overflows reorganize the expression for P(K),
# so the logs of large quantities cancel before exponentiating.
def logGamma(x):
    """ Expansion of log(Gamma(x)) for large x """
    lG = (x-0.5) * numpy.log(x) - x + 0.5*numpy.log(2.0*numpy.pi)
    return lG

# For K=0.0, log(beta**K) gives inf.
ekin = ekin[ekin > 0.0]
ekin.sort()

probability_canonical = numpy.exp(
    Nf/2.0 * numpy.log(beta*ekin) - beta*ekin - logGamma(Nf/2.0)
) / ekin

# Plot canonical distribution
plt.plot(
    ekin, probability_canonical,
    color=color, lw=2, label=f'Canonical distribution, T={temperature.inUnitsOf(Kelvin)}␣
↪K'
)
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```

kinetic_energy_distribution.py

Notes

- The Bussi-Donadio-Parrinello thermostat[1],[2] is a stochastic variant of the Berendsen thermostat. However, unlike the Berendsen thermostat it correctly samples the canonical NVT-ensemble. The factor for scaling the velocities is a random number, which is generated in such a way as to produce the canonical equilibrium distribution for the kinetic energy.
- You can specify one or more thermostats acting on tagged sub-groups of atoms of the configuration, by giving a list of `(tag-name, temperature)`-tuples instead of a single, global `reservoir_temperature` value.
- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

NVTNoseHoover

class NVTNoseHoover(*initial_velocity=None*, *time_step=None*, *reservoir_temperature=None*, *heating_rate=None*, *thermostat_timescale=None*, *chain_length=None*, *max_force_threshold=None*)

The NVT Nose-Hoover integrator class.

Parameters

- **initial_velocity** (*ConfigurationVelocities* | *ZeroVelocities* | *MaxwellBoltzmannDistribution*) – A class that implements a distribution of initial velocities for the particles in the MD simulation. **Default:** *MaxwellBoltzmannDistribution*
- **time_step** (*PhysicalQuantity* of type time) – The time-step interval used in the MD simulation. **Default:** `1.0 * fs`
- **reservoir_temperature** (*PhysicalQuantity* of type temperature | list) – The reservoir temperature in the simulation. The temperature can be given as a single temperature value for the entire system, or as a list of 2-tuples of str and *PhysicalQuantity* of type temperature, applying local thermostats to the tagged groups of atoms. E.g. `[('group1', 280.0 * Kelvin), ('group2', 320.0 * Kelvin)]`. **Default:** `300.0 * Kelvin`
- **heating_rate** (*PhysicalQuantity* of type temperature/time | None) – The heating rate of the target temperature. A value of None disables the heating of the system. **Default:** `0.0 * Kelvin / fs`
- **thermostat_timescale** (*PhysicalQuantity* of type time) – The time constant for Berendsen temperature coupling. **Default:** `100.0 * fs`
- **chain_length** (*int*) – The number of subsequent Nose-Hoover thermostats. Zero means no thermostat is invoked. **Default:** `3`
- **max_force_threshold** (*PhysicalQuantity* of type energy/length) – The maximum allowed force to run MD simulation with single time step. If maximum force in a MD simulation is larger than the threshold force, multiple time steps MD simulation is performed. **Default:** None

[1] Giovanni Bussi, Davide Donadio, and Michele Parrinello. Canonical sampling through velocity rescaling. *Journal of Chemical Physics*, 126(1):014101, 2007. doi:10.1063/1.2408420.

[2] Giovanni Bussi and Michele Parrinello. Stochastic thermostats: comparison of local and global schemes. *Computer Physics Communications*, 179(1-3):26–29, 2008. doi:10.1016/j.cpc.2008.01.006.

kineticEnergy(*configuration*)

Parameters
configuration (*DistributedConfiguration*) – The current configuration to calculate the kinetic energy of.

Returns
The kinetic energy of the current configuration.

Return type
PhysicalQuantity of type energy

thermostats()

Returns
The list of thermostats.

Return type
list

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Perform a molecular dynamics run of 50 steps on FCC Si, using the Nose-Hoover thermostat:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    fractional_coordinates=[[0.0, 0.0, 0.0], [0.25, 0.25, 0.25]]
    )

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_Si_1988b())
bulk_configuration.setCalculator(calculator)

# Set up MD method
method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=None
)

# Run MD simulation
```

(continues on next page)

(continued from previous page)

```
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=50,
    log_interval=10,
    method=method
)
```

nvtnosehoover.py

Apply two thermostats to tagged groups of atoms of the configuration:

```
# Set up MD method with two thermostats on tagged groups of atoms.
method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=[('region1', 300*Kelvin), ('region2', 600.0*Kelvin)],
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=None
)
```

nvtnosehoover2.py

Notes

- The Nose-Hoover-thermostat uses several subsequent thermostats to reproduce a canonical ensemble. For more details regarding the Nose-Hoover chain method, please consult Martyna et al.[1].
- You can specify one or more thermostats acting on tagged sub-groups of atoms of the configuration, by giving a list of `(tag-name, temperature)`-tuples in stead of a single, global `reservoir_temperature` value.
- If a non-zero `heating_rate` is specified, the reservoir temperature will be changed linearly during the simulation, according to the specified heating rate.

NamedPointDefect

class NamedPointDefect(*point_defect*, *index*, *name*, *constituent_indices=None*, *constituent_point_defects=None*)

Create an object that stores a point defect with extra information about the name, index and constituent defects (if any).

Parameters

- **point_defect** (*Vacancy* | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*) – The point defect to store
- **index** (*int*) – The index of unique defects of the specific defect.
- **name** (*str*) – The name of the defect.

[1] G.ăJ. Martyna, M.ăL. Klein, and M.ăTuckerman. Nose-Hoover chains: The canonical ensemble via continuous dynamics. *J. Chem. Phys.*, 97(4):2635–2643, 1992. doi:10.1063/1.463940.

- **constituent_indices** (*list of int | None*) – Indices of the defects that make up the named defect. None for single defects, list of int for compound defects.
- **constituent_point_defects** (*list of defect types | None*) – The defects that make up the named defect. None for single defects, list of defect types for compound defects.

constituentIndices()

Returns
Indices of the defects that make up the named defect. None for single defects, list of int for compound defects.

Return type
list of int | None

constituentPointDefects()

Returns
The defects that make up the named defect. None for single defects, list of defect types for compound defects.

Return type
list of defect types | None

index()

Returns
Return the defect unique index

Return type
int

name()

Returns
Return the name of the defect

Return type
str

pointDefect()

Returns
Return the stored point defect.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

uniqueString()

Return a unique string representing the state of the object.

Notes

The *NamedPointDefect* contains a point defect definition as well as some information regarding the symmetry of the defect. This information is used to uniquely identify symmetrically equivalent representations of the defect in *ChargedPointDefectConfiguration* and diffusion calculations. *NamedPointDefect* objects are automatically created by defect generators derived from the *BaseDefectGenerator* class.

NearestNeighbor

class NearestNeighbor(*md_trajectory*, *cutoff_radius=None*, *start_time=None*, *end_time=None*, *pair_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the time-dependent nearest neighbors from an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the nearest neighbors for.
- **cutoff_radius** (*PhysicalQuantity* of type length) – Cut off distance for nearest neighbors (must be positive). **Default:** `2.5 * Angstrom`
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **pair_selection** (*sequence*) – Only include contributions between this selection of atoms. A sequence has to contain two of the following types: Element, tag name, list of indices, or None. **Default:** all atoms pairs.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Returns
An array with the average number of nearest neighbors for each selected snapshot in the first column and its variance in the second column.

Return type
array

times()

Return the time values associated with the selected snapshots.

Usage Examples

Load an *MDTrajectory*, calculate the average nearest neighbor number and the corresponding variance as a function of time and plot the results:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

nearest_neighbor = NearestNeighbor(md_trajectory,
                                   start_time=10000.0*fs,
                                   end_time=50000.0*fs,
                                   cutoff_radius=2.5*Angstrom)

# Get the times in ps and the nearest neighbor values.
t       = nearest_neighbor.times().inUnitsOf(ps)
nn_data = nearest_neighbor.data()

# Plot the nearest neighbor number and its variance using pylab.
import pylab
```

(continues on next page)

(continued from previous page)

```
pylab.plot(t, nn_data[:, 0], label='Number of nearest neighbors')
pylab.plot(t, nn_data[:, 1], label='Variance nearest neighbors')
pylab.xlabel('t (ps)')
pylab.ylabel('<NN>')
pylab.legend()

pylab.show()
```

nearest_neighbor.py

Notes

This analysis calculates for each snapshot in an *MDTrajectory* the average number of nearest neighbors of a selected group of atoms (the first entry in `pair_selection`) with a second selected group of atoms (the second entry in `pair_selection`). By default the nearest neighbors are calculated among all atoms in the configuration.

In contrast to *CoordinationNumber*, which calculates the evolution of the number of atoms with a *given coordination number*, this analysis evaluates the *average number* of nearest neighbors along the simulation time.

NeumannBoundaryCondition

class NeumannBoundaryCondition(*value=None*)

A Neumann type boundary condition.

Parameters
value (`PhysicalQuantity` of type energy/length) – The constraint value to fix the derivative of the Hartree difference potential to. **Default:** `0.0 * eV/Angstrom`

value()

Returns
The constraint value.

Return type
`PhysicalQuantity` of type energy/length

NeutralAtom

class NeutralAtom(*electrode_constraint_length=None*)

Class for representing a neutral atom initial density matrix for Device simulations.

Parameters
electrode_constraint_length (*PhysicalQuantity* of type length) – Length over which the atoms in the electrode copy within the central region will be initialized with a linear combination of the converged density matrix of the electrodes and the neutral atom density matrix. The weight of the electrode density matrix is 1 at the boundary and 0 at the distance `electrode_constraint_length` from the boundary. **Default:** `10.0 * Angstrom`

electrodeConstraintLength()

Returns
The electrode constraint length.

Return type
PhysicalQuantity of type length

static initializeCentralRegionDensityMatrix(*device_builder*, *downgraded_calculator*, *schroedinger_container*)

Method for initializing the density matrix of the central region.

Parameters

- **device_builder** (DeviceLCAOBuilder | DeviceSemiEmpiricalBuilder) – The device builder.
- **downgraded_calculator** (*LCAOCalculator* | *HuckelCalculator* | *SlaterKosterCalculator*) – Not used.
- **schroedinger_container** (NLEngine.CompositeSchroedingerContainer) – The device Schroedinger container, the density matrix of which will be initialized.

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Setup a non-self-consistent calculation with a neutral atom initial density.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    initial_density_type=Neutral(electrode_constraint_length=0.*Ang))
    calculator=DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    iteration_control_parameters=NonSelfconsistent,
    )
```

Notes

- The *NeutralAtom* initial density type, corresponds to an initial density consisting of a superposition of atomic-like charge densities, constructed by occupying the basis orbital corresponding to the occupations of the atomic wave functions
- If the electrode_constraint_length is longer than the length of the electrode, still only the electrode copy atoms within the central region will be affected. However, the weighting of the electrode density matrix will be larger. For instance, if electrode_constraint_length is infinite, the initial density matrix of the electrode copy atoms in the central region will be the electrode density matrix.

NeutronScattering

class NeutronScattering(*md_trajectory*, *start_time=None*, *end_time=None*, *pair_selection=None*, *maximum_q_value=None*, *q_resolution=None*, *scattering_length=None*, *cutoff_radius=None*, *resolution=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the total neutron scattering structure factor from an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the neutron scattering for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** 0.0 * fs
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **pair_selection** (*sequence*) – Only include contributions between this selection of atoms. A sequence has to contain two of the following types: Element, tag name, list of indices, or None. **Default:** all atoms pairs
- **maximum_q_value** (*PhysicalQuantity* of type inverse length) – The maximum scattering vector length. **Default:** 15.0 / Angstrom
- **q_resolution** (*PhysicalQuantity* of type inverse length) – The resolution of the neutron structure factor curve. **Default:** 0.05/Angstrom
- **scattering_length** (*PhysicalQuantity* of type length) – A dictionary of elements and corresponding neutron scattering lengths. By default it is assumed that all elements have the same scattering length. **Default:** None
- **cutoff_radius** (*PhysicalQuantity* of type length.) – Upper limit on sampled distances. **Default:** Half the diagonal of the unit
- **resolution** (*PhysicalQuantity* of type length) – The bin size determining the resolution of the underlying radial-distribution-function. **Default:** 0.05 * Angstrom
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the neutron scattering structure factor.

qRange()

Return the list of scattering vector magnitudes.

Usage Examples

Load an *MDTrajectory* and calculate the neutron scattering structure factor using experimental values for the neutron scattering lengths of the different elements (see Ref.[1]):

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

# Define the neutron scattering lengths for the elements.
scattering_length={Aluminum : 0.00003449*Angstrom,
                   Oxygen   : 0.00005805*Angstrom}
```

(continues on next page)

[1] G.ăGutierrez and B.ăJohansson. Molecular dynamics study of structural properties of amorphous Al2O3. *Phys. Rev. B*, 65:104202, Feb 2002. doi:10.1103/PhysRevB.65.104202.

(continued from previous page)

```
neutron_scattering = NeutronScattering(md_trajectory,
                                       scattering_length=scattering_length)

# Get the q-values and the neutron structure function.
q_values = neutron_scattering.qRange().inUnitsOf(Angstrom**-1)
s_q      = neutron_scattering.data()

# Plot the data using pylab.
import pylab

pylab.plot(q_values, s_q, label='Structure factor of alumina')
pylab.xlabel('q (1/Ang)')
pylab.ylabel('S(q)')
pylab.legend()

pylab.show()
```

neutron_scattering.py

Notes

This object calculates the isotropic (i.e. averaged over all scattering angles) neutron scattering structure factor. It is therefore predominantly aimed at the characterization of amorphous materials.

The neutron scattering structure factor is calculated as described in Ref.[Page 3454, 1]:

$$S(q) = \frac{\sum_{\alpha,\beta} b_\alpha b_\beta (c_\alpha c_\beta)^{1/2} \left[S_{\alpha,\beta}(q) - \delta_{\alpha,\beta} + (c_\alpha c_\beta)^{1/2}\right]}{\left(\sum_\alpha b_\alpha c_\alpha\right)^{1/2}},$$

where c_α and b_α are the mole fraction and neutron scattering length of element α. The *PartialStructureFactor* is calculated from the partial radial distribution functions, $g_{\alpha,\beta}(r)$, as follows,

$$S_{\alpha,\beta}(q) = \delta_{\alpha,\beta} + 4\pi\rho(c_\alpha c_\beta)^{1/2} \int_0^{r_{max}} r^2 [g_{\alpha,\beta}(r) - 1] \frac{\sin(qr)}{qr} \frac{\sin(\pi r_{max} r)}{\pi r_{max} r} dr.$$

If no scattering lengths are provided (default), the partial structure factors are weighted only by the respective mole fractions, i.e. the scattering lengths are all set to one. As the scattering lengths are typically of the order of femto-meter, you should either provide scattering lengths for all elements or none at all to avoid overestimating the contributions of some elements.

If a single pair of elements, or a selection via indices or tags, is specified in the `pair_selection` parameter, the *PartialStructureFactor* of the selected pair is returned.

The `cutoff_radius` parameter determines the range of the radial distribution function. This parameter might have an influcence on the calculated structure factor at small scattering vectors.

NoStorage

class NoStorage

Do not cache self energies.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Examples on how to use *NoStorage()* and other storage strategies can be found in the *Usage Examples* of *Recursion-SelfEnergy*.

Notes

In each step of the SCF loop of a device calculation, the same self energies need to be evaluated.

The *NoStorage()* storage strategy specifies that each self energy is recalculated in every iteration. This strategy does not require additional memory, but results in poorer performance compared to the *StoreOnDisk()* and *SaveInMemory()* strategies.

NonEquilibriumContour

class NonEquilibriumContour(*args*)

Check the parameters given, and which of them are using the default value.

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

This class is the abstract base class for the different Contour types that are used to compute the non-equilibrium part of the density matrix at finite bias.

At this point there is only one NonEquilibriumContour available: *RealAxisContour*.

NonEquilibriumHeatExchange

class NonEquilibriumHeatExchange(*configuration*, *heating_power*, *heat_source*, *heat_sink*, *call_interval=None*, *update_profile_interval=None*, *profile_resolution=None*, *exchange_interval=1*)

A class that implements a heat flow by constant heating power exchange technique via a hook function.

Parameters

- **configuration** (*BulkConfiguration*) – The initial configuration on which the heat flow simulation will be performed.

- **heating_power** (*PhysicalQuantity* of type energy/time) – The thermal energy that is transferred from the cold to the hot region per time.
- **heat_source** (*str | list of ints | tuple of type (float, PhysicalQuantity of type length)*) – The heat source atoms can be defined by giving a tag that specifies which atoms are in the heat source, a list of atomic indices, or by defining a spatial region. The spatial region is defined as 2-tuple of the origin and length. The origin is the fractional position along the c-axis and the length is the thickness along the c-axis.
- **heat_sink** (*str | list of ints | tuple of type (float, PhysicalQuantity of type length)*) – The heat sink atoms can be defined by giving a tag that specifies which atoms are in the heat sink, a list of atomic indices, or by defining a spatial region. The spatial region is defined as 2-tuple of the origin and length. The origin is the fractional position along the c-axis and the length is the thickness along the c-axis.
- **call_interval** (*int*) – The interval to perform energy exchange by rescaling the velocities. **Default:** 1 (every step).
- **update_profile_interval** (*int*) – [description].
- **profile_resolution** (*PhysicalQuantity* of type length) – The spatial resolution for the on-the-fly calculation of the temperature profile. **Default:** 2.0*Angstrom.
- **exchange_interval** (*int*) – The interval to perform energy exchange by rescaling the velocities. **Default:** 1 (every step).

callInterval()

Returns
The call interval of this hook function.

Return type
int

dz()

Returns
Float version of the profile resolution.

Return type
float

exchangeInterval()

Returns
The hot and cold region velocity exchange interval.

Return type
int

heatCurrent()

Returns
The average heat current.

Return type
PhysicalQuantity of type energy/time

heatSink()

Returns
The heat sink.

Return type
SpatialHeatRegion

heatSource()

Returns
The head source.

Return type
SpatialHeatRegion

masses()

Returns
The masses for the entire system and for the two regions.

Return type
PhysicalQuantity of type mass

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a formatted string with the average temperature profile, as well as the average heat current.

Parameters
stream (*A stream that supports strings being written to using 'write'.*) – The stream the temperature profile is written to.

nslices()

Returns
The number of bins (slices) to store the temperature profile.

Return type
int

profileResolution()

Returns
The temperature profile resolution.

Return type
PhysicalQuantity of type length

temperatureProfile()

Returns
The temperature profile as a tuple containing bin-centers and corresponding temperature values.

Return type
(*PhysicalQuantity* of type length, *PhysicalQuantity* of type temperature)

uniqueString()

Return a unique string representing the state of the object.

updateProfileInterval()

Returns
The temperature profile update interval.

Return type
int

Usage Example

Perform a non-equilibrium molecular dynamics simulation with a constant heating power on a silicon crystal with a single germanium layer, to obtain the thermal conductance through this impurity layer.

```
potentialSet = Tersoff_SiGe_1989()
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)

method = NVEVelocityVerlet(
    time_step=1*femtoSecond,
    initial_velocity=initial_velocity,
)

# Use a heating power of 50 nW.
heating_power = 50.0e-9*Joule/Second
momentum_exchange_hook = NonEquilibriumHeatExchange(
    configuration=bulk_configuration,
    heating_power=heating_power,
    heat_source='heat_source',
    heat_sink='heat_sink',
    update_profile_interval=10,
    profile_resolution=2.0*Ang
)

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='Si_w_Ge_layer_NEMD.hdf5',
    steps=100000,
    log_interval=500,
    post_step_hook=momentum_exchange_hook,
    method=method
)
```

Here, only the molecular dynamics block of the script is shown. The full script can be found found in the file `nonequilibriumheatexchange.py`.

Notes

- The *NonEquilibriumHeatExchange* method can be used to run *MolecularDynamics* simulations with a constant heat flux between the two selected reservoirs. The technique can therefore be used to simulate the thermal conductance of a given configuration (see e.g,[1]).
- The reverse non-equilibrium heat exchange method is implemented as a class, which can be used as a `post_step_hook` in *MolecularDynamics*.
- The velocities in the two reservoir regions are rescaled in such a way that a well-defined amount kinetic energy is added to the atoms in the `heat_source` region whereas the same amount is taken away from the kinetic energy in the `heat_sink` region. The magnitude of the kinetic energy that is transferred per time from the `heat_sink` to the `heat_source` can be specified by the `heating_power` parameter. By default the rescaling is performed at every MD step, if desired, the interval can be given by the user via the parameter `exchange_interval`.
- If a non-zero `update_profile_interval` is specified, a temperature profile is stored on-the-fly during the simulation. After the MD run, the profiles may be accessed using the method `temperatureProfile`.
- The thermal bulk conductivity can be obtained by plotting the temperature profile of the system using the **MD-Analyzer** tool and fitting its gradient ∇T along the transport direction. The thermal conductivity is then obtained via Fouriers law

$$\kappa = -\frac{\dot{Q}}{A\nabla T}$$

where $\dot{Q}$ is the specified `heating_power` and A is the cross-sectional area perpendicular to the transport direction.

- The thermal conductance across an interface G (*Kapitza conductance*) can be obtained as

$$G = -\frac{\dot{Q}}{\Delta T}$$

where ΔT is the temperature drop across the interface.

NonEquilibriumMomentumExchange

class NonEquilibriumMomentumExchange(*configuration*, *heat_source*, *heat_sink*, *call_interval=None*, *update_profile_interval=None*, *profile_resolution=None*, *exchange_interval=1*)

A class that implements a heat flow by non-equilibrium momentum exchange (reverse non-equilibrium MD (RNEMD)) technique via a hook function.

Parameters

- **configuration** (*BulkConfiguration*) – The initial configuration on which the heat flow simulation will be performed.
- **call_interval** (*int*) – After each exchange_interval iterations, a velocity swapping is performed between the hot and the cold region.
- **heat_source** (*str | list of ints | tuple of type (float, PhysicalQuantity of type length)*) – The heat source atoms can be defined by giving a tag that specifies which atoms are in the heat source, a list of atomic indices, or by defining a spatial region. The spatial region is defined as 2-tuple of the origin and length. The origin is the fractional position along the c-axis and the length is the thickness along the c-axis.

[1] Philippe Jund and Remi Jullien. Molecular-dynamics calculation of the thermal conductivity of vitreous silica. *Phys. Rev. B*, 59:13707–13711, 1999. doi:10.1103/PhysRevB.59.13707.

- **heat_sink** (*str | list of ints | tuple of type (float, PhysicalQuantity of type length)*) – The heat sink atoms can be defined by giving a tag that specifies which atoms are in the heat sink, a list of atomic indices, or by defining a spatial region. The spatial region is defined as 2-tuple of the origin and length. The origin is the fractional position along the c-axis and the length is the thickness along the c-axis.
- **update_profile_interval** (*int*) – The interval at which a measurement of the temperature profile is performed and added to the average profile. Set to zero to disable on-the-fly measurement. **Default:** `0`.
- **profile_resolution** (*PhysicalQuantity* of type length) – The spatial resolution for the on-the-fly calculation of the temperature profile. **Default:** `2.0 * Angstrom`.
- **exchange_interval** (*int*) – After each exchange_interval iterations, a velocity swapping is performed between the hot and the cold region.

averageHeatCurrent()

Returns
The average heat current.

Return type
PhysicalQuantity of type energy/time

callInterval()

Returns
The call interval of this hook function.

Return type
int

dz()

Returns
Float version of the profile resolution.

Return type
float

exchangeInterval()

Returns
The hot and cold region velocity exchange interval.

Return type
int

heatSink()

Returns
The heat sink.

Return type
SpatialHeatRegion

heatSource()

Returns
The head source.

Return type
SpatialHeatRegion

masses()

Returns
The masses for the entire system and for the two regions.

Return type
PhysicalQuantity of type mass

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a formatted string with the average temperature profile, as well as the average heat current.

Parameters
stream (*A stream that supports strings being written to using 'write'.*) – The stream the temperature profile is written to.

nslices()

Returns
The number of bins (slices) to store the temperature profile.

Return type
int

profileResolution()

Returns
The temperature profile resolution.

Return type
PhysicalQuantity of type length

temperatureProfile()

Returns
The temperature profile as a tuple containing bin-centers and corresponding temperature values.

Return type
(*PhysicalQuantity* of type length, *PhysicalQuantity* of type temperature)

uniqueString()

Return a unique string representing the state of the object.

updateProfileInterval()

Returns
The temperature profile update interval.

Return type
int

Usage Example

Perform a molecular dynamics run of 5000 steps on a carbon nanotube. Momenta of two particles in the hot, respectively cold region are exchanged every 100 steps.

```
# Set up configuration
bulk_configuration = NanoTube(3,1)

# Add tags
```

(continues on next page)

(continued from previous page)

```
bulk_configuration.addTags('hot',  [24, 30, 31, 35, 36, 37, 41, 42, 45, 46, 47, 48, 49,␣
↪50, 51])
bulk_configuration.addTags('cold', [0, 1, 2, 3, 5, 6, 7, 10, 11, 16, 17])

# Set calculator
calculator = TremoloXCalculator(parameters=Tersoff_C_2010())
bulk_configuration.setCalculator(calculator)

method = NVEVelocityVerlet(
    time_step=0.5*femtoSecond,
    initial_velocity=MaxwellBoltzmannDistribution()
)

momentum_exchange_hook = NonEquilibriumMomentumExchange(
    configuration=bulk_configuration,
    exchange_interval=100,
    heat_source='hot',
    heat_sink='cold',
    update_profile_interval=0,
    profile_resolution=2.0*Ang
)

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='md_trajectory.hdf5',
    steps=5000,
    log_interval=50,
    measurement_hook=momentum_exchange_hook,
    method=method
)
```

nonequilibriummomentumexchange.py

Perform a non-equilibrium momentum exchange simulation on a polyvinyl chloride polymer system. The heat source atoms are determined at each exchange step as any atoms within 10 Angstrom of the origin of the c-axis and the heat sink atoms are any atoms within 10 Angstroms of the middle of the cell.

```
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)

method = NVEVelocityVerlet(
    time_step=1*femtoSecond,
    initial_velocity=initial_velocity,
)

momentum_exchange_hook = NonEquilibriumMomentumExchange(
    configuration=bulk_configuration,
    exchange_interval=100,
    heat_source=(0.0, 10.0*Angstrom),
    heat_sink=(0.5, 10.0*Angstrom),
```

(continues on next page)

(continued from previous page)

```
    update_profile_interval=0,
    profile_resolution=2.0*Ang
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='nemd_polymer.hdf5',
    steps=50000,
    log_interval=100,
    measurement_hook=momentum_exchange_hook,
    method=method
)

bulk_configuration = md_trajectory.lastImage()
nlprint(momentum_exchange_hook)
```

nemd_polymer.py

Notes

•The reverse non-equilibrium momentum-exchange (RNEMD) method is implemented as a class, which can be used as a `measurement_hook` in *MolecularDynamics*.

•The average heat current is recorded as a MD measurement. The time trace of this value can be used to monitor the convergence of the calculation. It can be easily accessed by opening the MD trajectory using the Movie Tool in the Builder. It may also be accessed by using the `measurement` method of *MDTrajectory*. For example:

```
times, average_heat_currents = md_trajectory.measurement('average_heat_current')
```

•The heat sink and heat source atoms can be selected in several different ways. The first way to is explicitly specify the indices of the atoms. The second way is to specify the names of tagged groups of atoms. The third way is to define a region of space whose atoms will be included. The third option assumes that the heat current is measured along the c-axis of the simulation cell and is defined in terms of an origin and a length. The origin is given in fractional coordinates and the length is given as a *PhysicalQuantity* with length units. This method is ideal for working with liquid systems where atoms may drift from their initial positions. See the descriptions of the arguments `heat_source` and `heat_sink` above for details.

•The momenta of the coldest particle in the `heat_source` group of atoms and the hottest particle in the `heat_sink` group are exchanged at a given interval. This exchange produces a temperature gradient, and an internal flow of energy between the two groups of atoms. Both quantities can be measured and the thermal conductance can be calculated using Fouriers law. For more details regarding the RNEMD technique, please consult Refs.[1] and[2].

•If a non-zero `update_profile_interval` is specified, a temperature profile is stored on-the-fly during the simulation. After the MD run, the profiles may be accessed using the method `temperatureProfile`.

[1] F.ăMueller-Plathe. A simple nonequilibrium molecular dynamics method for calculating the thermal conductivity. *J. Chem. Phys.*, 106(14):6082–6085, 1997. doi:10.1063/1.473271.

[2] C.ăNieto-Draghi and J.ăB. Avalos. Non-equilibrium momentum exchange algorithm for molecular dynamics simulation of heat flow in multi-component systems. *Mol. Phys.*, 101(14):2303–2307, 2003. doi:10.1080/0026897031000154338.

NonLinearCoefficientsParameters

class NonLinearCoefficientsParameters(*initial_coefficients=None*, *perform_optimization=None*, *max_force_rmse_change=None*, *max_steps=None*, *random_seed=None*, *regularization=None*, *energy_only=None*, *radial_function_order=None*, *zero_interaction_pairs=None*, *initial_coefficients_from_file=None*)

Class for storing the non-linear coefficients and parameters related to their optimization.

Parameters

- **initial_coefficients** (dict of {tuple (size 2) of `Element`: numpy-ndarray} | str | Random | FromAtomicNumbers) – The initial non-linear coefficients to use. Can either be a dict mapping the element pairs to the array with the initial coefficient values, or a filename pointing to a .mtp file to read the coefficients from. If an initial_coefficients_from_file is provided, this argument is ignored and initial coefficients are taken from the MTPParameters. Alternatively the parameter can be set to Random to use random initial values or set to FromAtomicNumbers to use the product of the element pairs atomic numbers. **Default:** Random if *perform_optimization* is True, FromAtomicNumbers otherwise.
- **perform_optimization** (*bool*) – Whether to optimize the non-linear coefficients during the fitting. **Default:** `False`
- **max_force_rmse_change** (*int*) – The convergence criterion for the change in the RMSE on the force values between subsequent steps in the optimization of the non-linear coefficients. Ignored if *perform_optimization* is False. **Default:** `0.05 * eV / Angstrom`
- **max_steps** (*int*) – The maximum number of optimization steps to take. Ignored if *perform_optimization* is False. **Default:** `100`
- **random_seed** (*int*) – The random seed used for generating the initial non-linear coefficients, if not given. **Default:** Generated automatically.
- **regularization** (*float*) – The regularization strength for the optimization. Ignored if *perform_optimization* is False. **Default:** `1.0e-2`
- **energy_only** (*bool*) – Whether to only optimize on the energy values. Ignored if *perform_optimization* is False. **Default:** `True`
- **radial_function_order** (*int*) – The order of Chebychev polynomials in the expansion of each radial function. Only used when CutoffFunctionChebyshevExpansion is selected as cutoff function. **Default:** 5
- **zero_interaction_pairs** (None | Sequence of tuples (size 2) of `PeriodicTableElement`) – If not None, the interactions between the given element pairs will be set to zero. **Default:** `None`
- **initial_coefficients_from_file** (*None | MTPParameters*) – MTPParameters object containing pre calculated non-linear coefficients. **Default:** `None`

energyOnly()

Returns
Whether to only optimize on the energy values.

Return type
float

initialCoefficients()

Returns
The initial non-linear coefficients, or None if not used.

Return type
numpy.ndarray (size N, N, M) of float | None

initialCoefficientsFromFile()

This has no corresponding instance variable

maxForceRMSEChange()

Returns
The convergence criterion for the change in the RMSE on the force values between subsequent steps in the optimization of the non-linear coefficients.

Return type
PhysicalQuantity of type force

maxForceRmseChange()

Returns
The convergence criterion for the change in the RMSE on the force values between subsequent steps in the optimization of the non-linear coefficients.

Return type
PhysicalQuantity of type force

maxSteps()

Returns
The maximum number of optimization steps to take.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

performOptimization()

Returns
Whether to optimize the non-linear coefficients during the fitting.

Return type
float

radialFunctionOrder()

Returns
The max. order of the Chebychev polynomial in the expansion of the radial functions.

Return type
int

randomSeed()

Returns
The random seed used for generating the initial non-linear coefficients, or None if they should be generated automatically.

Return type
int | None

regularization()

Returns
The regularization strength for the optimization.

Return type
float

setEnergyOnly(*energy_only*)

Method for setting the energy_only.

Parameters
energy_only (*bool*) – Whether to only optimize on the energy values.

setInitialCoefficients(*initial_coefficients*, *perform_optimization*)

Method for setting the initial coefficients.

Parameters

- **initial_coefficients** (dict of {tuple (size 2) of Element: numpy-ndarray} | str | Random | FromAtomicNumbers) – The initial non-linear coefficients to use. Can either be a dict mapping the element pairs to the array with the initial coefficient values, or a filename pointing to a .mtp file to read the coefficients from. Alternatively the parameter can be set to Random to use random initial values or set to FromAtomicNumbers to use the product of the element pairs atomic numbers.
- **perform_optimization** (*bool*) – Whether to optimize the non-linear coefficients during the fitting.

setMaxForceRmseChange(*max_force_rmse_change*)

Method for setting the max_force_rmse_change.

Parameters
max_force_rmse_change (*int*) – The convergence criterion for the change in the RMSE on the force values between subsequent steps in the optimization of the non-linear coefficients.

setMaxSteps(*max_steps*)

Method for setting the max steps.

Parameters
max_steps (*int*) – The maximum number of optimization steps to take.

setPerformOptimization(*perform_optimization*)

Method for setting the perform_optimization.

Parameters
perform_optimization (*bool*) – Whether to optimize the non-linear coefficients during the fitting.

setRadialFunctionOrder(*radial_function_order*)

Method for setting the radial function order.

Parameters
radial_function_order (*int*) – The order of Chebychev polynomials in the expansion of each radial function.

setRandomSeed(*random_seed*)

Method for setting the random seed.

Parameters
random_seed (*int*) – The random seed used for generating the initial non-linear coefficients, if not given.

setRegularization(*regularization*)

Method for setting the regularization.

Parameters
regularization (*float*) – The regularization strength for the optimization.

setZeroInteractionPairs(*zero_interaction_pairs*)

Method for setting the zero_interaction_pairs.

Parameters
zero_interaction_pairs (None | Sequence of tuples of PeriodicTableElement) – If not None, the interactions between the given element pairs will be set to zero.

uniqueString()

Return a unique string representing the state of the object.

zeroInteractionPairs()

Returns
The element pairs whose interaction should be set to zero.

Return type
list | None

Usage Examples

Set up of an example of *NonLinearCoefficientsParameters* using a specific random seed to initialize the initial non-linear coefficients:

```
# Define non-linear coefficients and enable optimization.
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
   perform_optimization=True,
   initial_coefficiants=Random,
   random_seed=12345,
)
```

Scan over different initial guesses for the non-linear coefficients:

Note: This can be automated using the function *scanOverNonLinearCoefficients*

```
# Scan over different random initial guesses.
number_of_initial_guesses = 8
all_fitting_parameters = []
for i in range(number_of_initial_guesses):
    # Set up different initial guesses for the non-linear coefficients.
    non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
      perform_optimization=True,
      random_seed=randomSeed(12345, i),
    )

    fitting_parameters = MomentTensorPotentialFittingParameters(
        basis_size=PredefinedBasisSmall,
        outer_cutoff_radii=4.0*Angstrom,
        mtp_filename='mtp_{}.mtp'.format(i+1),
        non_linear_coefficients_parameters=non_linear_coefficients_parameters,
    )
    all_fitting_parameters.append(fitting_parameters)
```

Read the initial non-linear coefficients from a .mtp file and include full energy, forces, and stress error in the optimization:

```
non_linear_coefficients_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    train_on_energy_only=False,
    initial_coefficients='mtp_parameters.mtp',
)
```

Notes

NonLinearCoefficientsParameters serves as an input argument for *MomentTensorPotentialFittingParameters*. The non-linear parameters determine the setting and optimization of the coefficients for the radial functions for each element pair. Two different kinds of radial function can be chosen: The original polynomial functions used in[1], which are orthonormalized via Chebyshev poynomials (default and recommended option), and the Chebyshev expansion as explained in[2]. These are set via the parameter `cutoff_function` in *MomentTensorPotentialFittingParameters*. Optimizing the non-linear coefficients can improve the accuracy when training a MTP to multi-element systems.

Note that different MTP basis numbers may require different number of non-linear coefficients.

The optimization of the non-linear coefficients is a local optimization which means the solution depends on the initial guess. In order to obtain a lower training and testing error it is recommended to scan over different initial guesses and use the one with the lowest training and test error, as reported in the `nlprint()` output of the *MomentTensorPotentialTraining* object.

For performance, it is recommended to train on energy only, which means that the non-linear optimization is carried out by only minimizing the energy error. Once a good fit is achieved, one can optionally run a final optimization, starting from the optimized non-linear coefficients, including the full energy, forces, and stress error by setting `train_on_energy_only=False`. This usually results in a slight additional reduction of forces and stress error.

The initial values of the non-linear coefficients can either be set randomly, read from an existing MTP parameter file, or specified explicitly. If the initial non-linear coefficients are given explicitly, they must be specified as dictionary that maps an element pair to the coefficients array. The shape of the array is $(\mu_{max},)$ for original polynomial functions and $(\mu_{max}, radial\_function\_order)$ for the Chebyshev expansion, where μ_{max} is the maximum value of the radial coefficients, used by the given MTP basis ($\mu_{max} = 4$ for `PredefinedBasisSmall` and typically $\mu_{max} = 6$ when specifying the number of basis functions explicitly).

NonLinearParticleSwarmOptimizationParameters

class NonLinearParticleSwarmOptimizationParameters(*population_size=None*, *number_of_generations_per_iteration=None*, *number_of_iterations=None*, *number_of_batches=None*, *initial_population=None*, *particle_swarm_optimization_parameters=None*, *processes_per_individual=None*, *radial_function_order=None*, *random_seed=None*, *initial_coefficients_from_file=None*)

[1] AlexanderăV Shapeev. Moment tensor potentials: A class of systematically improvable interatomic potentials. *Multiscale Modeling & Simulation*, 14(3):1153–1173, 2016.

[2] Konstantin Gubaev, EvgenyăV. Podryabinkin, GusăL.W. Hart, and AlexanderăV. Shapeev. Accelerating high-throughput searches for new alloys with active learning of interatomic potentials. *Comput. Mater. Sci.*, 156:148–156, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0927025618306372, doi:https://doi.org/10.1016/j.commatsci.2018.09.031.

Class for storing the non-linear coefficients and parameters related to their optimization using particle-swarm-optimization.

Parameters

- **population_size** (*int*) – The number of search particles in the swarm. **Default:** 32
- **number_of_generations_per_iteration** (*int*) – How many generations each swarm optimization should carry out per iteration (batch). **Default:** 20
- **number_of_iterations** (*int*) – The number of iterations (epochs) over different batches of the training data. This only needs to be increased if more than one batch is used. **Default:** 1
- **number_of_batches** (*int*) – The number of batches the training dataset is divided into. This can be used to reduce the load of the fitting for large training sets. 1 means use all training data. If set to a value greater than 1, it is recommended to increase the number_of_iterations parameter, too, to. **Default:** 1
- **initial_population** (*numpy.ndarray | str | Random | None*) – The initial values for the non-linear coefficients that the swarm should start from. Can be given as numpy array or as MTP filename. If given as filename the non-linear coefficients are read from the file. Missing coefficient values are filled with random values. If Random or None, random values are used for all coefficients.
- **particle_swarm_optimization_parameters** (*ParticleSwarmOptimizationParameters*) – The particle swarm optimization parameters set.
- **processes_per_individual** (*int | None*) – The number of processes used for each individual MTP fit. **Default:** All available processes in the process group.
- **radial_function_order** (*int*) – The order of Chebychev polynomials in the expansion of each radial function. Only used when CutoffFunctionChebyshevExpansion is selected as cutoff function. **Default:** 5
- **random_seed** (*int*) – The random seed used to control the splitting of the training data into batches.
- **initial_coefficients_from_file** (*None | MTPParameters*) – MTPParameters object containing pre calculated non-linear coefficients. **Default:** None

initialCoefficientsFromFile()

This has no corresponding instance variable

initialPopulation()

Returns
The user-specified initial values for the non-linear coefficients.

Return type
numpy.ndarray | Random

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfBatches()

Returns
The number of batches.

Return type
int

numberOfGenerationsPerIteration()

Returns
The number of generations per iteration (epoch).

Return type
int

numberOfIterations()

Returns
The number of iterations (epochs) over different training data batches.

Return type
int

particleSwarmOptimizationParameters()

Returns
The particle swarm optimization parameters set.

Return type
ParticleSwarmOptimizationParameters

populationSize()

Returns
The number of particles in the swarm.

Return type
int

processesPerIndividual()

Returns
The number of processes used for each individual MTP fit.

Return type
int

radialFunctionOrder()

Returns
The max. order of the Chebychev polynomial in the expansion of the radial functions.

Return type
int

randomSeed()

Returns
The random seed used for generating the initial non-linear coefficients, or None if they should be generated automatically.

Return type
int | None

setInitialPopulation(*initial_population*)

Parameters
initial_population (*numpy.ndarray | str | Random | None*) – The non-linear coefficients values that should be used to initialize the swarm.

setNumberOfBatches(*number_of_batches*)

Set the number of batches.

Parameters
number_of_batches (*int*) – The number of batches.

setNumberOfGenerationsPerIteration(*number_of_generations_per_iteration*)

Parameters
number_of_generations_per_iteration (*int*) – The number of generations per iteration (epoch).

setNumberOfIterations(*number_of_iterations*)

Parameters
number_of_iterations (*int*) – The number of iterations (epochs) over different training data batches.

setParticleSwarmOptimizationParameters(*particle_swarm_optimization_parameters*)

Parameters
particle_swarm_optimization_parameters (*ParticleSwarmOptimizationParameters* | None) – The particle swarm optimization parameters set.

setPopulationSize(*population_size*)

Parameters
population_size (*int*) – The number of particles in the swarm.

setProcessesPerIndividual(*processes_per_individual*)

Parameters
processes_per_individual (*int*) – The number of processes used for each individual MTP fit.

setRadialFunctionOrder(*radial_function_order*)

Method for setting the radial function order.

Parameters
radial_function_order (*int*) – The order of Chebychev polynomials in the expansion of each radial function.

setRandomSeed(*random_seed*)

Method for setting the random seed.

Parameters
random_seed (*int*) – The random seed used for generating the initial non-linear coefficients, if not given.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

A default setup of the *NonLinearParticleSwarmOptimizationParameters* object.

```
particle_swarm_optimization_parameters = ParticleSwarmOptimizationParameters(
    inertia_weight=0.8,
    local_acceleration=0.2,
    global_acceleration=0.8,
    omega=0.5,
)
```

(continues on next page)

(continued from previous page)

```
non_linear_particle_swarm_optimization_parameters =␣
↪NonLinearParticleSwarmOptimizationParameters(
    population_size=32,
    number_of_generations_per_iteration=20,
    number_of_iterations=1,
    number_of_batches=1,
    particle_swarm_optimization_parameters=particle_swarm_optimization_parameters,
)
```

This instance can then be passed into *MomentTensorPotentialFittingParameters*.

Notes

The *NonLinearParticleSwarmOptimizationParameters* object serves as input to *MomentTensorPotentialFittingParameters* to specify parameters for the particle swarm optimization (PSO) algorithm and further on to determine the non-linear coefficients in a moment tensor potential (MTP) model. The PSO algorithm is an optimization scheme which combines local and global exploitation to find an optimum solution. As indicated by the name the approach is initialized with a swarm of *particles* randomly distributed in the search space. Each particle, i, holds a vector of positions, velocities and local best positions denoted by $\vec{X}_i$, $\vec{V}_i$ and $\vec{P}_i$, respectively. The global best positions by the entire swarm is denoted $\vec{P}_g$. The particle properties then evolves to obtain best positions in search space going from increment t to $t+1$ while being affected by the local and global best positions of the swarm. The updated velocities at $t+1$ are determined according to[1],

$$\vec{V}_i(t+1) = w\vec{V}_i(t) + \alpha \mathrm{r}_1 \left[\vec{P}_i(t) - \vec{X}_i(t)\right] + \beta \mathrm{r}_2 \left[\vec{P}_g(t) - \vec{X}_i(t)\right]$$

where w is the inertia weight, α is the local acceleration coefficient and β is the global acceleration coefficient. r_1 and r_2 are random numbers. The updated positions at $t+1$ are calculates as,

$$\vec{X}_i(t+1) = \vec{X}_i(t) + \omega\vec{V}_i(t+1)$$

where ω is a velocity scaling factor.

The maximum number increments per MTP fit iteration is given by `number_of_generations_per_iteration` argument. The number of MTP fit iterations is specified by `number_of_iterations` argument. `population_size` specifies the number of particles in the swarm which corresponds to the number of coefficients. For small and medium-sized training datasets, `number_of_batches` and `number_of_iterations` can be set to 1 to train on the entire dataset. For larger datasets these values can be increased, which means the data will be divided into multiple batches and each iteration sees only a single batch. With each new iteration the training uses a different batch. This can reduce memory and speed up the fitting, especially if a subset of the training data is sufficient to optimize the non-linear parameters. At the end of each iteration a single linear fit on the entire training set using the current best set of non-linear coefficients is carried out to obtain the total error of this iteration. The parameter `processes_per_individual` specifies how many processes are used for each linear fit. This can be used to run multiple fits in parallel and balance the parallel efficiency versus memory consumption. PSO provides a more systematic optimization than scanning over different initial guesses or using local optimization via *NonLinearCoefficientsParameters* for the non-linear coefficients, and can result in a lower training error particularly for training datasets with many different elements, but it also takes longer time.

[1] Zahra Beheshti, SitiăMariyam Shamsuddin, and Sarina Sulaiman. Fusion Global-Local-Topology Particle Swarm Optimization for Global Optimization Problems. *Mathematical Problems in Engineering*, Feb 2014. URL: https://onlinelibrary.wiley.com/doi/10.1155/2014/907386, doi:https://doi.org/10.1155/2014/907386.

NonuniformGridConjugateGradientSolver

class NonuniformGridConjugateGradientSolver(*fine_grid_extension=None*, *scaling_factor=None*, *boundary_conditions=None*, *solvent_dielectric_constant=None*, *gpu_acceleration=None*)

A conjugate gradient-based Poisson solver for gated reduced dimensional systems. The solver uses a fine grid in the region determined by the atomic coordinates, and the support of the basis set. An increasingly coarser grid outside of this region is then used in the transverse directions to calculate the potential more efficiently. The solver assumes that the system has vacuum along at least one of the transverse directions. The fine grid region can be extended by specifying values for `fine_grid_extension`.

Moreover, the solver only supports orthorhombic cells.

Parameters

- **fine_grid_extension** (*PhysicalQuantity* of type length | list of *PhysicalQuantity* of type length) – An extension of the fine grid region, where the grid spacing is uniform, specified in units of length. The extension can be a single value; in that case the same value is used for both A and B directions, or a list specifying different values for A and B. The value of this parameter is subtracted from the bottom edge of the fine grid region, and added to its top edge. **Default:** 5% of the lattice vector length along each transverse direction.

- **scaling_factor** (*PhysicalQuantity* of type inverse length | list of *PhysicalQuantity* of type inverse length) – The scaling factor of the non-uniform region. The factor can be a single value; in that case the same value is used for both A and B directions, or a list specifying different values for A and B. The grid spacing in the nonuniform grid region with respect to the grid spacing in the uniform region increases approximately as:

  $\frac{\Delta x_{non-uniform}}{\Delta x_{uniform}} \approx sd + 1,$

  where s is the scaling factor, d the distance from the fine grid region and $\Delta x_{uniform}$ the uniform grid spacing as determined by the parameter *density_mesh_cutoff* in *NumericalAccuracyParameters*. In the limit $s \to 0$ the uniform grid spacing is recovered everywhere. Note that a too large scaling factor might severely reduce the quality of the results. **Default:** `[0.05, 0.05] * Angstrom**(-1)`

- **boundary_conditions** (list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*) – A list o lists of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:** - *BulkConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2]
```

– *SurfaceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition(), NeumannBoundaryCondition()]]
```

– *DeviceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition()] * 2]
```

- **solvent_dielectric_constant** (*float*) – The dielectric constant of the media surrounding the configuration. Must be a number larger than 0. **Default:** `1.0`

- **gpu_acceleration** (Enabled | Disabled | Automatic) – Whether to use GPU acceleration. If set to Automatic, the solver will use the GPU if available. **Default:** Automatic

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

fineGridExtension()

Returns
The extension of the fine grid region along the transverse directions, in units of length. If None, the extension will be determined automatically from the configurations lattice.

Return type
list of *PhysicalQuantity* of type length | None

gpuAcceleration()

Returns
The GPU acceleration flag.

Return type
Enabled | Disabled | Automatic

nlinfo()

Returns
The nlinfo.

Return type
dict

scalingFactor()

Returns
The scaling factor of the non-uniform region of the grid along the transverse directions.

Return type
list of *PhysicalQuantity* of type inverse length

solventDielectricConstant()

Returns
The solvent dielectric constant.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

NormConservingPseudoPotential

class NormConservingPseudoPotential(*file_name*, *principal=None*, *momenta=None*, *occupation=None*, *local_potential_cutoff_threshold=None*, *local_potential_cutoff_radius=None*)

Create a norm-conserving pseudopotential from file data.

Parameters

- **file_name** (*str*) – Name of the file containing the pseudopotential data.
- **principal** – The principal quantum numbers of the valence states. If `None` the information will be read from the pseudopotential file.
- **momenta** – The angular momenta of the valence states. If `None` the information will be read from the pseudopotential file.
- **occupation** – The occupations of the valence states. If `None` the information will be read from the pseudopotential file.
- **local_potential_cutoff_threshold** (*PhysicalQuantity*) – When the difference between the local potential and the Coulomb potential drop below this threshold value, the local potential is set equal to the exact Coulomb potential. Must be have units of energy. **Default:** 0.0 Hartree
- **local_potential_cutoff_radius** (*PhysicalQuantity*) – The local potential is set equal the Coulomb potential beyond this radius. If `False` the local potential will not be modified. Must otherwise be a *PhysicalQuantity* with units of length. **Default:** `False`

Type
principal: list

Type
momenta: list

Type
occupation: list

element()

Get the element described by the pseudopotential.

Returns
The element.

Return type
PeriodicTableElement

exchangeCorrelation()

Returns
The exchange-correlation of the pseudo-potential.

Return type
ExchangeCorrelation

fileName()

Get the file name.

Returns
The user given file name.

Return type
str

load()

Load the pseudopotential from file.

localPotentialCutoffRadius()

Get the maximum local potential to Coulomb potential cutoff radius.

Returns
The maximum cutoff radius in units of length or False if no maximum cutoff radius should be enforced.

Return type
PhysicalQuantity | False

localPotentialCutoffThreshold()

Get the local potential to Coulomb potential cutoff threshold.

Returns
The potential difference threshold in units of energy.

Return type
PhysicalQuantity

momenta()

Get the angular momenta of the pseudo wave functions.

Returns
A list of angular momenta.

Return type
list

nlinfo()

Returns
The nlinfo.

Return type
dict

occupation()

Get the occupations of the pseudo wave functions.

Returns
A list of occupations.

Return type
list

principal()

Get the principal quantum numbers of the pseudo wave functions.

Returns
A list of principal quantum numbers.

Return type
list

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a norm-conserving pseudo-potential for hydrogen, specified in the zipped file `H.LDAPZ.zip`:

```
pseudopotential = NormConservingPseudoPotential('normconserving/H.LDAPZ.zip')
```

To see the contents of `H.LDAPZ.zip`, go to the directory where QuantumATK is installed (`install_dir`) and unpack the pseudo-potential file:

```
$ cd install_dir/share/pseudopotentials/normconserving
$ unzip H.LDAPZ.zip
```

This will produce the file `H.LDAPZ.upf`. You can also specify the unzipped file:

```
pseudopotential = NormConservingPseudoPotential('normconserving/H.LDAPZ.upf')
```

Notes

ATK uses the Unified Pseudopotential Format (UPF) defined by the PWscf consortium. From their website it is possible to download tools that can convert several different kinds of pseudopotentials into the UPF format.

ATK comes with a built-in set of pseudopotentials, found in the directory `install_dir/share/pseudopotentials`, where `install_dir` is the name of the directory where QuantumATK was installed.

NormalizeByArea

class NormalizeByArea(*A=None*, *B=None*, *C=None*, *area=None*)

Normalize by area of specified cell face.

Parameters

- **A** (*bool*) – Use the A-vector for the face.
- **B** (*bool*) – Use the B-vector for the face.
- **C** (*bool*) – Use the C-vector for the face.
- **area** (*PhysicalQuantity* of type area) – The area to use. **Default:** Extracted from the configuration, based on the specified face.

NormalizeByLength

class NormalizeByLength(*A=None*, *B=None*, *C=None*, *length=None*)

Normalize by length of specified cell vector.

Parameters

- **A** (*bool*) – Use the A-vector of the cell.
- **B** (*bool*) – Use the B-vector of the cell.
- **C** (*bool*) – Use the C-vector of the cell.
- **length** (*PhysicalQuantity* of type length) – The length to use. **Default:** Extracted from the configuration, based on the specified vector.

NormalizeByNumberOfAtoms

class NormalizeByNumberOfAtoms(*atom_count=None*)

Normalize by number of atoms.

Parameters
atom_count (*int*) – The atom count to use. **Default:** Extracted from the configuration.

NormalizeByVolume

class NormalizeByVolume(*volume=None*)

Normalize by cell volume.

Parameters
volume (*PhysicalQuantity* of type volume.) – The volume to use. **Default:** Extracted from the configuration.

NudgedElasticBand

class NudgedElasticBand(*configuration_list*, *image_distance=None*, *generate_images=None*, *number_of_intermediate_images=None*, *wrap_pbc=None*, *endpoint_energies=None*, *endpoint_forces=None*, *endpoint_stresses=None*, *use_min_image_convention=None*, *remove_external_degrees_of_freedom=None*, *interpolation_algorithm=None*)

Nudged Elastic Band as a series of configurations.

Parameters

- **configuration_list** (list of *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – A list configurations that must be included in the series of configurations. Must hold at least two configurations. Additionally, all configurations must be of the same type, with the same elements, and with the same spatial regions.
- **image_distance** (*PhysicalQuantity* of type length) – Distance between the coordinates of successive configurations in the series of configurations if auto-generation is enabled. **Default:** 0.5*Angstrom.
- **generate_images** (*bool*) – Auto-generate intermediate images between initial configurations, using the specified `interpolation_algorithm`. No distance between successive images is greater than the `image_distance` parameter. Set `generate_images` to *False* to keep the images exactly as supplied in `configuration_list`. Specifying the interpolation algorithm directly through this parameter is deprecated and will be removed in the future. **Default:** True.
- **number_of_intermediate_images** (*int*) – The number of images between the endpoints. Mutually exclusive with image_distance. Requires configuration_list to contain exactly two configurations.
- **wrap_pbc** (*bool*) – For some plugins (e.g. VASP) the atoms have been wrapped back inside the box during the simulation. This means that the distance between images must be PBC aware. Setting this argument to *True* takes the wrapping into account. **Default:** False.
- **endpoint_energies** (sequence(2) of *PhysicalQuantity* of type energy) – The energies of the NEB endpoints may be supplied if they are known. This speeds up calculations. **Default:** Energies not known.
- **endpoint_forces** (sequence(2) of *PhysicalQuantity* of type force) – The forces of the NEB endpoints may be supplied if they are known. This speeds up calculations. **Default:** Forces not known.
- **endpoint_stresses** (sequence(2) of *PhysicalQuantity* of type stress) – The stresses of the NEB endpoints may be supplied if they are known. This speeds up calculations. **Default:** Stress not known.
- **use_min_image_convention** (*bool*) – Whether to compute the minimum image convention distance between the initial and final configurations. **Default:** False.

- **remove_external_degrees_of_freedom** (*bool*) – Whether to remove external degrees of freedom (rotation and translation) from the generated images. Only applicable to non-periodic systems (MoleculeConfiguration). **Default:** False.
- **interpolation_algorithm** (*LinearInterpolation* | ImageDependentPairPotential) – The interpolation algorithm to use when generating images. Note that *HalgrenLipscomb* and *ImageDependentPairPotential* add a small perturbation to the images to avoid numerical instabilities. Variable-cell NEBs only work with *LinearInterpolation*. **Default:** *LinearInterpolation*.

addTags(*tags*, *indices=None*)

Add a set of tags, to atoms matching a collection of indices, for each NEB image.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms.
- **indices** (*list | int | None*) – The list of indices to match atoms against. If None, then all atoms are matched. **Default:** All indices.

additionalMinima()

Method to determine additional minima besides the endpoints.

Returns
A list containing the indices of images that are minima on the reaction coordinate excluding the endpoints.

Return type
list of type int | None

bonds()

Get the list of bond connections to can be used for bonded potentials in ATK-ForceField. The same bond list applies to all images

Returns
An array with the the two atom indices for each bond along with the vector which periodic images this bond connects.

Return type
array

calculator()

Returns
The calculator attached to the first image.

Return type
Calculator

calculators()

Returns
A list of all attached calculators.

Return type
list of *Calculator*

cartesianCoordinates()

Returns
The coordinates of the all the images in the NEB. A MxNx3 array is returned where M is the number of images and N is the number of atoms per image.

Return type
PhysicalQuantity of type length

compare(*other*, *the_type=True*, *elements=True*, *ghosts=True*, *cell=True*, *positions=True*, *regions=True*, *tags=True*, *velocities=False*, *bonds=False*, *throws=False*, *msg='Compared NEB configurations differ.\n'*)

A method to compare two NEB configurations by comparing their images. It is possible to turn some comparisons off. If the comparison passes all the activated checks, the method returns True, otherwise False

Parameters

- **other** (NudgedElasticBand) – The other configuration to compare with.
- **the_type** (*bool*) – Whether to compare the type. **Default:** True.
- **elements** (*bool*) – Whether to compare the elements. **Default:** True.
- **ghosts** (*bool*) – Whether to compare the ghosts. **Default:** True.
- **cell** (*bool*) – Whether the compare the cells. **Default:** True.
- **positions** (*bool*) – Whether to compare the positions. **Default:** True.
- **regions** (*bool*) – Whether to compare the regions. **Default:** True.
- **tags** (*bool*) – Whether to compare the tags. **Default:** True.
- **velocities** (*bool*) – Whether to compare the velocities. **Default:** False.
- **bonds** (*bool*) – Whether to compare the bonds. **Default:** False.
- **throws** (*bool*) – Whether an exception should be raised on differences. **Default:** False.
- **msg** (*str*) – The message to use when throwing exceptions. **Default:** `Compared configurations differ.\n`.

Returns
True, if configurations are identical.

Return type
bool

configurationCopy(*new_coordinates=None*, *copy_calculator=False*)

Copy the NEB, with new coordinates.

Parameters

- **new_coordinates** (*PhysicalQuantity* of length units | None) – The new coordinates to set, given as a MxNx3 array where M is the number of images and N is the number of atoms in each image. **Default:** Leave coordinates unchanged.
- **copy_calculator** (*bool*) – Copy the calculator and set the initial_state/initial_spin appropriately. **Default:** False

Returns
A copy of the current NEB

Return type
NudgedElasticBand

configurationType()

Returns
The type of configurations stored in the nudged elastic band.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

constraintCorrections()

Returns
a dictionary containing the correction to energy, forces and stresses, if any, introduced by constraints during an NEB optimization.

Return type
dict

copy()

Returns
A copy of the current configuration.

Return type
NudgedElasticBand

deleteAtoms(**args*, ***kwargs*)

Remove atoms from a NEB.

All arguments and keyword arguments are passed on.

dielectricRegions()

Returns
The dielectric regions in the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

elements()

Returns
The elements in the nudged elastic band.

Return type
list of *PeriodicTableElement*

energies(*include_constraints=True*)

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output energies should include constraint corrections or not.

Returns
The energies of the images. None will be returned if the energies are not available.

Return type
PhysicalQuantity of type energy | None

enthalpies(*include_constraints=True*)

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output enthalpies should include constraint corrections or not.

Returns
The enthalpies of the images. None will be returned if the enthalpies are not available.

Return type
PhysicalQuantity of type energy | None

forces(*include_constraints=True*)

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output forces should include constraint corrections or not.

Returns
The forces of the images. If there are M images and N atoms a MxNx3 array of forces will be returned. None will be returned if the forces are not available.

Return type
PhysicalQuantity of type force | None

forwardBarrier(*include_constraints=True*)

Calculate the forward barrier of the NEB using the maximum energy image.

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output energies should include constraint corrections or not.

Returns
The forward barrier.

Return type
PhysicalQuantity of type energy

generateImages()

Returns
If the intermediate images should be auto-generated.

Return type
bool

ghostAtoms()

Returns
The list of ghost atoms.

Return type
list of ints

hermiteSpline(*x*, *y*, *npoints=20*)

Calculates a piecewise cubic Hermite spline.

Parameters

- **x** (*array*) – The x coordinates.
- **y** (*array*) – The values for each x coordinate.
- **npoints** (*int*) – The number of interpolated points between each input coordinate.

Returns
The new grid of x coordinates and the value of the interpolant.

Return type
2-tuple

image(*index*)

Parameters
index (*int*) – The index of the image.

Returns
The configuration at the specified index.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

imageDistance()

Returns
The image distance.

Return type
PhysicalQuantity of type length

images()

Returns
All configurations in the nudged elastic band.

Return type
list of *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

indicesFromTags(*tags*)
List the indices associated with a given collection of tags.

Parameters
tags (*list | str*) – A list of tags for which all matching indices should be extracted.

Returns
The list of indices corresponding to the specified tag name(s).

Return type
list of ints

interpolationAlgorithm()

Returns
The interpolation algorithm used.

Return type
InterpolationAlgorithm | None

maxEnergyImageIndex()

Returns
The index of the image with the highest energy.

Return type
int

metallicRegions()

Returns
The metallic regions for the configuration.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The NudgedElasticBand information.

Return type
dict

numberOf()

Returns
The image distance.

Return type
PhysicalQuantity of type length

potentialEnergies(*include_constraints=True*)

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output energies should include constraint corrections or not.

Returns
The energies of the images. None will be returned if the energies are not available.

Return type
PhysicalQuantity of type energy | None

reactionCoordinates()

Returns
The distance from the first image along the series of images.

Return type
PhysicalQuantity of type length

refine(*first*, *last*, *image_distance=None*, *number_of_intermediate_images=None*, *interpolation_algorithm=None*)

Refine a section of the reaction path. Any images between the specified starting and ending images are dropped in favor of the new refinement images.

Parameters

- **first** (*int*) – The starting image index in the reaction sub-path being refined.
- **last** (*int*) – The ending image index in the reaction sub-path being refined.
- **image_distance** (*PhysicalQuantity* of type length) – The maximum distance between successive configurations. **Default:** 0.5*Angstrom.
- **number_of_intermediate_images** (*int*) – The number of images between the first and last. Mutually exclusive with image_distance.
- **interpolation_algorithm** (*LinearInterpolation* | *ImageDependentPairPotential* | *HalgrenLipscomb*) – The type of interpolation to use. **Default:** *LinearInterpolation*.

reinterpolate(*interpolation_algorithm*, *constraints=None*, *progress_bar=None*)

Re-interpolate the current NEB using the given method.

Parameters

- **interpolation_algorithm** (*LinearInterpolation* | ImageDependentPairPotential) – The interpolation algorithm to use.
- **constraints** (*list of ints*) – List of atom indices that are kept fixed during interpolation. **Default:** [].
- **progress_bar** (*Class with a 'setValue' method*) – The progress to update while interpolating.

removeTags(*tags=None*, *indices=None*, *purge=False*)

Remove a set of tags from atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms. **Default:** All tags.
- **indices** (*list | int*) – The list of indices to match atoms against. **Default:** All indices.
- **purge** (*bool*) – When removing tags from the configuration, delete the tag completely when not associated with any atoms anymore. **Default:** False.

resetConstraintCorrections()

Reset the dictionary containing constraint corrections.

resetOptimizationState()

Resets the saved optimization state.

reverseBarrier(*include_constraints=True*)

Calculate the reverse barrier of the NEB using the maximum energy image.

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output energies should include constraint corrections or not.

Returns
The reverse barrier.

Return type
PhysicalQuantity of type energy

saveOptimizationState(*step*, *optimizer*, *previous_max_image*)

Saves the current optimization state. This is used during the NEB optimization process and its restart functionality.

Parameters

- **step** (*int*) – The iteration number in the NEB optimization.
- **optimizer** (*LBFGSOptimizer | FIREOptimizer*) – The optimizer method object from the NEB optimization.
- **previous_max_image** (*int*) – Maximum image from the previous optimization step.
- **previous_coordinates** (*array*) – Coordinates of each NEB image in the previous optimization step.

setBravaisLattice(**args*, ***kwargs*)

Method for setting the Bravais lattice. Only nudged elastic bands containing bulk configurations are affected.

Parameters

- **bravais_lattice** (BravaisLattice) – The new lattice.
- **image_index** (*int*) – Image index of the BulkConfiguration to be modified. **Default:** all images
- **conserve_coordinates** (*Cartesian | Fractional*) – The type of coordinates to conserve. **Default:** Cartesian

setCalculator(*calculator*, *initial_state=None*, *initial_spin=None*)

Attach a calculator to the *NudgedElasticBand*.

Parameters

- **calculator** (*Calculator* | None) – The calculator that should be attached to the configuration.
- **initial_state** (*MoleculeConfiguration | BulkConfiguration | DeviceConfiguration | SurfaceConfiguration* with an attached calculator | None) – The initial state to be used for this configuration. **Default:** No initial state.

- **initial_spin** (*InitialSpin* | None) – The initial spin to be used for this configuration. **Default:** InitialSpin(), except when used with a calculator of type *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*. In this case InitialSpin([0., 0., ...]) is used, i.e. the scaled spins for each atom are set to 0.

setCartesianCoordinates(*cartesian_coordinates*, *skip_checks=False*)

Set the Cartesian coordinates for the atoms in each image.

Parameters

- **cartesian_coordinates** (*PhysicalQuantity* of type length) – The new coordinates of the atoms in each image.
- **skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False

setDielectricRegions(**args*, ***kwargs*)

Set the dielectric regions for the configuration.

Parameters
dielectric_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

setMetallicRegions(**args*, ***kwargs*)

Set the metallic regions for the configuration.

Parameters
metallic_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shiftAtoms(**args*, ***kwargs*)

Shift atoms.

All arguments and keyword arguments are passed on.

split(*image_indices=None*)

Method to split the present NEB into multiple smaller NEBs.

Parameters
image_indices (*Sequence of type int*) – The list of images to split at. Allowed are only values in the set of intermediate images (i.e, excluding the endpoints). The sequence is automatically sorted and duplicates are removed. The specified images become the endpoints of the new NEBs. **Default:** Automatically determined by the additional minima found between the endpoints

Returns
A list of split NEBs.

Return type
Sequence of type *NudgedElasticBand*

startSeriesCalculation()

Start series calculation, in which the calculator is not copied with __call__.

stopSeriesCalculation()

Stop series calculation, and re-enable copying of the calculator with __call__.

stresses(*include_constraints=True*)

Parameters
include_constraints (*bool*) – If this object was generated by an NEB optimization with constraints, determine whether the output stresses should include constraint corrections or not.

Returns
The residual stress of the images, i.e. the actual stress minus the target stress is returned. If there are M images a Mx3x3 array of stress will be returned. None will be returned if the stress is not (has not been) calculated (e.g. in case of a molecular or fixed cell NEB).

Return type
PhysicalQuantity of type stress | None

symbols()

Returns
The element symbols of the configuration.

Return type
list of str

tags(*indices=None*)

List the tags associated with a given collection of indices. The list returned is the set union of tags associated with the given indices. If no collection of indices is provided, then all tags on the configuration are returned.

Parameters
indices (*list | int*) – The indices to check. **Default:** All indices.

Returns
The set union of tags present on the provided indices.

Return type
set

tangents()

Returns
The normalized NEB tangent directions. This is a (num_images, num_atoms + 3, 3) array.

Return type
ndarray of type float

targetPressure()

Returns
The target pressure if set.

Return type
PhysicalQuantity of type pressure | None

totalDistance()

Returns
The total distance from first to last image.

Return type
PhysicalQuantity of type length

uniqueElements()

Returns
The unique elements contained in the nudged elastic band.

Return type
list of *PeriodicTableElement*

uniqueString()

Return a unique string representing the state of the object.

Returns
The string.

Return type
str

update(*log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*)

Update all images using the calculator stored on the class.

The number of processes used to calculate the energy and forces on each image is determined by the ParallelParameters object that has been set on the attached calculator. By default, the update is parallelized over images and then over the per-image calculation.

The output from the calculation for each image will be logged to a separate file. Each files name is set by log_filename_prefix with the image ID as a suffix. If the log files already exist they will be appended to.

Parameters
log_filename_prefix (*Automatic | str | None*) – The logging output from each image will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** Automatic.

variableCell()

Returns
Whether this is a VC-NEB Configuration.

Return type
Bool

Usage Examples

Setup a nudged elastic band calculation for conversion of ethane to ethene, i.e.

$$C_2H_6 \rightarrow C_2H_4 + H_2$$

```
# Find the reaction path
optimized_neb = OptimizeNudgedElasticBand(
        neb_configuration,
        max_forces=0.05*eV/Ang)
```

neb_c6h6.py

Calculate the *TotalEnergy* for all images in a nudged elastic band configuration:

```
# Define elements.
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates.
c0 = [[  0.00,   0.00,   0.12],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom
```

(continues on next page)

(continued from previous page)

```
c1 = [[  0.00,   0.00,   1.20],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

# Set up a list of configurations.
configuration_list = [ MoleculeConfiguration(elements,c) for c in [c0,c1] ]

# Create the NudgedElasticBand object.
neb = NudgedElasticBand(configuration_list)

# Set a calculator on the configurations in the NEB - for analysis.
neb.setCalculator(LCAOCalculator())

# Create a Trajectory with a TotalEnergy analysis carried out on each configuration.
t = MakeTrajectory(neb, TotalEnergy)
```

maketrajectory2.py

Variable Cell NEB

A variable cell NEB is constructed when using bulk configurations that have different unit cells. When used with *OptimizeNudgedElasticBand()* both the atomic and lattice degrees of freedom are optimized. Great care must be taken when constructing the initial and final images. Just as in the regular NEB algorithm, the atom indices in the initial and final structure must match. Atom #1 in the initial structure will be moved to atom #1 in the final structure. This means that you cannot simply take two different crystal structures with the same number of atoms and expect the resulting pathway to be correct.

See the *Variable Cell NEB* section on the OptimizeNudgedElasticBand page for more information on variable cell NEB calculations.

Notes

The *NudgedElasticBand* configuration sets up a reaction path for a nudged elastic band calculation[1,2].

When `generate_images` is `True` the automatically generated images are created using a linear interpolation between the endpoints. Periodic boundary conditions are not considered in this interpolation, i.e. the interpolated pathway will not follow the minimum image convention.

•See also *OptimizeNudgedElasticBand()*

[1] G.ăHenkelman, B.ăP. Uberuaga, and H.ăJonsson. A climbing image nudged elastic band method for finding saddle points and minimum energy paths. *J. Chem. Phys.*, 2000.

[2] Daniel Sheppard, Penghao Xiao, William Chemelewski, DuaneăD. Johnson, and Graeme Henkelman. A generalized solid-state nudged elastic band method. *The Journal of Chemical Physics*, 136(7):074103, 2012. doi:10.1063/1.3684549.

NumericalAccuracyParameters

class NumericalAccuracyParameters(*density_mesh_cutoff=None*, *k_point_sampling=None*, *radial_step_size=None*, *density_cutoff=None*, *interaction_max_range=None*, *number_of_reciprocal_points=None*, *reciprocal_energy_cutoff=None*, *bands_per_electron=None*, *occupation_method=None*, *exx_grid_cutoff=None*, *exact_exchange_parameters=None*, *paw_grid_tolerance=None*, *compensation_charge_mesh_cutoff=None*, *grid_mesh_cutoff=None*, *electron_temperature=None*)

Class for representing the parameters for setting the numerical accuracy of a calculation.

Some parameter defaults are specific for each calculator, see *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *LCAOCalculator*, *PlaneWaveCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*, or *DeviceLCAOCalculator*.

Parameters

- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.
- **k_point_sampling** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling in reciprocal space given either as a grid (specifying the three Monkhorst-Pack indices or a *MonkhorstPackGrid* object), or a density (specifying a *MonkhorstPackGrid* object). **Default:** Specific for each calculator.
- **radial_step_size** (*PhysicalQuantity* of type length) – The maximum sampling step size in all the radial grids. Must be positive. **Default:** Specific for each calculator.
- **density_cutoff** (*float*) – The density cutoff determines the limit where a density is considered to to be zero. Smaller values therefore leads to longer ranges and less sparsity of the models. Must be positive. **Default:** `1.0e-6`
- **interaction_max_range** (*PhysicalQuantity* of type length) – The maximum allowed interaction distance between two orbitals. **Default:** Specific for each calculator.
- **number_of_reciprocal_points** (*int*) – The number of reciprocal points used for evaluating two-center integrals. Must be larger than 1. **Default:** `1024`
- **reciprocal_energy_cutoff** (*PhysicalQuantity* of type energy) – The energy cutoff in reciprocal space used for evaluating of the two-center integrals. Must be positive. **Default:** `1250 * Hartree`
- **bands_per_electron** (*float*) – The number of bands per electron. The number must be 1.0 or larger. Only used by the *PlaneWaveCalculator*. The `bands_per_electron` determines how many bands are included in the *PlaneWaveCalculator* scf loop for calculating the electron density. A value of 1.0 implies that only occupied states are included, which would only be correct at zero temperature. The default value of 1.2 means that if there are 10 electrons in a configuration there will be 5 occupied bands (assuming spin degeneracy) and one unoccupied band included in the calculation. In case of slow or non-converging calculations, increasing the `bands_per_electron` can in some cases help convergence. For *Bandstructure* calculations, the default number of bands included for a *PlaneWaveCalculator* is also determined by the `bands_per_electron`. **Default:** `1.2`
- **occupation_method** (*FermiDirac* | *GaussianSmearing* | *MethfesselPaxton* | *ColdSmearing*) – The method to calculate state occupations. **Default:** Specific for each calculator.
- **exx_grid_cutoff** (*PhysicalQuantity* | *GridSampling* | *OptimizedFFTGridSampling*) – The energy cutoff/grid sampling that determines the grid size used for representing the the local exact exchange potential in plane-wave hybrid functional calculations. For lossless calculations the grid sizes should be twice that needed to represent the wave functions. However, one may use a smaller grid for faster calculations. **Default:** The same value as used for density_mesh_cutoff

- **exact_exchange_parameters** (*ExactExchangeParameters*) – The parameters that control the accuracy of exact exchange.
- **paw_grid_tolerance** (*float*) – When using the real space grid for PAW, this parameters determines below what value the radial functions are considered to be zero when gridded.

bandsPerElectron()

Returns
The bands per electron.

Return type
float

compensationChargeMeshCutoff()

Deprecated: Use the general densityMeshCutoff

Returns
The energy cutoff/grid sampling used for the compensation charge.

Return type
PhysicalQuantity | *GridSampling* | *OptimizedFFTGridSampling*

densityCutoff()

Returns
The density cutoff.

Return type
float

densityMeshCutoff()

Returns
The density mesh cutoff.

Return type
PhysicalQuantity of type energy | *GridSampling*

electronTemperature()

Deprecated: Use occupationMethod().broadening() instead.

Returns
The electron temperature.

Return type
PhysicalQuantity of type energy

exactExchangeParameters()

Returns
parameters for setting the numerical accuracy of an exact exchange calculation.

Return type
ExactExchangeParameters

exxGridCutoff()

Returns
The energy cutoff/grid sampling used for the exact exchange potential.

Return type
PhysicalQuantity | *GridSampling* | *OptimizedFFTGridSampling*

gridMeshCutoff()

Deprecated: Use densityMeshCutoff instead.

Deprecated: Use `densityMeshCutoff()` instead.

Returns
The density mesh cutoff.

Return type
PhysicalQuantity of type energy | *GridSampling*

interactionMaxRange()

Returns
The interaction max range.

Return type
PhysicalQuantity of type length

kPointSampling()

Returns
The k-point sampling.

Return type
MonkhorstPackGrid | *KpointDensity*

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfReciprocalPoints()

Returns
The number of reciprocal points used in two-center integration.

Return type
int

occupationMethod()

Returns
The occupation method.

Return type
FermiDirac | *GaussianSmearing* | *MethfesselPaxton* | *ColdSmearing*

pawGridTolerance()

Returns
When using the real space grid for PAW, this parameters determines below what value the radial functions are considered to be zero when gridded.

Return type
float

radialStepSize()

Returns
The radial grid sampling.

Return type
PhysicalQuantity of type length

reciprocalEnergyCutoff()

Returns
The reciprocal cutoff energy.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define the k-point sampling and real space grid mesh-cutoff.

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=12.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(2, 1, 1),
    electron_temperature = 200*Kelvin
    )

calculator = HuckelCalculator(
    iteration_control_parameters=iteration_control_parameters,
    )
```

Specify the electron temperature in units of eV instead of Kelvin

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    electron_temperature=0.02 * electronVolt/boltzmann_constant
    )
```

Notes

•The distance between the points in the real space grid, Δx, is related to the `density_mesh_cutoff`, E^{grid}, through

$$\Delta x = \frac{\pi\hbar}{\sqrt{2mE^{grid}}}.$$

In atomic units $m = \hbar = 1$, thus for energies in Hartree and distances in Bohr, $\Delta x = \pi/\sqrt{2E^{\mathrm{grid}}}$.

•When setting `interaction_max_range` some matrix elements are set to zero. For very long ranged basis sets this can make the overlap matrix ill defined at certain k-points (i.e. it is not positive definite), in such cases the matrix diagonalization routine will give a segmentation fault. The cure is to change the `interaction_max_range`, i.e. either make it very large to include all long range elements, or make it small so no long range elements are included.

NumericalOrbital

class NumericalOrbital(*angular_momentum*, *radial_grid*, *function_values*, *radial_step_size=PhysicalQuantity(0.01, Bohr)*)

Constructor for a NumericalOrbital.

Parameters

- **angular_momentum** (*int*) – The Azimuthal Quantum Number (*l*) of the orbital.
- **radial_grid** (list of *PhysicalQuantity* type length.) – The radial values of the function
- **function_values** (*list of floats*) – The function values of the basis function
- **radial_step_size** (*PhysicalQuantity* of type length.) – The radial step size determining the distance between grid points on the linear radial grid. **Default:** `0.01 * Bohr`

angularMomentum()

Returns
The angular momentum.

Return type
int

functionValues()

Returns
The radial values of the function

Return type
list of floats

nlinfo()

Returns
The nlinfo.

Return type
dict

radialGrid()

Returns
The radial grid values.

Return type
list of *PhysicalQuantity* type length.

radialStepSize()

Returns
The radial step size determining the distance between grid points on the linear radial grid.

Return type
PhysicalQuantity of type length.

uniqueString()

Return a unique string representing the state of the object.

OPLSMinPotentialBuilder

class OPLSMinPotentialBuilder(*use_bonds=None*, *include_bond_stretching=None*, *include_bond_angles=None*, *include_torsions=None*, *include_inversions=None*, *include_lennard_jones=None*, *include_electrostatic=None*, *lennard_jones_cutoff=None*, *lennard_jones_smoothing_length=None*, *electrostatic_pairwise_cutoff=None*, *electrostatic_smoothing_length=None*, *electrostatic_spme_cutoff=None*, *electrostatic_accuracy=None*, *electrostatic_scale=None*, *use_electrostatic_direct_sum=None*, *ignore_missing_types=None*)

This object builds a OPLS-Min potential using TremoloX.

Parameters

- **use_bonds** (*bool*) – Controls if any bonding (stretching, angle, torsion and inversion) terms are to be included. **Default:** True.
- **include_bond_stretching** (*bool*) – Controls if bonding stretching terms are included. **Default:** use_bonds.
- **include_bond_angles** (*bool*) – Controls if angle bending terms are included. **Default:** use_bonds.
- **include_torsions** (*bool*) – Controls if bond torsion terms are included. **Default:** use_bonds.
- **include_inversions** (*bool*) – Controls if bond inversion torsion terms are included. **Default:** use_bonds.
- **include_lennard_jones** (*bool*) – Controls if Lennard-Jones terms are included. **Default:** use_bonds.
- **include_electrostatic** (*bool*) – Controls if electrostatic terms are included. **Default:** use_bonds.
- **lennard_jones_cutoff** (*PhysicalQuantity* of type distance) – Distance beyond which the Lennard-Jones terms are truncated. **Default:** 10.0 * Angstrom.
- **lennard_jones_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the Lennard-Jones terms are brought to zero at the cutoff. **Default:** 2.0 * Angstrom.
- **electrostatic_pairwise_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in pairwise summation. **Default:** 12 * Angstrom
- **electrostatic_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the electrostatic terms are brought to zero at the cutoff when using the direct electrostatic sum. **Default:** 2.0 * Angstrom
- **electrostatic_spme_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in SPME summation. **Default:** 7.5 * Angstrom
- **electrostatic_accuracy** (*float*) – Sets the relative accuracy of the SPME summation. **Default:** 0.0001
- **electrostatic_scale** (*float*) – Set the relative strength of the electrostatic terms. **Default:** 1.0
- **use_electrostatic_direct_sum** (*bool*) – Use the direct Coulomb sum to calculate the electrostatic terms in MoleculeConfigurations. **Default:** False
- **ignore_missing_types** (*bool*) – Whether or not molecules missing OPLS types are ignored. **Default:** False

assignAtomTypes(*configuration*, *search_types=None*, *overwrite=False*)

Automatically assign OPLS atom tags to a configuration.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration on which to assign tags.
- **search_types** (*list of str | None*) – List of types to assign onto the configuration. If specified only these types will be assigned, with OPLS_UNKNOWN assigned to atoms that dont match any of the given types. The default argument None means that all available types are assigned. **Default:** None.

•**overwrite** (*bool*) – Whether or not to assign new types if the configuration already has a valid set of types. **Default:** False

checkAtomTypes(*configuration*, *add_missing_terms=False*)

Function that tests that each atom has only one valid OPLS type.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration that is being tested.

•**add_missing_terms** (*bool*) – Whether or not to add terms for missing OPLS types. **Default:** False

createCalculator(*configuration*, *check_types=True*, *verbose=False*, *atomic_charges=None*, *charge_averaging=None*)

Create and return a calculator with the OPLS-Min potential.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration the potential is to be used with.

•**check_types** (*bool*) – Flag as to whether or not the correctness of the types are checked. **Default:** True

•**verbose** (*bool*) – Whether or not to print the terms assigned in the potential set. **Default:** False

•**atomic_charges** (*PhysicalQuantity of type charge*) – Custom atomic charges to be used in the potential.

•**charge_averaging** (*NLFlag | None*) – The type of averaging of charges to be used, if any. Accepted values are None, AtomConnectivity to average over atoms bonded to the same elements and AtomTypes to average over atoms of the same forcefield type. **Default:** None

Returns
The calculator containing the potential.

Return type
TremoloXCalculator

electrostaticAccuracy()

Returns
The relative accuracy of the electrostatic summation for SPME.

Return type
float

electrostaticPairwiseCutoff()

Returns
The cutoff used in pairwise electrostatic summation.

Return type
PhysicalQuantity of type length

electrostaticScale()

Returns
Scaling factor for the electrostatic terms.

Return type
float

electrostaticSmoothingLength()

Returns
Distance over which electrostatic terms are splined when using direct summation.

Return type
PhysicalQuantity of type length

`electrostaticSpmeCutoff()`

Returns
The cutoff used in SPME electrostatic summation.

Return type
PhysicalQuantity of type length

`ignoreMissingTypes()`

Returns
Whether or not missing types are ignored on molecules.

Return type
bool

`includeBondAngles()`

Returns
Whether or not bond angle terms are included in the potential set.

Return type
bool

`includeBondStretching()`

Returns
Whether or not bond stretching terms are included in the potential set.

Return type
bool

`includeElectrostatic()`

Returns
Whether or not electrostatic terms are included in the potential set.

Return type
bool

`includeInversions()`

Returns
Whether or not bond inversion terms are included in the potential set.

Return type
bool

`includeLennardJones()`

Returns
Whether or not Lennard Jones terms are included in the potential set.

Return type
bool

`includeTorsions()`

Returns
Whether or not bond torsion terms are included in the potential set.

Return type
bool

isOplsType(*type_label*)

Function that tests if a type is in the potential.

Parameters
type_label (*str*) – The type being tested.

Returns
Whether or not the tag is defined in the potential.

Return type
bool

lennardJonesCutoff()

Returns
Distance beyond which Lennard Jones terms are truncated.

Return type
PhysicalQuantity of type length

lennardJonesSmoothingLength()

Returns
Distance over which Lennard Jones terms are splined.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

oplsType(*index*, *configuration*, *add_missing_terms=False*)

Gives the OPLS type on an atom in a configuration, ignoring other tags.

Parameters

- **index** (*int*) – Index of the atom for which the type is sought.
- **configuration** (*AtomicConfiguration*) – The configuration containing the atom.
- **add_missing_terms** (*bool*) – Whether or not to add terms for missing OPLS types. **Default:** False

Returns
The type of the atom.

Return type
str

printTotalTypes(*type_class=None*)

Print a table of available atom types. This provides a list of types and descriptions which can be useful when doing manual type assignment.

Parameters
type_class (*str | None*) – Display only atom types with this class label. None displays all types. **Default:** None

printUsedTypes(*configuration*)

Print a table of the types used on the configuration.

Parameters
configuration (*AtomicConfiguration*) – The configuration whose types are being displayed.

totalCharge(*configuration*, *verbose=False*)

Return the total charge of the configuration according to the atom types on the molecule.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration whose charges are being reported.
- **verbose** (*bool*) – Flag as to whether or not additional information about charges is printed. **Default:** False

Returns
The total charge of the configuration according to the atom types on the molecule.

Return type
PhysicalQuantity with units of charge.

typePartialCharge(*type_label*)

Return the atomic partial charge associated with the given OPLS type label.

Parameters
type_label (*str*) – The type label for the OPLS atom type.

Returns
The atom charge associated with the type label.

Return type
PhysicalQuantity of type charge

uniqueString()

Return a unique string representing the state of the object.

useBonds()

Returns
Whether or not bond terms are included in the potential set.

Return type
bool

useElectrostaticDirectSum()

Returns
Whether or not direct Coulomb sums are used for molecules.

Return type
bool

Usage Examples

Build a bulk configuration of poly(vinyl chloride) plasticized with dimethyl sebacate using the OPLS-Min potential.

```
# ------------------------------------------
# Monomer And Included Molecules
# ------------------------------------------
PVC_monomer = nlread('PVC_Monomer.hdf5')[-1]
dimthyl_sebacate = nlread('Dimethyl_Sebacate.hdf5')[-1]

# ------------------------------------------------------------
# OPLS Potential Definition
# ------------------------------------------------------------
opls_min_potential_builder = OPLSMinPotentialBuilder()

# Display the tags and charges placed on the dimethyl sebacate molecule
opls_min_potential_builder.printUsedTypes(dimthyl_sebacate)
opls_min_potential_builder.totalCharge(dimthyl_sebacate, verbose=True)

# ------------------------------------------
# Polymer Sequence
# ------------------------------------------
polymer_sequence = PolymerSequence(
    monomer_configurations=[PVC_monomer],
    number_of_monomers=20,
    number_of_chains=20,
    tactic_ratio=0.5,
)

# ------------------------------------------------------------
# Polymer Monte Carlo Builder
# ------------------------------------------------------------
polymer_builder = PolymerMonteCarloBuilder(
    polymer_sequence=[polymer_sequence],
    density=1.4000*gram/cm**3,
    included_molecules=[(dimthyl_sebacate, 4)],
    monte_carlo_temperature=300.0*Kelvin,
    angle_sampling_points=20,
    repack_molecules=False,
    potential_builder=opls_min_potential_builder
)
bulk_configuration = polymer_builder.buildPolymerConfiguration()

# ------------------------------------------------------------
# Add OPLS Potential To Polymer Configuration
# ------------------------------------------------------------
calculator = opls_min_potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)
nlsave('PVC_DMS.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Force Capped Equilibration
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
equilibration_method = ForceCappedEquilibration(
    temperature=300.00*Kelvin,
    md_steps_per_force_capped_simulation=40000,
    md_time_step=0.50*fs,
    force_capped_simulations=4,
    starting_factor=1.040,
    ending_factor=0.800,
    fixed_indices=None,
)
bulk_configuration = equilibration_method.runEquilibration(bulk_configuration)
nlsave('PVC_DMS.hdf5', bulk_configuration)
```

`OPLS-Min_PVC_DMS_Example.py` `PVC_Monomer.hdf5` `Dimethyl_Sebacate.hdf5`

The given example above demonstrates both how to create a calculator that uses the OPLS-Min potential and also how to assign tags for the dimethyl sebacate molecule that is included in the polymer melt. Both the atom types and charges are printed. This shows both the appropriate types that have been selected and also that the default atom types dont have charge.

Notes

The OPLSMinPotentialBuilder class enables the creation of a *TremoloXCalculator* that implements the OPLS potential of Mayo, Olafson and Goddard[1]. This is a bonded valence forcefield that represents the potential energy of a molecule as a collection of simple functions based on bond lengths, bond angles, torsion angles, inversion angles and inter-atomic distances.

The OPLSMinPotentialBuilder class is very similar to the *OPLSPotentialBuilder* class. Both implement different versions of the OPLS potential. In the *OPLSPotentialBuilder* atom types have assigned atomic partial charges. This results in a large number of atom types required to capture many molecular charge polarizations. In the OPLSMinPotentialBuilder no charges are associated with each atom type, allowing each atom class to only have one type. The significantly simplifies assigning the correct atom types, and is most useful in calculations where atomic partial charges different to the ones in the *OPLSPotentialBuilder* are required.

Atomic partial charges for the configuration can be added when creating the calculator, or estimated using charge equilibration. Both the OPLSMinPotentialBuilder and *OPLSPotentialBuilder* use the same potential parameters for the bond lengths, angles, torsions and improper torsions in the configuration. There are however small differences in some Lennard-Jones non-bonding parameters, resulting from only having one atom type per class. The tagging scheme used by the OPLSMinPotentialBuilder broadly follows that used in the *OPLSPotentialBuilder*. Atom tags start with the prefix `OPLS_`, then contain a string representation of the atom class, and then a number representing the atom type number. In the OPLSMinPotentialBuilder there is only one atom type per class, so this number is `0` in all atom types.

[1] WilliamăL. Jorgensen, DavidăS. Maxwell, and Julian Tirado-Rives. Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids. *J. Am. Chem. Soc.*, 118:11225, November 1996. URL: https://pubs.acs.org/doi/10.1021/ja9621760, doi:10.1021/ja9621760.

OPLSPotentialBuilder

class OPLSPotentialBuilder(*use_bonds=None*, *include_bond_stretching=None*, *include_bond_angles=None*, *include_torsions=None*, *include_inversions=None*, *include_lennard_jones=None*, *include_electrostatic=None*, *lennard_jones_cutoff=None*, *lennard_jones_smoothing_length=None*, *electrostatic_pairwise_cutoff=None*, *electrostatic_smoothing_length=None*, *electrostatic_spme_cutoff=None*, *electrostatic_accuracy=None*, *electrostatic_scale=None*, *use_electrostatic_direct_sum=None*, *ignore_missing_types=None*)

This object builds a OPLS potential using TremoloX.

Parameters

•**use_bonds** (*bool*) – Controls if any bonding (stretching, angle, torsion and inversion) terms are to be included. **Default:** True.

•**include_bond_stretching** (*bool*) – Controls if bonding stretching terms are included. **Default:** use_bonds.

•**include_bond_angles** (*bool*) – Controls if angle bending terms are included. **Default:** use_bonds.

•**include_torsions** (*bool*) – Controls if bond torsion terms are included. **Default:** use_bonds.

•**include_inversions** (*bool*) – Controls if bond inversion torsion terms are included. **Default:** use_bonds.

•**include_lennard_jones** (*bool*) – Controls if Lennard-Jones terms are included. **Default:** use_bonds.

•**include_electrostatic** (*bool*) – Controls if electrostatic terms are included. **Default:** use_bonds.

•**lennard_jones_cutoff** (*PhysicalQuantity* of type distance) – Distance beyond which the Lennard-Jones terms are truncated. **Default:** 10.0 * Angstrom.

•**lennard_jones_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the Lennard-Jones terms are brought to zero at the cutoff. **Default:** 2.0 * Angstrom.

•**electrostatic_pairwise_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in pairwise summation. **Default:** 12 * Angstrom

•**electrostatic_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the electrostatic terms are brought to zero at the cutoff when using the direct electrostatic sum. **Default:** 2.0 * Angstrom

•**electrostatic_spme_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in SPME summation. **Default:** 7.5 * Angstrom

•**electrostatic_accuracy** (*float*) – Sets the relative accuracy of the SPME summation. **Default:** 0.0001

•**electrostatic_scale** (*float*) – Set the relative strength of the electrostatic terms. **Default:** 1.0

•**use_electrostatic_direct_sum** (*bool*) – Use the direct Coulomb sum to calculate the electrostatic terms in MoleculeConfigurations. **Default:** False

•**ignore_missing_types** (*bool*) – Whether or not molecules missing OPLS types are ignored. **Default:** False

assignAtomTypes(*configuration*, *search_types=None*, *overwrite=False*)

Automatically assign OPLS atom tags to a configuration.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration on which to assign tags.

•**search_types** (*list of str | None*) – List of types to assign onto the configuration. If specified only these types will be assigned, with OPLS_UNKNOWN assigned to atoms that dont match any of the given types. The default argument None means that all available types are assigned. **Default:** None.

•**overwrite** (*bool*) – Whether or not to assign new types if the configuration already has a valid set of types. **Default:** False

checkAtomTypes(*configuration*, *add_missing_terms=False*)

Function that tests that each atom has only one valid OPLS type.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration that is being tested.

•**add_missing_terms** (*bool*) – Whether or not to add terms for missing OPLS types. **Default:** False

createCalculator(*configuration*, *check_types=True*, *verbose=False*, *set_improper_torsion_indices=True*, *atomic_charges=None*, *charge_averaging=None*)

Create and return a calculator with the OPLS potential.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration the potential is to be used with.

•**check_types** (*bool*) – Whether or not the types are checked for basic consistency.

•**verbose** (*bool*) – Whether or not to print the terms assigned in the potential set. **Default:** False

•**set_improper_torsion_indices** (*bool*) – Whether or not to add improper torsions onto the input configuration. **Default:** True

•**atomic_charges** (*PhysicalQuantity* of type charge) – Custom atomic charges to be used in the potential.

•**charge_averaging** (*NLFlag | None*) – The type of averaging of charges to be used, if any. Accepted values are None, AtomConnectivity to average over atoms bonded to the same elements and AtomTypes to average over atoms of the same forcefield type. **Default:** None

Returns
The calculator containing the potential.

Return type
TremoloXCalculator

electrostaticAccuracy()

Returns
The relative accuracy of the electrostatic summation for SPME.

Return type
float

electrostaticPairwiseCutoff()

Returns
The cutoff used in pairwise electrostatic summation.

Return type
PhysicalQuantity of type length

electrostaticScale()

Returns
Scaling factor for the electrostatic terms.

Return type
float

`electrostaticSmoothingLength()`

Returns
Distance over which electrostatic terms are splined when using direct summation.

Return type
PhysicalQuantity of type length

`electrostaticSpmeCutoff()`

Returns
The cutoff used in SPME electrostatic summation.

Return type
PhysicalQuantity of type length

`ignoreMissingTypes()`

Returns
Whether or not missing types are ignored on molecules.

Return type
bool

`includeBondAngles()`

Returns
Whether or not bond angle terms are included in the potential set.

Return type
bool

`includeBondStretching()`

Returns
Whether or not bond stretching terms are included in the potential set.

Return type
bool

`includeElectrostatic()`

Returns
Whether or not electrostatic terms are included in the potential set.

Return type
bool

`includeInversions()`

Returns
Whether or not bond inversion terms are included in the potential set.

Return type
bool

`includeLennardJones()`

Returns
Whether or not Lennard Jones terms are included in the potential set.

Return type
bool

includeTorsions()

Returns
Whether or not bond torsion terms are included in the potential set.

Return type
bool

isOplsType(*type_label*)

Function that tests if a type is in the potential.

Parameters
type_label (*str*) – The type being tested.

Returns
Whether or not the tag is defined in the potential.

Return type
bool

lennardJonesCutoff()

Returns
Distance beyond which Lennard Jones terms are truncated.

Return type
PhysicalQuantity of type length

lennardJonesSmoothingLength()

Returns
Distance over which Lennard Jones terms are splined.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

oplsType(*index*, *configuration*, *add_missing_terms=False*)

Gives the OPLS type on an atom in a configuration, ignoring other tags.

Parameters

- **index** (*int*) – Index of the atom for which the type is sought.
- **configuration** (*AtomicConfiguration*) – The configuration containing the atom.
- **add_missing_terms** (*bool*) – Whether or not to add terms for missing OPLS types. **Default:** False

Returns
The type of the atom.

Return type
str

printTotalTypes(*type_class=None*)

Print a table of available atom types. This provides a list of types and descriptions which can be useful when doing manual type assignment.

Parameters
type_class (*str | None*) – Display only atom types with this class label. None displays all types. **Default:** None

printUsedTypes(*configuration*)

Print a table of the types used on the configuration.

Parameters
configuration (*AtomicConfiguration*) – The configuration whose types are being displayed.

totalCharge(*configuration*, *verbose=False*)

Return the total charge of the configuration according to the atom types on the molecule.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration whose charges are being reported.
- **verbose** (*bool*) – Flag as to whether or not additional information about charges is printed. **Default:** False

Returns
The total charge of the configuration according to the atom types on the molecule.

Return type
PhysicalQuantity with units of charge.

typePartialCharge(*type_label*)

Return the atomic partial charge associated with the given OPLS type label.

Parameters
type_label (*str*) – The type label for the OPLS atom type.

Returns
The atom charge associated with the type label.

Return type
PhysicalQuantity of type charge

uniqueString()

Return a unique string representing the state of the object.

useBonds()

Returns
Whether or not bond terms are included in the potential set.

Return type
bool

useElectrostaticDirectSum()

Returns
Whether or not direct Coulomb sums are used for molecules.

Return type
bool

Usage Examples

Build a bulk configuration of poly(vinyl chloride) plasticized with dimethyl sebacate using the OPLS potential.

```
# ------------------------------------------
# Monomer And Included Molecules
# ------------------------------------------
PVC_monomer = nlread('PVC_Monomer.hdf5')[-1]
dimthyl_sebacate = nlread('Dimethyl_Sebacate.hdf5')[-1]

# ------------------------------------------------------------
# OPLS Potential Definition
# ------------------------------------------------------------
opls_potential_builder = OPLSPotentialBuilder()

# Display the tags and charges placed on the dimethyl sebacate molecule
opls_potential_builder.printUsedTypes(dimthyl_sebacate)
opls_potential_builder.totalCharge(dimthyl_sebacate, verbose=True)

# ------------------------------------------
# Polymer Sequence
# ------------------------------------------
polymer_sequence = PolymerSequence(
    monomer_configurations=[PVC_monomer],
    number_of_monomers=20,
    number_of_chains=20,
    tactic_ratio=0.5,
)

# ------------------------------------------------------------
# Polymer Monte Carlo Builder
# ------------------------------------------------------------
polymer_builder = PolymerMonteCarloBuilder(
    polymer_sequence=[polymer_sequence],
    density=1.4000*gram/cm**3,
    included_molecules=[(dimthyl_sebacate, 4)],
    monte_carlo_temperature=300.0*Kelvin,
    angle_sampling_points=20,
    repack_molecules=False,
)
bulk_configuration = polymer_builder.buildPolymerConfiguration()

# ------------------------------------------------------------
# Add OPLS Potential To Polymer Configuration
# ------------------------------------------------------------
calculator = opls_potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)
nlsave('PVC_DMS.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Force Capped Equilibration
# ------------------------------------------------------------
equilibration_method = ForceCappedEquilibration(
```

(continues on next page)

(continued from previous page)

```
    temperature=300.00*Kelvin,
    md_steps_per_force_capped_simulation=40000,
    md_time_step=0.50*fs,
    force_capped_simulations=4,
    starting_factor=1.040,
    ending_factor=0.800,
    fixed_indices=None,
)
bulk_configuration = equilibration_method.runEquilibration(bulk_configuration)
nlsave('PVC_DMS.hdf5', bulk_configuration)
```

`OPLS_PVC_DMS_Example.py` `PVC_Monomer.hdf5` `Dimethyl_Sebacate.hdf5`

The given example above demonstrates both how to create a calculator that uses the OPLS-AA potential and also how to assign tags for the dimethyl sebacate molecule that is included in the polymer melt. Both the atom types and charges are printed. This shows both the appropriate types that have been selected and also that the overall charge of the molecule is zero.

Notes

The OPLSPotentialBuilder class enables the creation of a *TremoloXCalculator* that implements the OPLS-AA potential of Mayo, Olafson and Goddard[1]. This is a bonded valence forcefield that represents the potential energy of a molecule as a collection of simple functions based on bond lengths, bond angles, torsion angles, inversion angles and inter-atomic distances. The overall mathematical form of the OPLS-AA potential can be given as:

$$E(\mathbf{x}) = \sum_{i}^{bonds} k_i(r_i - r_0)^2 + \sum_{i}^{angles} k_i(\theta_i - \theta_0)^2 + \sum_{i}^{torsions} \sum_{n=1}^{3} k_{i,n}(1 - \cos(n\phi_i - \delta_{i,n}))$$
$$+ \sum_{i}^{inversion} k_i(1 - \cos(n\chi_i)) + \sum_{ij}^{atoms} 4\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \sum_{ij}^{atoms} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Here r, θ, ϕ and χ represent inter-atomic and bond distances, angles, torsions and inversions respectively in the configuration.

To create a OPLS-AA calculator, an instance of the *OPLSPotentialBuilder* must first be created. The constructor for this class has a number of options that controls the way in which the calculator built. Individual types of interactions can be excluded by setting the `include_` options to `False` By default all possible terms are included. All valence terms can also be excluded by setting the argument `use_bonds` to False. This is appropriate for configurations consisting of single atoms that are not bonded to each other.

As the *OPLSPotentialBuilder* class uses the bonding in the configuration to determine appropriate potentials, bonds should be set on the configuration. This can be done with the `findBonds()` method in either the *MoleculeConfiguration* or *BulkConfiguration* class. This automatically assigns bonds between atoms that are within the covalent radii of each other. Specific bonds can also be assigned with the `setBonds()`, which takes a list of the indices of pairs of atoms that are to be considered bonded.

There are then a number of options that control the non-bonding interactions. The parameter `lennard_jones_cutoff` sets the distance beyond which Lennard-Jones interactions are truncated. To avoid integration problems in molecular dynamics, Lennard-Jones interactions are brought to zero at the cutoff using a spline function. The width of this spline can be set using the `lennard_jones_smoothing_length` parameter. There are no specific hydrogen hydrogen bonding terms defined within the OPLS-AA potential.

[1] WilliamăL. Jorgensen, DavidăS. Maxwell, and Julian Tirado-Rives. Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids. *J. Am. Chem. Soc.*, 118:11225, November 1996. URL: https://pubs.acs.org/doi/10.1021/ja9621760, doi:10.1021/ja9621760.

Electrostatic interactions are a little more complicated. By default the *OPLSPotentialBuilder* adds a *CoulombSPME* electrostatic solver for calculators to be used with a bulk configuration, and a *CoulombDSF* electrostatic solver to calculators to be used with a molecule configuration. When adding an *CoulombSPME* solver the cutoff used for calculating the real-space interactions can be set using the `electrostatic_spme_cutoff` argument. Likewise the accuracy of the smoothed particle mesh Ewald solver can be set using the `electrostatic_accuracy` argument. In the case where a *CoulombDSF* solver is added to the potential, the cutoff used can be set using the `electrostatic_pairwise_cutoff` argument. In cases where molecular electrostatics are to be compared with values calculated in a bulk configuration, an explicit splined Coulomb potential can be used for molecules instead of the damped shifted force potential. In this case a *CoulombN2Spline* electrostatic solver can be added by using `use_electrostatic_direct_sum` argument. As with the damped shifted force potential the cutoff is controlled using the `electrostatic_pairwise_cutoff` argument, and like the Lennard-Jones potential is brought to zero with a spline whose length is defined by the `electrostatic_smoothing_length` argument. For all Coulomb solvers the total electrostatic energies and forces can be scaled up or down using the `electrostatic_scale` argument.

Once the *OPLSPotentialBuilder* object is instantiated, a calculator is created using the `createCalculator` method. This takes a configuration and returns the appropriate OPLS-AA calculator for that configuration. To use electrostatic potentials in the calculator, atomic partial charges have to be assigned to each atom. The OPLS-AA defines appropriate charges for each atom type, which by default are added to the created calculator. It is also possible to add pre-calculated charges from different sources using the `atomic_charges` argument. This argument takes a list of charges that have the same ordering as the atoms in the configuration. The partial atomic charges can also be averaged according to a few different schemes. This is controlled by the `charge_averaging` parameter, which takes an NLFlag which indicates the type of averaging. The flag `AtomConnectivity` averages charges on each atom based on the element and the other elements it is bonded to. The flag `AtomTypes` averages according to the assigned OPLS-AA type. For the calculator to recognize appropriate inversion terms, improper torsion indices also need to be set on the configuration. By default this is done automatically, however existing improper torsions can be preserved by setting the `set_improper_torsion_indices` to `False`. As the OPLS-AA potential contains a number of different potentials, details of each potential used to create the calculator can be printed out using the `verbose` argument, which is `False` by default.

When creating a OPLS-AA calculator for a configuration, the configuration must have the appropriate tags for each atom type. Atom types in the OPLS-AA forcefield are grouped into classes. Atom types in a class all have the same valence potential terms and individual Lennard-Jones and electrostatic potentials. This reduces the number of parameters required to construct a transferable forcefield while still allowing specific tailoring of the potentials to different molecules. Tags used by the OPLS-AA forcefield all begin with the prefix `OPLS_`. The next section of tag gives the label of the class of the atomic type, and the final section gives the number of the type in its class. Appropriate OPLS tags can be assigned automatically using the method `assignAtomTypes`. This method uses the bonding environment of each atom to estimate appropriate types. These tags are then added to the input configuration. Type definitions have been added for atom types common to polymers, although not all atom types have a definition. In cases where the *OPLSPotentialBuilder* is unable to recognize the type of an atom, the tag `OPLS_UNKNOWN` is added to that atom.

The OPLSPotentialBuilder contains a few methods to assist in manually assigning the correct types to configurations in cases where the types cannot be automatically assigned. Atom types in the OPLS-AA potential are generally defined for specific functional groups. As a result, atomic partial charges for a given functional group usually add to zero or an integer charge for charged groups. To discover the appropriate types for a configuration, the method `printTotalTypes` prints a table of the available atomic types, with a short description of the kinds of atoms this type represents. As they are defined together, types for different functional groups are often grouped together. Types for just one class can also be printed using the `type_class` argument. The types assigned on a specific configuration can also be displayed using the `printUsedTypes` method. As functional groups generally have zero overall charge, correctly typed configurations should also have zero overall charge. Calculating the total assigned charge to the configuration is therefore a useful check to make sure the correct types have been added to the configuration. The total assigned charge to a configuration using the OPLS forcefield can be calculated using the `totalCharge` method.

OneShotSelfConsistentBornApproximation

class OneShotSelfConsistentBornApproximation(*device_configuration=None*, *energies=None*, *kpoints=None*, *kpoints_weights=None*, *electron_phonon_self_energies=None*, *loa_order=None*, *calculate_local_density=None*)

Class for performing one-shot self-consistent Born approximation. The currents are calculated using a Pade approximant, which is optimal for the given `loa_order`.

Parameters

- **device_configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which the one-shot SCBA currents should be calculated. The bias voltage used in the calculation will correspond to the one defined by `electrode_voltages` on the calculator.
- **energies** – The energies used in the one-shot SCBA calculation. **Default:** `numpy.linspace(-1, 1, 401) * eV`
- **kpoints** (*MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – The k-points used in the one-shot SCBA calculation. **Default:** The sampling along A and B used for the self-consistent calculation.
- **kpoints_weights** (*sequence of float*) – The weight of each k-point. **Default:** The weights corresponding to the MonkhorstPackGrid, or a list of [1.0, 1.0,] if k-points are specified as floats.
- **electron_phonon_self_energies** (list of instances of *AcousticDeformationPotentialSelfEnergy* | *OpticalDeformationPotentialSelfEnergy*) – A list of models used to determine the Electron-Phonon scattering self energy. If None is specified, no electron-phonon self energy is included.
- **loa_order** (*int*) – The order of the generalized lowest-order approximation (LOA). Must be a non-negative integer. **Default:** 1
- **calculate_local_density** (*bool*) – Whether to calculate local current density, besides the current. **Default:** `False`

ballisticCurrent(*positive_current_convention=None*, *spin=None*)

Parameters

- **positive_current_convention** (`LeftToRight` | `RightToLeft`) – The convention for the direction of the positive current. **Default:** `RightToLeft`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin flag. **Default:** `Spin.Sum`

Returns
The non-interacting component of the current, i.e. the 0th order component.

Return type
PhysicalQuantity of type current

calculateLocalDensity()

Returns
Whether to calculate the local densities.

Return type
bool

current(*positive_current_convention=None*, *spin=None*)

Parameters

- **positive_current_convention** (`LeftToRight` | `RightToLeft`) – The convention for the direction of the positive current. **Default:** `RightToLeft`

- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin flag. **Default:** `Spin.Sum`

Returns
The calculated interacting current.

Return type
PhysicalQuantity of type current

electronPhononSelfEnergies()

Returns
The objects describing the electron-phonon self energy models.

Return type
sequence of *AcousticDeformationPotentialSelfEnergy* | *OpticalDeformationPotentialSelfEnergy*

energies()

Returns
The energy used in this SCBA calculation.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero used for the energy scale in energy dpeendent quantities.

Return type
PhysicalQuantity of type energy.

kpoints()

Returns
The k-point grid used in the one-shot SCBA calculation.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

kpointsWeights()

Returns
The weights of the k-points.

Return type
sequence of float

loaOrder()

Returns
The order of the LOA.

Return type
int

localCurrentDensityData(*positive_current_convention=None*)

Parameters
positive_current_convention (`LeftToRight` | `RightToLeft`) – The convention for the direction of the positive current. **Default:** `RightToLeft`

Returns
A tuple with the z coordinates, energies, and the calculated local charge density. The local current density is a 2D array specifying the value for each z coordinate and energy.

Return type

tuple of (
PhysicalQuantity of type length, *PhysicalQuantity* of type energy, *PhysicalQuantity* of type current / energy)

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString()`

Return a unique string representing the state of the object.

OnsiteShifts

`class OnsiteShifts`

Class for representing the OnsiteShifts databases. This class is not directly exposed, but can be accessed through ATK_U and ATK_W instances.

All calculations have been performed with GGA.PBE.

`OnsiteShifts.__call__(element, orbitals, identifier=None)`

The call function used for accessing the parameters stored in the database.

Parameters

- **element** (*PeriodicTableElement*) – An element in the periodic table to get the the data for.
- **orbitals** (*list of str*) – The list of orbitals to obtain the data for. If there is no stored calculated data for the requested orbital, the value `0.0*eV` will be returned. Supported orbitals are [1s, 2s, 2p, 3s, 3p, 4s, 3d, 4p, 5s, 4d, 5p, 6s, 4f, 5d, 6p, 7s, 5f, 6d, 7p].
- **identifier** (`'ae' | 'ae-exc' | 'ncp'`) – An identifier string for the data base entry. There are all-electron non-relativistic calculations (ae), all-electron with an excited valence configuration (ae-exc), and norm-conserving pseudopotential calculations (ncp). **Default:** For elements up to and including Zn the all electron ae database will be used as default, except for the 3d metals Ti, V, Mn, Fe, Co, Ni, for which the all- electron with excited configuration ae-exc database will be used. For heavier elements, from Ga and above, the normconserving pseudo-potential ncp database will be used, except for Kr, Sr, Xe, Cs and Ba (and the Lanthanoids and Actinoids when supported) for which the ae values will be used.

Returns
The stored data for the given combination of identifier, element and orbitals.

Return type
PhysicalQuantity of type energy.

ATK_U

This database contains the onsite Hartree shifts.

Print the U parameters for iron, calculated from a pseudo-potential, all-electron, and excited-state all-electron calculation:

```
print ATK_U(Iron, ['1s','2s','2p','3s','3p','3d','4s','4p'], 'ncp')
print ATK_U(Iron, ['1s','2s','2p','3s','3p','3d','4s','4p'], 'ae')
print ATK_U(Iron, ['1s','2s','2p','3s','3p','3d','4s','4p'], 'ae-exc')
```

For more details, see *Onsite Hartree shift parameters*.

ATK_W

This database contains the onsite spin splitting.

Print the valence W parameters for iron, calculated from a pseudo-potential, all-electron, and excited-state all-electron calculation:

```
print ATK_W(Iron, ['3d','4s','4p'], 'ncp')
print ATK_W(Iron, ['3d','4s','4p'], 'ae')
print ATK_W(Iron, ['3d','4s','4p'], 'ae-exc')
```

For more details, see *Spin polarization*.

OpenMXBasisSet

class OpenMXBasisSet(*filename*, *tier=None*, *element=None*, *orbitals=None*, *occupations=None*, *hubbard_u=None*, *filling_method=None*, *pseudopotential=None*, *onsite_spin_orbit_split=None*, *onsite_spin_orbit_coupling=None*, *filter_mesh_cutoff=None*, *atomic_species=None*, *projector_shift=None*, *dft_half_parameters=None*)

For a given element, this class represents the basis set read from an external file.

Parameters

- **filename** (*str*) – Filename of the VPS pseudopotential file.
- **tier** (*int*) – Basis subset to use. Mutually exclusive with `atomic_species`. **Default:** 1 (corresponding to single-zeta set).
- **element** (*PeriodicTableElement*) – The element associated with this basis set. **Default:** None.
- **orbitals** (*list*) – The set of orbitals that forms this basis set. Each element in the list should be an instance of *ConfinedOrbital*, *AnalyticalSplit*, *PolarizationOrbital*, *NumericalOrbital*, or *HydrogenOrbital*. **Default:** None.
- **occupations** (*list of floats*) – The initial occupation of the basis orbitals. **Default:** None.

- **hubbard_u** (*PhysicalQuantity* of type energy) – The Hubbard U for each orbital shell. **Default:** None.
- **filling_method** (`SphericalSymmetric` | `Anisotropic`) – The method used for setting up the initial occupation. **Default:** `SphericalSymmetric`.
- **pseudopotential** (*NormConservingPseudoPotential*) – The pseudopotential to be used for generating this basis set. **Default:** None.
- **onsite_spin_orbit_split** (*PhysicalQuantity* of type energy) – Spin-orbit splitting for each orbital. **Default:** No splitting.
- **onsite_spin_orbit_coupling** (*list*) – The relative coupling term for spin-orbit interaction. **Default:** None.
- **filter_mesh_cutoff** (positive *PhysicalQuantity* of type energy) – Specify a cutoff for filtering the basis functions. **Default:** None.
- **atomic_species** (*string*) – Uncontracted basis specification. Mutually exclusive with `tier`. This string is interpreted as `(<orbital name><number of uncontracted shells>)...` e.g. `"s4p3d1"` **Default:** None.
- **projector_shift** (*PseudoPotentialProjectorShift*) – Pseudopotential projector shift for each angular momentum. **Default:** None
- **dft_half_parameters** (*DFTHalfParameters* | `Automatic` | `Disabled`) – The DTF-1/2 parameters used for this basis set in the LCAO calculation. When set to `Automatic` the DFT-1/2 parameters will be populated from a table of optimized parameters. By setting it to `Disabled` DFT-1/2 will not be active for this basis set. This is used to disable DFT-1/2 for some elements or atomic sites while DFT-1/2 may be active for other atoms in the configuration. **Default:** `Automatic`

angularMomenta()

Returns
The angular momenta of the basis set.

Return type
list

atomicSpecies()

Returns
The uncontracted basis description as a dictionary.

Return type
dict

atomicSpeciesString()

Returns
The uncontracted basis description as a string.

Return type
str

dftHalfParameters()

The DFT-1/2 parameters.

Returns
The DFT-1/2 parameters.

Return type
DFTHalfParameters

element()

Returns
The element associated with this basis set.

Return type
PeriodicTableElement

filename()

Returns
The user given filename of the OpenMX PAO basis set file.

Return type
str

fillingMethod()

Returns
The method used for setting up the initial occupation.

Return type
SphericalSymmetric | Anisotropic

filterMeshCutoff()

Returns
The cutoff for the filter function

Return type
PhysicalQuantity of type energy

hubbardU()

Returns
The Hubbard U energy for each orbital shell.

Return type
PhysicalQuantity of type energy

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfValenceElectrons()

Private function for the getting the total number of electrons.

occupations()

Returns
The occupations associated with the orbitals.

Return type
list

onsiteSpinOrbitCoupling()

Returns
The relative coupling term for spin-orbit interaction.

Return type
list

onsiteSpinOrbitSplit()

Returns
The spin-orbit splitting of each orbital

Return type
PhysicalQuantity of type energy or None.

orbitals()

Returns
The orbitals that forms this basis set.

Return type
list of *ConfinedOrbital* | *AnalyticalSplit* | *PolarizationOrbital* | *NumericalOrbital* | *HydrogenOrbital*

projectorShift()

Returns
The projector shift energies specified for each angular momentum.

Return type
PseudoPotentialProjectorShift

pseudopotential()

Returns
The pseudopotential used in generating the basis set.

Return type
NormConservingPseudoPotential

tier()

Returns
The basis subset level used in the calculation.

Return type
int | None

uniqueString()
Return a unique string representing the state of the object.

OpticalDeformationPotentialSelfEnergy

class OpticalDeformationPotentialSelfEnergy(*coupling=None*, *phonon_energy=None*)
A class which represent a deformation potential model for acoustic phonon modes.

Parameters

- **coupling** (*PhysicalQuantity* of type energy) – The electron-phonon coupling strength of the optical modes. The coupling strength can be given as a single value, or a 1D array specifying the value for each atom (in units of energy). **Default:** `0 * eV`
- **phonon_energy** (*PhysicalQuantity* of type energy) – The energy of the optical modes, assumed constant. **Default:** `0.063 * eV`

coupling()

Returns
The electron-phonon coupling strength of the acoustic modes.

Return type
PhysicalQuantity with unit `eV**2`

phononEnergy()

Returns
The optical phonon energy, assumed constant.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

OpticalSpectrum

class OpticalSpectrum(*configuration*, *kpoints=None*, *energies=None*, *broadening=None*, *bands_below_fermi_level=None*, *bands_above_fermi_level=None*, *optical_intraband_parameters=None*, *method=None*, *transition_matrix_operator=None*)

Constructor for the optical spectrum object.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the optical spectrum.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which to calculate the transitions. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the optical spectrum. **Default:** `numpy.linspace(0, 4, 101)*eV`
- **broadening** (*PhysicalQuantity* of type energy) – The broadening parameter used for the optical spectrum. **Default:** `0.1 * eV`
- **bands_below_fermi_level** (*int*) – The maximum number of valence bands per principal spin channel to include at each k-point. If the number specified is larger than the actual number of occupied states, all are used. **Default:** `4`
- **bands_above_fermi_level** (*int*) – The maximum number of conduction bands per principal spin channel to include at each k-point. If the total number of requested bands is larger than the basis size, the maximally possible number of bands is used. **Default:** `4`
- **optical_intraband_parameters** (`OpticalIntrabandParameters` | `None`) – If given the intraband contribution will be included in the OpticalSpectrum. This container should hold the plasma_frequency and inverse_life_time parameters. **Default:** `None`, i.e. optical intraband parameters is not included.
- **method** (`None` | `Full` | *KDotPExpansion3D*) – The method used for the eigenvalue and transition matrix elements computation.

  The default is to perform an exact diagonalization to determine the eigenvalues and the wave-functions, with which we calculate the momentum matrix elements, exactly at each requested k-point.

  Alternatively, the k.p expansion method can be used to interpolate the eigenvalues and the transition matrix elements at the requested k-points using the exact eigenvalues and eigenfunctions calculated in the SCF calculation.

NOTE: Only the *PlaneWaveCalculator* supports the k.p method at the moment.

- **transition_matrix_operator** (MomentumOperator | VelocityOperator **Default:** MomentumOperator) – The transition matrix operator, either the momentum operator or the velocity operator. The default option is velocity operator for normconserving LCAO and semi-empirical calculators and momentum operator for other calculator options (PAW LCAO and PlaneWave calculators).

The velocity operator is only supported for the normconserving LCAO and the semi-empirical calculator.

absorption(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The absorption tensor, the dimension is (3, 3, N), where N is the number of energies.

Return type
ndarray.

broadening()

Returns
The broadening parameter used for the optical spectrum.

Return type
PhysicalQuantity of type energy

energies()

Returns
The energies for which to calculate the optical spectrum.

Return type
PhysicalQuantity of type energy

evaluateDielectricConstant(*spin=None*)

Function for evaluating the real part of the dielectric constant using the Kubo-Greenwood formalism.

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The real part of the dielectric constant.

Return type
numpy.array

evaluateImaginaryDielectricConstant(*spin=None*)

Function for evaluating the imaginary part of the dielectric constant using the Kubo- Greenwood formalism.

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The imaginary part of the dielectric constant.

Return type
numpy.array

evaluateImaginaryPolarizability(*spin=None*)

Function for evaluating the imaginary part of the polarizability using the Kubo-Greenwood formalism.

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The imaginary part of the polarizability.

Return type
PhysicalQuantity with the unit Hartree**3 * vacuum_permitivity

evaluatePolarizability(*spin=None*)

Function for evaluating the real part of the polarizability using the Kubo-Greenwood formalism.

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The real part of the polarizability.

Return type
PhysicalQuantity with the unit Hartree**3 * vacuum_permitivity

imaginarySusceptibility(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
Imaginary part of the susceptibility tensor.

Return type
numpy.ndarray.

maxConductionStates()

Returns
The maximum number of conduction states to include.

Return type
int

maxValenceStates()

Returns
The maximum number of valence states to include.

Return type
int

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Returns
The method used for the eigenvalue and transition matrix elements computation.

Return type
Full | *KDotPExpansion3D*

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

opticalConductivity(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The optical conductivity, the dimension is (3, 3, N), where N is the number of energies.

Return type
ndarray.

opticalIntrabandParameters()

Returns
The optical_intraband_parameters set on the optical spectrum.

Return type
OpticalIntrabandParameters

realSusceptibility(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
Real part of the susceptibility tensor.

Return type
numpy.ndarray.

reflectivity(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum) – The spin to calculate for. **Default:** Spin.Sum

Returns
The reflectivity tensor, the dimension is (3, 3, N), where N is the number of energies.

Return type
Dimensionless ndarray.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

transitionMatrixOperator()

Returns
The transition matrix operator.

Return type
MomentumOperator | VelocityOperator

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *OpticalSpectrum* of silicon, save it to a file, and print out the dielectric tensor:

```
# Set up silicon crystal.
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    cartesian_coordinates=[[ 0.     , 0.     , 0.],
                           [ 1.35765, 1.35765, 1.35765]]*Angstrom
    )
# Setup calculator.
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(6, 6, 6),
    )

# Setup DFT calculator using TB09 meta-GGA.
calculator = LCAOCalculator(
    basis_set=LDABasis.DoubleZetaDoublePolarized,
    exchange_correlation=MGGA.TB09LDA,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Optical spectrum.
optical_spectrum = OpticalSpectrum(
    configuration=bulk_configuration,
    kpoints=MonkhorstPackGrid(15,15,15),
    energies=numpy.linspace(0,5,101)*eV,
    broadening=0.1*eV,
    bands_below_fermi_level=4,
    bands_above_fermi_level=4,
    )

nlsave('si_optical.hdf5', optical_spectrum)

# Print out the dielectric constant (Experimental value is 11.9).
print('Dielectric Tensor')
print(optical_spectrum.evaluateDielectricConstant()[:,:,0])
```

si_optical.py

Notes

•We use the Kubo-Greenwood formula to calculate the susceptibility tensor[1][2]

$$\chi_{ij}(\omega) = \frac{e^2}{\hbar m_e^2 V} \sum_{nm\mathbf{k}} \frac{f_{m\mathbf{k}} - f_{n\mathbf{k}}}{\omega_{nm}^2(\mathbf{k})[\omega_{nm}(\mathbf{k}) - \omega - i\Gamma/\hbar]} p_{nm}^i(\mathbf{k}) p_{mn}^j(\mathbf{k}) \,,$$

where $p_{nm}^i = \langle n\mathbf{k}|\mathbf{p}^i|m\mathbf{k}\rangle$ is the i-th component (labeling electrons) of the momentum operator between state n and m, m_e (e) is the electron mass (charge), V the volume, Γ the energy broadening, $\hbar\omega_{nm} = E_n - E_m$ and $f_{n\mathbf{k}}$ the Fermi function evaluated at the band energy $E_n(\mathbf{k})$.

•The response coefficients, the relative dielectric constant, ϵ_r, polarizability, α, and optical conductivity, σ, are related to the susceptibility as

$$\begin{aligned}
\epsilon_r(\omega) &= (1 + \chi(\omega)), \\
\alpha(\omega) &= V\epsilon_0\chi(\omega), \\
\sigma(\omega) &= -i\omega\epsilon_0\chi(\omega).
\end{aligned}$$

The derivation of the last relation can be found in Ref.[3].

Note: Accounting for spin, the relation between the dielectric constant and the susceptibility can be written as $\epsilon_r(\omega) = 1 + \sum_\sigma \chi_\sigma(\omega)$. The dielectric constant has no explicit spin dependency. For calculations of type *polarized*, textual representations of the susceptibility will show the spin resolved result, i.e. `Spin.Up` and `Spin.Down` components while the sum over spins always is given for the dielectric constant. Nonetheless, it is possible to obtain the spin-resolved dielectric constant using the corresponding query methods with either `Spin.Up` or `Spin.Down` for the spin parameter. In these cases, the vacuum contribution is split up equally between up and down components. This feature is maintained for backward compatibility reasons and might disappear in future releases.

•The refractive index, n, is related to the complex dielectric constant through

$$n + i\kappa = \sqrt{\epsilon_r},$$

here κ is the extinction coefficient. In terms of the real (ϵ_1) and complex parts (ϵ_2) of the dielectric constant

$$\begin{aligned}
n &= \sqrt{\frac{\sqrt{\epsilon_1^2 + \epsilon_2^2} + \epsilon_1}{2}}, \\
\kappa &= \sqrt{\frac{\sqrt{\epsilon_1^2 + \epsilon_2^2} - \epsilon_1}{2}}.
\end{aligned}$$

•The optical absorption coefficient is related to the extinction coefficient through[4]

$$\alpha_a = 2\frac{\omega}{c}\kappa.$$

The reflectivity is given by[5]

$$R = \frac{(1-n)^2 + \kappa^2}{(1+n)^2 + \kappa^2}.$$

[1] WalterăA. Harrison. *Solid State Theory*. McGraw-Hill, 1970.

[2] J.ăE. Sipe and EdăGhahramani. Nonlinear optical response of semiconductors in the independent-particle approximation. *Phys. Rev. B*, 48(16):11705–11722, October 1993. URL: https://link.aps.org/doi/10.1103/PhysRevB.48.11705 (visited on 2019-05-10), doi:10.1103/PhysRevB.48.11705.

[3] RichardăM. Martin. *Electronic structure: Basic theory and practical methods*. Cambridge University Press, New York, 2004. ISBN 0-521-78285-6.

[4] DavidăJ. Griffiths. *Introduction to Electrodynamics (3rd Edition)*. Prentice Hall, 1999. ISBN 9780138053260.

[5] M.ăP. Desjarlais. Density functional calculations of the reflectivity of shocked xenon with ionization based gap corrections. *Contrib. Plasma Phys.*, 45(3-4):300–304, 2005. doi:10.1002/ctpp.200510034.

- The definition of `bands_above_fermi_level` and `bands_below_fermi_level` depends on the spin type of the calculation, as described in the *Notes* of the *EffectiveMass* analysis object.

Thus from the dielectric constant a number of other optical properties are easily obtained. For further background information on optical spectra see *Optical response functions*.

Intraband contribution

In metals both the interband term described above and an intraband term from plasmons contribute to the dielectric response, $\chi_{ij} = \chi_{ij}^{\text{inter}} + \chi_{ij}^{\text{intra}}$. Within the Drude model of free electrons we can obtain the intraband contribution as[6]

$$\chi_{ij}^{\text{intra}}(\omega) = -\frac{\omega_{\text{P},ij}^2}{\omega^2 + i\omega\gamma} = -\frac{\omega_{\text{P},ij}^2}{\omega^2 + \gamma^2} + i\frac{\omega_{\text{P},ij}^2\gamma}{\omega^3 + \omega\gamma^2}$$

Here ω_{P} is the plasma frequency and $\gamma(T) = \hbar/\tau(T)$ is a temperature dependent broadening obtained from the phonon limited life-time $\tau(T)$. The plasma frequency can be obtained from the band energies and velocities integrated over the Fermi surface:

$$\omega_{\text{P},ij}^2 = \frac{e^2}{V\varepsilon_0}\sum_{n,\mathbf{k}} g_s g_k \mathbf{v}_i(n\mathbf{k})\mathbf{v}_j(n\mathbf{k})\delta(\epsilon_F - \epsilon_{n\mathbf{k}})$$

Here V is the unit cell volume, g_s is the spin degeneracy factor, g_k is the $\mathbf{k}$-point weights, $\mathbf{v}(n\mathbf{k}) = 1/\hbar\partial\epsilon_{n\mathbf{k}}/\partial\mathbf{k}$ is the band velocity of a band indexed n and i, j are Cartesian directions.

The plasma frequency can be set to a known value or obtained from the following example:

```
plasma_frequency = calculatePlasmaFrequency(
    bulk_configuration,
    electron_bands=electron_bands,
    kpoints_grid=MonkhorstPackGrid(15, 15, 15),
    use_kpoint_symmetry=True)
```

Here `electron_bands` is a list of indices of electron bands included.

The intraband contribution is included in the optical spectrum calculation if the optional input `optical_intraband_parameters = OpticalIntrabandParameters(plasma_frequency, inverse_life_time)` is supplied to the *OpticalSpectrum*:

```
optical_spectrum = OpticalSpectrum(
    bulk_configuration,
    kpoints=MonkhorstPackGrid(15, 15, 15),
    energies=numpy.linspace(0, 5, 101) * eV,
    broadening=0.1 * eV,
    bands_above_fermi_level=4,
    bands_below_fermi_level=4,
    optical_intraband_parameters=OpticalIntrabandParameters(0.1 * eV, 1.8 *␣
↪femtoSecond**(-1)),
)
```

The `plasma_frequency` can be a single number (as the above example) or a tensor (as obtained by the full calculation). The phonon limited `inverse_life_time` can be obtained from the *ElectronPhononCoupling* or *Mobility* analysis objects.

[6] MichielăJ. van Setten, Suleyman Er, Geert Brocks, RobertăA. deăGroot, and GillesăA. deăWijs. First-principles study of the optical properties of MgxTi(1-x)H2. *Phys. Rev. B*, 79(12):125117, mar 2009. URL: https://link.aps.org/doi/10.1103/PhysRevB.79.125117 (visited on 2019-10-11), doi:10.1103/PhysRevB.79.125117.

OptimizeDeviceConfiguration

class OptimizeDeviceConfiguration(*device_configuration*, *filename*, *object_id*, *calculator=None*, *optimization_region_length=None*, *optimization_region_center=None*, *passivate_electrode_surfaces=None*, *optimize_geometry_parameters=None*, *log_filename_prefix=None*, *perform_device_optimization=None*)

Class for optimizing a device configuration.

Parameters

- **device_configuration** (*DeviceConfiguration*) – The device configuration to optimize.
- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **calculator** (Calculator) – The bulk calculator to use for the optimization of the central region of the device. **Default:** A bulk calculator with settings similar to the calculator set on the given device_configuration.
- **optimization_region_length** (*PhysicalQuantity* of type length) – The length along the transport direction of the region to be optimized within the central region of the device. Note that the length of the optimization region is automatically capped to keep the given electrode extensions fixed on either side. **Default:** 10.0 * Angstrom
- **optimization_region_center** (*PhysicalQuantity* of type length) – The position along the transport direction of the center of the optimization region. **Default:** The center along the transport direction of the central region of the device after the left and right electrode extensions have been removed. The maximum valid number of repeats of the minimal electrodes are used to calculate the electrode extensions.
- **passivate_electrode_surfaces** (*sequence (size 2) of bool*) – Whether the left and right surfaces of the central region of the device along the transport direction should be passivated when extracting the central region for optimization. **Default:** (True, True) for LCAO, plane-wave and semi-empirical calculators; (False, False) for forcefield calculators.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters to use for optimizing the geometry. Note that pre_step_hook and post_step_hook must be left unset. Also note that max_stress, target_stress, restart_strategy and enable_optimization_stop_file do not have any effect since stress is not optimized, and the restart mechanism within *OptimizeGeometry()* is disabled. **Default:** OptimizeGeometryParameters()
- **log_filename_prefix** (str | LogToStdOut) – Filename prefix for the logging output of the calculations, each to be stored in a separate file. If LogToStdOut, all logging will instead be sent to standard output. **Default:** 'optimizedeviceconfiguration'
- **perform_device_optimization** (*bool*) – Whether to perform a relaxation using the device geometry obtained after the bulk rigid relaxation. When True, device_configuration must have a calculator. The optimization region is determined by optimization_region_center and optimization_region_length. **Default:** False

calculator()

Returns
The bulk calculator used for optimizing the geometry.

Return type
Calculator

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

deviceCalculator()

Returns
The device calculator.

Return type
Calculator

filename()

Returns
The filename where the study object is stored.

Return type
str

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizationRegionCenter()

Returns
The position along the transport direction of the center of the optimization region.

Return type
PhysicalQuantity of type length

`optimizationRegionLength()`

Returns
The length along the transport direction of the optimization region. Note that the optimization region might be capped in order to keep at least one repeat of the minimal electrode fixed on either side.

Return type
PhysicalQuantity of type length

`optimizeGeometryParameters()`

Returns
The parameters used for optimizing the geometry.

Return type
OptimizeGeometryParameters

`passivateElectrodeSurfaces()`

Returns
Whether the left and right surfaces of the central region of the device along the transport direction are passivated when extracting the central region for optimization.

Return type
tuple (size 2) of bool

`performDeviceOptimization()`

Returns
Whether an additional optimization will be performed for the device configuration obtained after the bulk rigid relaxation.

Return type
bool

`result()`

Retrieve the optimized device configuration.

Returns
The optimized device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

`saveToFileAfterUpdate()`

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

`uniqueString()`

Return a unique string representing the state of the object.

`update()`

Run the calculations for the study object.

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *OptimizeDeviceConfiguration* object can be used to optimize a device configuration using the so-called Bulk Rigid Relaxation (BRR) method. In the BRR method, after building the device configuration, the central region bulk is extracted and relaxed with *FixAtomConstraints* constraints on the atoms belonging to the left electrode extension, and *RigidBody* constraints on the atoms belonging to the right electrode extension (i.e., the central region can expand or contract along the transport direction during the optimization). The device is then reassembled from the optimized central region bulk. This is a computationally efficient approach and will usually capture a large fraction of the required geometry relaxation.

Note that the length and position of the region to be relaxed within the devices central region can be controlled with `optimization_region_length` and `optimization_region_center`. The contrained regions on either side must at least cover the electrode extensions, but might extend beyond this.

After performing the calculation, the resulting optimized device configuration can be retrieved by calling the `result` method.

Note: The BRR method does not calculate the total energy using a full device geometry, hence the reassembled device may not be exactly in the global minimum-energy geometry. Some types of electronic structure calculations may be sensitive to this difference, others may not. If you want to ensure that the reassembled device is in the minimum-energy geometry, set `perform_device_optimization=True`, and a geometry optimization will be performed for the device using the calculator on the original `device_configuration`.

Usage Examples

This example shows how to perform an optimization of a carbon nanotube device.

First, we define the device system. This is given in the following file (for clarity, we split this from the *OptimizeDeviceConfiguration* study itself):

`setup_cnt_device.py`

We can now proceed with the study. Note that this example is only intended to give an overview of the usage of *OptimizeDeviceConfiguration*. The `SingleZeta` basis set used in this example should not be used for a proper, physical device configuration optimization.

```
# Read the device configuration with an attached calculator.
device_configuration = nlread('setup_cnt_device.py', DeviceConfiguration)[0]

# Set up parameters for the device configuration optimization.
optimize_geometry_parameters = OptimizeGeometryParameters(
    max_forces=0.2 * eV / Angstrom,
    max_stress=0.8 * GPa,
    max_steps=200,
    max_step_length=0.2 * Angstrom,
    optimizer_method=LBFGS(),
)

# Set up the object and perform the relaxation.
optimize_device_configuration = OptimizeDeviceConfiguration(
```

(continues on next page)

(continued from previous page)

```
    device_configuration=device_configuration,
    filename=u'optimize_cnt_device.hdf5',
    object_id='optimizedeviceconfiguration',
    optimization_region_length=10.0 * Angstrom,
    passivate_electrode_surfaces=(True, True),
    optimize_geometry_parameters=optimize_geometry_parameters,
    log_filename_prefix='optimize_cnt_device',
)
optimize_device_configuration.update()

# Fetch the resulting device configuration after optimization.
optimized_device = optimize_device_configuration.result()

# Print a report.
nlprint(optimize_device_configuration)
```

optimize_cnt_device.py

Here is the resulting report from the `nlprint` command:

```
+-------------------------------------------------------------------------------+
| Optimize Device Configuration Report                                          |
+-------------------------------------------------------------------------------+
| Device configuration:                                                         |
+-------------------------------------------------------------------------------+
| Central region (including electrode extensions):                              |
|     Length (before optimization): 16.34 Ang                                   |
|     Length (after optimization): 16.47 Ang                                    |
|     Number of atoms: 96                                                       |
| Electrode extensions:                                                         |
|     Length (left): 7.10 Ang                                                   |
|     Length (right): 7.10 Ang                                                  |
|     Number of atoms (left): 42                                                |
|     Number of atoms (right): 42                                               |
+-------------------------------------------------------------------------------+
| Initial configuration details:                                                |
+-------------------------------------------------------------------------------+
| Optimization region:                                                          |
|     Length: 2.13 Ang                                                          |
|     Left edge position: 7.10 Ang                                              |
|     Right edge position: 9.24 Ang                                             |
|     Number of atoms: 12                                                       |
| Constrained regions:                                                          |
|     Length (left): 7.10 Ang                                                   |
|     Length (right): 7.10 Ang                                                  |
|     Number of atoms (left): 42                                                |
|     Number of atoms (right): 42                                               |
+-------------------------------------------------------------------------------+
| Bulk rigid relaxation details:                                                |
+-------------------------------------------------------------------------------+
| Optimization region:                                                          |
|     Length: 2.26 Ang                                                          |
|     Left edge position: 7.10 Ang                                              |
```

(continues on next page)

(continued from previous page)

```
|     Right edge position: 9.37 Ang                                              |
|     Number of atoms: 12                                                        |
| Constrained regions:                                                           |
|     Length (left): 7.10 Ang                                                    |
|     Length (right): 7.10 Ang                                                   |
|     Number of atoms (left): 42                                                 |
|     Number of atoms (right): 42                                                |
+--------------------------------------------------------------------------------+
| Device optimization disabled                                                   |
+--------------------------------------------------------------------------------+
```

OptimizeGeometryParameters

class OptimizeGeometryParameters(*max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *disable_stress=None*, *optimize_cell=None*, *optimizer_method=None*, *target_stress=None*, *pre_step_hook=None*, *post_step_hook=None*, *constrain_bravais_lattice=None*, *trajectory_interval=None*, *restart_strategy=None*, *enable_optimization_stop_file=None*, *write_raw_stress=None*, *write_raw_forces=None*, *compute_stress=None*)

Class for parameters used in optimizing the geometry of a configuration.

Parameters

- **max_forces** (*PhysicalQuantity* of type force) – The maximum forces determining that the optimization should stop. **Default:** 0.05*eV/Ang
- **max_stress** (*PhysicalQuantity* of type pressure) – The maximum stress determining that the optimization should stop. **Default:** 0.1*GPa
- **max_steps** (*A positive integer.*) – The maximum number of steps. **Default:** 200
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang
- **constraints** (*A list of integers and Constraints objects.*) – A list of indices of the atom with fixed positions, and Constraints objects. **Default:** []
- **trajectory_filename** (*str.*) – The filename of the file to be used for storing the trajectory in.
- **trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. **Default:** None
- **disable_stress** – **Deprecated:** from v2022.03, use `optimize_cell` parameter instead.
- **optimize_cell** (*bool*) – will be change during the optimization. Enabling the stress calculation for bulk configurations. **Default:** True for BulkConfigurations otherwise False
- **optimizer_method** (*LBFGS* | *FIRE* | *VelocityProjectionOptimization* | `ConjugateGradient`) – The optimizer to use for optimizing the structure. **Default:** LBFGS
- **target_stress** – The target stress under which the cell should be optimized. Must be given as a single pressure value, or a stress vector in Voigt or matrix notation. **Default:** 0*GPa
- **pre_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just before each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** No hook.

•**post_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just after each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** No hook.

•**constrain_bravais_lattice** (*bool*) – Enable preserving the Bravais lattice symmetry of the configuration. **Default:** True if the `target_stress` is commensurate with the lattice symmetries

•**trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1

•**restart_strategy** (`NoRestart` | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** `RestartFromTrajectory()`

•**enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the geometry optimization. If `True`, creation of the stop file will stop the optimization at the next step. The name of the stop file will be shown in the log output; it will be `stop-geometry-optimization-uniqueID`, where `uniqueID` is a randomly generated identifier for this optimization. The file must be created in the current working directory. **Default:** `True`

•**write_raw_stress** – **Deprecated:** from v2025.06, use `compute_stress` parameter instead.

•**write_raw_forces** – **Deprecated:** from v2025.06, forces are always written to trajectory.

•**compute_stress** (*bool*) – Compute stress for each optimization step. Using this flag, stress can be regardless if `optimize_cell` is disabled. Default `Automatic` corresponds to same as `optimize_cell`. **Default:** `Automatic`

computeStress()

Query method for the compute_stress parameter.

constrainBravaisLattice()

Query method for the constrain_bravais_lattice parameter.

constraints()

Query method for the constraints.

classmethod defaultOptimizeCell(*configuration_type*, *is_neb*, *neb_variable_cell=None*)

The default enable stress depend on the configuration type.

Parameters

•**configuration_type** (*type*) – The configuration type.

•**is_neb** (*bool*) – True, if neb configuration.

•**neb_variable_cell** (*bool*) – Whether this is a variable cell NEB Configuration.

Returns
True, for Bulk configurations, otherwise False.

Return type
bool

disableStress()

Query method for the disable stress.

enableOptimizationStopFile()

Query method for the enable_optimization_stop_file parameter.

maxForces()

Query method for the max forces.

maxStepLength()

Query method for the max step length.

maxSteps()

Query method for the max steps.

maxStress()

Query method for the max stress.

nlinfo()

Returns
The nlinfo.

Return type
dict

optimizeCell()

Query method for the optimize cell.

optimizerMethod()

Query method for the optimizer method.

postStepHook()

Query method for the post-step hook.

preStepHook()

Query method for the pre-step hook.

restartStrategy()

Query method for the restart_strategy parameter.

setConstrainBravaisLattice(*constrain_bravais_lattice*)

Method for setting the constrain_bravais_lattice parameter.

setConstraints(*constraints*)

Method for setting the constraints.

setDisableStress(*disable_stress*)

Method for setting the disable_stress.

setEnableOptimizationStopFile(*enable_optimization_stop_file*, *default_enable_optimization_stop_file*)

Method for setting the enable optimization stop file parameter.

setMaxForces(*max_forces*)

Method for setting the max forces.

setMaxStepLength(*max_step_length*)

Method for setting the max step length.

setMaxSteps(*max_steps*)

Method for setting the max steps.

setMaxStress(*max_stress*)

Method for setting the max stress.

setOptimizeCell(*optimize_cell*)

Method for setting the disable_stress and optimize cell.

setOptimizerMethod(*optimizer_method*)
Method for setting the optimizer method.

setPostStepHook(*post_step_hook*)
Method for setting the post-step hook.

setPreStepHook(*pre_step_hook*)
Method for setting the pre-step hook.

setRestartStrategy(*restart_strategy*, *default_restart_strategy*)
Method for setting the restart_strategy parameter.

setTargetStress(*target_stress*)
Method for setting the target stress tensor.

setTrajectoryFilename(*trajectory_filename*)
Method for setting the trajectory filename.

setTrajectoryInterval(*trajectory_interval*, *default_trajectory_interval*)
Method for setting the trajectory_interval parameter.

setTrajectoryObjectId(*trajectory_object_id*)
Method for setting the trajectory object ID.

setWriteRawStress(*write_raw_stress*)
Method for setting the write_raw_stress.

targetStress()
Query method for the target stress tensor.

trajectoryFilename()
Query method for the trajectory filename.

trajectoryInterval()
Query method for the trajectory_interval parameter.

trajectoryObjectId()
Query method for the trajectory object ID.

uniqueString()
Return a unique string representing the state of the object.

writeRawForces()
Query method to check if raw force will be written on the trajectory.

writeRawStress()
Query method to check if raw stress will be written on the trajectory.

OptimizedFFTGridSampling

class OptimizedFFTGridSampling(*energy_cutoff*, *maximum_average_prime_factor=None*, *maximum_grid_enlargement=None*)

Create a grid sampling with sizes that are optimized for FFTs.

The Fast Fourier Transform algorithm works by splitting the grid into a hierachy of chunks, and the algorithms works most efficiently when the chunk sizes are small prime numbers. A good measure of the efficiency of the FFT is the average of the prime factors of the grid size. This object tries to find a grid sampling which is at least equivalent to a provided energy cutoff with as low an average prime factor of the grid sizes as possible within some constraints.

Parameters

- **energy_cutoff** (*PhysicalQuantity* of type energy.) – The energy cutoff to be used to determine the grid sampling. The energy cutoff must be a positive energy.
- **maximum_average_prime_factor** (*float*) – The maximum value for the average prime factor of the grid points in each direction. **Default:** 5.0
- **maximum_grid_enlargement** (*float*) – The grid can at maximum be expanded by this number. The default value of 0.1 implies that the number of grid points in each direction is allowed to be 10% larger when searching for a favorable prime factorization. **Default:** 0.1

energyCutoff()

Returns
The energy cutoff.

Return type
PhysicalQuantity of type energy

maximumAveragePrimeFactor()

Returns
The maximum average prime factor.

Return type
float

maximumGridEnlargement()

Returns
The maximum grid enlargement.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()
Return a unique string representing the state of the object.

OrbitalMoment

class OrbitalMoment(*configuration*, *kpoints=None*, *temperature=None*)
Class for calculating the orbital moment for a BulkConfiguration or a MoleculeConfiguration.

Parameters

- **configuration** (*BulkConfiguration*) – The BulkConfiguration for which to calculate the inter-site coupling matrix.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points with which to calculate the orbital moment. **Default:** The k-point sampling used on the calculator.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Fermi distribution. **Default:** `300 * Kelvin`

atomResolvedOrbitalMoment()

Returns
The orbital magnetic moment for each atom.

Return type
PhysicalQuantity of Bohr magneton

kpoints()

Returns
The k-points with which to calculate the Heisenberg exchange constants.

Return type
class:~.*MonkhorstPackGrid* | *RegularKpointGrid*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

orbitalMoment()

Returns
The total orbital magnetic moment.

Return type
PhysicalQuantity of Bohr magneton

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

temperature()

Returns
The temperature for the thermal smearing of the Fermi distribution.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *OrbitalMoment* for iron.

```
# Set minimal log verbosity
setVerbosity(MinimalLog)

# %% bulk_iron

# Set up lattice
lattice = BodyCenteredCubic(2.8665*Angstrom)

# Define elements
elements = [Iron]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_iron = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )


# %% InitialSpin

scaled_spins = [
    1.0,
]
initial_spin = InitialSpin(
    scaled_spins=scaled_spins
)
nlsave('orbital_moment_results.hdf5', initial_spin)


# %% Set LCAOCalculator

# %% LCAOCalculator

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = SGGA.PBE

k_point_sampling = KpointDensity(
    density_a=7.0*Angstrom,
    density_b=7.0*Angstrom,
    density_c=7.0*Angstrom
)

numerical_accuracy_parameters = NumericalAccuracyParameters(
```

(continues on next page)

(continued from previous page)

```
    k_point_sampling=k_point_sampling
)

checkpoint_handler = NoCheckpointHandler

calculator = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    checkpoint_handler=checkpoint_handler
)


# %% Set Calculator

bulk_iron.setCalculator(
    calculator=calculator,
    initial_spin=initial_spin
)

bulk_iron.update()

nlsave('orbital_moment_results.hdf5', bulk_iron)


# %% Set LCAOCalculator

# %% LCAOCalculator

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = SOGGA.PBE

k_point_sampling = KpointDensity(
    density_a=7.0*Angstrom,
    density_b=7.0*Angstrom,
    density_c=7.0*Angstrom
)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling
)

checkpoint_handler = NoCheckpointHandler

calculator_1 = LCAOCalculator(
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    checkpoint_handler=checkpoint_handler
)
```

(continues on next page)

(continued from previous page)

```
# %% Set Calculator

bulk_iron.setCalculator(
    calculator=calculator_1,
    initial_spin=initial_spin,
    initial_state=bulk_iron
)

bulk_iron.update()

nlsave('orbital_moment_results.hdf5', bulk_iron)


# %% OrbitalMoment

kpoints = MonkhorstPackGrid(
    na=22,
    nb=22,
    nc=22
)

orbital_moment = OrbitalMoment(
    configuration=bulk_iron,
    kpoints=kpoints
)
nlsave('orbital_moment_results.hdf5', orbital_moment)
```

Notes

The orbital moment is calculated following[1] as

$$\langle \mathbf{L}_i \rangle = \mathrm{Re} \sum_{\alpha \mathbf{k}} w_{\mathbf{k}} f_{\alpha,\mathbf{k}} \sum_{\lambda\mu\nu} c^{\alpha *}_{\lambda}(\mathbf{k}) \left[\mathbf{L}_i\right]_{\lambda\mu} \mathbf{S}_{\mu\nu}(\mathbf{k}) c^{\alpha}_{\nu}(\mathbf{k})$$

where α denotes band index, $\mathbf{k}$ is a k-point with corresponding weight $w_{\mathbf{k}}$ and $f_{\alpha,\mathbf{k}}$ is the occupation factor. The second sum runs over basis function orbitals and $c^{\alpha}_{\lambda}(\mathbf{k})$ it the eigenvector component for basis orbital λ and band index α calculated at the k-point $\mathbf{k}$. The angular momentum operator is $\mathbf{L}$ and $\mathbf{S}(\mathbf{k})$ is the basis orbital overlap matrix.

OzakiContour

class OzakiContour(*number_of_poles=None*)

An equilibrium contour using the Ozaki contour points and weights defined in[1].

Parameters

number_of_poles (*int*) – The total number of poles to include in the calculation.

[1] G.ăAutes, C.ăBarreteau, D.ăSpanjaard, and M.ăDesjonqueres. Magnetism of iron: from the bulk to the monatomic wire. *Journal of Physics: Condensed Matter*, 18(29):6785–6813, jul 2006. doi:10.1088/0953-8984/18/29/018.

[1] Taisuke Ozaki. Continued fraction representation of the Fermi-Dirac function for large-scale electronic structure calculations. *Phys. Rev. B*, 75:035123, 2007. doi:10.1103/PhysRevB.75.035123.

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfPoles()

Returns
The number of poles.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Example

One can use the OzakiContour by defining it as an equilibrium contour. One has to supply the number of poles that will be used in the approximation.

```
equilibrium_contour = OzakiContour(number_of_poles=100)
```

To use it in a calculation of the equilibrium density matrix one has to set the equilibrium contour explicitly on the *ContourParameters* object,

```
contour_parameters = ContourParameters(equilibrium_contour=equilibrium_contour)
```

and save the contour_parameters on the DeviceCalculator (e.g. *DeviceLCAOCalculator*)

```
device_calculator = DeviceLCAOCalculator(contour_parameters=contour_parameters)
```

Notes

The OzakiContour presents a different approach to calculating the equilibrium density matrix D by integrating the Greens function G,

$$D = \Im\left[-\frac{2}{\pi}\int_{-\infty}^{+\infty} dE f(E-\mu)G(E+i0^{+})\right] ,$$

in which $f(E)$ is the Fermi-Dirac distribution and μ is the Fermi level.

This integral can be solved using the residue theorem:

$$\oint dz G(z)f(z) = -2\pi i k T \sum_{z_\nu} G(z_\nu)$$

in which the sum on the right hand side runs over the poles of the integrant included in the contour.

The Ozaki method tries to solve this integral by representing the Fermi-Dirac distribution as a continued fraction. If the fraction is broken off at a certain level, say M, the poles of this function can be found through the solution of a generalized eigenvalue problem of size M. The accuracy increases as more poles are included.

More information can be found in the references[Page 3538, 1],[2].

PEXSISolver

class PEXSISolver(*number_of_poles*, *fermi_level_bracket=None*, *number_of_electrons_tolerance=None*, *number_of_inertia_steps=None*, *maximum_number_of_pexsi_iterations=None*, *temperature=None*, *options=None*, *processes_per_pole=None*)

Calculate the density matrix using the PEXSI algorithm.

Parameters

- **number_of_poles** (*int*) – The number of poles to use. Must be even.
- **fermi_level_bracket** (*PhysicalQuantity* of type energy) – The bracket in which to search for the Fermi level. **Default:** (10, 10)*Hartree
- **number_of_electrons_tolerance** (*tuple of floats*) – The bracket for the tolerance for the number of electrons calculated by the PEXSI algorithm, given as (minimum, maximum). **Default:** (1.e-4, 1.0)
- **number_of_inertia_steps** (*int*) – The number of SCF steps that should use inertia counting. **Default:** 5
- **maximum_number_of_pexsi_iterations** (*int*) – The maximum number of PEXSI iterations per SCF step. **Default:** 10
- **temperature** (*PhysicalQuantity* of type temperature) – The smearing of the Fermi distribution. **Default:** 300*Kelvin
- **processes_per_pole** (*int > 0*) – The number of processes to use per pole. In case the number of processes exceeds the number of poles a larger number of processes per pole will be used. **Default:** 1

fermiLevelBracket()

Returns
The Fermi level bracket.

Return type
PhysicalQuantity of type energy

maximumNumberOfPEXSIIterations()

Returns the maximum number of PEXSI iterations per SCF cycle.

Returns
The maximum number of PEXSI iterations per SCF cycle.

Return type
int

numberOfElectronsTolerance()

Returns
The tolerance for the number of electrons as (final, initial) tolerance.

Return type
tuple of floats

numberOfInertiaSteps()

Returns
The maximum number of PEXSI iterations per SCF cycle.

[2] T.ăOzaki, K.ăNishio, and H.ăKino. Efficient implementation of the nonequilibrium green function method for electronic transport calculations. *Phys. Rev. B*, 81:035116, 2010. doi:10.1103/PhysRevB.81.035116.

Return type
int

numberOfPoles()

Returns
The number of poles.

Return type
int

processesPerPole()

Returns
The number of processes to use per pole.

Return type
int

temperature()

Returns
The temperature of the Fermi distribution.

Return type
PhysicalQuantity of type temperature

uniqueString()
Return a unique string representing the state of the object.

Usage Example

Define a density matrix method that uses the PEXSI solver, supplying the number of poles to use in the pole expansion.

```
density_matrix_method = PEXSISolver(number_of_poles=64)
```

For more control, one can specify various other parameters.

```
density_matrix_method = PEXSISolver(
    number_of_poles=64,
    fermi_level_bracket=(-10.0, 10.0) * Hartree,
    number_of_electrons_tolerance=(1.e-4, 1.0),
    number_of_inertia_steps=5,
    maximum_number_of_pexsi_iterations=10)
```

To use the PEXSISolver when using equivalent bulk to generate the initial density for a device calculation, set it on the *AlgorithmParameters* passed to the *EquivalentBulk* object.

```
algorithm_parameters = AlgorithmParameters(
    density_matrix_method=PEXSISolver(number_of_poles=64))

initial_density_type = EquivalentBulk(algorithm_parameters=algorithm_parameters)

device_algorithm_parameters = DeviceAlgorithmParameters(
    initial_density_type=initial_density_type)
```

(continues on next page)

(continued from previous page)

```
device_calculator = DeviceLCAOCalculator(
    device_algorithm_parameters=device_algorithm_parameters)
```

Notes

The implemented pole expansion and selective inversion (PEXSI) method is based on the work by Lin et al.,[1],[2].

The method calculates the density matrix D by means of a pole expansion

$$D = \Im \sum_{l=1}^{P} \omega_l(\beta) \left(H - (z_l + \mu)S\right)^{-1}$$

with expansion weights ω_l as a function of the inverse temperature β and the shifts z_l. The method calculates only the required elements $(H - (z_l + \mu)S)^{-1}_{ij}$ using selective inversion.

Note that for the moment, the implemented method is only usable for the Γ-point and unpolarized calculations.

The computational scaling of this method depends on the dimensionality of the system. For quasi-1D systems like nanotubes, it scales linearly with regards to the number of electrons, i.e. $\mathcal{O}(N_e)$. For quasi-2D systems it scales with $\mathcal{O}(N_e^{1.5})$, and for general 3D systems it scales with $\mathcal{O}(N_e^2)$.

The implemented method is multi-level parallel: On the first level, it parallelizes over all poles in the pole expansion. On the second level, it can also distribute the calculation of each pole over multiple processors. This distribution is done automatically, as soon as the calculation runs on enough processors.

The PEXSI algorithm can be roughly divided into two parts. First, an inertia counting scheme is followed to identify a trial Fermi level within the supplied `fermi_level_bracket`. Afterwards, the actual pole expansion is done to calculate the density matrix for this trial Fermi level. From this, the number of electrons corresponding to this density matrix can be calculated, allowing the trial Fermi level to be updated once more by doing a Newton step. This PEXSI cycle is repeated at most `maximum_number_of_pexsi_iterations` times, until the calculated number of electrons corresponds to the exact number of electrons within the current tolerance as explained below. After `number_of_inertia_steps` SCF steps have elapsed, subsequent steps will not use the inertia counting scheme anymore, as by this point the calculated Fermi level should be close to the true Fermi level.

In this implementation of the PEXSI method, the tolerance on $\Delta N_e(\mu)$ - which is the deviation from the required number of electrons for a given trial Fermi level - is adapted dynamically based on the current state of the SCF cycle. In the first iteration, the tolerance starts out at the maximum, which is the upper bound of the given `number_of_electrons_tolerance` bracket. After each iteration, the largest absolute change in the density matrix is computed, and the tolerance for the following iteration is set to that value, with the lower bound of the tolerance as a minimum. By following this scheme, a SCF iteration typically only requires very few PEXSI iterations.

[1] Lin Lin, Jianfeng Lu, Lexing Ying, Roberto Car, and Weinan E. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Commun. Math. Sci.*, 7(3):755–777, 09 2009.

[2] Lin Lin, Mohan Chen, Chao Yang, and Lixin He. Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion. *Journal of Physics: Condensed Matter*, 25(29):295501, 2013. doi:10.1088/0953-8984/25/29/295501.

PPCGSolver

class PPCGSolver(*absolute_tolerance=None*, *relative_tolerance=None*, *maximum_number_of_iterations=None*, *block_size=None*, *rr_period=None*, *buffer_states=None*, *initialization_method=None*)

Density matrix method that uses a blocked preconditioned conjugate gradient algorithm to improve on an initial guess for the eigenstates of the Hamiltonian.

Parameters

- **absolute_tolerance** (*float*) – The absolute tolerance up to which the residuals have to be converged for the algorithm to end. **Default:** 1.0e-8
- **relative_tolerance** (*float*) – The tolerance up to which the residuals have to be converged relative to the original value of the residuals. **Default:** 1.0e-99
- **maximum_number_of_iterations** – The maximum number of iterations to take. Given as a non-negative integer. **Default:** 2
- **block_size** (*int*) – The size of the inner diagonalization problem that is to be solved i.e. how many states are coupled during the update in each iteration. Given as a positive integer. **Default:** 5
- **rr_period** (*int*) – The Rayleigh-Ritz period, i.e. every how many iterations we perform a full subspace diagonalization. Given as a positive integer. **Default:** maximum_number_of_iterations + 1
- **buffer_states** (*float*) – Buffer states are states that are used in the optimization algorithm, but not used to check the convergence. They add stability to the algorithm if there are many near degenerate states and speed up the convergence of lower lying states. Given as a fraction of the number of occupied states between 0 and 1 (not inclusive). **Default:** 0.05
- **initialization_method** (*RandomBlochWaveInitialization* | *BasisSetInitialization*) – The method used to obtain the initial states for the Generalized Davidson algorithm, see *Initialization methods*. **Default:** BasisSetInitialization()

absoluteTolerance()

Returns
The absolute tolerance up to which the residuals have to be converged for the algorithm to end.

Return type
float

blockSize()

Returns
The size of the inner diagonalization problem that is to be solved. (i.e. How many states are coupled during the update in each iteration)

Return type
int

bufferStates()

Returns
The number of buffer states, given as a fraction of the number of occupied states. Buffer states are states that are used in the optimization algorithm, but not used to check the convergence. They add stability to the algorithm if there are many near degenerate states and speed up the convergence of lower lying states.

Return type
float

initializationMethod()

Returns
The method used to obtain the initial states for the residual minimizer.

Return type
RandomBlochWaveInitialization | BasisSetInitialization

maximumNumberOfIterations()

Returns
The maximum number of iterations for the ppcg algorithm.

Return type
int

relativeTolerance()

Returns
The amount the residual of the states has to be reduced compared to the residual of the initial guess.

Return type
float

rrPeriod()

Returns
The Rayleigh-Ritz period, i.e. every how many iterations we perform a full subspace diagonalization.

Return type
int

uniqueString()
Return a unique string representing the state of the object.

Usage Example

Define a density matrix method that uses the default *PPCGSolver*:

```
algorithm_parameters = AlgorithmParameters(
   density_matrix_method=PPCGSolver())
```

This will apply the settings we have found to generally lead to the fastest convergence of the SCF loop.

A more robust eigensolver can be defined, e.g.:

```
density_matrix_method = PPCGSolver(
    relative_tolerance=0.001,
    absolute_tolerance=1.0e-7,
    maximal_number_of_iterations=100,
    block_size=100,
    rr_period=2)

algorithm_parameters = AlgorithmParameters(
   density_matrix_method=density_matrix_method)
```

By increasing the `block_size` we slow down the algorithm but make the inner iterations more accurate. The parameters `rr_period` determines after how many iterations a subspace diagonalization is performed. By default this is

chosen to be larger than the number of iterations so a subspace diagonalization will only be performed at the end. By doing this multiple times we can make the algorithm more stable.

A different initial guess for the states can be chosen by setting the `initialization_method`:

```
density_matrix_method = PPCGSolver(
    initialization_method=RandomBlochWaveInitialization(),
)

algorithm_parameters = AlgorithmParameters(
    density_matrix_method=density_matrix_method)
```

Notes

Algorithm

The *PPCGSolver* class implements the projected preconditioned conjugate gradient algorithm (PPCG) as explained in[1].

This method is a combination of a generalized davidson and a conjugate gradient method. The method has a better parallel scaling for large systems. In the example figures *Absolute Scaling* and *Relative Scaling* we compare the performance and scaling of the PPCG algorithm against our default Generalized Davidson solver for a pristine 128 Si FCC system at the Gamma point.

Figure *Absolute Scaling* shows the wall time for the number of processes going from 1 to 32. Figure *Relative Scaling* shows the speedup compared to the calculation run in serial.

Absolute Scaling

Relative Scaling

PairPotential

class PairPotential(*element_pair*, *distances*, *values*)

Input class for setting the repulsive pair potential between two elements in the *HuckelCalculator*.

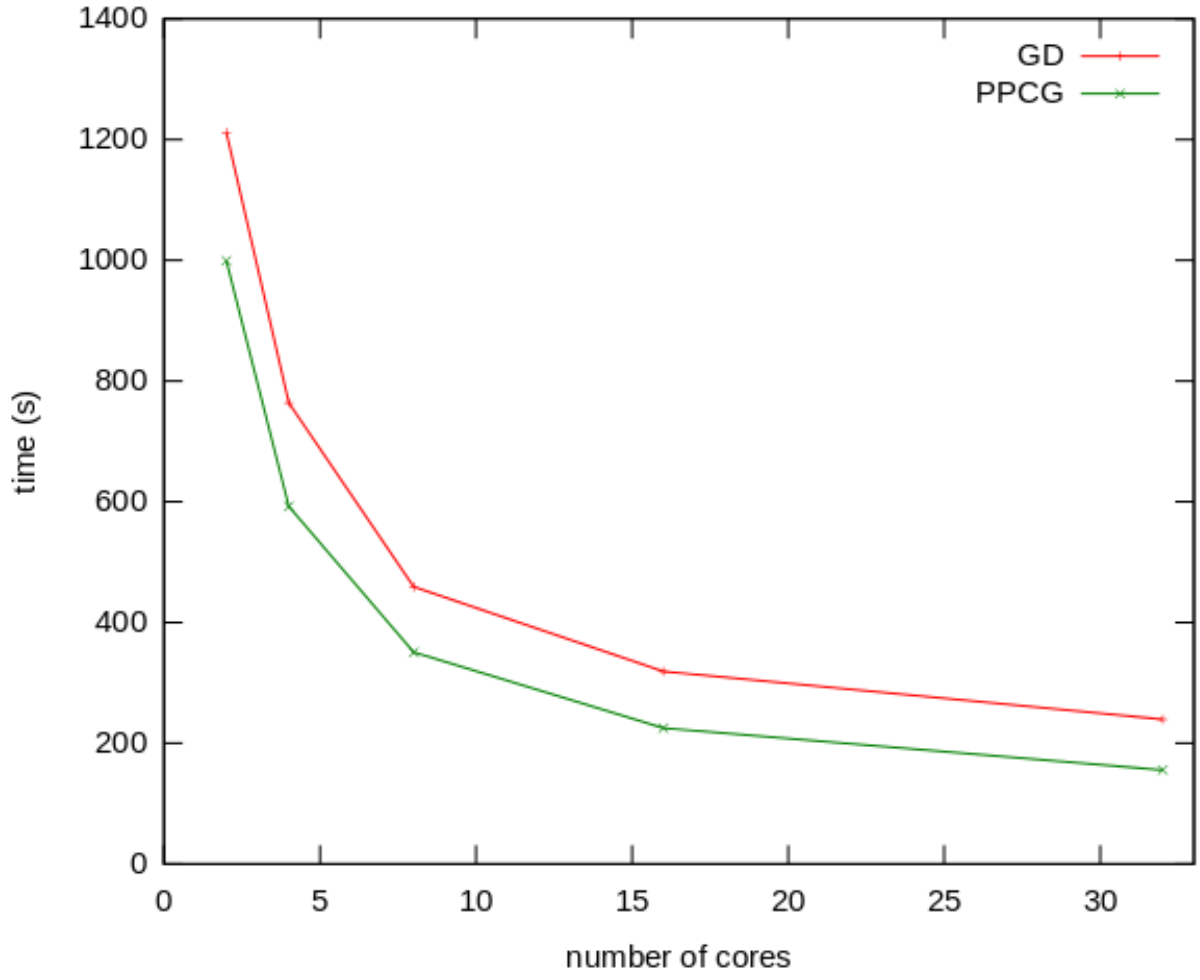
Parameters

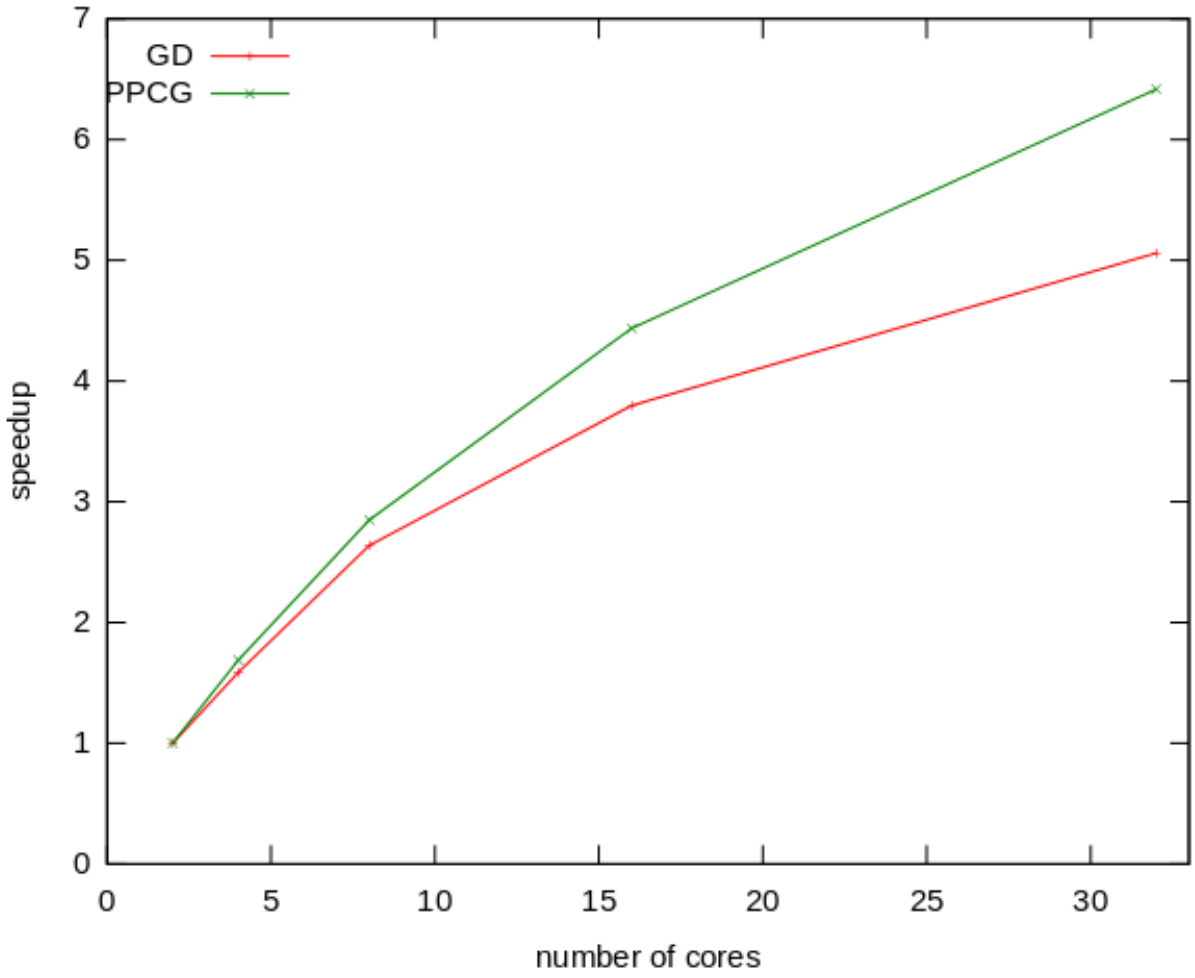
- **element_pair** (tuple of *PeriodicTableElement*) – The two elements to set the pair potential for.
- **distances** (*PhysicalQuantity* of type length) – A list of distances for the pair potential. The length of this vector should be len(values).
- **values** (*PhysicalQuantity* of type energy) – The values of the pair potential at the given distances. The length of this vector should be len(distances).

distances()

Returns
The distances of the pair potential.

[1] Eugene Vecharynski, Chao Yang, and JohnăE. Pask. A projected preconditioned conjugate gradient algorithm for computing many extreme eigenpairs of a Hermitian matrix. *Journal of Computational Physics*, 290:73–89, 2015. URL: http://www.sciencedirect.com/science/article/pii/S002199911500100X, doi:10.1016/j.jcp.2015.02.030.


1400
1200
1000
800
600
400
200
0
time (s)
0
5
10
15
20
25
30
number of cores
GD
PPCG


GD
PPCG
speedup
number of cores
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30

Return type
PhysicalQuantity of type length

elementPair()

Returns
The element pair of the pair potential.

Return type
tuple of *PeriodicTableElement*

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

values()

Returns
The values of the pair potential.

Return type
PhysicalQuantity of type energy

Usage Examples

Make a repulsive exponential pair potential between two carbon atoms

```
# Define a cutoff distance
rc = 4.

# Define a list of distances
distances = numpy.linspace(0.0,rc,401)

# Evaluate an exponential function at the distance
# (subtract the value in the endpoint to make it go to 0)
values = 50.*(numpy.exp(-3.*distances)-numpy.exp(-3.*rc))

# Setup the pair potential object
pair_potential = PairPotential(
    element_pair = (Carbon, Carbon),
    distances = distances * Bohr,
    values = values * eV,
    )
```

Notes

The *PairPotential* is used to define a pair wise interaction between two elements. It is used to define the repulsive term for *HuckelCalculator*, *DeviceHuckelCalculator*, *SlaterKosterCalculator* and *DeviceSlaterKosterCalculator*.

ParallelConjugateGradientSolver

class ParallelConjugateGradientSolver(*boundary_conditions=None*, *solvent_dielectric_constant=None*, *gpu_acceleration=None*)

The parallel conjugate gradient-based Poisson solver.

Parameters

- **boundary_conditions** (list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*) – A list of shape (3,2) specifying the boundary conditions on the 6 surfaces of the unit cell for the configuration. **Default:**

– *MoleculeConfiguration*: `[[MultipoleBoundaryCondition()] * 2] * 3`

– *BulkConfiguration*: `[[PeriodicBoundaryCondition()] * 2] * 3`

– *SurfaceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition(), NeumannBoundaryCondition()]]
```

– *DeviceConfiguration*:

```
[[PeriodicBoundaryCondition()] * 2,
 [PeriodicBoundaryCondition()] * 2,
 [DirichletBoundaryCondition()] * 2]
```

- **solvent_dielectric_constant** (*float*) – The dielectric constant of the media surrounding the configuration. Must be a number larger than 0. **Default:** `1.0`
- **gpu_acceleration** (`Enabled` | `Disabled` | `Automatic`) – Whether to use GPU acceleration. If set to `Automatic`, the solver will use the GPU if available. **Default:** `Automatic`

boundaryConditions()

Returns
The boundary conditions for the solver.

Return type
list of *DirichletBoundaryCondition* | *NeumannBoundaryCondition* | *PeriodicBoundaryCondition* | *MultipoleBoundaryCondition*

gpuAcceleration()

Returns
The GPU acceleration flag.

Return type
`Enabled` | `Disabled` | `Automatic`

nlinfo()

Returns
The nlinfo.

Return type
dict

solventDielectricConstant()

Returns
The solvent dielectric constant.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a *ParallelConjugateGradientSolver* with specified boundary conditions on the 6 faces:

```
poisson_solver = ParallelConjugateGradientSolver(
    boundary_conditions=[
        [DirichletBoundaryCondition(), NeumannBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Define a *ParallelConjugateGradientSolver* with specified boundary conditions in the A, B, and C-direction, but similar boundary conditions on opposite faces:

```
poisson_solver = ParallelConjugateGradientSolver(
    boundary_conditions=[
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
        [PeriodicBoundaryCondition(), PeriodicBoundaryCondition()]
    ]
)

calculator = LCAOCalculator(
    poisson_solver=poisson_solver
)
```

Notes

•By setting the `solvent_dielectric_constant` it is possible to perform calculations of solvents. In this case the volume of the configuration is defined by inscribing each atom in a sphere of size given by the van der Waals radius of the element. Inside the volume of the configuration the dielectric constant is 1, outside the volume of the configuration the dielectric constant is equal to the value of `solvent_dielectric_constant`.

•See also *Poisson solvers*.

•`ParallelConjugateGradientSolver` supports GPU acceleration. See *GPU acceleration of proprietary feature calculations*.

ParallelDevicePerformanceProfile

class ParallelDevicePerformanceProfile(*configuration*, *processes_and_threads*, *equilibrium_methods=None*, *non_equilibrium_methods=None*)

Class for performing timing and memory profiles of the different methods available for calculating the Greens function and lesser Greens function.

Parameters

•**configuration** (*DeviceConfiguration*) – The device configuration with an attached calculator to profile.

•**processes_and_threads** (*list of tuples of int*) – The configurations of number of processes and threads to run, as list of tuples. E.g., [(1, 1), (2, 4)] will run a calculation with 1 MPI process and 1 thread per process, and a calculation with 2 MPI processes and 4 threads per process.

•**equilibrium_methods** (GreensFunction | SparseGreensFunction | sequence of (GreensFunction | SparseGreensFunction)) – The methods benchmarked for the equilibrium calculation. If no methods should be benchmarked for the equilibrium calculation, an empty list can be specified. **Default:**: (GreensFunction, SparseGreensFunction)

•**non_equilibrium_methods** – The methods benchmarked for the non-equilibrium calculation. If no methods should be benchmarked for the non-equilibrium calculation, an empty list can be specified. **Default:**: (GreensFunction, SparseGreensFunction)

equilibriumMethods()

Returns
The equilibrium methods profiled.

Return type
tuple of (GreensFunction | SparseGreensFunction)

generateScript(*temporary_filename*)

Generate a script to run a DevicePerformanceProfile.

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print out the profiling report.

Parameters
stream (*file*) – The stream to write to. This stream must support strings being written to it using write. **Default:** sys.stdout

nonEquilibriumMethods()

Returns
The non-equilibrium methods profiled.

Return type
tuple of (GreensFunction | SparseGreensFunction)

processesAndThreads()

Returns
The list of processes and threads configurations profiled.

Return type
tuple of (int, int)

runAsSubProcess()

Run a serie of DevicePerformanceProfile for each combination of processes and threads provided.

uniqueString()

Return a unique string representing the state of the object.

Notes

This object extends the functionalities of *DevicePerformanceProfile* by running several profiles for different parallel settings, defined by the input parameter `processes_and_threads`. The aim is to help the user to to determine the best combination of parallelization strategy and Greens function algorithm for the simulation of a given device, without the need to run *DevicePerformanceProfile* several times.

The input parameters are the same as in *DevicePerformanceProfile*, except for `processes_and_threads`. This parameter determines on how many processes the contour point calculation is parallelized (mimicking the usage of `processes_per_contour_point`), and how many threads per process are utilized.

For each method and each processes and threads configuration the elapsed time and memory usage are reported.

Note: The profiler has to be launched in serial mode (e.g. without using `mpiexec`, or using `mpiexec -n 1`). The parallel runs are spawned by the profiler itself. In order to obtain reliable results, there should be enough resources available for the spawned processes. I.e., the number of available physical cores should be at least equal to the largest sum of processes and threads. For example, if 2 processes and 2 threads are defined in `processes_and_threads`, then 4 physical cores should be available. If fewer cores are available, the results may be unreliable because the cores will be over-occupied. Note that some clusters do not support process spawning. On such systems *ParallelDevicePerformanceProfile* will not work.

Usage Example

The following script will read a configuration from file and run a profile for 3 different combinations of processes and threads:

(1, 1): A single process per contour point and a single thread.

(1, 2): A single process per contour point and two threads per process.

(2, 1): Two processes per contour point and a single threads per process.

```
device_configuration = nlread('device.hdf5', DeviceConfiguration)[-1]

profile = ParallelDevicePerformanceProfile(
    device_configuration,
    processes_and_threads=[(1, 1), (1, 2), (2, 1)])
```

(continues on next page)

(continued from previous page)

```
nlprint(profile)
```

Here is an example of the output you might get, divided in different sections. First we see an header listing the available profiles corresponding to the different entries of `processes_and_threads`:

```
+-------------------------------------------------------------------------------+
| Parallel Device Performance Profile                                           |
|                                                                               |
| 3 available device performance profiles:                                      |
|        1 process          1 thread                                            |
|        1 process          2 threads                                           |
|        2 processes        1 thread                                            |
+-------------------------------------------------------------------------------+
```

It follows a detailed report of memory and timing for each profile, with the same structure as the output of *DevicePerformanceProfile*.

```
+-------------------------------------------------------------------------------+
| Device Performance Profile (1)                                                |
|   1 process       1 thread                                                    |
+-------------------------------------------------------------------------------+
| Contour point timing (s):                                                     |
|                                   EQ        NEQ                               |
| GreensFunction                 12.46      18.41                              |
| SparseGreensFunction           45.76      32.15                              |
|                                                                               |
| Fastest EQ method (by 3.7 times): GreensFunction                              |
| Fastest NEQ method (by 1.7 times): GreensFunction                             |
+-------------------------------------------------------------------------------+
| Peak memory usage/process (MB):                                               |
|                                   EQ        NEQ                               |
| GreensFunction               1901.38    3966.34                             |
| SparseGreensFunction         3536.50    2596.16                             |
|                                                                               |
| Most memory-efficient EQ method (by 1.9 times): GreensFunction                |
| Most memory-efficient NEQ method (by 1.5 times): SparseGreensFunction         |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Device Performance Profile (2)                                                |
|   1 process       2 threads                                                   |
+-------------------------------------------------------------------------------+
| Contour point timing (s):                                                     |
|                                   EQ        NEQ                               |
| GreensFunction                  8.24      13.19                               |
| SparseGreensFunction           39.78      30.08                              |
|                                                                               |
| Fastest EQ method (by 4.8 times): GreensFunction                              |
| Fastest NEQ method (by 2.3 times): GreensFunction                             |
+-------------------------------------------------------------------------------+
| Peak memory usage/process (MB):                                               |
|                                   EQ        NEQ                               |
```

(continues on next page)

(continued from previous page)

```
| GreensFunction               1887.05    3985.32                                |
| SparseGreensFunction         3525.32    2629.77                                |
|                                                                                |
| Most memory-efficient EQ method (by 1.9 times): GreensFunction                 |
| Most memory-efficient NEQ method (by 1.5 times): SparseGreensFunction          |
+--------------------------------------------------------------------------------+
+--------------------------------------------------------------------------------+
| Device Performance Profile (3)                                                 |
|   2 processes     1 thread                                                     |
+--------------------------------------------------------------------------------+
| Contour point timing (s):                                                      |
|                                  EQ        NEQ                                 |
| GreensFunction                  9.11      21.22                                |
| SparseGreensFunction           39.42      34.22                                |
|                                                                                |
| Fastest EQ method (by 4.3 times): GreensFunction                               |
| Fastest NEQ method (by 1.6 times): GreensFunction                              |
+--------------------------------------------------------------------------------+
| Peak memory usage/process (MB):                                                |
|                                  EQ        NEQ                                 |
| GreensFunction               1450.88    3919.13                              |
| SparseGreensFunction         2224.71    1982.50                              |
|                                                                                |
| Most memory-efficient EQ method (by 1.5 times): GreensFunction                 |
| Most memory-efficient NEQ method (by 2.0 times): SparseGreensFunction          |
+--------------------------------------------------------------------------------+
```

At the bottom we have a summary of time and memory consumption for the different processes and threads configurations.

```
+--------------------------------------------------------------------------------+
| Summary Report                                                                 |
|                                                                                |
| The best and worst case scenario for resource occupation (time and peak        |
| memory) is reported for each method. The resource occupation is normalized     |
| to the number of physical cores utilized, here referred to as Processing       |
| Units (PU).                                                                    |
+--------------------------------------------------------------------------------+
| Equilibrium GreensFunction:                                                    |
|                                    Time*PU (s)     Memory/PU (MB)              |
|  1 process    1 thread  (1 PU)    12.46 (best)    1901.38 (worst)             |
|  1 process    2 threads (2 PU)    16.47            943.52 (best)              |
|  2 processes  1 thread  (2 PU)    18.23 (worst)   1450.88                     |
+--------------------------------------------------------------------------------+
| Equilibrium SparseGreensFunction:                                              |
|                                    Time*PU (s)     Memory/PU (MB)              |
|  1 process    1 thread  (1 PU)    45.76 (best)    3536.50 (worst)             |
|  1 process    2 threads (2 PU)    79.57 (worst)   1762.66 (best)              |
|  2 processes  1 thread  (2 PU)    78.83           2224.71                     |
+--------------------------------------------------------------------------------+
| Non-equilibrium GreensFunction:                                                |
|                                    Time*PU (s)     Memory/PU (MB)              |
```

(continues on next page)

(continued from previous page)

```
|  1 process    1 thread  (1 PU)    18.41 (best)    3966.34 (worst)            |
|  1 process    2 threads (2 PU)    26.38           1992.66 (best)             |
|  2 processes  1 thread  (2 PU)    42.44 (worst)   3919.13                    |
+------------------------------------------------------------------------------+
| Non-equilibrium SparseGreensFunction:                                        |
|                                    Time*PU (s)     Memory/PU (MB)            |
|  1 process    1 thread  (1 PU)    32.15 (best)    2596.16 (worst)            |
|  1 process    2 threads (2 PU)    60.15           1314.88 (best)             |
|  2 processes  1 thread  (2 PU)    68.44 (worst)   1982.50                    |
+------------------------------------------------------------------------------+
```

In the summary both the memory and time are normalized to the number of processing units (PU) utilized. The number of PUs is defined as the sum between the number of processes and the number of threads and it can be interpreted simply as the number of physical cores utilized.

On the right column the peak memory per PU is reported. On the left column the total CPU time, defined as wallclock time times the number of PUs utilized, is indicated.

Interpreting the Summary Report

The real simulation wallclock time and memory consumption depends on how many contour points can be simultaneously calculated, and it is not possible to give a general estimate upfront. The user should be able to interpret the quantities reported in the summary, especially when comparing different number of PUs. Lets consider as an example an equilibrium calculation with `GreensFunction` method. From the summary we read

```
+------------------------------------------------------------------------------+
| Equilibrium GreensFunction:                                                  |
|                                    Time*PU (s)     Memory/PU (MB)            |
|  1 process    1 thread  (1 PU)    12.46 (best)    1901.38 (worst)            |
|  1 process    2 threads (2 PU)    16.47            943.52 (best)             |
|  2 processes  1 thread  (2 PU)    18.23 (worst)   1450.88                    |
+------------------------------------------------------------------------------+
```

Assume that 48 contour points will be calculated, and that the calculation will run on 48 physical cores. In this case a contour integration will take roughly as much as indicated in the `Time*PU` field (i.e. 12.46 seconds for 1 process and 1 thread, 16.47 s for 1 process and 2 threads, etc.). The total memory usage will be given multiplying the quantities in the memory column by 48, because all PUs will be engaged in the calculation. In this case the indication of `best` and `worst` time and memory are faithful.

But what if we have for example 96 physical cores available? In this case the calculation launched with 1 process and 1 threads per contour point will have some resources kept idle, because it will utilize at most one physical core per contour point. The wallclock time will be still approximately 12.46 s with no speedup, and similarly the total memory will be still given by (1901*48)MB.

Both configurations with 2 PUs will be able to run all contour points simultaneously, thus utilizing twice the number of physical cores with respect to the case with 1 PUs. Therefore the wallclock time will be approximately half the time in the `Time*PU` column (i.e., 8.2 s for 1 process, 2 threads and 9.1 for 2 processes, 1 thread). The total memory consumption will be (943*96)MB for the 1 process, 2 threads case, and (1451*96)MB for the 2 processes, 1 thread case. In this case, using 2 physical cores per contour point is advantageous.

ParallelParameters

class ParallelParameters(*processes_per_neb_image=None*, *processes_per_individual=None*, *processes_per_bias_point=None*, *processes_per_saddle_search=None*)

Class for representing the parallel parameters for a calculation.

Parameters

- **processes_per_neb_image** (*int*) – The number of processes that should be used to calculate each NEB image. **Default:** Automatically determine the number of processes that should be used to calculate a single configuration in parallel based upon the total number of MPI processes.
- **processes_per_individual** (*int*) – The number of processes that should be used to calculate each individual in the global optimization calculation. **Default:** Automatically determine the number of processes that should be used to calculate a single configuration in parallel based upon the total number of MPI processes.
- **processes_per_bias_point** (*int*) – The number of processes that should be used to calculate each bias point in an IVCurve calculation. **Default:** Automatically determine the number of processes that should be used to calculate a single configuration in parallel based upon the total number of MPI processes.
- **processes_per_saddle_search** (*int*) – The number of processes that should be used in each saddle search in an AKMC simulation. **Default:** 1

Note: The `processes_per_neb_image` argument can be set on the command line with the option `--processes-per-neb-image=<number>`. The number of processes per configuration specified on the command line takes precedence over the number specified in the constructor.

---

nlinfo()

Returns
The nlinfo.

Return type
dict

processesPerBiasPoint()

Returns
The number of processes that should be used to calculate each bias point in an IVCurve calculation.

Return type
int

processesPerIndividual()

Returns
The number of processes to use per individual in the global optimization routines. A value of None means that the number will be determined automatically.

Return type
int | None

processesPerNebImage()

Returns
The number of processes to use per NEB image. A value of None means that the number will be determined automatically.

Return type
int | None

processesPerSaddleSearch()

Returns
The number of processes to use per saddle search in an AKMC simulation.

Return type
int

classmethod processesPerXAlternatives()

Returns
Options list of pairs (values, names).

Return type
tuple

setProcessesPerNebImage(*processes_per_neb_image*)

Parameters
processes_per_neb_image (*int | None TYPE*) – Set the number of processes to use per NEB image. A value of None means that the number will be determined automatically.

uniqueString()

Return a unique string representing the state of the object.

ParameterGrid

class ParameterGrid(*spatial_regions=None*, *function=None*, *configuration=None*)

Provides an interface to setup a scalar field filled with custom values.

Parameters

- **spatial_regions** (*array of SpatialRegion objects*) – The spatial regions in which to set the c parameter. **Default:** None
- **function** (*function*) – A python function. Takes one keyword argument position which has to be a triple of coordinates (x,y,z) in units of length. Function returns a unitless scalar (float). E.g function=(lambda position : sum(position.inUnitsOf(Bohr)**2)). **Default:** None
- **configuration** (*MoleculeConfiguration | BulkConfiguration | DeviceConfiguration | SurfaceConfiguration*) – The configuration based on which to construct the grid. Must have a calculator attached. **Default:** None

PartialCharges

class PartialCharges(*configuration*)

Class for calculating the partial charges of a configuration with an attached classical calculator.

Parameters
configuration (*BulkConfiguration*) – The configuration for which the partial charges should be calculated.

evaluate(*atom_index=None*)

Return the partial charges of the selected atom or all atoms.

Parameters
atom_index (*int*) – Index of the selected atom. **Default:** All atoms.

Returns
A single float if *atom_index* is not None, otherwise a list of floats.

Return type
float | list of float

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

potentialName()
Return the name of the TremoloX potential for which the partial charges have been calculated.

Returns
The name of the TemoloX potential.

Return type
str

setMetatext(*metatext*)
Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Analyze the partial charges of a quartz crystal that are used in the *TremoloXCalculator* with a *ReaxFFPotential*:

```
# ------------------------------------------------------------
# Set up a SiO2-quartz crystal.
# ------------------------------------------------------------
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

fractional_coordinates = [[ 0.4697,  0.0000,  0.00000 ],
                          [ 0.0000,  0.4697,  0.66667 ],
                          [ 0.5303,  0.5303,  0.33333 ],
                          [ 0.4135,  0.2669,  0.11910 ],
                          [ 0.2669,  0.4135,  0.547567],
```

(continues on next page)

(continued from previous page)

```
                           [ 0.7331,  0.1466,  0.785767],
                           [ 0.5865,  0.8534,  0.214233],
                           [ 0.8534,  0.5865,  0.452433],
                           [ 0.1466,  0.7331,  0.8809  ]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define the TremoloX-calculator with a ReaxFF-potential.
potentialSet = ReaxFF_CHONSiPtZrYBaTi_2013()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Set up the PartialCharges analysis object.
partial_charges = PartialCharges(bulk_configuration)

# Print the partial charges to the log file.
nlprint(partial_charges)

# Extract and store the partial charges into an array.
q = partial_charges.evaluate().inUnitsOf(elementary_charge)
```

partial_charges.py

Notes

The *PartialCharges* analysis object can only be used with the *TremoloXCalculator*. It returns the partial charges that are used by the coulomb solver (e.g. the *CoulombDSF* method) of the selected *TremoloXPotentialSet*.

If a *ReaxFFPotential* is used, where the partial charges are calculated via the charge equilibriation (QEq) method, the QEq method is invoked when the *PartialCharges* object is initialized.

PartialElectronDensity

class PartialElectronDensity(*configuration*, *kpoints=None*, *band_indices=None*, *energy_range=None*, *spectrum_method=None*, *density_mesh_cutoff=None*)

Class for calculating the partial electron density.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the partial electron density.
- **kpoints** – The k-points for which to perform the calculation. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.

- **band_indices** (*list of int*) – The band indices of the Bloch states to include in the partial density. **Default:** All occupied bands
- **energy_range** (list of two *PhysicalQuantity* of type energy.) – Specifies the band energy range, relative to the Fermi level, of the bands to include in the partial density. Needs to be an interval or None. **Default:** All energies below the Fermi level.
- **spectrum_method** (GaussianBroadening) – The method to use for calculating the partial electron density. **Default:** GaussianBroadening(0.1*eV)
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

bandIndices()

Returns
The band indicies to include in the partial electron density.

Return type
list of int.

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity of type length^{-4}

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

energyRange()

Returns
The band energy range, relative to the Fermi level, of the bands to include in the partial density.

Return type
list of two *PhysicalQuantity* of type energy.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Sum`, `Spin.X`, `Spin.Y`, `Spin.Z`) components is returned.

Return type
PhysicalQuantity of type length^{-3}

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

kpoints()

Returns
The kpoints given as input.

Return type
MonkhorstPackGrid | *RegularKpointGrid* | numpy.ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print a string containing an ASCII table useful for plotting the partial electron density.

Parameters
stream (*file-like*) – The stream to write to. This should be an object that supports strings being written to it using a *write* method. **Default:** `sys.stdout`

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** Spin.All

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Analyze the partial electron density of a silver (111) surface. This structure is characterized by a surface state around the Γ-point. In the example, the kpoint sampling is specified with a *RegularKpointGrid* in a kpoint region around the Γ-point.

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [2.88903, 0.0, 0.0]*Angstrom
vector_b = [-1.44451, 2.50197, 0.0]*Angstrom
vector_c = [0.0, 0.0, 40.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver]

# Define coordinates
fractional_coordinates = [[ 0.749999669084,  0.5           ,  0.175654000711],
                          [ 0.749999669084,  0.5           ,  0.352570000323],
                          [ 0.749999669084,  0.5           ,  0.529485999935],
                          [ 0.749999669084,  0.5           ,  0.706401999547],
                          [ 0.416666424719,  0.833333341397,  0.234626000582],
                          [ 0.416666424719,  0.833333341397,  0.411542000194],
                          [ 0.416666424719,  0.833333341397,  0.588457999806],
                          [ 0.416666424719,  0.833333341397,  0.765373999418],
                          [ 0.083334237113,  0.166666658603,  0.293598000453],
                          [ 0.083334237113,  0.166666658603,  0.470514000065],
                          [ 0.083334237113,  0.166666658603,  0.647429999677],
                          [ 0.083334237113,  0.166666658603,  0.824345999289]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#-------------------------------------
# Basis Set
#-------------------------------------
basis_set = [
    LDABasis.Silver_SingleZetaPolarized,
    ]

#-------------------------------------
# Exchange-Correlation
```

(continues on next page)

(continued from previous page)

```
#----------------------------------------
exchange_correlation = LDA.PZ

k_point_sampling = MonkhorstPackGrid(
    na=11,
    nb=11,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=45.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------
# Partial Electron Density
# ------------------------------------------------------------
kpoints = RegularKpointGrid(
    ka_range=[-0.05, 0.05],
    kb_range=[-0.05, 0.05],
    kc_range=[0.0, 0.0],
    na=5,
    nb=5,
    nc=1,
    )

partial_electron_density = PartialElectronDensity(
    configuration=bulk_configuration,
    kpoints=kpoints,
    energy_range=[-0.5, 0.5]*eV,
    band_indices=All,
    spectrum_method=GaussianBroadening(0.1*eV),
    )
```

partial_electron_density.py

Notes

The *PartialElectronDensity* can be calculated by specifying the

1. `energy_range`. Only electronic states with energies in the specified energy range $[E_{min}; E_{max}]$ will contribute to the partial electron density. The Fermi occupation factor used to construct the full electron density is not taken into account for the partial electron density. If $E_{max} > 0$ unoccupied states will therefore contribute to the partial density.
2. `band_indices`. Only states from the specified band indices will contribute to the partial electron density. By default, all bands are taken into account and the selection is determined by the energy range.
3. `kpoints`. The partial electron density is calculated from the eigenstates at the specified k-points.
4. `spectrum_method`. The method used to integrate the density. Currently only `GaussianBroadening` is supported. Each electronic state is assigned a weight which is computed by integrating a Gaussian centered at the states energy over the specified energy range.

The partial electron density is defined as:

$$n(\mathbf{r}) = \sum_i \sum_{\mathbf{k}} w_{\mathbf{k}} |\psi_{i,\mathbf{k}}(\mathbf{r})|^2 w(\epsilon_{i,\mathbf{k}}),$$

where the sum over band indices i includes only the specified bands (by default all bands). The weights $w_{\mathbf{k}}$ and $w(\epsilon_{i,\mathbf{k}})$ are the k-point weights and the Gaussian weights, respectively. The energy dependent Gaussian weight is computed as the integral of a normalized Gaussian centered at $\epsilon_{i,\mathbf{k}}$ with standard deviation σ over the specified energy range, i.e.

$$\begin{aligned} w(\epsilon_{i,\mathbf{k}}) &= \int_{E_{min}}^{E_{max}} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{(E-\epsilon_{i,\mathbf{k}})^2}{2\sigma^2}\right) dE \\ &= \frac{1}{2}\left[\mathrm{erf}\left(\frac{E_{max}-\epsilon_{i,\mathbf{k}}}{\sqrt{2}\sigma}\right) - \mathrm{erf}\left(\frac{E_{min}-\epsilon_{i,\mathbf{k}}}{\sqrt{2}\sigma}\right)\right], \end{aligned}$$

with erf being the error function.

Note that in combination with the projector-augmented wave (PAW) method, $n(\mathbf{r})$ is the smooth pseudo valence density, which does not necessarily integrate to the correct number of electrons. However, outside of the augmentation spheres, i.e. in the region where the bonds are located, $n(\mathbf{r})$ does correspond to the all electron valence density.

Tersoff-Hamann approximation of simulated STM

The *PartialElectronDensity* can be used to simulate scanning tunneling microscopy (STM) images within the Tersoff-Hamann approximation[1] assuming that the tip wave function has s-orbital character. When simulating an STM image, the `energy_range` can be interpreted as the bias difference applied between the tip and the surface.

Notice that the Tersoff-Hamann approximation is the simplest approach to simulated STM images and it is not always applicable. Also bare in mind that far away from a surface the partial density will be zero for an LCAO calculation since the basis orbitals have a finite range.

[1] J.ăTersoff and D.ăR. Hamann. Theory of the scanning tunneling microscopy. *Phys. Rev. B*, 31:805, 1985. doi:10.1103/PhysRevB.31.805.

PartialStructureFactor

class PartialStructureFactor(*md_trajectory*, *start_time=None*, *end_time=None*, *pair_selection=None*, *maximum_q_value=None*, *q_resolution=None*, *cutoff_radius=None*, *resolution=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the PartialStructureFactor object.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the partial structure factor for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** 0.0 * fs
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last frame time
- **pair_selection** (*sequence*) – Only consider interactions between these groups of atoms. A sequence has to contain containing two of the following types: Element, tag name, list of indices, or None. **Default:** all interactions
- **maximum_q_value** (*PhysicalQuantity* of type inverse length) – The maximum scattering vector length. **Default:** 15.0 / Angstrom
- **q_resolution** (*PhysicalQuantity* of type inverse length) – The binning of the scattering vectors. **Default:** 0.05 / Angstrom
- **cutoff_radius** (*PhysicalQuantity* of type length) – Upper limit on sampled distances. **Default:** Half the diagonal of the unit cell
- **resolution** (*PhysicalQuantity* of type length) – The binning of the radial distribution function. **Default:** 0.05 * Angstrom
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the partial structure factor.

qRange()

Return the list of scattering vector magnitudes.

Usage Examples

Load an *MDTrajectory* and calculate the *PartialStructureFactor* between aluminum and oxygen atoms:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

partial_sq = PartialStructureFactor(md_trajectory,
                                    pair_selection=[Aluminum, Aluminum])

# Get the q-values and the partial structure factor.
q_values = partial_sq.qRange().inUnitsOf(Angstrom**-1)
s_q      = partial_sq.data()

# Plot the data using pylab.
import pylab
```

(continues on next page)

(continued from previous page)

```
pylab.plot(q_values, s_q, label='Al-Al structure factor')
pylab.xlabel('q (1/Ang)')
pylab.ylabel('S(q)')
pylab.legend()

pylab.show()
```

partial_structure_factor.py

Notes

This object calculates the isotropic (i.e. averaged over all scattering angles) *PartialStructureFactor*. It is therefore predominantly aimed at the characterization of amorphous materials.

The partial structure factor is calculated as described in Ref.[1]:

$$S_{\alpha,\beta}(q) = \delta_{\alpha,\beta} + 4\pi\rho(c_\alpha c_\beta)^{1/2} \int_0^{r_{max}} r^2[g_{\alpha,\beta}(r) - 1]\frac{\sin(qr)}{qr}\frac{\sin(\pi r_{max} r)}{\pi r_{max} r}dr$$

where $g_{\alpha,\beta}(r)$ is the partial radial distribution function.

If no element pair is selected, then all atoms are treated equally and the full radial distribution function is used to calculate $S(q)$.

If the `pair_selection` parameter contains partially overlapping lists, the delta-function in the equation for $S(q)$ is evaluated as $\sqrt{\frac{N_a N_b}{N^2}}$.

The `cutoff_radius` parameter determines the range of the radial distribution function. This parameter might have an influence on the calculated structure factor at small scattering vectors.

ParticleIdentifier

class ParticleIdentifier(*symbol*, *tag=None*)

Constructor of the ParticleIdentifier

Parameters

- **symbol** (*str*) – The chemical symbol of the particle type.
- **tag** (*str | sequence of str | None*) – The tag name this particle type should be associated with.

addTag(*tag*)

Add a tag to this particle identifier.

If this particle identifier already contains this tag, a RuntimeError is raised.

Parameters
tag (*str*) – The tag(s) that belong to this particle identifier.

static generateFrom(*input*)

Create and return a ParticleIdentifier object from a variety of different object types. Possible inputs are:

- A ParticleType object. In this case, the ParticleIdentifier that is stored in the ParticleType is returned.

[1] G.ăGutierrez and B.ăJohansson. Molecular dynamics study of structural properties of amorphous Al2O3. *Phys. Rev. B*, 65:104202, Feb 2002. doi:10.1103/PhysRevB.65.104202.

- A ParticleIdentifier object. In this case, a deep copy of the ParticleIdentifier is returned.
- A string. A new ParticleIdentifier is build by using the content of the string as the symbol of the Particle Identifier. The tags are set to None.

Parameters
input (*str | ParticleType | ParticleIdentifier*) – The input the particle identifier should be generate from.

getSymbol()

Return the symbol of this object.

Returns
The symbol of this particle type (without any tags).

Return type
str

getTags()

Return the tags associated with this object as a list of str. The tags are sorted ascending.

getTremoloSymbol()

Return a string that uniquely describes this particle identifier. The string is created by adding the tags to the particle symbol, separated by a fixed tag separator string.

setTags(*tag*)

Set the tag(s).

Parameters
tag (*str | sequence of str | None*) – The tag(s) that belong to this particle identifier.

Usage Examples

Use a ParticleIdentifier to distinguish between different atom types of the same element, e.g. hydrogen in a methyl- and hydroxyl-group in a methanol molecule.

This is how ParticleIdentifiers can be used without tags:

```
# Set up a new TremoloXPotentialSet
potential_set = TremoloXPotentialSet(name='Methanol_bonded')

# Add particle types.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=0.1166*elementary_charge)
)

potential_set.addParticleType(
    ParticleType.fromElement(Oxygen, charge=-0.6497*elementary_charge)
)
# Set up a new bond potential for C-H bonds and add it to the potential set.
bond_potential = HarmonicBondPotential(
    particleType1=ParticleIdentifier('C'),
    particleType2=ParticleIdentifier('O'),
    k=13.87651*eV/Ang**2,
    r_0=1.410*Ang,
)
potential_set.addPotential(bond_potential)
```

If the potential should act on a tagged group of atoms, the *ParticleType* and *ParticleIdentifier* should be used like this:

```
# Add particle types.
potential_set.addParticleType(
    ParticleType.fromElement(Carbon, charge=0.1166*elementary_charge)
)

# Add particle types for hydroxyl hydrogen (with HO-tag).
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.4215*elementary_charge, tags='HO')
)

# Set up a new bond potential for C-H bonds and add it to the potential set.
bond_potential = HarmonicBondPotential(
    particleType1=ParticleIdentifier('H', tag='HO'),
    particleType2=ParticleIdentifier('O'),
    k=23.9803*eV/Ang**2,
    r_0=0.960*Ang,
)
potential_set.addPotential(bond_potential)
```

The full script can be found found in the file `particle_identifier.py`.

Notes

- ParticleIdentifiers can be used instead of atomic symbols to specify which particle types a potential should act on. This is helpful when dealing with different particle types of the same element, which are only distinguished by tags. As shown in the usage example, such a situation is very common in biomolecular force fields, where elements in different hybridization states or chemical environments can have different types with different potential parameters.
- The tagged particle type needs to be created by giving the tag name in *ParticleType*. Then this particle type refers only to those atoms of the given element which have this tag. All remaining atoms of this element then correspond to another *ParticleType* which has been created without tag. Tag names that are not used with any *ParticleType* are ignored in this *TremoloXPotentialSet*.

ParticleSwarmOptimizationParameters

class ParticleSwarmOptimizationParameters(*inertia_weight=None*, *local_acceleration=None*, *global_acceleration=None*, *omega=None*)

ParticleSwarmOptimizationParameters constructor.

Parameters

- **inertia_weight** (*Non negative float*) – The inertia weights parameter. This parameter serves as a scaling factor going from the previous iteration velocities to the next. **Default:** 0.8
- **local_acceleration** (*Non negative float*) – The local acceleration parameter. This parameter weights the probability for the particle to move away from the local best fit. **Default:** 0.2
- **global_acceleration** (*Non negative float*) – The global acceleration parameter. This parameter weights the probability for the particle to move away from the global best fit. **Default:** 0.8
- **omega** (*Non negative float*) – The weight to determine how much the fits should be shifted based on the velocities and the previous fit. **Default:** 0.5

globalAcceleration()

Returns
The global acceleration parameter.

Return type
float

inertiaWeight()

Returns
The inertia weight.

Return type
float

localAcceleration()

Returns
The local acceleration parameter.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

omega()

Returns
The fit shift parameter.

Return type
float

uniqueString()
Return a unique string representing the state of the object.

ParticleType

class ParticleType(*symbol*, *mass=None*, *charge=None*, *sigma=None*, *sigma14=None*, *epsilon=None*, *epsilon14=None*, *atomicNumber=None*, *tags=None*)

Constructor of the ParticleType class.

Parameters

- **symbol** (*str*) – The element symbol of the particle.
- **mass** (*PhysicalQuantity of type mass*) – The mass of particles of this type.
- **charge** (*PhysicalQuantity of type charge*) – The nominal charge of the particle. This value is used when evaluating the Coulomb interactions.
- **sigma** (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials.
- **sigma14** (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials if the bond order is 3.

•**epsilon** (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials.

•**epsilon14** (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials if the bond order is 3.

•**atomicNumber** (*int*) – The atomic number of this particle type.

•**tags** (*str, sequence of str or None*) – Make this particle type correspond only to particles that have the given tags assigned to them.

class TagCheck

Check class for tags. It checks whether some given input is either * a string (which corresponds to a single tag) * None (which corresponds to no tags) * a sequence of strings (which corresponds to multiple strings)

and converts the input into a list of strings.

apply(*name*, *value*)

Check if a given value is feasible.

Parameters

•**name** (*str*) – The name of the variable that is currently investigated. It is only used when writing error messages.

•**value** – The value under investigation.

Returns
The (possibly transformed) value that should be used.

addTag(*tag*)

Add a tag to this ParticleType.

Parameters
tag (*str*) – The tag name to be added.

distinguishableFrom(*otherParticleType*)

Determine if this ParticleType is distinguishable from another ParticleType. Distinguishable means here, that both ParticleTypes have different ParticleIdentifiers.

Parameters
otherParticleType (ParticleType) – The other ParticleType which the current type should be compared with.

Returns
True if the particle types are the same, False if they are different.

Return type
bool

static fromElement(*element*, *charge=None*, *sigma=None*, *sigma14=None*, *epsilon=None*, *epsilon14=None*, *tags=None*)

Create a ParticleType by using the information that is already stored in a given ATK Element.

Parameters

•**element** (PeriodicTableElement) – The element of the new ParticleType.

•**charge** (*PhysicalQuantity of type charge*) – The nominal charge of the particle. This value is. used when evaluating the Coulomb interactions.

•**sigma** (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials.

•**sigma14** (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials. if the bond order is 3.

•**epsilon** (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials.

- **epsilon14** (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials if the bond order is 3.
- **tags** (*str, sequence of str or None*) – Make this particle type correspond only to particles that have the given tags assigned to them.

Returns
The new ParticleType object

Return type
ParticleType

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

getParticleIdentifier()

Return the ParticleIdentifier object that uniquely defines the particles that belong to this particle type.

getTremoloSymbol()

Get the symbol that will be passed to the Tremolo-X compute core.

setCharge(*charge*)

Set the charge of the particle.

Parameters
charge (*PhysicalQuantity of type charge*) – The nominal charge of the particle. This value is used when evaluating the Coulomb interactions.

setEpsilon(*epsilon*)

Set the epsilon parameter of the particle.

Parameters
epsilon (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials.

setEpsilon14(*epsilon14*)

Set the epsilon14 parameter of the particle.

Parameters
epsilon14 (*PhysicalQuantity of type energy*) – Epsilon parameter for Lennard-Jones like potentials if the bond order is 3.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

setSigma(*sigma*)

Set the sigma parameter of the particle.

Parameters
sigma (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials.

setSigma14(*sigma14*)

Sets the sigma14 parameter of the particle.

Parameters
sigma14 (*PhysicalQuantity of type length*) – Sigma parameter for Lennard-Jones like potentials if the bond order is 3.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')
```

(continues on next page)

(continued from previous page)

```
# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O',
                                         r_0=2.1*Angstrom,
                                         k=2.0067*1/Ang,
                                         E_0=0.340554*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=1.0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O',
                                         r_0=3.618701*Angstrom,
                                         k=1.379316*1/Ang,
                                         E_0=0.042395*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=22.0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Create a new ParticleType only for hydrogen atoms with tag HO.

```
# Add particle types for hydroxyl hydrogen (with HO-tag).
potential_set.addParticleType(
    ParticleType.fromElement(Hydrogen, charge=0.4215*elementary_charge, tags='HO')
)
```

Notes

The class is used for defining properties of a particle type. You can specify the mass, the partial charge, and the Lennard-Jones parameters.

The partial charges specified here only become active, if a Coulomb solver (e.g. *CoulombDSF*) is added to the *TremoloXPotentialSet*. The same holds for the Lennard-Jones parameters `sigma` and `epsilon`, which require a Lennard-Jones pair potential (e.g. *LennardJonesPotential*) between all particle type pairs.

The Lennard-Jones parameters for 1-4 interactions are relevant for *bonded force fields*. Here, the arguments `sigma14` and `epsilon14` specify the Lennard-Jones parameters that are used between atoms which are separated by exactly 3 bonds when a *LennardJonesPotential* is defined with `bonded_mode='mode_14'`.

The static method `ParticleType.fromElement()` can be used to automatically take the particle properties from the given element.

Particle types for a given element can be split up into sub-types which technically behave as different elements. This can be achieved using atom tags on the configuration. A tagged particle type can be created by specifying the tag name in *ParticleType*. Then this particle type refers only to those atoms of the given element which have this tag. The force field will treat them as if they belonged to a separate element. All remaining atoms of this element then correspond to another *ParticleType* which has been created without tag. Tag names that are not used with any *ParticleType* are ignored in this *TremoloXPotentialSet*. Note that not all potentials support this separation of elements via tags.

The parameters `mass` and `atomic_number` are deprecated and not used by the force-field engine any longer.

PartitionCoefficient

class PartitionCoefficient(*solute_species*, *solvent_a*, *solvent_b*, *temperature=None*, *parameters=None*)

Calculating partition coefficients for solutes between two immiscible solvent phases.

Parameters

- **solute_species** (*CosmoRealSpecies* | Sequence of *CosmoRealSpecies*) – The solutes being dissolved between the two phases.
- **solvent_a** (*CosmoRSMixture* | *MoleculeConfiguration* | *CosmoRealSpecies*) – The solvent phase A.
- **solvent_b** (*CosmoRSMixture* | *MoleculeConfiguration* | *CosmoRealSpecies*) – The solvent phase B.
- **temperature** (*PhysicalQuantity* of type temperature) – The solvent temperature. **Default:** `298 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

parameters()

Returns
The COSMO real solvent parameters

Return type
CosmoRSParameters

partitionCoefficient()

Returns
The partition coefficient for each given solute.

Return type
float

temperature()

Returns
The solvent temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the octanol-water partition coefficient for a solute.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
octanol = database.exportSpecies('1-octanol')
acetonitrile = database.exportSpecies('acetonitrile')

# Saturated octanol contains ~24mol% water.
octanol_phase = CosmoRSMixture([(octanol, 0.76), (water, 0.24)])

# Determine the partition coefficient.
partition_coefficient = PartitionCoefficient(
    acetonitrile,
    octanol_phase,
    water,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Retrieve the logP.
logp = partition_coefficient.partitionCoefficient()[0]
nlprint(f'The partition coefficient for acetonitrile is {logp}')
```

octanol_water_example.py

Notes

The *PartitionCoefficient* object allows one to calculate the partition coefficient, $logP$, of a solute given any two immiscible solvent phases. The partition coefficient provides a measure of a solute distribution between two immiscible phases and is defined as

$$logP = log_{10}\left(\frac{[solute]_A}{[solute]_B}\right)$$

The subscripts A and B refer to each phase hence $[solute]_A$ and $[solute]_B$ are the concentration of the solute in the respective phase. Notice that the relative concentrations represents an equilibrium of the process

$$solute(A) \rightleftarrows solute(B)$$

Therefore, a negative $logP$ corresponds to an equilibrium towards phase B and positive $logP$ an equilibrium towards phase A. The partition coefficient is calculated and retrieved from the *PartitionCoefficient* as demonstrated in the **Usage Example**. Note that a phase may also be defined as a *CosmoRSMixture*.

In COSMO-RS the partition coefficient can be defined as:

$$logP = \frac{\mu_B^{solv} - \mu_A^{solv}}{\ln(10)RT} + \log_{10}(V_m^B) - \log_{10}(V_m^A)$$

Here μ_A^{solv} and μ_B^{solv} represent the chemical potential of the molecule dissolved in the A and B phases, respectively. Likewise V_m^A and V_m^B are the respective molar volumes. In this definition the partition coefficient is based on comparing concentrations, which is also the definition used in QuantumATK. It is also possible to define the partition coefficient in terms of mole fractions. In this case the molar volumes would be omitted.

PeriodicBoundaryCondition

class PeriodicBoundaryCondition

A periodic boundary condition.

PeriodicTableElement

class PeriodicTableElement

Base class for all the elements in the PeriodicTable.

atomicMass()

Return the atomic mass of the element. @return the atomic mass.

atomicNumber()

Return the atomic number of the element. @return the atomic number.

covalentBonds()

Return the covalent bonds of the element. @return the covalent bonds.

covalentRadius()

Return the covalent radius of the element. @return the covalent radius.

crystalType()

Return the crystal type of the element. @return the crystal type.

latticeBARatio()

Return the lattice b a ratio of the element. @return the lattice b a ratio.

latticeCARatio()

Return the lattice c a ratio of the element. @return the lattice c a ratio.

latticeConstant()

Return the lattice constant of the element. @return the lattice constant.

metal()

Return the metal of the element. @return the metal.

name()

Return the name of the element. @return the name.

rgbColor()

Return the rgb color of the element. @return the rgb color.

symbol()

Return the symbol of the element. @return the symbol.

vanDerWaalsRadius()

Return the van der waals radius of the element. @return the van der waals radius.

Usage Examples

Use the elements `Hydrogen` and `Oxygen` to set up a *MoleculeConfiguration*.

```
# Set up elements and positions
elm = [ Oxygen, Hydrogen, Hydrogen ]
pos = [[ 0.000, 0.000, 0.0],
       [ 0.757, 0.586, 0.0],
       [-0.757, 0.586, 0.0]]*Angstrom

# Add them to a configuration
h2o = MoleculeConfiguration(elm, pos)
```

`molecule_configuration.py`

Alternatively, the shorter names `H` and `O` can also be used as in

```
# Set up elements and positions
elm =[PeriodicTable.O, PeriodicTable.H, PeriodicTable.H]
pos =[[ 0.000, 0.000, 0.0],
      [ 0.757, 0.586, 0.0],
      [-0.757, 0.586, 0.0]]*Angstrom

# Add them to a configuration
h2o = MoleculeConfiguration(elm, pos)
```

You can use the query functions to retrieve element specific properties, such as name, mass, etc.

```
element_name = Silicon.name()
element_mass = Silicon.atomicMass()
```

Periodic table

The entire set of elements from the periodic system, as well as some of their key properties are listed in *Atomic data*.

PhononBandstructure

class PhononBandstructure(*dynamical_matrix*, *configuration=None*, *route=None*, *points_per_segment=None*, *qpoints=None*, *number_of_bands=None*, *projection_list=None*)

Analysis class for calculating the phonon bandstructure for a bulk configuration.

Parameters

- **dynamical_matrix** (*DynamicalMatrix*) – The DynamicalMatrix to calculate the phonon bandstructure from.
- **configuration** (*BulkConfiguration*) – The configuration for which the vibrational modes should be calculated. If no calculator is attached to the configuration, the calculator from dynamical_matrix is used. Currently, configuration must be the same as the configuration from dynamical_matrix and is therefore not needed. **Default:** The configuration from `dynamical_matrix`
- **route** (*list of str*) – The route to take through the Brillouin-zone as a list of symmetry points of the unit cell, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `qpoints`. **Default:** Unit-cell dependent route.
- **points_per_segment** (*Positive int*) – The number of points per segment of the route. **Default:** `20`.
- **qpoints** (*list of lists of floats*) – A list of 3-dimensional fractional q-points at which to calculate the energies of the bands e.g. `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. The shape is (Q, 3) where Q is the number of q-points. This option is mutually exclusive to `route`, and `points_per_segment`. **Default:** Unit-cell dependent route.
- **number_of_bands** (Positive int | `All`) – The number of bands. **Default:** `All` (All bands are included).
- **projection_list** (*ProjectionList*) – A projection list object defining a projection. **Default:** If no projection list is specified all orbitals will be used.

conductionBandEdge()

Override base method for getting the conduction band edge.

directBandGap()

Override base method for getting the direct band-gap.

evaluate()

Returns
The eigenvalues for the phonon bandstructure. The shape is (Q, B) where Q is the number of q-points and B is the number of bands.

Return type
`PhysicalQuantity` of type energy

indirectBandGap()

Override base method for getting the indirect band-gap.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

qpoints()

Returns
The list of q-points at which the energies of the bands are calculated as a list of 3-dimensional fractional q-points. The shape is (Q, 3) where Q is the number of q-points.

Return type
list of lists of floats

route()

Returns
The route through the Brillouin-zone as a list of symmetry points of the unit cell.

Return type
list of str

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

valenceBandEdge()

Override base method for getting the valence band edge.

Usage Examples

Calculate the *PhononBandstructure* of silicon using the *TremoloXCalculator* calculator with the Stillinger-Weber potential[1]:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[0.0 , 0.0 , 0.0 ],
                          [0.25, 0.25, 0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Dynamical matrix
# ------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    configuration=bulk_configuration,
    repetitions=Automatic,
    atomic_displacement=0.01*Angstrom,
    acoustic_sum_rule=False,
    symmetrize=True,
    finite_difference_method=Central,
    )

# ------------------------------------------------------------
# Phonon bandstructure
# ------------------------------------------------------------
phonon_bandstructure = PhononBandstructure(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    route=['G', 'X', 'W', 'L', 'G', 'K', 'X', 'U', 'W', 'K', 'L'],
    points_per_segment=20,
    number_of_bands=All
    )
nlsave('si_phonon_bandstructure.nc', phonon_bandstructure)
```

phonon_bandstructure.py

Note: The *PhononBandstructure* can be visualized by selecting the file `si_phonon_bandstructure.nc` in the **Lab Floor** of QuantumATK and opening it using the **Bandstructure Analyzer** or the **2D Plot** tool.

Notes

•The details of the force constant calculation to obtain the dynamical matrix, can be found under *DynamicalMatrix*.

•To export the data of a *PhononBandstructure*, use the method *nlprint*.

•Symmetry points of the Brillouin zones can be found in BravaisLattices.

PhononDensityOfStates

class PhononDensityOfStates(*dynamical_matrix*, *configuration=None*, *qpoints=None*, *number_of_bands=None*)

Constructor for the phonon density of states object.

Parameters

- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to calculate the phonon density of states from.
- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration for which the vibrational modes should be calculated. If no calculator is attached to the configuration, the calculator from dynamical_matrix is used. Currently, configuration must be the same as the configuration from dynamical_matrix and is therefore not needed. **Default:** The configuration from `dynamical_matrix`
- **qpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The q-points for which the phonon density of states should be calculated. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **number_of_bands** (*int*) – The number of bands. **Default:** All bands

elements()

Query method for the elements in the configuration used for calculating the density of states.

Returns
A list of the elements.

Return type
list of *PeriodicTableElement*

energies()

Query method for the energies used for generating the last spectrum.

Returns
The energies used for generating the last spectrum.

Return type
PhysicalQuantity of type energy.

energyMax()

Query method for the maximum energy eigenvalue.

Returns
The maximum energy eigenvalue.

Return type
PhysicalQuantity of type energy

energyMin()

Query method for the minimum energy eigenvalue.

Returns
The minimum energy eigenvalue.

Return type
PhysicalQuantity of type energy

entropy(*temperature=PhysicalQuantity(300.0, K)*)

Calculate the entropy of the phonons.

The entropy is evaluated using the quasi-harmonic approximation (see Eq.(5.5) in Dove, Martin T. (1993), Introduction to lattice dynamics, Cambridge university press).

Parameters
`temperature` (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the entropy at. **Default:** `300 * Kelvin`

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

`evaluate()`

Return the default phonon density of states spectrum.

Returns
The phonon density of states.

Return type
PhysicalQuantity with the unit `1 / meV`

`freeEnergy(`*temperature=PhysicalQuantity(300.0, K)*`)`

Calculate the Helmholtz free energy of the molecule / lattice.

Parameters
`temperature` (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the entropy at. **Default:** `300 * Kelvin`

Returns
The free energy of the molecule / lattice (without internal energy).

Return type
PhysicalQuantity with the unit `eV`

`gaussianSpectrum(`*energies=None*, *broadening=None*`)`

Return the phonon density of states spectrum using Gaussian broadening.

Parameters

- `energies` (*PhysicalQuantity* of type energy) – List of energies for which the spectrum should be calculated. **Default:** Energies from emin to emax
- `broadening` (*PhysicalQuantity* of type energy) – Broadening of the Gaussian. **Default:** `0.01 * eV`

Returns
The density of states spectrum.

Return type
PhysicalQuantity with the unit `1 / meV`

`internalEnergy(`*temperature=PhysicalQuantity(300.0, K)*`)`

Calculate the internal harmonic energy of the molecule / lattice at the given temperature.

Parameters
`temperature` (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the internal energy at. **Default:** `300 * Kelvin`

Returns
The internal energy of the molecule / lattice.

Return type
PhysicalQuantity with the unit `eV`

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`tetrahedronSpectrum(energies=None)`

Return the phonon density of states spectrum using the tetrahedron method.

Parameters
energies (*PhysicalQuantity* of type energy) – List of energies for which the spectrum should be calculated. **Default:** Energies from emin to emax

Returns
The density of states spectrum.

Return type
PhysicalQuantity with the unit `1 / meV`

`uniqueString()`

Return a unique string representing the state of the object.

`zeroPointEnergy()`

Calculate the zero point energy in the quasi-harmonic approximation.

Returns
The zero point energy of the molecule / lattice

Return type
PhysicalQuantity with the unit `eV`

Usage Examples

Calculate the phonon density of states of a silicon crystal using the *TremoloXCalculator* calculator with the Stillinger-Weber potential[1]. The example also shows how to calculate the vibrational entropy of the crystal unit cell.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
```

(continues on next page)

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

(continued from previous page)

```
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[0.0 , 0.0 , 0.0 ],
                          [0.25, 0.25, 0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Dynamical matrix
# ------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    configuration=bulk_configuration,
    repetitions=Automatic,
    atomic_displacement=0.01*Angstrom,
    acoustic_sum_rule=False,
    symmetrize=True,
    finite_difference_method=Central,
    )

# ------------------------------------------------------------
# Phonon density of states
# ------------------------------------------------------------
phonon_density_of_states = PhononDensityOfStates(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix,
    qpoints=MonkhorstPackGrid(10,10,10),
    )

nlsave('si_phonon_dos.nc', phonon_density_of_states)

# Print the results ot the log
nlprint(phonon_density_of_states)

# Calculate the vibrational entropy at 300 K
entropy = phonon_density_of_states.entropy(300.0*Kelvin)
```

si_phonon_dos.py

Note: The spectrum can be shown by selecting the file `si_phonon_dos.nc` in the **Lab Floor** of QuantumATK and opening it using the **2D Plot** tool.

Plot the Gaussian and tetrahedron spectrum using pylab:

```
# read phonon DOS object from file
dos = nlread('si_phonon_dos.nc', PhononDensityOfStates)[0]

# make list of energies
energies = numpy.arange(0.0, 80.0, 0.2)*meV

# calculate the DOS spectrum with two different methods
dos_t = dos.tetrahedronSpectrum(energies)
dos_g = dos.gaussianSpectrum(energies, broadening = 1.0*meV)

#plot the spectra using pylab
import pylab
pylab.figure()
pylab.plot(energies.inUnitsOf(meV), dos_t.inUnitsOf(meV**-1))
pylab.plot(energies.inUnitsOf(meV), dos_g.inUnitsOf(meV**-1))
pylab.xlabel("Energy (meV)")
pylab.ylabel("Phonon DOS (1/meV)")
pylab.show()
```

si_phonon_dos_plot.py

Notes

•The details of the force constant calculation to obtain the dynamical matrix can be found under *DynamicalMatrix*.

•The routine utilizes the symmetries of the crystal to reduce the computational load.

•The default spectrum is a Gaussian spectrum for a *MoleculeConfiguration* and a *BulkConfiguration* with less than 10 k-points. For a *BulkConfiguration* with more than 10 k-points, the tetrahedron method is used.

•The implementation of the tetrahedron method follows Ref.[2]

The *PhononDensityOfStates* class can also be used to calculate thermodynamic properties in the harmonic approximation. The following properties can be calculated:

•The free energy at a given temperature can be calculated via the `freeEnergy()` method:

$$F = -\frac{1}{N_q}\sum_{q,s}\frac{E(q,s)}{2} + \frac{1}{\beta N_q}\sum_{q,s}\ln(1-\exp(-\beta E(q,s)))$$

where $\beta = 1/k_B T$ and N_q denotes the number of q-points to sum over.

•The internal energy at a given temperature, including zero-point energy, can be calculated via the `internalEnergy()` method:

$$U = -\frac{1}{N_q}\sum_{q,s}\frac{E(q,s)}{2} + \frac{1}{N_q}\sum_{q,s}\frac{E(s,q)}{\exp(\beta E(s,q))-1}$$

[2] P.ăE. Blochl, O.ăJepsen, and O.ăK. Andersen. Improved tetrahedron method for Brillouin-zone integrations. *Phys. Rev. B*, 49:16223–16233, Jun 1994. doi:10.1103/PhysRevB.49.16223.

In this formula, the first part describes the zero-point energy and the second part stands for the finite temperature contribution of the phonons to the internal energy.

•The entropy at a given temperature can be calculated via the `entropy()` method:

$$S = \frac{1}{N_q}\sum_{q,s}\frac{E(q,s)}{T}\frac{\exp(-\beta E(q,s))}{1-\exp(-\beta E(q,s))} - \frac{k_B}{N_q}\sum_{q,s}\ln(1-\exp(-\beta E(q,s)))$$

Note that all these quantities do not include any potential energy contributions, as calculated via *TotalEnergy*, so in order to get the absolute free energy or internal energy values, one would have to add these contributions on top.

PhononTransmissionSpectrum

class PhononTransmissionSpectrum(*dynamical_matrix*, *configuration=None*, *energies=None*, *qpoints=None*, *qpoints_weights=None*, *self_energy_calculator=None*, *infinitesimal=None*)

Class for representing the phonon transmission spectrum for a given device configuration.

Parameters

•**dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use for the calculation.

•**configuration** (*DeviceConfiguration*) – The device configuration for which the vibrational modes should be calculated. If no calculator is attached to the configuration, the calculator from dynamical_matrix is used. Currently, configuration must be the same as the configuration from dynamical_matrix and is therefore not needed. **Default:** The configuration from `dynamical_matrix`

•**energies** (*PhysicalQuantity* of type energy.) – The energies for which the transmission spectrum should be calculated. **Default:** `numpy.linspace(0, 0.5, 101)*eV`

•**qpoints** – The q-points for which to calculate the transmission spectrum. This can either be given as a q-point grid or a list of fractional q-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. Note that the q-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

•**qpoints_weights** – The weight of each q-point for which the transmission spectrum should be calculated. **Default:** The weights corresponding to the MonkhorstPackGrid, or a list of [1.0, 1.0,] if q-points are specified as floats.

•**self_energy_calculator** – The SelfEnergyCalculator to be used for the transmission spectrum. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())` :type self_energy_calculator: *DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*

•**infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the transmission calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

adaptive()

Returns
True if adaptive grids are used, otherwise False.

Return type
bool

energies()

Returns
The energies used in this transmission spectrum

Return type
PhysicalQuantity of type energy

evaluate(*qpoints=None*)

Evaluates the q-point averaged transmission spectrum.

Parameters
qpoints (*MonkhorstPackGrid* | *RegularKpointGrid* | list(n) of list(3) of float.) – The set of q-points to average over. The q-points must be compatible with the q-points used in the constructor. **Default:** Average over all q-points.

Returns
The q-point averaged transmission spectrum.

Return type
array(n_energies)

infinitesimal()

Returns
The infinitesimal used for calculating the transmission.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

qpoints()

Returns
The q-points used in this transmission spectrum. The shape of the array is (number_of_qpoints, 3)

Return type
numpy.ndarray

qpointsWeights()

Returns
The weights of each q-point used in this transmission spectrum.

Return type
list.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

thermalConductance(*phonon_temperature=None*)

Calculate the thermal conductance of the phonon spectrum.

Parameters
phonon_temperature (Positive *PhysicalQuantity* of type temperature.) – The temperature of the phonon system.
Default: `300*Kelvin`

Returns
The thermal conductance of the phonon spectrum.

Return type
PhysicalQuantity with unit Joule/Second/Kelvin

transmission()

Return the transmission coefficients of the phonon transmission spectrum.

Returns
The transmission coefficients for each energy and q-point. The shape of the array is (number_of_energies, number_of_qpoints)

Return type
numpy.ndarray

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *PhononTransmissionSpectrum* for a generic bulk silicon device:

```
# -----------------------------------------------------------
# Left electrode
# -----------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
left_electrode_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon]

# Define coordinates
left_electrode_coordinates = [[ 0.      ,  0.      ,  0.678825],
                              [ 2.7153  ,  2.7153  ,  0.678825],
                              [ 1.35765 ,  1.35765 ,  2.036475],
                              [ 4.07295 ,  4.07295 ,  2.036475],
                              [ 2.7153  ,  0.      ,  3.394125],
                              [ 0.      ,  2.7153  ,  3.394125],
                              [ 4.07295 ,  1.35765 ,  4.751775],
                              [ 1.35765 ,  4.07295 ,  4.751775]]*Angstrom

# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
```

(continues on next page)

(continued from previous page)

```
    cartesian_coordinates=left_electrode_coordinates
    )

# -----------------------------------------------------------
# Right electrode
# -----------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
right_electrode_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,␣
↪Silicon,
                            Silicon]

# Define coordinates
right_electrode_coordinates = [[ 0.      ,  0.      ,  0.678825],
                               [ 2.7153  ,  2.7153  ,  0.678825],
                               [ 1.35765 ,  1.35765 ,  2.036475],
                               [ 4.07295 ,  4.07295 ,  2.036475],
                               [ 2.7153  ,  0.      ,  3.394125],
                               [ 0.      ,  2.7153  ,  3.394125],
                               [ 4.07295 ,  1.35765 ,  4.751775],
                               [ 1.35765 ,  4.07295 ,  4.751775]]*Angstrom

# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )

# -----------------------------------------------------------
# Central region
# -----------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 21.7224]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
central_region_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
central_region_coordinates = [[  0.      ,   0.      ,   0.678825],
                              [  2.7153  ,   2.7153  ,   0.678825],
                              [  1.35765 ,   1.35765 ,   2.036475],
                              [  4.07295 ,   4.07295 ,   2.036475],
                              [  2.7153  ,   0.      ,   3.394125],
                              [  0.      ,   2.7153  ,   3.394125],
                              [  4.07295 ,   1.35765 ,   4.751775],
                              [  1.35765 ,   4.07295 ,   4.751775],
                              [  0.      ,   0.      ,   6.109425],
                              [  2.7153  ,   2.7153  ,   6.109425],
                              [  1.35765 ,   1.35765 ,   7.467075],
                              [  4.07295 ,   4.07295 ,   7.467075],
                              [  2.7153  ,   0.      ,   8.824725],
                              [  0.      ,   2.7153  ,   8.824725],
                              [  4.07295 ,   1.35765 ,  10.182375],
                              [  1.35765 ,   4.07295 ,  10.182375],
                              [  0.      ,   0.      ,  11.540025],
                              [  2.7153  ,   2.7153  ,  11.540025],
                              [  1.35765 ,   1.35765 ,  12.897675],
                              [  4.07295 ,   4.07295 ,  12.897675],
                              [  2.7153  ,   0.      ,  14.255325],
                              [  0.      ,   2.7153  ,  14.255325],
                              [  4.07295 ,   1.35765 ,  15.612975],
                              [  1.35765 ,   4.07295 ,  15.612975],
                              [  0.      ,   0.      ,  16.970625],
                              [  2.7153  ,   2.7153  ,  16.970625],
                              [  1.35765 ,   1.35765 ,  18.328275],
                              [  4.07295 ,   4.07295 ,  18.328275],
                              [  2.7153  ,   0.      ,  19.685925],
                              [  0.      ,   2.7153  ,  19.685925],
                              [  4.07295 ,   1.35765 ,  21.043575],
                              [  1.35765 ,   4.07295 ,  21.043575]]*Angstrom

# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )

device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode]
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)
```

(continues on next page)

(continued from previous page)

```
device_configuration.setCalculator(calculator)
device_configuration.update()

# ------------------------------------------------------------
# Phonon transmission spectrum
# ------------------------------------------------------------
phonon_transmission_spectrum = PhononTransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(0,0.5,101)*eV,
    qpoints=MonkhorstPackGrid(5,5),
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )

nlsave('si_phonon_transmission.nc', phonon_transmission_spectrum)
```

phonon_transmission.py

Calculate the thermal conductance for different phonon temperatures:

```
temp = numpy.linspace(100, 1000, 19)
conductances = []

for t in temp:
    conductance = phonon_transmission_spectrum.thermalConductance(t*Kelvin)
    conductances.append(conductance)

print("Phonon temperature   Thermal Conductance")
output_format = "      %6.0f K              %s"
for i in range(len(temp)):
    print(output_format % (temp[i], conductances[i]))
```

thermal_conductance.py

Notes

The calculation of the phonon transmission essentially follows the formalism employed to calculate the electronic *TransmissionSpectrum*, by using the substitutes:

$$E\mathbf{S}_C \rightarrow \omega^2\mathbf{M}\,,$$

and

$$\mathbf{H}_C \rightarrow \mathbf{K}_C\,,$$

where $\mathbf{M}$ is the diagonal matrix with the element masses and $\mathbf{K}_C$ is the dynamical matrix of the central region. Similar replacements are made for the electrodes. For more details see, e.g. Ref.[1].

Note: The phonon transmission calculated via the non-equilibrium Greens function formalism gives the ideal, ballistic contributions to the thermal transport through the device. It does not account for anharmonic effects, such as

[1] T.ăMarkussen, A.-P. Jauho, and M.ăBrandbyge. Electron and phonon transport in silicon nanowires: Atomistic approach to thermoelectric properties. *Phys. Rev. B*, 79:035415, Jan 2009. doi:10.1103/PhysRevB.79.035415.

phonon-phonon-scattering. To take such effects into account as well, you should consider other approaches, e.g. based on molecular dynamics simulations.

To export the data of a transmission spectrum, use the method *nlsave*.

The details and parameters of the force constant calculation to obtain the dynamical matrix, can be found under *DynamicalMatrix*.

Unit of transmission

A fully transmitting channel contributes 1. See also, *TransmissionEigenvalues*.

For a *PhononTransmissionSpectrum*, the first index (i.e. the spin index) in the list, that is returned by `transmission()`, must always be 0, as phonons do not have a spin.

Current

The heat current can be calculated from the transmission coefficients,

$$J(T_L, T_R) = \frac{\hbar}{2\pi} \int_0^\infty \omega T(\omega) \left[n_B(T_L) - n_B(T_R) \right] d\omega \,,$$

where $n_B = (e^{\hbar\omega/k_B T} - 1)^{-1}$ is the Bose-Einstein distribution of the phonons, $T_{L/R}$ are the phonon temperatures of the left/right electrode, and $T(\omega)$ is the transmission coefficient at frequency ω .

Thermal conductance

In the limit of small temperature differences, ΔT, between the left and right electrodes, with $T_L = T + \Delta T/2$ and $T_R = T - \Delta T/2$, the thermal conductance at average temperature, T, is calculated from the phonon transmission as follows:

$$\kappa_{ph}(T) = \frac{\hbar}{2\pi k_B T^2} \int_0^\infty \omega^2 T(\omega) \frac{e^{\hbar\omega/k_B T}}{(e^{\hbar\omega/k_B T} - 1)^2} d\omega.$$

Photocurrent

class Photocurrent(*configuration*, *energies=None*, *photon_energies=None*, *kpoints=None*, *photon_polarization=None*, *energy_resolution=None*, *calculate_all_transmissions=None*)

Class for calculating the photocurrent and photon-mediated transmission in a device.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with an attached *LCAOCalculator* for which to calculate the photocurrent.
- **energies** (list of *PhysicalQuantity* of type energy | Automatic) – The energies for which to calculate the photon-mediated transmission. **Default:** `Automatic`. Automatically setup energies based on the electrode Fermi levels and the photon energies.
- **photon_energies** (list of *PhysicalQuantity* of type energy) – The photon energies to consider. Must be positive. **Default:** `[1.0] * eV`

- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points for which to identify the band ladders. Note that the k-points must be in the xy-plane. **Default:** `[0.0, 0.0, 0.0]` (Gamma-point)
- **photon_polarization** (*list (size 3) of complex float*) – The polarization of the photon illuminating the device configuration. **Default:** `[0.0, 0.0, 1.0]`
- **energy_resolution** (*PhysicalQuantity* of type energy) – The distance between two energy points, when the energies are determined automatically. This parameter is only used, when energies=None or energies=Automatic **Default:** `0.05 * eV`
- **calculate_all_transmissions** (*bool*) – Boolean to specify if the photo-induced transmissions functions should be calculated at all possible energies. By default this is false, implying that the transmission functions only are calculated at the energies and photon energies, that would contribute to the total photo-current. If post-processing by shifting the Fermi level is wanted, the `calculate_all_transmissions` parameter can be set to `True`. **Default:** `False`

calculateAllTransmissions()

Returns
Boolean controlling whether all transmissions are calculated.

Return type
bool

electrodeFermiLevels()

Returns
The electrode Fermi levels in absolute energy.

Return type
list (size 2) of *PhysicalQuantity* of type energy

energies()

Returns
The energies for which the photon-mediated transmission is calculated.

Return type
list of *PhysicalQuantity* of type energy

energyResolution()

Returns
The energy resolution used to setup the energies.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero used for the energy scale.

Return type
PhysicalQuantity of type energy

evaluate(*photon_modes=None*, *bias=None*, *temperature=None*, *fermi_shift=None*, *relative_permittivity=None*, *relative_permeability=None*, *photon_flux=None*, *spin=None*, *emission=None*)

Evaluate the photocurrent.

Parameters

•**photon_modes** (All | int | list of int) – The indices of the photons for which to calculate the photocurrent. Must be non-negative. **Default:** All

•**bias** (*PhysicalQuantity* of type voltage) – The voltage for which the current should be calculated. **Default:** The bias of the *DeviceConfiguration*.

•**temperature** (*PhysicalQuantity* of type temperature) – The temperature fow which the current should be calculated. **Default:** 300*Kelvin

•**fermi_shift** (*PhysicalQuantity* of type energy) – The shift of the Fermi level such that the current is evaluated at the energy E = fermi_energy + fermi_shift. **Default:** 0.0*eV

•**relative_permittivity** (*float*) – The permittivity of the central region compared to vacuum. Must be positive. **Default:** 1.0

•**relative_permeability** (*float*) – The permeability of the central region compared to vacuum. Must be positive. **Default:** 1.0

•**photon_flux** (*PhysicalQuantity* of type 1/(area*time)) – The photon flux of the light incident on the central region. **Default:** 1.0*Second**-1*Angstrom**-2

•**spin** (Spin.Up | Spin.Down | Spin.Sum) – The spin component for which to calculate the current. **Default:** Spin.Sum

•**emission** (*bool*) – Boolean controlling whether photon emissions are included. **Default:** False

Returns
The photocurrent as a list with the current for each of the photon energies.

Return type
list of *PhysicalQuantity* of type current

kpoints()

Returns
The fractional k-points used in the calculation.

Return type
list of list (size 3) of float

maximumTemperature()

Returns
The maximum trmperature used to setup the energies.

Return type
PhysicalQuantity of type temperature

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

photonEnergies()

Returns
The photon energies to consider.

Return type
list of *PhysicalQuantity* of type energy

photonMediatedTransmission(*photon_mode=None*, *carrier_type=None*, *initial_lead=None*, *final_lead=None*, *relative_permittivity=None*, *relative_permeability=None*, *photon_flux=None*, *spin=None*)

Evaluate the photon-mediated transmission.

Parameters

- **photon_mode** (*int*) – The indices of the photons for which to calculate the photon-mediated transmission. Must be non-negative. **Default:** `0`
- **carrier_type** (`Electron` | `Hole`) – The carrier type for which to calculate the transmission. **Default:** `Electron`
- **initial_lead** (`Left` | `Right`) – The lead that the unexcited carrier enters from. **Default:** `Left`
- **final_lead** (`Left` | `Right`) – The lead that the excited carrier exits to. **Default:** `Right`
- **relative_permittivity** (*float*) – The permitivity of the central region compared to vacuum. Must be positive. **Default:** `1.0`
- **relative_permeability** (*float*) – The permability of the central region compared to vacuum. Must be positive. **Default:** `1.0`
- **photon_flux** (*PhysicalQuantity* of type 1/(area*time)) – The photon flux of the light incident on the central region. **Default:** `1.0*Second**-1*Angstrom**-2`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component for which to calculate the transmission. **Default:** `Spin.Sum`

Returns
The photon-mediated transmission.

Return type
list (size `len(kpoints)`, `len(energies)`) of float

photonPolarization()

Returns
The photon polarization.

Return type
list (size 3) of complex float

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

The script below shows how to setup a *Photocurrent* calculation for a silicon p-n junction.

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Two-probe Configuration
# ------------------------------------------------------------

# ------------------------------------------------------------
# Left Electrode
# ------------------------------------------------------------

# Set up lattice
vector_a = [3.84001408591, 0.0, 0.0]*Angstrom
vector_b = [0.0, 3.84001408591, 0.0]*Angstrom
vector_c = [0.0, 0.0, 10.8612]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
left_electrode_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon]

# Define coordinates
left_electrode_coordinates = [[  1.920007042956,  -0.            ,   0.678825      ],
                              [  1.920007042956,   1.920007042956,   2.036475      ],
                              [  0.            ,   1.920007042956,   3.394125      ],
                              [  0.            ,   0.            ,   4.751775      ],
                              [  1.920007042956,  -0.            ,   6.109425      ],
                              [  1.920007042956,   1.920007042956,   7.467075      ],
                              [  0.            ,   1.920007042956,   8.824725      ],
                              [  0.            ,   0.            ,  10.182375      ␣
↪]]*Angstrom

# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )

# Add external potential
external_potential = AtomicCompensationCharge([
    ('doping_0', 0.0200195107106),
    ('doping_1', -0.0200195107106)
    ])

left_electrode.setExternalPotential(external_potential)

# ------------------------------------------------------------
# Right Electrode
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Set up lattice
vector_a = [3.84001408591, 0.0, 0.0]*Angstrom
vector_b = [0.0, 3.84001408591, 0.0]*Angstrom
vector_c = [0.0, 0.0, 10.8612]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
right_electrode_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,␣
↪Silicon,
                            Silicon]

# Define coordinates
right_electrode_coordinates = [[  1.920007042956,  -0.            ,   0.678825      ],
                               [  1.920007042956,   1.920007042956,   2.036475      ],
                               [  0.            ,   1.920007042956,   3.394125      ],
                               [  0.            ,   0.            ,   4.751775      ],
                               [  1.920007042956,  -0.            ,   6.109425      ],
                               [  1.920007042956,   1.920007042956,   7.467075      ],
                               [  0.            ,   1.920007042956,   8.824725      ],
                               [  0.            ,   0.            ,  10.182375      ␣
↪]]*Angstrom

# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )

# Add external potential
external_potential = AtomicCompensationCharge([
    ('doping_0', 0.0200195107106),
    ('doping_1', -0.0200195107106)
    ])

right_electrode.setExternalPotential(external_potential)

# ------------------------------------------------------------
# Central Region
# ------------------------------------------------------------

# Set up lattice
vector_a = [3.84001408591, 0.0, 0.0]*Angstrom
vector_b = [0.0, 3.84001408591, 0.0]*Angstrom
vector_c = [0.0, 0.0, 76.0284]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
central_region_elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
```

(continues on next page)

(continued from previous page)

```
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
                           Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon]

# Define coordinates
central_region_coordinates = [[  1.920007042956,  -0.            ,   0.678825      ],
                              [  1.920007042956,   1.920007042956,   2.036475      ],
                              [  0.            ,   1.920007042956,   3.394125      ],
                              [  0.            ,   0.            ,   4.751775      ],
                              [  1.920007042956,  -0.            ,   6.109425      ],
                              [  1.920007042956,   1.920007042956,   7.467075      ],
                              [  0.            ,   1.920007042956,   8.824725      ],
                              [  0.            ,   0.            ,  10.182375      ],
                              [  1.920007042956,  -0.            ,  11.540025      ],
                              [  1.920007042956,   1.920007042956,  12.897675      ],
                              [  0.            ,   1.920007042956,  14.255325      ],
                              [  0.            ,   0.            ,  15.612975      ],
                              [  1.920007042956,  -0.            ,  16.970625      ],
                              [  1.920007042956,   1.920007042956,  18.328275      ],
                              [  0.            ,   1.920007042956,  19.685925      ],
                              [  0.            ,   0.            ,  21.043575      ],
                              [  1.920007042956,  -0.            ,  22.401225      ],
                              [  1.920007042956,   1.920007042956,  23.758875      ],
                              [  0.            ,   1.920007042956,  25.116525      ],
                              [  0.            ,   0.            ,  26.474175      ],
                              [  1.920007042956,  -0.            ,  27.831825      ],
                              [  1.920007042956,   1.920007042956,  29.189475      ],
                              [  0.            ,   1.920007042956,  30.547125      ],
                              [  0.            ,   0.            ,  31.904775      ],
                              [  1.920007042956,  -0.            ,  33.262425      ],
                              [  1.920007042956,   1.920007042956,  34.620075      ],
                              [  0.            ,   1.920007042956,  35.977725      ],
                              [  0.            ,   0.            ,  37.335375      ],
                              [  1.920007042956,  -0.            ,  38.693025      ],
                              [  1.920007042956,   1.920007042956,  40.050675      ],
                              [  0.            ,   1.920007042956,  41.408325      ],
                              [  0.            ,   0.            ,  42.765975      ],
                              [  1.920007042956,  -0.            ,  44.123625      ],
                              [  1.920007042956,   1.920007042956,  45.481275      ],
                              [  0.            ,   1.920007042956,  46.838925      ],
                              [  0.            ,   0.            ,  48.196575      ],
                              [  1.920007042956,  -0.            ,  49.554225      ],
                              [  1.920007042956,   1.920007042956,  50.911875      ],
                              [  0.            ,   1.920007042956,  52.269525      ],
                              [  0.            ,   0.            ,  53.627175      ],
                              [  1.920007042956,  -0.            ,  54.984825      ],
                              [  1.920007042956,   1.920007042956,  56.342475      ],
                              [  0.            ,   1.920007042956,  57.700125      ],
                              [  0.            ,   0.            ,  59.057775      ],
                              [  1.920007042956,  -0.            ,  60.415425      ],
                              [  1.920007042956,   1.920007042956,  61.773075      ],
```

(continues on next page)

(continued from previous page)

```
                                [  0.            ,   1.920007042956,  63.130725      ],
                                [  0.            ,   0.            ,  64.488375      ],
                                [  1.920007042956,  -0.            ,  65.846025      ],
                                [  1.920007042956,   1.920007042956,  67.203675      ],
                                [  0.            ,   1.920007042956,  68.561325      ],
                                [  0.            ,   0.            ,  69.918975      ],
                                [  1.920007042956,  -0.            ,  71.276625      ],
                                [  1.920007042956,   1.920007042956,  72.634275      ],
                                [  0.            ,   1.920007042956,  73.991925      ],
                                [  0.            ,   0.            ,  75.349575      ␣
↪]]*Angstrom

# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )

# Add external potential
external_potential = AtomicCompensationCharge([
    ('doping_0', 0.0200195107106),
    ('doping_1', -0.0200195107106)
    ])

central_region.setExternalPotential(external_potential)

device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode],
    equivalent_electrode_lengths=[10.8612, 10.861199999999997]*Angstrom,
    )

# Add tags
device_configuration.addTags('doping_0', [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
↪ 12,
                                           13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
↪ 25,
                                           26, 27])
device_configuration.addTags('doping_1', [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↪ 40,
                                           41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52,
↪ 53,
                                           54, 55])

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
```

(continues on next page)

(continued from previous page)

```
    BasisGGASG15.Silicon_Low,
    ]

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = GGAHalf.PBE

#----------------------------------------
# Numerical Accuracy Settings
#----------------------------------------
left_electrode_k_point_sampling = MonkhorstPackGrid(
    na=4,
    nb=4,
    nc=87,
    )
left_electrode_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=left_electrode_k_point_sampling,
    density_mesh_cutoff=45.0*Hartree,
    )

right_electrode_k_point_sampling = MonkhorstPackGrid(
    na=4,
    nb=4,
    nc=87,
    )
right_electrode_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=right_electrode_k_point_sampling,
    density_mesh_cutoff=45.0*Hartree,
    )

device_k_point_sampling = MonkhorstPackGrid(
    na=4,
    nb=4,
    nc=87,
    )
device_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=device_k_point_sampling,
    density_mesh_cutoff=45.0*Hartree,
    )

#----------------------------------------
# Poisson Solver Settings
#----------------------------------------
left_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

right_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
```

(continues on next page)

(continued from previous page)

```
                          [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                          [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

#----------------------------------------
# Contour Integral Settings
#----------------------------------------
equilibrium_contour = SemiCircleContour(
    integral_lower_bound=1.43322362959*Hartree,
    )
contour_parameters = ContourParameters(
    equilibrium_contour=equilibrium_contour,
    )

#----------------------------------------
# Electrode Calculators
#----------------------------------------
left_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=left_electrode_numerical_accuracy_parameters,
    poisson_solver=left_electrode_poisson_solver,
    )

right_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=right_electrode_numerical_accuracy_parameters,
    poisson_solver=right_electrode_poisson_solver,
    )

#----------------------------------------
# Device Calculator
#----------------------------------------
calculator = DeviceLCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=device_numerical_accuracy_parameters,
    contour_parameters=contour_parameters,
    electrode_calculators=
        [left_electrode_calculator, right_electrode_calculator],
    )

device_configuration.setCalculator(calculator)
nlprint(device_configuration)
device_configuration.update()
nlsave('silicon_pn_photocurrent.hdf5', device_configuration)

# ------------------------------------------------------------
# Photocurrent
# ------------------------------------------------------------
kpoints = MonkhorstPackGrid(
```

(continues on next page)

(continued from previous page)

```
    na=1,
    nb=1,
    )

photocurrent = Photocurrent(
    configuration=device_configuration,
    energies=numpy.linspace(-0.1, 4.1, 51)*eV,
    photon_energies=numpy.linspace(1, 4, 5)*eV,
    kpoints=kpoints,
    photon_polarization=[0+0j, 0+0j, 1+0j],
    )
nlsave('silicon_pn_photocurrent.hdf5', photocurrent)
```

Notes

Notice that in the script above the *Photocurrent* is only calculated for a single k-point and a few photon energies in order to have a simple example. For converged calculations one needs to increase the k-point sampling to something like:

```
kpoints = MonkhorstPackGrid(
  na=21,
  nb=21,
  )
```

It is important that the `energies` cover all possible transitions. Internally, the program will perform calculations at energy points E and $E - \hbar\omega$, which means that the lower energy limit should be slightly below (due to thermal energy smearing) the lowest electrode chemical potential (in the example above this is `0.0*eV`). The highest energy should be slightly larger than the highest photon energy, i.e., something like:

```
energies = numpy.linspace(- 0.1, max(photon_energies) + 0.1, number_of_points) * eV
```

The calculation time for a *Photocurrent* object depends on the number of k-points, energy points, and photon energy points. For a single photon energy, the calculation will take around two to three times the time for a normal *TransmissionSpectrum* calculation with the same energies and k-point sampling. However, for many photon energies the calculations are often rather time-consuming.

Theory

The photocurrent is calculated using first-order perturbation theory within the 1st Born approximation. The implementation follows the theory in Ref.[1] and the implementation described in Refs.[2] and[3].

In brief, the electron-light interaction is added to the Hamiltonian as:

$$\hat{H} = \hat{H}_0 + \frac{e}{m_0}\mathbf{A}\cdot\hat{\mathbf{p}},$$

[1] LindorăE. Henrickson. Nonequilibrium photocurrent modeling in resonant tunneling photodetectors. *Journal of Applied Physics*, 91(10):6273–6281, 2002. URL: http://aip.scitation.org/doi/abs/10.1063/1.1473677, doi:10.1063/1.1473677.

[2] Jingzhe Chen, Yibin Hu, and Hong Guo. First-principles analysis of photocurrent in graphene PN junctions. *Phys. Rev. B*, 85:155441, 2012. URL: https://link.aps.org/doi/10.1103/PhysRevB.85.155441, doi:10.1103/PhysRevB.85.155441.

[3] Lei Zhang, Kui Gong, Jingzhe Chen, Lei Liu, YuăZhu, DiăXiao, and Hong Guo. Generation and transport of valley-polarized current in transition-metal dichalcogenides. *Phys. Rev. B*, 90:195428, 2014.

where $\hat{H}_0$ is the Hamiltonian without the electron-light interaction, e is the electron charge, m_0 is the free electron mass, $\hat{p}$ is the momentum operator, and $\mathbf{A}$ is the electromagnetic vector potential. Assuming a single-mode monochromatic light source, we have:

$$\hat{A} = \left(\frac{\hbar\sqrt{\mu_r\epsilon_r}}{2N\omega\epsilon c} I_\omega \right)^{1/2} \left(\mathbf{a}\hat{b}e^{-i\omega t} + \mathbf{a}^*\hat{b}^\dagger e^{i\omega t} \right),$$

where $\mathbf{a}$ is the (possibly complex) polarization vector, μ_r is the relative magnetic susceptibility and ϵ_r the relative dielectric constant, ϵ is the dielectric constant, and ω and c are the frequency and speed of light, respectively. $\hat{b}^\dagger$ and $\hat{b}$ are the photon creation and annihilation operators. I_ω is the photon flux, defined as the number of photons (N) per unit time per unit area:

$$I_\omega = \frac{Nc}{V\sqrt{\mu_u\epsilon_r}}.$$

The first-order coupling matrix now becomes:

$$M_{ln} = \frac{e}{m_0} \left(\frac{\hbar\sqrt{\mu_r\epsilon_r}}{2N\omega\epsilon c} I_\omega \right)^{1/2} \langle l|\hat{\mathbf{p}} \cdot \mathbf{a}|n\rangle,$$

where $|n\rangle$ is an LCAO basis function.

The 1st Born electron-photon self-energies are:

$$\mathbf{\Sigma}^{>}_{ph} = [N\mathbf{M}^\dagger\mathbf{G}^{>}_0(E^+)\mathbf{M} + (N+1)\mathbf{M}\mathbf{G}^{>}_0(E^-)\mathbf{M}^\dagger]$$

$$\mathbf{\Sigma}^{<}_{ph} = [N\mathbf{M}\mathbf{G}^{<}_0(E^-)\mathbf{M}^\dagger + (N+1)\mathbf{M}^\dagger\mathbf{G}^{>}_0(E^-)\mathbf{M}],$$

where $E^\pm = E \pm \hbar\omega$. The Greens function including electron-photon interactions to first order is then

$$\mathbf{G}^{>/<} = \mathbf{G}^r_0(\mathbf{\Sigma}^{>/<}_L + \mathbf{\Sigma}^{>/<}_R + \mathbf{\Sigma}^{>/<}_{ph})\mathbf{G}^a_0.$$

In the above two equations $\mathbf{G}^{r,>,<}_0$ denote the non-interacting Greens functions, and $\mathbf{\Sigma}^{>/<}_{L,R}$ are the lesser and greater self-energies due to coupling to the electrodes. The current in electrode α (left or right) with spin σ is calculated as:

$$I_{\alpha,\sigma} = \frac{e}{\hbar}\int \frac{dE}{2\pi} \sum_k T_\alpha(E,k,\sigma),$$

where the effective transmission coefficients are given by[Page 3604, 3]:

$$T_\alpha(E,k,\sigma) = \mathrm{Tr}\left\{ i\Gamma_\alpha(E,k)[1-f_\alpha]G^< + f_\alpha G^> \right\}_{\sigma\sigma}.$$

PiezoelectricTensor

class PiezoelectricTensor(*configuration*, *kpoints_a=None*, *kpoints_b=None*, *kpoints_c=None*, *optimize_geometry=None*, *optimize_geometry_parameters=None*, *strain=None*, *symmetrize=None*)

Class for calculating the piezoelectric tensor of a configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration with attached calculator that supports berry-phase polarization calculations.

- **kpoints_a** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the first direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **kpoints_b** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the second direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **kpoints_c** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the third direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **optimize_geometry** (*bool*) – Boolean to control if the internal coordinates should be optimized during strain. **Default:** `False`.
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters for the geometry optimization. Note that the parameter `enable_optimization_stop_file` is not used. **Default:** A default *OptimizeGeometryParameters* object.
- **strain** (*float*) – The magnitude of the strain to be applied. **Default:** `0.01`
- **symmetrize** (*bool*) – Whether to calculate the symmetrized piezoelectric tensor. The symmetries used are those of the input configuration. **Default:** `True`

evaluate()

The piezoelectric tensor d_{kij} is returned as *PhysicalQuantity* array of shape (6, 3). Column *k* contains the values (d_{kxx}, d_{kyy}, d_{kzz}, d_{kyz}, d_{kxz}, d_{kxy}) where *k* is 0 for *x*, 1 for *y*, and 2 for *z*.

Returns
The piezoelectric tensor.

Return type
PhysicalQuantity of type charge per area

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

symmetrize()

Returns
Whether the tensor is symmetrized.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *PiezoelectricTensor* of GaAs in a simple cubic unit cell:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleCubic(5.6537*Angstrom)

# Define elements
elements = [Arsenic, Arsenic, Gallium, Gallium, Arsenic, Arsenic, Gallium,
            Gallium]

# Define coordinates
fractional_coordinates = [[ 0.75,  0.25,  0.75],
                          [ 0.25,  0.75,  0.75],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.75,  0.75,  0.25],
                          [ 0.  ,  0.  ,  0.  ],
                          [ 0.5 ,  0.5 ,  0.  ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 5),
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------
# Piezoelectric tensor
# ------------------------------------------------------------
piezoelectric_tensor = PiezoelectricTensor(
    configuration=bulk_configuration,
    kpoints_a=MonkhorstPackGrid(20,5,5),
```

(continues on next page)

(continued from previous page)

```
    kpoints_b=MonkhorstPackGrid(5,20,5),
    kpoints_c=MonkhorstPackGrid(5,5,20),
    )

nlprint(piezoelectric_tensor)
```

piezo_GaAs.py

The output from the calculation is:

```
+-------------------------------------------------------------------------------+
| Piezoelectric Tensor Report                                                   |
+-------------------------------------------------------------------------------+
|                                                                               |
| Tensor in units of [C/m**2]:                                                  |
|                                                                               |
|           x              y              z                                    |
| xx    7.22801e-20   -7.25228e-20    0.00000e+00                              |
| yy    0.00000e+00    7.22801e-20   -7.22801e-20                              |
| zz   -7.22801e-20   -1.41172e-22    7.22801e-20                              |
| yz   -8.15246e-01   -8.82326e-24    8.82326e-24                              |
| xz    1.10291e-23   -8.15246e-01   -1.21320e-23                              |
| xy   -8.82326e-24    8.82326e-24   -8.15246e-01                              |
|                                                                               |
+-------------------------------------------------------------------------------+
```

Notes

Due to the symmetry of the zinc-blende structure, only the shear components are non-zero. As an example, an xy-shear strain (bottom line) leads to a polarization in the z-direction (rightmost column), and likewise the yz- and xz-shear strains are non-zero.

- The calculated quantity is the *proper piezoelectric tensor*, as defined in ref.[1].
- In the example above, the piezoelectric tensor is calculated assuming homogeneous strains. This is sometimes called the clamped-ion piezoelectric tensor. The experimental tensor will also have a contribution from internal strains, i.e. a relative displacement of the Ga sublattice to the As sublattice. This contribution can be included by setting the parameter optimize_geometry to True.
- The calculated value of $-0.82 C/m^2$ compares well with previous calculations Ref.[2].

[1] David Vanderbilt. Berry-phase theory of proper piezoelectric response. *Journal of Physics and Chemistry of Solids*, 61(2):147–151, 2000.

[2] R.ăD. King-Smith and D.ăVanderbilt. Theory of polarization of crystalline solids. *Phys. Rev. B*, 47:1651–1654, Jan 1993. doi:10.1103/PhysRevB.47.1651.

PlaneWaveCalculator

class PlaneWaveCalculator(*wave_function_cutoff=None*, *basis_set=None*, *exchange_correlation=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *poisson_solver=None*, *algorithm_parameters=None*, *charge=None*, *fixed_spin_moment=None*, *correction_extension=None*, *checkpoint_handler=None*, *store_wave_functions=None*, *processes_per_kpoint=None*, *parallel_parameters=None*)

Class for representing calculations using the ATK-PW plane wave method for BulkConfigurations.

Parameters

- **wave_function_cutoff** (*PhysicalQuantity* of type energy) – The energy cutoff of the plane wave basis functions to include the basis set. This value determines the size of the basis set. See *Wave function cutoff*. **Default:** 0.25 * density_cutoff from numerical_accuracy_parameters
- **basis_set** (*BasisSet* | PAWDataSet) – An object describing the basis set to use. Note that this parameter is only used to determine the pseudopotentials to use. **Default:** BasisGGAPseudoDojo.Medium
- **exchange_correlation** (*ExchangeCorrelation*) – The exchange correlation to use for this calculation. **Default:** GGA.PBE
- **numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the calculation. **Default:**

```
NumericalAccuracyParameters(
    density_cutoff=1.0e-6,
    density_mesh_cutoff=80.0*Hartree,
    interaction_max_range=20.0*Angstrom,
    k_point_sampling=MonkhorstPackGrid(1, 1, 1),
    number_of_reciprocal_points=1024,
    occupation_method=FermiDirac(1000.0*Kelvin),
    radial_step_size=0.001*Bohr,
    reciprocal_energy_cutoff=1250.0*Hartree
    bands_per_electron=1.2)
```

- **iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the calculation. For non-self-consistent, set this to NonSelfconsistent. **Default:**

```
IterationControlParameters(
    tolerance=1.0e-5,
    max_steps=100,
    algorithm=PulayMixer(),
    damping_factor=0.1,
    number_of_history_steps=20,
    start_mixing_after_step=0,
    mixing_variable=HamiltonianVariable,
    linear_dependence_threshold=0.0,
    preconditioner=Preconditioner.Off)
```

- **poisson_solver** (*FastFourierSolver*) – The Poisson solver to calculate the electrostatic potential. **Default:** A default *FastFourierSolver* object
- **algorithm_parameters** (*AlgorithmParameters*) – The *AlgorithmParameters* used for calculating the density matrix. **Default:**

```
AlgorithmParameters(
    density_matrix_method=GeneralizedDavidsonSolver(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    store_energy_density_matrix=Automatic,
    scf_restart_step_length=0.1*Angstrom
    use_symmetries=Automatic)
```

- **charge** (*float*) – The charge of the system, a charge of -1 corresponds to one extra electron. **Default:** `0.0`
- **fixed_spin_moment** (float | `False`) – Total spin moment (per unit cell) to use, defined as $\Delta N = N_\uparrow - N_\downarrow$, where $N_\uparrow$ and $N_\downarrow$ are the number of electrons in the Up and Down spin channels, respectively. When specified the spin moment will be fixed at the given value by introducing separate Fermi levels for the Up and Down spin channels. Can only be specified when doing a calculation with polarized spin. If set to `False` the spin moment will not be fixed - a single Fermi level is used. **Default:** `False`
- **correction_extension** (*GrimmeDFTD2* | *GrimmeDFTD3*) – The correction extension to used, when calculating energy, forces and stress. **Default:** None.
- **checkpoint_handler** (*CheckpointHandler*) – The checkpoint handler used for specifying the save-file and the time interval. **Default:** No checkpoint handling.
- **store_wave_functions** (*bool*) – Flag that decides whether or not the wave functions should be saved along with the potential.

  Default: `True` For Hybrid calculations
  `False` All other cases.
- **processes_per_kpoint** (`Automatic` | int) – The number of processes to use per kpoint. When set to `Automatic` the number will be determined automatically from the total number of k-points and processes such as to keep the number as small as possible and at the same time minimize the number of idle processes. One may set this number manually in order to reduce the memory requirements for each process. **Default:** `Automatic`
- **parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:**

```
ParallelParameters(
    processes_per_neb_image=None,
    processes_per_individual=None,
    processes_per_bias_point=None,
    processes_per_saddle_search=1)
```

algorithmParameters()

Query method for the algorithm parameters.

Returns
The algorithm parameters object.

Return type
AlgorithmParameters

basisSet()

Get the basis set.

Returns
The basis set set on the calculator.

Return type
BasisSet

charge()

Get the charge of the system.

Returns
The charge in units of the electron charge.

Return type
float

checkpointHandler()

Return the checkpoint handler.

correctionExtension()

Returns
The correction extension or None if not set.

Return type
GrimmeDFTD2 | *GrimmeDFTD3* | None

exchangeCorrelation()

Get the exchange-correlation.

Returns
The exchange correlation set on the calculator.

Return type
ExchangeCorrelation

fixedSpinMoment()

Get the fixed spin moment.

Returns
The fixed spin moment or `False` if the spin moment is not held fixed.

Return type
float | `False`

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Query method for the IterationControlParameters.

Returns
The iteration control parameters set on the calculator.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numericalAccuracyParameters()

Query method for the NumericalAccuracyParameters.

Returns
The numerical accuracy parameters set on the calculator.

Return type
NumericalAccuracyParameters

parallelParameters()

Returns
The parallel parameters object.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver set on the calculator.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

processesPerKpoint()

Returns
The number of processes to use per kpoint.

Return type
int

setBasisSet(*basis_set*)

Set the basis set.

Parameters
basis_set (*BasisSet*) – The basis set to check and set.

setCheckpointHandler(*checkpoint_handler*)

Set the checkpoint handler. :param checkpoint_handler: The CheckpointHandler to set on the calculator. :type checkpoint_handler: *CheckpointHandler*

setCorrectionExtension(*correction_extension*)

Set the iteration correction extension.

Parameters
correction_extension (*GrimmeDFTD2* | *GrimmeDFTD3* | None) – The correction extension to use.

setExchangeCorrelation(*exchange_correlation*)

Set the exchange-correlation.

Parameters
exchange_correlation (*ExchangeCorrelation*) – The exchange correlation to set.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The iteration control parameters to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The numerical accuracy parameters to set.

setParallelParameters(*parallel_parameters*)

Method for setting the parallel parameters.

Parameters
parallel_parameters (*ParallelParameters* | None) – The parallel parameters to set. If None a default version of the parameters is used.

setPoissonSolver(*poisson_solver*)

Set the Poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver to set on the calculator.

storeWaveFunctions()

Returns
The flag that decides whether or not the wave functions should be stored on disk.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible. @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

waveFunctionCutoff()

Get the plane wave cutoff energy.

Returns
The plane wave cutoff energy.

Return type
PhysicalQuantity of type energy

Notes

Wave function cutoff

A calculation with the *PlaneWaveCalculator* allows to represent the Bloch states in a systematically improvable discrete Fourier expansion:

$$\psi_{\mathbf{k},n}(\mathbf{r}) = \sum_{\mathbf{G}}^{|\mathbf{k}+\mathbf{G}|<Q_{\max}} C_{\mathbf{k}+\mathbf{G},n}\, e^{i(\mathbf{k}+\mathbf{G})\cdot\mathbf{r}},$$

where $\mathbf{G}$ is a reciprocal lattice vector. The number of plane waves to include in the expansion is determined by the reciprocal lattice vector cutoff length $Q_{\max} = \frac{1}{2}E_{\text{cut}}$ where E_{cut} is the kinetic energy cutoff of the wave functions set by the keyword argument `wave_function_cutoff`. By increasing the wave function cutoff one increases the number of plane wave basis functions used in the calculation. This leads to a better approximation of the exact wave function, but at an increased computational cost.

Density mesh cutoff

Since the density is defined as the square of the wave function amplitudes:

$$\rho(\mathbf{r}) = \sum_{\mathbf{k},n} f(\epsilon_{\mathbf{k},n})|\psi_{\mathbf{k},n}(\mathbf{r})|^2$$

we, in principle, need twice the wave function resolution for the density. By default, we set the `density_mesh_cutoff` on *NumericalAccuracyParameters* to 4 times the `wave_function_cutoff`. In some cases this may be unnecessary, and can be overwritten by the user by specifying it explicitly.

Bands per electron

Since the plane wave basis is typically very large, a direct diagonalization of the Hamiltonian will be very inefficient and very time consuming. Instead iterative eigensolvers (*GeneralizedDavidsonSolver* and *PPCGSolver*) are used to solve for the lowest eigenstates that are needed to describe the ground state. The number of bands to solve for is set by the `bands_per_electron` keyword in *NumericalAccuracyParameters*. One has to be sure to solve for all bands that are fully or partly occupied through the SCF cycle, but keeping the number as low as possible ensures the fastest calculation.

By default `bands_per_electron` is set to 1.2 which corresponds to adding upwards of 20% more bands than there are bands below the Fermi level (for a non-metal). This is a safe setting for semi-conductors. For metals one might need to increase the number of bands, especially when using a large smearing. For insulators fewer bands will likely suffice. For closed shell systems, one can set it to 1.0.

Initialization methods

The iterative eigensolvers generally need a starting guess for the wave functions and the density used in constructing the initial Hamiltonian. The initialization method is set on the eigensolver supplied to the `AlgorithmParameters` and can be one of the following two:

- *BasisSetInitialization* uses the localized atomic orbitals from the *BasisSet* specified as input parameter to generate an initial guess for the wave functions.
- *RandomBlochWaveInitialization* uses random orthogonal wave functions as initial guess.

Usage Examples

Si FCC calculation with default parameters

The following script is an example of how to set up a *PlaneWaveCalculator* for a Si FCC system with all defaults.

```
# Set up a Bravais lattice.
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements.
elements = [Silicon, Silicon]

# Define coordinates.
fractional_coordinates = [[0.0, 0.0, 0.0],
                      [0.25, 0.25, 0.25]]

# Set up configuration.
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates)

# Set up calculator.
calculator = PlaneWaveCalculator()
bulk_configuration.setCalculator(calculator)

# Run the scf loop.
bulk_configuration.update()
```

Accurate Si FCC calculation

If we want to perform a more accurate calculation, with a custom `density_mesh_cutoff`, an example script could be:

```
# Define a non-default cutoff.
wave_function_cutoff = 80 * Hartree

# Make some custom numerical accuracy parameters.
numerical_accuracy_parameters = NumericalAccuracyParameters(
    # Large k-point sampling.
```

(continues on next page)

(continued from previous page)

```
    k_point_sampling=MonkhorstPackGrid(11, 11, 11),
    # Take the density mesh cutoff as 2.5 times the wave function cutoff instead of 4.
    density_mesh_cutoff=2.5 * wave_function_cutoff)

# Converge to a higher than default tolerance.
iteration_control_parameters = IterationControlParameters(tolerance=1.0e-7)

# Set up calculator.
calculator = PlaneWaveCalculator(
    # Take a higher number of bands (default for Si-fcc is 4).
    bands_above_fermi_level=10,
    wave_function_cutoff=wave_function_cutoff,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters)
```

PlumedMetadynamics

class PlumedMetadynamics(*configuration*, *timestep*, *plumed_commands*, *logfile=None*, *temporary_plumed_file=None*, *charges=None*)

Constructor for the PlumedMetadynamics hook class.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration on which the metadynamics MD simulation should be run.
- **timestep** (*PhysicalQuantity* of type time) – The timestep used in the MD simulation.
- **plumed_commands** (*str*) – A string containing the native PLUMED commands, that should be executed by PLUMED during the simulation.
- **logfile** (*str*) – The name of the file, plumed writes its log information to. **Default:** plumed.log
- **temporary_plumed_file** – The name of the temporary file, that is used to write the plumed commands to. **Default:** plumed_tmp.dat
- **charges** (*PhysicalQuantity* of type charge) – The partial charges on the atoms. **Default:** None.

Usage Example

This example script runs a metadynamics simulation of a germanium interstitial in a silicon crystal. The bias potential acts on the cartesian X-coordinate of the germanium atom, gradually pushing it out of its original interstitial site and facilitating its diffusion in X-direction.

```
# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)
```

(continues on next page)

(continued from previous page)

```
method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    friction=0.01*femtoSecond**-1,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

# The bottom layer of atoms will be fixed to provide an absolute reference frame.
fix_atom_indices_0 = [0, 1, 4, 5, 32, 33, 36, 37]
constraints = [FixAtomConstraints(fix_atom_indices_0)]

# Set up the PLUMED command
plumed_command = """\
# Change the units to Ang, fs, eV
UNITS LENGTH=A TIME=fs ENERGY=96.48533645956869
# Use the cartesian position of the Ge interstitial atom as CV.
p: POSITION ATOM=65
# Put a metadynamics bias on the x component of the position CV p.
METAD ARG=p.x SIGMA=0.2 HEIGHT=0.05 PACE=50 LABEL=restraint
# Print the current position and bias potential value every 10 steps.
PRINT ARG=p.x,p.y,p.z,restraint.bias STRIDE=10 FILE=COLVAR
"""

# Set up the Plumed hook-class.
plumed_hook = PlumedMetadynamics(
    configuration=bulk_configuration,
    timestep=1.0*fs,
    plumed_commands=plumed_command,
    logfile='plumed_example.log',
)

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='si_ge_metadynamics_md.hdf5',
    steps=30000,
    log_interval=200,
    post_step_hook=plumed_hook,
    method=method
)

bulk_configuration = md_trajectory.lastImage()
```

Here, only the Molecular Dynamics block of the script is shown. The full script can be found found in the file `script_si_ge_metadynamics.py`

Notes

- This is a hook class around the PLUMED code to run *Metadynamics* simulations[1]. As it modifies the forces and stress, it must be invoked as a `post_step_hook` in a *MolecularDynamics* simulation (more information about hook functions can be found in the *MolecularDynamics* manual).
- Metadynamics is a powerful technique for enhancing sampling in molecular dynamics simulations and reconstructing the free-energy surface as a function of few selected degrees of freedom, often referred to as collective variables (CVs)[2].
- Metadynamics adds an adaptive, history-dependent bias potential, which gradually fills up free energy minima until the system can effortlessly escape these minima and continue exploring the remaining phase space. Once converged, the negative bias potential can be used as an estimate of the free energy profile along the chosen CVs.
- To run a Metadynamics simulation you need to specify the native PLUMED command as a string in the python script, which is then passed to the *PlumedMetadynamics* hook object. See the PLUMED manual for a list of all avaliable CVs and commands.
- Note that PLUMED internally uses the units nm, ps, kJ/mol, meaning that all parameters in the plumed command need to be given in those units. By adding the line.

```
UNITS LENGTH=A TIME=fs ENERGY=96.48533645956869
```

to the plumed commands, you can switch to units Å, fs, and eV, which may be convenient for QuantumATK.

- Moreover, note that the atom indices in PLUMED start at 1, whereas in QuantumATK they start at 0.
- The bias potential is stored in a file (default filename `HILLS`). The free energy profile can be extracted and plotted from that file by invoking the script `analyze_hills.py`. The script may need to be adapted for the specific application at hand.
- It is also possible to run an equilibrium simulation without metadynamics bias and just monitor the values of the selected CVs. This can be achieved by invoking the PRINT command.
- You should cite Ref.[Page 3618, 1] in all publications using PLUMED.

PointDefectLuminescence

class PointDefectLuminescence(*charged_point_defect*, *initial_charge_state*, *final_charge_state*, *supercell_repetitions*, *fractional_displacements=None*, *filename=None*, *object_id=None*, *number_of_processes_per_task=None*, *log_filename_prefix=None*, *save_displaced_configurations=None*)

Perform a study of luminescence spectrum line shapes due to recombination of free carriers with a defect state.

Parameters

- **charged_point_defect** (*ChargedPointDefect*) – The charged point defect study for which to extract the initial and final charge state from.
- **initial_charge_state** (*int*) – The initial charge state of the defect, as a discrete multiple of elementary charge. A charge of `-1` corresponds to one extra electron.
- **final_charge_state** (*int*) – The final charge state of the defect, as a discrete multiple of elementary charge. A charge of `-1` corresponds to one extra electron.

[1] GarethăA. Tribello, Massimiliano Bonomi, Davide Branduardi, Carlo Camilloni, and Giovanni Bussi. PLUMED 2: New feathers for an old bird. *Comput. Phys. Comm.*, 185(2):604–613, 2014. doi:10.1016/j.cpc.2013.09.018.

[2] Barducci, Alessandro and Bonomi, Massimiliano and Parrinello, Michele. Metadynamics. *WIREs: Comput. Mol. Sci.*, 1(5):826–843, 2011. doi:10.1002/wcms.31.

- **supercell_repetitions** (*sequence (size 3) of int*) – The supercell of the (charge point defect study) bulk unit cell given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **fractional_displacements** (*sequence of float*) – The fractional displacements from the center displacement configuration in terms of the generalized configuration coordinate delta. Both the initial and final charge state configuration will be the center displacement configuration. **Default:** `[-0.02, -0.01, 0.0, 0.01, 0.02]`
- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*. **Default:** The file name of the input charge point defect study.
- **object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file. **Default:** `'point_defect_luminescence_' + initial_charge_state + '_' + final_charge_state`
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of processes does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively. **Default:** `None`
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of SRH recombination study. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'point_defect_luminescence_'`
- **save_displaced_configurations** (*bool*) – Whether the displaced updated configuration used to determine the total energies should be saved to disk. If `True`, they can be inspected or reused, possibly saving computational time if running again the study, or if shared with other studies. However the file where the study is saved will be significantly larger. **Default:** `False`

calculateEffectiveVibrationFrequency(*charge_state_id*)

Calculate the effective vibration frequency of the charge state.

Parameters
charge_state_id (`InitialChargeState` | `FinalChargeState`) – The charge state for which the effective vibration frequency is calculated.

Returns
The frequency of the effective phonon mode associated with the given charge state, evaluate by quadratic fit of the total energy as a function of the displacement in generalized configuration coordinates.

Type
PhysicalQuantity of type frequency.

calculateLuminescenceLineshape(*energies=None*, *temperature=None*, *gaussian_broadening=None*)

Calculate the normalized luminescence lineshape.

Parameters

- **energies** (*PhysicalQuantity* of type energy) – The energies at which the luminescence lineshape is evaluated. **Default:** An energy range 2eV below the zero phonon line energy, up to the zero phonon line energy
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the luminescence lineshape is evaluated. **Default:** 300 * Kelvin
- **gaussian_broadening** (*PhysicalQuantity* of type energy) – The standard deviation of the gaussian broadening used instead of a delta function in the phonon selection rules. **Default:** 0.8 times the final state oscillator energy.

static calculateLuminescenceLineshapeFromParameters(*zero_phonon_line_energy*, *generalized_configuration_coordinate_delta*, *initial_state_effective_vibration_frequency*, *final_state_effective_vibration_frequency*, *energies=None*, *temperature=None*, *gaussian_broadening=None*)

Calculate the normalized luminescence lineshape.

Parameters

- **zero_phonon_line_energy** (*PhysicalQuantity* of type energy.) – The zero phonon line energy, i.e. the energy of the transition with zero initial and final vibrational quantum number.
- **generalized_configuration_coordinate_delta** (*PhysicalQuantity* of type sqrt(mass) * length) – The displacement between the initial and final state in generalized configuration coordinates.
- **initial_state_effective_vibration_frequency** (*PhysicalQuantity* of type frequency) – The effective frequency of the harmonic oscillator associated with the initial state.
- **final_state_effective_vibration_frequency** (*PhysicalQuantity* of type frequency) – The effective frequency of the harmonic oscillator associated with the final state.
- **energies** (*PhysicalQuantity* of type energy) – The energies at which the luminescence lineshape is evaluated. **Default:** An energy range 2 eV below the zero phonon line energy and 1 eV above.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the luminescence lineshape is evaluated. **Default:** 300 * Kelvin
- **gaussian_broadening** (*PhysicalQuantity* of type energy) – The standard deviation of the gaussian broadening used instead of a delta function in the phonon selection rules. **Default:** 0.8 times the final state oscillator energy.

calculateZeroPhononLineEnergy()

Calculate the zero phonon line energy.

Returns
The zero phonon line energy.

Type
PhysicalQuantity of type energy.

chargedPointDefect()

Returns
The charged point defect study from which the initial and final charge state is extracted from.

Return type
ChargedPointDefect

configurationCoordinateDelta()

Returns
The calculated change in the configuration coordinate. If not available, returns None.

Return type
PhysicalQuantity of type length | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

displacedConfigurationGeneralizedConfigurationCoordinate(*charge_state_id*, *fractional_displacement*)

Retrieve the generalized configuration coordinate for the displaced configuration.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The generalized configuration coordinate for the displaced configuration. If not available, returns None.

Return type
PhysicalQuantity of type length square root mass | None

displacedConfigurationTotalEnergy(*charge_state_id*, *fractional_displacement*)

Retrieve the total energy for the displaced configuration.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The calculated total energy analysis object for the displaced configuration. If not available, returns None.

Return type
TotalEnergy | None

filename()

Returns
The filename where the study object is stored.

Return type
str

finalChargeState()

Returns
The final charge state of the defect, as a discrete multiple of elementary charge. A charge of -1 corresponds to one extra electron.

Return type
int

fractionalDisplacements()

Returns
The fractional displacements from the center displacement configuration in terms of the generalized configuration coordinate delta.

Return type
sequence of float

generalizedConfigurationCoordinateDelta()

Returns
The calculated change in the generalized configuration coordinate. If not available, returns None.

Return type
PhysicalQuantity of type length square root mass | None

huangRhysFactor(*charge_state_id*)

Evaluate the dimensionless Huang-Rhys factor for the effective vibrational mode associated with a given charge state:

$$S_{i,f} = \frac{1}{2\hbar}(\Delta Q)^2 \Omega_{i,f}$$

Parameters
charge_state_id (InitialChargeState | FinalChargeState) – The charge state for which the Huang-Rhys factor is calculated.

Returns
The calculated Huang-Rhys factor. If not available yet, returns None.

Return type
float | None

initialChargeState()

Returns
The initial charge state of the defect, as a discrete multiple of elementary charge. A charge of -1 corresponds to one extra electron.

Return type
int

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlinfo()

Returns
Structured information about the Study.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

saveDisplacedConfigurations()

Returns
Whether the displaced configurations should be saved.

Return type
bool

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

supercellRepetitions()

Returns
The supercell of the (charge point defect study) bulk unit cell given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
sequence (size 3) of int

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

updatedDisplacedConfiguration(*charge_state_id*, *fractional_displacement*)

Retrieve the updated displaced configuration, if available.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The updated displaced configuration. If not available, returns None.

Return type
BulkConfiguration | None

Polarization

class Polarization(*configuration*, *kpoints_a=None*, *kpoints_b=None*, *kpoints_c=None*, *cartesian_directions=None*)

Class for calculating the polarization of a configuration.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration with attached calculator that supports Berry-phase polarization calculations.
- **kpoints_a** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the first direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **kpoints_b** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the second direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **kpoints_c** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the third direction. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **cartesian_directions** (list of `CartesianDirection.{X,Y,Z}`) – Specify whether the polarization should only be evaluated along specific Cartesian directions. Only supported for orthorhombic lattices. **Default:** `[CartesianDirection.X, CartesianDirection.Y, CartesianDirection.Z]`

cartesianDirections()

Returns
The Cartesian components for which the tensor is evaluated.

Return type
list of `CartesianDirection.{X,Y,Z}`

cartesianQuantum()

Returns the cartesian polarization quantum, which for direction i is $P_i = \frac{|e|R_i}{\Omega}$, where $|e|$ is the electronic charge, R_i is the i' th lattice vector, and Ω is the unit cell volume.

Returns
The cartesian polarization quantum.

Return type
PhysicalQuantity of type charge per area

electronicFractionalPolarization()

Returns
The electronic fractional polarization as a length 3 array corresponding to the x, y, and z direction. Note, that all values are wrapped to the interval [-0.5, 0.5]

Return type
`numpy.array`

ionicFractionalPolarization()

Returns the purely ionic fractional polarization $P_i = \sum_j Z_j^{ion} \tau_j$, where Z_j^{ion} and τ_j are the valence charge and fractional coordinate of atom j .

Returns
The ionic fractional polarization as a length 3 array corresponding to the x, y, and z direction. Note, that all values are wrapped to the interval [-0.5, 0.5]

Return type
numpy.array

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

totalCartesianPolarization()

Returns
The total cartesian polarization.

Return type
PhysicalQuantity of type charge per area

totalFractionalPolarization()

Returns the sum of the electronic and ionic parts polarization.

Returns
The total fractional polarization as a length 3 array corresponding to the x, y, and z direction. Note, that all values are wrapped to the interval [-0.5, 0.5]

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the polarization of tetragonal $BaTiO_3$ using the experimental structure and lattice constants:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
```

(continues on next page)

(continued from previous page)

```
lattice = SimpleTetragonal(3.9945*Angstrom, 4.0335*Angstrom)

# Define elements
elements = [Barium, Titanium, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.      ,  0.      ,  0.      ],
                          [ 0.5     ,  0.5     ,  0.51427 ],
                          [ 0.5     ,  0.5     ,  0.974477],
                          [ 0.5     ,  0.      ,  0.487618],
                          [ 0.      ,  0.5     ,  0.487618]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
    GGABasis.Oxygen_DoubleZetaPolarized,
    GGABasis.Titanium_DoubleZetaPolarized,
    GGABasis.Barium_DoubleZetaPolarized,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = GGA.PBE

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(5, 5, 5),
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------
# Polarization
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
polarization = Polarization(
    configuration=bulk_configuration,
    kpoints_a=MonkhorstPackGrid(11,5,5),
    kpoints_b=MonkhorstPackGrid(5,11,5),
    kpoints_c=MonkhorstPackGrid(5,5,11),
    )

nlprint(polarization)
```

polarization.py

The output from the calculation is:

```
+-------------------------------------------------------------------------------+
| Polarization                                                                  |
+-------------------------------------------------------------------------------+
| Electronic fractional polarization.                                           |
| Values wrapped to the interval [-0.5,0.5]                                     |
|       [ -9.11083600e-12 ]                                                     |
|  Pe=  [  2.57089606e-11 ]                                                     |
|       [ -4.64533955e-01 ]                                                     |
+-------------------------------------------------------------------------------+
| Ionic fractional polarization.                                                |
| Values wrapped to the interval [-0.5,0.5]                                     |
|       [  0.00000000e+00 ]                                                     |
|  Pi=  [  0.00000000e+00 ]                                                     |
|       [ -2.44642000e-01 ]                                                     |
+-------------------------------------------------------------------------------+
| Total fractional polarization. Pt = Pe + Pi.                                  |
| Values wrapped to the interval [-0.5,0.5]                                     |
|       [ -9.11083600e-12 ]                                                     |
|  Pt=  [  2.57089606e-11 ]                                                     |
|       [  2.90824045e-01 ]                                                     |
+-------------------------------------------------------------------------------+
| Total cartesian polarization.                                                 |
|       [ -9.05991488e-12 ]                                                     |
|  Pt=  [  2.55652714e-11 ] C/Meter**2                                          |
|       [  2.92022177e-01 ]                                                     |
+-------------------------------------------------------------------------------+
| Polarization quantum.                                                         |
|       [  9.94410928e-01 ]                                                     |
|  Pq=  [  9.94410928e-01 ] C/Meter**2                                          |
|       [  1.00411978e+00 ]                                                     |
+-------------------------------------------------------------------------------+
```

Notes

•Note that the implementation does not work for metallic systems. Usage in 2D systems should be done with thorough testing of the used settings and results.

•The output contains five calculated quantities. First, the electronic fractional polarization, Pe, is calculated from the Berry phase obtained from the occupied bands, as described in Ref.[1]. The three values correspond to the x,y, and z direction.

•The second quantity, Pi, is the purely ionic fractional polarization $P_i = \sum_j Z_j^{ion} \tau_j$, where Z_j^{ion} and τ_j are the valence charge and fractional coordinate of atom j .

•The third quantity, Pt, is the total fractional polarization which is the sum of the electronic and ionic parts. Note, that all fractional polarizations are wrapped to the interval [-0.5,0.5] which explains the sign change of the polarization in the z- direction. The sum Pe(z) + Pi(z) = -0.709 is outside the range [-0.5,0.5] and is thus wrapped to the interval [-0.5,0.5] by adding a fractional quantization quantum (equal to 1), i.e. Pt(z) = -0.709 + 1 = 0.291.

•It is an important finding of the so-called modern theory of polarization, that the polarization is a multivalued quantity or a lattice, c.f. Ref.[Page 3628, 1]. This is why the values are wrapped to the interval [-0.5,0.5].

•The fourth quantity is the total cartesian polarization given in units of C/m^2.

•The fifth quantity is the cartesian polarization quantum, which for direction i is $P_i = \frac{|e|R_i}{\Omega}$, where $|e|$ is the electronic charge, R_i is the i' th lattice vector, and Ω is the unit cell volume.

PolarizationOrbital

class PolarizationOrbital(*base_orbital*)

Class for representing the polarization of a confined orbital.

Parameters
base_orbital (*ConfinedOrbital*) – The confined orbital from which the polarization orbital should be determined.

angularMomentum()

Returns
The angular momentum.

Return type
int

baseOrbital()

Returns
The confined orbital used for generating the polarization orbital.

Return type
ConfinedOrbital

nlinfo()

Returns
The nlinfo.

Return type
dict

[1] R.ăD. King-Smith and D.ăVanderbilt. Theory of polarization of crystalline solids. *Phys. Rev. B*, 47:1651–1654, Jan 1993. doi:10.1103/PhysRevB.47.1651.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a *BasisSet* for hydrogen:

```
hydrogen_1s = ConfinedOrbital(
    principal_quantum_number=1,
    angular_momentum=0,
    radial_cutoff_radius=5.28603678847*Bohr,
    confinement_start_radius=0.8 * 5.28603678847*Bohr,
    additional_charge=0.0,
    confinement_strength=20.000*Hartree*Bohr,
    radial_step_size=0.001*Bohr,
    )

hydrogen_1s_polarization = PolarizationOrbital(hydrogen_1s)

my_hydrogen_basis = BasisSet(
    element=Hydrogen,
    orbitals=[hydrogen_1s, hydrogen_1s_polarization],
    occupations=[1.0 ,0.0],
    pseudopotential=NormConservingPseudoPotential('normconserving/H.LDAPZ.zip'),
    )
```

Notes

The polarization orbital is generated by perturbing the `base_orbital` (ϕ_l^{base}) by an electric field E pointing in the z-direction:

$$[\hat{H}_0 + Ez](\phi_l^{\text{base}} + \delta\phi) = \varepsilon_l(\phi_l^{\text{base}} + \delta\phi).$$

The perturbed orbital, $\delta\phi$, is calculated by first order perturbation theory and has angular momentum $l + 1$. The polarization orbital is constructed from $\delta\phi$.

Further information about the basis functions can be found in *LCAO basis set*.

PolarizationParameters

class PolarizationParameters(*kpoints_a=None*, *kpoints_b=None*, *kpoints_c=None*, *cartesian_directions=None*)

Class for parameters used to evaluate *Polarization* analysis.

Parameters

- **kpoints_a** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the first direction. **Default:** `MonkhorstPackGrid(n_aa, n_ab, n_ac)`, where *n_aa*, *n_ab*, and *n_ac* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_aa = 2 * na + 1, n_ab = nb, n_ac = nc`.

- **kpoints_b** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the second direction. **Default:** `MonkhorstPackGrid(n_ba, n_bb, n_bc)`, where *n_ba*, *n_bb*, and *n_bc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ba = na, n_bb = 2 * nb + 1, n_bc = nc`.
- **kpoints_c** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-point sampling to use for integrating along the third direction. **Default:** `MonkhorstPackGrid(n_ca, n_cb, n_cc)`, where *n_ca*, *n_cb*, and *n_cc* are determined from the sampling `(na, nb, nc)` used for the self consistent calculation as follows: `n_ca = na, n_cb = nb, n_cc = 2 * nc + 1`.
- **cartesian_directions** (list of `CartesianDirection.{X,Y,Z}`) – Specify whether the polarization should only be evaluated along specific Cartesian directions. Only supported for orthorhombic lattices. **Default:** `[CartesianDirection.X, CartesianDirection.Y, CartesianDirection.Z]`

asDict()

Returns
The input parameters as keyword, value dictionary.

Return type
dict

cartesianDirections()

Returns
The Cartesian components for which the tensor is evaluated.

Return type
list of `CartesianDirection.{X,Y,Z}`

kpointsA()

Returns
The k-point sampling used for integrating along the first direction.

Return type
MonkhorstPackGrid

kpointsB()

Returns
The k-point sampling used for integrating along the second direction.

Return type
MonkhorstPackGrid

kpointsC()

Returns
The k-point sampling used for integrating along the third direction.

Return type
MonkhorstPackGrid

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

PolymerEquilibration

class PolymerEquilibration(*configuration*, *filename*, *object_id*, *log_filename_prefix='polymer_equilibration_'*, *max_temperature=None*, *final_temperature=None*, *max_pressure=None*, *final_pressure=None*, *isotropic_pressure=None*, *time_scale_factor=None*, *log_interval=None*, *trajectory_interval=None*)

Create a polymer simulation study object that executes a 21 step equilibration protocol. This is useful for equilibrating polymer configurations created by the MonteCarloPolymerBuilder to the correct density and temperature for the potential model.

Parameters

- **configuration** (BulkConfiguration) – The configuration to be equilibrated.
- **filename** (*str*) – The full or relative path to save the restart information to.
- **object_id** (*str*) – The object id to use when saving the restart information.
- **log_filename_prefix** (*str*) – The prefix used in generated log files. **Default:** polymer_equilibration_.
- **max_temperature** (*PhysicalQuantity* of type temperature) – The maximum temperature used in the equilibration. **Default:** 1000 Kelvin.
- **final_temperature** (*PhysicalQuantity* of type temperature) – The final temperature the configuration is to be equilibrated to. **Default:** 300 Kelvin.
- **max_pressure** (*PhysicalQuantity* of type pressure) – The maximum pressure used in the equilibration. **Default:** 50000 bar.
- **final_pressure** (*PhysicalQuantity* of type pressure) – The final pressure the configuration is to be equilibrated to. **Default:** 1 bar.
- **isotropic_pressure** (*bool*) – If True, couples the cell vectors with each other and w.r.t. pressure. Isotropic pressure couples the 3 unit cell vectors together, maintaining the cell shape. Turning this off couples the vectors independently, allowing cell shape changes. It is recommended to turn off this option for non-isotropic materials or materials that contain periodic surfaces.
- **time_scale_factor** (*float*) – Factor to scale the total number of MD steps. A scale factor of one corresponds to 1.56 ns of simulation time. **Default:** 1.0.
- **log_interval** (*int*) – The frequency with which entries are written to the log files. **Default:** 100.
- **trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. **Default:** 1000.

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

equilibratedConfiguration()

Return the final equilibrated structure from the equilibration simulation.

Returns
The final equilibrated configuration.

Return type
BulkConfiguration

filename()

Returns
The filename where the study object is stored.

Return type
str

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

trajectory(*step_number*)

Returns
The trajectory segment corresponding to the given step number. If that task has not finished running, then None is returned.

Return type
MDTrajectory | None

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

writeFullTrajectory(*path*, *object_id=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*)

Parameters

- **path** (*str*) – The path to save the full MD trajectory to.
- **object_id** (*str*) – The object id of the trajectory in the HDF5 file. **Default:** Automatically chosen name

Returns
The full trajectory combining all equilibration steps.

Return type
MDTrajectory

Usage Examples

Create a polymer model of poly(methyl methacrylate). The script below uses the *ForceCappedEquilibration* object to remove atomic overlaps in the initial structure. The *PolymerEquilibration* object is then used to equilibrate the generated polymer system so that it is ready for production calculations of the properties of interest.

```
# Load the monomer for the polymer
monomer = nlread('Methyl_Methacrylate.hdf5')[-1]

# Define the polymer sequence as containing 10 chains of 100 monomers each, joined␣
↪atactically.
sequence = PolymerSequence(
    [monomer],
    number_of_monomers=100,
    number_of_chains=10,
)

# Build the initial polymer structure to a correct density for the given polymer.
builder = PolymerMonteCarloBuilder(
    sequence,
    density=1.18*gram*cm**-3,
)
configuration = builder.buildPolymerConfiguration()

# Place an OPLS potential on the configuration as the basis of the force capped␣
↪equilibration.
potential = OPLSPotentialBuilder()
calculator = potential.createCalculator(configuration)
```

(continues on next page)

(continued from previous page)

```
configuration.setCalculator(calculator)

# Equilibrate the structure and save the resulting configuration to disk.
forcecap_equilibrator = ForceCappedEquilibration()
equilibrated_structure = forcecap_equilibrator.runEquilibration(configuration)
nlsave('ForceCapped_Configuration.hdf5', equilibrated_structure)

# Create a polymer equilibration object and run the equilibration process
polymer_equilibrator = PolymerEquilibrator(
    configuration=equilibrated_structure,
    filename='PolymerEquilibrationRestart.hdf5',
    object_id='PolymerEquilibration'
)
polymer_equilibrator.update()

# Return the final equilibrated configuration and save it to disk
final_configuration = polymer_equilibrator.equilibratedConfiguration()
nlsave('Final_Configuration.hdf5', final_configuration)
```

`PolymerEquilibration_Example.py` `Methyl_Methacrylate.hdf5`

Notes

Properly equilibrating polymer models is one of the significant challenges in simulating polymeric materials. As a result of the usually long timescales involved in polymer relaxation, this process is often accelerated by using cycles of high temperature and high pressure simulations. The *PolymerEquilibration* object automates one such process, by implementing a 21 step protocol for polymer equilibration[1]. The 21 steps involves 7 cycles of high temperature/pressure simulation and annealing. The temperature and pressure are varied between a maximum temperature and pressure and the final temperature and pressure at the desired conditions. Details of each step is shown in the table below.

[1] LaurenăJ. Abbott, KyleăE. Hart, and CorayăM. Colina. Polymatic: a generalized simulated polymerization algorithm for amorphous polymers. *Theor. Chem. Acc.*, 132:1334, 2013. URL: https://link.springer.com/article/10.1007/s00214-013-1334-z, doi:10.1007/s00214-013-1334-z.

Cycle	Ensemble	Time (ps)	Conditions
1	NVT	50	T_{max}
2	NVT	50	T_{final}
3	NPT	50	T_{final}, $0.02 \times P_{max}$
4	NVT	50	T_{max}
5	NVT	100	T_{final}
6	NPT	50	T_{final}, $0.6 \times P_{max}$
7	NVT	50	T_{max}
8	NVT	100	T_{final}
9	NPT	50	T_{final}, P_{max}
10	NVT	50	T_{max}
11	NVT	100	T_{final}
12	NPT	5	T_{final}, $0.5 \times P_{max}$
13	NVT	5	T_{max}
14	NVT	10	T_{final}
15	NPT	5	T_{final}, $0.1 \times P_{max}$
16	NVT	5	T_{max}
17	NVT	10	T_{final}
18	NPT	5	T_{final}, $0.01 \times P_{max}$
19	NVT	5	T_{max}
20	NVT	10	T_{final}
21	NPT	800	T_{final}, P_{final}

The *PolymerEquilibration* object has 3 required arguments. These are a *BulkConfiguration* object contining the polymer system to be equilibrated, a file where ongoing details of the calculation can be written and an object id for the data in the file. As equilibration may take a significant amount of computer time, PolymerEquilibration objects are restartable. If the simulation is interupted at some point it can be restarted from that point by calling the same *PolymerEquilibration* creation and update commands. The necessary information for the restart is contained in the file specified in the required filename argument.

The *PolymerEquilibration* class also has a number of optional arguments to control the equilibration process. The maximum and final temperatures can be defined using the `max_temperature` and `final_temperature` arguments respectively. Similarly the maximum and final pressures can be set using the `max_pressure` and `final_pressure` arguments. The coupling of pressure with the lattice vectors is set with the parameter `isotropic_pressure`. Setting this to `True` couples the three lattice together with pressure allowing the unit cell volume to vary. Setting this to `False` couples the three lattice vectors independently to pressure, allowing the cell lengths to change differently. This is useful in cases where the system is anisotropic, such as in polymer-surface interfaces. The number of molecular dynamics steps can be controlled using the `scale` argument. This provides a scaling factor for the number of steps and also the total simulation time used in the equilibration. By default the equilibration performs 1,560,000 steps, requiring 1.56ns of simulation time. This can be increased or decreased using the `scale` argument. Output of each equilibration trajectory to a file can also be controlled by using the `trajectory_filename` argument to specify a file to write to and the `trajectory_interval` argument to specify the frequency of writing to the trajectory. Output to log files can also be controlled using the `log_filename_prefix` and `log_interval` commands. During the equilibration a new log file is created for each step in the equilibration process. The prefix of each filename can be specified using the `log_filename_prefix` argument, while the frequency of writing to the log file is controlled using the `log_interval` argument.

Once the *PolymerEquilibration* object is created the equilibration is run using the `update` method. Once the equilibration is complete the final equilibrated configuration can be returned using the `equilibratedConfiguration` method.

PolymerMonteCarloBuilder

class PolymerMonteCarloBuilder(*polymer_sequence*, *density=None*, *host_configuration=None*, *random_seed=None*, *backbone_angle=None*, *monte_carlo_temperature=None*, *angle_sampling_points=None*, *included_molecules=None*, *self_avoiding=None*, *rotate_sidechains=None*, *repack_molecules=None*, *local_interacting_monomers=None*, *monte_carlo_steps_per_monomer=None*, *potential_builder=None*)

Create a PolymerMonteCarloBuilder object, which is used to create polymer melt structures. Polymer chains of specified sequences are created with Monte Carlo randomized torsions.

Parameters

- **polymer_sequence** (*PolymerSequence* | list) – The polymer sequence. Can be a single sequence or a list of sequences, in which case a polymer blend of the given sequences would be generated.
- **density** (*PhysicalQuantity* of type mass / volume | None) – Target density for the final polymer melt. Must be None if a host configuration is given.
- **host_configuration** (*BulkConfiguration* | None) – The bulk configuration in which the chains are placed. Typically an empty configuration with the desired cell size, but it can also contain other molecules or particles. Must be None if a density value is given.
- **random_seed** (*int | None*) – Seed for the random number generator.
- **backbone_angle** (*PhysicalQuantity* of type angle | None | RandomAngle) – Set the backbone torsions to this angle. None means that the angles are sampled based on their torsion energy. RandomAngle sets the torsions randomly
- **monte_carlo_temperature** (*PhysicalQuantity* of type temperature) – The temperature at which the backbone torsion angles are sampled. **Default:** 300 K.
- **angle_sampling_points** (*int*) – The number of points that are used to map sample the backbone torsion energies. **Default:** 20
- **included_molecules** (*List of tuples of MoleculeConfiguration and int.*) – List of tuples containing molecular configurations and numbers of each molecule to be included in the melt **Default:** None
- **self_avoiding** (*bool*) – Whether or not the polymer is initially built to avoid overlapping itself. **Default:** True.
- **rotate_sidechains** (*bool*) – Whether or not the torsions on the sidechains are randomized. **Default:** True.
- **repack_molecules** (*bool*) – Whether or not the polymer chains are repacked with the Monte Carlo density algorithm. **Default:** True if a host configuration is included, False otherwise.
- **local_interacting_monomers** (*int*) – The number of local monomers left and right to the central monomer, that are included in the Monte Carlo energy calculation. **Default:** 3
- **monte_carlo_steps_per_monomer** (*int*) – Sets the number of additional Monte Carlo torsion randomization steps that are carried out based on the number of monomers for the chain **Default:** 0
- **potential_builder** (*DreidingPotentialBuilder* | *OPLSPotentialBuilder* | *UFFPotentialBuilder*) – A potential builder object that creates the potential used in setting the torsions of the chains. **Default:** *DreidingPotentialBuilder*

buildPolymerConfiguration(*attach_calculator=True*)

Run the building process.

Parameters
attach_calculator (*bool*) – A flag to tell if the full calculator should be attached to the configuration.

Returns
The final configuration containing the polymer melt.

Return type
BulkConfiguration

monomerList(*sequence_index*)

Returns the list of monomers in the given polymer sequence.

Parameters
sequence_index (*int*) – The sequence index for the sequence that the monomers are from.

Returns
The list of monomers.

Return type
list

previewConfiguration()

Generates an unoptimized preview configuration that can be used in the Workflow Builder or to set up a force field.

Returns
The preview configuration containing the same chemical composition as the final polymer.

Return type
BulkConfiguration

Usage Examples

Create a configuration of a blend of poly(vinyl chloride) and poly(methyl methacrylate) using the OPLS-AA potential.

```
# ------------------------------------------
# Monomer Molecule 1
# ------------------------------------------

pvc_monomer = nlread('PVC_Monomer.hdf5')[-1]
pmma_monomer = nlread('PMMA_Monomer.hdf5')[-1]

# Create the polymer sequence object
polymer_sequence_pvc = PolymerSequence(
    monomer_configurations=[pvc_monomer],
    number_of_monomers=20,
    number_of_chains=10,
    tactic_ratio=0.5,
)

# Create the polymer sequence object
polymer_sequence_pmma = PolymerSequence(
    monomer_configurations=[pmma_monomer],
    number_of_monomers=20,
    number_of_chains=10,
    tactic_ratio=0.5,
)

# ------------------------------------------------------------
# Polymer Monte Carlo Builder
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
potential_builder = OPLSPotentialBuilder(
    include_electrostatic=False
)

polymer_builder = PolymerMonteCarloBuilder(
    polymer_sequence=[polymer_sequence_pvc, polymer_sequence_pmma],
    density=1.3000*gram/cm**3,
    monte_carlo_temperature=300.0*Kelvin,
    angle_sampling_points=20,
    repack_molecules=False,
    potential_builder=potential_builder
)
bulk_configuration = polymer_builder.buildPolymerConfiguration()
nlsave('PVC_PMMA_Polymer.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Force Capped Equilibration
# ------------------------------------------------------------
equilibration_method = ForceCappedEquilibration(
    temperature=300.00*Kelvin,
    md_steps_per_force_capped_simulation=40000,
    md_time_step=0.50*fs,
    force_capped_simulations=4,
    starting_factor=1.040,
    ending_factor=0.800,
)
bulk_configuration = equilibration_method.runEquilibration(bulk_configuration)
nlsave('PVC_PMMA_Polymer.hdf5', bulk_configuration)
```

`Polymer_Blend_Example.py` `PVC_Monomer.hdf5` `PMMA_Monomer.hdf5`

Create an aluminum and teflon metal/polymer interface using the DREIDING potential.

```
# ------------------------------------------------------------
# Configurations
# ------------------------------------------------------------
surface_slab = nlread('Aluminum_Slab.hdf5')[-1]
teflon_monomer = nlread('Teflon_Monomer.hdf5')[-1]

# ------------------------------------------------------------
# Teflon Polymer Sequence
# ------------------------------------------------------------
polymer_sequence = PolymerSequence(
    monomer_configurations=[teflon_monomer],
    number_of_monomers=20,
    number_of_chains=9,
    tactic_ratio=0.5,
)

# ------------------------------------------------------------
# Polymer Monte Carlo Builder
# ------------------------------------------------------------
polymer_builder = PolymerMonteCarloBuilder(
```

(continues on next page)

(continued from previous page)

```
    polymer_sequence=polymer_sequence,
    host_configuration=surface_slab,
    monte_carlo_temperature=300.0*Kelvin,
    angle_sampling_points=20,
    repack_molecules=True,
)
bulk_configuration = polymer_builder.buildPolymerConfiguration()
nlsave('Teflon_Aluminum_Interface.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Force Capped Equilibration
# ------------------------------------------------------------
equilibration_method = ForceCappedEquilibration(
    temperature=300.00*Kelvin,
    md_steps_per_force_capped_simulation=40000,
    md_time_step=0.50*fs,
    force_capped_simulations=4,
    starting_factor=1.040,
    ending_factor=0.800,
    fixed_indices=bulk_configuration.indicesFromTags('HOST_CONFIGURATION'),
)
bulk_configuration = equilibration_method.runEquilibration(bulk_configuration)
nlsave('Teflon_Aluminum_Interface.hdf5', bulk_configuration)
```

`Polymer_Surface_Interface_Example.py` `Aluminum_Slab.hdf5` `Teflon_Monomer.hdf5`

Notes

The PolymerMonteCarloBuilder class enables the generation of different types of polymer configurations. Structures such as polymer melts and blends, polymer/solid interfaces and polymers with included molecules can be built.

To create a configuration containing polymers, it is first necessary to create a *PolymerSequence* object for each type of polymer chain in the configuration. These define the types of polymers constructed by the *PolymerMonteCarloBuilder*. The polymers defined by the *PolymerSequence* can contain some randomness, so each chain is not always chemically identical. Any number of polymer sequences can be defined, allowing for the creation of different types of polymer blends.

Once the polymers in the configuration are defined, an instance of the *PolymerMonteCarloBuilder* is created. The constructor for this class contains a number of arguments that controls how the polymer system is to be built. One set of arguments determines the bulk cell and the components placed within the polymer configuration. in addition to the polymers defined by the list of *PolymerSequence* objects the polymer configurations can optionally contain a host configuration and included molecules. These are specified with the arguments `host configuration` and `included_molecules` respectively. The `host_configuration` takes a *BulkConfiguration*, and is typically used to create polymer/surface interfaces or polymers containing nanoparticles. The argument `included_molecules` takes a list of tuples containing a molecule configuration and the number of that molecule to be included in the polymer matrix. This can be used to add plasticizers or other molecules dissolved within the polymer.

When building the polymer system, if a host configuration not is provided then the size of the polymer bulk is determined by the polymer density. This is specified with the `density` argument. Here a cubic translation cell is constructed such that the final density of the polymer configuration matches the specified density. If a host configuration is given then the specified polymers and included molecules are placed directly within the host configuration. If a pure polymer configuration in a specific translation cell shape is desired, this can be created by giving an empty configuration as the host configuration. When using host configurations it is often necessary to include a repacking step. This can be

done by setting the `repack_molecules` argument to `True`. This uses the `RepackMoleculesMonteCarlo` class to repack the polymer molecules so as to even out the density within the simulation cell[1][2]. This has the general effect of taking polymers out of surfaces or other solid structures where they may have been inserted, and placing them within the empty space in the host configuration. This step can also be added for configurations without an initial host.

There are also a number of options that control how individual polymer chains are built. By default the *PolymerMonteCarloBuilder* constructs polymers with randomized torsion angles. For each torsion angle several values are trialled and one is selected based on a Boltzmann weighting of the energy of each torsion angle. To prevent the ends of the polymer chain from attracting each other, when calculating the energy only a small number of monomers either side of the monomer being modified are included in the non-bonding calculation[3]. The number of monomers considered can be set with the argument `local_interacting_monomers`.

In the Monte Carlo process which modifies the torsion angles, the argument `monte_carlo_temperature` sets the temperature to be used for Boltzmann weighting each torsion. This determines how important changes in energy are in selecting a torsion angle. The number of torsion angles trialled for each bond is set with the argument `angle_sampling_points`. Non-bonding interactions can be ignored in calculating the energy of each torsion angle using the argument `self_avoiding`. If this is set to `False`, non-bonding interactions are ignored while valence bonding terms, including the bond torsion potential, are still included. The randomization of side-chain torsions, that is torsions not along the backbone of the polymer, is specified with the `rotate_sidechains` argument. This can be important in cases where polymers contain bulky and/or non-conjugated side groups. The Monte Carlo angle randomization can also be turned off completely using the `backbone_angle` argument. Giving an angle to this argument causes each polymer chain to be built with that torsion angle along the backbone. This can be used to create straight linear polymer chains. The torsion angles can also be uniformly randomized by passing the NLFlag `RandomAngle` to this argument.

The potential used to determine the torsional and non-bonded energies along each polymer chain can also be specified using the argument `potential_builder`. This argument takes an instance of one of the potential builder classes. The type of builder determines which potential is used. By default the *DreidingPotentialBuilder* is used, but it is also possible to use the *OPLSPotentialBuilder* or *UFFPotentialBuilder* classes. In general, due to the difficulty of correctly estimating electrostatic interactions in highly distorted structures, electrostatic interactions are ignored by both the *PolymerMonteCarloBuilder* class and also the *ForceCappedEquilibration* class. Moreover, it is recommended that potential builders should have electrostatic interactions explicitly excluded with the argument `include_electrostatic` set to `False`. This is to avoid calculating the *QEqAtomicCharges* (e.g. used within in the :class:` ~.DreidingPotentialBuilder` or *UFFPotentialBuilder*) on preliminary configurations with unfavorable close atom contacts, which may cause problems in some cases.

When using the Monte Carlo procedure to set the torsions for the polymer chain, the torsions are first adjusted for each monomer sequentially along the chain. After this initial randomization it is possible to perform additional Monte Carlo torsion moves on the torsions for each monomer. In this second cycle the ordering of the monomers is selected at random. This second cycle is enabled with the argument `monte_carlo_steps_per_monomer`. This takes an integer which represents the number of additional torsion randomization steps performed per monomer. This allows for longer polymer chains to have a number of Monte Carlo torsion steps in accordance with its length. Setting this parameter to either `None` or `0` skips these additional Monte Carlo torsion steps.

To aid in the analysis of polymer structures generated with the PolymerMonteCarloBuilder, tags describing different parts of the configuration are added. Each polymer molecule is tagged with the label `POLYMER_MOLECULE_#`, where `#` represents and index starting from zero. Included molecules are labeled with tag `INCLUDED_MOLECULE_#` where `#` is a counter of the types of molecules included. The label `HOST_CONFIGURATION` is added to atoms from an initial host configuration used as the starting point for the configuration. In addition to these components of the configuration the

[1] LiviaăA. Moreira, Guojie Zhang, Franziska Muller, Torsten Stuehn, and Kurt Kremer. Direct equilibration and characterization of polymer melts for computer simulations. *Macromol. Theory Simul.*, 24:419, June 2015. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mats.201500013, doi:10.1002/mats.201500013.

[2] Antonio DeăNicola, Toshihiro Kawakatsu, and Giuseppe Milano. Generation of well-relaxed all-atom models of large molecular weight polymer melts: A hybrid particle-continuum approach based on particle-field molecular dynamics simulations. *J. Chem. Theory. Comput.*, 10:5651, October 2014. URL: https://pubs.acs.org/doi/abs/10.1021/ct500492h, doi:10.1021/ct500492h.

[3] Panagiotis-Nikolaos Tzounis, StefanosăD. Anogiannakis, and DorosăN. Theodorou. General methodology for estimating the stiffness of polymer chains from Their chemical constitution: A single unperturbed chain Monte Carlo algorithm. *Macromolecules*, 50:4575, June 2017. URL: https://pubs.acs.org/doi/abs/10.1021/acs.macromol.7b00645, doi:10.1021/acs.macromol.7b00645.

ends of each polymer chain, defined as the head and tail atoms of the monomers at the end of the chains, are labeled with the tags `HEAD_CONNECT` and `TAIL_CONNECT`. These polymer tags and end tags are used in some types of polymer structural analysis.

Once the *PolymerMonteCarloBuilder* class has been created, the polymer system can be built using the `buildPolymerConfiguration()` function. This returns a *BulkConfiguration* containing the polymer chains placed somewhat randomly in the cell. Due to the randomized placement atom overlaps may be present in the returned structure. These can be resolved by pre-equilibrating the structure using the *ForceCappedEquilibration* class. The calculator used to determine the torsional energies can also be attached to the returned configuration using the argument `attach_calculator`. As building large configurations can be time consuming, it is also possible to rapidly create a non-randomized preview configuration. This can be useful in providing a template for creating calculators with various potentials. The preview configuration is returned by the function `previewConfiguration()`. The single monomers defined in each polymer sequence can be returned with the function `monomerList()`.

PolymerSegmentAnalyzer

class PolymerSegmentAnalyzer(*md_trajectory*, *polymer_tags=None*, *end_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *atoms_per_segment=None*, *gui_worker=None*)

Create an anylzer to calculate the segment distances in polymer chains.

Parameters

- **md_trajectory** (*MDTrajectory* | `AtomicConfiguration`) – The MDTrajectory or AtomicConfiguration that contains the polymers.
- **polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used. Tags can also be set automatically with the flag Automatic
- **end_tags** (*list of type str | None*) – A list of tags for the ends of each polymer chain. If not specified default tags are used.
- **start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **atoms_per_segment** (*int*) – The number of atoms that make up one monomer or segment. **Default:** 1.
- **gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

averageSegmentLength(*polymer_index*)

Return the average segment length for the given polymer.

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The average segment length.

Return type
PhysicalQuantity of type length

backboneAtomIndexes(*polymer_index*)

Return the atomic indexes of the backbone atoms for the given polymer.

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The index of the backbone atoms for the given polymer chain.

Return type
list of int

bondCorrelation(*polymer_index*)

Return the bond correlation function for the segments for the given polymer.

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The bond correlation for the given polymer chain.

Return type
ndarray

characteristicRatio(*polymer_index*)

Return the characteristic ratio for the given polymer chain.

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The characteristic array, index by segment distance, for the given polymer chain.

Return type
ndarray

polymerTags()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

rmsSegmentDistance(*polymer_index*)

Return the RMS distance between segments separated by a certain number of segments. These distances are averaged over the entire polymer chain

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The RMS segment-segment distance for the given polymer chain.

Return type
PhysicalQuantity of type length

segmentPositions(*polymer_index*)

Return the positions of the segments for the given polymer.

Parameters
polymer_index (*int*) – An index to the polymer chain, based on the order specified by the polymer chain tags.

Returns
The index of the segment positions for the given polymer chain.

Return type
PhysicalQuantity of type length

times()

Returns
Times for each image used in the analysis.

Return type
PhysicalQuantity of type time

Usage Examples

Calculate the bond correlation and characteristic ratio of a polystyrene melt during a molecular dynamics simulation starting from a non-equilibrated structure.

```
# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('Polystyrene.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Polystyrene_Trajectory.hdf5',
    steps=100000,
    log_interval=10,
```

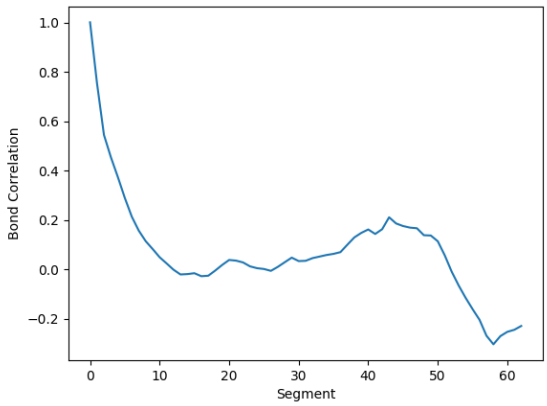
(continues on next page)

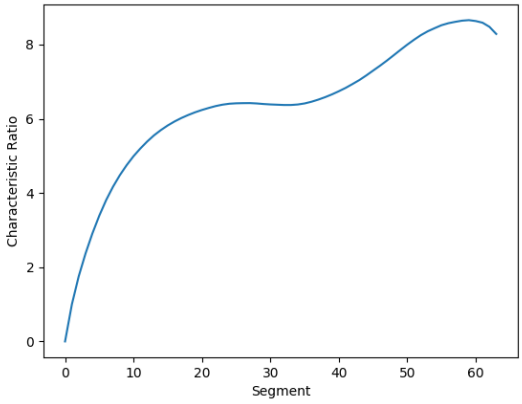
(continued from previous page)

```
    trajectory_interval=1000,
    method=method
)

# ------------------------------------------------------------
# Calculate End to End Distance
# ------------------------------------------------------------
analyzer = PolymerSegmentAnalyzer(
      md_trajectory,
      start_time=0*ps,
      end_time=100*ps,
      end_tags=['HEAD_CONNECT', 'TAIL_CONNECT'],
      atoms_per_segment=2
)

number_of_segments = 64
number_of_chains = 8

bond_correlation = numpy.zeros(number_of_segments-1)
characteristic_ratio = numpy.zeros(number_of_segments)
segments = numpy.arange(number_of_segments)

for i in range(number_of_chains):
    bond_correlation += analyzer.bondCorrelation(i)
    characteristic_ratio += analyzer.characteristicRatio(i)

bond_correlation /= number_of_chains
characteristic_ratio /= number_of_chains

# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
import pylab
pylab.figure()
pylab.plot(segments[:number_of_segments-1], bond_correlation)
pylab.xlabel('Segment')
pylab.ylabel('Bond Correlation')
pylab.savefig('Bond_Correlation_Plot.png')

pylab.figure()
pylab.plot(segments, characteristic_ratio)
pylab.xlabel('Segment')
pylab.ylabel('Characteristic Ratio')
pylab.savefig('Characteristic_Ratio_Plot.png')
```

`PolymerSegmentAnalyzer_Example.py` `Polystyrene.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate bond correlation function and the characteristic ratio for the polymer system. This is shown in the two plots below.





Notes

This class enables the calculation of the details of the arrangements of different polymer backbone segments from a *MDTrajectory* or *BulkConfiguration*.

From a statistical point of view, a polymer can be viewed as a number of segments forming a chain. These segments may be atoms or monomer units. Ideally this chain should form a self-avoiding random walk, however the chemical nature of the polymer system causes deviations from this ideal case. Analyzing the relative positions of the polymer segments allows the deviations from ideal to be characterized. It can also give insight into the representative quality of the polymer structure[1].

Information about the polymer segments is calculated from the backbone atoms. These are defined as the atoms that lie on the bonded path between the two ends of the polymer chain. When segments are constructed from multiple atoms, the segment position is taken as the average position of the atoms. One important measure of the polymer chain is the bond-bond correlation, R_{bb}, which gives a measure of the decay of geometrical information along the polymer chain. This can be given as:

$$R_{bb}(n) = \frac{\langle \mathbf{b}_i \cdot \mathbf{b}_{i+n} \rangle}{\langle b \rangle^2}$$

Here $\mathbf{b}_i$ is the ith inter-segment distance and $\langle b \rangle$ is the average inter-segment distance.

[1] LiviaăA. Moreira, Guojie Zhang, Franziska Muller, Torsten Stuehn, and Kurt Kremer. Direct equilibration and characterization of polymer melts for computer simulations. *Macromol. Theory Simul.*, 24:419, June 2015. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mats.201500013, doi:10.1002/mats.201500013.

Another important property is the average square distance between segments separated by n bonds, $\langle r_n^2 \rangle$. This property can be used to calculate the characteristic ratio C_n, which can be given as[2]:

$$C_n = \frac{\langle r_n^2 \rangle}{n \langle b \rangle^2}$$

In well equilibrated chains the value of C_n should asymptote to the value C_∞ at large values of n

To calculate the polymer segment descriptors a *PolymerSegmentAnalyzer* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration* and polymer segment properties for the specified configurations. The specific frames in the *MDTrajectory* to analyzed can be selected using the `start_time`, `end_time` and `time_resolution` arguments. The polymer chains analyzed in the configuration can be specified with the `polymer_tags` argument. This takes a list of tags of polymer molecules in the configuration. By default polymers tagged with `POLYMER_MOLECULE_#` are analyzed, where `#` is a numerical index. Tags indicating the start and end of each polymer chain also need to be added. The argument `end_tags` takes a list of strings that contain the tags for the end atoms. These tags can be specified in any order. By default the tags `HEAD_CONNECT` and `TAIL_CONNECT` are used. All the default polymer and end tags are placed on polymer configurations built with the *PolymerMonteCarloBuilder*. It is also possible to control the size of each polymer segment using the `atoms_per_segment` argument. Using this argument monomer units and be considered as comprising segments, rather than individual atoms.

Once the *PolymerSegmentAnalyzer* object is created and the values of each of the descriptors has been calculated, the values can be queried using the remaining functions. Each function takes a required `polymer_index` argument, which specifies the index of the polymer chain for which the data is sought. Data for each chain is returned separately to allow for systems with chains of different lengths. The indices of the atoms that make up the backbone of each polymer can be found using the `backboneAtomIndexes()` function. Likewise the Cartesian positions of each segment is returned by the `segmentPositions()` function. When segments consist of more than one atom these positions will be distinct from the backbone atomic positions. The average segment length of each polymer is returned by the `averageSegmentLength()` function. These functions all return arrays of data, indexed first by the image in the molecular dynamics trajectory that the data is calculated from.

Other available descriptors of the polymer system are averaged over the selected molecular dynamics images, and so the data returned by these functions is indexed over the integer distance between segments. One such function is the `bondCorrelation()` function, which returns the bond correlation function for the polymer averaged over the available molecular dynamics images. In a similar manner `rmsSegmentDistance()` returns the square root of the average square distance, and `characteristicRatio()` returns the previously defined characteristic ratio.

In cases where all of the polymers in the system are the same length, it is possible to easily average these properties over the polymer chains. The included example illustrates how this may be performed for the bond correlation function and the characteristic ratio.

PolymerSequence

class PolymerSequence(*monomer_configurations*, *number_of_monomers=10*, *number_of_chains=1*, *tactic_ratio=None*, *monomer_ratios=None*, *monomer_additions=None*, *monomer_block_sizes=None*, *block_contamination=None*, *head_tail_flip_probability=None*, *alpha_group=None*, *omega_group=None*, *random_seed=None*)

Calculate sequences of polymer monomers so that they can be used by the Monte Carlo polymer builder to create polymer melts. Different types of sequences can be generated depending on the type of polymer being studied.

Parameters

[2] WayneăL. Mattice, CarinăA. Helfer, and AlexeiăP. Sololov. On the relationship between the characteristic ratio of a finite chain, Cn, and the asymptotic limit, Cinf. *Macromolecules*, 36:9924, December 2003. URL: https://pubs.acs.org/doi/full/10.1021/ma0304527?src=recsys, doi:10.1021/ma0304527.

•**monomer_configurations** (list of type MoleculeConfiguration) – A list with each entry containing a *MoleculeConfiguration* containing a monomer to be used in the polymer sequence. The atoms on each end where the polymer connects should be labelled with the tags HEAD_CONNECT and TAIL_CONNECT respectively. The first atom of any groups to be removed to form the polymer should be labeled with the tags HEAD_GROUP_A and TAIL_GROUP_A. These 4 tags are mandatory for each monomor. Optionally in the case of tactic monomers additional groups on each end should be labeled with the tags HEAD_GROUP_B and TAIL_GROUP_B respectively.

•**number_of_monomers** (*int*) – The number of monomers to add into each polymer chain.

•**number_of_chains** (*int*) – The number of chains to generate for the polymer melt

•**tactic_ratio** (*float*) – Determines how biased the selection of monomers is towards the same (0) or alternating (1) monomer addition points, for monomers with two stereochemical substitution groups. These are isotactic and syndiotactic polymers respectively. Atactic random polymers are specified with 0.5, which shows no bias towards either addition point. **Default:** 0.5

•**monomer_ratios** (*1D list of floats*) – A list of the relative amount of each monomer in the polymer system. This, along with monomer_additions, determines how likely each monomer is to be selected in the body of a polymer chain. These ratios alone also specify the probability of a chain being started with each monomer. **Default:** Each monomer is equally likely to be selected.

•**monomer_additions** (*2D list of floats*) – A list of probabilities of adding each monomer onto an existing one. This can be used to make specific alternating structures. For each monomer the probability of adding any other monomer needs to be specified, so that an NxN array is required. **Default:** Each monomer has equal addition probability

•**monomer_block_sizes** (*1D list of integers*) – A list of block sizes for each monomer. Specifying all block sizes as 1 results in a random polymer. **Default:** All the monomer block sizes are set to 1.

•**block_contamination** (*float*) – A float that sets the probability of contamination by other monomers in a block polymer. 0 means each block is strictly created, 1 results in a random co-polymer. **Default:** 0

•**head_tail_flip_probability** (*float | None*) – Probability that a monomer will be flipped so that it connects to join head to head and tail to tail. Monomer orientation is completely randomized by setting this to 0.5. Setting this this probability to 1 will flip all monomers, so that head to tail connections are preserved. **Default:** 0

•**alpha_group** (*MoleculeConfiguration* | None) – Specifies the configuration to be used at the start of the chain building process. In this configuration the tags HEAD_CONNECT and TAIL_CONNECT can point to the same atom, although this is not necessary. As with monomers mandatory HEAD_GROUP_A and TAIL_GROUP_A as well as optional HEAD_GROUP_B and TAIL_GROUP_B can be added.

•**omega_group** (*MoleculeConfiguration* | None) – Specifies the configuration to be used at the end of the chain building process. In this configuration the tags HEAD_CONNECT and TAIL_CONNECT can point to the same atom, although this is not necessary. As with monomers mandatory HEAD_GROUP_A and TAIL_GROUP_A as well as optional HEAD_GROUP_B and TAIL_GROUP_B can be added.

•**random_seed** (*int | None*) – The seed for the random number generator. Must be a positive integer. This can be used so that the same sequence can be generated multiple times. **Default:** Arbitrary random seed

allMonomerConfigurations()

Returns
The list of monomer configurations, including alpha and omega end groups.

Return type
list of MoleculeConfigurations

alphaGroup()

Returns
The configuration used for the alpha group, if set. Returns None if not.

Return type
MoleculeConfiguration | None

`alphaId()`

Returns
The integer ID of the alpha end group, if set. Returns None if not.

Return type
int | None

`blockContamination()`

Returns
The probability of block contamination in the sequence

Return type
float

`headTailFlipProbability()`

Returns
The probability of flipping a monomer so that it attaches head-head and tail-tail.

Return type
float

`monomer(index)`

Returns a single monomer, by reference. The indexing also includes alpha/omega groups.

Parameters
index (*int*) – The monomer configuration to be returned. The numbering starts from zero and goes through first monomers, then end groups, if defined.

Returns
The monomer refered to by index.

Return type
MoleculeConfiguration

`monomerAdditions()`

Returns
The matrix of addition probabilites for the polymer sequence

Return type
2D list of floats

`monomerBlockSizes()`

Returns
The block sizes of each monomer in the polymer sequence

Return type
list of int

`monomerConfigurations()`

Returns
The list of monomer configurations, excluding end groups

Return type
list of *MoleculeConfiguration*

monomerConnectionPoints(*index*)

Calculate the number of connection points that a monomer has, based on counting the number of group tags.

Parameters
index (*int*) – The index of the monomer with the connections being counted. The index includes referencing the alpha/omega groups.

Returns
The number of connection points.

Return type
int

monomerFraction()

Return a list of the fraction of each monomer in the polymer sequence.

Returns
List of the relative fraction of each monomer in the sequence

Return type
1D numpy array of float

monomerRatios()

Returns
The fraction of each monomer in the polymer sequence

Return type
list of floats

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfChains()

Returns
The number of chains in the polymer sequence

Return type
int

numberOfMonomers()

Returns
The number of monomers in each polymer chain

Return type
int

numberOfSelectableMonomers()

Returns
The number of monomers used in creating the polymer sequence, excluding the alpha and omega groups.

Return type
int

numberOfUniqueMonomers()

Returns
The number of unique monomer used in the polymer sequence, including the alpha and omega groups.

Return type
int

omegaGroup()

Returns
The configuration used for the omega group, if set. Returns None if not.

Return type
MoleculeConfiguration | None

omegaId()

Returns
The integer ID of the omega end group, if set. Returns None if not.

Return type
int | None

randomSeed()

Returns
The random seed used for the random number generator if set, else None.

Return type
int | None

sequence()

Returns the generated polymer sequence. This is a list of 2D numpy integer arrays. Each element of the returned list is an ndarray defining a single polymer chain. The ndarray is of size Nx3, where N is the number of monomers. For each monomer in the ndarray the first index defines the identity of the monomer, the second defines the connection point and the third defines the monomer orientation.

Returns
The integer sequence of monomers, tacticites and orientations.

Return type
List of ndarray.

tacticRatio()

Returns
The tactic ratio for the polymer chains

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a monomer of methyl methacrylate, including the correct tags, and create a sequence for isotactic poly(methyl methacrylate)

```
# Define the methyl methacrylate monomer.
# First define elements.
elements = [Carbon, Carbon, Hydrogen, Hydrogen, Hydrogen, Carbon, Hydrogen,
            Hydrogen, Hydrogen, Carbon, Hydrogen, Hydrogen, Hydrogen, Carbon,
            Oxygen, Oxygen, Carbon, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 0.000053397617,  0.759772999835, -0.            ],
                         [-0.000053397617, -0.759772999835,  0.            ],
                         [-1.029486689503, -1.169152751252,  0.            ],
                         [ 0.514760310502, -1.169153624436, -0.89158       ],
                         [ 0.514760310502, -1.169153624436,  0.89158       ],
                         [-0.702972150254,  1.318819958684, -1.217534630654],
                         [-1.778994926992,  1.349871730145, -1.034368525823],
                         [-0.338426412209,  2.328764163503, -1.416490865706],
                         [-0.498848545074,  0.68155340126 , -2.080342189714],
                         [-0.702972150254,  1.318819958684,  1.217534630654],
                         [-1.779477850521,  1.34478390329 ,  1.036425819709],
                         [-0.494517798241,  0.684267886286,  2.08145298674 ],
                         [-0.34232383993 ,  2.330614899434,  1.413534093682],
                         [ 1.40596301868 ,  1.318867926842, -0.            ],
                         [ 1.84446749371 ,  2.344022687006, -0.473701649871],
                         [ 2.276658987984,  0.518045908871,  0.638886174971],
                         [ 3.418813357507,  0.987763274498,  0.630284279326],
                         [ 3.36418399788 ,  2.078158275911,  0.650119210394],
                         [ 3.96408273255 ,  0.634713205819,  1.508066389582],
                         [ 3.938029190653,  0.665721116071, -0.274655366275]]*Angstrom

# Set up configuration
methyl_methacrylate = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)

# Add tags
# Note that the tail groups are on the carbons of the methyl groups, not hydrogens.
methyl_methacrylate.addTags('HEAD_CONNECT', [1])
methyl_methacrylate.addTags('HEAD_GROUP_A', [4])
methyl_methacrylate.addTags('TAIL_CONNECT', [0])
methyl_methacrylate.addTags('TAIL_GROUP_A', [5])
methyl_methacrylate.addTags('TAIL_GROUP_B', [9])

# Add bonds
bonds = [[ 0,  1],
         [ 0,  5],
         [ 0,  9],
         [ 0, 13],
         [ 1,  2],
```

(continues on next page)

(continued from previous page)

```
         [ 1,  3],
         [ 1,  4],
         [ 5,  6],
         [ 5,  7],
         [ 5,  8],
         [ 9, 10],
         [ 9, 11],
         [ 9, 12],
         [13, 14],
         [13, 15],
         [15, 16],
         [16, 17],
         [16, 18],
         [16, 19]]
methyl_methacrylate.setBonds(bonds)

# Create the sequence with a tactic ratio of 0, giving an isotactic sequence.
sequence = PolymerSequence(
    [methyl_methacrylate],
    number_of_monomers=40,
    number_of_chains=20,
    tactic_ratio=0,
)
```

tactic_polymer.py

Define a phenyl endgroup and create a random copolymer of 25% styrene and 75% vinyl chloride, capped with a phenyl endgroup.

```
# Load the monomers into memory
styrene = nlread('Styrene.hdf5', class_type=MoleculeConfiguration)[-1]
vinyl_chloride = nlread('Vinyl_Chloride.hdf5', class_type=MoleculeConfiguration)[-1]

# Define a phenyl endgroup
# Define elements
elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Hydrogen, Hydrogen,
            Hydrogen, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 0.            , -0.000000000011, -1.399795      ],
                         [ 0.            ,  1.212252999995, -0.699760000009],
                         [ 0.            ,  1.212253000005,  0.699759999991],
                         [ 0.            ,  0.000000000011,  1.399795      ],
                         [-0.            , -1.212252999995,  0.699760000009],
                         [-0.            , -1.212253000005, -0.699759999991],
                         [ 0.            , -0.000000000019, -2.500678      ],
                         [ 0.            ,  2.165670999991, -1.250363000016],
                         [ 0.            ,  2.165671000009,  1.250362999984],
                         [ 0.            ,  0.000000000019,  2.500678      ],
                         [-0.            , -2.165670999991,  1.250363000016],
                         [-0.            , -2.165671000009, -1.250362999984]]*Angstrom

# Set up configuration
```

(continues on next page)

(continued from previous page)

```
phenyl = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)

# Add tags
phenyl.addTags('HEAD_CONNECT', [2])
phenyl.addTags('HEAD_GROUP_A', [8])
phenyl.addTags('TAIL_CONNECT', [2])
phenyl.addTags('TAIL_GROUP_A', [8])

# Add bonds
bonds = [[ 0,  1],
         [ 0,  5],
         [ 0,  6],
         [ 1,  2],
         [ 1,  7],
         [ 2,  3],
         [ 2,  8],
         [ 3,  4],
         [ 3,  9],
         [ 4,  5],
         [ 4, 10],
         [ 5, 11]]
phenyl.setBonds(bonds)

# Create the sequence made of 25% styrene and 75% vinyl chloride monomers randomly
# arranged.  Also cap the polymer with the phenyl group.
sequence = PolymerSequence(
    [styrene, vinyl_chloride],
    number_of_monomers=40,
    number_of_chains=20,
    monomer_ratios=[0.25, 0.75],
    omega_group=phenyl,
)
```

random_copolymer.py

Load two *MoleculeConfiguration* containing the monomers for the condensation polymer PET and create an alternating polymer from these monomers

```
# Load the monomers into memory
ethylenediamine = nlread('Ethylene_Glycol.hdf5', class_type=MoleculeConfiguration)[-1]
terephthalic_acid = nlread('Terephthalic_Acid.hdf5', class_type=MoleculeConfiguration)[-
↪1]

# Create the polymer sequence with alternating monomers
# This is done by setting the diagonal elements of monomer_additions to zero,
# and the off-diagonal elements to 1.
sequence = PolymerSequence(
    [ethylenediamine, terephthalic_acid],
    number_of_monomers=40,
    number_of_chains=20,
```

(continues on next page)

(continued from previous page)

```
    monomer_additions=[[0, 1], [1, 0]]
)
```

`condensation_polymer.py Ethylene_Glycol.hdf5 Terephthalic_Acid.hdf5`

Load two *MoleculeConfiguration* containing the monomers methyl acrylate and propylene and create diblock copolymer from these monomers.

```
# Load the monomers into memory
methyl_acrylate = nlread('Methyl_Acrylate.hdf5', class_type=MoleculeConfiguration)[-1]
propylene = nlread('Propylene.hdf5', class_type=MoleculeConfiguration)[-1]

# Create the polymer sequence with two polymer blocks
# This is done by setting the monomer_additions to enforce two different blocks.
# To make an A-B block copolymer the sum of the block sizes need to be as long as
# the polymer chain
sequence = PolymerSequence(
    [methyl_acrylate, propylene],
    number_of_monomers=40,
    number_of_chains=20,
    monomer_additions=[[0, 1], [1, 0]],
    monomer_block_sizes=[10, 30],
)
```

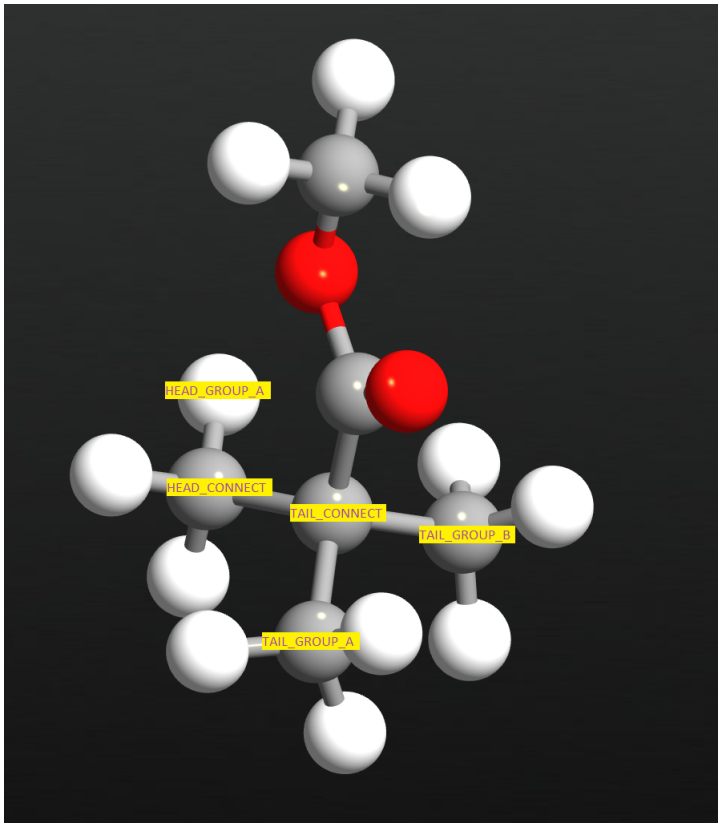
`block_copolymer.py Methyl_Acrylate.hdf5 Propylene.hdf5`

Notes

The PolymerSequence object creates properly randomized sequences of monomers from which polymer melts can be created. The different options control how the monomers are selected for the sequence and determine the types of polymers that can be created.

Specific tags must be placed on the monomers, so that builders such as the *PolymerMonteCarloBuilder* can determine how different monomers are to be connected. To mark the atoms that form part of the backbone of the polymer chain, the tags *HEAD_CONNECT* and *TAIL_CONNECT* are used. In constructing the polymer chain a bond is created between the *TAIL_CONNECT* atom of one monomer and the *HEAD_CONNECT* atom of the next monomer. In order to make room for this bond one of the groups connected to each atom must be removed. These groups are specified by placing the tag *HEAD_GROUP_A* on the atom in the group that is bonded to the *HEAD_CONNECT* atom, with the same pattern being repeated for the monomer tail. In cases where the polymer chain has atoms with two different side groups, such as in vinyl polymers, removing a group to add the chain creates a stereochemical center. In this case specifying only one group for removal means that only one stereochemistry will be present in the polymer. To have different stereochemistries a second removal group can be specified using the *HEAD_GROUP_B* or *TAIL_GROUP_B* tags. The group that is removed to form the polymer will then be selected based on the `tactic_ratio` parameter.

The following image shows an example of how the tags can be set for the monomer used to make the polymer poly(methyl methacrylate). Note that as monomers are entered in a chemically saturated state, the double bond in methyl methacrylate has been replaced with a hydrogen and methyl group on the HEAD and TAIL ends of the monomer respectively. Removing these groups to form the polymer chain approximates chemical addition to the double bond. Also note that as the HEAD atom contains two identical hydrogens, only one removal group is specified. Conversely on the TAIL atom two different functional groups are present, and so two different methyl groups are tagged for possible removal to create different stereochemistries.



Specifying removal groups works similarly for condensation polymers, where the polymerization takes place by removing small molecules, such as water. Here the atoms in each monomer that form the leaving molecule can be specified by placing a *HEAD/TAIL_GROUP_A/B* tag on the atom that connects to the *HEAD/TAIL_CONNECT* backbone atom. For instance, in the above condensation polymer example where PET is formed, the hydroxyl oxygen is tagged as the HEAD/TAIL_GROUP_A on the terephthalic acid. Likewise an amine hydrogen is tagged as the HEAD/TAIL_GROUP_A on the ethylenediamine. When the HEAD_CONNECT and TAIL_CONNECT atoms on each monomer are connected the hydroxyl group from the terephthalic acid and the hydrogen from the ethylenediamine are removed, resulting in the two monomers forming an amide group. The removed atoms can form a water molecule.

The monomers used for the alpha and omega end groups follow a slightly different scheme. Here, as each monomer may only have one atom which is bonded to cap the polymer chain, the HEAD and TAIL atoms may be the same. If this is the case then the same monomer may be used either as a starting or ending group.

The above examples show how some common types of polymers may be created. Tacticity is controlled globally through the `tactic_ratio` parameter. The general proportion of each monomer in the resulting system is specified with the `monomer_ratios` parameter. This is useful for creating random copolymers with different ratios of monomers. It also controls the probability of selecting different monomers to start the polymer chain. The parameter `monomer_additions` controls the probability of adding one monomer onto another. Each row in this matrix corresponds to a monomer type and describes the probabilities of adding the next monomer type. This is useful for creating alternating copolymers and some types of block copolymers, especially diblock copolymers. This is also often necessary in creating condensation polymers such as PET or nylon, where two different monomers react to form the polymer chain. In general the probability of adding a monomer either outside or at the start of a block is the product of the monomer ratio and monomer addition probability for that monomer.

Monomer blocks within polymers can be created using the `monomer_block_sizes` parameter, which specifies how large blocks should be for each monomer. It is also possible to contaminate blocks through the `block_contamination` parameter. For each monomer in a block the probability of that monomer being different from the others is given by this parameter. Which monomer to contaminate the block with is controlled by the `monomer_ratios` and

monomer_additions parameters.

The orientation of monomers being added to the polymer chain can also be controlled with the parameter head_tail_flip_probability. By default, polymer chains are created by connecting the head group of the incoming monomer to the tail group of the previous one. The head_tail_flip_probability controls the probability that an incoming monomer is re-orientated so that the the tail group is connected to the tail group of the previous monomer, allowing the next monomer to connect to the head group of the existing monomer. Complete randomization of the monomer orientation is achieved by setting head_tail_flip_probability to 0.5. Setting this parameter to 0 maintains the same orientation for all monomers, while setting it to 1 maintains the same reversed orientation for all monomers.

The alpha_group and omega_group parameters are designed to allow specific end groups to be placed on polymer chains. These groups may be the result of how polymers are synthesized, or as the result of a post-synthetic functionalization step. The molecules used as end groups attach onto the polymer chain, and only need one connection point. This allows for the HEAD and TAIL tags to be placed on the same atoms.

Once the *PolymerSequence* object is created it internally creates a polymer sequence. This is returned with the sequence function. The generated polymer sequence is stored as a list of two dimensional integer numpy arrays. Each array in the list describes one chain in the polymer melt. In the numpy arrays the first dimension indexes each monomer position in the chain. The second dimension has two elements. The first element contains the integer index of the monomer at that position in the chain (including the alpha and omega end groups), which is ordered according to the input order of the monomers. The second element is a number indicating which groups need to be removed to form the polymer chain. Selecting either *HEAD_GROUP_A* or *HEAD_GROUP_B* adds either 0 or 2 respectively onto this number. Likewise selecting either *TAIL_GROUP_A* or *TAIL_GROUP_B* adds either 0 or 1 respectively to this number. In this way the combination of groups selected is stored in a two digit binary fashion.

PositionRestraint

class PositionRestraint(*atom_selection=None*, *spring_constant=None*, *reference_positions=None*)

Class to apply spring restraints to tether the atomic position to its original values.

Parameters

- **atom_selection** (*sequence of int | str | None*) – The selection of atoms that should be restrained. Can be given as list of indices or a tag. **Default:** All atoms.
- **spring_constant** (*PhysicalQuantity* of type energy / length ** 2) – The spring constant used for the tethering springs. **Default:** 0.5 * eV / Ang ** 2.
- **reference_positions** (*PhysicalQuantity* of type length) – If specified, the given positions will be used as reference positions. If set to None, the positions of the first configuration in the simulation will be used.

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

When studying crystallization at amorphous interfaces with *SteeredMolecularDynamics* the *PositionRestraint* can be used to prevent the crystal atoms from melting.

```
bulk_configuration.setCalculator(calculator)

# MolecularDynamics
diffraction_peak = DiffractionPeak(
    cutoff_radius=10.0*Ang,
    q_value=2.1/Ang,
    included_atoms=[Cobalt],
)

hook_function = SteeredMolecularDynamics(
    collective_variable=diffraction_peak,
    velocity=300/1000/ps,
    spring_constant=0.1*eV,
    measurement_interval=50,
)

method = Langevin(
    initial_velocity=MaxwellBoltzmannDistribution(
        temperature=1400.0*Kelvin,
        remove_center_of_mass_momentum=None
    ),
    reservoir_temperature=1400.0*Kelvin,
    friction=0.05*1/fs
)
constraints = [PositionRestraint(atom_selection='silicon', spring_constant=1.0*eV/
↪Ang**2)]
md_trajectory_1 = MolecularDynamics(
    configuration=bulk_configuration,
    constraints=constraints,
    trajectory_filename='restrained_steered_MD.hdf5',
    steps=2000000,
    post_step_hook=[
        hook_function,
    ],
    measurement_hook=hook_function.measurements,
    log_interval=1000,
    method=method
)
```

position_restraint.py

Notes

The *PositionRestraint* object can be used to tether the positions of selected atoms to their initial positions via harmonic springs. This allows the atoms to vibrate and include thermal effects, while they would not be able to leave their positions permanently, which avoids structural changes. Instead of the initial positions one can also specify different reference positions via the `reference_positions` parameters. A *PositionRestraint* object can be passed as constraint in *MolecularDynamics* or *OptimizeGeometry*.

PristineConfiguration

class PristineConfiguration(*configuration*, *reference_calculator*, *band_gap_calculator=None*, *phonon_calculator=None*, *symmetry_tolerance=None*, *supercell_repetitions=None*, *finite_size_correction_parameters=None*, *include_vibrations=None*)

Create a pristine reference for a defect.

Parameters

- **configuration** (*BulkConfiguration*) – The unit cell of the pristine material.
- **reference_calculator** (`All calculators`) – The calculator used for the formation energy. This calculator must be suitable for calculating both pristine configuration and defect energies.
- **band_gap_calculator** (`All calculators` | None) – The calculator used for the band gap. If not given the reference calculator is used for the band gap.
- **phonon_calculator** (`All calculators` | None) – The calculator used for the vibrational energy. If not given, the reference calculator is used to calculate the phonon density of states. As with the reference calculator, this calculator is used on both pristine and defect configurations.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance used for calculating the symmetry. **Default:** 0.1 Angstrom
- **supercell_repetitions** (*Tuple of integers of length 3*) – The supercell repetitions used to create the defect. **Default:** (1, 1, 1)
- **finite_size_correction_parameters** (*IsotropicFiniteSizeCorrectionParameters* | *InterfaceFiniteSizeCorrectionParameters* | Disabled | None) – Parameters for the finite size correction applied to the defect. If Disabled, finite size corrections are ignored. **Default:** IsotropicFiniteSizeCorrectionParameters
- **include_vibrations** (*bool*) – Whether vibrational terms are included in calculating the formation energy. The configuration will be optimized with the phonon calculator before calculating vibrations. **Default:** False

addChargeCorrection()

Returns
Whether calculations are done for correcting charged states.

Return type
bool

bandGapCalculator()

Returns
The calculator used for the band gap. None if no specific calculator provided.

Return type
`All calculators` | None

chargeCorrectionString()

Returns
A string describing the charge correction.

Return type
str

conductionBandMinimum(*band_gap*)

Get the conduction band minimum using either the formation energy or band gap calculator.

Parameters
band_gap (*bool*) – Whether the value from the band gap calculator is returned. If no band gap calculator was given then None is returned.

Returns
The calculated conduction band minimum.

Return type
PhysicalQuantity of type energy

configuration()

Returns
The unit cell that defines the pristine material.

Return type
BulkConfiguration

electrostaticPotential()

Returns
The pristine electrostatic difference potential.

Return type
ElectrostaticDifferencePotential

fermiLevel(*band_gap*)

Get the conduction band minimum using either the formation energy or band gap calculator.

Parameters
band_gap (*bool*) – Whether the value from the band gap calculator is returned. If no band gap calculator was given then None is returned.

Returns
The calculated Fermi level.

Return type
PhysicalQuantity of type energy

finiteSizeCorrectionParameters()

Returns
The parameters for the finite size correction.

Return type
float

includeVibrations()

Returns
Whether vibrational corrections are calculated.

Return type
bool

phononCalculator()

Returns
The calculator used for the vibrational energy. None if no specific calculator provided.

Return type
`All calculators` | None

phononDensityOfStates()

Returns
The calculated phonon density of states if vibrational corrections are enabled. None if not.

Return type
PhononDensityOfStates | None

pristineSymmetry()

Returns
The symmetry information of the pristine unit cell.

Return type
dict

referenceCalculator()

Returns
The original given reference calculator.

Return type
`All calculators`

stress()

Returns
The calculated stress in the reference material.

Return type
Stress

superCell()

Returns
The super cell of the pristine material using the given repetitions.

Return type
BulkConfiguration

supercellRepetitions()

Returns
The supercell repetitions used to create the defect.

Return type
tuple

supercellSymmetry()

Returns
The symmetry information of the supercell.

Return type
dict

symmetryTolerance()

Returns
The tolerance used for detecting the symmetry.

Return type
PhysicalQuantity of type length

totalEnergy()

Returns
The reference energy calculated with the formation energy calculator.

Return type
TotalEnergy

uniqueString()

Return a unique string representing the state of the object.

unitCell()

Returns
The unit cell that defines the pristine material.

Return type
BulkConfiguration

update()

Perform the necessary calculations

valenceBandMaximum(*band_gap*)

Get the valence band maximum using either the formation energy or band gap calculator.

Parameters
band_gap (*bool*) – Whether the value from the band gap calculator is returned. If no band gap calculator was given then None is returned.

Returns
The calculated valence band maximum.

Return type
PhysicalQuantity of type energy

vibrations()

Returns
The vibrations if vibrational corrections are enabled. None if not.

Return type
PhysicalQuantity of type inverse time | None

Notes

The *PristineConfiguration* object contains the information about the original host material used in *ChargedPointDefectConfiguration* calculations. Where applicable, the information in the *PristineConfiguration* object is used in also calculating the defect, ensuring consistent settings between a range of defect calculations.

The *PristineConfiguration* is created by supplying a host configuration and a reference energy calculator. This calculator is also used as the reference calculator in the *ChargedPointDefectConfiguration* calculation. The defect supercell size is given with the `supercell_repetitions` argument. The correction for the defect charge can be specified using the `finite_size_correction_parameters` argument. If no correction is desired, the keyword `Disabled` should be given.

Some calculators can also be specified for specific parts of the calculation. This is useful in cases where the appropriate reference energy calculator is unsuitable. A band gap calculator can be given using the `band_gap_calculator` argument. This calculation is performed with the pristine unit cell, and thus this calculator can be reasonably more computationally expensive than the reference calculator. Additionally, a calculator for the vibrational correction can be given by using the `phonon_calculator` argument and setting the `include_vibrations` argument to `True`. If vibrations are included here they should also be included in the calculation of atomic chemical potentials.

ProcessesPerNode

class ProcessesPerNode

Flag to specify the number of processes running on a single node.

ProjectedDensityOfStates

class ProjectedDensityOfStates(*configuration*, *kpoints=None*, *projections=None*, *energies=None*, *energy_zero_parameter=None*, *bands_above_fermi_level=None*, *spectrum_method=None*)

Class for calculating the projected density of states for a configuration.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with an attached calculator for which to calculate the projected density of states.
- **kpoints** – The k-points for which to calculate the projected density of states. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **projections** (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for the calculating the weights. **Default:** `[Projection(spin=Spin.Up), Projection(spin=Spin.Down)]`.
- **energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the projected density of states. The energies are relative to the zero of energy specified in `energy_zero_parameter`. **Default:** `numpy.arange(-2.0, 2.0, 0.01) * eV`
- **energy_zero_parameter** (`FermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `FermiLevel`
- **bands_above_fermi_level** (int | `All`) – The number of bands above the Fermi level per principal spin channel. Must be a non-negative integer. **Default:** `All` (All bands are included).
- **spectrum_method** (`GaussianBroadening` | `TetrahedronMethod` | `Automatic`) – The method to use for calculating the projected density of states. If `Automatic` is set then the tetrahedron method is used if there are more than 10 k-points in any direction, else the Gaussian broadening method is used. **Default:** `TetrahedronMethod` for BulkConfigurations and `GaussianBroadening` for MoleculeConfigurations.

bandsAboveFermiLevel()

Returns
The number of bands above the Fermi level per principal spin channel.

Return type
int

densityOfStates(*normalization=None*)
Returns the possible normalized density of states.

Parameters
normalization (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** False

Returns
The full density of states as a vector of the length of the number of energies.

Return type
PhysicalQuantity of type reciprocal energy

directBandGap()

Returns
The direct band gap.

Return type
PhysicalQuantity of type energy

energies()

Returns
The energies.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy.

evaluate(*projection_index=None*, *normalization=None*)
The projected density of states for a given projection.

Parameters

- **projection_index** (*int*) – The index of the projection to query the projected density of states for. Negative indexing can be used such that e.g. projection_index=-1 will return the projected density of states for the last projection. **Default:** The projected density of states for each projection is returned.
- **normalization** (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** False

Returns
The projected density of states as a vector of the length of the number of energies, e. If projection_index is not specified, an array of shape (p, e) is returned, where p is the number of projections.

Return type
PhysicalQuantity of type reciprocal energy

fermiLevel(*spin=None*)

Parameters
spin (Spin.Up | Spin.Down | Spin.All) – The spin the Fermi level should be returned for. Must be either Spin.Up, Spin.Down, or Spin.All. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** Spin.Up

Returns
The Fermi level in absolute energy.

Return type
PhysicalQuantity of type energy

indirectBandGap()

Returns
The indirect band gap.

Return type
PhysicalQuantity of type energy

kpoints()

Returns
The k-point sampling used for calculating the projected density of states.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the PDOS.

Return type
dict

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

projections()

Returns
The projections used for the calculated projected density of states.

Return type
list of class:~.*Projection* | class:~.*Projection*

setMetatext(*metatext*)
Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Calculate the Projected Density of States for bulk gallium arsenide, by projecting on each shell per each element

```
lattice = FaceCenteredCubic(5.6537*Angstrom)
elements = [Gallium, Arsenic]
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(7, 7, 7)

numerical_accuracy_parameters = NumericalAccuracyParameters(
            k_point_sampling=k_point_sampling,
                )

calculator = LCAOCalculator(
            numerical_accuracy_parameters=numerical_accuracy_parameters,
                )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# ProjectedDensityOfStates
# ------------------------------------------------------------
pdos = ProjectedDensityOfStates(
    configuration=bulk_configuration,
    projections=ProjectOnShellsByElement,
    kpoints=MonkhorstPackGrid(11, 11, 11),
    energies=numpy.arange(-4.0, 4.0, 0.01) * eV
    )
```

`gaas_pdos.py`

The same resuls can be obtained by defining explicitely the list of projections:

```
# ------------------------------------------------------------
# ProjectedDensityOfStates
# ------------------------------------------------------------
projections = [Projection(l_quantum_numbers=[0], atoms=[Gallium]),
               Projection(l_quantum_numbers=[1], atoms=[Gallium]),
               Projection(l_quantum_numbers=[2], atoms=[Gallium]),
               Projection(l_quantum_numbers=[0], atoms=[Arsenic]),
               Projection(l_quantum_numbers=[1], atoms=[Arsenic]),
               Projection(l_quantum_numbers=[2], atoms=[Arsenic]),]

pdos = ProjectedDensityOfStates(
    configuration=bulk_configuration,
    projections=projections,
    kpoints=MonkhorstPackGrid(11, 11, 11),
    energies=numpy.arange(-4.0, 4.0, 0.01) * eV)
```

More flexible projections can be achieved with algebraic operations, refer to the the documentation of *Projection*.

Notes

The *ProjectedDensityOfStates* is used to visualize the contribution of different orbitals to the density of states.

Recalling that the density of states can be written as

$$D(\epsilon) = \sum_n \delta\left(\epsilon - \epsilon_n\right)$$

where n includes all the quantum numbers of the system, we can define the Projected Density of States associated to a given projection M as

$$D_M(\epsilon) = \sum_n \delta\left(\epsilon - \epsilon_n\right) \langle\psi_n|\hat{\mathbf{P}}_M|\psi_n\rangle$$

ψ_n are the eigenstates and $\hat{\mathbf{P}}_M$ is a projection operator defined according to the description contained in the documentation of *Projection*.

Note: The *ProjectedDensityOfStates* will always be positive (in units of eV^{-1}) for projections on up/down spin, but might be both positive and negative (in units of $\frac{\hbar}{2}\mathrm{eV}^{-1}$) for projections involving the Pauli spin matrices. See the *Usage Example* in *Projection* for some more information.

ProjectedLocalDensityOfStates

class ProjectedLocalDensityOfStates(*configuration*, *method=None*, *energies=None*, *kpoints=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *density_mesh_cutoff=None*, *processes_per_energy=None*)

Class for representing the projected local density of states.

Parameters

•**configuration** (*DeviceConfiguration*.) – The configuration to calculate the projected local DOS of.

•**method** (`LocalDeviceDensityOfStates` | `DeviceDensityOfStates`) – The method used for calculating the projected local DOS. Either the class of *LocalDeviceDensityOfStates* or *DeviceDensityOfStates* should be supplied, not instances of those classes. **Default:** `LocalDeviceDensityOfStates`

•**energies** (*PhysicalQuantity* of type energy) – The energies for which the density of states should be calculated. **Default:** `numpy.linspace(-2.0, 2.0, 200) * eV`

•**kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which the projected local DOS should be calculated. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.

•**contributions** (`Left` | `Right` | `All`) – The density contributions to include in the projected local DOS. **Default:** `All`

•**self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to use. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`

•**energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`

•**infinitesimal** (*PhysicalQuantity* of type energy) – Small energy, used to move the calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6 * eV`

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** The density mesh cutoff set on the calculator.

•**processes_per_energy** (int | `All` | `Automatic`) – The number of processes to use for each energy point. If the number of processes is larger than the number of k-points, it is highly recommended to set this to a value between the number of k-points and the total number of processes (`All`). This option is only used if `method` is `LocalDeviceDensityOfStates`. **Default:** All processes calculate each energy point collaboratively.

contributions()

Returns
The contributions.

Return type
`Left` | `Right` | `All`

electrodeFermiLevels()

Returns
The Fermi levels of the left and right electrodes in absolute energies.

Return type
PhysicalQuantity of type energy.

`electrodeVoltages()`

Returns
The electrode voltages.

Return type
PhysicalQuantity of type electrical potential

`energies()`

Returns
The energies.

Return type
PhysicalQuantity of type energy

`energyZero()`

Returns
The energy zero used for the energy scale in this transmission spectrum.

Return type
PhysicalQuantity of type energy.

`evaluate(`*spin=None*`)`

Return the projected local DOS for the specified spin.

Parameters
spin (`Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown`) – The spin to select. **Default:** `Spin.Sum`

`infinitesimal()`

Returns
The infinitesimal.

Return type
PhysicalQuantity of type energy

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`method()`

Returns
The method used for calculating the projected local DOS.

Return type
`LocalDeviceDensityOfStates | DeviceDensityOfStates`

`nlprint(`*stream=None*`)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

zSlicing()

Returns
The z-slicing of the cell.

Return type
PhysicalQuantity of type length

Usage Examples

Notes

ProjectedPhononDensityOfStates

class ProjectedPhononDensityOfStates(*dynamical_matrix*, *qpoints=None*, *projections=None*, *energies=None*, *number_of_bands=None*, *spectrum_method=None*)

Class for calculating the projected density of states for a configuration.

Parameters

•**dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to calculate the phonon density of states from.

•**qpoints** – The q-points for which to calculate the projected density of states. **Default:** KpointDensity(density_a=12*Ang, density_b=12*Ang, density_c=12*Ang).

•**projections** (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for the calculating the weights. **Default:** `CartesianProjectionOnDirections`.

•**energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the projected density of states. The energies are relative to the zero of energy specified in `energy_zero_parameter`. **Default:** `numpy.arange(0.0, 200.0, 0.1) * meV`

•**number_of_bands** (*int*) – The number of bands. **Default:** All bands

•**spectrum_method** (`GaussianBroadening` | `TetrahedronMethod` | `Automatic`) – The method to use for calculating the projected density of states. If `Automatic` is set then the tetrahedron method is used if there are more than 10 q-points in any direction, else the Gaussian broadening method is used. **Default:** `TetrahedronMethod` for BulkConfigurations and `GaussianBroadening` for MoleculeConfigurations.

densityOfStates(*normalization=None*)

Returns the possible normalized density of states.

Parameters
normalization (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** `False`

Returns
The full density of states as a vector of the length of the number of energies.

Return type
PhysicalQuantity of type reciprocal energy

`energies()`

Returns
The energies.

Return type
PhysicalQuantity of type energy

`entropy(`*temperature=PhysicalQuantity(300.0, K)*`)`

Calculate the entropy of the phonons.

The entropy is evaluated using the quasi-harmonic approximation (see Eq.(5.5) in Dove, Martin T. (1993), Introduction to lattice dynamics, Cambridge university press).

Parameters
`temperature` (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the entropy at. **Default:** `300 * Kelvin`

Returns
The entropy.

Return type
PhysicalQuantity with the unit `meV / Kelvin`

`evaluate(`*projection_index=None*, *normalization=None*`)`

The projected density of states for a given projection.

Parameters

- `projection_index` (*int*) – The index of the projection to query the projected density of states for. Negative indexing can be used such that e.g. projection_index=-1 will return the projected density of states for the last projection. **Default:** The projected density of states for each projection is returned.
- `normalization` (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** `False`

Returns
The projected density of states as a vector of the length of the number of energies, *e*. If `projection_index` is not specified, an array of shape (*p*, *e*) is returned, where *p* is the number of projections.

Return type
PhysicalQuantity of type reciprocal energy

`freeEnergy(`*temperature=PhysicalQuantity(300.0, K)*`)`

Calculate the Helmholtz free energy of the molecule / lattice.

Parameters
`temperature` (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the entropy at. **Default:** `300 * Kelvin`

Returns
The free energy of the molecule / lattice (without internal energy).

Return type
PhysicalQuantity with the unit `eV`

`internalEnergy(`*temperature=PhysicalQuantity(300.0, K)*`)`

Calculate the internal harmonic energy of the molecule / lattice at the given temperature.

Parameters
temperature (*PhysicalQuantity* of type temperature.) – The temperature to evaluate the internal energy at. **Default:** `300 * Kelvin`

Returns
The internal energy of the molecule / lattice.

Return type
PhysicalQuantity with the unit `eV`

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the PDOS.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

numberOfBands()

Returns
The number of bands to include.

Return type
int | All

projections()

Returns
The projections used for the calculated projected density of states.

Return type
list of class:~.*Projection* | class:~.*Projection*

qpoints()

Returns
The q-point sampling used for calculating the projected density of states.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString`()

Return a unique string representing the state of the object.

`zeroPointEnergy`()

Calculate the zero point energy in the quasi-harmonic approximation.

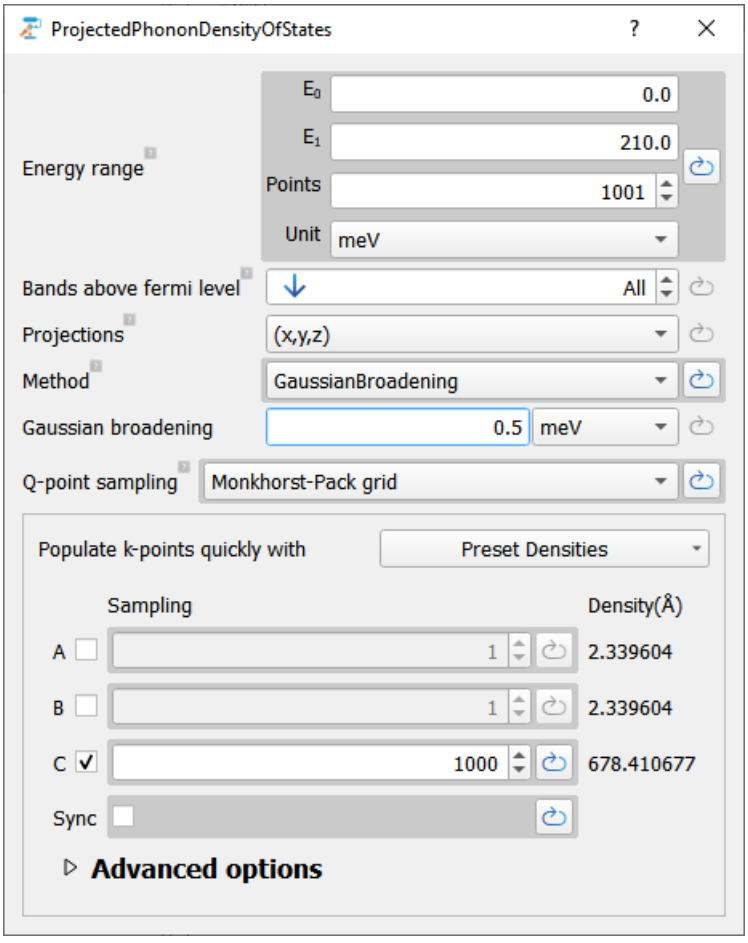
Returns
The zero point energy of the molecule / lattice

Return type
PhysicalQuantity with the unit `eV`

Usage Example

Calculate the projected phonon density of states (PPDOS) of a carbon nanotube (CNT) using the *TremoloXCalculator* calculator with a Tersoff potential. In this example, the PPDOS is projected on x,y,z movement of the atoms.

The figure below shows how to setup the calculation using the work flow builder and the Projected Phonon Density of States editor.



The available projections are

•(x,y,z) : Project on (x, y, z) motion of all the atoms. (Default projection type)

•Sites : Project on each atom, sum over x,y,z.

•Tags : Project on each tag, sum over x,y,z.

•Elements : Project on each element, sum over x,y,z.

•Sites and (x,y,z) : Project on each atom and x,y,z. This will generate 3*N projection, where N is the number of atoms.

•Tags and (x,y,z) : Project on each tag and x,y,z.

•Elements and (x,y,z) : Project on each tag and x,y,z.

```
# %% (6,0) Carbon Nanotube

# Set up lattice
vector_a = [14.700167046094046, 0.0, 0.0]*Angstrom
vector_b = [0.0, 14.700167046094046, 0.0]*Angstrom
vector_c = [0.0, 0.0, 4.26258]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,
            Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,
            Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon]

# Define coordinates
fractional_coordinates = [[ 0.659867810732,  0.5           ,  0.666666666667],
                          [ 0.638449585341,  0.579933905366,  0.833333333333],
                          [ 0.659867810732,  0.5           ,  0.333333333333],
                          [ 0.579933905366,  0.638449585341,  0.666666666667],
                          [ 0.5           ,  0.659867810732,  0.833333333333],
                          [ 0.638449585341,  0.579933905366,  0.166666666667],
                          [ 0.579933905366,  0.638449585341,  0.333333333333],
                          [ 0.420066094634,  0.638449585341,  0.666666666667],
                          [ 0.361550414659,  0.579933905366,  0.833333333333],
                          [ 0.5           ,  0.659867810732,  0.166666666667],
                          [ 0.420066094634,  0.638449585341,  0.333333333333],
                          [ 0.340132189268,  0.5           ,  0.666666666667],
                          [ 0.361550414659,  0.420066094634,  0.833333333333],
                          [ 0.361550414659,  0.579933905366,  0.166666666667],
                          [ 0.340132189268,  0.5           ,  0.333333333333],
                          [ 0.420066094634,  0.361550414659,  0.666666666667],
                          [ 0.5           ,  0.340132189268,  0.833333333333],
                          [ 0.361550414659,  0.420066094634,  0.166666666667],
                          [ 0.420066094634,  0.361550414659,  0.333333333333],
                          [ 0.579933905366,  0.361550414659,  0.666666666667],
                          [ 0.638449585341,  0.420066094634,  0.833333333333],
                          [ 0.5           ,  0.340132189268,  0.166666666667],
                          [ 0.579933905366,  0.361550414659,  0.333333333333],
                          [ 0.638449585341,  0.420066094634,  0.166666666667]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

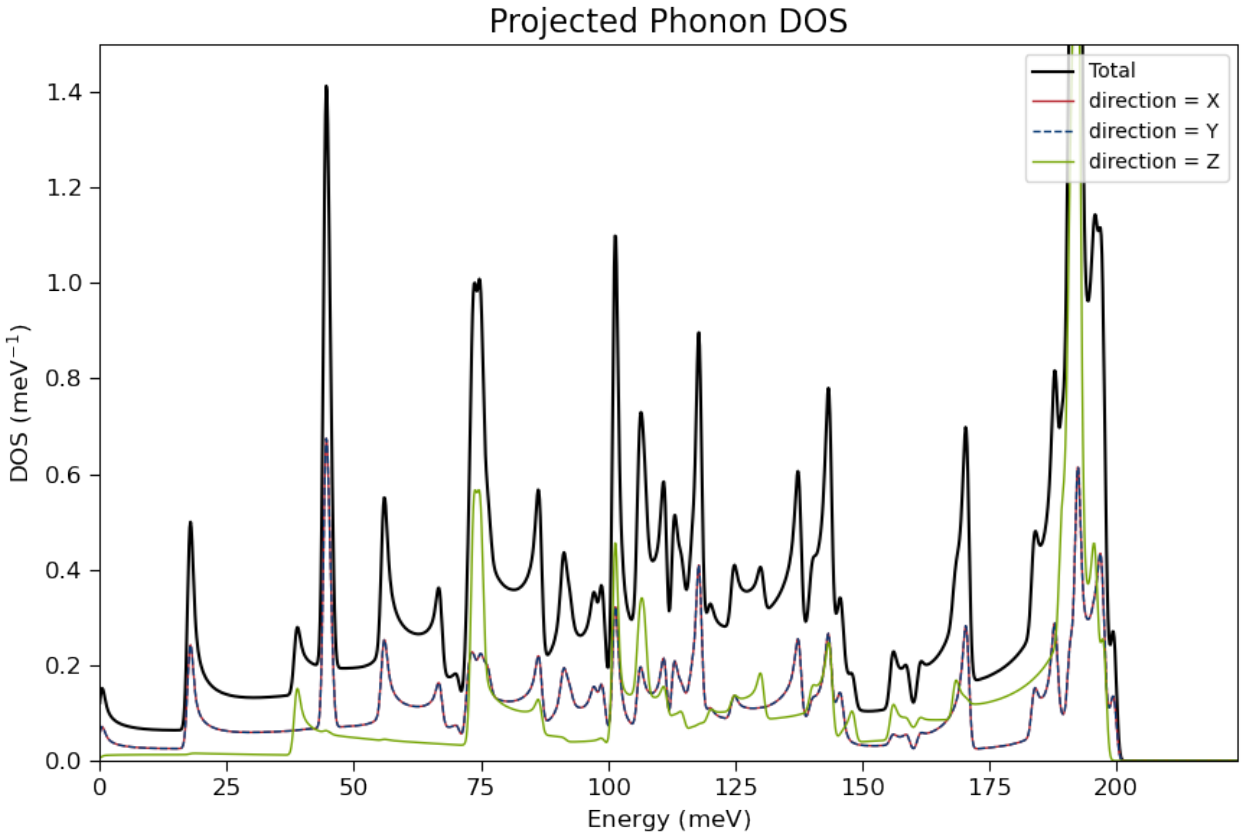
(continued from previous page)

```
# %% Set ForceFieldCalculator

# %% ForceFieldCalculator
potentialSet = Tersoff_C_2010()
calculator = TremoloXCalculator(parameters=potentialSet)

# %% Set Calculator
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# %% OptimizeGeometry
optimized_configuration = OptimizeGeometry(
    configuration=bulk_configuration,
    max_forces=0.01*eV/Angstrom,
    constraints=[
        BravaisLatticeConstraint()
    ],
    trajectory_filename='CNT_results.hdf5'
)

nlsave('CNT_results.hdf5', optimized_configuration)


# %% DynamicalMatrix
dynamical_matrix = DynamicalMatrix(
    configuration=optimized_configuration,
    filename='CNT_results.hdf5',
    object_id='dm',
    calculator=calculator,
    repetitions=(1, 1, 5)
)
dynamical_matrix.update()


# %% ProjectedPhononDensityOfStates
qpoints = MonkhorstPackGrid(
    na=1,
    nb=1,
    nc=1000
)

projectedphonondensityofstates = ProjectedPhononDensityOfStates(
    dynamical_matrix=dynamical_matrix,
    qpoints=qpoints,
    energies=numpy.linspace(0.0, 210.0, 1001)*meV,
    spectrum_method=GaussianBroadening(
        broadening=0.5*meV
    )
)
nlsave('CNT_results.hdf5', projectedphonondensityofstates)
```

cnt_projected_phonon_dos.py

Analyze results

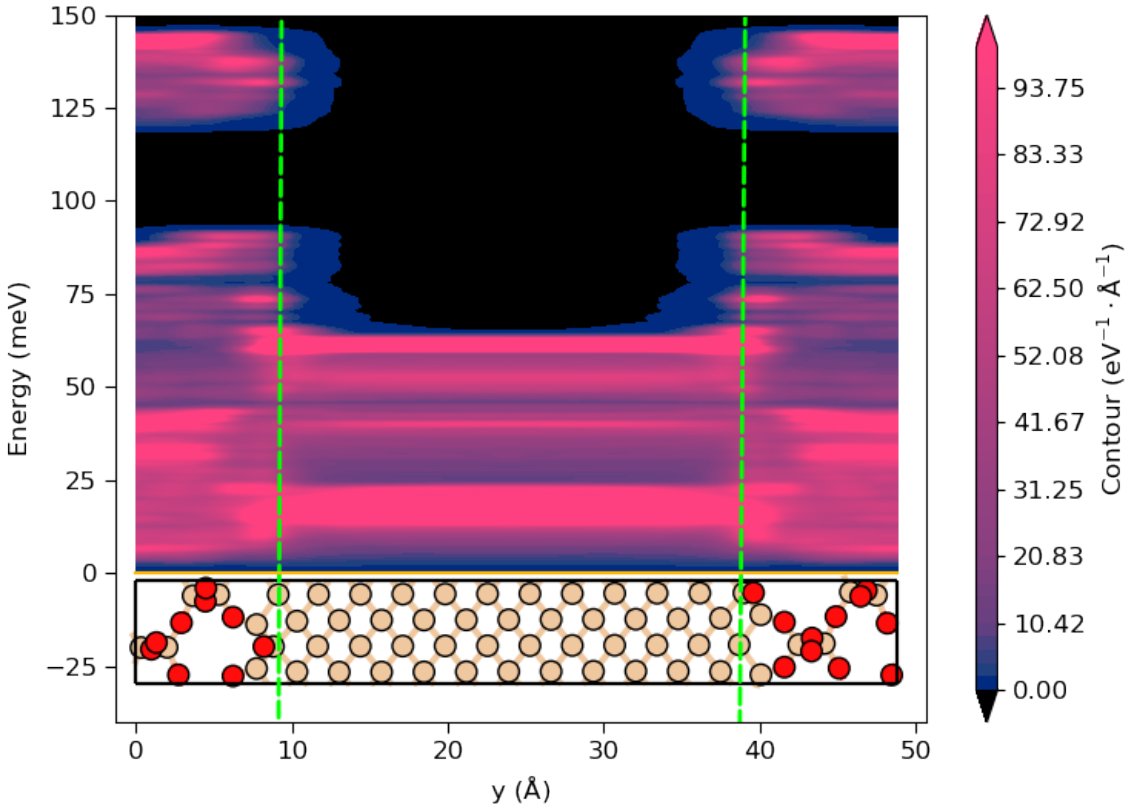
When the calculation has finished the results can be visualized with the **Projected DOS analyzer** as illustrated below.



In this case, the projections on x- and y directions are equal due to the rotational symmetry of the CNT. The z-direction movement is, on the other hand significantly different.

Local Phonon Density of States

If the PPDOS has been calculated with site projections (`Sites` or `Sites and (x,y,z)`), it is possible to plot the phonon density of states as function of spatial coordinates, i.e. the local phonon density of states. This is illustrated in the image below, where the PPDOS is calculated for a SiO2-Si-SiO2 structure as shown in the plot.



It is evident that some of the high-energy phonon modes in the SiO2 decays into the central Si region. Such phonon tunneling effects may have important consequences for the phonon-limited mobility in the Si region.

Projection

class Projection(*spin=None*, *atoms=None*, *l_quantum_numbers=None*, *m_quantum_numbers=None*)

Initialize the class by defining the orbitals to include in the projection.

Parameters

- **spin** (None | `Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z`) – Spin components to include in the projection. **Default:** `Spin.Sum`
- **atoms** (None | list of *PeriodicTableElement* | list of str | list of int | `All`) – Atoms to include in the projection, defined either by element type, by tag, or by the index of individual atoms. **Default:** `All`
- **l_quantum_numbers** (None | list of int | `All`) – Shells for the selected atoms to include in the projection. **Default:** `All`
- **m_quantum_numbers** (None | list of int | `All`) – Orbitals for the selected shells to include in the projection. **Default:** `All`

atoms()

Returns
The atoms included in the projection.

Return type
list of *PeriodicTableElement* | list of str | list of int | `All`

lQuantumNumbers()

Returns
The l-quantum numbers included in the projection.

Return type
list of int | All

label()

Get the label for this projection.

Returns
The label used for this projection.

Return type
str

mQuantumNumbers()

Returns
The m-quantum numbers included in the projection.

Return type
list of int | All

setLabel(*label*)

Set the label for this projection.

Parameters
label (*str*) – The new label.

spin()

Returns
The spin components included in the projection.

Return type
Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z

uniqueString()

Return a unique string representing the state of the object.

Usage Example

The user can declare projection instances and also combine different projections by using the algebraic operations sum, product and difference. The sum will perform the union between the orbital sets, the product the intersection and the difference the disjuntive union. For example

```
# Define a projection on all orbitals
p1 = Projection()

# Define a projection on s, p orbitals of Oxygen
p2 = Projection(atoms=[Oxygen], l_quantum_numbers=[0, 1])

# Define a projection on Spin Up
p3 = Projection(spin=Spin.Up)

# Define a projection on all orbitals except s and p orbitals of Oxygen
```

(continues on next page)

(continued from previous page)

```
p4 = p1 - p2

# We can define a projection on s and p orbital of Oxygen for Spin Up
# in two equivalent ways.
p5 = p2 * p3

# or
p5 = Projection(atoms=[Oxygen], l_quantum_numbers=[0, 1], spin=Spin.Up)
```

Note: Projections on the x, y, z Pauli spin matrices with `Spin.X`, `Spin.Y`, `Spin.Z` work differently from other projections; instead of defining a projection subspace based on sets of orbitals, they are operators returning the observable corresponding to spin along each of the three coordinate axes in Euclidean space.

As such, algebraic combinations of projections are undefined when projecting on the Pauli spin matrices. This means that the algebraic combination of two projections is not allowed if either of the two is projecting on one of the Pauli matrices for a particular set of orbitals and the other isnt projecting on the same Pauli matrix for the same set of orbitals.

For example, the following combinations **are** allowed:

```
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Y, atoms=[Oxygen])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Z, atoms=[Oxygen])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Up, atoms=[Oxygen])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Down, atoms=[Oxygen])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Sum, atoms=[Oxygen])
Projection(spin=Spin.Up, atoms=[Silicon]) + Projection(spin=Spin.Down, atoms=[Silicon])
```

These combinations, however, **are not** allowed:

```
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Y, atoms=[Silicon])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Z, atoms=[Silicon])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Up, atoms=[Silicon])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Down, atoms=[Silicon])
Projection(spin=Spin.X, atoms=[Silicon]) + Projection(spin=Spin.Sum, atoms=[Silicon])
```

(same for the + and * operators).

Abbreviations

ATK supports a number of keyword which automatically generate a list of projections onto commonly used sets of orbitals. The available keywords are

ProjectOnUpDownSpin	Projections on spin up and spin down
ProjectOnXYZSpin	Projections on the x, y, z Pauli spin matrices
ProjectOnSites	Projections on each available atomic site
ProjectOnTags	Projections on each available tag for atomic sites present in the configuration
ProjectOnElements	Projections on each available element
ProjectOnShellsByElement	Projections on each available shell per each available element
ProjectOnOrbitalsByElement	Projections on each available orbital per each available element

For example, for bulk Gallium Arsenide the keyword `ProjectOnElements` is equivalent to

```
projections = [Projection(atoms=[Gallium]), Projection(atoms=[Arsenic])]
```

Notes

The *Projection* instances do not contain the list of projection orbitals immediately after declaration. The available information is translated to a set of orbital indexes for a specific configuration only inside analysis objects. The orbital indexes are then used to construct a projection operator, according to the procedure illustrated below.

Definition of the Projection operator

The definition of a projection operator is trivial in an orthogonal basis, where we can define

$$\hat{\mathbf{P}}_M = \sum_{m \in M} |m\rangle\langle m|$$

This operator is idempotent and hermitian, and the identity operator can be easily expressed as the sum of a projection onto a subspace and the projection onto the complementary subspace:

$$\hat{\mathbf{I}} = \hat{\mathbf{P}}_M + \hat{\mathbf{P}}_{\bar{M}}$$

For a non-orthogonal basis this definition is not valid anymore and we have to formulate the projection taking into account the overlap between basis functions. In this case it is not possible in general to define a projection operator which is both hermitian and idempotent, and several different definitions are possible. A detailed treatment of this issue and the derivation of some of the formulas used here can be found in[1].

For a non-orthogonal basis the identity operator is written as

$$\hat{\mathbf{I}} = \sum_{i,j} |i\rangle S^{-1}_{ij} \langle j|$$

where S^{-1} is the inverse of the overlap matrix, defined as usual by the inner products between basis functions $S_{ij} = \langle i|j\rangle$. We may then write the projection on a subspace M as

$$\hat{\mathbf{P}}_M = \sum_{m \in M, i} |m\rangle S^{-1}_{mi} \langle i|$$

This projection fulfills the condition $\hat{\mathbf{I}} = \hat{\mathbf{P}}_M + \hat{\mathbf{P}}_{\bar{M}}$. However this operator is not hermitian and we may as well use its adjoint

$$\hat{\mathbf{P}}^{\dagger}_M = \sum_{m \in M, i} |i\rangle S^{-1}_{mi} \langle m|$$

[1] M.ăSoriano and J.ăJ. Palacios. Theory of projections with nonorthogonal basis sets: Partitioning techniques and effective Hamiltonians. *Phys. Rev. B*, 90:075128, Aug 2014. doi:10.1103/PhysRevB.90.075128.

Also this operator fulfills the condition $\hat{\mathbf{I}} = \hat{\mathbf{P}}_M^\dagger + \hat{\mathbf{P}}_{\bar{M}^\dagger}$. The identity operator can be decomposed in additional different ways and the choice of the projection is not unique[Page 3679, 1]. The advantage of those two definitions is that the projection subspace and its complement are orthogonal. This implies that the sum of the weights associated to a projection and its complement is one by construction. We can as well choose to use a linear combination of the projection operator and its adjoint and define a new projector as $\frac{1}{2}\left(\hat{\mathbf{P}}_M^\dagger + \hat{\mathbf{P}}_M\right)$. The matrix representation of such linear combination is given by

$$P_M = \frac{1}{2}\left(\tilde{P}_M S + S\tilde{P}_M\right)$$

$\tilde{P}_M$ is a diagonal matrix such that $P_m = 1$ if $m \in M$ and $P_m = 0$ otherwise. This is the matrix representation of the projection implemented in QuantumATK. The same expression can also be derived from the taylor expansion to the first order in S of the projection operator in the orthogonal representation, after an inverse Löwdin transform[2].

ProjectionGenerator

class ProjectionGenerator(*spin=None*, *atoms=None*, *l_quantum_numbers=None*, *m_quantum_numbers=None*)

Initialize the class by defining which components to iterate over when generating the list of projections.

Parameters

- **spin** (None | UpDownProjection | XYZProjection | False) – Determine whether to iterate over spin components when generating the list of projections. This can be done either on the up/down components or on the x/y/z Pauli spin matrices. If False, the spin projection will be Spin.Sum. **Default:** False
- **atoms** (None | SitesProjection | TagsProjection | ElementsProjection | False) – Determine whether to iterate over atoms when generating the list of projections. This can be done either by individual sites, tags present in the configuration, or element types. **Default:** False
- **l_quantum_numbers** (*None | bool*) – Determine whether to iterate over shells when generating the list of projections. **Default:** False
- **m_quantum_numbers** (*None | bool*) – Determine whether to iterate over orbitals when generating the list of projections. **Default:** False

generate(*configuration*)

Generate and return the list of projections for a given configuration.

Parameters
configuration (*BulkConfiguration* | *MoleculeConfiguration*) – The bulk configuration with an attached calculator defining the basis for which to generate the list of projections.

Returns
The list of projections.

Return type
list of *Projection*

uniqueString()

Return a unique string representing the state of the object.

[2] G.ăPenazzi, A.ăPecchia, V.ăGupta, and T.ăFrauenheim. A Self Energy Model of Dephasing in Molecular Junctions. *The Journal of Physical Chemistry C*, 120(30):16383–16392, 2016. doi:10.1021/acs.jpcc.6b04185.

ProjectionList

class ProjectionList(*atoms=None*, *elements=None*, *angular_momenta=None*)

Class for representing a list of orbitals. Results of many analysis objects, e.g. MolecularEnergySpectrum, can be projected onto this limited set of orbitals.

Parameters

- **atoms** (list of int | `All`) – List of integers specifying the atom indices to include in the projection list. **Default:** `All`
- **elements** (list of *PeriodicTableElement* | `All`) – List of elements to include in the projection list **Default:** `All`
- **angular_momenta** (list of int | `All`) – List of angular momenta, specified as integers (0=s, 1=p,), to include in the projection list **Default:** `All`

angularMomenta()

Returns
The angular momenta which are included in the projection list.

Return type
list of int

atoms()

Returns
The atom indices which are included in the projection list.

Return type
list of int

elements()

Returns
The elements which are included in the projection list.

Return type
list of *PeriodicTableElement*

includeAll()

Returns
`True` if we include all orbitals as targets.

Return type
bool

orbitalIndex(*configuration*)

Generate a list with indices of the orbitals in the projection list

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to extract orbital index from

Returns
A list with orbital indices

Return type
list of int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the Nitrogen s level within an ammonia molecule:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates= [[ 0.      ,  0.      ,  0.124001],
                            [ 0.      ,  0.941173, -0.289336],
                            [ 0.81508 , -0.470587, -0.289336],
                            [-0.81508 , -0.470587, -0.289336]]*Angstrom )

# Define the calculator
calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Calculate and save the molecular energy spectrum
molecular_energy_spectrum = MolecularEnergySpectrum(
    configuration=molecule_configuration,
    energy_zero_parameter=AbsoluteEnergy,
    projection_list=ProjectionList(elements=[Nitrogen], angular_momenta = [0])
    )
nlsave('molecular.hdf5', molecular_energy_spectrum)

# Extract the energy
energy = molecular_energy_spectrum.evaluate()
occ = molecular_energy_spectrum.occupation()

# ... and print it out
print("level   energy(eV)    occupation")
for i in range(len(energy[0])):
    print('  %d  %12.4f  %12.4f  ' % (i, energy[0][i].inUnitsOf(eV),occ[0][i]))
```

nh3.py

PseudoPotentialProjectorShift

class PseudoPotentialProjectorShift(*s_orbital_shift=None*, *p_orbital_shift=None*, *d_orbital_shift=None*, *f_orbital_shift=None*, *g_orbital_shift=None*)

Class for representing projector shift energies for each angular momentum.

Parameters

- **s_orbital_shift** (*PhysicalQuantity* of type energy.) – The projector shift of the s orbitals. **Default:** `0.0 * eV`.
- **p_orbital_shift** (*PhysicalQuantity* of type energy.) – The projector shift of the p orbitals. **Default:** `0.0 * eV`.
- **d_orbital_shift** (*PhysicalQuantity* of type energy.) – The projector shift of the d orbitals. **Default:** `0.0 * eV`.
- **f_orbital_shift** (*PhysicalQuantity* of type energy.) – The projector shift of the f orbitals. **Default:** `0.0 * eV`.
- **g_orbital_shift** (*PhysicalQuantity* of type energy.) – The projector shift of the g orbitals. **Default:** `0.0 * eV`.

dOrbitalShift()

Returns
The project shift for the d-orbitals.

Return type
PhysicalQuantity

`fOrbitalShift()`

Returns
The project shift for the f-orbitals.

Return type
PhysicalQuantity

`gOrbitalShift()`

Returns
The project shift for the g-orbitals.

Return type
PhysicalQuantity

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`pOrbitalShift()`

Returns
The project shift for the p-orbitals.

Return type
PhysicalQuantity

`sOrbitalShift()`

Returns
The project shift for the s-orbitals.

Return type
PhysicalQuantity

`uniqueString()`

Return a unique string representing the state of the object.

PulayMixer

`class PulayMixer(`*noncollinear_mixing=None*, *restart_strategy=None*, *history_storage=None*, *noncollinear_restart=None*`)`

Class for representing a Pulay mixer.

Parameters

- **`noncollinear_mixing`** (*`bool`*) – A special routine is used for the noncollinear mixing if this flag is used. **Default:** `False`

- **restart_strategy** (NoRestart | *AdaptiveHistoryRestart*) – The restart strategy for the SCF loop. **Default:** None
- **history_storage** (Duplicated | Distributed) – Flag indicating whether the history should be stored on each process (Duplicated) or distributed across all processes (Distributed). **Default:** Automatic: Distributed in the absence of a preconditioner and noncollinear mixing. Duplicated otherwise. **Deprecated:** from v2020.9, input will be ignored. When noncollinear_mixing is True, storage will be Duplicated, otherwise we use distributed storage.

historyStorage()

Returns
The flag indicating how the history is stored.

Return type
Duplicated | Distributed

nlinfo()

Returns
The nlinfo.

Return type
dict

noncollinearMixing()

Returns
True if the non-collinear restart mixing algorithm should be used.

Return type
bool

noncollinearRestart()

Deprecated: Use noncollinearMixing() instead.

Deprecated: from v2016.0, use noncollinearMixing() instead.

Returns
True if the non-collinear restart mixing algorithm should be used.

Return type
bool

restartStrategy()

Returns
The restart strategy.

Return type
NoRestart | *AdaptiveHistoryRestart*

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Set up a Pulay mixer with automatic memory distribution

```
# Iteration control parameters with automatic Pulay mixer
mixer=PulayMixer()
iteration_control_parameters=IterationControlParameters(algorithm=mixer)
```

Set up a Pulay mixer that uses the non-collinear restart mixing algorithm (and thus non-distributed history storage)

```
# Iteration control parameters with Pulay mixer (non-collinear mixing)
mixer=PulayMixer(noncollinear_mixing=True)
iteration_control_parameters=IterationControlParameters(algorithm=mixer)
```

Set up a Pulay mixer with explicitly distributed memory

```
# Iteration control parameters with distributed Pulay mixer.
mixer=PulayMixer(history_storage=Distributed)
iteration_control_parameters=IterationControlParameters(algorithm=mixer)
```

Notes

PulayMixer implements the Pulay-Kerker method[1].

In the Pulay mixing scheme the best guess for the next input is given by,

$$h^{\mathrm{in}}_{n+1} = \eta \sum_{i=n-n^{\mathrm{hist}}}^{n} \alpha_i h^{\mathrm{out}}_i + (1-\eta) \sum_{i=1}^{n-n^{\mathrm{hist}}} \alpha_i h^{\mathrm{in}}_i$$

where h^{in}_i is the input value and h^{out}_i is the output value of the `mixing_variable` at iteration i. Currently there is support for using `HamiltonianVariable` for mixing. The parameter η is the `damping_factor` and n^{hist} the `number_of_history_steps` used for finding the input value for the next iteration. The variables α_i are obtained by minimizing the residual error of the `mixing_variable` assuming a linear relation between the residual errors of different iteration numbers.

The *PulayMixer* is available in both a memory-distributed and a non-memory-distributed version. The non-distributed version has a slight performance advantage, but in cases where the SCF history uses a considerable amount of memory, the distributed version can be used to avoid being memory limited.

In the default case, memory distribution is activated if the stored history would use more than 50 MB. However, memory distribution cannot be used when either a preconditioner or non-collinear mixing is used. In those cases, memory distribution will be disabled by default. If memory distribution was explicitly requested with *history_storage=Distributed*, an exception will be raised when the mixer is used. This is to avoid exceeding the expected memory usage by silently falling back to duplicated history storage.

[1] P.ăPulay. Convergence acceleration of iterative sequences. the case of scf iteration. *Chem. Phys. Lett.*, 73(2):393–398, jul 1980. URL: http://www.sciencedirect.com/science/article/pii/0009261480803964.

QEqAtomicCharges

class QEqAtomicCharges(*r_cut=PhysicalQuantity(10.0, Ang)*, *error=PhysicalQuantity(1e-05, 1 / Ang)*, *chi=None*, *eta=None*, *shielding=None*, *max_cycles=50*, *mixing=0.5*)

Enables the calculation of atomic partial charges using the QEq charge equilibration scheme of Rappe and Goddard.

Parameters

- **r_cut** (*PhysicalQuantity* of type length) – The cutoff used in determining shielded charge-charge interactions. **Default:** 10 Angstrom.
- **error** (*PhysicalQuantity* of type inverse length) – The error in the 1/r lattice sum. **Default:** 0.00001 Angstrom
- **chi** (*dict*) – Custom atomic electronegativies, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type energy. **Default:** Electronegativities from the QEq method.
- **eta** (*dict*) – Custom atomic hardnesses, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type energy. **Default:** Atomic hardnesses from the QEq method.
- **shielding** (*dict*) – Custom electrostatic shielding parameters, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type inverse distance. **Default:** Shielding parameters from the QEq method.
- **max_cycles** (*int*) – The maximum number of iterations performed in QEq calculations. **Default:** 50
- **mixing** (*float*) – The linear mixing between new and old charges calculated by QEq. **Default:** 0.5

charges(*configuration*, *total_charge=None*, *decompose_bulk=True*)

Calculate the atomic charges on a given configuration.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration whose atomic charges are being calculated.
- **total_charge** (*PhysicalQuantity* of type charge) – The total charge of the configuration. **Default:** Match the integer charge of the configuration.
- **decompose_bulk** (*bool*) – Whether or not the configuration, if a BulkConfiguration, is decomposed into its parts to obtain the charges. **Default:** True

Returns
An array of atomic charges.

Return type
PhysicalQuantity of type charge

Usage Examples

Define a simulation box of 900 methanol molecules with a Dreiding calculator using QEq calculated charges

```
# ------------------------------------------------------------------------------
↪-----------
# Create Methanol Molecule
# ------------------------------------------------------------------------------
↪-----------
# Define elements
elements = [Carbon, Hydrogen, Hydrogen, Hydrogen, Oxygen, Hydrogen]

# Define coordinates
```

(continues on next page)

(continued from previous page)

```
cartesian_coordinates = [[ 0.048333359398, -0.11839206808 , -0.068353692376],
                         [ 1.138333281204, -0.11839206808 , -0.068353692376],
                         [-0.31499994787 , -0.11839206808 , -1.096015473978],
                         [-0.31499994787 , -1.008373277446,  0.445477198425],
                         [-0.363333307269,  0.889981209366,  0.513830890801],
                         [-0.013333332377,  1.747302557837,  0.018856179479]]*Angstrom

# Set up configuration
methanol = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)

# Add tags
methanol.addTags('DREI_C_3',  [0])
methanol.addTags('DREI_H_',   [1, 2, 3])
methanol.addTags('DREI_H__A', [5])
methanol.addTags('DREI_O_3',  [4])

# Add bonds
bonds = [[0, 1],
         [0, 2],
         [0, 3],
         [0, 4],
         [4, 5]]
methanol.setBonds(bonds)

# ------------------------------------------------------------------------------------------
↪-----------
# Calculate Atomic Charges
# ------------------------------------------------------------------------------------------
↪-----------
# Create the charger and calculate the charges
charger = QEqAtomicCharges()
methanol_charges = charger.charges(methanol)

# Print charges
print('METHANOL CHARGES')
print("----------------------------")
print(f'{"Number":<8s}{"Atom":<8s}{"Charge":>12s}')
print("----------------------------")
for i, q in enumerate(methanol_charges):
    element = methanol.elements()[i].symbol()
    charge = q.inUnitsOf(elementary_charge)
    print(f'{i+1:<8d}{element:<8s}{charge:>12.4f}')
print("----------------------------")

# ------------------------------------------------------------------------------------------
↪-----------
# Create A Box Of Methanol Molecules
# ------------------------------------------------------------------------------------------
↪-----------
```

(continues on next page)

(continued from previous page)

```
# Define the volume in which the molecules should be packed
u1, u2, u3 = numpy.diag([40.0] * 3) * Angstrom
cell = UnitCell(u1, u2, u3)

# Set the size of the buffer region
buffer_size = 2.0 * Angstrom

# Set Packmol parameters
tolerance = 2.0 * Angstrom
maximum_loops = 30

# Run the Packmol script to generate the packed configuration
methanol_box, packing_successful, packmol_message = runPackmol(
    [(methanol, 900)],
    cell,
    tolerance,
    buffer_size,
    maximum_loops,
)

# ------------------------------------------------------------------------------------------
↪-----------
# Add A Dreiding Calculator With The Calculated Charges
# ------------------------------------------------------------------------------------------
↪-----------
# Repeat the charges for each molecule
methanol_box_charges = numpy.tile(
    methanol_charges.inUnitsOf(elementary_charge),
    900
) * elementary_charge

# Create the potential builder
potential_builder = DreidingPotentialBuilder()

# Create the calculator, setting the atomic charges
calculator = potential_builder.createCalculator(
    methanol_box,
    atomic_charges=methanol_box_charges
)

# Assign the calculator to the configuration
methanol_box.setCalculator(calculator)

# ------------------------------------------------------------------------------------------
↪-----------
# Save The Configuration
# ------------------------------------------------------------------------------------------
↪-----------
nlsave('Methanol_Box.hdf5', methanol_box)
```

calculating_charges_example.py

Notes

The QEqAtomicCharges class implements the QEq charge equilibration scheme of Rappe and Goddard[1]. The general charge equilibration method assumes that the change of energy in response to charge polarization can be given as a simple Taylor series expansion, such that the total energy of the molecule, E, can be given as:

$$E(\mathbf{q}) = \sum_{i}^{N} \chi_i q_i + \eta_i q_i^2 + \sum_{j<i}^{N} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Here $\mathbf{q}$ is the vector of atomic charges, χ_i is the atomic electronegativity, η_i is the atomic hardness and r_{ij} is the inter-atomic distance. Getting the derivative of the energy with respect to an atomic charge gives the chemical potential for that atom. Assuming that the chemical potential of each atom is equal and that the charges sum to a particular value gives a series of linear equations that can be solved for the atomic charges. This gives a simple and yet reasonably effective method for estimating the atomic partial charges.

One modification of this idea in the QEq scheme is that the charge-charge interaction term in the above equation is not treated with point charges, but rather as the electrostatic interactions of two normalized Slater functions. This leads to a better approximation of charge shielding at short ranges. The Slater functions on each atom can be given as:

$$\phi(\mathbf{r}; n, \zeta) = N r^{n-1} e^{-\zeta r}$$

Here N is a normalization constant, ζ is the principal quantum number and ζ is the orbital exponent. This sets the effective size of the Slater function on the atom. As Slater electrostatic integrals are difficult to calculate, a number of effective approximations of this term have been suggested[2].

Periodic systems also present an additional challenge to the QEq scheme. Due to the long-range nature of electrostatic interactions, in periodic systems the charge-charge interactions should be calculated with Ewald summation[3]. As the Slater electrostatic term approaches $\frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$ at long distances, this long range interaction can be replaced with the appropriate Ewald sum.

Another important modification in QEq is in how hydrogen is treated. Since the atomic orbitals of hydrogen are significantly different to its molecular orbitals, it is not sufficient to use atomic parameters in calculating charge equilibration in molecules. Charge dependent corrections to the atomic hardness and Slater orbital exponent are therefore used to make hydrogen respond to charge polarization in a more molecular way. As these corrections are charge dependent this means that the QEq charge calculations have to be solved self consistently. This can be done using a simple linear mixing scheme.

To calculate the partial atomic charges for a configuration, an instance of the QEqAtomicCharges class first has to be created. This class has a few parameters to control how it is instantiated. The cutoff used to calculate the charge-charge interactions is controlled with the parameter `r_cut`. In the case of a *MoleculeConfiguration* this defines the distance beyond which electrostatic interactions are ignored. For large molecules such as polymers this may need to be increased so that all interactions are considered. In the case of a *BulkConfiguration* this defines the distance beyond which electrostatic interactions are assumed to be $\frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$ and calculated with Ewald summation. Before this distance electrostatic interactions are assumed to be Slater electrostatic integrals.

In the QEq scheme each atom has three parameters, the atomic electronegativity, hardness and Slater orbital exponent. These can be modified using the parameters `chi`, `eta` and `shielding` respectively. This can be important in cases where atomic charges are being sought for ionic systems. In the QEq scheme the energy of the atom is expanded around a neutral charge. In ionic systems it is more appropriate to expand the energy around the ionic charge of the atom. This different atomic reference state is reflected in different atomic parameters. Each parameter takes dictionary

[1] AnthonyăK. Rappe and WilliănăA. GoddardăIII. Charge equilibration for molecular dynamics simulations. *J. Phys. Chem.*, 95:3358, April 1991. URL: https://pubs.acs.org/doi/10.1021/j100161a070, doi:10.1021/j100161a070.

[2] Akifumi. Oda and Shuichi Hirono. Geometry-dependent atomic charge calculations using charge equilibration method with empirical two-center Coulombic terms. *J. Mol. Struct. THEOCHEM*, 634:159, September 2003. URL: https://www.sciencedirect.com/science/article/pii/S0166128003003385, doi:10.1016/S0166-1280(03)00338-5.

[3] Sundar Ramachandran, T.ăG. Lenz, W.ăM. Skiff, and A.ăK. Rappe. Toward an understanding of zeolite Y as a cracking catalyst with the use of periodic charge equilibration. *J. Phys. Chem.*, 100:5898, April 1996. URL: https://pubs.acs.org/doi/10.1021/jp952864q, doi:10.1021/jp952864q.

of *PeriodicTableElement* and PhysicalQuanity of appropriate units. If parameters are not specified for an atom the default atomic parameters are used. The following example shows how to explicitly set the default atomic parameters for the methanol example above.

```
charger = QEqAtomicCharges(
   chi={Hydrogen: 4.528*eV, Carbon: 5.343*eV, Oxygen: 8.741*eV},
   eta={Hydrogen: 6.9452*eV, Carbon: 5.063*eV, Oxygen: 6.682*eV},
   shielding={Hydrogen: 2.0216/Ang, Carbon: 1.6469/Ang, Oxygen: 1.8685/Ang}
)
methanol_charges = charger.charges(methanol)
```

Finally there are two parameters that control the iterative solving of the QEq equations. This is only necessary in configurations that contain hydrogen. The `mixing` parameter takes a number between 0 and 1 that represents the amount of charges to add from the current and previous solutions to add at each iteration. The parameter `max_cycles` sets the maximum number of self consistency iterations. In some cases these parameters may need to be modified to help the self consistent solution to converge.

Once the QEqAtomicCharges class is instantiated, the atomic charges can be calculated using the `charges` method. This takes the configuration of the system and returns an array of atomic charges. The total charge of the configuration can be set using the `total_charge` argument. In cases where the charges are sought for a group of individual molecules, the charges can be calculated individually for each molecule using the `decompose_bulk` argument. This is useful in cases such as a polymer melt or a molecular liquid. Calculating the charges of each molecule individually ensures that each molecule has the correct total charge, as well as removing any artifacts of how the molecules are packed. It also can speed up the calculation of the charges significantly. This is currently only supported for *BulkConfiguration* configurations.

QNumbers

class QNumbers(*md_trajectory*, *element_selection*, *fuzz_factor=None*, *start_time=None*, *end_time=None*, *calculate_distribution=True*, *time_resolution=None*, *info_panel=None*)

Class for calculating the extended coordination number (Q-Numbers) for an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the Q-numbers for.
- **element_selection** – The triple of elements that is used to calculate the q-numbers. The first element is the central element in the coordination, the second element specifies the required first neighbor to the central atom, and the third element denotes that the first neighbor needs to have at least one neighbor atom of that element.
- **fuzz_factor** (*float*) – The fuzz factor which the sum of the covalent radii is multiplied with, to obtain the cutoff radii. **Default:** 1.2
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **calculate_distribution** (*bool*) – A flag to determine whether the distribution of Q-numbers or the time-dependent evolution of Q-Numbers along the trajectory should be calculated. **Default:** True (the distribution is calculated)
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Returns
The q-number number distribution if calculate_distribution has been selected, other wise an array with dimensions (n_snapshots, q_max+1) with the time-dependent evolution of the various q-numbers.

Return type
numpy array

qNumbers()

Returns
The Q-number values associated with the histogram if calculate_distribution has been selected, None, otherwise.

Return type
numpy array | None

times()

Returns
The MD-times in case the time-evolution has been calculated, None otherwise.

Return type
PhysicalQuantity of type time | None

Usage Examples

Load an *MDTrajectory* and calculate the q-number distribution for a O-Si-O path:

```
md_trajectory = nlread('sio2.nc')[-1]

q_number = QNumbers(
    md_trajectory,
    element_selection=[Oxygen, Silicon, Oxygen],
    fuzz_factor=1.2,
    calculate_distribution=True,
)

# Get the histogram and the associated the coordination numbers.
bin_edges = q_number.qNumbers() - 0.4
histogram = q_number.data()

# Plot the data using pylab.
import pylab

pylab.bar(bin_edges, histogram, label='O-Si-O', width=0.8)
pylab.xlabel('Q-number')
pylab.ylabel('Histogram')
pylab.legend()

pylab.show()
```

q_numbers.py

Calculate and plot the time evolution of the q-numbers for a O-Si-O path:

```
md_trajectory = nlread('sio2.nc')[-1]
```

(continues on next page)

(continued from previous page)

```
q_number = QNumbers(
    md_trajectory,
    element_selection=[Oxygen, Silicon, Oxygen],
    fuzz_factor=1.2,
    calculate_distribution=False,
)


# Get the histogram and the associated the coordination numbers.
times = q_number.times().inUnitsOf(fs)
q_number_evolution = q_number.data()

# Plot the data using pylab.
import pylab

# Make a plot for each q-number.
n_q_numbers = q_number_evolution.shape[1]
for i in range(n_q_numbers):
    label = 'Q%i O-Si-O' % i
    pylab.plot(times, q_number_evolution[:, i], label=label)
    pylab.xlabel('Time (fs)')
    pylab.ylabel('Q-number count')


pylab.legend()

pylab.show()
```

q_numbers2.py

Notes

The Q-Number analysis is an extended coordination number. The coordination of each atom of the central element with atoms of the primary neighbor element is counted. In contrast to the regular *CoordinationNumber* analysis, the primary neighbors must have at least one neighbor atom to the secondary neighbor element[1].

This quantity is often used to interpret NMR or Raman experiments, to identify e.g. the coordination number of oxide tetrahedra, such as SiO_4.

This analysis has two modes, distribution and time evolution, which can be switched via the `calculate distribution` argument. In the distribution analysis the histogram of the Q-numbers is plotted, similar to the coordination number analysis. In the time evolution analysis, the change of the number of atoms associated with a Q-number is analyzed as a function the simulation time. This analysis is performed each Q-number detected in the snapshots of the simulation, as shown in the second example, this yields $Q_{max} + 1$ data sets, where Q_{max} is the largest Q-Number found in the simulation.

The neighbors are identified via cutoff radii, which are determined as the sum of the covalent radii of the elements multiplied by the `fuzz_factor`, similar to the way the bonds are displayed in the **Viewer**.

In contrast to most other MD-Analysis functions, only elements can be given in the `element_selection`.

[1] H.ăManzano, J.ăS. Dolado, M.ăGriebel, and J.ăHamaekers. A molecular dynamics study of the aluminosilicate chains structure in Al-rich calcium silicate hydrated (C-S-H) gels. *Phys. Stat. Sol. (a)*, 205(6):1324–1329, 2008. doi:10.1002/pssa.200778175.

QuantumATKMTPPotential

class QuantumATKMTPPotential(*file*, *suppress_intercept=False*, *group_name=''*)

Moment Tensor Potential for Synopsys-QuantumATK pre-trained MTP models

Parameters

- **file** (*str*) – The name of a file containing the data for a fitted moment tensor potential.
- **suppress_intercept** (*bool*) – Suppress the intercept of the linear model. This will cause the energy to deviate from the reference by some system-independent additive constant, but will ensure that the energy of the empty system is 0.
- **group_name** (*str*) – Optional name of a hdf5-group in file containing the data for a fitted moment tensor potential, if the data is not in the root-group of file

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Method that returns the defaults.

Returns
The defaults.

Return type
dict

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

This class is used with Synopsys pre-trained moment tensor potential (MTP) parameter files. For more information consult the *MTPPotential* manual page.

QuasiParticleConfigurationInteraction

class QuasiParticleConfigurationInteraction(*configuration*, *eigensolutions*, *number_of_particles=None*, *basis_state_indices=None*, *dielectric_constant=None*, *excitation_order=None*, *two_particle_integral_method=None*)

A class to set up and solve a many-body body hamiltonian using a Configuration Interaction (CI) method on selected confined quasi-particle levels of a bulk (e.g., electron or hole states in a quantum dot).

Parameters

- **configuration** (*BulkConfiguration*) – The configuration for which the CI eigensolution should be calculated.
- **eigensolutions** (*Eigensolutions*.) – The single-particle eigensolutions to be used as a basis for the Configuration Interaction method.
- **number_of_particles** (*int*) – The number of particles to be included in the many-body problem. Note that for amounts larger than 2 the complexity explodes and the calculation might become too heavy. **Default:** 2
- **basis_state_indices** (*list of int*) – A list of single particle state indices be used as basis for the CI method. The indices are referred to the lowest unoccupied state: 0 is the LUMO, +1 is the LUMO+1, -1 is the HOMO etc. **Default:** [0, 1]
- **dielectric_constant** (*float*) – The dielectric constant of the material, used to include a screening in the two-particle Coulomb interaction. It should be equal to the high frequency dielectric constant of the material where the quasi-particle basis states are confined. **Default:** 1
- **excitation_order** (*float*) – The maximum allowed excitation order in the CI expansion. 1 corresponds to single excitations (CIS), 2 to double excitations (CISD), and so on. If left unspecified, full-CI is performed. **Default:** Full-CI is performed.
- **two_particle_integral_method** (*TwoParticleCoulombInteractionFFTMethod* | *TwoParticleCoulombInteractionPointChargeMethod*) – Select the method used to evaluate two particle Coulomb integrals. **Default:** *TwoParticleCoulombInteractionFFTMethod*.

basisStateIndices()

Returns
the indices identifying the states used as quasi-particle basis, with the LUMO as reference for the index 0.

Return type
list of int

dielectricConstant()

Returns
the dielectric constant used to screen the two-particle Coulomb interaction.

Return type
float

eigenvalues()

Returns
the many-body eigenvalues, as absolute energies.

Return type
PhysicalQuantity of type energy.

excitationOrder()

Returns
the excitation order used in the CI expansion.

Return type
int.

manyBodyProbabilityDensity(*state_index*, *eigensolutions*, *density_mesh_cutoff=None*)

Calculate the real space probability density associated to a many body state.

Parameters

- **state_index** (*int*) – The many-body eigenstate for which the probability density is calculated. The index follow the eigenvalue ordering, from lowest to highest.

- **eigensolutions** (*Eigensolutions*) – The single-particle eigensolutions to be used as a basis for the Configuration Interaction method. Note that these must be consistent with the eigensolutions provided as input parameter in the object constructor.
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the density grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** The density mesh cutoff set on the calculator.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the object.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfParticles()

Returns
the number of particles in the many-body system.

Return type
float

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

singleParticleEigenvalues()

Returns
the single-particle eigenvalues of the selected basis states, as absolute energies.

Return type
PhysicalQuantity of type energy.

twoParticleIntegrals()

Get the two-particle Coulomb interaction integrals that are required to set up the many-body Hamiltonian.

The return value is a map between four single particle state indices (a, b, c, d) and the two-particle spatial integrals:

$$(a,b|c,d) = \int \phi_a^*(\mathbf{r}_1)\phi_b(\mathbf{r}_1)V(\mathbf{r}_1-\mathbf{r}_2)\phi_c^*(\mathbf{r}_2)\phi_d(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2$$

The indices go from 0 to N-1, where N is the number of single particle basis states utilized.

Returns
A map between indices and the corresponding two-particle interaction integral value.

Return type
dict

`uniqueString()`

Return a unique string representing the state of the object.

RSL2Potential

`class RSL2Potential(particleType1, particleType2, a, b, c, r_cut=None)`

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of type energy*) – Potential parameter.
- **b** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **c** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

`classmethod getAllParameterNames()`

Return the names of all used parameters as a list.

`getAllParameters()`

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

`static getDefaults()`

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

`getParameter(parameterName)`

Get the current value of the parameter parameterName.

`setParameter(parameterName, value)`

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

The **revised central force** potential of Stillinger and Rahman **(RSL2)**[1], defines an interaction of the form

$$V_{ij}(r) = \frac{a_{ij}}{1 + \exp(b_{ij}(r - c_{ij}))} .$$

RadialDistribution

class RadialDistribution(*md_trajectory*, *start_time=None*, *end_time=None*, *cutoff_radius=None*, *resolution=None*, *pair_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the radial distribution from an *MDTrajectory*.

Parameters

- **md_trajectory** (*MDTrajectory* | *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The MDTrajectory or configuration the radial distribution should be calculated for.
- **cutoff_radius** (*PhysicalQuantity* of type length) – Upper limit on sampled distances (must be positive). **Default:** Half the diagonal of the unit cell
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last frame time.
- **resolution** (*PhysicalQuantity* of type length) – The binning of the histogram. **Default:** `0.05 * Angstrom`
- **pair_selection** (*sequence*) – Only include contributions between this selection of atoms. Either None or a sequence containing two of the following types: Element, tag name, list of indices, or None. **Default:** All atoms pairs are considered.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the radial distribution function.

distances()

Return the distance values associated with the radial distribution function.

Usage Examples

Load an *MDTrajectory* and calculate the *RadialDistribution* function between Al and O:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

rdf = RadialDistribution(md_trajectory,
                         cutoff_radius=8.0*Angstrom,
                         pair_selection=[Aluminum, Oxygen])

# Get the bin_centers and the histogram of the radial distribution.
distances = rdf.distances().inUnitsOf(Angstrom)
histogram = rdf.data()
```

(continues on next page)

[1] F.ăH. Stillinger and A.ăRahman. Revised central force potentials for water. *J. Chem. Phys.*, 1978. doi:10.1063/1.435738.

(continued from previous page)

```
# Plot the data using pylab.
import pylab

pylab.plot(distances, histogram, label='Al-O RDF')
pylab.xlabel('r (Ang)')
pylab.ylabel('g(r)')
pylab.legend()

pylab.show()
```

radial_distribution.py

Calculate the *RadialDistribution* function between the first atom in the configuration and the rest of the system:

```
rdf = RadialDistribution(md_trajectory,
                         cutoff_radius=20.0*Angstrom,
                         pair_selection=[[0], None])
```

radial_distribution_2.py

Notes

The radial distribution (or pair correlation) function is calculated as

$$g(r) = \frac{1}{4\pi r^2}\frac{1}{N\rho}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\langle\delta(r-|\mathbf{r}_i-\mathbf{r}_j|)\rangle$$

It is also possible to calculate the partial radial distribution by specifying two selections A and B (elements or index lists) in `pair_selection`. In that case the radial distribution is calculated as

$$g_{AB}(r) = \frac{1}{4\pi r^2}\frac{N}{\rho N_A N_B}\sum_{i\in A}\sum_{j\in B, j\neq i}^{N}\langle\delta(r-|\mathbf{r}_i-\mathbf{r}_j|)\rangle$$

The normalization via the density ensures that for large distances the radial distribution approaches unity.

As the radial distribution assumes a homogeneous system, this analysis is not well-defined for trajectories of *DeviceConfiguration* or *SurfaceConfiguration* types. In that case only the radial distribution of the central region atoms is calculated, using infinite electrodes as boundary conditions.

RadiusOfGyration

class RadiusOfGyration(*md_trajectory*, *polymer_tags=None*, *start_time=None*, *end_time=None*, *time_resolution=None*, *mass_weight=None*, *gui_worker=None*)

Analyzer to calculate the radius of gyration and shape factors of polymer chains.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or AtomicConfiguration that contains the polymers.
- **polymer_tags** (*list of type str | None | Automatic*) – A list of tags for the polymers. If not specified default tags are used. Tags can also be set automatically with the flag Automatic

- **start_time** (*PhysicalQuantity* of the type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of the type time) – The end time. **Default:** The last time image.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis. **Default:** the MD time step.
- **mass_weight** (*bool*) – Whether or not masses of atoms are included in the calculations. **Default:** True.
- **gui_worker** (*PolymerAnalysisWorker*) – Handle to the worker, for displaying the calculation progress. **Default:** None.

acylindricalFactor()

Returns
The acylindrical factor for each polymer chain. This is returned as a 2D array indexed by trajectory image then polymer chain.

Return type
ndarray

anisotropicFactor()

Returns
The anisotropic factor for each polymer chain. This is returned as a 2D array indexed by trajectory image then polymer chain.

Return type
ndarray

asphericalFactor()

Returns
The aspherical factor for each polymer chain. This is returned as a 2D array indexed by trajectory image then polymer chain.

Return type
ndarray

components()

Returns
The components of the radius of gyration. This is returned as a 3D array indexed by trajectory image, polymer chain, component.

Return type
PhysicalQuantity of type length

polymerTags()

Return the polymer tags used for the analysis.

Returns
A list of the polymer tags.

Return type
list

radius()

Returns
The radius of gyration of the polymers. This is returned as a 2D array indexed by trajectory image and then polymer chain.

Return type
PhysicalQuantity of type length

`times()`

Returns
Times for each image used in the analysis.

Return type
`PhysicalQuantity` of type time

Usage Examples

Calculate the average radius of gyration of polystyrene chains during a molecular dynamics simulation starting from a well equilibrated configuration.

```
# ------------------------------------------------------------
# Initial Configuration
# ------------------------------------------------------------
bulk_configuration = nlread('Polystyrene.hdf5')[-1]

# ------------------------------------------------------------
# OPLS-AA Calculator
# ------------------------------------------------------------
potential_builder = OPLSPotentialBuilder()
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------
initial_velocity = ConfigurationVelocities(
    remove_center_of_mass_momentum=True
)

method = NPTMartynaTobiasKlein(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    reservoir_pressure=1*bar,
    thermostat_timescale=100*femtoSecond,
    barostat_timescale=500*femtoSecond,
    initial_velocity=initial_velocity,
    heating_rate=0*Kelvin/picoSecond,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='Polystyrene_Trajectory.hdf5',
    steps=100000,
    log_interval=10,
```

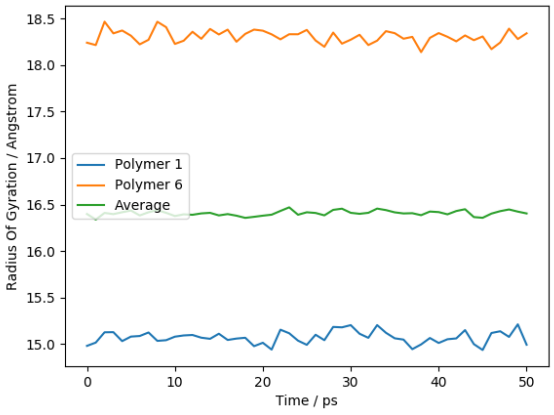
(continues on next page)

(continued from previous page)

```
    trajectory_interval=1000,
    method=method
)

# ------------------------------------------------------------
# Calculate Radius Of Gyration
# ------------------------------------------------------------
analyzer = RadiusOfGyration(
    md_trajectory,
    start_time=0*ps,
    end_time=50*ps
)

radius_of_gyration = analyzer.radius()
radius_of_gyration_average = radius_of_gyration.mean(axis=1)
time = analyzer.times().convertTo(ps)

# ------------------------------------------------------------
# Plot The Results
# ------------------------------------------------------------
import pylab
pylab.figure()
pylab.plot(time, radius_of_gyration[:, 0], label='Polymer 1')
pylab.plot(time, radius_of_gyration[:, 5], label='Polymer 6')
pylab.plot(time, radius_of_gyration_average, label='Average')
pylab.xlabel('Time / ps')
pylab.ylabel('Radius Of Gyration / Angstrom')
pylab.legend()
pylab.savefig('Radius_Of_Gyration_Plot.png')
```

`RadiusOfGyration_Example.py` `Polystyrene.hdf5`

The script will run a short molecular dynamics calculation from the given starting structure and then calculate the radius of gyration for two individual polymers, as well as the average. This is shown in the following plot



Notes

This class enables the calculation of the radius of gyration and polymer chain shape descriptors from a *MDTrajectory* or *BulkConfiguration*[1]. The radius of gyration, R_g can be defined as:

$$R_g = \sqrt{\frac{\sum_i^N m_i(\mathbf{r}_i - \mathbf{r}_{COM})^2}{\sum_i^N m_i}}$$

Here m_i represents the mass of the atoms, $\mathbf{r}_i$ represents the atomic positions and $\mathbf{r}_{COM}$ represents the center of mass. As atomic masses are included, R_g defined here is the mass-weighted radius of gyration. Assuming each atomic mass is equivalent, produces the unweighted radius of gyration. The radius of gyration gives a measure of the overall size of the polymer chain.

Details about the shape can also be calculated. The principal components of the radius of gyration can be found by diagonalizing the gyration tensor. This gives three components λ_x, λ_y and λ_z such that:

$$R_g^2 = \lambda_x^2 + \lambda_y^2 + \lambda_z^2$$

These components of the radius of gyration can give further information about the overall shape of the polymer chain. The asphericity of a polymer chain b measures the distance of the chain from a spherically symmetric shape, and is given as:

$$b = \lambda_x^2 - \frac{1}{2}\left(\lambda_y^2 + \lambda_z^2\right)$$

Similarly the acylindricity of the polymer chain c measures the deviation of the chain from a cylindrical shape. This can be defined as:

$$c = \lambda_y^2 - \lambda_z^2$$

Finally, the anisotropy of the polymer chain κ^2 measures the amount to which the shape of the polymer is orientated is a particular direction. This can be defined as:

$$\kappa^2 = \frac{3}{2}\frac{\lambda_x^4 + \lambda_y^4 + \lambda_z^4}{(\lambda_x^2 + \lambda_y^2 + \lambda_z^2)^2} - \frac{1}{2}$$

To calculate each of these shape measures a *RadiusOfGyration* object first needs to be created. This takes a *MDTrajectory* or *BulkConfiguration* and calculates each of the size and shape measures for the specified configurations. The specific frames in the *MDTrajectory* to be analyzed can be selected using the `start_time`, `end_time` and `time_resolution` arguments. Whether or not mass weighting is used can be selected with the `mass_weight` argument. By default, mass weighting is used when calculating the radius of gyration. The polymer chains analyzed in the configuration can be specified with the `polymer_tags` argument. This takes a list of tags of polymer molecules in the configuration. By default polymers tagged with `POLYMER_MOLECULE_#` are analyzed, where `#` is a numerical index. These tags are added automatically to polymer configurations built with the *PolymerMonteCarloBuilder*, or can be added using the function *tagPolymerMolecules()*

Once the *RadiusOfGyration* object is created and the values of each of the descriptors has been calculated, the values can be queried using the remaining methods. Each of these returns arrays of values indexed first by the trajectory frame, then by the polymer. Averaging over all frames can be done by averaging over axis 0, while averaging over polymer chains is achieved by averaging over axis 1.

[1] DorosăN. Theodorou and UlrichăW. Suter. Shape of unperturbed linear polymers: polypropylene. *Macromolecules*, 18:1206, June 1985. URL: https://pubs.acs.org/doi/abs/10.1021/ma00148a028, doi:10.1021/ma00148a028.

RamanSpectrum

class RamanSpectrum(*configuration*, *dynamical_matrix*, *susceptibility_derivatives*, *qpoint=None*, *phonon_modes=None*, *method=None*, *polarization_in=None*, *polarization_out=None*, *energies=None*, *broadening=None*, *number_of_angles=None*, *polarization_orthogonal=None*, *second_harmonics_generation_susceptibility=None*)

Analysis class for calculating the Raman tensor and spectrum for a bulk configuration.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration for which the vibrational modes should be calculated. If no calculator is attached to the configuration, the calculator from dynamical_matrix is used. Currently, configuration must be the same as the configuration from dynamical_matrix and is therefore not needed. **Default:** The configuration from `dynamical_matrix`

•**dynamical_matrix** (*DynamicalMatrix*) – The DynamicalMatrix to calculate the phonon modes from.

•**susceptibility_derivatives** (*SusceptibilityDerivatives*) – SusceptibilityDerivatives object.

•**qpoint** (*list of 3 floats*) – fractional q-point where phonon modes used for Raman signals are evaluated, e.g. `[0.0, 0.0, 0.0]`. **Default:** Gamma point, G=[0.0, 0.0, 0.0].

•**phonon_modes** (list with positive ints | `All`) – The phonon modes to include. **Default:** `All` (All phonon modes are included).

•**method** (`PolarizationDependent | PolarizationAveraged`.) – Keyword specifying if the results should be calculated with a light polarization dependent or averaged result. When `method=PolarizationDependent`, `polarization_in` must be provided, `polarization_out` can be given but is defaulted to the `polarization_in` value. When `method=PolarizationAveraged`, `polarization_in/out` is not used. **Default:** `PolarizationDependent`

•**polarization_in** (*list of 3 floats*) – Incoming light polarization vector, e.g. `[1.0, 0.0, 0.0]`. The vector will be normalized internally. **Default:** `[1, 0, 0]`

•**polarization_out** (*list of 3 floats*) – Scattered light polarization vector, e.g. `[1.0, 0.0, 0.0]`. The vector will be normalized internally. Only a single scattered light polarization vector will be calculated. This option is mutually exclusive to `polarization_orthogonal` and `number_of_angles`, which will results in multiple scattered light polarizations.

•**energies** (*PhysicalQuantity* of type energy) – The energies where Raman spectrum is evaluated. **Default:** `1500 points from 0.0 to 10 percent above highest phonon frequency`

•**broadening** (*PhysicalQuantity* of type energy) – The broadening parameter used for the Raman spectrum. **Default:** `0.25 * meV (~2 cm1)`

•**number_of_angles** (*integer*) – Argument used for `method=PolarizationDependent`. If not None a polar spectrum will be generated with the scattered light polarization rotated between polarization_in and polarization_orthogonal (orthogonal polarization). This option is mutually exclusive to `polarization_out`. **Default:** None

•**polarization_orthogonal** (*list of 3 floats*) – Argument used for `method=PolarizationDependent`. Light polarization vector used for polar spectrum that is orthogonal to polarization_in. Scattered light polarization will be rotated between polarization_in and polarization_orthogonal, g_s = cos(theta) * polarization_in + sin(theta) * polarization_orthogonal. The vector will be normalized internally. **Default:** None

•**second_harmonics_generation_susceptibility** (*SecondHarmonicsGenerationSusceptibility* | `None`) – If given a correction from polar phonons will be included in the susceptibility derivatives. An optical_spectrum and born_effective_charge will be taken from the dynamical matrix argument including polar phonon splitting. **Default:** `None`, i.e. the polar phonon correction is not included.

broadening()

Returns
The broadening parameter used for the Raman spectrum.

Return type
PhysicalQuantity of type energy.

energies()

Returns
The energies used for the spectrum.

Return type
PhysicalQuantity of type energy.

intensities(*angle_index=None*)

Returns
The Raman tensor for the phonon modes projected onto light polarizations. For several angles this gives the polar Raman spectrum for the phonon modes for a particular angle index. The shape is (N, M) where N is the number of angles angles and M is the number of phonon modes.

Return type
ndarray with floats.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

method()

Returns
The method used for the calculation. Either `PolarizationDependent` or `PolarizationAveraged`.

Return type
NLFlag

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

numberOfAngles()

Returns
The number of angle degrees used for the polar Raman spectrum.

Return type
one-dimensional ndarray with floats.

phononModes()

Query method for the phonon modes.

Returns
The list of phonon mode indices used in the calculation.

Return type
list of int.

phononWaveNumbers()

Returns
The wave numbers used for the spectrum.

Return type
PhysicalQuantity of type inverse length.

polarAngleList()

Returns
The angle degrees used for the polar Raman spectrum.

Return type
one-dimensional ndarray with floats | None

polarizationOrthogonal()

Returns
The normalized polarization that scattered light is rotated to in polar spectrum.

Return type
ndarray with 3 floats each.

polarizationsIn()

Returns
Normalized incoming light polarization vector

Return type
numpy.ndarray with 3 floats.

polarizationsOut()

Returns
Normalized scattered light polarization vectors

Return type
list of numpy.ndarray with 3 floats.

qpoint()

Returns
The q-point for which the phonon modes and Raman signals are calculated.

Return type
ndarray with 3 floats.

ramanSpectrum(*angle_index=None*)

Parameters
angle_index (*int*) – The index of the scattered light, given in the list polarizationsOut().

Returns
The raman_spectrum summed over phonon modes for a given angle index

Return type
one-dimensional ndarray with floats.

ramanTensor()

Returns
The Raman tensor for the phonon modes. The shape is (M, 3, 3) where M is the number of phonon modes.

Return type
ndarray with floats.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

susceptibilityDerivativesPolarCorrection()

Returns
The correction to the susceptibility derivatives from polar phonons related to the second harmonics generation susceptibility, Born effective charges and optical spectrum. The two latter are obtained from the dynamical matrix with polar phonon splitting. It will only be nonzero if a second_harmonics_generation_susceptibility was provided as an input. The shape is (M, 3, 3) where M is the number of degrees of freedom.

Return type
ndarray with floats.

uniqueString()

Return a unique string representing the state of the object.

waveNumberAxis()

Returns
The wave numbers used for the spectrum.

Return type
PhysicalQuantity of type inverse length.

Usage Examples

Calculate the *RamanSpectrum* for an incoming and scattered light polarization equal to [1, 0, 0].

```
raman_spectrum = RamanSpectrum(
  configuration,
  dynamical_matrix,
  susceptibility_derivatives,
  method=PolarizationDependent,
  polarization_in=[1, 0, 0],
  polarization_out=[1, 0, 0])
```

When giving `polarization_orthogonal`, a polar spectrum is generated for a number of out polarizations given by `number_of_angles`.

```
raman_spectrum = RamanSpectrum(
    configuration,
    dynamical_matrix,
    susceptibility_derivatives
    method=PolarizationDependent,
```

(continues on next page)

(continued from previous page)

```
    polarization_in=[1, 0, 0],
    polarization_orthogonal=[0, 1, 0],
    number_of_angles=101)
```

When using `method=PolarizationAveraged`, the light polarization is averaged and not given as input.

```
raman_spectrum = RamanSpectrum(
    configuration,
    dynamical_matrix,
    susceptibility_derivatives
    method=PolarizationAveraged)
```

Notes

Raman spectroscopy is a powerful noninvasive technique for material characterization that uniquely extracts information about vibrational and chemical properties, inhomogeneities, strain, crystallinity, electron-phonon coupling and anharmonicities in a local environment. Hereby Raman spectroscopy has become one of the main tools for characterization of emerging materials. Below we give a brief introduction to the theory of light-matter scattering and outline the implementation of Raman scattering in QuantumATK.

Light scattering in materials

The classical description of light scattering is based on the polarization of a material in the presence of an electric field. The polarization is given by

$$P_\alpha = p_\alpha + \varepsilon_0 \chi^{(1)}_{\alpha\beta} E_\beta + \varepsilon_0 \chi^{(2)}_{\alpha\beta\gamma} E_\beta E_\gamma + ...$$

where p_α is the permanent dipole moment, α, β, γ are Cartesian directions and χ is the dielectric susceptibility tensor. The electric field can be expressed as a vector $\mathbf{E} = E_0 \mathbf{g} e^{i(\mathbf{k}_l \mathbf{r} - \omega_l t)}$ with $\mathbf{g}$ giving the light polarization vector, which is orthogonal to the propagation direction, and $\mathbf{k}_l, \omega_l$ giving the wavevector and angular frequency of the light, respectively. The linear response function $\chi^{(1)}_{\alpha\beta}$ describes how light can be modulated elastically from infrared and Rayleigh scattering or inelastically from Raman scattering. The nonlinear response function $\chi^{(2)}_{\alpha\beta\gamma}$ describes higher order phenomena such as second harmonic generation.

Phenomenological linear response

Focusing on the linear response in the electric field, assuming a time-dependent electric field of the form $E_\alpha = E_{0\alpha} \cos(\omega_l t)$ and Taylor expanding to lowest order in the vibrational modulation of the polarization by a time-dependent normal mode vibration $u_\lambda = u_{0\lambda} \cos(\omega_\lambda t)$, we have:

$$P_\alpha \approx$$

$$p_{0\alpha} + \sum_\lambda \left.\frac{\partial p_{0\alpha}}{\partial u_\lambda}\right|_{u=0} u_{0\lambda} \cos(\omega_\lambda t)$$

$$+$$

$$\varepsilon_0 \chi^{(1)}_{0\alpha\beta} E_{0\beta} \cos(\omega_l t) + \sum_\lambda \left.\frac{\partial \chi^{(1)}_{\alpha\beta}}{\partial u_\lambda}\right|_{u=0} u_{0\lambda} \cos(\omega_\lambda t) E_{0\beta} \cos(\omega_l t)$$

Here λ labels the phonon modes. The second term describes a dipole moment oscillating with the vibrational mode frequency. Vibrational modes for which the permanent dipole moment changes with displacement are called infrared

active modes. The third term describes the Rayleigh scattering at the same frequency as the incident light. Finally the fourth term gives rise to Raman scattering. From the trigonometric identity $2\cos(\omega_l t)\cos(\omega_\lambda t) = \cos((\omega_l - \omega_\lambda)t) + \cos((\omega_l + \omega_\lambda)t)$ we see that this light induced polarization is Raman shifted by the phonon frequencies. If the susceptibility changes with a given vibration the mode is Raman active.

Raman signals - Polarization dependent method

In solids the Raman spectra depends on the polarization of the incoming and scattered light compared to the crystal directions. The nonresonant Raman power cross-section from one-phonon excitations is within the Placzek approximation given by[1][2]:

$$\frac{d\sigma(\omega)}{dV} = N_{cells}\frac{\omega_s}{c^4 V}|\mathbf{g}_s \cdot \alpha(\lambda) \cdot \mathbf{g}_i^T|^2 \times \frac{\hbar}{2\omega_\lambda}(n_\lambda + 1)\,\delta(\omega - \omega_\lambda)$$

where $\omega_{s/i}$ ($\mathbf{g}_{s/i}$) is the frequency (polarization vector) of the scattered/incoming light, N_{cells} is the the number of unit cells in the sample, V is the unit cell volume, n_λ is the Bose-Einstein occupation of phonon mode λ. Energy conservation imposes $\omega_s = \omega_i \mp \omega_\lambda$ for Stokes and anti-Stokes processes, respectively. We use the Kubo-Greenwood formula to calculate the 3×3 susceptibility matrix, see *OpticalSpectrum*. The derivatives are calculated using the finite differences, see *SusceptibilityDerivatives*.

Raman Tensor

The Raman tensor is given by

$$\alpha_{\alpha\beta}(\lambda) = V\left.\frac{\partial \chi^{(1)}_{\alpha\beta}}{\partial u_\lambda}\right|_{u=0} = V\sum_\mu \sum_l \frac{\partial \chi_{\alpha\beta}}{\partial x_l} e_l^\lambda / \sqrt{m_\mu}$$

where μ is the atom index, m_μ is the atom mass, e_l^λ is the phonon eigenvector of mode λ and the derivative is evaluated with finite-differences. The Raman tensor is a 3×3 matrix for each phonon mode.

Raman Spectrum

We define the Raman spectrum as the broadened intensities obtained from the projected Raman tensor:

$$I_\lambda^{Raman} = |\mathbf{g}_s \cdot \alpha(\lambda) \cdot \mathbf{g}_i^T|$$

This is a number/intensity for each phonon mode λ which can then be combined to a spectrum by using an artificial broadening σ. Several broadening frameworks can be used, but we apply the Gaussians:

$$I(\omega) = \sum_\lambda I_\lambda^{Raman}\delta(\omega - \omega_\lambda) \approx \sum_\lambda I_\lambda^{Raman}\frac{\exp\left(-(\omega - \omega_\lambda)^2/2\sigma^2\right)}{\sqrt{2\pi}\sigma}\,.$$

[1] P.ăUmari and Alfredo Pasquarello. Infrared and Raman spectra of disordered materials from first principles. *Diamond and Related Materials*, 14(8):1255–1261, August 2005. URL: http://www.sciencedirect.com/science/article/pii/S0925963504004510 (visited on 2018-01-09), doi:10.1016/j.diamond.2004.12.007.

[2] Dirk Porezag. Infrared intensities and Raman-scattering activities within density-functional theory. *Phys. Rev. B*, 54(11):7830–7836, 1996. doi:10.1103/PhysRevB.54.7830.

Raman signals - Polarization averaged results

From above we have the polarization dependent expression for the Raman power cross-section. One can derive an expression for the polarization averaged power cross-section. This is useful for modeling of at least two cases: (i) the crystal orientation and light polarization is unknown or (ii) experiments are averaged over a (large) variation of crystal orientations. A typical example of the latter is for a collection of molecules in a liquid with arbitrary orientations. The polarization averaged expression is given by:

$$\frac{d\sigma(\omega)}{dV} = N_{cells}\frac{\omega_s}{c^4 V}\frac{I^{Raman}}{45}\frac{\hbar}{2\omega_\lambda}\left(n_\lambda + 1\right)\delta(\omega - \omega_\lambda)$$

The polarization averaged Raman intensity is given by

$$I_\lambda^{Raman} = 45\tilde{\alpha}_\lambda^2 + 7\tilde{\beta}_\lambda^2$$

where

$$\tilde{\alpha}_\lambda = \frac{1}{3}\left(\alpha_{xx} + \alpha_{yy} + \alpha_{zz}\right)$$

is the average polarization and

$$\tilde{\beta}_\lambda^2 = \frac{1}{2}[(\alpha_{xx} - \alpha_{yy})^2 + (\alpha_{xx} - \alpha_{zz})^2 + (\alpha_{yy} - \alpha_{zz})^2] + 3(\alpha_{xy}^2 + \alpha_{xz}^2 + \alpha_{xz}^2)$$

is the anisotropy factor of the susceptibility tensor derivative.

Raman signals - Polar Raman spectrum

It is often interesting to see how the Raman intensities for each phonon mode changes with angle between the input and scattered light. This is actually also measurable if one has a polarizer in the Raman setup that can be rotated. We plot a polar Raman spectrum by rotating the scattered light polarization between the input polarization $\mathbf{g}_i$ and an orthogonal polarization direction $\mathbf{g}_\perp$. Specifically we rotate the scattered light polarization as

$$\mathbf{g}_s(\theta) = \cos(\theta)\mathbf{g}_i + \sin(\theta)\mathbf{g}_\perp$$

The projected Raman tensor

$$I_\lambda^{Raman}(\theta) = |\mathbf{g}_s(\theta)\cdot\alpha(\lambda)\cdot\mathbf{g}_i^T(\theta)|$$

then gives the intensity spectrum as a function of polarization rotation angle.

RandomBlochWaveInitialization

class RandomBlochWaveInitialization

The plane-wave random initialization method class.

Usage Example

Define a random initial subspace for a *GeneralizedDavidsonSolver*:

```
GeneralizedDavidsonSolver(initialization_method=RandomBlochWaveInitialization())
```

RandomDirection

class RandomDirection(*magnitude=None*)

Used to generate random directions.

Parameters
magnitude (*PhysicalQuantity of type length or tuple.*) – The magnitude of the returned directions. If this is a scalar, all the directions will have the same magnitude. If it is a tuple, the magnitude will be randomly chosen between the two values. **Default:** 1 * Angstrom

getMagnitude()

Get the magnitude of the directions.

Returns
The magnitude of the directions.

Return type
PhysicalQuantity of type length

getNextDirection()

Get the next direction.

Returns
The next random direction.

Return type
numpy.ndarray

initialize(*num_dimensions*)

Initialize the generator with the number of dimensions.

Parameters
num_dimensions (*int*) – The number of dimensions of the generated vectors.

isInitialized()

Check if the generator is initialized.

Returns
True if the generator is initialized.

Return type
bool

reset()

Reset the direction generator.

Notes

RandomDirection is used to generate initial directions for the different saddle search methods in *AdaptiveKineticMonteCarlo*. *RandomDirection* uses a naive approach where a set of number of random numbers are first generated. The random numbers are used to create a vector which is normalized and then multiplied with the specified magnitude.

After instantiating a new *RandomDirection* object, the *initialize()* method should be called the desired number of vector components an argument.

RandomDisplacementsParameters

class RandomDisplacementsParameters(*reference_configurations*, *supercell_repetitions_list=None*, *strain_tensors=None*, *sample_size=None*, *system_sizes=None*, *atomic_rattling_amplitudes=None*, *cell_rattling_amplitudes=None*, *random_seed=None*, *log_filename_prefix=None*, *data_tag=None*)

Class for storing parameters for generating a set of training configurations using a series of random atomic displacements.

Parameters

- **reference_configurations** (*MoleculeConfiguration* | *BulkConfiguration* | sequence of[MoleculeConfiguration | *BulkConfiguration*] | *Table*) – One or more reference configurations to be used to generate the training set. For bulk configurations, this is the unit cell from which a supercell can be defined and additional strain can be applied.
- **supercell_repetitions_list** (*sequence (size 3) of int | sequence of sequence (size 3) of int | None*) – The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. Cannot be specified if there are molecules in the list of reference configurations. If None: No repetitions. **Default:** `None`.
- **strain_tensors** (*sequence (size 3) of sequence (size 3) of float | sequence of sequence (size 3) of sequence (size 3) of float | None*) – The list of strains to apply to each configuration. Cannot be specified if there are molecules in the list of reference configurations. If None: No strain. **Default:** `None`.
- **sample_size** (*int*) – The number of training configurations to generate for each combination of reference configuration, supercell, strain tensor, atomic rattling amplitude, and cell rattling amplitude. **Default:** `20`.
- **system_sizes** (*int | sequence of int*) – Minimum number of atoms in the system. If None then supercell_repetitions_list will be used to do the repetition of the cell. If set, supercell_repetitions_list will be ignored. **Default:** `None`
- **atomic_rattling_amplitudes** (*PhysicalQuantity* of type length | sequence of|PHYSICALQUANTITY| of typ e length) – The list of maximum random displacements of the atomic positions. **Default:** `0.1 * Angstrom`.
- **cell_rattling_amplitudes** (*float | sequence of [float, None]*) – The list of maximum random strains of the cell volume. Only applies to bulk configurations. If None: No strain. DEFAULT| `None`.
- **random_seed** (*int*) – The random seed used for generating the displacements. **Default:** Generated automatically.
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting. **Default:** `None`.

atomicRattlingAmplitudes()

Returns
The list of maximum random displacements of the atomic positions.

Return type
list of *PhysicalQuantity* of type length

`cellRattlingAmplitudes()`

Returns
The list of maximum random strains of the cell volume (None for no strain).

Return type
list of [float | None]

`dataTag()`

Returns
The selection tag added to the data in the training set.

Return type
str

`logFilenameIdentifier()`

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

`logFilenamePrefix()`

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | `LogToStdOut` | None

`randomSeed()`

Returns
The random seed used for generating the displacements, or None if it should be generated automatically.

Return type
int | None

`referenceConfigurations()`

Returns
The list of reference configurations to be used to generate the training set.

Return type
list of [*MoleculeConfiguration* | *BulkConfiguration*]

`sampleSize()`

Returns
The number of training configurations for each combination of list parameters.

Return type
int

`strainTensors()`

Returns
The list of strains to apply to each configuration (None for no strain).

Return type
list of [numpy.ndarray | None]

supercellRepetitionsList()

Returns
The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
list of tuple (size 3) of int

systemSizes()

Parameters
system_sizes (*int | sequence of int*) – Minimum number of atoms in the system. If None then supercell_repetitions_list will be used to do the repetition of the cell. If set, supercell_repetitions_list will be ignored. **Default:** None

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Setup of a training set of quartz using *RandomDisplacementsParameters*.

```
# Set up lattice
lattice = Hexagonal(4.966693338812276*Angstrom, 5.4647751809821905*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.469217958328,  0.              ,  0.              ],
                          [ 0.              ,  0.469217958328,  0.666666666667],
                          [ 0.530782041672,  0.530782041672,  0.333333333333],
                          [ 0.409091887977,  0.269718793416,  0.115058361235],
                          [ 0.269718793416,  0.409091887977,  0.551608638765],
                          [ 0.730281206584,  0.139373094561,  0.781725361235],
                          [ 0.590908112023,  0.860626905439,  0.218274638765],
                          [ 0.860626905439,  0.590908112023,  0.448391361235],
                          [ 0.139373094561,  0.730281206584,  0.884941638765]]

# Set up reference configuration of quartz
reference_configuration = BulkConfiguration(
   bravais_lattice=lattice,
   elements=elements,
   fractional_coordinates=fractional_coordinates,
)

# Set up the list of isotropic volume strains that should be applied.
volume_strain = [0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15]
identity_matrix = numpy.eye(3)

# Define RandomDisplacementsParameters used to generate the training set.
```

(continues on next page)

(continued from previous page)

```
training_sets = RandomDisplacementsParameters(
   reference_configurations=reference_configuration,
   supercell_repetitions_list=[(1, 1, 1), (2, 2, 2)],
   strain_tensors=[identity_matrix*s**0.3333333 for s in volume_strain],
   sample_size=10,
   atomic_rattling_amplitudes=0.15*Angstrom,
   cell_rattling_amplitudes=0.07,
   log_filename_prefix=None,
)
```

Note here that each combination of input arguments creates `sample_size=10` configurations.

Notes

The *RandomDisplacementsParameters* class can be used to generate training configurations by randomly displacing the atoms of a given reference configuration around their equilibrium positions. Additionally, isotropic and anisotropic strain can be applied to the cell. For the actual training data generation a *RandomDisplacementsParameters* object needs to be passed into a *MomentTensorPotentialTraining* object.

To achieve a broadly applicable training set of configurations, a combination of different displacement magnitudes should be used. Small displacement magnitudes typically provide a good description of vibrations and phonons, while larger displacement magnitudes typically increase the stability of the simulation over a wider temperate range below the melting point. This provides a very efficient way to generate training data for crystal materials without having to run long expensive ab-initio MD simulations. It is recommended to use various supercell sizes to improve the training of the total energy.

A good default protocol, that provides a balanced combination of different displacement and strain magnitudes can be set up via the function *crystalTrainingRandomDisplacements*.

Note, that such a displacement protocol can also be used to generate training data for crystal surfaces and interfaces. Here, one would include the most relevant surfaces or interface configurations, such as combinations of different cleave planes, and proceed as outlined above.

RandomSpin

class RandomSpin(*scaled_spins=None*)

Constructor for RandomSpin.

Parameters
scaled_spins (*list*) – The limit for the initial random polarization for each atom. **Default:** `1.0 for each atom.`.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

See *InitialSpin* for a detailed discussion on setting the initial spin.

Notes

RandomSpin works essentially the same way *InitialSpin* does, except that the input parameter `scaled_spins` is used as a limit α_i for the initial spin polarization on each atom i . The actual initial spin set for atom i will be a random number in the interval $[0, \alpha_i]$ (or $[\alpha_i, 0]$ if $\alpha_i < 0$).

ReactionTemplate

class ReactionTemplate(*image_list=None*, *bond_cutoff=None*, *angle_exclusion=None*, *exclude_rings_below_size=None*, *relative_reaction_probability=None*)

Create a reaction template object that describes the bond changes and reacting functional groups for a specific reaction sequence.

Parameters

- **image_list** (*list*) – A list of MoleculeConfigurations that define the reaction template.
- **bond_cutoff** (PhysicalQuantity of type length | list) – The bond cutoff used in the defined reactions.
- **angle_exclusion** (PhysicalQuantity of type angle | list) – The angle exclusion used in the defined reactions.
- **exclude_rings_below_size** (*int | All | list*) – The ring exclusion size used in the defined reactions.
- **relative_reaction_probability** (*list*) – The relative reaction probabilities used in the defined reactions.

findReactiveGroupMap(*configuration*, *add_tags_to_configuration=False*)

Find the reactive groups in the starting configuration.

Parameters

- **configuration** (BulkConfiguration) – The configuration to find the reactive groups in.
- **add_tags_to_configuration** (*bool*) – Whether or not reactive tags are added to the configuration. **Default:** False.

Returns
The list of dictionaries defining the reactive groups.

Return type
list

numberOfReactionSteps()

Returns
The number of reaction steps in the template.

Return type
int

reactions(*bond_cutoff=None*, *angle_exclusion=None*, *exclude_rings_below_size=None*, *relative_reaction_probability=None*)

Return the reactions defined by the template.

Parameters

- **bond_cutoff** (PhysicalQuantity of type length | list) – The bond cutoff used in the defined reactions.

- **angle_exclusion** (PhysicalQuantity of type angle | list) – The angle exclusion used in the defined reactions.
- **exclude_rings_below_size** (*int | All | list*) – The ring exclusion size used in the defined reactions.
- **relative_reaction_probability** (*list*) – The relative reaction probabilities used in the defined reactions.

Returns
The list of CrosslinkReaction objects that defines the reaction scheme.

Return type
list

templateConfiguration(*index*)

Return the MoleculeConfiguration used to define the reaction.

Parameters
index (*int*) – Index of the desired configuration.

Returns
The template MoleculeConfiguration.

Return type
MoleculeConfiguration

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Build a network polymer system based on an amine-epoxy reaction with a reaction template

```
# Load the templates.
template_a = nlread('Amine_Epoxy_Templates.hdf5')[0]
template_b = nlread('Amine_Epoxy_Templates.hdf5')[1]
template_c = nlread('Amine_Epoxy_Templates.hdf5')[2]

# Load the initial configuration for cross-linking.
configuration = nlread('Amine_Epoxy_Crosslink.hdf5')[-1]

# Create the ReactionTemplate object.
template = ReactionTemplate(
    image_list=[template_a, template_b, template_c],
    bond_cutoff=4*Angstrom,
    exclude_rings_below_size=0
)

# Define the CrosslinkConnector, using the ReactionTemplate.
crosslink_connector = CrosslinkConnector(
    configuration,
    reactions=template.reactions(),
    reactive_atom_groups=template.findReactiveGroupMap(configuration),
    max_number_of_reactions=None
)

# Define the CrosslinkBuilder object, using the OPLS potential to model the material.
opls_potential = OPLSPotentialBuilder()
```

(continues on next page)

(continued from previous page)

```
polymer_builder = CrosslinkBuilder(
    crosslink_connector,
    cycles_to_terminate=None,
    reaction_percentage=100,
    nvt_steps_per_crosslink=1000,
    npt_steps_per_crosslink=1000,
    potential_builder=opls_potential,
    temperature=480*Kelvin,
    pressure=1*bar,
    bond_search_increment=0.5*Angstrom,
    max_bond_search_radii=6*Angstrom
)

# Run the polymer building simulation.
polymer_builder.runCrosslinking(trajectory_filename='Thermoset_Amine_Epoxy.hdf5')
```

`Amine_Epoxy_Crosslink.py` `Amine_Epoxy_Crosslink.hdf5` `Amine_Epoxy_Templates.hdf5`

Note that the example here has been created to be illustrative of the process of building a thermoset material. For high-quality simulations it may be necessary to increase the number of molecular dynamics steps or the size of the system.

Notes

The *ReactionTemplate* class provides an alternative way to define reactions performed by the *CrosslinkConnector*. On initialization the object requires a list of *MoleculeConfiguration* objects that define the bonding of the atoms at each stage of the reaction. By analyzing the changes in bonding between successive configurations, the *ReactionTemplate* class determines the *CrosslinkReaction* objects required to perform those reactions. The individual molecules in each reactant stage also act as templates of the possible reactive groups. These molecules are matched against atoms in the configuration being reacted, to find all of the possible reactive groups. Both the reactions and the reactive groups can then be used to define a *CrosslinkConnector* object.

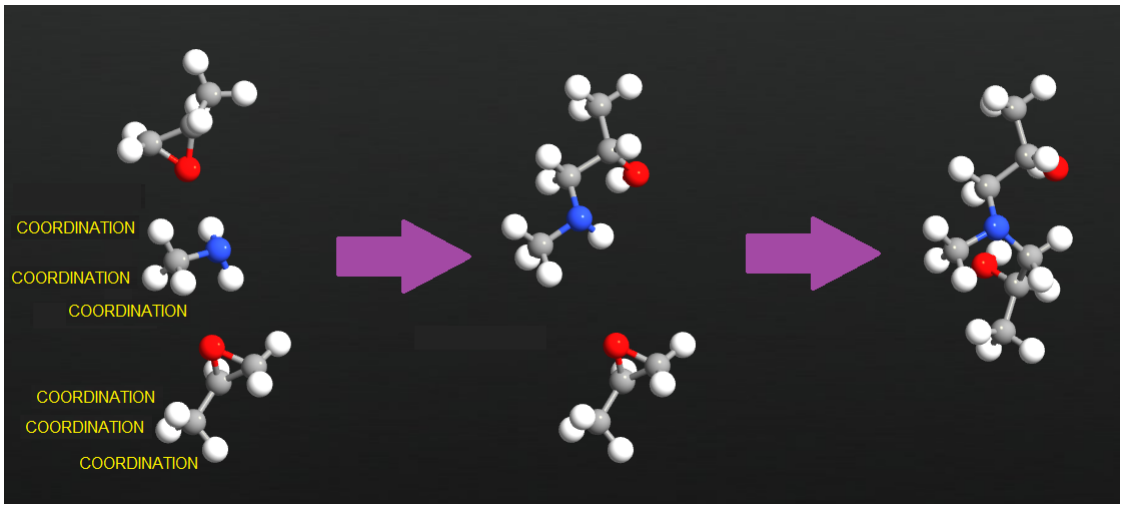


Fig. 4.34: The template configurations for an amine-epoxy reaction scheme.

The configurations that are used to define the amine-epoxy reaction used in the example is shown in the figure above. When creating configurations to model a reaction sequence, one constraint is that the atoms in each configuration must

be in the same order. This is so that otherwise identical atoms can be identified across configurations. When creating multiple reaction steps, all of the molecules that participate in the reaction must be present at each stage, even if they do not react at that stage. In the case of the epoxy-amine example, it can be seen that the template requires two epoxy molecules, even though only one reacts at each stage. The *ReactionTemplate* class is also able to find reactive groups in a configuration. It does this by finding the segments of the configuration molecular graph that match molecules in a reactant step in the reaction sequence. To ensure that the template molecules remain general and chemically sensible, it is possible to set which atoms are used to match reactive groups in the reacting configuration. Atoms labeled with the tag `COORDINATION` are only used to indicate the coordination number of the atom that they are bonded to, rather than being searched for explicitly. So in the amine-epoxy example, the hydrogens on the methyl groups bonded to both the amine and the epoxy are labeled with the tag `COORDINATION`. This means that when the *ReactionTemplate* class looks for reactive groups in a configuration, it matches amine or epoxy groups that are bonded to 4-coordinate carbons. The nature of the atoms that the carbon is bonded to, aside from the epoxy or amine, is not taken into account. Note also that as it looks explicitly for 4-coordinate carbons, it will not match aromatic groups, such as the aromatic amine in aniline. To match aromatic carbons, a 3-coordinate carbon must be specified.

Note: As the atoms in configurations used to describe a reaction need to be in the same order, it can be easiest to create the initial configuration in the Molecule Builder. The configurations for additional stages can then be created by duplicating the initial configuration. As it can change atom ordering, it is also recommended to turn off `Auto passivation` in the Molecular Builder when changing the bonding in a configuration.

Atoms to be removed from the configuration after a reaction can be specified by removing all bonds to those atoms. Single atoms with no bonds in a configuration beyond the initial configuration are assumed to form small molecules that are removed after the reaction. As an example, in the condensation reaction of an alcohol with a carboxylic acid, a water molecule is produced along with the ester. To remove the water molecule after the reaction, the bonds between the hydrogens and oxygen atom in the water molecule need to be removed. The product configuration will therefore contain an ester molecule, a single oxygen atom and two single hydrogen atoms.

On initialization the `MoleculeConfiguration` that defines the chemistry of the reaction. Optionally, it is also possible to specify some parameters of the *CrosslinkReaction* objects that are created by the template. The arguments `bond_cutoff`, `angle_exclusion`, `exclude_rings_below_size` and `relative_reaction_probability` are all passed to the *CrosslinkReaction* objects and have the same meaning. The first three arguments can be specified with either a single value, which is passed to all *CrosslinkReaction* objects, or a list with a value for each object. As it only makes sense to have different values for different reactions, the `relative_reaction_probability` argument only accepts a list of values for each reaction.

Once the *ReactionTemplate* object is created, the reactive groups in a configuration can be found using the method `findReactiveGroupMap`. This takes as an argument the configuration which is to be reacted. It also takes an optional argument `add_tags_to_configuration`, which specifies whether or not the tags defining different reactions are added to the configuration. Placing tags on the configuration is not necessary when using a reaction template, and so by default the method does not place tags on the configuration. However, as tags are altered by the *CrosslinkBuilder*, adding tags can help to analyze the progress of a reaction. The `findReactiveGroupMap` method returns a list of dictionaries that define all of the reactive groups in the configuration. This is in the correct format to be directly passed to the *CrosslinkConnector*. A copy of the *CrosslinkReaction* objects contained by the template can also be returned with the method `reactions`. This takes the same 4 optional arguments as the initialization method, allowing for the parameters of the returned reaction objects to be changed without re-defining the template. The number of reaction steps in the template can also be returned with the method `numberOfReactionSteps`. Finally the *MoleculeConfiguration* objects used to define each stage of the reaction can be returned with the method `templateConfiguration`. This takes an integer argument indicating the index of the configuration that is returned.

RealAxisContour

class RealAxisContour(*real_axis_point_density=None*, *real_axis_infinitesimal=None*, *real_axis_kbt_padding_factor=None*)

A non-equilibrium contour using the real axis contour points and weights defined in[1].

Parameters

- **real_axis_point_density** (*PhysicalQuantity* of type energy) – Spacing between the points on bias window integration line close to the real axis. **Default:** `0.001 * Hartree`
- **real_axis_infinitesimal** (*PhysicalQuantity* of type energy) – Small imaginary shift of the bias window integration line from the real axis. **Default:** `0.001 * Hartree`
- **real_axis_kbt_padding_factor** (*float > 0.0*) – The integration line is extended with the amount fk_bT below and above the electrode Fermi levels where f is this value. **Default:** `5.0`

nlinfo()

Returns
The nlinfo.

Return type
dict

realAxisInfinitesimal()

Returns
The imaginary shift.

Return type
PhysicalQuantity of type energy

realAxisKbtPaddingFactor()

Returns
The padding factor.

Return type
float

realAxisPointDensity()

Returns
The spacing between the points.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

[1] M.ăBrandbyge, J.-L. Mozos, P.ăOrdejon, J.ăTaylor, and K.ăStokbro. Density-functional method for nonequilibrium electron transport. *Phys. Rev. B*, 65:165401, Mar 2002. doi:10.1103/PhysRevB.65.165401.

Usage Example

One can use the RealAxisContour by defining it as an non-equilibrium contour

```
non_equilibrium_contour = RealAxisContour()
```

which constructs a RealAxisContour with all defaults. Alternatively, more parameters that alter the accuracy of the approximation can be specified, e.g.

```
non_equilibrium_contour = RealAxisContour(
    real_axis_point_density=0.002*Hartree,
    real_axis_infinitesimal=0.003*Hartree,
    real_axis_kbt_padding_factor=10.0)
```

To use it in a calculation of a the density matrix for a system with finite bias, the *NonEquilibriumContour* object is passed to the *ContourParameters* object,

```
contour_parameters = ContourParameters(non_equilibrium_contour=non_equilibrium_contour)
```

and saved on the calculator

```
device_calculator = DeviceLCAOCalculator(contour_parameters=contour_parameters)
```

Notes

If a system has a finite bias, the density matrix has both an equilibrium and a non-equilibrium contribution (See *Notes*):

$$D_{\mu\nu} = D^L_{\mu\nu} + \Delta^R_{\mu\nu} = D^R_{\mu\nu} + \Delta^L_{\mu\nu}$$

The RealAxisContour provides a method to solve the non-equilibrium part Δ through a numerical integration of:

$$\Delta^R_{\mu\nu} = \int_{-\infty}^{+\infty} d\epsilon\, \rho^L_{\mu\nu}(\epsilon)\,[f(\epsilon-\mu_R) - f(\epsilon-\mu_L)]$$

in which ρ^L is the spectral density matrix. The integration is performed over the bias window, extended by the `real_axis_kbt_padding_factor`.

To increase the accuracy of the approximation the user can specify the `real_axis_point_density`.

Because of the finite accuracy of the approximation the left and right approaches to calculate the density matrix can lead to a different result. The integration error can be quantified by:

$$e_{\mu\nu} = D^L_{\mu\nu} + \Delta^R_{\mu\nu} - (D^R_{\mu\nu} + \Delta^L_{\mu\nu})$$

For ways to deal with this error see *DoubleContour*.

More information about this approach can be found in[Page 3719, 1].

RealSpaceSelfEnergyPDOS

class RealSpaceSelfEnergyPDOS(*configuration*, *primitive_configuration*, *kpoints=None*, *energies=None*, *infinitesimal=None*, *projections=None*, *primitive_layer_extensions=None*, *automatic_primitive_layer_extensions=True*, *use_recursive_method=None*)

Parameters

- **configuration** (*BulkConfiguration*) – The configuration of the supercell with attached calculator for which to calculate the projected DOS.
- **primitive_configuration** (*BulkConfiguration*) – The configuration to be used for self-energy calculations.
- **kpoints** – The k-points for which to calculate the projected density of states. The kpoints are used to sample the real-space self-energy for the primitive configuration in the A- and B directions only. **Default:** KpointDensity(30 * Ang)
- **energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the projected density of states. The energies are relative to the Fermi energy of the primitive configuration. **Default:** `numpy.linspace(-1.0, 1.0, 101) * eV`
- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the transmission calculation away from the real axis. **Default:** `1.0e-4*eV`
- **projections** (list of *Projection* | *Projection* | *ProjectionGenerator*) – The projections used for the calculating the weights. **Default:** `ProjectOnElements`.
- **primitive_layer_extensions** – The number of times the primitive cell is repeated inwards. This determines where the self-energy is added to the configuration. Given as a list of int for A, B, and C direction.
- **automatic_primitive_layer_extensions** (*bool*) – Boolean controlling if the primitive_layer_extensions should be determined automatically based on the range of the Hamiltonian.
- **use_recursive_method** (*bool*) – Boolean to determine if the recursive method should be used or not.

automaticPrimitiveLayerExtensions()

Returns
Boolean determining if the recursive method should be used or not.

Return type
bool.

bandsAboveFermiLevel()

Returns
The number of bands above the Fermi level per principal spin channel.

Return type
int

densityOfStates(*normalization=None*)

Returns the possible normalized density of states.

Parameters
normalization (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** `False`

Returns
The full density of states as a vector of the length of the number of energies.

Return type
PhysicalQuantity of type reciprocal energy

directBandGap()

Returns
The direct band gap.

Return type
PhysicalQuantity of type energy

energies()

Returns
The energies.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero.

Return type
PhysicalQuantity of type energy.

evaluate(*projection_index=None*, *normalization=None*)
The projected density of states for a given projection.

Parameters

- **projection_index** (*int*) – The index of the projection to query the projected density of states for. Negative indexing can be used such that e.g. projection_index=-1 will return the projected density of states for the last projection. **Default:** The projected density of states for each projection is returned.
- **normalization** (False` | *NormalizeByVolume()* | *NormalizeByNumberOfAtoms()*) – Normalization scheme to use for the DOS. **Default:** `False`

Returns
The projected density of states as a vector of the length of the number of energies, e. If `projection_index` is not specified, an array of shape (p, e) is returned, where p is the number of projections.

Return type
PhysicalQuantity of type reciprocal energy

fermiLevel(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin the Fermi level should be returned for. Must be either `Spin.Up`, `Spin.Down`, or `Spin.All`. Only when the band structure is calculated with a fixed spin moment will the Fermi level depend on spin. **Default:** `Spin.Up`

Returns
The Fermi level in absolute energy.

Return type
`PhysicalQuantity` of type energy

indirectBandGap()

Returns
The indirect band gap.

Return type
PhysicalQuantity of type energy

infinitesimal()

Returns
The imaginary energy part.

Return type
PhysicalQuantity of type energy

kpoints()

Returns
The k-point sampling used for calculating the projected density of states.

Return type
MonkhorstPackGrid | *RegularKpointGrid*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the PDOS.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveLayerExtensions()

Returns
The number of times the primitive cell is repeated inwards. This determines where the self-energy is added to the configuration.

Return type
int

projections()

Returns
The projections used for the calculated projected density of states.

Return type
list of class:~.*Projection* | class:~.*Projection*

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()
Return a unique string representing the state of the object.

useRecursiveMethod()

Returns
Boolean determining if the recursive method should be used or not.

Return type
bool.

Usage Examples

Calculate the RealSpaceSelfEnergyPDOS for a graphene sheet with a Stone-Wales defect in the middle.

```
# Set up lattice
lattice = SimpleOrthorhombic(4.26258*Angstrom, 15.0*Angstrom, 4.922*Angstrom)

# Define elements
elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon]

# Define coordinates
fractional_coordinates = [[ 0.083333333333,  0.5           ,  0.124999739368],
                          [ 0.083333333333,  0.5           ,  0.625000086877],
                          [ 0.416666666667,  0.5           ,  0.124999739368],
                          [ 0.416666666667,  0.5           ,  0.625000086877],
                          [ 0.583333333333,  0.5           ,  0.374999913123],
                          [ 0.583333333333,  0.5           ,  0.875000260632],
                          [ 0.916666666667,  0.5           ,  0.374999913123],
                          [ 0.916666666667,  0.5           ,  0.875000260632]]

# Set up configuration
primitive_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )


# %% SemiEmpiricalCalculator
hamiltonian_parametrization = SlaterKosterHamiltonianParametrization(
    basis_set=Hancock.C_Basis)

k_point_sampling = KpointDensity(
    density_a=4.0*Angstrom,
    density_b=4.0*Angstrom,
    density_c=4.0*Angstrom
)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling
)
```

(continues on next page)

(continued from previous page)

```
calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    checkpoint_handler=NoCheckpointHandler
)


# %% Set Calculator
primitive_configuration.setCalculator(calculator)
primitive_configuration.update()


# Set up lattice
lattice = SimpleOrthorhombic(46.88838*Angstrom, 15.0*Angstrom, 54.
↪141999999999996*Angstrom)

# Define elements
elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,
            
```

(continued from previous page)

```
            Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,
```

(continued from previous page)

```
            Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,
```

(continued from previous page)

```
                        [ 0.007575757576,  0.5           ,  0.193181794488],
                        [ 0.007575757576,  0.5           ,  0.238636371534],
                        [ 0.037878787879,  0.5           ,  0.193181794488],
                        [ 0.037878787879,  0.5           ,  0.238636371534],
                        [ 0.05303030303 ,  0.5           ,  0.215909083011],
                        [ 0.05303030303 ,  0.5           ,  0.261363660057],
                        [ 0.083333333333,  0.5           ,  0.215909083011],
                        [ 0.083333333333,  0.5           ,  0.261363660057],
                        [ 0.007575757576,  0.5           ,  0.284090885397],
                        [ 0.007575757576,  0.5           ,  0.329545462443],
                        [ 0.037878787879,  0.5           ,  0.284090885397],
                        [ 0.037878787879,  0.5           ,  0.329545462443],
                        [ 0.05303030303 ,  0.5           ,  0.30681817392 ],
                        [ 0.05303030303 ,  0.5           ,  0.352272750967],
                        [ 0.083333333333,  0.5           ,  0.30681817392 ],
                        [ 0.083333333333,  0.5           ,  0.352272750967],
                        [ 0.007575757576,  0.5           ,  0.374999976306],
                        [ 0.007575757576,  0.5           ,  0.420454553352],
                        [ 0.037878787879,  0.5           ,  0.374999976306],
                        [ 0.037878787879,  0.5           ,  0.420454553352],
                        [ 0.05303030303 ,  0.5           ,  0.397727264829],
                        [ 0.05303030303 ,  0.5           ,  0.443181841876],
                        [ 0.083333333333,  0.5           ,  0.397727264829],
                        [ 0.083333333333,  0.5           ,  0.443181841876],
                        [ 0.007575757576,  0.5           ,  0.465909067215],
                        [ 0.007575757576,  0.5           ,  0.511363644262],
                        [ 0.037878787879,  0.5           ,  0.465909067215],
                        [ 0.037878787879,  0.5           ,  0.511363644262],
                        [ 0.05303030303 ,  0.5           ,  0.488636355738],
                        [ 0.05303030303 ,  0.5           ,  0.534090932785],
                        [ 0.083333333333,  0.5           ,  0.488636355738],
                        [ 0.083333333333,  0.5           ,  0.534090932785],
                        [ 0.007575757576,  0.5           ,  0.556818158124],
                        [ 0.007575757576,  0.5           ,  0.602272735171],
                        [ 0.037878787879,  0.5           ,  0.556818158124],
                        [ 0.037878787879,  0.5           ,  0.602272735171],
                        [ 0.05303030303 ,  0.5           ,  0.579545446648],
                        [ 0.05303030303 ,  0.5           ,  0.625000023694],
                        [ 0.083333333333,  0.5           ,  0.579545446648],
                        [ 0.083333333333,  0.5           ,  0.625000023694],
                        [ 0.007575757576,  0.5           ,  0.647727249033],
                        [ 0.007575757576,  0.5           ,  0.69318182608 ],
                        [ 0.037878787879,  0.5           ,  0.647727249033],
                        [ 0.037878787879,  0.5           ,  0.69318182608 ],
                        [ 0.05303030303 ,  0.5           ,  0.670454537557],
                        [ 0.05303030303 ,  0.5           ,  0.715909114603],
                        [ 0.083333333333,  0.5           ,  0.670454537557],
                        [ 0.083333333333,  0.5           ,  0.715909114603],
                        [ 0.007575757576,  0.5           ,  0.738636339943],
                        [ 0.007575757576,  0.5           ,  0.784090916989],
                        [ 0.037878787879,  0.5           ,  0.738636339943],
                        [ 0.037878787879,  0.5           ,  0.784090916989],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.05303030303 ,  0.5           ,  0.761363628466],
                        [ 0.05303030303 ,  0.5           ,  0.806818205512],
                        [ 0.083333333333,  0.5           ,  0.761363628466],
                        [ 0.083333333333,  0.5           ,  0.806818205512],
                        [ 0.007575757576,  0.5           ,  0.829545430852],
                        [ 0.007575757576,  0.5           ,  0.875000007898],
                        [ 0.037878787879,  0.5           ,  0.829545430852],
                        [ 0.037878787879,  0.5           ,  0.875000007898],
                        [ 0.05303030303 ,  0.5           ,  0.852272719375],
                        [ 0.05303030303 ,  0.5           ,  0.897727296421],
                        [ 0.083333333333,  0.5           ,  0.852272719375],
                        [ 0.083333333333,  0.5           ,  0.897727296421],
                        [ 0.007575757576,  0.5           ,  0.920454521761],
                        [ 0.007575757576,  0.5           ,  0.965909098807],
                        [ 0.037878787879,  0.5           ,  0.920454521761],
                        [ 0.037878787879,  0.5           ,  0.965909098807],
                        [ 0.05303030303 ,  0.5           ,  0.943181810284],
                        [ 0.05303030303 ,  0.5           ,  0.98863638733 ],
                        [ 0.083333333333,  0.5           ,  0.943181810284],
                        [ 0.083333333333,  0.5           ,  0.98863638733 ],
                        [ 0.098484848485,  0.5           ,  0.01136361267 ],
                        [ 0.098484848485,  0.5           ,  0.056818189716],
                        [ 0.128787878788,  0.5           ,  0.01136361267 ],
                        [ 0.128787878788,  0.5           ,  0.056818189716],
                        [ 0.143939393939,  0.5           ,  0.034090901193],
                        [ 0.143939393939,  0.5           ,  0.079545478239],
                        [ 0.174242424242,  0.5           ,  0.034090901193],
                        [ 0.174242424242,  0.5           ,  0.079545478239],
                        [ 0.098484848485,  0.5           ,  0.102272703579],
                        [ 0.098484848485,  0.5           ,  0.147727280625],
                        [ 0.128787878788,  0.5           ,  0.102272703579],
                        [ 0.128787878788,  0.5           ,  0.147727280625],
                        [ 0.143939393939,  0.5           ,  0.124999992102],
                        [ 0.143939393939,  0.5           ,  0.170454569148],
                        [ 0.174242424242,  0.5           ,  0.124999992102],
                        [ 0.174242424242,  0.5           ,  0.170454569148],
                        [ 0.098484848485,  0.5           ,  0.193181794488],
                        [ 0.098484848485,  0.5           ,  0.238636371534],
                        [ 0.128787878788,  0.5           ,  0.193181794488],
                        [ 0.128787878788,  0.5           ,  0.238636371534],
                        [ 0.143939393939,  0.5           ,  0.215909083011],
                        [ 0.143939393939,  0.5           ,  0.261363660057],
                        [ 0.174242424242,  0.5           ,  0.215909083011],
                        [ 0.174242424242,  0.5           ,  0.261363660057],
                        [ 0.098484848485,  0.5           ,  0.284090885397],
                        [ 0.098484848485,  0.5           ,  0.329545462443],
                        [ 0.128787878788,  0.5           ,  0.284090885397],
                        [ 0.128787878788,  0.5           ,  0.329545462443],
                        [ 0.143939393939,  0.5           ,  0.30681817392 ],
                        [ 0.143939393939,  0.5           ,  0.352272750967],
                        [ 0.174242424242,  0.5           ,  0.30681817392 ],
                        [ 0.174242424242,  0.5           ,  0.352272750967],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.098484848485,  0.5           ,  0.374999976306],
                        [ 0.098484848485,  0.5           ,  0.420454553352],
                        [ 0.128787878788,  0.5           ,  0.374999976306],
                        [ 0.128787878788,  0.5           ,  0.420454553352],
                        [ 0.143939393939,  0.5           ,  0.397727264829],
                        [ 0.143939393939,  0.5           ,  0.443181841876],
                        [ 0.174242424242,  0.5           ,  0.397727264829],
                        [ 0.174242424242,  0.5           ,  0.443181841876],
                        [ 0.098484848485,  0.5           ,  0.465909067215],
                        [ 0.098484848485,  0.5           ,  0.511363644262],
                        [ 0.128787878788,  0.5           ,  0.465909067215],
                        [ 0.128787878788,  0.5           ,  0.511363644262],
                        [ 0.143939393939,  0.5           ,  0.488636355738],
                        [ 0.143939393939,  0.5           ,  0.534090932785],
                        [ 0.174242424242,  0.5           ,  0.488636355738],
                        [ 0.174242424242,  0.5           ,  0.534090932785],
                        [ 0.098484848485,  0.5           ,  0.556818158124],
                        [ 0.098484848485,  0.5           ,  0.602272735171],
                        [ 0.128787878788,  0.5           ,  0.556818158124],
                        [ 0.128787878788,  0.5           ,  0.602272735171],
                        [ 0.143939393939,  0.5           ,  0.579545446648],
                        [ 0.143939393939,  0.5           ,  0.625000023694],
                        [ 0.174242424242,  0.5           ,  0.579545446648],
                        [ 0.174242424242,  0.5           ,  0.625000023694],
                        [ 0.098484848485,  0.5           ,  0.647727249033],
                        [ 0.098484848485,  0.5           ,  0.69318182608 ],
                        [ 0.128787878788,  0.5           ,  0.647727249033],
                        [ 0.128787878788,  0.5           ,  0.69318182608 ],
                        [ 0.143939393939,  0.5           ,  0.670454537557],
                        [ 0.143939393939,  0.5           ,  0.715909114603],
                        [ 0.174242424242,  0.5           ,  0.670454537557],
                        [ 0.174242424242,  0.5           ,  0.715909114603],
                        [ 0.098484848485,  0.5           ,  0.738636339943],
                        [ 0.098484848485,  0.5           ,  0.784090916989],
                        [ 0.128787878788,  0.5           ,  0.738636339943],
                        [ 0.128787878788,  0.5           ,  0.784090916989],
                        [ 0.143939393939,  0.5           ,  0.761363628466],
                        [ 0.143939393939,  0.5           ,  0.806818205512],
                        [ 0.174242424242,  0.5           ,  0.761363628466],
                        [ 0.174242424242,  0.5           ,  0.806818205512],
                        [ 0.098484848485,  0.5           ,  0.829545430852],
                        [ 0.098484848485,  0.5           ,  0.875000007898],
                        [ 0.128787878788,  0.5           ,  0.829545430852],
                        [ 0.128787878788,  0.5           ,  0.875000007898],
                        [ 0.143939393939,  0.5           ,  0.852272719375],
                        [ 0.143939393939,  0.5           ,  0.897727296421],
                        [ 0.174242424242,  0.5           ,  0.852272719375],
                        [ 0.174242424242,  0.5           ,  0.897727296421],
                        [ 0.098484848485,  0.5           ,  0.920454521761],
                        [ 0.098484848485,  0.5           ,  0.965909098807],
                        [ 0.128787878788,  0.5           ,  0.920454521761],
                        [ 0.128787878788,  0.5           ,  0.965909098807],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.143939393939,  0.5           ,  0.943181810284],
                        [ 0.143939393939,  0.5           ,  0.98863638733 ],
                        [ 0.174242424242,  0.5           ,  0.943181810284],
                        [ 0.174242424242,  0.5           ,  0.98863638733 ],
                        [ 0.189393939394,  0.5           ,  0.01136361267 ],
                        [ 0.189393939394,  0.5           ,  0.056818189716],
                        [ 0.219696969697,  0.5           ,  0.01136361267 ],
                        [ 0.219696969697,  0.5           ,  0.056818189716],
                        [ 0.234848484848,  0.5           ,  0.034090901193],
                        [ 0.234848484848,  0.5           ,  0.079545478239],
                        [ 0.265151515152,  0.5           ,  0.034090901193],
                        [ 0.265151515152,  0.5           ,  0.079545478239],
                        [ 0.189393939394,  0.5           ,  0.102272703579],
                        [ 0.189393939394,  0.5           ,  0.147727280625],
                        [ 0.219696969697,  0.5           ,  0.102272703579],
                        [ 0.219696969697,  0.5           ,  0.147727280625],
                        [ 0.234848484848,  0.5           ,  0.124999992102],
                        [ 0.234848484848,  0.5           ,  0.170454569148],
                        [ 0.265151515152,  0.5           ,  0.124999992102],
                        [ 0.265151515152,  0.5           ,  0.170454569148],
                        [ 0.189393939394,  0.5           ,  0.193181794488],
                        [ 0.189393939394,  0.5           ,  0.238636371534],
                        [ 0.219696969697,  0.5           ,  0.193181794488],
                        [ 0.219696969697,  0.5           ,  0.238636371534],
                        [ 0.234848484848,  0.5           ,  0.215909083011],
                        [ 0.234848484848,  0.5           ,  0.261363660057],
                        [ 0.265151515152,  0.5           ,  0.215909083011],
                        [ 0.265151515152,  0.5           ,  0.261363660057],
                        [ 0.189393939394,  0.5           ,  0.284090885397],
                        [ 0.189393939394,  0.5           ,  0.329545462443],
                        [ 0.219696969697,  0.5           ,  0.284090885397],
                        [ 0.219696969697,  0.5           ,  0.329545462443],
                        [ 0.234848484848,  0.5           ,  0.30681817392 ],
                        [ 0.234848484848,  0.5           ,  0.352272750967],
                        [ 0.265151515152,  0.5           ,  0.30681817392 ],
                        [ 0.265151515152,  0.5           ,  0.352272750967],
                        [ 0.189393939394,  0.5           ,  0.374999976306],
                        [ 0.189393939394,  0.5           ,  0.420454553352],
                        [ 0.219696969697,  0.5           ,  0.374999976306],
                        [ 0.219696969697,  0.5           ,  0.420454553352],
                        [ 0.234848484848,  0.5           ,  0.397727264829],
                        [ 0.234848484848,  0.5           ,  0.443181841876],
                        [ 0.265151515152,  0.5           ,  0.397727264829],
                        [ 0.265151515152,  0.5           ,  0.443181841876],
                        [ 0.189393939394,  0.5           ,  0.465909067215],
                        [ 0.189393939394,  0.5           ,  0.511363644262],
                        [ 0.219696969697,  0.5           ,  0.465909067215],
                        [ 0.219696969697,  0.5           ,  0.511363644262],
                        [ 0.234848484848,  0.5           ,  0.488636355738],
                        [ 0.234848484848,  0.5           ,  0.534090932785],
                        [ 0.265151515152,  0.5           ,  0.488636355738],
                        [ 0.265151515152,  0.5           ,  0.534090932785],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.189393939394,  0.5           ,  0.556818158124],
                        [ 0.189393939394,  0.5           ,  0.602272735171],
                        [ 0.219696969697,  0.5           ,  0.556818158124],
                        [ 0.219696969697,  0.5           ,  0.602272735171],
                        [ 0.234848484848,  0.5           ,  0.579545446648],
                        [ 0.234848484848,  0.5           ,  0.625000023694],
                        [ 0.265151515152,  0.5           ,  0.579545446648],
                        [ 0.265151515152,  0.5           ,  0.625000023694],
                        [ 0.189393939394,  0.5           ,  0.647727249033],
                        [ 0.189393939394,  0.5           ,  0.69318182608 ],
                        [ 0.219696969697,  0.5           ,  0.647727249033],
                        [ 0.219696969697,  0.5           ,  0.69318182608 ],
                        [ 0.234848484848,  0.5           ,  0.670454537557],
                        [ 0.234848484848,  0.5           ,  0.715909114603],
                        [ 0.265151515152,  0.5           ,  0.670454537557],
                        [ 0.265151515152,  0.5           ,  0.715909114603],
                        [ 0.189393939394,  0.5           ,  0.738636339943],
                        [ 0.189393939394,  0.5           ,  0.784090916989],
                        [ 0.219696969697,  0.5           ,  0.738636339943],
                        [ 0.219696969697,  0.5           ,  0.784090916989],
                        [ 0.234848484848,  0.5           ,  0.761363628466],
                        [ 0.234848484848,  0.5           ,  0.806818205512],
                        [ 0.265151515152,  0.5           ,  0.761363628466],
                        [ 0.265151515152,  0.5           ,  0.806818205512],
                        [ 0.189393939394,  0.5           ,  0.829545430852],
                        [ 0.189393939394,  0.5           ,  0.875000007898],
                        [ 0.219696969697,  0.5           ,  0.829545430852],
                        [ 0.219696969697,  0.5           ,  0.875000007898],
                        [ 0.234848484848,  0.5           ,  0.852272719375],
                        [ 0.234848484848,  0.5           ,  0.897727296421],
                        [ 0.265151515152,  0.5           ,  0.852272719375],
                        [ 0.265151515152,  0.5           ,  0.897727296421],
                        [ 0.189393939394,  0.5           ,  0.920454521761],
                        [ 0.189393939394,  0.5           ,  0.965909098807],
                        [ 0.219696969697,  0.5           ,  0.920454521761],
                        [ 0.219696969697,  0.5           ,  0.965909098807],
                        [ 0.234848484848,  0.5           ,  0.943181810284],
                        [ 0.234848484848,  0.5           ,  0.98863638733 ],
                        [ 0.265151515152,  0.5           ,  0.943181810284],
                        [ 0.265151515152,  0.5           ,  0.98863638733 ],
                        [ 0.280303030303,  0.5           ,  0.01136361267 ],
                        [ 0.280303030303,  0.5           ,  0.056818189716],
                        [ 0.310606060606,  0.5           ,  0.01136361267 ],
                        [ 0.310606060606,  0.5           ,  0.056818189716],
                        [ 0.325757575758,  0.5           ,  0.034090901193],
                        [ 0.325757575758,  0.5           ,  0.079545478239],
                        [ 0.356060606061,  0.5           ,  0.034090901193],
                        [ 0.356060606061,  0.5           ,  0.079545478239],
                        [ 0.280303030303,  0.5           ,  0.102272703579],
                        [ 0.280303030303,  0.5           ,  0.147727280625],
                        [ 0.310606060606,  0.5           ,  0.102272703579],
                        [ 0.310606060606,  0.5           ,  0.147727280625],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.325757575758,  0.5           ,  0.124999992102],
                        [ 0.325757575758,  0.5           ,  0.170454569148],
                        [ 0.356060606061,  0.5           ,  0.124999992102],
                        [ 0.356060606061,  0.5           ,  0.170454569148],
                        [ 0.280303030303,  0.5           ,  0.193181794488],
                        [ 0.280303030303,  0.5           ,  0.238636371534],
                        [ 0.310606060606,  0.5           ,  0.193181794488],
                        [ 0.310606060606,  0.5           ,  0.238636371534],
                        [ 0.325757575758,  0.5           ,  0.215909083011],
                        [ 0.325757575758,  0.5           ,  0.261363660057],
                        [ 0.356060606061,  0.5           ,  0.215909083011],
                        [ 0.356060606061,  0.5           ,  0.261363660057],
                        [ 0.280303030303,  0.5           ,  0.284090885397],
                        [ 0.280303030303,  0.5           ,  0.329545462443],
                        [ 0.310606060606,  0.5           ,  0.284090885397],
                        [ 0.310606060606,  0.5           ,  0.329545462443],
                        [ 0.325757575758,  0.5           ,  0.30681817392 ],
                        [ 0.325757575758,  0.5           ,  0.352272750967],
                        [ 0.356060606061,  0.5           ,  0.30681817392 ],
                        [ 0.356060606061,  0.5           ,  0.352272750967],
                        [ 0.280303030303,  0.5           ,  0.374999976306],
                        [ 0.280303030303,  0.5           ,  0.420454553352],
                        [ 0.310606060606,  0.5           ,  0.374999976306],
                        [ 0.310606060606,  0.5           ,  0.420454553352],
                        [ 0.325757575758,  0.5           ,  0.397727264829],
                        [ 0.325757575758,  0.5           ,  0.443181841876],
                        [ 0.356060606061,  0.5           ,  0.397727264829],
                        [ 0.356213523227,  0.5           ,  0.443153499244],
                        [ 0.280303030303,  0.5           ,  0.465909067215],
                        [ 0.280303030303,  0.5           ,  0.511363644262],
                        [ 0.310606060606,  0.5           ,  0.465909067215],
                        [ 0.310606060606,  0.5           ,  0.511363644262],
                        [ 0.325757575758,  0.5           ,  0.488636355738],
                        [ 0.325757575758,  0.5           ,  0.534090932785],
                        [ 0.356628173158,  0.5           ,  0.488616788689],
                        [ 0.356628183572,  0.5           ,  0.534110475119],
                        [ 0.280303030303,  0.5           ,  0.556818158124],
                        [ 0.280303030303,  0.5           ,  0.602272735171],
                        [ 0.310606060606,  0.5           ,  0.556818158124],
                        [ 0.310606060606,  0.5           ,  0.602272735171],
                        [ 0.325757575758,  0.5           ,  0.579545446648],
                        [ 0.325757575758,  0.5           ,  0.625000023694],
                        [ 0.356213521049,  0.5           ,  0.579573760159],
                        [ 0.356060606061,  0.5           ,  0.625000023694],
                        [ 0.280303030303,  0.5           ,  0.647727249033],
                        [ 0.280303030303,  0.5           ,  0.69318182608 ],
                        [ 0.310606060606,  0.5           ,  0.647727249033],
                        [ 0.310606060606,  0.5           ,  0.69318182608 ],
                        [ 0.325757575758,  0.5           ,  0.670454537557],
                        [ 0.325757575758,  0.5           ,  0.715909114603],
                        [ 0.356060606061,  0.5           ,  0.670454537557],
                        [ 0.356060606061,  0.5           ,  0.715909114603],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.280303030303,  0.5           ,  0.738636339943],
                        [ 0.280303030303,  0.5           ,  0.784090916989],
                        [ 0.310606060606,  0.5           ,  0.738636339943],
                        [ 0.310606060606,  0.5           ,  0.784090916989],
                        [ 0.325757575758,  0.5           ,  0.761363628466],
                        [ 0.325757575758,  0.5           ,  0.806818205512],
                        [ 0.356060606061,  0.5           ,  0.761363628466],
                        [ 0.356060606061,  0.5           ,  0.806818205512],
                        [ 0.280303030303,  0.5           ,  0.829545430852],
                        [ 0.280303030303,  0.5           ,  0.875000007898],
                        [ 0.310606060606,  0.5           ,  0.829545430852],
                        [ 0.310606060606,  0.5           ,  0.875000007898],
                        [ 0.325757575758,  0.5           ,  0.852272719375],
                        [ 0.325757575758,  0.5           ,  0.897727296421],
                        [ 0.356060606061,  0.5           ,  0.852272719375],
                        [ 0.356060606061,  0.5           ,  0.897727296421],
                        [ 0.280303030303,  0.5           ,  0.920454521761],
                        [ 0.280303030303,  0.5           ,  0.965909098807],
                        [ 0.310606060606,  0.5           ,  0.920454521761],
                        [ 0.310606060606,  0.5           ,  0.965909098807],
                        [ 0.325757575758,  0.5           ,  0.943181810284],
                        [ 0.325757575758,  0.5           ,  0.98863638733 ],
                        [ 0.356060606061,  0.5           ,  0.943181810284],
                        [ 0.356060606061,  0.5           ,  0.98863638733 ],
                        [ 0.371212121212,  0.5           ,  0.01136361267 ],
                        [ 0.371212121212,  0.5           ,  0.056818189716],
                        [ 0.401515151515,  0.5           ,  0.01136361267 ],
                        [ 0.401515151515,  0.5           ,  0.056818189716],
                        [ 0.416666666667,  0.5           ,  0.034090901193],
                        [ 0.416666666667,  0.5           ,  0.079545478239],
                        [ 0.44696969697 ,  0.5           ,  0.034090901193],
                        [ 0.44696969697 ,  0.5           ,  0.079545478239],
                        [ 0.371212121212,  0.5           ,  0.102272703579],
                        [ 0.371212121212,  0.5           ,  0.147727280625],
                        [ 0.401515151515,  0.5           ,  0.102272703579],
                        [ 0.401515151515,  0.5           ,  0.147727280625],
                        [ 0.416666666667,  0.5           ,  0.124999992102],
                        [ 0.416666666667,  0.5           ,  0.170454569148],
                        [ 0.44696969697 ,  0.5           ,  0.124999992102],
                        [ 0.44696969697 ,  0.5           ,  0.170454569148],
                        [ 0.371212121212,  0.5           ,  0.193181794488],
                        [ 0.371212121212,  0.5           ,  0.238636371534],
                        [ 0.401515151515,  0.5           ,  0.193181794488],
                        [ 0.401515151515,  0.5           ,  0.238636371534],
                        [ 0.416666666667,  0.5           ,  0.215909083011],
                        [ 0.416666666667,  0.5           ,  0.261363660057],
                        [ 0.44696969697 ,  0.5           ,  0.215909083011],
                        [ 0.44696969697 ,  0.5           ,  0.261363660057],
                        [ 0.371212121212,  0.5           ,  0.284090885397],
                        [ 0.371212121212,  0.5           ,  0.329545462443],
                        [ 0.401515151515,  0.5           ,  0.284090885397],
                        [ 0.401515151515,  0.5           ,  0.329545462443],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.416666666667,  0.5           ,  0.30681817392 ],
                        [ 0.416666666667,  0.5           ,  0.352272750967],
                        [ 0.44696969697 ,  0.5           ,  0.30681817392 ],
                        [ 0.44696969697 ,  0.5           ,  0.352272750967],
                        [ 0.371212121212,  0.5           ,  0.374999976306],
                        [ 0.371240359473,  0.5           ,  0.420375651895],
                        [ 0.401515151515,  0.5           ,  0.374999976306],
                        [ 0.40147819959 ,  0.5           ,  0.420234171319],
                        [ 0.416666666667,  0.5           ,  0.397727264829],
                        [ 0.416830246448,  0.5           ,  0.442472684368],
                        [ 0.44696969697 ,  0.5           ,  0.397727264829],
                        [ 0.446964365225,  0.5           ,  0.441630846122],
                        [ 0.371664378445,  0.5           ,  0.465870080888],
                        [ 0.372180228933,  0.5           ,  0.511363646197],
                        [ 0.402227804715,  0.5           ,  0.465570661071],
                        [ 0.403476107656,  0.5           ,  0.511363626191],
                        [ 0.418393892774,  0.5           ,  0.488180891795],
                        [ 0.418393900636,  0.5           ,  0.534546403019],
                        [ 0.449724972222,  0.5           ,  0.486037437389],
                        [ 0.449724958085,  0.5           ,  0.53668982691 ],
                        [ 0.371664358597,  0.5           ,  0.556857273536],
                        [ 0.371240334711,  0.5           ,  0.602351706327],
                        [ 0.402227842037,  0.5           ,  0.557156539186],
                        [ 0.401478190901,  0.5           ,  0.602493019943],
                        [ 0.416830209076,  0.5           ,  0.580254791521],
                        [ 0.416666666667,  0.5           ,  0.625000023694],
                        [ 0.446964368584,  0.5           ,  0.581096211981],
                        [ 0.44696969697 ,  0.5           ,  0.625000023694],
                        [ 0.371212121212,  0.5           ,  0.647727249033],
                        [ 0.371212121212,  0.5           ,  0.69318182608 ],
                        [ 0.401515151515,  0.5           ,  0.647727249033],
                        [ 0.401515151515,  0.5           ,  0.69318182608 ],
                        [ 0.416666666667,  0.5           ,  0.670454537557],
                        [ 0.416666666667,  0.5           ,  0.715909114603],
                        [ 0.44696969697 ,  0.5           ,  0.670454537557],
                        [ 0.44696969697 ,  0.5           ,  0.715909114603],
                        [ 0.371212121212,  0.5           ,  0.738636339943],
                        [ 0.371212121212,  0.5           ,  0.784090916989],
                        [ 0.401515151515,  0.5           ,  0.738636339943],
                        [ 0.401515151515,  0.5           ,  0.784090916989],
                        [ 0.416666666667,  0.5           ,  0.761363628466],
                        [ 0.416666666667,  0.5           ,  0.806818205512],
                        [ 0.44696969697 ,  0.5           ,  0.761363628466],
                        [ 0.44696969697 ,  0.5           ,  0.806818205512],
                        [ 0.371212121212,  0.5           ,  0.829545430852],
                        [ 0.371212121212,  0.5           ,  0.875000007898],
                        [ 0.401515151515,  0.5           ,  0.829545430852],
                        [ 0.401515151515,  0.5           ,  0.875000007898],
                        [ 0.416666666667,  0.5           ,  0.852272719375],
                        [ 0.416666666667,  0.5           ,  0.897727296421],
                        [ 0.44696969697 ,  0.5           ,  0.852272719375],
                        [ 0.44696969697 ,  0.5           ,  0.897727296421],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.371212121212,  0.5           ,  0.920454521761],
                        [ 0.371212121212,  0.5           ,  0.965909098807],
                        [ 0.401515151515,  0.5           ,  0.920454521761],
                        [ 0.401515151515,  0.5           ,  0.965909098807],
                        [ 0.416666666667,  0.5           ,  0.943181810284],
                        [ 0.416666666667,  0.5           ,  0.98863638733 ],
                        [ 0.44696969697 ,  0.5           ,  0.943181810284],
                        [ 0.44696969697 ,  0.5           ,  0.98863638733 ],
                        [ 0.462121212121,  0.5           ,  0.01136361267 ],
                        [ 0.462121212121,  0.5           ,  0.056818189716],
                        [ 0.492424242424,  0.5           ,  0.01136361267 ],
                        [ 0.492424242424,  0.5           ,  0.056818189716],
                        [ 0.507575757576,  0.5           ,  0.034090901193],
                        [ 0.507575757576,  0.5           ,  0.079545478239],
                        [ 0.537878787879,  0.5           ,  0.034090901193],
                        [ 0.537878787879,  0.5           ,  0.079545478239],
                        [ 0.462121212121,  0.5           ,  0.102272703579],
                        [ 0.462121212121,  0.5           ,  0.147727280625],
                        [ 0.492424242424,  0.5           ,  0.102272703579],
                        [ 0.492424242424,  0.5           ,  0.147727280625],
                        [ 0.507575757576,  0.5           ,  0.124999992102],
                        [ 0.507575757576,  0.5           ,  0.170454569148],
                        [ 0.537878787879,  0.5           ,  0.124999992102],
                        [ 0.537878787879,  0.5           ,  0.170454569148],
                        [ 0.462121212121,  0.5           ,  0.193181794488],
                        [ 0.462121212121,  0.5           ,  0.238636371534],
                        [ 0.492424242424,  0.5           ,  0.193181794488],
                        [ 0.492424242424,  0.5           ,  0.238636371534],
                        [ 0.507575757576,  0.5           ,  0.215909083011],
                        [ 0.507575757576,  0.5           ,  0.261363660057],
                        [ 0.537878787879,  0.5           ,  0.215909083011],
                        [ 0.537878787879,  0.5           ,  0.261363660057],
                        [ 0.462121212121,  0.5           ,  0.284090885397],
                        [ 0.462121212121,  0.5           ,  0.329545462443],
                        [ 0.492424242424,  0.5           ,  0.284090885397],
                        [ 0.492424242424,  0.5           ,  0.329545462443],
                        [ 0.507575757576,  0.5           ,  0.30681817392 ],
                        [ 0.507575757576,  0.5           ,  0.352272750967],
                        [ 0.537878787879,  0.5           ,  0.30681817392 ],
                        [ 0.537878787879,  0.5           ,  0.352272750967],
                        [ 0.462121212121,  0.5           ,  0.374999976306],
                        [ 0.461982668182,  0.5           ,  0.419551128898],
                        [ 0.492424242424,  0.5           ,  0.374999976306],
                        [ 0.492562786363,  0.5           ,  0.419551128897],
                        [ 0.507575757576,  0.5           ,  0.397727264829],
                        [ 0.507581089321,  0.5           ,  0.441630846122],
                        [ 0.537878787879,  0.5           ,  0.397727264829],
                        [ 0.537715208097,  0.5           ,  0.442472684368],
                        [ 0.462428771889,  0.5           ,  0.462736595679],
                        [ 0.477272727273,  0.5           ,  0.523369067868],
                        [ 0.492116682656,  0.5           ,  0.462736595679],
                        [ 0.477272727273,  0.5           ,  0.499358217002],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.504820482324,  0.5           ,  0.486037437389],
                        [ 0.50482049646 ,  0.5           ,  0.53668982691 ],
                        [ 0.536151561771,  0.5           ,  0.488180891795],
                        [ 0.536151553909,  0.5           ,  0.534546403019],
                        [ 0.462428837995,  0.5           ,  0.559990799958],
                        [ 0.461982628074,  0.5           ,  0.603176222834],
                        [ 0.492116616551,  0.5           ,  0.559990799958],
                        [ 0.492562826472,  0.5           ,  0.603176222834],
                        [ 0.507581085961,  0.5           ,  0.581096211981],
                        [ 0.507575757576,  0.5           ,  0.625000023694],
                        [ 0.53771524547 ,  0.5           ,  0.580254791521],
                        [ 0.537878787879,  0.5           ,  0.625000023694],
                        [ 0.462121212121,  0.5           ,  0.647727249033],
                        [ 0.462121212121,  0.5           ,  0.69318182608 ],
                        [ 0.492424242424,  0.5           ,  0.647727249033],
                        [ 0.492424242424,  0.5           ,  0.69318182608 ],
                        [ 0.507575757576,  0.5           ,  0.670454537557],
                        [ 0.507575757576,  0.5           ,  0.715909114603],
                        [ 0.537878787879,  0.5           ,  0.670454537557],
                        [ 0.537878787879,  0.5           ,  0.715909114603],
                        [ 0.462121212121,  0.5           ,  0.738636339943],
                        [ 0.462121212121,  0.5           ,  0.784090916989],
                        [ 0.492424242424,  0.5           ,  0.738636339943],
                        [ 0.492424242424,  0.5           ,  0.784090916989],
                        [ 0.507575757576,  0.5           ,  0.761363628466],
                        [ 0.507575757576,  0.5           ,  0.806818205512],
                        [ 0.537878787879,  0.5           ,  0.761363628466],
                        [ 0.537878787879,  0.5           ,  0.806818205512],
                        [ 0.462121212121,  0.5           ,  0.829545430852],
                        [ 0.462121212121,  0.5           ,  0.875000007898],
                        [ 0.492424242424,  0.5           ,  0.829545430852],
                        [ 0.492424242424,  0.5           ,  0.875000007898],
                        [ 0.507575757576,  0.5           ,  0.852272719375],
                        [ 0.507575757576,  0.5           ,  0.897727296421],
                        [ 0.537878787879,  0.5           ,  0.852272719375],
                        [ 0.537878787879,  0.5           ,  0.897727296421],
                        [ 0.462121212121,  0.5           ,  0.920454521761],
                        [ 0.462121212121,  0.5           ,  0.965909098807],
                        [ 0.492424242424,  0.5           ,  0.920454521761],
                        [ 0.492424242424,  0.5           ,  0.965909098807],
                        [ 0.507575757576,  0.5           ,  0.943181810284],
                        [ 0.507575757576,  0.5           ,  0.98863638733 ],
                        [ 0.537878787879,  0.5           ,  0.943181810284],
                        [ 0.537878787879,  0.5           ,  0.98863638733 ],
                        [ 0.55303030303 ,  0.5           ,  0.01136361267 ],
                        [ 0.55303030303 ,  0.5           ,  0.056818189716],
                        [ 0.583333333333,  0.5           ,  0.01136361267 ],
                        [ 0.583333333333,  0.5           ,  0.056818189716],
                        [ 0.598484848485,  0.5           ,  0.034090901193],
                        [ 0.598484848485,  0.5           ,  0.079545478239],
                        [ 0.628787878788,  0.5           ,  0.034090901193],
                        [ 0.628787878788,  0.5           ,  0.079545478239],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.55303030303 ,  0.5           ,  0.102272703579],
                        [ 0.55303030303 ,  0.5           ,  0.147727280625],
                        [ 0.583333333333,  0.5           ,  0.102272703579],
                        [ 0.583333333333,  0.5           ,  0.147727280625],
                        [ 0.598484848485,  0.5           ,  0.124999992102],
                        [ 0.598484848485,  0.5           ,  0.170454569148],
                        [ 0.628787878788,  0.5           ,  0.124999992102],
                        [ 0.628787878788,  0.5           ,  0.170454569148],
                        [ 0.55303030303 ,  0.5           ,  0.193181794488],
                        [ 0.55303030303 ,  0.5           ,  0.238636371534],
                        [ 0.583333333333,  0.5           ,  0.193181794488],
                        [ 0.583333333333,  0.5           ,  0.238636371534],
                        [ 0.598484848485,  0.5           ,  0.215909083011],
                        [ 0.598484848485,  0.5           ,  0.261363660057],
                        [ 0.628787878788,  0.5           ,  0.215909083011],
                        [ 0.628787878788,  0.5           ,  0.261363660057],
                        [ 0.55303030303 ,  0.5           ,  0.284090885397],
                        [ 0.55303030303 ,  0.5           ,  0.329545462443],
                        [ 0.583333333333,  0.5           ,  0.284090885397],
                        [ 0.583333333333,  0.5           ,  0.329545462443],
                        [ 0.598484848485,  0.5           ,  0.30681817392 ],
                        [ 0.598484848485,  0.5           ,  0.352272750967],
                        [ 0.628787878788,  0.5           ,  0.30681817392 ],
                        [ 0.628787878788,  0.5           ,  0.352272750967],
                        [ 0.55303030303 ,  0.5           ,  0.374999976306],
                        [ 0.553067254955,  0.5           ,  0.420234171319],
                        [ 0.583333333333,  0.5           ,  0.374999976306],
                        [ 0.583305095072,  0.5           ,  0.420375651895],
                        [ 0.598484848485,  0.5           ,  0.397727264829],
                        [ 0.598331931318,  0.5           ,  0.443153499244],
                        [ 0.628787878788,  0.5           ,  0.397727264829],
                        [ 0.628787878788,  0.5           ,  0.443181841876],
                        [ 0.552317649831,  0.5           ,  0.465570661071],
                        [ 0.551069346889,  0.5           ,  0.511363626191],
                        [ 0.582881076101,  0.5           ,  0.465870080888],
                        [ 0.582365225613,  0.5           ,  0.511363646197],
                        [ 0.597917281387,  0.5           ,  0.488616788689],
                        [ 0.597917270974,  0.5           ,  0.534110475119],
                        [ 0.628787878788,  0.5           ,  0.488636355738],
                        [ 0.628787878788,  0.5           ,  0.534090932785],
                        [ 0.552317612509,  0.5           ,  0.557156539186],
                        [ 0.553067263645,  0.5           ,  0.602493019943],
                        [ 0.582881095949,  0.5           ,  0.556857273536],
                        [ 0.583305119835,  0.5           ,  0.602351706327],
                        [ 0.598331933496,  0.5           ,  0.579573760159],
                        [ 0.598484848485,  0.5           ,  0.625000023694],
                        [ 0.628787878788,  0.5           ,  0.579545446648],
                        [ 0.628787878788,  0.5           ,  0.625000023694],
                        [ 0.55303030303 ,  0.5           ,  0.647727249033],
                        [ 0.55303030303 ,  0.5           ,  0.69318182608 ],
                        [ 0.583333333333,  0.5           ,  0.647727249033],
                        [ 0.583333333333,  0.5           ,  0.69318182608 ],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.598484848485,  0.5           ,  0.670454537557],
                        [ 0.598484848485,  0.5           ,  0.715909114603],
                        [ 0.628787878788,  0.5           ,  0.670454537557],
                        [ 0.628787878788,  0.5           ,  0.715909114603],
                        [ 0.55303030303 ,  0.5           ,  0.738636339943],
                        [ 0.55303030303 ,  0.5           ,  0.784090916989],
                        [ 0.583333333333,  0.5           ,  0.738636339943],
                        [ 0.583333333333,  0.5           ,  0.784090916989],
                        [ 0.598484848485,  0.5           ,  0.761363628466],
                        [ 0.598484848485,  0.5           ,  0.806818205512],
                        [ 0.628787878788,  0.5           ,  0.761363628466],
                        [ 0.628787878788,  0.5           ,  0.806818205512],
                        [ 0.55303030303 ,  0.5           ,  0.829545430852],
                        [ 0.55303030303 ,  0.5           ,  0.875000007898],
                        [ 0.583333333333,  0.5           ,  0.829545430852],
                        [ 0.583333333333,  0.5           ,  0.875000007898],
                        [ 0.598484848485,  0.5           ,  0.852272719375],
                        [ 0.598484848485,  0.5           ,  0.897727296421],
                        [ 0.628787878788,  0.5           ,  0.852272719375],
                        [ 0.628787878788,  0.5           ,  0.897727296421],
                        [ 0.55303030303 ,  0.5           ,  0.920454521761],
                        [ 0.55303030303 ,  0.5           ,  0.965909098807],
                        [ 0.583333333333,  0.5           ,  0.920454521761],
                        [ 0.583333333333,  0.5           ,  0.965909098807],
                        [ 0.598484848485,  0.5           ,  0.943181810284],
                        [ 0.598484848485,  0.5           ,  0.98863638733 ],
                        [ 0.628787878788,  0.5           ,  0.943181810284],
                        [ 0.628787878788,  0.5           ,  0.98863638733 ],
                        [ 0.643939393939,  0.5           ,  0.01136361267 ],
                        [ 0.643939393939,  0.5           ,  0.056818189716],
                        [ 0.674242424242,  0.5           ,  0.01136361267 ],
                        [ 0.674242424242,  0.5           ,  0.056818189716],
                        [ 0.689393939394,  0.5           ,  0.034090901193],
                        [ 0.689393939394,  0.5           ,  0.079545478239],
                        [ 0.719696969697,  0.5           ,  0.034090901193],
                        [ 0.719696969697,  0.5           ,  0.079545478239],
                        [ 0.643939393939,  0.5           ,  0.102272703579],
                        [ 0.643939393939,  0.5           ,  0.147727280625],
                        [ 0.674242424242,  0.5           ,  0.102272703579],
                        [ 0.674242424242,  0.5           ,  0.147727280625],
                        [ 0.689393939394,  0.5           ,  0.124999992102],
                        [ 0.689393939394,  0.5           ,  0.170454569148],
                        [ 0.719696969697,  0.5           ,  0.124999992102],
                        [ 0.719696969697,  0.5           ,  0.170454569148],
                        [ 0.643939393939,  0.5           ,  0.193181794488],
                        [ 0.643939393939,  0.5           ,  0.238636371534],
                        [ 0.674242424242,  0.5           ,  0.193181794488],
                        [ 0.674242424242,  0.5           ,  0.238636371534],
                        [ 0.689393939394,  0.5           ,  0.215909083011],
                        [ 0.689393939394,  0.5           ,  0.261363660057],
                        [ 0.719696969697,  0.5           ,  0.215909083011],
                        [ 0.719696969697,  0.5           ,  0.261363660057],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.643939393939,  0.5           ,  0.284090885397],
                        [ 0.643939393939,  0.5           ,  0.329545462443],
                        [ 0.674242424242,  0.5           ,  0.284090885397],
                        [ 0.674242424242,  0.5           ,  0.329545462443],
                        [ 0.689393939394,  0.5           ,  0.30681817392 ],
                        [ 0.689393939394,  0.5           ,  0.352272750967],
                        [ 0.719696969697,  0.5           ,  0.30681817392 ],
                        [ 0.719696969697,  0.5           ,  0.352272750967],
                        [ 0.643939393939,  0.5           ,  0.374999976306],
                        [ 0.643939393939,  0.5           ,  0.420454553352],
                        [ 0.674242424242,  0.5           ,  0.374999976306],
                        [ 0.674242424242,  0.5           ,  0.420454553352],
                        [ 0.689393939394,  0.5           ,  0.397727264829],
                        [ 0.689393939394,  0.5           ,  0.443181841876],
                        [ 0.719696969697,  0.5           ,  0.397727264829],
                        [ 0.719696969697,  0.5           ,  0.443181841876],
                        [ 0.643939393939,  0.5           ,  0.465909067215],
                        [ 0.643939393939,  0.5           ,  0.511363644262],
                        [ 0.674242424242,  0.5           ,  0.465909067215],
                        [ 0.674242424242,  0.5           ,  0.511363644262],
                        [ 0.689393939394,  0.5           ,  0.488636355738],
                        [ 0.689393939394,  0.5           ,  0.534090932785],
                        [ 0.719696969697,  0.5           ,  0.488636355738],
                        [ 0.719696969697,  0.5           ,  0.534090932785],
                        [ 0.643939393939,  0.5           ,  0.556818158124],
                        [ 0.643939393939,  0.5           ,  0.602272735171],
                        [ 0.674242424242,  0.5           ,  0.556818158124],
                        [ 0.674242424242,  0.5           ,  0.602272735171],
                        [ 0.689393939394,  0.5           ,  0.579545446648],
                        [ 0.689393939394,  0.5           ,  0.625000023694],
                        [ 0.719696969697,  0.5           ,  0.579545446648],
                        [ 0.719696969697,  0.5           ,  0.625000023694],
                        [ 0.643939393939,  0.5           ,  0.647727249033],
                        [ 0.643939393939,  0.5           ,  0.69318182608 ],
                        [ 0.674242424242,  0.5           ,  0.647727249033],
                        [ 0.674242424242,  0.5           ,  0.69318182608 ],
                        [ 0.689393939394,  0.5           ,  0.670454537557],
                        [ 0.689393939394,  0.5           ,  0.715909114603],
                        [ 0.719696969697,  0.5           ,  0.670454537557],
                        [ 0.719696969697,  0.5           ,  0.715909114603],
                        [ 0.643939393939,  0.5           ,  0.738636339943],
                        [ 0.643939393939,  0.5           ,  0.784090916989],
                        [ 0.674242424242,  0.5           ,  0.738636339943],
                        [ 0.674242424242,  0.5           ,  0.784090916989],
                        [ 0.689393939394,  0.5           ,  0.761363628466],
                        [ 0.689393939394,  0.5           ,  0.806818205512],
                        [ 0.719696969697,  0.5           ,  0.761363628466],
                        [ 0.719696969697,  0.5           ,  0.806818205512],
                        [ 0.643939393939,  0.5           ,  0.829545430852],
                        [ 0.643939393939,  0.5           ,  0.875000007898],
                        [ 0.674242424242,  0.5           ,  0.829545430852],
                        [ 0.674242424242,  0.5           ,  0.875000007898],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.689393939394,  0.5           ,  0.852272719375],
                        [ 0.689393939394,  0.5           ,  0.897727296421],
                        [ 0.719696969697,  0.5           ,  0.852272719375],
                        [ 0.719696969697,  0.5           ,  0.897727296421],
                        [ 0.643939393939,  0.5           ,  0.920454521761],
                        [ 0.643939393939,  0.5           ,  0.965909098807],
                        [ 0.674242424242,  0.5           ,  0.920454521761],
                        [ 0.674242424242,  0.5           ,  0.965909098807],
                        [ 0.689393939394,  0.5           ,  0.943181810284],
                        [ 0.689393939394,  0.5           ,  0.98863638733 ],
                        [ 0.719696969697,  0.5           ,  0.943181810284],
                        [ 0.719696969697,  0.5           ,  0.98863638733 ],
                        [ 0.734848484848,  0.5           ,  0.01136361267 ],
                        [ 0.734848484848,  0.5           ,  0.056818189716],
                        [ 0.765151515152,  0.5           ,  0.01136361267 ],
                        [ 0.765151515152,  0.5           ,  0.056818189716],
                        [ 0.780303030303,  0.5           ,  0.034090901193],
                        [ 0.780303030303,  0.5           ,  0.079545478239],
                        [ 0.810606060606,  0.5           ,  0.034090901193],
                        [ 0.810606060606,  0.5           ,  0.079545478239],
                        [ 0.734848484848,  0.5           ,  0.102272703579],
                        [ 0.734848484848,  0.5           ,  0.147727280625],
                        [ 0.765151515152,  0.5           ,  0.102272703579],
                        [ 0.765151515152,  0.5           ,  0.147727280625],
                        [ 0.780303030303,  0.5           ,  0.124999992102],
                        [ 0.780303030303,  0.5           ,  0.170454569148],
                        [ 0.810606060606,  0.5           ,  0.124999992102],
                        [ 0.810606060606,  0.5           ,  0.170454569148],
                        [ 0.734848484848,  0.5           ,  0.193181794488],
                        [ 0.734848484848,  0.5           ,  0.238636371534],
                        [ 0.765151515152,  0.5           ,  0.193181794488],
                        [ 0.765151515152,  0.5           ,  0.238636371534],
                        [ 0.780303030303,  0.5           ,  0.215909083011],
                        [ 0.780303030303,  0.5           ,  0.261363660057],
                        [ 0.810606060606,  0.5           ,  0.215909083011],
                        [ 0.810606060606,  0.5           ,  0.261363660057],
                        [ 0.734848484848,  0.5           ,  0.284090885397],
                        [ 0.734848484848,  0.5           ,  0.329545462443],
                        [ 0.765151515152,  0.5           ,  0.284090885397],
                        [ 0.765151515152,  0.5           ,  0.329545462443],
                        [ 0.780303030303,  0.5           ,  0.30681817392 ],
                        [ 0.780303030303,  0.5           ,  0.352272750967],
                        [ 0.810606060606,  0.5           ,  0.30681817392 ],
                        [ 0.810606060606,  0.5           ,  0.352272750967],
                        [ 0.734848484848,  0.5           ,  0.374999976306],
                        [ 0.734848484848,  0.5           ,  0.420454553352],
                        [ 0.765151515152,  0.5           ,  0.374999976306],
                        [ 0.765151515152,  0.5           ,  0.420454553352],
                        [ 0.780303030303,  0.5           ,  0.397727264829],
                        [ 0.780303030303,  0.5           ,  0.443181841876],
                        [ 0.810606060606,  0.5           ,  0.397727264829],
                        [ 0.810606060606,  0.5           ,  0.443181841876],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.734848484848,  0.5           ,  0.465909067215],
                        [ 0.734848484848,  0.5           ,  0.511363644262],
                        [ 0.765151515152,  0.5           ,  0.465909067215],
                        [ 0.765151515152,  0.5           ,  0.511363644262],
                        [ 0.780303030303,  0.5           ,  0.488636355738],
                        [ 0.780303030303,  0.5           ,  0.534090932785],
                        [ 0.810606060606,  0.5           ,  0.488636355738],
                        [ 0.810606060606,  0.5           ,  0.534090932785],
                        [ 0.734848484848,  0.5           ,  0.556818158124],
                        [ 0.734848484848,  0.5           ,  0.602272735171],
                        [ 0.765151515152,  0.5           ,  0.556818158124],
                        [ 0.765151515152,  0.5           ,  0.602272735171],
                        [ 0.780303030303,  0.5           ,  0.579545446648],
                        [ 0.780303030303,  0.5           ,  0.625000023694],
                        [ 0.810606060606,  0.5           ,  0.579545446648],
                        [ 0.810606060606,  0.5           ,  0.625000023694],
                        [ 0.734848484848,  0.5           ,  0.647727249033],
                        [ 0.734848484848,  0.5           ,  0.69318182608 ],
                        [ 0.765151515152,  0.5           ,  0.647727249033],
                        [ 0.765151515152,  0.5           ,  0.69318182608 ],
                        [ 0.780303030303,  0.5           ,  0.670454537557],
                        [ 0.780303030303,  0.5           ,  0.715909114603],
                        [ 0.810606060606,  0.5           ,  0.670454537557],
                        [ 0.810606060606,  0.5           ,  0.715909114603],
                        [ 0.734848484848,  0.5           ,  0.738636339943],
                        [ 0.734848484848,  0.5           ,  0.784090916989],
                        [ 0.765151515152,  0.5           ,  0.738636339943],
                        [ 0.765151515152,  0.5           ,  0.784090916989],
                        [ 0.780303030303,  0.5           ,  0.761363628466],
                        [ 0.780303030303,  0.5           ,  0.806818205512],
                        [ 0.810606060606,  0.5           ,  0.761363628466],
                        [ 0.810606060606,  0.5           ,  0.806818205512],
                        [ 0.734848484848,  0.5           ,  0.829545430852],
                        [ 0.734848484848,  0.5           ,  0.875000007898],
                        [ 0.765151515152,  0.5           ,  0.829545430852],
                        [ 0.765151515152,  0.5           ,  0.875000007898],
                        [ 0.780303030303,  0.5           ,  0.852272719375],
                        [ 0.780303030303,  0.5           ,  0.897727296421],
                        [ 0.810606060606,  0.5           ,  0.852272719375],
                        [ 0.810606060606,  0.5           ,  0.897727296421],
                        [ 0.734848484848,  0.5           ,  0.920454521761],
                        [ 0.734848484848,  0.5           ,  0.965909098807],
                        [ 0.765151515152,  0.5           ,  0.920454521761],
                        [ 0.765151515152,  0.5           ,  0.965909098807],
                        [ 0.780303030303,  0.5           ,  0.943181810284],
                        [ 0.780303030303,  0.5           ,  0.98863638733 ],
                        [ 0.810606060606,  0.5           ,  0.943181810284],
                        [ 0.810606060606,  0.5           ,  0.98863638733 ],
                        [ 0.825757575758,  0.5           ,  0.01136361267 ],
                        [ 0.825757575758,  0.5           ,  0.056818189716],
                        [ 0.856060606061,  0.5           ,  0.01136361267 ],
                        [ 0.856060606061,  0.5           ,  0.056818189716],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.871212121212,  0.5           ,  0.034090901193],
                        [ 0.871212121212,  0.5           ,  0.079545478239],
                        [ 0.901515151515,  0.5           ,  0.034090901193],
                        [ 0.901515151515,  0.5           ,  0.079545478239],
                        [ 0.825757575758,  0.5           ,  0.102272703579],
                        [ 0.825757575758,  0.5           ,  0.147727280625],
                        [ 0.856060606061,  0.5           ,  0.102272703579],
                        [ 0.856060606061,  0.5           ,  0.147727280625],
                        [ 0.871212121212,  0.5           ,  0.124999992102],
                        [ 0.871212121212,  0.5           ,  0.170454569148],
                        [ 0.901515151515,  0.5           ,  0.124999992102],
                        [ 0.901515151515,  0.5           ,  0.170454569148],
                        [ 0.825757575758,  0.5           ,  0.193181794488],
                        [ 0.825757575758,  0.5           ,  0.238636371534],
                        [ 0.856060606061,  0.5           ,  0.193181794488],
                        [ 0.856060606061,  0.5           ,  0.238636371534],
                        [ 0.871212121212,  0.5           ,  0.215909083011],
                        [ 0.871212121212,  0.5           ,  0.261363660057],
                        [ 0.901515151515,  0.5           ,  0.215909083011],
                        [ 0.901515151515,  0.5           ,  0.261363660057],
                        [ 0.825757575758,  0.5           ,  0.284090885397],
                        [ 0.825757575758,  0.5           ,  0.329545462443],
                        [ 0.856060606061,  0.5           ,  0.284090885397],
                        [ 0.856060606061,  0.5           ,  0.329545462443],
                        [ 0.871212121212,  0.5           ,  0.30681817392 ],
                        [ 0.871212121212,  0.5           ,  0.352272750967],
                        [ 0.901515151515,  0.5           ,  0.30681817392 ],
                        [ 0.901515151515,  0.5           ,  0.352272750967],
                        [ 0.825757575758,  0.5           ,  0.374999976306],
                        [ 0.825757575758,  0.5           ,  0.420454553352],
                        [ 0.856060606061,  0.5           ,  0.374999976306],
                        [ 0.856060606061,  0.5           ,  0.420454553352],
                        [ 0.871212121212,  0.5           ,  0.397727264829],
                        [ 0.871212121212,  0.5           ,  0.443181841876],
                        [ 0.901515151515,  0.5           ,  0.397727264829],
                        [ 0.901515151515,  0.5           ,  0.443181841876],
                        [ 0.825757575758,  0.5           ,  0.465909067215],
                        [ 0.825757575758,  0.5           ,  0.511363644262],
                        [ 0.856060606061,  0.5           ,  0.465909067215],
                        [ 0.856060606061,  0.5           ,  0.511363644262],
                        [ 0.871212121212,  0.5           ,  0.488636355738],
                        [ 0.871212121212,  0.5           ,  0.534090932785],
                        [ 0.901515151515,  0.5           ,  0.488636355738],
                        [ 0.901515151515,  0.5           ,  0.534090932785],
                        [ 0.825757575758,  0.5           ,  0.556818158124],
                        [ 0.825757575758,  0.5           ,  0.602272735171],
                        [ 0.856060606061,  0.5           ,  0.556818158124],
                        [ 0.856060606061,  0.5           ,  0.602272735171],
                        [ 0.871212121212,  0.5           ,  0.579545446648],
                        [ 0.871212121212,  0.5           ,  0.625000023694],
                        [ 0.901515151515,  0.5           ,  0.579545446648],
                        [ 0.901515151515,  0.5           ,  0.625000023694],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.825757575758,  0.5           ,  0.647727249033],
                        [ 0.825757575758,  0.5           ,  0.69318182608 ],
                        [ 0.856060606061,  0.5           ,  0.647727249033],
                        [ 0.856060606061,  0.5           ,  0.69318182608 ],
                        [ 0.871212121212,  0.5           ,  0.670454537557],
                        [ 0.871212121212,  0.5           ,  0.715909114603],
                        [ 0.901515151515,  0.5           ,  0.670454537557],
                        [ 0.901515151515,  0.5           ,  0.715909114603],
                        [ 0.825757575758,  0.5           ,  0.738636339943],
                        [ 0.825757575758,  0.5           ,  0.784090916989],
                        [ 0.856060606061,  0.5           ,  0.738636339943],
                        [ 0.856060606061,  0.5           ,  0.784090916989],
                        [ 0.871212121212,  0.5           ,  0.761363628466],
                        [ 0.871212121212,  0.5           ,  0.806818205512],
                        [ 0.901515151515,  0.5           ,  0.761363628466],
                        [ 0.901515151515,  0.5           ,  0.806818205512],
                        [ 0.825757575758,  0.5           ,  0.829545430852],
                        [ 0.825757575758,  0.5           ,  0.875000007898],
                        [ 0.856060606061,  0.5           ,  0.829545430852],
                        [ 0.856060606061,  0.5           ,  0.875000007898],
                        [ 0.871212121212,  0.5           ,  0.852272719375],
                        [ 0.871212121212,  0.5           ,  0.897727296421],
                        [ 0.901515151515,  0.5           ,  0.852272719375],
                        [ 0.901515151515,  0.5           ,  0.897727296421],
                        [ 0.825757575758,  0.5           ,  0.920454521761],
                        [ 0.825757575758,  0.5           ,  0.965909098807],
                        [ 0.856060606061,  0.5           ,  0.920454521761],
                        [ 0.856060606061,  0.5           ,  0.965909098807],
                        [ 0.871212121212,  0.5           ,  0.943181810284],
                        [ 0.871212121212,  0.5           ,  0.98863638733 ],
                        [ 0.901515151515,  0.5           ,  0.943181810284],
                        [ 0.901515151515,  0.5           ,  0.98863638733 ],
                        [ 0.916666666667,  0.5           ,  0.01136361267 ],
                        [ 0.916666666667,  0.5           ,  0.056818189716],
                        [ 0.94696969697 ,  0.5           ,  0.01136361267 ],
                        [ 0.94696969697 ,  0.5           ,  0.056818189716],
                        [ 0.962121212121,  0.5           ,  0.034090901193],
                        [ 0.962121212121,  0.5           ,  0.079545478239],
                        [ 0.992424242424,  0.5           ,  0.034090901193],
                        [ 0.992424242424,  0.5           ,  0.079545478239],
                        [ 0.916666666667,  0.5           ,  0.102272703579],
                        [ 0.916666666667,  0.5           ,  0.147727280625],
                        [ 0.94696969697 ,  0.5           ,  0.102272703579],
                        [ 0.94696969697 ,  0.5           ,  0.147727280625],
                        [ 0.962121212121,  0.5           ,  0.124999992102],
                        [ 0.962121212121,  0.5           ,  0.170454569148],
                        [ 0.992424242424,  0.5           ,  0.124999992102],
                        [ 0.992424242424,  0.5           ,  0.170454569148],
                        [ 0.916666666667,  0.5           ,  0.193181794488],
                        [ 0.916666666667,  0.5           ,  0.238636371534],
                        [ 0.94696969697 ,  0.5           ,  0.193181794488],
                        [ 0.94696969697 ,  0.5           ,  0.238636371534],
```

(continues on next page)

(continued from previous page)

```
                        [ 0.962121212121,  0.5           ,  0.215909083011],
                        [ 0.962121212121,  0.5           ,  0.261363660057],
                        [ 0.992424242424,  0.5           ,  0.215909083011],
                        [ 0.992424242424,  0.5           ,  0.261363660057],
                        [ 0.916666666667,  0.5           ,  0.284090885397],
                        [ 0.916666666667,  0.5           ,  0.329545462443],
                        [ 0.94696969697 ,  0.5           ,  0.284090885397],
                        [ 0.94696969697 ,  0.5           ,  0.329545462443],
                        [ 0.962121212121,  0.5           ,  0.30681817392 ],
                        [ 0.962121212121,  0.5           ,  0.352272750967],
                        [ 0.992424242424,  0.5           ,  0.30681817392 ],
                        [ 0.992424242424,  0.5           ,  0.352272750967],
                        [ 0.916666666667,  0.5           ,  0.374999976306],
                        [ 0.916666666667,  0.5           ,  0.420454553352],
                        [ 0.94696969697 ,  0.5           ,  0.374999976306],
                        [ 0.94696969697 ,  0.5           ,  0.420454553352],
                        [ 0.962121212121,  0.5           ,  0.397727264829],
                        [ 0.962121212121,  0.5           ,  0.443181841876],
                        [ 0.992424242424,  0.5           ,  0.397727264829],
                        [ 0.992424242424,  0.5           ,  0.443181841876],
                        [ 0.916666666667,  0.5           ,  0.465909067215],
                        [ 0.916666666667,  0.5           ,  0.511363644262],
                        [ 0.94696969697 ,  0.5           ,  0.465909067215],
                        [ 0.94696969697 ,  0.5           ,  0.511363644262],
                        [ 0.962121212121,  0.5           ,  0.488636355738],
                        [ 0.962121212121,  0.5           ,  0.534090932785],
                        [ 0.992424242424,  0.5           ,  0.488636355738],
                        [ 0.992424242424,  0.5           ,  0.534090932785],
                        [ 0.916666666667,  0.5           ,  0.556818158124],
                        [ 0.916666666667,  0.5           ,  0.602272735171],
                        [ 0.94696969697 ,  0.5           ,  0.556818158124],
                        [ 0.94696969697 ,  0.5           ,  0.602272735171],
                        [ 0.962121212121,  0.5           ,  0.579545446648],
                        [ 0.962121212121,  0.5           ,  0.625000023694],
                        [ 0.992424242424,  0.5           ,  0.579545446648],
                        [ 0.992424242424,  0.5           ,  0.625000023694],
                        [ 0.916666666667,  0.5           ,  0.647727249033],
                        [ 0.916666666667,  0.5           ,  0.69318182608 ],
                        [ 0.94696969697 ,  0.5           ,  0.647727249033],
                        [ 0.94696969697 ,  0.5           ,  0.69318182608 ],
                        [ 0.962121212121,  0.5           ,  0.670454537557],
                        [ 0.962121212121,  0.5           ,  0.715909114603],
                        [ 0.992424242424,  0.5           ,  0.670454537557],
                        [ 0.992424242424,  0.5           ,  0.715909114603],
                        [ 0.916666666667,  0.5           ,  0.738636339943],
                        [ 0.916666666667,  0.5           ,  0.784090916989],
                        [ 0.94696969697 ,  0.5           ,  0.738636339943],
                        [ 0.94696969697 ,  0.5           ,  0.784090916989],
                        [ 0.962121212121,  0.5           ,  0.761363628466],
                        [ 0.962121212121,  0.5           ,  0.806818205512],
                        [ 0.992424242424,  0.5           ,  0.761363628466],
                        [ 0.992424242424,  0.5           ,  0.806818205512],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.916666666667,  0.5           ,  0.829545430852],
                          [ 0.916666666667,  0.5           ,  0.875000007898],
                          [ 0.94696969697 ,  0.5           ,  0.829545430852],
                          [ 0.94696969697 ,  0.5           ,  0.875000007898],
                          [ 0.962121212121,  0.5           ,  0.852272719375],
                          [ 0.962121212121,  0.5           ,  0.897727296421],
                          [ 0.992424242424,  0.5           ,  0.852272719375],
                          [ 0.992424242424,  0.5           ,  0.897727296421],
                          [ 0.916666666667,  0.5           ,  0.920454521761],
                          [ 0.916666666667,  0.5           ,  0.965909098807],
                          [ 0.94696969697 ,  0.5           ,  0.920454521761],
                          [ 0.94696969697 ,  0.5           ,  0.965909098807],
                          [ 0.962121212121,  0.5           ,  0.943181810284],
                          [ 0.962121212121,  0.5           ,  0.98863638733 ],
                          [ 0.992424242424,  0.5           ,  0.943181810284],
                          [ 0.992424242424,  0.5           ,  0.98863638733 ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Add tags
bulk_configuration.addTags('5-ring',      [399, 481, 485, 488, 490])
bulk_configuration.addTags('6-ring',      [397, 399, 402, 404, 406, 488])
bulk_configuration.addTags('6-ring-bulk', [77, 79, 82, 84, 86, 168])
bulk_configuration.addTags('7-ring',      [481, 483, 484, 485, 486, 487, 569])

# Add tags
bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# RealSpaceSelfEnergyPDOS
real_space_self_energy_pdos = RealSpaceSelfEnergyPDOS(
    bulk_configuration,
    primitive_configuration,
    kpoints=MonkhorstPackGrid(1001, 1, 1),
    energies=numpy.linspace(-1, 1, 201) * eV,
    infinitesimal=2e-3*eV,
    primitive_layer_extensions=(1,1,1),
    automatic_primitive_layer_extensions=True,
    projections=ProjectOnTags,
    use_recursive_method=True
    )

nlsave('analysis.hdf5', real_space_self_energy_pdos)
```

Notes

The *RealSpaceSelfEnergyPDOS* object calculates the projected density of states (PDOS) using the real-space self-energy (RSSE) method described in[1]. The method is intended to be used to study the PDOS of isolated defects in an other-wise crystalline environment.

In the RSSE method, the interaction between periodic repeated images is removed such that one obtains the PDOS for a single defect in an infinite crystal. In a normal PDOS calculation performed with the *ProjectedDensityOfStates* object, the eigen-states and -energies are obtained from repeated super cells containing the defect. The interaction between the periodic images can affect the results and it often requires very large super cells to converge the results to the isolated defect limit.

The RealSpaceSelfEnergyPDOS improves on this since the interactions between periodic images is removed by the use of real-space self energies as explained in[Page 3747, 1].

When performing a RealSpaceSelfEnergyPDOS calculation, it is implicitly assumed that the effective potential in the boundary regions of the super-cell are converged to its bulk value (except maybe for a rigid energy shift). Before performing the RealSpaceSelfEnergyPDOS calculation it is advised to inspect the *EffectivePotential* or *HartreeDifferencePotential* to assure that the potential in the boundary regions is converged. This it particularly important in systems where a significant charge transfer takes place around the defect.

System setup and k-point sampling

When performing a *RealSpaceSelfEnergyPDOS* calculation there are some particular requirements to how the configuration is setup and in which directions it is needed to perform k-point averaging.

•The self-energy is calculated from the provided primitive configuration.

•The configuration containing the defect must have a unit cell, which is an integer repetition of the primitive configuration.

•The configuration containing the defect must have a boundary region, where the atoms are in the same positions as the repeated primitive configuration. In practice it is advised to setup the configuration containing the defect in the following steps (see figure below):

1)Start with the primitive configuration for the bulk host material.

2)Repeat the primitive configuration a suitable number of times in the desired directions.

3)Create the defect of interest in the center of the repeated configuration.

4)Optimize the geometry while constraining the atoms in the boundary region (usually 1-2 repetitions of the primitive configuration).

•The configuration must always be extended along the C-direction as this is the axis along which the self-energy is being calculated initially. This is similar to a NEGF calculation for a DeviceConfiguration. The C-direction is thus treated as a truly infinite direction and no k-point sampling is needed along the C-axis.

•k-point averaging should be performed along the other extended directions. In the graphene example above the graphene sheet is extended in the (A,C) plane and the *RealSpaceSelfEnergyPDOS* must therefore be k-point sampled along the A-direction only.

•The required number of k-points depends on the configuration and on the *infinitesimal* input. The *infinitesimal* is similar to a Gaussian broadening in *ProjectedDensityOfStates*. The most accurate results are obtained for high k-point sampling and small infinitesimal.

[1] Nick Papior, Gaetano Calogero, Susanne Leitherer, and Mads Brandbyge. Removing all periodic boundary conditions: Efficient nonequilibrium Greens function calculations. *Phys. Rev. B*, 100:195417, Nov 2019. URL: https://link.aps.org/doi/10.1103/PhysRevB.100.195417, doi:10.1103/PhysRevB.100.195417.

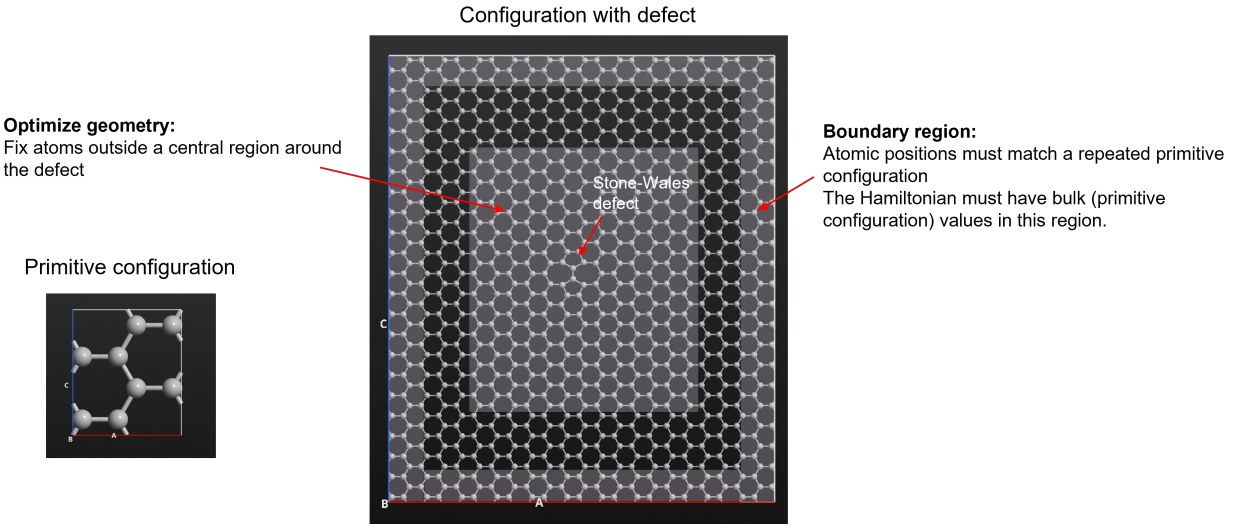


Fig. 4.35: Illustration of primitive configuration and configuration with the defect from the usage example above.

ReaxFFAtomicCharges

class ReaxFFAtomicCharges(*r_cut=PhysicalQuantity(10.0, Ang)*, *chi=None*, *eta=None*, *shielding=None*)

This object calculates atomic charges based on the ReaxFF charge equilibration method.

Parameters

- **r_cut** (*PhysicalQuantity* of type length) – The cutoff used in determining charge-charge interactions. **Default:** 10 Angstrom.
- **chi** (*dict*) – Custom atomic electronegativies, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type energy. **Default:** Electronegativities from the ReaxFF CHONSSiCaCsKSrNaMgAlCu_2015 potential set.
- **eta** (*dict*) – Custom atomic hardnesses, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type energy. **Default:** Atomic hardnesses from the ReaxFF CHONSSiCaCsKSrNaMgAlCu_2015 potential set.
- **shielding** (*dict*) – Custom electrostatic shielding parameters, as a dictionary with PeriodicTableElement objects as keys and values as *PhysicalQuantity* of type inverse distance. **Default:** Shielding parameters from the ReaxFF CHONSSiCaCsKSrNaMgAlCu_2015 potential set.

charges(*configuration*, *total_charge=None*, *decompose_bulk=True*)

Calculate the atomic charges on a given configuration.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration whose atomic charges are being calculated.
- **total_charge** (*PhysicalQuantity* of type charge) – The total charge of the configuration. **Default:** Match the integer charge of the configuration.
- **decompose_bulk** (*bool*) – Whether or not the configuration, if a BulkConfiguration, is decomposed into its parts to obtain the charges. **Default:** True

Returns
An array of atomic charges.

Return type
PhysicalQuantity of type charge

Usage Examples

Define a simulation box of 900 methanol molecules with a Dreiding calculator using ReaxFF calculated charges

```
# ------------------------------------------------------------------------------------------
↪----------
# Create Methanol Molecule
# ------------------------------------------------------------------------------------------
↪----------
# Define elements
elements = [Carbon, Hydrogen, Hydrogen, Hydrogen, Oxygen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 0.048333359398, -0.11839206808 , -0.068353692376],
                         [ 1.138333281204, -0.11839206808 , -0.068353692376],
                         [-0.31499994787 , -0.11839206808 , -1.096015473978],
                         [-0.31499994787 , -1.008373277446,  0.445477198425],
                         [-0.363333307269,  0.889981209366,  0.513830890801],
                         [-0.013333332377,  1.747302557837,  0.018856179479]]*Angstrom

# Set up configuration
methanol = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)

# Add tags
methanol.addTags('DREI_C_3',  [0])
methanol.addTags('DREI_H_',   [1, 2, 3])
methanol.addTags('DREI_H__A', [5])
methanol.addTags('DREI_O_3',  [4])

# Add bonds
bonds = [[0, 1],
         [0, 2],
         [0, 3],
         [0, 4],
         [4, 5]]
methanol.setBonds(bonds)

# ------------------------------------------------------------------------------------------
↪----------
# Calculate Atomic Charges
# ------------------------------------------------------------------------------------------
↪----------
# Create the charger and calculate the charges
charger = ReaxFFAtomicCharges()
methanol_charges = charger.charges(methanol)
```

(continues on next page)

(continued from previous page)

```
# Print charges
print('METHANOL CHARGES')
print("----------------------------")
print(f'{"Number":<8s}{"Atom":<8s}{"Charge":>12s}')
print("----------------------------")
for i, q in enumerate(methanol_charges):
    element = methanol.elements()[i].symbol()
    charge = q.inUnitsOf(elementary_charge)
    print(f'{i+1:<8d}{element:<8s}{charge:>12.4f}')
print("----------------------------")

# ------------------------------------------------------------------------------------------
↪----------
# Create A Box Of Methanol Molecules
# ------------------------------------------------------------------------------------------
↪----------
# Define the volume in which the molecules should be packed
u1, u2, u3 = numpy.diag([40.0] * 3) * Angstrom
cell = UnitCell(u1, u2, u3)

# Set the size of the buffer region
buffer_size = 2.0 * Angstrom

# Set Packmol parameters
tolerance = 2.0 * Angstrom
maximum_loops = 30

# Run the Packmol script to generate the packed configuration
methanol_box, packing_successful, packmol_message = runPackmol(
    [(methanol, 900)],
    cell,
    tolerance,
    buffer_size,
    maximum_loops,
)

# ------------------------------------------------------------------------------------------
↪----------
# Add A Dreiding Calculator With The Calculated Charges
# ------------------------------------------------------------------------------------------
↪----------
# Repeat the charges for each molecule
methanol_box_charges = numpy.tile(
    methanol_charges.inUnitsOf(elementary_charge),
    900
) * elementary_charge

# Create the potential builder
potential_builder = DreidingPotentialBuilder()

# Create the calculator, setting the atomic charges
```

(continues on next page)

(continued from previous page)

```
calculator = potential_builder.createCalculator(
    methanol_box,
    atomic_charges=methanol_box_charges
)

# Assign the calculator to the configuration
methanol_box.setCalculator(calculator)

# ------------------------------------------------------------------------------------------
↪-----------
# Save The Configuration
# ------------------------------------------------------------------------------------------
↪-----------
nlsave('Methanol_Box.hdf5', methanol_box)
```

calculating_charges_example.py

Notes

The ReaxFFAtomicCharges class implements the charge equilibration scheme included in the ReaxFF forcefield[1]. The general charge equilibration method assumes that the change of energy in response to charge polarization can be given as a simple Taylor series expansion, such that the total energy of the molecule, E, can be given as:

$$E(\mathbf{q}) = \sum_{i}^{N} \chi_i q_i + \eta_i q_i^2 + \sum_{j<i}^{N} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Here $\mathbf{q}$ is the vector of atomic charges, χ_i is the atomic electronegativity, η_i is the atomic hardness and r_{ij} is the inter-atomic distance. Getting the derivative of the energy with respect to an atomic charge gives the chemical potential for that atom. Assuming that the chemical potential of each atom is equal and that the charges sum to a particular value gives a series of linear equations that can be solved for the atomic charges. This gives a simple and yet reasonably effective method for estimating the atomic partial charges.

One of the main modifications of this idea in the ReaxFF charge equilibration scheme is that here the atomic ionization data are treated as adjustable parameters. These are typically tuned to replicate specific charges, and are therefore not necessarily physically meaningful. The adjustable nature of these parameters also means that there are a number of ReaxFF potential sets with parameters tuned to different applications. This is distinct from the QEq charge equilibration method where the ionization data generally retains their atomic values.

Another important modification used in ReaxFF charge equilibration is that the charge-charge electrostatics are calculated with the shielded Coulomb potential

$$V(r_{ij};\gamma_{ij}) = \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{\left[r^3 + (1/\gamma_{ij})^3\right]^{1/3}}$$

This introduces a pairwise shielding parameter γ_{ij}. This is calculated as the geometric mean of atomic shielding parameters. The charge-charge interactions are also multiplied by a 7th order polynomial tapering function. This ensures that the interactions are reduced to zero at the cutoff. In periodic system the combination of adjustable atomic parameters, shielded Coulomb function and tapering function allow the charge-charge interactions to converge without Ewald summation. This means that the specified interaction distance cutoff can be used in both molecular and bulk configurations.

[1] A.ăC.ăT van Duin, S.ăDasgupta, F.ăLorant, and W.ăA. GoddardăIII. ReaxFF: A Reactive Force Field for Hydrocarbons. *J. Phys. Chem. A*, 105(41):9396–9409, 2001. doi:10.1021/jp004368u.

To calculate the partial atomic charges for a configuration, an instance of the ReaxFFAtomicCharges class first has to be created. This class has a few parameters to control how it is instantiated. The cutoff used to calculate the charge-charge interactions is controlled with the parameter `r_cut`. This defines the distance beyond which electrostatic interactions are ignored.

In the ReaxFF scheme each atom has three parameters, the atomic electronegivity, hardness and electrostatic shielding. These can be modified using the parameters `chi`, `eta` and `shielding` respectively. These parameters allow different ReaxFF potential sets to be used. By default the *ReaxFFAtomicCharges* class uses the potential set of van Duin et. al.[2]. This potential set supports H, C, N, O, Mg, Al, Si, S, K, Cu, Sr and Cs atoms. Each parameter takes dictionary of *PeriodicTableElement* and `PhysicalQuanity` of appropriate units. If parameters are not specified for an atom the default atomic parameters are used. The following example shows how to explicitly set the default atomic parameters for the methanol example above.

```
charger = QEqAtomicCharges(
   chi={Hydrogen: 3.7248*eV, Carbon: 5.9666*eV, Oxygen: 8.5000*eV},
   eta={Hydrogen: 9.6093*eV, Carbon: 7.0000*eV, Oxygen: 8.3122*eV},
   shielding={Hydrogen: 0.8203/Ang, Carbon: 0.9000/Ang, Oxygen: 1.0898/Ang}
)
methanol_charges = charger.charges(methanol)
```

Once the ReaxFFAtomicCharges class is instantiated, the atomic charges can be calculated using the `charges` method. This takes the configuration of the system and returns an array of atomic charges. The total charge of the configuration can be set using the `total_charge` argument. In cases where the charges are sought for a group of individual molecules, the charges can be calculated individually for each molecule using the `decompose_bulk` argument. This is useful in cases such as a polymer melt or a molecular liquid. Calculating the charges of each molecule individually ensures that each molecule has the correct total charge, as well as removing any artifacts of how the molecules are packed. It also can speed up the calculation of the charges significantly. This is currently only supported for *BulkConfiguration* configurations.

ReaxFFPotential

class ReaxFFPotential(*file=None*, *qeq=None*, *qeqEps=None*, *qeqMaxIter=None*, *qeqSolver=None*, *lgvdw=None*, *bond_rcut=None*, *hbond_rcut=None*, *thb_bond_rcut=None*, *thb_bond_rcutsq=None*, *multiwell=None*, *multiwell_torsion=None*, *smoothed_torsion=None*, *smoothed_lone_pair=None*, *smoothed_sbo2=None*, *bo_cutoff=None*, *strict_bondpairs=None*, *parameter_container=None*)

Constructor of the ReaxFF potential.

Parameters

- **file** (*str*) – The name of the file that contains the ReaxFF parameters. If parameter_container is not None, file will be ignored.
- **qeq** (*bool*) – Flag for switching the charge equilibration (QeQ) on or off.
- **qeqEps** (*float*) – The accuracy of the QeQ method.
- **qeqMaxIter** (*int*) – Maximal number of iterations for the QeQ method.
- **qeqSolver** – The type of solver that will be used. Must be one of the following variables:

ReaxFFPotential.sparse ReaxFFPotential.dense (only in case of 1 MPI process)

- **lgvdw** (*bool*) – Flag for switching the LG correction on or off.

[2] GeorgeăM. Psofogiannakis, JohnăF. McCleerey, Eugenio Jaramillo, and Adri C.ăT. van Duin. ReaxFF reactive molecular dynamics simulation of the hydration of Cu-SSZ-13 zeolite and the formation of Cu dimers. *J. Phys. Chem. C.*, 119:6678, March 2015. URL: https://pubs.acs.org/doi/abs/10.1021/acs.jpcc.5b00699, doi:10.1021/acs.jpcc.5b00699.

•**bond_rcut** (*PhysicalQuantity of type length*) – Potential parameter.

•**hbond_rcut** (*PhysicalQuantity of type length*) – Potential parameter.

•**thb_bond_rcut** (*float*) – Potential parameter.

•**thb_bond_rcutsq** (*float*) – Potential parameter.

•**multiwell** (*bool*) – Potential parameter.

•**multiwell_torsion** (*bool*) – Potential parameter.

•**smoothed_torsion** (*bool*) – Potential parameter.

•**smoothed_lone_pair** (*bool*) – Potential parameter.

•**smoothed_sbo2** (*bool*) – Potential parameter.

•**bo_cutoff** (*float*) – Potential parameter.

•**strict_bondpairs** (*bool*) – Potential parameter.

•**parameter_container** (*ReaxFFParameterContainer*) – A ReaxFFParameterContainer which contains the ReaxFF parameters. If parameter_container is not None, the file will be ignored.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*name*, *particleTypes=None*, *multiangle_index=None*)

Get the value of a single parameter.

Parameters

•**name** (*str*) – The name of the parameter.

•**particleTypes** – see setParameter

•**multiangle_index** – see setParameter

getParameterByIndex(*block*, *index*, *particleTypes=None*, *multiangle_index=None*)

Get the value of a single parameter.

Parameters

•**block** (*ReaxFFBlock*) – The block that contains the parameter.

•**index** (*int*) – The index of the parameter. Example: The 2nd parameter in the angles block, p_val1, has index 2. The index is 1-based.

•**particleTypes** – see setParameter

•**multiangle_index** – see setParameter

setParameter(*name*, *value*, *particleTypes=None*, *multiangle_index=None*)

Assign a value to a single parameter.

Parameters

•**name** (*str*) – The name of the parameter.

- **value** (*PhysicalQuantity or float or int*) – The value the parameter will be set to.
- **particleTypes** (*None or ParticleType or list(ParticleType) or tuple(ParticleType)*) – The particle types that identify the parameter. For torsion paremeters, the 1st and 4th particleType may be set to ParticleTypes(0) if all particle types are meant.
- **multiangle_index** (*int*) – If the parameter is an angle/torsion parameter, and multiwell/multiwell_torsion is set to True, then multiangle_index is used to specify which of the parameter blocks is meant. The index is 0-based, which means that multiangle_index = 0 specifies the first parameter block.

setParameterByIndex(*block*, *index*, *value*, *particleTypes=None*, *multiangle_index=None*)

Assign a value to a single parameter.

Parameters

- **block** (*ReaxFFBlock*) – The block that contains the parameter.
- **index** (*int*) – The index of the parameter. Example: The 2nd parameter in the angles block, p_val1, has index 2. The index is 1-based.
- **value** (*PhysicalQuantity or float or int*) – The value the parameter will be set to.
- **particleTypes** – See setParameter
- **multiangle_index** – See setParameter

Usage Examples

Define a potential for rutile titanium dioxide by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a TiO2 rutile crystal
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleTetragonal(4.593*Angstrom, 2.959*Angstrom)

# Define elements
elements = [Titanium, Titanium, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.0000,  0.0000,  0.0000],
                          [ 0.5000,  0.5000,  0.5000],
                          [ 0.3051,  0.3051,  0.0000],
                          [ 0.6949,  0.6949,  0.0000],
                          [ 0.8051,  0.1949,  0.5000],
                          [ 0.1949,  0.8051,  0.5000]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='ReaxFF_CHONTi_2012')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='H',
                                          mass=1.00794*atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='Ti',
                                          mass=47.867*atomic_mass_unit))

# Add the ReaxFF potential to the potential set
potentialSet.addPotential(ReaxFFPotential(file='ffield.reax.Ti_C_H_O',
                                          qeq=True,
                                          qeqEps=1.000000e-06,
                                          qeqMaxIter=200,
                                          lgvdw=False,
                                          bond_rcut=5.0*Angstrom,
                                          hbond_rcut=7.5*Angstrom,
                                          thb_bond_rcut=0.001000,
                                          thb_bond_rcutsq=0.000010,
                                          multiwell=False,
                                          strict_bondpairs=True))

# Create a new TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

Notes

The *ReaxFFPotential* implements the **reactive force field (ReaxFF)** to describe reactive processes[1].

You can only change the numerical parameters such as cutoff radii, convergence parameters and others. The actual potential parameters are read from a file, specified by the `file` argument. It is generally suggested to use only the predefined parameter sets.

You should carefully investigate and test whether the chosen ReaxFF potential is suitable for your desired application, as the potential may have been parametrized in a completely different context, even if it supports the same elements.

[1] A.ăC.ăT van Duin, S.ăDasgupta, F.ăLorant, and W.ăA. GoddardăIII. ReaxFF: A Reactive Force Field for Hydrocarbons. *J. Phys. Chem. A*, 105(41):9396–9409, 2001. doi:10.1021/jp004368u.

RecursionSelfEnergy

class RecursionSelfEnergy(*save_self_energies=None*, *tolerance=None*, *maximum_iteration=None*, *sparse_threshold=None*, *storage_strategy=None*, *enable_block_algorithm=None*)

Self-energy calculator based on the Recursion method. This method is an iterative scheme for getting self energies, with convergence $O(2^N)$

Parameters

- **storage_strategy** (*SaveInMemory* | *StoreOnDisk* | *NoStorage*) – The way self energies are stored between iterations. **Default:** *SaveInMemory()*
- **tolerance** (*float > 0*) – The tolerance of convergence of the self energies before termination of the recursion. **Default:** `1e-13`
- **maximum_iteration** (*int > 0*) – The maximum number of recursion steps allowed before termination. **Default:** `400`
- **sparse_threshold** (*float > 0*) – Self energies (in Hartree) smaller than this value treated as 0 when storing as sparse matrix. **Default:** `1e-12`
- **enable_block_algorithm** (*bool*) – Flag that determines whether an algorithm exploiting the block structure of the transfer matrices should be used if possible. **Default:** `True`
- **save_self_energies** – **Deprecated:** from v2017.0, use `storage_strategy` instead.

enableBlockAlgorithm()

Returns
Whether to use the algorithm that exploits the block structure of the transfer matrices.

Return type
bool

maximumIteration()

Returns
The maximum number of recursion steps.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

saveSelfEnergies()

Deprecated: from v2017.0

sparseThreshold()

Returns
Values treated as 0.

Return type
float

storageStrategy()

Returns
The way self energies are stored between iterations.

Return type
SaveInMemory | *StoreOnDisk* | *NoStorage*

tolerance()

Returns
The tolerance of convergence.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define that the self energies on the complex contour are calculated with the recursion method, and the self energies calculated in the first iteration of the SCF loop are saved in memory and reused in the following iterations.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_complex=RecursionSelfEnergy(storage_strategy=SaveInMemory()),
)
```

Storing the self energies potentially requires a significant amount of memory. Therefore it can be useful to define that the self energies on the complex contour are not saved.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_complex=RecursionSelfEnergy(storage_strategy=NoStorage()),
)
```

While this reduces the required memory, it also has a negative impact on performance.

Instead of saving the self energies in memory, they can stored on disk. This setting provides the same performance as *SaveInMemory()* and requires sufficient disk space (potentially hundreds of gigabytes).

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_complex=RecursionSelfEnergy(storage_strategy=StoreOnDisk()),
)
```

Notes

The *RecursionSelfEnergy* uses the iterative scheme by Lopez, Lopez and Rubio[1] for calculating the self energy matrix.

[1] M.ăP. LopezăSancho, J.ăM. LopezăSancho, and J.ăRubio. Highly convergent schemes for the calculation of bulk and surface Green functions. *J. Phys. F: Metal Physics*, 15(4):851, 1985. URL: http://stacks.iop.org/0305-4608/15/i=4/a=009.

ReferenceChemicalPotential

class ReferenceChemicalPotential(*element*, *internal_energy=None*, *entropy=None*, *reference_temperature=None*)

A class defining the chemical potential for an atomic species.

Parameters

- **element** (*PeriodicTableElement*) – The element for which the atomic chemical potential is defined.
- **internal_energy** (*PhysicalQuantity* of type energy) – The internal energy per atom of the specified element in some reference configuration. **Default:** 0*eV.
- **entropy** (*PhysicalQuantity* of type energy per temperature) – The entropy per atom of the specified element in some reference configuration. **Default:** 0*eV/Kelvin.
- **reference_temperature** (*PhysicalQuantity* of type temperature) – Temperature at which the reference data is defined.

chemicalPotential(*temperature=None*)

Calculate the chemical potential.

Returns
The chemical potential of the atom calculated from the internal energy and entropy per atom of the specified element in some reference configuration.

Return type
PhysicalQuantity of type energy | None

electronicInternalEnergy()

Returns
The internal energy component of the atomic chemical potential.

Return type
PhysicalQuantity of type energy

element()

Returns
The element for which the atomic chemical potential is defined.

Return type
PeriodicTableElement

entropy(*temperature=None*)

Returns
The internal energy component of the atomic chemical potential.

Return type
PhysicalQuantity of type energy per temperature

referenceTemperature()

Returns
The reference temperature for the chemical potential.

Return type
PhysicalQuantity of type temperature.

uniqueString()

Return a unique string representing the state of the object.

vibrationalInternalEnergy(*temperature=None*)

Returns
The vibrational part of the internal energy. In this reference it is defined as 0 eV, as vibrational energy is included in the specified internal energy.

Return type
PhysicalQuantity of type energy

Notes

The *ReferenceChemicalPotential* object defines an atomic chemical potential used in a *ChargedPointDefectConfiguration* calculation. The *ReferenceChemicalPotential* provides the reservoir energy for the element being added or removed from the pristine material to form the defect. This energy is used in calculating the formation energy and concentration of the defect. The value of the reservoir energy can depend on the nature of the environment around the defect. The atomic chemical potential also depends on the reference calculator used in the calculation, as it also includes the energy of the isolated atom according to this calculator.

The *ReferenceChemicalPotential* object is created by specifying the element for the potential. Optionally the internal energy of the atom is given using the `internal_energy` argument. The atomic entropy, which is usually derived from the vibrational entropy in the reference state, can also be given with the `entropy` argument. If the entropy is given, the total value of the chemical potential is temperature dependent.

RegularKpointGrid

class RegularKpointGrid(*ka_range=None*, *kb_range=None*, *kc_range=None*, *na=None*, *nb=None*, *nc=None*, *symmetries=None*, *force_timereversal=None*)

Parameters

- **ka_range** (*list(2) of float | float*) – Interval for k-points in the kA-direction given in fractional coordinates. If a single number is given there will be no k-points in this direction (*na* will be automatically be set to 1 as default). **Default:** [-0.5, 0.5]
- **kb_range** (*list(2) of float | float*) – Interval for k-points in the kB-direction given in fractional coordinates. If a single number is given there will be no k-points in this direction (*nb* will be automatically be set to 1 as default). **Default:** [-0.5, 0.5]
- **kc_range** (*list(2) of float | float*) – Interval for k-points in the kC-direction given in fractional coordinates. If a single number is given there will be no k-points in this direction (*nc* will be automatically be set to 1 as default). **Default:** [-0.5, 0.5]
- **na** (*Positive int.*) – Number of grid points in the kA-direction. **Default:** 2 when *ka_range* is a list | 1 when *ka_range* is a float
- **nb** (*Positive int.*) – Number of grid points in the kB-direction. **Default:** 2 when *kb_range* is a list | 1 when *kb_range* is a float
- **nc** (*Positive int.*) – Number of grid points in the kC-direction. **Default:** 2 when *kc_range* is a list | 1 when *kc_range* is a float
- **symmetries** (*list(n_symmetries) of (array(3, 3), array(3)).*) – A list of symmetries to apply for reducing the k-points. **Default:** Empty list, which means ATK will determine the relevant symmetries.
- **force_timereversal** (*bool*) – Whether to enforce time reversal symmetry. **Default:** True

`allKpoints()`

Returns
All the k-points of the *RegularKpointGrid* before symmetry reduction.

Return type
ndarray(N,3) of float where N is the number of k-points.

`allKpointsWeights()`

Returns
The weight of all k-points before symmetry reduction.

Return type
ndarray(n) for float where n is the number of k-points before symmetry reduction.

`foldoutMapVector()`

Query method for getting the map between the symmetry reduced k-points and unreduced k-points.

Returns
A list with indices matching the k-points in the symmetry reduced set and in the unreduced set.

Return type
ndarray(N) with integer values between 0 and M-1. N is the number of k-points in the unreduced set, M is the number of k-points in the reduced set.

`forceTimereversal()`

Returns
Whatever force_timereversal is enabled or disabled.

Return type
bool

`kaRange()`

Returns
The range of k-points in the kA-direction.

Return type
list(2) of float.

`kbRange()`

Returns
The range of k-points in the kB-direction.

Return type
list(2) of float.

`kcRange()`

Returns
The range of k-points in the kC-direction.

Return type
list(2) of float.

`kpoints()`

Returns
The symmetry reduced kpoints.

Return type
ndarray(N,3) of float where N is the number of symmetry reduced k-points.

`kpointsWeights()`

Returns
The weights of the symmetry reduced k-points.

Return type
ndarray(n) for floats, where n is the number of unreduced number of k-points.

`na()`

Returns
The number of k-points along the kA-direction.

Return type
Positive int.

`nb()`

Returns
The number of k-points along the kB-direction.

Return type
Positive int.

`nc()`

Returns
The number of k-points along the kC-direction.

Return type
Positive int.

`nlinfo()`

Returns
The nlinfo.

Return type
dict

`symmetries()`

Returns
The symmetries to apply for reducing the k-points.

Return type
list(n_symmetries) of (array(3, 3), array(3)).

`uniqueString()`

Return a unique string representing the state of the object.

RelaxDimerParameters

class RelaxDimerParameters(*max_forces=None*, *max_steps=None*, *max_step_length=None*, *finite_difference_method=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *trajectory_interval=None*, *restart_strategy=None*, *remove_drift=None*, *enable_optimization_stop_file=None*, *trial_step_length=None*, *min_rot_force=None*, *max_rot_force=None*, *trial_angle=None*, *max_number_of_rotations=None*, *extrapolate_rot_force=None*)

Class for representing RelaxDimerParameters.

Parameters

•**max_forces** (*PhysicalQuantity* with force units) – The maximum force convergence criterion. **Default:** 0.05 * eV / Angstrom

•**max_steps** (*int*) – The maximum number of optimization steps allowed. **Default:** 200

•**max_step_length** (*PhysicalQuantity* with length units) – The maximum step length the optimizer may take. **Default:** 0.2 * Angstrom

•**finite_difference_method** (NLFlag (Forward | Central)) – The finite difference scheme to use. Forward uses only one configuration for the force calculation, while Central uses two configurations. Thus, Forward is about a third faster than Central. However, Central is more accurate and stable which can result in overall fewer steps to convergence. If Forward is used, it is recommended to adjust the dimer_separation parameter in the *class: `~.DimerConfiguration* object to a smaller value. **Default:** Central

•**constraints** (*list of ints*) – List of atom indices that are kept fixed during optimization. **Default:** [].

•**trajectory_filename** (*str | None*) – The filename to write the trajectory to. **Default:** The trajectory is not saved.

•**trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. If the object-id already exists in the given file, it will be overwritten. **Default:** None

•**log_filename_prefix** (Automatic | str | None) – The logging output from each dimer endpoint will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** Automatic.

•**trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1

•**restart_strategy** (NoRestart | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** RestartFromTrajectory()

•**remove_drift** (*bool*) – Controls if the drift is removed from the forces. **Default:** True

•**enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the dimer relaxation. If True, creation of the stop file will stop the relaxation at the next step. The name of the stop file will be shown in the log output; it will be stop-dimer-relaxation-uniqueID, where uniqueID is a randomly generated identifier for this relaxation. The file must be created in the current working directory. **Default:** True

•**trial_step_length** (*PhysicalQuantity* of type length) – The trial step to determine the ideal step. **Default:** 0.001 * Angstrom

•**min_rot_force** (*PhysicalQuantity* of type force) – The minimum rotational force necessary to attempt a rotation. **Default:** 0.1 * eV/Angstrom

•**max_rot_force** (*PhysicalQuantity* of type force) – The maximum rotational force that terminates the eigenmode optimization. A rotation will always be performed before checking this parameter. **Default:** 1.0 * eV/Angstrom

•**trial_angle** (*float*) – The trial angle in radians. **Default:** pi / 4.0 (45 degrees)

•**max_number_of_rotations** (*Positive int*) – The maximum number of rotations. **Default:** 1

•**extrapolate_rot_force** (*bool*) – Use an extrapolation scheme to estimate the rotational force. **Default:** False

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

RemoveElementFromSubstrate

class RemoveElementFromSubstrate(*element*, *only_fixed_layer=False*)

Removes the specified element from the configuration at the end of the molecular dynamics simulation. This can be used to remove ions after a sputtering event. By default atoms of the specified element is removed from the substrate except in the fixed layer. There is an option to remove atoms only from the fixed layer.

Parameters

•**element** (PeriodicTableElement) – The element that should be removed.

•**only_fixed_layer** (*bool*) – Remove atoms only from the fixed layer.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run an etching simulation of nitrogen on a silica surface using a ReaxFF potential, with an additional RemoveElementFromSubstrate hook function after each event which removes excess nitrogen.

```
# Attach ReaxFF calculator.
potentialSet = ReaxFF_CHONSSi_2012(strict_bondpairs = None)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
bulk_configuration.setCalculator(calculator)

surface_process_simulation = SurfaceProcessSimulation(
    substrate=bulk_configuration,
    filename='sio2_n_etching.hdf5',
    object_id='sps_0',
    random_seed=7,
    temperature=300.0*Kelvin,
    fixed_thickness=4.0*Angstrom,
    thermostat_thickness=8*Angstrom,
    log_interval=10,
)
```

(continues on next page)

(continued from previous page)

```
# Run 5 deposition events, each 20 ps long.
surface_process_simulation.addSequence(
    molecule=Nitrogen,
    number_of_events=5,
    md_time=20*ps,
    time_step=1.0*fs,
    mean_kinetic_energy=25*eV,
    mean_incident_angle=0*Degrees,
    std_kinetic_energy=0.3*eV,
    std_incident_angle=2*Degrees,
    post_md_hooks=[RemoveElementFromSubstrate(Nitrogen)],
)

surface_process_simulation.update()
```

sio2_n_etching.py

The results from the simulation can be visualized using the Movie Tool on the QATK LabFloor.

See also ref.spshookscommon.notes.

RepeatESPPartialCharges

class RepeatESPPartialCharges(*configuration*, *potential*, *total_charge=None*, *tag_prefix=None*, *charge_weight=None*, *inner_vdw_shell=None*, *outer_vdw_shell=None*, *stride=None*, *r_cut=None*, *damping_factor=None*)

Calculate atomic partial charges to reproduce the electrostatic potential based on the REPEAT algorithm.

Parameters

- **configuration** (MoleculConfiguration | *BulkConfiguration*) – The configuration for which the charges are fitted.
- **potential** (*ElectrostaticDifferencePotential*) – The electrostatic potential the charges are fitted to.
- **total_charge** (*PhysicalQuantity* of type charge) – The total charge of the configuration.
- **tag_prefix** (*str*) – Prefix of the tags on the configuration used to constrain charges.
- **charge_weight** (*float | list of float*) – Weights used to constrain fitted charges to the partial charges on the configuration.
- **inner_vdw_shell** (*float*) – Ratio of the van der Waals radius of the atoms used to exclude grid points close to an atom.
- **outer_vdw_shell** (*float*) – Ratio of the van der Waals radius of the atoms used to exclude grid points far away from any atom.
- **stride** (*int*) – Stride used to skip grid points in the electrostatic potential. A value of 1 specifies that every grid point is used.
- **r_cut** (*PhysicalQuantity* of type length) – The cutoff used in calculating electrostatic interactions in *BulkConfiguration*.
- **damping_factor** (*PhysicalQuantity* of type inverse length) – The damping factor used in calculating electrostatic interactions in *BulkConfiguration*.

dampingFactor()

Returns
Damping factor used for fitted electrostatic interactions in bulk materials.

Return type
PhysicalQuantity of type inverse length

evaluate(*atom_index=None*)

Return the partial charges of the selected atom or all atoms.

Parameters
atom_index (*int*) – Index of the selected atom. **Default:** All atoms.

Returns
A single charge if *atom_index* is not None, otherwise an array of charges.

Return type
PhysicalQuantity of type charge

groupIndices()

Returns
Indices of groups of atoms constrained to have the same fitted charge.

Return type
list of lists

innerVdwShell()

Returns
The inner radius factor of the shell for electrostatic potential grid points.

Return type
float

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

outerVdwShell()

Returns
The outer radius of the shell for electrostatic potential grid points.

Return type
float

partialCharges()

Returns
The fitted partial charges for the molecule.

Return type
PhysicalQuantity of type charge

`rCut()`

Returns
Electrostatic potential distance cutoff used in bulk materials.

Return type
PhysicalQuantity of type length

`relativeRootMeanSquareError()`

Returns
The relative root mean squared error of the electrostatic potential fit.

Return type
float

`restrainCharges()`

Returns
The reference charges used to weight the fitted charges.

Return type
PhysicalQuantity of type charge

`restrainWeights()`

Returns
The weights used to restrain the fitted charges.

Return type
ndarray

`setMetatext(`*metatext*`)`
Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`stride()`

Returns
Density of electrostatic potential points selected for fitting partial charges.

Return type
int

`tags()`

Returns
Tags used to constrain atoms to have the same partial charges.

Return type
list of str

`totalCharge()`

Returns
The total charge of the configuration the charges are being fitted to.

Return type
PhysicalQuantity of type charge

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Assign partial charges to the battery electrolyte molecule ethylene carbonate. In this example the OPLS potential is used to constrain atoms with the same type to have the same partial charges.

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Molecule Configuration
# ------------------------------------------------------------
# Define elements
elements = [Carbon, Oxygen, Oxygen, Oxygen, Carbon, Carbon, Hydrogen, Hydrogen,
            Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 1.609431413032,  0.050029424266, -0.200214465477],
                         [ 2.821872471776,  0.074345609581, -0.33257452873 ],
                         [ 0.786730069642,  0.285844748805, -1.270661995358],
                         [ 1.046027032925, -0.213547522321,  1.021790323208],
                         [-0.320536198865, -0.108625807588,  0.763421170161],
                         [-0.484549258604,  0.072564433479, -0.73758247059 ],
                         [-0.839575976013, -1.027933189299,  1.111755833993],
                         [-0.732343636487,  0.769867954492,  1.307351219363],
                         [-0.918484269576, -0.844342167145, -1.192644130163],
                         [-1.139565310547,  0.94179651573 , -0.964104219255]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
)

# Add tags
molecule_configuration.addTags('OPLS_CT_226', [4, 5])
molecule_configuration.addTags('OPLS_C_32',   [0])
molecule_configuration.addTags('OPLS_HC_63',  [6, 7, 8, 9])
molecule_configuration.addTags('OPLS_OS_22',  [2, 3])
molecule_configuration.addTags('OPLS_O_23',   [1])

# Add bonds
bonds = [[0, 1],
         [0, 2],
         [0, 3],
         [2, 5],
         [3, 4],
         [4, 5],
```

(continues on next page)

(continued from previous page)

```
         [4, 6],
         [4, 7],
         [5, 8],
         [5, 9]]
molecule_configuration.setBonds(bonds)

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=125.0*Hartree,
)

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
)

molecule_configuration.setCalculator(calculator)
nlprint(molecule_configuration)
molecule_configuration.update()
nlsave('ethylene_earbonate_charges.hdf5', molecule_configuration)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
molecule_configuration = OptimizeGeometry(
    molecule_configuration,
    max_forces=0.05*eV/Ang,
    max_steps=200,
    max_step_length=0.2*Ang,
    trajectory_filename='ethylene_earbonate_charges_trajectory.hdf5',
    trajectory_interval=5.0*Minute,
    restart_strategy=RestartFromTrajectory(),
    optimize_cell=False,
    optimizer_method=LBFGS(),
    enable_optimization_stop_file=True,
)
nlsave('ethylene_earbonate_charges.hdf5', molecule_configuration)
nlprint(molecule_configuration)

# ------------------------------------------------------------
# Electrostatic Difference Potential
# ------------------------------------------------------------
electrostatic_difference_potential = ElectrostaticDifferencePotential(molecule_
↪configuration)
nlsave('ethylene_earbonate_charges.hdf5', electrostatic_difference_potential)

# ------------------------------------------------------------
# Repeat ESP Partial Charges
# ------------------------------------------------------------
repeat_esp_partial_charges = RepeatESPPartialCharges(
    molecule_configuration,
```

(continues on next page)

(continued from previous page)

```
    electrostatic_difference_potential,
    total_charge=0*elementary_charge,
    inner_vdw_shell=1.0,
    outer_vdw_shell=3.0,
    stride=1,
    tag_prefix='OPLS_',
    )
molecule_configuration.setPartialCharges(repeat_esp_partial_charges.evaluate())
nlsave('ethylene_earbonate_charges.hdf5', molecule_configuration)
nlsave('ethylene_earbonate_charges.hdf5', repeat_esp_partial_charges)
```

ethylene_carbonate_charges.py

Notes

The `RespeatESPPartialCharges` class fits atomic partial charges to either a *MoleculeConfiguration* or *BulkConfiguration* based on a given electrostatic potential. This potential is typically calculated with density functional theory (DFT). Theseo partial charges can then be used in a forcefield based calculation to model electrostatic interactions between atoms. As the charges are fit to a DFT electrostatic potential, it allows the long-range electrostatic interactions to be similar to those predicted by DFT, resulting in a more accurate simulation.

The fitting processes uses the REPEAT algorithm to assign the partial charges[1]. This fits the charges using a linear least squares approach, minimizing the difference between the supplied electrostatic potential and the potential calculated from the fitted partial charges. The grid points in the electrostatic potential used in the fitting are selected such that they are all within a shell around the molecule. Grid points that are too close to an atom are within the electron density around the atom where the simple approximation of point charges breaks down. Similarly the electrostatic potential far from any atom is weakly influenced by many surrounding atoms and therefore not useful in isolating the contribution of single atoms to the electrostatic potential. The inner and outer radii used to define the shell of grid points used in the fitting can be set using the options `inner_vdw_shell` and `outer_vdw_shell`. These are scale factors that are combined with the van der Waals radius of each atom to determine the shell radii. Depending on the density mesh cutoff used in the DFT calculation, the electrostatic potential can also have a large number of closely spaced grid points. It is possible to down-sample the electrostatic potential grid, skipping over a set number of points in each direction. This is done using the `stride` argument, which takes an integer describing the increment used when reading points from the grid. The default value of 1 uses all the grid points in the given electrostatic potential.

As the charges are fit using a simple least squares method, it is possible that some atoms may be assigned chemically unreasonable charges, such as charges exceeding the number of valence electrons. This is most commonly an issue in configurations that have buried atoms, that is atoms that are not near any electrostatic grid points used in the fitting. This may happen for instance in atoms contained within a bulk surface. To help charges retain chemically meaningful values, some constraints can be added to the fit. One constraint that can be added is to constrain groups of atoms to have the same partial charge. This is done using tags. Tags identifying atom that have the same charge can be added to the configuration with a set prefix, with one tag for each atom. The tag prefix is then given as input to the `RespeatESPPartialCharges` class using the `tag_prefix` argument. Tags from bonded forcefields such as DREIDING or OPLS can be used, or custom tags can also be set. The given example with ethylene carbonate uses the OPLS tags to constrain atoms with the same OPLS type to have the same partial charge.

It is also possible to set reference charges for the fit, and weight the fitting towards these charges. This weighting can give reasonable values for charges that are not well determined by the electrostatic potential, while still allowing some influence from the potential. Reference charges are taken from the existing atomic partial charges on the configuration. Weights for each atom are given with the argument `charge_weight`. If a single value is given, this value is used for

[1] Carlos Campana, Bastien Mussard, and TomăK. Woo. Electrostatic potential derived atomic charges for periodic systems Using a modified error functional. *Journal of Chemical Theory and Computation*, 5(10):2866–2878, 2009. URL: https://pubs.acs.org/doi/10.1021/ct9003405, doi:https://doi.org/10.1021/ct9003405.

all atoms. Giving individual values for each atom allows only under determined charges to be weighted to particular values. The size of the weight determines how strongly the existing partial charge influences the values of the fitted charges.

In molecular and bulk configurations the summation of the fitted electrostatic potential is done in slightly different ways. In molecules the fitted electrostatic potential is represented with the normal Coulomb potential, which does not require additional parameters. In the case of bulk configurations the fitted electrostatic potential is represented with a damped shifted force potential[2]. This requires both a distance cutoff and a damping factor. These can be modified using the arguments `r_cut` and `damping_factor` respectively.

Repulsive12Potential

class Repulsive12Potential(*particleType1*, *particleType2*, *c*, *r_cut=None*, *bonded_mode=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **c** (*PhysicalQuantity of type length**12 * energy*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.
- **bonded_mode** – Either Repulsive12Potential.evaluateAll or Repulsive12Potential.evaluate14 or Repulsive12Potential.evaluate15Only.

  If evaluateAll is chosen, interactions between all particles - even those that are connected by fixed bonds - are evaluated, using the sigma and epsilon parameters.

  If evaluate14 is chosen, interactions between particles that are connected by a bond path of length 1, 2, or 3 are omitted.

  If evaluate15Only is chosen, only interactions between particles that are connected by a bond path of length 4 are considered.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

[2] De-Li Chen, AbrahamăC. Stern, Brian Space, and J.ăKarl Johnson. Atomic charges derived from electrostatic potentials for molecular and periodic systems. *Journal of Physical Chemistry A*, 114(37):10225–10233, 2009. URL: https://pubs.acs.org/doi/10.1021/jp103944q, doi:https://doi.org/10.1021/jp103944q.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O',
                                         r_0=2.1*Angstrom,
                                         k=2.0067*1/Ang,
```

(continues on next page)

(continued from previous page)

```
                                          E_0=0.340554*eV,
                                          r_i=6.0*Angstrom,
                                          r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O',
                                                 r_cut=7.5*Angstrom,
                                                 c=1.0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O',
                                          r_0=3.618701*Angstrom,
                                          k=1.379316*1/Ang,
                                          E_0=0.042395*eV,
                                          r_i=6.0*Angstrom,
                                          r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O',
                                                 r_cut=7.5*Angstrom,
                                                 c=22.0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *Repulsive12Potential* defines a purely repulsive potential:

$$V_{ij}(r) = \frac{c_{ij}}{r^{12}} .$$

RestartFromTrajectory

class RestartFromTrajectory(*trajectory_filename=None*, *object_id=None*, *restore_optimizer_state=None*)

Class for restarting a geometry optimization from a previously saved trajectory.

Parameters

- **trajectory_filename** (*str*) – The filename used to look for the restart trajectory.
- **object_id** (*str*) – The ID of the object in the file. Can only be specified together with a *trajectory_filename*.
- **restore_optimizer_state** (*bool*) – Whether to restart using the state of the optimizer method object as well as the atomic positions. **Default:** `True` if the optimizer method object is present on the trajectory and compatible with the new optimizer method, `False` otherwise.

objectId()

Returns
The object ID of the restart trajectory.

Return type
str | None

restoreOptimizerState()

Returns
Whether to restart using the state of the optimizer method object.

Return type
bool | None

trajectoryFilename()

Returns
The filename used to look for the restart trajectory.

Return type
str | None

uniqueString()

Return a unique string representing the state of the object.

Notes

The *RestartFromTrajectory* class is intended to facilitate restarts of *OptimizeGeometry()* and *OptimizeNudgedElasticBand()*, as well as other objects which accept *OptimizeGeometryParameters*, using an existing trajectory file.

If *trajectory_filename* is not specified in *RestartFromTrajectory*, the filename of the current optimization object will be used to search for a compatible trajectory to restart from. In case no trajectory can be found, the restart behavior falls back to `NoRestart`.

If specified, the `object_id` parameter will determine the object in the file that is used for restart. For *OptimizeNudgedElasticBand()* the specified `object_id` is treated as prefix and all matching *NudgedElasticBand* objects in the file are searched for. The one with the highest suffix is used. If the `object_id` parameter is not specified, the `trajectory_object_id` in *OptimizeGeometry()* or *OptimizeNudgedElasticBand()* will be used if specified there. Otherwise the trajectory file will be searched for all objects matching the expected type with numbered IDs, and the highest one will be used. In case no matching object can be found, the restart behavior falls back to `NoRestart`.

Before restarting from a given object, the simulations are checked for consistency, including elements, type of configuration, constraints, optimizer method, and calculator (only for NEB). If the consistency check fails, a warning is printed to the log file and the restart behavior falls back to `NoRestart`.

ReversibleScalingThermodynamicIntegrationHook

class ReversibleScalingThermodynamicIntegrationHook(*total_steps*, *temperature=None*, *equilibration_steps=None*, *lambda_min=None*, *measurement_call_interval=None*)

Post step hook for calculating the temperature dependence of the free energy using reversible scaling

Parameters

- **total_steps** (*int*) – The total number of simulation steps.
- **temperature** (*float*) – The temperature of the system
- **equilibration_steps** (*int*) – The number of steps to equilibrate at the reference state.
- **lambda_min** (*float*) – Minimum value of the switching parameter used during the simulation.

•**measurement_call_interval** (*int*) – The interval between calculating measurement data

callInterval()

Returns
The call interval of this hook function.

Return type
int

reference()

Returns the reference system.

Returns
The reference system

Return type
BaseReference

referenceFreeEnergy()

Returns the free energy of the reference system.

Returns
The enegy of the reference system

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

RigidBody

class RigidBody(*atom_selection*, *apply_to_cartesian_direction=None*, *fix_cartesian_direction=None*)

This constraint type treats a group of atoms as a rigid body with only translational motions during a *MolecularDynamics()* or *OptimizeGeometry()* simulation.

Parameters

•**atom_selection** (*list of ints | list of str | str*) – The list of indices or the list of tags of the atoms whose positions should be constrained.

•**apply_to_cartesian_direction** (*list(3) of bools*) – Select the cartesian axes along which rigid body constraints should be applied. **Default:** [True, True, True]

•**fix_cartesian_direction** (*list(3) of bools*) – Select the cartesian axes along which the center-of-mass of the rigid body should be fixed. This only affects axes selected by apply_to_cartesian_direction. **Default:** [False, False, False]

atomIndices()

Returns
The indices of the rigid-body-atoms. An exception is raised if the constraint has not been initialized with a configuration.

Return type
list of type int

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by the rigid body.

Return type
int

tags()

Returns
The tags used for defining the fixed atoms.

Return type
list of type string

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Optimize the geometry of a silicon-germanium-interface treating both crystal regions as rigid bodies:

```
# ------------------------------------------------------------
# Si-Ge-interface
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 11.588]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Germanium, Germanium, Germanium, Germanium, Germanium,
            Germanium, Germanium, Germanium]

# Define coordinates
fractional_coordinates = [[ 0.            ,  0.            ,  0.            ],
                          [ 0.5           ,  0.5           ,  0.            ],
                          [ 0.25          ,  0.25          ,  0.117159993096],
                          [ 0.75          ,  0.75          ,  0.117159993096],
                          [ 0.5           ,  0.            ,  0.234319986193],
                          [ 0.            ,  0.5           ,  0.234319986193],
                          [ 0.75          ,  0.25          ,  0.351479979289],
                          [ 0.25          ,  0.75          ,  0.351479979289],
                          [ 0.184141715464,  0.            ,  0.51178805661 ],
                          [ 0.684141715464,  0.5           ,  0.51178805661 ],
                          [ 0.434141715464,  0.25          ,  0.633841042458],
                          [ 0.934141715464,  0.75          ,  0.633841042458],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.684141715464,  0.            ,  0.755894028305],
                          [ 0.184141715464,  0.5           ,  0.755894028305],
                          [ 0.934141715464,  0.25          ,  0.877947014153],
                          [ 0.434141715464,  0.75          ,  0.877947014153]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = Tersoff_SiGe_1989()
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# Define a rigid body containing the silicon atoms.
rigid_body_si = RigidBody([0, 1, 2, 3, 4, 5, 6, 7])

# Define a rigid body containing the germanium atoms.
rigid_body_ge = RigidBody([8, 9, 10, 11, 12, 13, 14, 15])

# Optimize the interface under rigid body contraints.
bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.05*eV/Ang,
        max_steps=200,
        max_step_length=0.2*Ang,
        constraints=[rigid_body_si, rigid_body_ge],
        trajectory_filename=None,
        optimize_cell=False,
        optimizer_method=LBFGS(),
        )
```

rigid_body.py

Notes

- *RigidBody* constraints can be used in *OptimizeGeometry* and *MolecularDynamics* simulations. At each optimization or molecular dynamics step, all atoms within a rigid body undergo the same translational displacement, which is determined by the force on the center-of-mass of the rigid body atoms. Rotational motions are completely switched off.
- In optimization simulations, the force on the center-of-mass of each rigid body is minimized, until the `max_force` threshold is reached.
- A rigid body must not share atoms with other rigid bodies, or with the set of constrained atoms.
- It is not recommended to run cell-optimizations (i.e. optimization simulations with `optimize_cell=True`) or NPT

simulations (e.g. *NPTBerendsen*) when rigid body constraints are used.

- With the `fix_cartesian_direction` parameter you can completely fix selected components of the center-of-mass-vector of the rigid body.
- Via the `apply_to_cartesian_direction` parameter you can select along which of the three cartesian directions the atoms are treated as a rigid body. This means that for instance by setting `apply_to_cartesian_direction=[True, True, False]`, all atoms within the rigid body would move cooperatively as a rigid body in x- and y-direction, whereas in z-direction all atoms would be displaced independent of each other, as if no rigid body had been defined on these atoms.

STM

class STM(*tip_configuration*, *sample_configuration*, *voltage=None*, *temperature=None*, *wigner_seitz_radius=None*, *cutoff_distance=None*, *k_point=None*, *energy_step=None*, *infinitesimal=None*, *z_cut_tip=None*, *z_cut_sample=None*)

Constructor for the STM object.

Parameters

- **tip_configuration** (*BulkConfiguration*) – Configuration representing the STM tip. The unit cell of the configuration must be rectangular.
- **sample_configuration** (*BulkConfiguration*) – Configuration representing the STM sample. The unit cell of the configuration must be rectangular.
- **voltage** (*PhysicalQuantity* of type electric potential) – Voltage between tip and sample. **Default:** `1 * Volt`
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature for the thermal smearing of the Fermi distribution. **Default:** `300 * Kelvin`
- **wigner_seitz_radius** (*PhysicalQuantity* of type length) – The Wigner-Seitz radius giving the reference electron density. **Default:** `8 * Bohr`
- **cutoff_distance** (*PhysicalQuantity* of type length) – The cutoff distance for the Gaussian filter of high k vectors. **Default:** `0.25 * Bohr`
- **k_point** (*PhysicalQuantity* of type inverse length) – The k-point at which the STM image is calculated. **Default:** `[0.0, 0.0, 0.0] * Bohr**-1`
- **energy_step** (*PhysicalQuantity* of type energy) – The energy step in energy integral. **Default:** `1.0e-4 * eV`
- **infinitesimal** (*PhysicalQuantity* of type energy) – The broadening parameter of the energy levels. **Default:** `0.05 * eV`
- **z_cut_tip** (*list(2) of ints*) – Two z-indices defining the region within the bulkside electron density to be removed from the tip system. **Default:** `[0, 0]`
- **z_cut_sample** (*list(2) ints*) – Two z-indices defining the region within the bulkside electron density to be removed from the sample system. **Default:** `[0, 0]`

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

– line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

currentSTMDisplacementMatrix()

Returns
The current STM displacement matrix. The shape of the matrix is (*N_a*, *N_b*, *N_c*), where *N_a*, *N_b* and *N_c* are the number of grid points in different Cartesian directions.

Return type
PhysicalQuantity of type current

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length.) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length.) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length.) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The gradient at the specified point for the given spin. An error is raised, if the underlying data does not contain the spin projection of the requested type. In case that Spin.All is used, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z)

components is returned. Note that in the spin unpolarized case the Spin.X, Spin.Y and Spin.Z entries are zero, and for the spin polarized case the Spin.X and Spin.Y are zero.

Return type
PhysicalQuantity

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The value at the specified point for the given spin. An error is raised, if the underlying data does not spin projection of the requested type. In case Spin.All is used, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned. Note that in the spin unpolarized case the Spin.X, Spin.Y and Spin.Z entries are zero, and for the spin polarized case the Spin.X and Spin.Y are zero.

Return type
PhysicalQuantity

evaluateSTMImage(*constant_z=None*, *constant_current=None*, *spin=None*)

Evaluate the STM data in a given operation mode: Constant height or constant current.

Parameters

- **constant_z** (*PhysicalQuantity* of type length) – Select the STM operation mode to be constant height. **Default:** None
- **constant_current** (*PhysicalQuantity* of type current) – Select the STM operation mode to be constant current. **Default:** None
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
In constant height mode, a 2D array with the unit Ampere. In constant current mode, a 3D image with the data given as a NLEngine.RealGrid3D object, is returned.

Return type
PhysicalQuantity with the unit Ampere | NLEngine.RealGrid3D

gridCoordinate(*i*, *j*, *k*)

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

primitiveVectors()

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

sampleConstraintFunction()

Returns
The sample constraint function. The shape of the returned array is (3, *N_a*, *N_b*, *N_c*), where *N_a*, *N_b* and *N_c* are the number of grid points in different Cartesian directions.

Return type
PhysicalQuantity of type inverse length

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

`spin()`

Returns
The spin the STM image is calculated for, always `Spin.Sum`.

Return type
`Spin.Sum`

`spinProjection(spin=None)`

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (`Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown`) – The spin component to project on. **Default:** The spin the object was created with. If the spin was `Spin.All`, `Spin.Sum` will be used for the projection.

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

`temperature()`

Returns
The temperature for the thermal smearing of the Fermi distribution.

Return type
PhysicalQuantity of type temperature

`tipConstraintFunction()`

Returns
The tip constraint function. The shape of the returned array is (3, *N_a*, *N_b*, *N_c*), where *N_a*, *N_b* and *N_c* are the number of grid points in different Cartesian directions.

Return type
PhysicalQuantity of type inverse length

`toArray()`

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

`uniqueString()`

Return a unique string representing the state of the object.

`unit()`

Returns
The unit of the data in the grid.

Return type
A physical unit.

`unitCell()`

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

`voltage()`

Returns
Voltage between tip and sample.

Return type
PhysicalQuantity of type electric potential

`volumeElement()`

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

SaddleSearchParameters

`class SaddleSearchParameters(`*time_step=None*, *langevin_friction=None*, *minimization_period=None*, *escape_time_accuracy=None*, *max_time=None*, *optimizer_method=None*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *min_neb_images=None*, *max_neb_images=None*, *max_neb_steps=None*, *max_neb_forces=None*, *neb_image_distance=None*, *neb_spring_constant=None*, *neb_climbing_image=None*, *lanczos_relative_error=None*, *lanczos_maximum_iterations=None*, *finite_difference=None*, *finite_difference_method=None*, *structure_comparison_tolerance=None*, *structure_comparison_energy_tolerance=None*, *saddle_offset_length=None*, *max_energy_difference=None*)

Class that stores all the parameters associated with saddle searches.

Parameters

- `time_step` (*PhysicalQuantity* of type time) – The time-step interval used in the molecular dynamics simulation. **Default:** 1.0*fs
- `langevin_friction` (*PhysicalQuantity* of type frequency) – The friction (coupling) with the reservoir. **Default:** 5.0*ps**-1
- `minimization_period` (*PhysicalQuantity* of type time) – The time interval between minimizing the molecular dynamics trajectory to determine if the system has left the initial state. **Default:** 400*fs
- `escape_time_accuracy` (*PhysicalQuantity* of type time) – The accuracy with which the escape time should be determined. This quantity must be less than or equal to the minimization period. **Default:** `minimization_period`/3
- `max_time` (*PhysicalQuantity* of type time) – The maximum total length of time the MD simulation is allowed to run for. **Default:** 5000*fs

- **optimizer_method** (*LBFGS* | *FIRE* | *VelocityProjectionOptimization* | ConjugateGradient) – The optimization algorithm to use for the minimizations, saddle optimization, and nudged elastic band calculations. **Default:** *LBFGS*
- **max_forces** (*PhysicalQuantity* of type force) – Optimization is stopped when the maximum force on an atom is less than max_forces. There is a separate convergence criterion (max_neb_forces) for the *NudgedElasticBand* optimization. **Default:** 0.01*eV/Angstrom
- **max_steps** (*int*) – The maximum number of steps during an optimization. There is a separate max steps (neb_max_steps) parameter for *NudgedElasticBand* optimization. **Default:** 400
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Angstrom
- **min_neb_images** (*int*) – The minimum number of *NudgedElasticBand* images in a band. **Default:** 3
- **max_neb_images** (*int*) – The maximum number of *NudgedElasticBand* images in a band. **Default:** 7
- **max_neb_steps** (*int*) – The maximum number of *NudgedElasticBand* optimization steps. **Default:** 100
- **max_neb_forces** (*PhysicalQuantity* of type force) – The *NudgedElasticBand* optimization is stopped when the maximum force on an atom is less than max_neb_forces. **Default:** 0.1*eV/Angstrom
- **neb_image_distance** (*PhysicalQuantity* of type distance) – The distance between the *NudgedElasticBand* images. **Default:** 0.5*Angstrom
- **neb_spring_constant** (*PhysicalQuantity* of type force constant (e.g. eV/Angstrom**2)) – The spring constant used for the *NudgedElasticBand* relaxation. **Default:** 5.0*eV/Angstrom**2
- **neb_climbing_image** (*bool*) – Flag indicating if the climbing image algorithm should be used to find a transition state. **Default:** False
- **lanczos_relative_error** (*float*) – Stop the calculation when the relative change in the eigenvalue is less than this number. **Default:** 0.01
- **lanczos_maximum_iterations** (*int*) – The maximum number of iterations in the Lanczos algorithm. **Default:** 20
- **finite_difference** (*PhysicalQuantity* of type length) – The step size for the finite difference estimation of the second derivatives. **Default:** 0.01*Angstrom
- **finite_difference_method** (*Forward | Central*) – The type of finite difference used to estimate the second derivatives. **Default:** Forward
- **structure_comparison_tolerance** (*PhysicalQuantity* of type length) – The maximum distance mismatch between two atoms to determine if they are the same structure. **Default:** 0.1*Angstrom
- **structure_comparison_energy_tolerance** (*PhysicalQuantity* of type energy) – This energy tolerance is used to determine if two structures are different based upon energy alone. Configurations that are closer than this tolerance a full geometry comparison. **Default:** 0.02*eV
- **saddle_offset_length** (*PhysicalQuantity* of type length) – The distance along the negative mode from the saddle that a configuration is displaced before being minimized to determine if it is connected to the initial minimum. **Default:** 0.2*Angstrom
- **max_energy_difference** (*PhysicalQuantity* of type energy) – If the final state is higher in energy than the initial state by this amount, then the search will be terminated. If the value is None then no energy comparison is performed. **Default:** 10.0*eV

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Setup a saddle search using a tight convergence criterion.

```
saddle_search_parameters = SaddleSearchParameters(
    max_forces=0.001*eV/Angstrom,
    )
```

parameters.py

Notes

These parameters are normally passed to the *AdaptiveKineticMonteCarlo* class.

SaveInMemory

class SaveInMemory

Cache self energies in memory.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Examples on how to use *SaveInMemory()* and other storage strategies can be found in the *Usage Examples* of *RecursionSelfEnergy*.

Notes

In each step of the SCF loop of a device calculation, the same self energies need to be evaluated.

The *SaveInMemory()* storage strategy specifies that each self energy is calculated only once and stored in memory. The self energies are reused in subsequent iterations. This strategy can significantly improve performance, but requires additional memory.

The *StoreOnDisk()* strategy provides a similar performance improvement without the additional memory requirements.

ScatteringStates

class ScatteringStates(*save_scattering_states=None*)

The scattering states method used to calculate the Greens function.

Parameters
save_scattering_states (*bool*) – Whether calculated scattering states should be cached. **Default:** `True`

saveScatteringStates()

Returns
True when the scattering states are saved.

Return type
bool

Usage Examples

Setup a device calculation that computes the non-equilibrium contribution to the contour integral via the scattering states method while computed scattering states are cached.

```
non_equilibrium_method = ScatteringStates(save_scattering_states=True)
device_algorithm_parameters = DeviceAlgorithmParameters(
    non_equilibrium_method=non_equilibrium_method,
    )
calculator = DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    )
```

SecondHarmonicsGenerationSusceptibility

class SecondHarmonicsGenerationSusceptibility(*configuration*, *kpoints=None*, *energies=None*, *broadening=None*, *bands_below_fermi_level=None*, *bands_above_fermi_level=None*, *use_symmetries=None*, *occupation_tolerance=None*, *tensor_indices=None*)

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the optical spectrum.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which to calculate the transitions. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **energies** (*PhysicalQuantity* of type energy) – The energies for which to calculate the second harmonics generation susceptibility. **Default:** `numpy.linspace(0, 4, 101) * eV`
- **broadening** (*PhysicalQuantity* of type energy) – The broadening parameter used for the optical spectrum. **Default:** `0.1 * eV`
- **bands_below_fermi_level** (int | `All`) – The maximum number of valence band states per principal spin channel to include at each k-point. **Default:** All valence bands
- **bands_above_fermi_level** (int | `All`) – The maximum number of conduction band states per principal spin channel to include at each k-point. **Default:** All conduction bands

- **use_symmetries** (*bool*) – Boolean to control if symmetries should be used to reduce k-points and abc-indices. **Default:** True
- **occupation_tolerance** (*float*) – Terms with Fermi occupation differences smaller than the occupation tolerance will be skipped. If one is only interested in the low temperature limit, this can be set to a finite value, like 0.1 in order to speed up calculations. If the temperature set on the calculator should be fully respected, the occupation_tolerance should be set to 0. **Default:** 0.1
- **tensor_indices** (*list of tuples.*) – List of tuples of x,y,z indices (0, 1, or 2) to be calculated, like [(0, 0, 0), (0, 1, 2)] for calculating the (x,x,x) and (x,y,z) component of the SHG tensor. **Default:** By default the list of indices is determined by the symmetries of the system and only those components, which contribute will be calculated. If `use_symmetries=False` all indices will be calculated.

bandsAboveFermiLevel()

Returns
The number of above the Fermi level included in the calculation.

Return type
int

bandsBelowFermiLevel()

Returns
The number of below the Fermi level included in the calculation.

Return type
int

broadening()

Returns
The broadening parameter used for the spectrum.

Return type
PhysicalQuantity of type energy

energies()

Returns
The energies for which to calculate the spectrum.

Return type
PhysicalQuantity of type energy

evaluate(*spin=None*)

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.All`) – The user provided spin parameter. **Default:** `Spin.Sum`

Returns
Returns the total SHG susceptibility tensor. The tensor has the shape (number_of_spins, number_of_enegies, 3, 3, 3) if `spin=Spin.All`, otherwise the shape is (number_of_enegies, 3, 3, 3).

Return type
PhysicalQuantity of type Length / Volt

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

useSymmetries()

Returns
Boolean controlling if symmetries should be applied or not.

Return type
bool

Usage Examples

Calculate the *SecondHarmonicsGenerationSusceptibility* of GaAs using GGA-1/2 exchange-correlation.:

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.6537*Angstrom)

# Define elements
elements = [Gallium, Arsenic]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
#--------------------------------------
# Basis Set
#--------------------------------------
basis_set = [
    GGABasis.Gallium_DoubleZetaPolarized,
    GGABasis.Arsenic_DoubleZetaPolarized,
    ]

#--------------------------------------
# Exchange-Correlation
#--------------------------------------
exchange_correlation = GGAHalf.PBE

k_point_sampling = MonkhorstPackGrid(
    na=13,
    nb=13,
    nc=13,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=45.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=FermiDirac(100.0*Kelvin*boltzmann_constant),
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('GaAs.hdf5', bulk_configuration)

# -----------------------------------------------------------
# Second Harmonics Generation Susceptibility
# -----------------------------------------------------------
kpoint_grid = MonkhorstPackGrid(
    na=14,
    nb=14,
    nc=14,
    )

second_harmonics_generation_susceptibility = SecondHarmonicsGenerationSusceptibility(
    configuration=bulk_configuration,
    kpoints=kpoint_grid,
    energies=numpy.linspace(0, 5, 101)*eV,
    broadening=0.1*eV,
    bands_below_fermi_level=4,
    bands_above_fermi_level=8,
```

(continues on next page)

(continued from previous page)

```
    use_symmetries=True,
    tensor_indices=None,
    )
nlsave('GaAs.hdf5', second_harmonics_generation_susceptibility)
```

gaas_shg.py

Notes

Theory

The optical polarization $P_a(\omega)$ can in general be expressed in terms of responce functions, called susceptibilities, and the total macroscopic electric field $\mathbf{E}(\omega)$[1]:

$$P_a(\omega) = \chi^{(1)}_{ab}(-\omega,\omega)E^b(\omega) + \chi^{(2)}_{abc}(-\omega'-\omega'',\omega',\omega'')E^b(\omega')E^c(\omega'') + ...$$

The first-order, or linear, susceptibility $\chi^{(1)}_{ab}(-\omega,\omega) = \chi^{(1)}_{ab}(\omega)$ is a rank two tensor and available in QuantumATK from an *OpticalSpectrum* calculation.

The second order susceptibility is a rank three tensor and in general it depends on two frequency arguments, ω' and ω''. The second harmonics generation (SHG) susceptibility, $\chi^{(2)}_{SHG}(-2\omega,\omega,\omega)$ is the special case, where $\omega' = \omega''$. This is obviously relevant for studying second harmonic generation in e.g. lasers applications. In the low-frequncy limit $\omega \to 0$, $\chi^{(2)}_{SHG}(-2\omega,\omega,\omega)$ coinsides with the linear electro-optic (LEO) susceptibility $\chi^{(2)}_{LEO}(-\omega,\omega,0)$[2] and can thus be used to obtain the electronic contribution to the low-frequency electro-optic tensor.

The *SecondHarmonicsGenerationSusceptibility* class calculates the second harmonics genreration susceptibility tensor following the works of Sipe et al.[3Page 3789, 2] and Sharma et al.[4]. The methodology is based on perturbation theory within the independent particle approximation.

The total susceptibility can be divided into three terms: The interband transitions $\chi^{(2)}_{inter}(-2\omega,\omega,\omega)$, the intraband transitions $\chi^{(2)}_{intra}(-2\omega,\omega,\omega)$, and a modulation of interband terms by intraband terms, $\chi^{(2)}_{mod}(-2\omega,\omega,\omega)$: such that

$$\chi^{(2)}(-2\omega,\omega,\omega) = \chi^{(2)}_{inter}(-2\omega,\omega,\omega) + \chi^{(2)}_{intra}(-2\omega,\omega,\omega) + \chi^{(2)}_{mod}(-2\omega,\omega,\omega)$$

Detailed expressions for the three terms are given in Eqs. A1-A3 in Ref.[4].

[1] R.ăLeitsmann, W.ăG. Schmidt, P.ăH. Hahn, and F.ăBechstedt. Second-harmonic polarizability including electron-hole attraction from band-structure theory. *Phys. Rev. B*, 71:195209, May 2005. URL: https://link.aps.org/doi/10.1103/PhysRevB.71.195209, doi:10.1103/PhysRevB.71.195209.

[2] James L.ăP. Hughes and J.ăE. Sipe. Calculation of second-order optical response in semiconductors. *Phys. Rev. B*, 53:10751–10763, Apr 1996. URL: https://link.aps.org/doi/10.1103/PhysRevB.53.10751, doi:10.1103/PhysRevB.53.10751.

[3] J.ăE. Sipe and EdăGhahramani. Nonlinear optical response of semiconductors in the independent-particle approximation. *Phys. Rev. B*, 48(16):11705–11722, October 1993. URL: https://link.aps.org/doi/10.1103/PhysRevB.48.11705 (visited on 2019-05-10), doi:10.1103/PhysRevB.48.11705.

[4] S.ăSharma, J.ăK. Dewhurst, and C.ăAmbrosch-Draxl. Linear and second-order optical response of III-V monolayer superlattices. *Phys. Rev. B*, 67:165332, Apr 2003. URL: https://link.aps.org/doi/10.1103/PhysRevB.67.165332, doi:10.1103/PhysRevB.67.165332.

Tensor indices and symmetries

By default, the *SecondHarmonicsGenerationSusceptibility* will be constructed with the parameter `use_symmetries=True` implying that the k-point sampling is reduced to the irreducible part of the Brillouin zone and that the $\chi^{(2)}_{abc}(-2\omega, \omega, \omega)$ tensor is symmetrized. This is the recommended setting since it significantly speeds up the calculation time due to the k-point reduction. The symmetrization of the tensor implies that only certain tensor elements are non-zero. For cubic structures, like GaAs in the script above, only tensor elements of the form $(a, b, c) = (x, y, z)$ i.e. where $a \neq b \neq c$ are non-zero. For other symmetry classes the non-zero elements will be different.

If `use_symmetries=False` by default all tensor elements will be calculated. In that case it is possible to specify which indices to be calculated using the `tensor_indices` input parameters.

Convergence of results

When performing *SecondHarmonicsGenerationSusceptibility* calculations it is important to check that the calculated susceptibility is converged with respect to the numerical parameters. The important parameters to check convergence for are

1)K-point sampling. Increase the k-point sampling to ensure the results are converged.

2)Broadening. The energy broadening should inprinciple be infinitesimally small. Usually a value of `0.1*eV` is reasonable, but results should be checked by lowering the value of the `broadening` parameter. Note that when lowering the `broadening` the k-point sampling needs to be increased in order to have comparable results.

3)Bands above and below the Fermi level. The parameters `bands_below_fermi_level` and `bands_above_fermi_level` determines how many bands are included in the calculation. The calculated spectra needs to be converged wrt. to the number of bands included. Increase the number of bands above and below the Fermi level to verify convergence.

In general these numbers can be chosen by inspecting the band structure and selecting the bands that contribute to an energy range of interest. Reducing the number of included bands significantly speeds up the simulation.

SelfDiffusion

class SelfDiffusion(*md_trajectory*, *start_time=None*, *end_time=None*, *atom_selection=None*, *anisotropy=None*, *time_resolution=None*, *info_panel=None*)

Constructor for the SelfDiffusion object.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the self diffusion for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`.
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** all atoms.
- **anisotropy** (*list of type int | int | None*) – The list of Cartesian directions (x=0, y=1, z=2) to calculate the anisotropic self diffusion in or a single Cartesian direction. By default an isotropic calculation is performed. **Default:** None.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.

data()

Return the self-diffusion values.

times()

Return the time values.

Usage Examples

Load an *MDTrajectory*, and calculate the self-diffusion of all aluminum atoms.

```
# Load the trajectory
md_trajectory = nlread('alumina_trajectory.hdf5')[-1]

# Calculate the self diffusion
self_diffusion = SelfDiffusion(
    md_trajectory,
    atom_selection=Aluminum
)

# Get the times in ps and the MSD values in Ang**2.
times     = self_diffusion.times()
diffusion = self_diffusion.data()

# Plot the diffusion coefficient calculated with different time limits
model = Plot.PlotModel(ps, cm**2/Second)
model.title().setText('Diffusion coefficient of aluminum')
model.xAxis().setLabel('Time')
model.yAxis().setLabel(r'$D_{self}$')
model.legend().setVisible(False)

# Add diffusion integral
line = Plot.Line(times, diffusion)
line.setColor('blue')

# Add average
average = Plot.Average()
line.addItem(average)
max_time = times[-1]
average.setBounds(max_time*0.1, max_time*0.9)
average.setColor('red')

# Add line to the plot
model.addItem(line)

# Set limits and show the plot
model.setLimits()
Plot.show(model)
```

self_diffusion.py

Notes

The *SelfDiffusion* is calculated from the velocity auto-correlation function using the Green-Kubo expression:

$$D_{self} = \frac{1}{3}\int_0^\infty d\tau \langle \mathbf{v}_i(t)\mathbf{v}_i(t+\tau)\rangle$$

Here the ensemble average $< ... >$ averages over both all selected atoms i and time origins t.

The diffusion coefficient can also be calculated with the *MeanSquareDisplacement* analysis by calculating the gradient of the mean-squared displacement. These two analyses provide complementary methods of calculating the diffusion coefficient.

When calculating quantities using Green-Kubo expressions, it is important that the trajectory both contains a short enough time step to be able to determine the decay in the auto-correlation function, as well as a long enough time scale to encompass the auto-correlation time length and provide enough time origins for accurate sampling. When calculating the self diffusion, ideally as the velocity auto-correlation decreases to zero the above integral should converge to a static value at longer time scales. Practically however, longer times in the auto-correlation function corresponds to fewer time origins in the ensemble average. This increases the sampling error and can lead to non-convergence in the velocity auto-correlation functions and subsequently the self diffusion coefficient.

The *SelfDiffusion* analysis method returns the calculated self diffusion coefficient using different time limits. The optimal value of the self diffusion coefficient is usually where the auto-correlation has decayed sufficiently for the integral to converge, and also there are enough time origins for accurate sampling. The auto-correlation time scale can vary significantly depending on the potential acting on the atoms. For instance single atoms on which only non-bonding forces are acting typically have a longer auto-correlation time compared to atoms that have strong molecular bonds. In the molecular case relatively short trajectory time steps of the order of 10fs may be required.

By default, all elements are taken into account, but a specified selection can be given as well. The `atom_selection` parameter accepts an element, a tag name, or a list of atom indices to select which velocities are included in the calculation. This can be useful for when only certain atoms should be included in the analysis, such as when there are constrained atoms in the system that excluded. It is also possible to select to only calculate diffusion in specific cartesian directions using the `anisotropy` argument. This can be useful in cases such as calculating the diffusion rate of atoms between two surfaces.

SemiCircleContour

class SemiCircleContour(*integral_lower_bound=None*, *circle_eccentricity=None*, *logarithmic_bunching=None*, *circle_points=None*, *fermi_line_points=None*, *fermi_function_poles=None*)

An equilibrium contour using the semi-circle contour points and weights defined in[1].

Parameters

- **integral_lower_bound** (*PhysicalQuantity* of type energy) – The distance between the lowest Fermi-level and the lowest energy circle contour point. **Default:** An energy determined on the chosen pseudopotentials or 1.5 Hartree for the semi-empirical calculators.
- **circle_eccentricity** (*float*) – The eccentricity of the circle contour. This should be a float between 0 and 1. 0 is a circle, 1 is a line. **Default:** `0.3`
- **logarithmic_bunching** (*float*) – Logarithmic bunching of the circle contour around the Fermi level. This should be a float between 0 and 1. 0 means no bunching, equidistant points, 1 means all bunched, centred on the Fermi Level. **Default:** `0.3`

[1] M.ăBrandbyge, J.-L. Mozos, P.ăOrdejon, J.ăTaylor, and K.ăStokbro. Density-functional method for nonequilibrium electron transport. *Phys. Rev. B*, 65:165401, Mar 2002. doi:10.1103/PhysRevB.65.165401.

- **circle_points** (*int > 2*) – The number of points on the circle contour. **Default:** 30
- **fermi_line_points** (*int*) – The number of points on the straight line from the Fermi energy level up to infinity. This should be an integer in the range 1 to 11 (inclusive). **Default:** 10
- **fermi_function_poles** (*int > 0*) – The number of poles of the Fermi function to include. Determines the imaginary shift of the straight line from the Fermi level to infinity. **Default:** 8

circleEccentricity()

Returns
The eccentricity of the circle contour.

Return type
float

circlePoints()

Returns
The number of circle points.

Return type
int

fermiFunctionPoles()

Returns
The number of poles for the Fermi function.

Return type
int

fermiLinePoints()

Returns
The number of points on the line from the Fermi energy level up to infinity.

Return type
int

integralLowerBound()

Returns
The distance between the lowest Fermi-level to the lowest energy circle contour point.

Return type
PhysicalQuantity of type energy

logarithmicBunching()

Returns
The logarithmic bunching.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()
Return a unique string representing the state of the object.

Usage Example

One can use the SemiCircleContour by defining it as an equilibrium contour

```
equilibrium_contour = SemiCircleContour()
```

which constructs a SemiCircleContour with all defaults. Alternatively, more parameters that alter the accuracy of the approximation can be specified, e.g.

```
equilibrium_contour = SemiCircleContour(
        circle_eccentricity=0.1,
        logarithmic_bunching=0.2,
        circle_points=100,
        fermi_line_points=11,
        fermi_function_poles=10)
```

To use it in a calculation of the equilibrium density matrix,

the equilibrium contour explicitly on the *ContourParameters* object, the *EquilibriumContour* object is passed to the *ContourParameters* object,

```
contour_parameters = ContourParameters(equilibrium_contour=equilibrium_contour)
```

and saved on the calculator

```
device_calculator = DeviceLCAOCalculator(contour_parameters=contour_parameters)
```

Notes

The SemiCircleContour is a method to calculate the equilibrium density matrix D by performing an integration of the Greens Function G.

$$D = \Im\left[-\frac{2}{\pi}\lim_{R\to\infty}\int_{-R}^{+R} dE f(E-\mu)G(E+i0^{+})\right] ,$$

in which $f(E)$ is the Fermi-Dirac distribution and μ the Fermi level.

This integral can be solved by using the residue theorem:

$$\oint dz G(z)f(z) = -2\pi i k T \sum_{z_\nu} G(z_\nu)$$

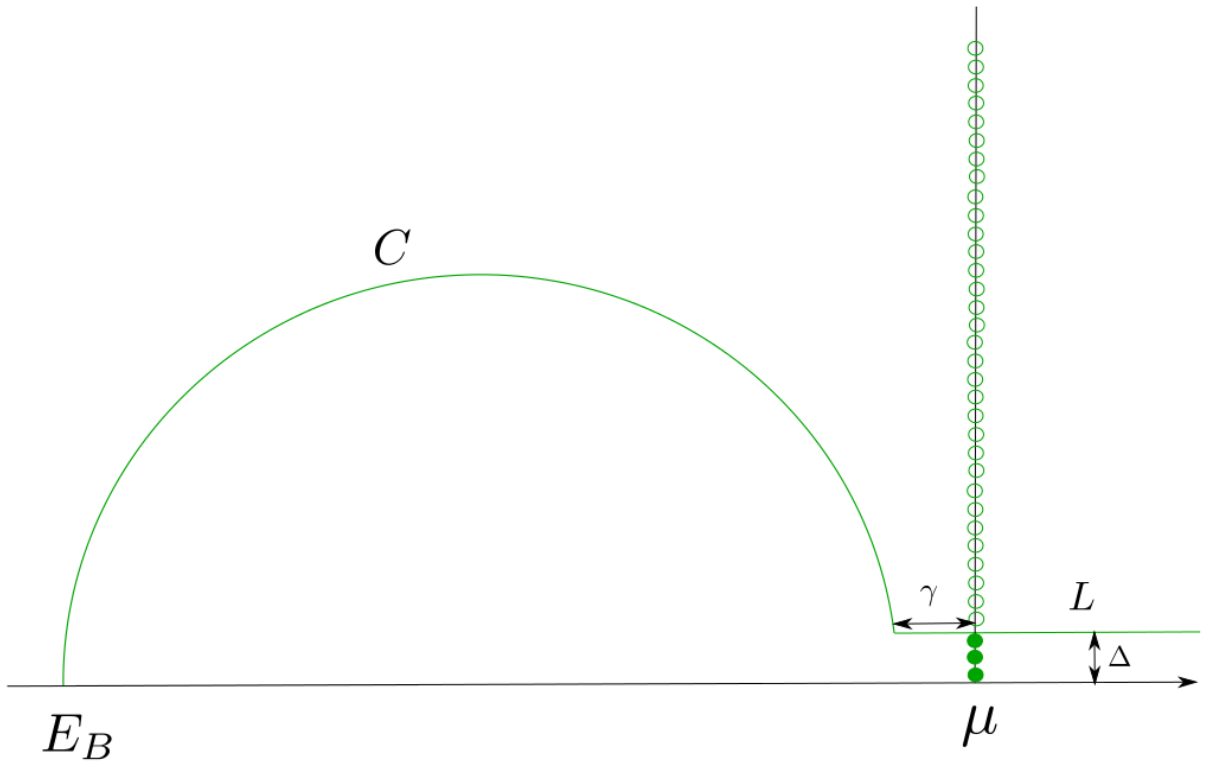
in which the sum on the right hand side runs over the poles of the integrant included in the contour.

The SemiCircleContour defines the contour in the upper-half of the imaginary plane and is comprised of a semicircle, a semi-infinite line segment and a finite number of Fermi poles.

The semicircle C starts from the lower bound E_B controlled by `integral_lower_bound`. The parameter `circle_eccentricity` defines the eccentricity of the semi circle, while adjusting `logarithmic_bunching` alters the distribution of the semicircles contour points around the fermi level. The end point of the semicircle is defined

at a distance γ below the Fermi energy μ and Δ above the real axis. The line segment L runs from the semicircles end point to $+\infty$. The distance Δ between the line segment and the real axis is determined by the number of `fermi_poles_poles` included in the contour.

The precision of the contour integration improves as `circle_points`, the number of contour points on the semicircle, `fermi_line_points`, the number of contour points on the line segment, or `fermi_poles_poles` increases.



More information about this approach can be found in[Page 3792, 1]

SemiEmpiricalCalculator

class SemiEmpiricalCalculator(*hamiltonian_parametrization*, *pair_potentials=None*, *charge=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *poisson_solver=None*, *checkpoint_handler=None*, *spin_polarization=None*, *fixed_spin_moment=None*, *algorithm_parameters=None*, *parallel_parameters=None*)

Class for representing calculations using semi-empirical models for configurations of the type *MoleculeConfiguration* and *BulkConfiguration*.

Parameters

- **hamiltonian_parametrization** (*HuckelHamiltonianParametrization* | *SlaterKosterHamiltonianParametrization* | *NRLHamiltonianParametrization*) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

•**pair_potentials** (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations. **Default:** No pair potential

•**charge** (*float*) – The charge of the system; a charge of -1 corresponds to one extra electron. **Default:** 0.0

•**numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent semi-empirical calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 1),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1.0e-6,
    interaction_max_range=10.0*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(300.0*Kelvin))
```

•**iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self- consistent semi-empirical calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`.

•**poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourierSolver* for a *BulkConfiguration* without any metallic or dielectric `SpatialRegion`.

•**checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop. **Default:** A default *CheckpointHandler* object.

•**spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`

•**fixed_spin_moment** (float | `False`) – Total spin moment (per unit cell) to use, defined as $\Delta N = N\_N$, where N_up and N_down are the number of electrons in the Up and Down spin channels, respectively. When specified the spin moment will be fixed at the given value by introducing separate Fermi levels for the Up and Down spin channels. Can only be specified when doing a calculation with polarized spin. If set to `False` the spin moment will not be fixed - a single Fermi level is used. **Default:** `False`

•**algorithm_parameters** (*AlgorithmParameters*) – The *AlgorithmParameters* used for calculating the density matrix. **Default:**

```
AlgorithmParameters(
    density_matrix_method=DiagonalizationSolver(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    store_energy_density_matrix=Automatic,
    scf_restart_step_length=0.1*Angstrom)
```

•**parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:**

```
ParallelParameters(
    processes_per_neb_image=None,
    processes_per_individual=None,
    processes_per_bias_point=None,
    processes_per_saddle_search=1)
```

algorithmParameters()

Returns
The algorithm parameters.

Return type
AlgorithmParameters

basisSet()

Returns
The basis set associated with the Hamiltonian parametrization.

Return type
BasisSet

charge()

Returns
The charge of the system.

Return type
float

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

fixedSpinMoment()

Get the fixed spin moment.

Returns
The fixed spin moment or `False` if the spin moment is not held fixed.

Return type
float | `False`

hamiltonianParametrization()

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
`SemiEmpiricalHamiltonianParametrization`

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is `NonSelfconsistent`.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfSpins()

Returns
The number of spins.

Return type
int

numericalAccuracyParameters()

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setCheckpointHandler(*checkpoint_handler*)
Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
Unpolarized | Polarized | Noncollinear | SpinOrbit

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible. @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Usage Examples

Perform a self-consistent semi-empirical calculation for a maximum of 20 SCF steps by setting the *IterationControlParameters*.

```
iteration_control_parameters = IterationControlParameters(
    max_steps=20,
    )

# In this example, we use a default Huckel-type Hamiltonian parametrization.
hamiltonian_parametrization = HuckelHamiltonianParametrization()

calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    iteration_control_parameters=iteration_control_parameters,
    )
```

Restart a semi-empirical calculation using the self-consistent state from a previous calculation

```
# Read in the BulkConfiguration with the old SCF state.
old_calculation = nlread('filename.hdf5', BulkConfiguration)[0]

# Define the BulkConfiguration with the same number of atoms and the same elements.
new_calculation = BulkConfiguration(...)

# Define the calculator.
old_calculator = old_calculation.calculator()

# Make a clone of the old calculator.
new_calculator = old_calculator()

# Attach the calculator and use the old initial state.
new_calculation.setCalculator(new_calculator, initial_state=old_calculation)
```

Notes

For the usage and details of the different Hamiltonian parametrizations, see *HuckelHamiltonianParametrization*, *SlaterKosterHamiltonianParametrization* and *NRLHamiltonianParametrization*.

Default is that the semi-empirical calculation is non-self-consistent. To make it self-consistent you must set the *IterationControlParameters*.

SemiGrandCanonicalMonteCarloHook

class SemiGrandCanonicalMonteCarloHook(*elements*, *chemical_potential=None*, *temperature=None*, *swap_ratio=None*, *random_seed=None*, *call_interval=None*)

A hook function that implements semi-grand canonical Monte Carlo simulation

Parameters

- **elements** (Sequence of *PeriodicTableElement*) – The elements being swapped between.
- **chemical_potential** (*PhysicalQuantity* of type energy) – The chemical potential of swapping from the first to the second atom. **Default:** 0 eV
- **temperature** (*PhysicalQuantity* of type temperature) – The Monte Carlo selection temperature. **Default:** 300 Kelvin
- **swap_ratio** (*float*) – The fraction of available atoms that are attempted to be swapped at each call. **Default:** 0.01
- **random_seed** (*int*) – The seed for the random number generator used to select the Monte Carlo moves.
- **call_interval** (*int*) – The number of dynamics steps between each Monte Carlo step.

callInterval()

Returns
The call interval of this hook function.

Return type
int

chemicalPotential()

Returns
The chemical potential difference used in calculating swap probabilities.

Return type
PhysicalQuantity of type energy

elements()

Returns
List of the elements being swapped in the Monte Carlo step.

Return type
list of *PeriodicTableElement*

monteCarloQuantity()

The value being averaged in the Monte Carlo simulation. In this simulation this is the fraction of the second atom in the configuration.

Returns
The ratio of second atoms to the first in the configuration.

Return type
float

`randomSeed()`

Returns
The random seed used to create the Monte Carlo random number generator.

Return type
int

`swapElementCount(`*configuration*`)`

Count the number of atoms that have been swapped to the second element.

Parameters
configuration (*BulkConfiguration*) – The current configuration.

Returns
The number of elements

Return type
int

`swapRatio()`

Returns
The fraction of atoms to attempt to swap in each Monte Carlo cycle.

Return type
float

`temperature()`

Returns
The temperature of the simulation.

Return type
PhysicalQuantity of type temperature

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

In this example the semi-grand canonical Monte Carlo method is used to study the composition of a gold-silver alloy. The chemical potential is chosen to be close to the change in free energy of the two materials. The result of the calculation is a *ConfigurationDataContainer* that contains the composition at each step as a measurement. Averaging this value over the second half of the simulation should give a composition of approximately 23% silver and 77% gold.

A workflow for the calculation can be downloaded `here` with a corresponding python script `Au_Ag_MonteCarlo.py`. The simulation may take around an hour to run, given the number of samples required.

Notes

The *SemiGrandCanonicalMonteCarloHook* class implements Monte Carlo simulation in the semi-grand canonical ensemble. In this ensemble the number of particles, the difference in chemical potential, temperature and pressure are held constant. This allows for swapping identities of particles in the simulation with the given change in chemical potential. Volume changes are also allowed when controlling the pressure.

In a semi-grand canonical Monte Carlo simulation the acceptance probability of swapping one element for another is given as:

$$p_{acc}(o \rightarrow n) = \min\left[1, \exp\left(\frac{-\Delta E - \Delta\mu N \Delta x}{k_b T}\right)\right]$$

Here ΔE is the change in energy between the old and new states, $\Delta\mu$ is the change in the chemical potential between the two elements, N is the number of atoms able to be exchanged and Δx is the change in the fractional composition of the elements.

The semi-grand canonical ensemble is useful for studying the thermodynamics of alloys[1]. Using this method the average composition between two elements can be calculated at a specific chemical potential. Calculating the composition at a range of chemical potentials can then give the relationship between the composition and the chemical potential. In a binary this is related to the free energy such that:

$$\Delta\mu(x, T) = \frac{\partial G(x, T)}{\partial x}$$

Here T is the temperature and G is the free energy. Integrating the chemical potential with respect to the composition then gives the free energy. This can be then used to determine the thermodynamics of mixing of the two materials, and if the elements are miscible under the given conditions.

Semi-grand canonical Monte Carlo simulations are normally carried out as part of a hybrid Monte Carlo simulation using the *HybridMonteCarlo()* method. The dynamics steps handle sampling the relative positions of the atoms, while the Monte Carlo steps sample possible elemental configurations. In creating the class the first argument, `elements`, takes a tuple of the two elements being exchanged during the simulation. Other elements are assumed not to be able to be exchanged. This can be used to simulate the thermodynamics of exchanging one element for another in a ternary material or an oxide. Note also that the order of the atoms matters in this argument. The results of the simulation are generally expressed with reference to the second element. So the difference in chemical potential is given as the change in chemical potential in going from the first to the second element. The composition is also expressed as the fraction of the second element in the configuration.

The `swap_ratio` argument is used to specify the number of swap moves performed in each Monte Carlo cycle. This is expressed as a fraction of the number of atoms available to be swapped. Similarly the `call_interval` argument specifies the number of dynamics steps between a Monte Carlo cycle.

SemiGrandCanonicalTransitionMatrixHook

class SemiGrandCanonicalTransitionMatrixHook(*elements*, *chemical_potential=None*, *temperature=None*, *swap_ratio=None*, *random_seed=None*, *call_interval=None*, *pnpd_update_interval=None*, *measurement_interval=None*)

A hook function that implements semi-grand canonical Monte Carlo simulation

Parameters

- **elements** (Sequence of *PeriodicTableElement*) – The elements being swapped between.

[1] Kyeongpung Lee, Yutack Park, and Seungwu Han. Ab initio construction of full phase diagram of MgO-CaO eutectic system using neural network interatomic potentials. *Phys. Rev. Materials*, 6:113802, Jun 2022. URL: https://journals.aps.org/prmaterials/abstract/10.1103/PhysRevMaterials.6.113802, doi:10.1103/PhysRevMaterials.6.113802.

•**chemical_potential** (*PhysicalQuantity* of type energy) – The chemical potential of swapping from the first to the second atom. **Default:** 0 eV

•**temperature** (*PhysicalQuantity* of type temperature) – The Monte Carlo selection temperature **Default:** 300 Kelvin

•**swap_ratio** (*float*) – The fraction of available atoms that are attempted to be swapped at each call. **Default:** 0.01

•**random_seed** (*int*) – The seed for the random number generator used to select the Monte Carlo moves.

•**call_interval** (*int*) – The number of dynamics steps between each Monte Carlo step. **Default:** 1

•**pnpd_update_interval** (*int*) – The number of Monte Carlo steps between each update of the particle number probability distribution. Note that this is based on Monte Carlo steps, not dynamics steps. **Default:** 1

•**measurement_interval** (*int*) – The number of dynamics steps between recording the particle number probability distribution. **Default:** 1

callInterval()

Returns
The call interval of this hook function.

Return type
int

chemicalPotential()

Returns
The chemical potential difference used in calculating swap probabilities.

Return type
PhysicalQuantity of type energy

collectionMatrix()

Returns
The tri-banded collection in which sampled transition probabilities are collected.

Return type
ndarray

elements()

Returns
List of the elements being swapped in the Monte Carlo step.

Return type
list of *PeriodicTableElement*

measurementInterval()

Returns
The number of dynamics steps between writing data to the trajectory.

Return type
int

monteCarloQuantity()

The value being averaged in the Monte Carlo simulation. In this simulation this is the fraction of the second atom in the configuration.

Returns
The ratio of second atoms to the first in the configuration.

Return type
float

particleNumberProbabilityDistribution()

Returns
The relative probability of being in each composition state.

Return type
ndarray

pnpdUpdateInterval()

Returns
The number of dynamics steps between measurements of the PNPD.

Return type
int

randomSeed()

Returns
The random seed used to create the Monte Carlo random number generator.

Return type
int

sampleCount()

Returns
The number of transition samples from each state.

Return type
ndarray

swapElementCount(*configuration*)
Count the number of atoms that have been swapped to the second element.

Parameters
configuration (*BulkConfiguration*) – The current configuration.

Returns
The number of elements

Return type
int

swapRatio()

Returns
The fraction of atoms to attempt to swap in each Monte Carlo cycle.

Return type
float

temperature()

Returns
The temperature of the simulation.

Return type
PhysicalQuantity of type temperature

`uniqueString()`

Return a unique string representing the state of the object.

In this example the semi-grand canonical transition matrix Monte Carlo method is used to study the composition of a gold-silver alloy. The result of the calculation is a *ConfigurationDataContainer* that contains the particle number probability distribution (PNPD) for the system. Since these metals are miscible, the PNPD should have a single maximum at the average composition value of around 25% gold and 75% silver.

A workflow for the calculation can be downloaded `here` with a corresponding python script `Au_Ag_Transition_Matrix.py`. This simulation may take around several hours to run, given the number of samples required.

Notes

The *SemiGrandCanonicalTransitionMatrixHook* defines a transition matrix Monte Carlo simulation in the semi-grand canonical ensemble. This is a complimentary technique to the *SemiGrandCanonicalMonteCarloHook* method. Both simulations are carried out in the semi-grand canonical ensemble and can be used to determine the composition of materials at different chemical potentials. While in normal Monte Carlo methods the property of interest is directly sampled, in transition matrix Monte Carlo the probability of transitioning from one state to another is sampled[1]. Here this is the number of each element in the material. As only one atom is swapped at a time, the probability of being in any state can be calculated. This is known as the *particle number probability density* (PNPD). In order to effectively calculate the PNPD it is necessary to sample all states. This is done by biasing the selection by the negative change in the PNPD. Since the PNPD is needed to effectively sample the states, this is calculated and updated during the simulation. Once the converges the simulation samples evenly across all states.

The PNPD relates straightforwardly to the relative free energy of each state such that:

$$W(N;\mu,V,T) = k_B T \ln \Pi(N;\mu,V,T)$$

Here $\Pi(N;\mu,P,T)$ is the PNPD at given particle number N and chemical potential, temperature and pressure and W is similarly the free energy. To get the average system property from the PNPD, that property is averaged over the available states. For instance, the average number of elements in the material $\langle N \rangle$ can be given as:

$$\langle N \rangle = \frac{\sum N \Pi(N;\mu,P,T)}{\sum \Pi(N;\mu,P,T)}$$

The PNPD can also be re-weighted to different chemical potential values. This allows exploring the whole range of chemical potentials with one simulation. The PNPD at a different chemical potential can be given as:

$$\ln \Pi(N;\mu,P,T) = \ln \Pi(N;\mu_0,P,T) + \frac{N(\mu-\mu_0)}{k_B T} + C$$

Here μ_0 is the chemical potential that the simulations was originally carried out at, μ is the new chemical potential and C is a normalization constant.

One of the main advantages of transition matrix simulations is that it samples the entire space of compositions, which works well with systems that have multiple stable states. In the case of immiscible metals, there can be a minimum in the free energy close to each pure state, and then a large free energy barrier as the elements mix. As the chemical potential changes the relative energies of the minima change, leading to one being more populated than the other. This situation is very difficult to model with traditional Monte Carlo techniques, but can be explored with transition Matrix techniques. The trade-off here is that transition Matrix Monte Carlo methods require at least an order of magnitude more steps than a traditional Monte Carlo simulation to reach convergence. This is due to the transition Matrix simulation scanning the entire space of configurations rather than just a small amount around a minimum in the free energy.

[1] DanielăW. Siderius and VincentăK. Shen. Use of the Grand Canonical Transition-Matrix Monte Carlo Method to Model Gas Adsorption in Porous Materials. *J. Phys. Chem. C*, pages 5861, Jun 2013. URL: https://pubs.acs.org/doi/10.1021/jp400480q, doi:10.1021/jp400480q.

To practically carry out a transition Matrix Monte Carlo simulation, it is recommended that the chemical potential should be near the difference in free energy between the two materials. The PNPD stored in the trajectory is the natural log of the PNPD, as this is numerically much easier to handle. In addition to the arguments to the arguments from the *SemiGrandCanonicalMonteCarloHook*, the *SemiGrandCanonicalTransitionMatrixHook* has two additional arguments that control the frequency of updating and storing the PNPD. Frequent updating of the PNPD can slow the simulation in some cases. This is controlled by the argument `pnpd_update_interval`, which gives the number of Monte Carlo cycles between PNPD updates. Similarly `measurement_interval` gives the frequency with which the PNPD is written to the trajectory. This should be done often enough so that convergence in the PNPD can be observed, but not so much as to take large amounts of space in the trajectory. This frequency is also based on the number of dynamics steps between recording each PNPD. As well as the PNPD the trajectory also records the number of samples from each state. This is stored as the measurement `sample_count`. For a converged PNPD each state should be sampled enough times to have an accurate probability of transitioning from that state.

SequentialImageDependentPairPotential

class SequentialImageDependentPairPotential(*spring_constant=None*, *constraints=None*, *break_symmetry=None*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *optimizer_method=None*, *memory_size=None*)

Use the Image Dependent Pair Potential method for interpolating images.

Parameters

- **spring_constant** (*PhysicalQuantity* of type force per length) – The spring constant used between the images. **Default:** 1.0*eV/Ang**2.
- **constraints** (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained. **Default:** []
- **break_symmetry** (*bool*) – If the symmetry should be broken. This can be necessary for interpolation to work and to prevent unphysically close atoms if the endpoints are symmetry equivalent. **Default:** False.
- **max_forces** (*PhysicalQuantity* of type eV/Ang) – The maximum forces when the optimization should stop. **Default:** 0.05*eV/Ang.
- **max_steps** (*int*) – The maximum number of optimization steps to take after adding an image. **Default:** 200.
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.
- **optimizer_method** (FIRE | *LBFGS.*) – The optimizer to use for optimizing the structure. **Default:** LBFGS

Returns
The generated images.

Return type
list of *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* classes

breakSymmetry()

Returns
True, if the symmetry should be broken.

Return type
bool

constraints()

Returns
The constrained atoms.

Return type
list of type int

`interpolate`(*neb*)

Interpolate the given NEB and set the new images on it. The interpolation algorithm on the returned NEB is replaced by the regular Image Dependent Pair Potential algorithm.

Parameters
neb (NudgedElasticBand) – The NEB to interpolate.

Returns
The interpolated NEB or None if the interpolation failed.

Return type
NudgedElasticBand | None

`maxForces`()

Returns
The maximum forces when the optimization should stop.

Return type
PhysicalQuantity of type eV/Ang

`maxStepLength`()

Returns
The maximum step length the optimizer may take.

Return type
PhysicalQuantity of type length

`maxSteps`()

Returns
The maximum number of optimization steps to take after adding an image.

Return type
int

`memorySize`()

Returns
The memory size which is used for the LBFGS method.

Return type
int

`optimizerMethod`()

Returns
The optimizer to use for optimizing the structure.

Return type
FIRE | *LBFGS*

`setConstraints`(*constraints*)

Set the constraints.

Parameters
constraints (*sequence of ints | list of str | str*) – List of atom indices that are kept fixed during interpolation or the list of tags of the atoms whose positions should be constrained.

springConstant()

Returns
The spring constant used between the images.

Return type
PhysicalQuantity of type force per length

tags()

Returns
The tags used for defining the constraints.

Return type
bool

uniqueString()
Return a unique string representing the state of the object.

Notes

The standard *ImageDependentPairPotential* (IDPP) improves upon *LinearInterpolation* of *NudgedElasticBand* by avoiding close atoms as a result of rotations or translations. However, the IDPP itself starts from a linear initial path which can lead to unphysical configurations for complex systems with many moving atoms.

The sequential image dependent pair potential[1] addresses this shortcoming of the IDPP by building the path sequentially, starting from the end points and adding images one by one.

ShieldingTensors

class ShieldingTensors(*configuration*, *finite_difference_q=None*)
Class for calculating and representing NMR shielding tensors using the Gauge Including Projector Augmentated Wave (GIPAW) method.

> **Warning:** It does not work for metals, only semiconductors and insulators.

Parameters

- **configuration** (*BulkConfiguration*) – Configuration with a calculator that supports GIPAW calculations.
- **finite_difference_q** (*PhysicalQuantity*) – The magnitude of the reciprocal space displacement to use when evaluating the finite difference q derivative in the GIPAW method. Must be a *PhysicalQuantity* of dimension inverse length. **Default:** 0.01 Bohr^(-1)

[1] Yorick L.ăA. Schmerwitz, Vilhjálmur Ásgeirsson, and Hannes Jónsson. Improved initialization of optimal path calculations using sequential traversal over the image-dependent pair potential surface. *Journal of Chemical Theory and Computation*, 20(1):155–163, 2024. URL: https://doi.org/10.1021/acs.jctc.3c01111, doi:10.1021/acs.jctc.3c01111.

components()

Get the components to the shielding tensors for all atoms.

Returns
A dictionary with the components of shielding tensors for the requested atoms.

Return type
dict

evaluate()

Get the calculated total shielding tensors for all atoms.

Returns
A list of shielding tensors for the requested atoms.

Return type
list

evaluateAnisotropicChemicalShielding()

Determine the anisotropic chemical shielding for each atom. Using Haeberlen convention:

```
sigma_anisotropy = sigma_{zz} - (sigma_{xx}+sigma_{yy})/2
                 = 3 * reduced_sigma_anisotropy / 2
```

Where:

```
|sigma_{zz}-sigma_iso| > |sigma_{yy}-sigma_iso| > |sigma_{xx}-sigma_iso|
```

Returns
The anisotropic chemical shielding for each atom.

Return type
list

evaluateAsymmetryParameter()

Determine the asymmetry parameter for each atom. Using Haeberlen convention:

eta = (sigma_{xx} - sigma_{yy})/reduced_sigma_anisotropy

Where:

```
|sigma_{zz}-sigma_iso| > |sigma_{yy}-sigma_iso| > |sigma_{xx}-sigma_iso|
```

Returns
The asymmetry parameter for each atom.

Return type
list

evaluateIsotropicChemicalShielding()

Determine the isotropic chemical shielding for each atom.

Returns
The isotropic chemical shielding for each atom.

Return type
list

evaluatePrincipalAxis()

Determine the principel axis (eigenvectors and eigenvalues of the shielding tensor) for each atom. Using Haeberlen convention:

```
|sigma_{zz}-sigma_iso| > |sigma_{yy}-sigma_iso| > |sigma_{xx}-sigma_iso|
```

Returns
The principel axis for each atom.

Return type
two ndarray

evaluateReducedAnisotropicChemicalShielding()

Determine the reduced anisotropic chemical shielding for each atom. Using Haeberlen convention: reduced_sigma_anisotropy = sigma_{zz} - sigma_iso

Where:

```
|sigma_{zz}-sigma_iso| > |sigma_{yy}-sigma_iso| > |sigma_{xx}-sigma_iso|
```

Returns
The anisotropic chemical shielding for each atom.

Return type
list

evaluateSkew()

Determine the skew parameter for each atom. Using Herzfeld-Berger convention:

Skew = 3*(sigma_{22} - sigma_{iso})/span (-1 =< Skew =< 1)
= 3*(sigma_{22} - sigma_{iso})/(sigma_{11} - sigma_{33})

Where: sigma_{11} > sigma_{22} > sigma_{33} (std. convention)

Returns
The span parameter for each atom.

Return type
list

evaluateSpan()

Determine the span parameter for each atom. Using Herzfeld-Berger convention:

Span = (sigma_{11} - sigma_{33}) (Span >= 0)

Where: sigma_{11} > sigma_{22} > sigma_{33} (std. convention)

Returns
The span parameter for each atom.

Return type
list

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

ShockleyReadHallRecombination

class ShockleyReadHallRecombination(*charged_point_defect*, *initial_charge_state*, *final_charge_state*, *supercell_repetitions*, *initial_quantum_numbers*, *final_quantum_number*, *fractional_displacements=None*, *electron_phonon_coupling_configurations=None*, *electron_phonon_coupling_k_point=None*, *rotate_to_pure_spin_states=None*, *supercell_scaling_correction=None*, *filename=None*, *object_id=None*, *number_of_processes_per_task=None*, *log_filename_prefix=None*, *save_displaced_configurations=None*)

Perform a study of the Shockley-Read-Hall (SRH) recombination between an initial and a final defect charge state.

Parameters

- **charged_point_defect** (*ChargedPointDefect*) – The charged point defect study for which to extract the initial and final charge state from.
- **initial_charge_state** (*int*) – The initial charge state of the defect, as a discrete multiple of elementary charge. A charge of `-1` corresponds to one extra electron.
- **final_charge_state** (*int*) – The final charge state of the defect, as a discrete multiple of elementary charge. A charge of `-1` corresponds to one extra electron.
- **supercell_repetitions** (*sequence (size 3) of int*) – The supercell of the (charge point defect study) bulk unit cell given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **initial_quantum_numbers** (*sequence of int (nonnegative)*) – The quantum numbers for the initial states contributing to the calculation of the electron-phonon coupling matrix element.
- **final_quantum_number** (*int (nonnegative)*) – The quantum number for the final state in the calculation of the electron-phonon coupling matrix element.
- **fractional_displacements** (*sequence of float*) – The fractional displacements from the center displacement configuration in terms of the generalized configuration coordinate delta. Both the initial and final charge state configuration will be the center displacement configuration. **Default:** `[-0.02, -0.01, 0.0, 0.01, 0.02]`
- **electron_phonon_coupling_configurations** (`InitialChargeState` | `FinalChargeState` | [InitialChargeState, FinalChargeState]) – Determine whether the configuration used to evaluate the electron-phonon coupling matrix element is the initial charge state configuration or the final charge state configuration or that both can be evaluated. **Default:** `[InitialChargeState, FinalChargeState]`

•**electron_phonon_coupling_k_point** (*list(3) of floats*) – The k-point of the electronic states used to evaluate the electron phonon coupling matrix element, in fractional coordinates. **Default:** The k-point from the configuration calculator if only one k-point is present, `[0.0, 0.0, 0.0]`

•**rotate_to_pure_spin_states** (*bool*) – Boolean to control if the eigenstates should be rotated to pure spin states or not when evaluating the electron-phonon coupling matrix elements. This is only relevant for non-collinear or spin-orbit calculations. **Default:** `True`

•**supercell_scaling_correction** (`Disabled` | *ShockleyReadHallSupercellScalingCorrection*) – Specifies the parameters for calculating the scaling factor for the interaction between the defect and the bulk wavefunction when the defect has a finite charge. `Disabled` means no scaling factors are calculated. **Default:** `Disabled`

•**filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*. **Default:** The file name of the input charge point defect study.

•**object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file. **Default:** `'srh_recombination_' + initial_charge_state + '_' + final_charge_state`

•**number_of_processes_per_task** (int | `ProcessesPerNode`) – The number of processes that will be used to execute each task. If the total number of processes does not divide evenly into the tasks, some tasks may have less than this number of processes. **Default:** `AllProcesses`, all available processes execute each task collaboratively.

•**log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of SRH recombination study. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'srhrecombination_'`

•**save_displaced_configurations** (*bool*) – Whether the displaced updated configuration used to determine the total energies should be saved to disk. If `True`, they can be inspected or reused, possibly saving computational time if running again the study, or if shared with other studies. However the file where the study is saved will be significantly larger. **Default:** `False`

calculateCaptureCoefficient(*temperature=None*, *degeneracy_factor=None*, *charge_state_id=None*, *initial_quantum_numbers=None*, *gaussian_broadening=None*, *sommerfeld_temperature_scaling_factor=None*, *effective_mass=None*, *dielectric_constant=None*, *include_supercell_scaling_factor=None*, *spin=None*)

Calculate the capture coefficient.

Parameters

•**temperature** – The temperature(s) at which the capture coefficient is evaluated. **Default:** 300 * Kelvin

•**degeneracy_factor** (*int*) – The configurational degeneracy of the final state. **Default:** 1

•**charge_state_id** (`InitialChargeState` | `FinalChargeState`) – Determine whether the configuration used to evaluate the electron-phonon coupling matrix element is the initial charge state configuration or the final charge state configuration. **Default:** If the electron phonon coupling is only calculated for one configuration, use the one available configuration. Otherwise use *FinalChargeState*.

•**initial_quantum_numbers** (*sequence of int (nonnegative)*) – The quantum numbers for the initial states contributing to the calculation of the electron-phonon coupling matrix element. Note that root sum square of all contributions from individual initial quantum numbers is used. **Default:** All available initial quantum numbers.

•**gaussian_broadening** (*PhysicalQuantity* of type energy) – The standard deviation of the gaussian broadening used instead of a delta function in the phonon selection rules. **Default:** the final state oscillator energy.

•**sommerfeld_temperature_scaling_factor** (`Analytical` | `Numerical` | `Disabled`) – Whether the temperature scaling via inclusion of the Sommerfeld parameter should be considered, and if so whether the analytical or numerical expression should be used. **Default:** `Disabled`

•**effective_mass** (*PhysicalQuantity* of type mass.) – The effective mass of the carrier. Only used if sommerfeld_scaling_factor is not `Disabled`. **Default:** 1 * electron_mass

•**dielectric_constant** (*float*) – The relative dielectric constant of the host material. Only used if sommerfeld_scaling_factor is not Disabled. **Default:** The value provided in the charged point defect.

•**include_supercell_scaling_factor** (*bool*) – If True, the squared electron-phonon coupling is multiplied by a supercell scaling factor which corrects for distortion in the bulk state due to attractive or repulsive interaction with the localized charged defects state. **Default:** True.

•**spin** (Spin.Up | Spin.Down) – The spin component to calculate the electron-phonon matrix element for. Only used for polarized. For non-collinear and spin-orbit Spin.All is returned.

static calculateCaptureCoefficientFromParameters(*supercell_volume*, *generalized_configuration_coordinate_delta*, *energy_difference*, *electron_phonon_coupling*, *initial_state_effective_vibration_frequency*, *final_state_effective_vibration_frequency*, *degeneracy_factor=1*, *temperature=PhysicalQuantity(300.0, K)*, *gaussian_broadening=None*, *sommerfeld_temperature_scaling_factor=None*, *defect_charge_ratio=-1*, *effective_mass=PhysicalQuantity(1.0, me)*, *dielectric_constant=1.0*)

Utility to calculate the capture coefficient using raw physical parameters as input.

Parameters

•**supercell_volume** (*PhysicalQuantity* of type volume.) – The volume of the defect supercell.

•**generalized_configuration_coordinate_delta** (*PhysicalQuantity* of type sqrt(mass) * length) – The displacement between the initial and final state in generalized configuration coordinates.

•**energy_difference** (*PhysicalQuantity* of type energy.) – The energy difference between excited state and ground state, i.e. the distance between valence band edge and transition state level for hole recombination or the distance between the transition state level and conduction band edge for electron recombination.

•**electron_phonon_coupling** (*PhysicalQuantity* of type energy / sqrt(mass) / length) – The electron-phonon coupling between the initial and final state.

•**initial_state_effective_vibration_frequency** (*PhysicalQuantity* of type frequency) – The effective frequency of the harmonic oscillator associated with the initial state.

•**final_state_effective_vibration_frequency** (*PhysicalQuantity* of type frequency) – The effective frequency of the harmonic oscillator associated with the final state.

•**degeneracy_factor** (*PhysicalQuantity* of type temperature | array of *PhysicalQuantity* of type temperature) – The configurational degeneracy of the final state. **Default:** 1

•**temperature** – The temperature(s) at which the capture coefficient is evaluated. **Default:** 300 * Kelvin

•**gaussian_broadening** (*PhysicalQuantity* of type energy) – The standard deviation of the gaussian broadening used instead of a delta function in the phonon selection rules. **Default:** the final state oscillator energy.

•**sommerfeld_temperature_scaling_factor** (Analytical | Numerical | Disabled) – Whether the temperature scaling via inclusion of the Sommerfeld parameter should be considered, and if so whether the analytical or numerical expression should be used. **Default:** Disabled

•**defect_charge_ratio** (*float*) – The ratio between the defect charge (before recombination, i.e. the initial charge state), and the carrier charge. Only used if sommerfeld_scaling_factor is not Disabled.

•**effective_mass** (*PhysicalQuantity* of type mass.) – The effective mass of the carrier. Only used if sommerfeld_scaling_factor is not Disabled. **Default:** 1 * electron_mass

- **dielectric_constant** (*float*) – The relative dielectric constant of the host material. Only used if sommerfeld_scaling_factor is not Disabled. **Default:** 1.0

calculateEffectiveVibrationFrequency(*charge_state_id*)

Calculate the effective vibration frequency of the charge state.

Parameters
charge_state_id (InitialChargeState | FinalChargeState) – The charge state for which the effective vibration frequency is calculated.

Returns
The frequency of the effective phonon mode associated with the given charge state, evaluate by quadratic fit of the total energy as a function of the displacement in generalized configuration coordinates.

Type
PhysicalQuantity of type frequency.

calculateElectronPhononCouplingMatrixElement(*initial_quantum_number*, *charge_state_id*, *spin=None*)

Calculate the electron-phonon coupling matrix element.

Parameters

- **initial_quantum_number** (*int*) – The quantum number of the bulk-like state for which the electron-phonon coupling is calculated.
- **charge_state_id** (InitialChargeState | FinalChargeState) – Determine whether the configuration used to evaluate the electron-phonon coupling matrix element is the initial charge state configuration or the final charge state configuration.
- **spin** (Spin.Up | Spin.Down | Spin.All) – The spin component to calculate the matrix element for. Only used for polarized. For non-collinear and spin-orbit Spin.All is returned. **Default:** Spin.Up

Returns
The electron-phonon coupling matrix element.

Return type
PhysicalQuantity of type energy over length square root mass.

chargedPointDefect()

Returns
The charged point defect study from which the initial and final charge state is extracted from.

Return type
ChargedPointDefect

configurationCoordinateDelta()

Returns
The calculated change in the configuration coordinate. If not available, returns None.

Return type
PhysicalQuantity of type length | None

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

displacedConfigurationGeneralizedConfigurationCoordinate(*charge_state_id*, *fractional_displacement*)

Retrieve the generalized configuration coordinate for the displaced configuration.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The generalized configuration coordinate for the displaced configuration. If not available, returns None.

Return type
PhysicalQuantity of type length square root mass | None

displacedConfigurationTotalEnergy(*charge_state_id*, *fractional_displacement*)

Retrieve the total energy for the displaced configuration.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The calculated total energy analysis object for the displaced configuration. If not available, returns None.

Return type
TotalEnergy | None

electronPhononCouplingConfigurations()

Returns
A list of flags identifying the charge state configurations for which the electron-phonon coupling matrix element can be evaluated.

Return type
list of InitialChargeState and/or FinalChargeState

electronPhononCouplingKPoint()

Returns
The k-point of the electronic states used to evaluate the electron phonon coupling matrix element, in fractional coordinates.

Return type
list(3) of floats

filename()

Returns
The filename where the study object is stored.

Return type
str

finalChargeState()

Returns
The final charge state of the defect, as a discrete multiple of elementary charge. A charge of -1 corresponds to one extra electron.

Return type
int

finalQuantumNumber()

Returns
The quantum number for the final state in the calculation of the electron-phonon coupling matrix element.

Return type
int (nonnegative)

fractionalDisplacements()

Returns
The fractional displacements from the center displacement configuration in terms of the generalized configuration coordinate delta.

Return type
sequence of float

generalizedConfigurationCoordinateDelta()

Returns
The calculated change in the generalized configuration coordinate. If not available, returns None.

Return type
PhysicalQuantity of type length square root mass | None

huangRhysFactor(*charge_state_id*)

Evaluate the dimensionless Huang-Rhys factor for the effective vibrational mode associated with a given charge state:

$$S_{i,f} = \frac{1}{2\hbar}(\Delta Q)^2 \Omega_{i,f}$$

Parameters
charge_state_id (InitialChargeState | FinalChargeState) – The charge state for which the Huang-Rhys factor is calculated.

Returns
The calculated Huang-Rhys factor. If not available yet, returns None.

Return type
float | None

initialChargeState()

Returns
The initial charge state of the defect, as a discrete multiple of elementary charge. A charge of -1 corresponds to one extra electron.

Return type
int

initialQuantumNumbers()

Returns
The quantum numbers for the initial states contributing to the calculation of the electron-phonon coupling matrix element.

Return type
sequence of int (nonnegative)

`logFilenamePrefix()`

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

`nlinfo()`

Returns
Structured information about the Study.

Return type
dict

`nlprint(stream=None)`
Print a string containing an ASCII table useful for plotting the Study object.

Parameters
`stream` (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`numberOfProcessesPerTask()`

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

`numberOfProcessesPerTaskResolved()`

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

`objectId()`

Returns
The name of the study object in the file.

Return type
str

`rotateToPureSpinStates()`

Returns
Whether the eigenstates should be rotated to pure spin states or not when evaluating the electron-phonon coupling matrix elements.

Return type
bool

saveDisplacedConfigurations()

Returns
Whether the displaced configurations should be saved.

Return type
bool

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

spins()

Returns
The available spin components.

Return type
list of Spin.Up | Spin.Down | Spin.All

supercellRepetitions()

Returns
The supercell of the (charge point defect study) bulk unit cell given as the number of repetitions of the bulk unit cell along the (a, b, c) directions.

Return type
sequence (size 3) of int

supercellScalingCorrection()

Returns
Specifies the parameters for calculating the scaling factor for the interaction between the defect and the bulk wavefunction when the defect has a finite charge. Disabled means no scaling factors are calculated.

Return type
ShockleyReadHallSupercellScalingCorrection

supercellScalingCorrectionFactors(*charge_state_id*, *spin=None*)

Retrieve the supercell scaling correction factors.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state configuration for which the supercell scaling correction factors is retrieved.
- **spin** (Spin.Up | Spin.Down | Spin.All) – The spin component to extract the scaling factors for. **Default:** Spin.Up

Returns
Retrieve the supercell scaling correction factors as a list. If not available, returns None.

Return type
list of float | None

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

updatedDisplacedConfiguration(*charge_state_id*, *fractional_displacement*)

Retrieve the updated displaced configuration, if available.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The charge state of the center displacement configuration from which the displaced configuration is displaced from. Either the charge state of the initial configuration or the final configuration.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
The updated displaced configuration. If not available, returns None.

Return type
BulkConfiguration | None

wavefunctionOverlapsAndEnergyDifferences(*charge_state_id*, *fractional_displacement*)

Retrieve the wavefunction overlaps and energy differences.

Parameters

- **charge_state_id** (InitialChargeState | FinalChargeState) – The configuration for which the overlap integrals and energy differences for the evaluation of electron-phonon coupling have been calculated.
- **fractional_displacement** (*float*) – The fragment displacement from the center displacement configuration. Must be among the fractional displacements.

Returns
Retrieve the wavefunction overlaps and energy differences as a container object. If not available, returns None.

Return type
WavefunctionOverlapsAndEnergyDifferences | None

Notes

The *ShockleyReadHallRecombination* study object allows to calculate the capture rate associated to a Shockley-Read-Hall (SRH) non-radiative recombination mediated by a trap state, following the the methodology developed by Alkauskas et al.[1].

The carrier capture is characterized by a transition between an initial and a final charge states in a deep level defect, therefore an accurate study of the defect itself is required beforehand.

This analysis can be performed using a *ChargedPointDefect*, whose results can be in turn used as input quantities to the *ShockleyReadHallRecombination* study.

Note: A very accurate charged point defect study is required to obtain physical capture rates. In particular, the defect configuration should be relaxed to tight accuracy, and all simulation parameters should be accurately chosen (e.g. exchange-correlation functional) and well converged.

The *ShockleyReadHallRecombination* object performs 3 main operations:

- Total energy analysis of the charge state configurations in generalized configuration coordinates, to determine the vibrational properties of the system.
- Evaluation of electron-phonon coupling matrix elements between the carrier bulk states and the defect trap state.

[1] Audrius Alkauskas, Qimin Yan, and ChrisăG. Vanăde Walle. First-principles theory of nonradiative carrier capture via multiphonon emission. *Phys. Rev. B*, 90:075202, Aug 2014. URL: https://link.aps.org/doi/10.1103/PhysRevB.90.075202, doi:10.1103/PhysRevB.90.075202.

•Post-processing to evaluate capture rates based on those quantities.

Choice of initial and final charge state

The *ShockleyReadHallRecombination* requires information about the initial and final defect charge states involved in the carrier capture, specified in the input parameters `initial_charge_state` and `final_charge_state`. For example, the process of capturing a hole by a ionized defect state with a transition level 0/-1 in gap is identified by `initial_charge_state=-1` and `final_charge_state=0`. Hence, the choice of those parameters is dictacted by the process which must be simulated and the defect transition state levels available for that process.

Choice of initial and final quantum numbers

In order to calculate the capture rate, the electron-phonon coupling matrix element between the initial bulk-like carrier state and the final localized trap state must be evaluated. To this end, Kohn-Sham states from the defect configuration are utilized. In particular, knowledge of the quantum number of a state representative of a localized defect state in gap, and of the quantum numbers of the initial bulk-like states, are required.

Unluckily, there is no automated way to determine those states, and some inspection of the *ChargedPointDefect* is required. For each charge state it should be inspected whether a localized state in gap exists. If this is the case, the corresponding quantum number can be selected as `final_quantum_number`. The bulk-like states will be specified using the parameter `initial_quantum_numbers`. In principle, both the initial and final charge state configurations can be used, as long as a well distinguished localized defect state in gap is present. However, a neutral charge state configuration (if any) should be preferred, since it will circumvent artifacts due to the finite supercell size.

An example of how such parameters can be determined is given in the section below, and a theoretical discussion can be found in ref.[Page 3820, 1].

Scaling factors

The nominal capture rate is determined as in equation 22 in ref.[Page 3820, 1]. This quantity can be scaled to take into account temperature effects and supercell size effects. The former scaling factor is introduced when evaluating the capture rate through the method `calculateCaptureCoefficient()`, while the latter is only calculated for non-neutral charge states and requires an extra calculation, controlled by the input parameter *supercell_scaling_correction*.

The temperature scaling is determined as described in reference[2], while the evaluation of the supercell scaling correction follows the methodology implemented in Nonrad by by Turianski et al.[3]. By definition, the supercell scaling correction is set to 1 for the neutral charge state.

[2] R.ăPassler. Relationships between the nonradiative multiphonon carrier-capture properties of deep charged and neutral centres in semiconductors. *physica status solidi (b)*, 78(2):625–635, 1976. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pssb.2220780222, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/pssb.2220780222, doi:https://doi.org/10.1002/pssb.2220780222.

[3] MarkăE. Turiansky, Audrius Alkauskas, Manuel Engel, Georg Kresse, Darshana Wickramaratne, Jimmy-Xuan Shen, CyrusăE. Dreyer, and ChrisăG. Vanăde Walle. Nonrad: Computing nonradiative capture coefficients from first principles. *Comput. Phys. Commun.*, 267:108056, October 2021. doi:10.1016/j.cpc.2021.108056.

Usage example

This example performes a study of hole capture rate in Gallium Nitride with Carbon substitutional defects. The calculation runs in about one day on a single node with 40 cores, and calculation parameters have been chosen to trade off accuracy and execution time. Results for the same type of capture mechanism are reported in[Page 3820, 1], [Page 3821, 3], but please note that for a meaningful comparison consistent simulation parameters should be chosen.

First, the charged point defect is evaluated as in the script below. A `MGGA.SCAN` functional is used for geometry relaxations and formation energy calculations. The transition state levels are corrected using an `HybridGGA.HSE` functional. Such combination provides results comparable with literature at a fraction of the execution time which would be required for a full calculation using hybrid functional. Optimization parameters tighter than the default will be required to evaluate reliably the vibrational properties.

The *ChargedPointDefect* can include multiple charge states, in this case we know beforehand that we are interested in the charge states -1 and 0.

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(5.546690740087016*Angstrom, 3.2093295699579034*Angstrom, 5.
↪241822783047967*Angstrom)

# Define elements
elements = [Gallium, Gallium, Nitrogen, Nitrogen, Gallium, Gallium, Nitrogen,
            Nitrogen]

# Define coordinates
fractional_coordinates = [[ 0.338086773247, -0.            ,  0.005917824105],
                          [ 0.661913226753,  0.            ,  0.505917824105],
                          [ 0.329542526042,  0.            ,  0.379082175895],
                          [ 0.670457473958, -0.            ,  0.879082175895],
                          [ 0.838086773247,  0.5           ,  0.005917824105],
                          [ 0.161913226753,  0.5           ,  0.505917824105],
                          [ 0.829542526042,  0.5           ,  0.379082175895],
                          [ 0.170457473958,  0.5           ,  0.879082175895]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Exchange-Correlation
#---------------------------------------
```

(continues on next page)

(continued from previous page)

```
exchange_correlation = MGGA.SCAN

basis_set = [
    BasisGGAPseudoDojo.Nitrogen_Medium,
    BasisGGAPseudoDojo.Gallium_Medium,
    BasisGGAPseudoDojo.Carbon_Medium,
    ]

k_point_sampling = KpointDensity(
    density_a=4.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=95.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    exchange_correlation=MGGA.SCAN,
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('charge_point_defect.hdf5', bulk_configuration)

# ------------------------------------------------------------
# Charged Point Defect
# ------------------------------------------------------------
# Defect
point_defect = Substitutional(
    element=Carbon,
    site_index=2,
    unit_cell_index=(0, 0, 0))

# Charge states
charge_states = [-1, 0]

# Atomic chemical potentials
atomic_chemical_potentials = [
    AtomicChemicalPotential(element=Carbon, internal_energy=None),
    AtomicChemicalPotential(element=Nitrogen, internal_energy=None)
]

# Supercell repetitions
supercell_repetitions_list = [
    [2, 3, 2],
]

# File name.
filename = 'charge_point_defect.hdf5'
```

(continues on next page)

(continued from previous page)

```
band_gap_calculator = calculator(exchange_correlation=HybridGGA.HSE06)

charged_point_defect = ChargedPointDefect(
    bulk_configuration=bulk_configuration,
    formation_energy_calculator=calculator,
    band_gap_calculator=band_gap_calculator,
    filename=filename,
    object_id='charged_point_defect',
    point_defect=point_defect,
    charge_states=charge_states,
    atomic_chemical_potentials=atomic_chemical_potentials,
    supercell_repetitions_list=supercell_repetitions_list,
    relax_atomic_coordinates=True,
    dielectric_constant=8.9,
    log_filename_prefix='charged_point_defect_',
    number_of_processes_per_task=None,
    elastic_correction_method=None,
    optimize_geometry_parameters=OptimizeGeometryParameters(max_forces=0.01*eV/Angstrom),
    random_seed=1
)
charged_point_defect.update()
```

charged_point_defect.py

Once the calculation is completed, we can verify in the labfloor that the transition level (-1/0) is located at 1.03 eV above Valence band, in agreement with reported computational results.

Before setting up the *ShockleyReadHallRecombination* object, we inspect the charged point defect to determine the electronic quantum numbers required to evaluate the electron-phonon coupling matrix element. To this end, we evaluate the *ElectronicInverseParticipationRatio* (IPR), which gives an indication of the degree of eigenstate localization in the system.

The following snippet extracts the configurations from the charged point defect study and calculated the IPR for both charge states.

```
from SMW import *

cpd = nlread('charged_point_defect.hdf5', ChargedPointDefect)[-1]

defect_configuration = cpd.defectConfiguration(
    (2, 3, 2),
    charge_state=0)

ipr = ElectronicInverseParticipationRatio(defect_configuration)
nlsave('defect_configuration_0.hdf5', ipr)

defect_configuration = cpd.defectConfiguration(
    (2, 3, 2),
    charge_state=-1)

ipr = ElectronicInverseParticipationRatio(defect_configuration)
nlsave('defect_configuration_-1.hdf5', ipr)
```

`ipr_evaluation.py`

The result can be visualized on the labfloor, a plot like the one below is visualized:

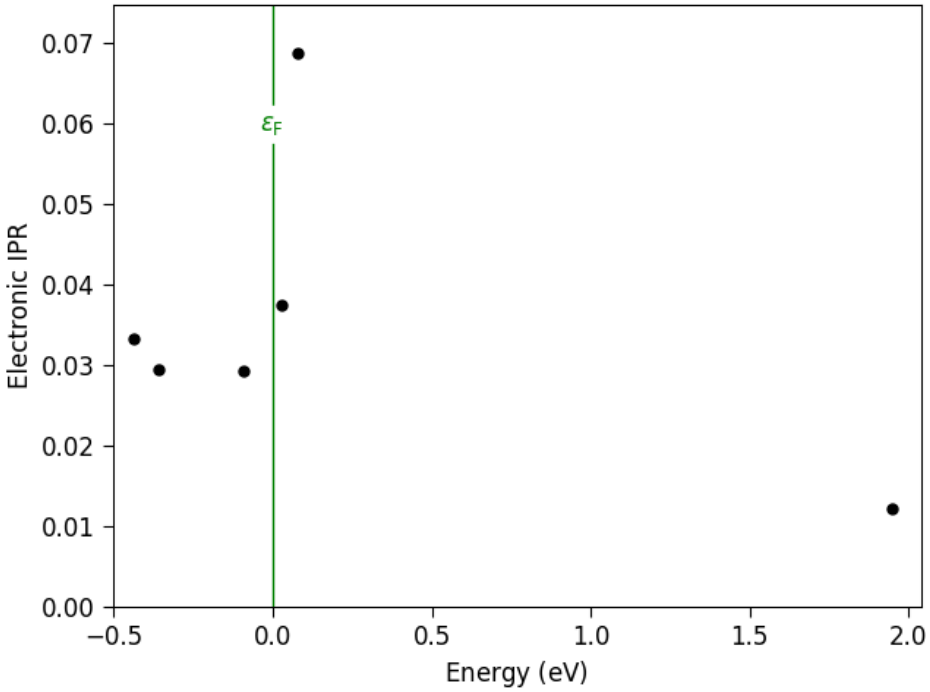


Fig. 4.36: Electronic Inverse Participation Ratio for the neutral charge state.

We identify a strongly localized state in gap. The state on the right corresponds to the conduction band edge, while the states below the localized one are bulk-like states, partially localized by the interaction with the defect. The *ElectronicInverseParticipationRatio* allows us to identify the defect state with the quantum number *431*, and as bulk-like states we choose the three states below, since in the ideal bulk the recombination would involve the three nearly degenerate top valence bands.

Note: A *FatBandstructure* with projection on the defect site a and *BlochState* for visual inspection can be utilized to further inspect the Kohn-Sham states and decide whether a localized defect state can be identified.

In this specific case, we could identify the defect and bulk-like states in the neutral charge state configuration.

Once this preliminary analysis of the charged point defect is done, we can setup and run the *ShockleyReadHallRecombination* object as shown in the script below.

```
from SMW import *

charged_point_defect = nlread('charged_point_defect.hdf5', ChargedPointDefect)[-1]

initial_quantum_numbers = [430, 429, 428]
final_quantum_number = 431

srh = ShockleyReadHallRecombination(
    charged_point_defect,
    initial_charge_state=-1,
```

(continues on next page)

(continued from previous page)

```
    final_charge_state=0,
    supercell_repetitions=charged_point_defect.supercellRepetitionsList()[-1],
    initial_quantum_numbers=initial_quantum_numbers,
    final_quantum_number=final_quantum_number,
    fractional_displacements=[-0.05, -0.025, 0., 0.025, 0.05],
    object_id='srh-medium-scan-hse',
    filename='shockley_read_hall.hdf5',
    log_filename_prefix='shockley_read_hall')

srh.update()
```

shockley_read_hall.py

The results can be inspected using `nlprint`, as shown below, or the getters implemented on the object.

```
+-------------------------------------------------------------------------------+
| Shockley-Read-Hall (SRH) Recombination Report                                 |
+-------------------------------------------------------------------------------+
| Initial charge state:            -1                                           |
| Effective vibration frequency:    5.04e+01  meV                               |
| Huang-Rhys factor:                1.91e+01                                    |
+-------------------------------------------------------------------------------+
| Initial quantum number    Electron-phonon coupling    Supercell scaling       |
|                           (eV / Å amu^0.5)            correction              |
| 430                       1.10e-02                    n/a                     |
| 429                       6.84e-03                    n/a                     |
| 428                       6.74e-02                    n/a                     |
+-------------------------------------------------------------------------------+
| Final charge state:               0                                           |
| Effective vibration frequency:    5.13e+01  meV                               |
| Huang-Rhys factor:                1.94e+01                                    |
+-------------------------------------------------------------------------------+
| Initial quantum number    Electron-phonon coupling    Supercell scaling       |
|                           (eV / Å amu^0.5)            correction              |
| 430                       6.60e-03                    1.00e+00                |
| 429                       2.99e-03                    1.00e+00                |
| 428                       4.00e-02                    1.00e+00                |
+-------------------------------------------------------------------------------+
| Final quantum number:                          431                            |
| Generalized configuration coordinate delta:    1.78e+00  Å amu^0.5            |
| Transition state level (from valence band):    1.03e+00  eV                   |
| Supercell volume:                              1.12e+03  Å^3                  |
| Supercell repetitions:                         2 x 3 x 2                      |
+-------------------------------------------------------------------------------+
```

Alternatively, convenience methods can be used to generate plots. `_nlplotConfigurationDiagram` visualizes the total energy diagram in generalized configuration coordinates, while `_nlplotOverlapIntegrals` and `_nlplotEigenvalueDifferences` visualize the components utilized to calculate the electron-phonon coupling matrix element.

```
# Visualize the configuration coordinate diagram.
srh._nlplotConfigurationDiagram()
# Visualize the overlap integrals for a given charge state.
```

(continues on next page)

(continued from previous page)

```
srh._nlplotOverlapIntegrals(charge_state_id=FinalChargeState)
# Visualize the eigenvalue differences for a given charge state.
srh._nlplotEigenvalueDifferences(charge_state_id=FinalChargeState)
```

The capture rate can be calculated at different temperaturs using the method `calculateCaptureCoefficient()`. *ShockleyReadHallRecombination* implements also a static utility `calculateCaptureCoefficientFromParameters()` to calculate the capture rate by giving explicitely each physical parameter, e.g. for sensitivity studies or to override some of the parameters obtained from the study object calculation.

ShockleyReadHallRecombinationRate

class ShockleyReadHallRecombinationRate(*bulk_configuration*, *field_direction*, *kpoints=None*, *temperature=None*)

A class for calculating the Shockley-Read-Hall (SRH) recombination rate and velocity.

Parameters

- **bulk_configuration** (*BulkConfiguration*) – The bulk configuration to calculate the mobility which then again is used to calculate the thermal and band velocities.
- **field_direction** (*list of 3 floats | numpy.ndarray of length 3*) – The direction of the velocity in Cartesian coordinates.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid*) – The k-points for which to calculate the density of states. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature to use for the carrier occupation function. **Default:** 300.0 * Kelvin

calculateRecombinationRate(*voltage*, *trap_energy*, *trap_density*, *capture_coefficient_electron=None*, *capture_coefficient_hole=None*, *capture_cross_section_electron=None*, *capture_cross_section_hole=None*)

Calculate the recombination rate.

Parameters

- **voltage** (*PhysicalQuantity* of type voltage) – The applied voltage.
- **trap_energy** (*PhysicalQuantity* of type energy) – The trap energy level relative to the valence band maximum.
- **trap_density** (*PhysicalQuantity* of type inverse volume | *PhysicalQuantity* of type inverse surface) – The trap density.
- **capture_coefficient_electron** (*PhysicalQuantity* of type volume/time) – The capture coefficient for electrons.
- **capture_coefficient_hole** (*PhysicalQuantity* of type volume/time) – The capture coefficient for holes.
- **capture_cross_section_electron** (*PhysicalQuantity* of type surface.) – The capture cross section for electrons.
- **capture_cross_section_hole** (*PhysicalQuantity* of type surface.) – The capture cross section for holes.

Returns
The excess carrier density, the recombination rate and the recombination velocity.

Return type
(*PhysicalQuantity* of type 1/time, *PhysicalQuantity* of type 1/time, *PhysicalQuantity* of type length/time | None)

fieldDirection()

Parameters
field_direction (*list of 3 floats | numpy.ndarray of length 3*) – The direction of the velocity in Cartesian coordinates.

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

temperature()

Returns
The temperature used for the carrier occupation function.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

ShockleyReadHallSupercellScalingCorrection

class ShockleyReadHallSupercellScalingCorrection(*cutoff=None*, *upper_limit=None*)

Parameters

- **cutoff** (*float*) – Tolerance for determining the plateau. **Default:** 0.01
- **upper_limit** (*float*) – Upper limit for the radial integration. **Default:** 5.0

cutoff()

Returns
Tolerance for determining the plateau.

Return type
float

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

upperLimit()

Returns
Upper limit for the radial integration.

Return type
float

SingleContour

class SingleContour(*direction=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*)

A simple flag-like class representing either a left or right single contour.

Parameters
direction (NLFlag.Left | NLFlag.Right) – The direction to use in this *SingleContour*. **Default:** NLFlag.Left

direction()

Returns
The directions.

Return type
NLFlag.Left | NLFlag.Right

Notes

For systems with a finite bias there are equivalent ways to obtain the density matrix (see *Notes*). In principle the left and right estimates of the density matrix are equivalent, but differ in practice because of the approximate integration. The SingleContour object is used to specify which one to use.

When:

```
contour_parameters = ContourParameters(method=SingleContour(direction=Left))
```

is specified, the density matrix is calculated as:

$$D = D^L + \Delta^R \, .$$

If, however:

```
contour_parameters = ContourParameters(method=SingleContour(direction=Right))
```

is specified, the left non-equilibirum contribution is used:

$$D = D^R + \Delta^L \, .$$

SlaterKosterCalculator

class SlaterKosterCalculator(*basis_set*, *pair_potentials=None*, *charge=None*, *numerical_accuracy_parameters=None*, *iteration_control_parameters=None*, *poisson_solver=None*, *checkpoint_handler=None*, *spin_polarization=None*, *fixed_spin_moment=None*, *dynamical_matrix_parameters=None*, *algorithm_parameters=None*, *hamiltonian_derivatives_parameters=None*, *parallel_parameters=None*)

Class for representing calculations using a Slater-Koster tightbinding model for configurations of the type MoleculeConfiguration and BulkConfiguration.

Parameters

- **basis_set** (*SlaterKosterTable* | *DFTBDirectory* | *HotbitDirectory*) – An object describing the basis set used for the SlaterKoster calculation.
- **pair_potentials** (*DFTBDirectory* | *HotbitDirectory* | list of *PairPotential*) – The repulsive pair potentials used for total energy and force calculations **Default:** No pair potential
- **charge** (*float*) – The charge of the system; a charge of -1 corresponds to one extra electron. **Default:** `0.0`
- **numerical_accuracy_parameters** (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent SlaterKoster calculation. **Default:**

```
NumericalAccuracyParameters(
    density_mesh_cutoff=10.0*Hartree,
    k_point_sampling=MonkhorstPackGrid(1, 1, 1),
    radial_step_size=0.01*Angstrom,
    density_cutoff=1.0e-6,
    interaction_max_range=10.0*Angstrom,
    number_of_reciprocal_points=1024,
    reciprocal_energy_cutoff=1250.0*Hartree,
    occupation_method=FermiDirac(300.0*Kelvin))
```

- **iteration_control_parameters** (*IterationControlParameters*) – The *IterationControlParameters* used for the self-consistent SlaterKoster calculation. For non-self-consistent calculations set this parameter to `NonSelfconsistent`. **Default:** `NonSelfconsistent`
- **poisson_solver** (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential. **Default:** Configuration dependent. *FastFourierSolver* for a *BulkConfiguration* without any metallic or dielectric `SpatialRegion`.
- **checkpoint_handler** (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the SCF-loop. **Default:** A default *CheckpointHandler* object.
- **spin_polarization** (`Unpolarized` | `Polarized` | `Noncollinear` | `SpinOrbit`) – Flag indicating if the calculation is spin-polarized or not. **Default:** `Unpolarized`
- **fixed_spin_moment** (float | `False`) – Total spin moment (per unit cell) to use, defined as $\Delta N = N\_N$, where N_up and N_down are the number of electrons in the Up and Down spin channels, respectively. When specified the spin moment will be fixed at the given value by introducing separate Fermi levels for the Up and Down spin channels. Can only be specified when doing a calculation with polarized spin. If set to `False` the spin moment will not be fixed - a single Fermi level is used. **Default:** `False`
- **dynamical_matrix_parameters** (*not used*) – **Deprecated:** from v2015, see the *DynamicalMatrix* analysis object.
- **algorithm_parameters** (*AlgorithmParameters*) – The *AlgorithmParameters* used for calculating the density matrix. **Default:**

```
AlgorithmParameters(
    density_matrix_method=DiagonalizationSolver(),
    store_grids=True,
    store_basis_on_grid=Automatic,
    store_energy_density_matrix=Automatic,
    scf_restart_step_length=0.1*Angstrom,
    use_symmetries=False)
```

•**hamiltonian_derivatives_parameters** (*not used*) – **Deprecated:** from v2015, see the *HamiltonianDerivatives* analysis object.

•**parallel_parameters** (*ParallelParameters*) – The parameters used to control parallelization options. **Default:**

```
ParallelParameters(
    processes_per_neb_image=None,
    processes_per_individual=None,
    processes_per_bias_point=None,
    processes_per_saddle_search=1)
```

algorithmParameters()

Returns
The algorithm parameters.

Return type
AlgorithmParameters

basisSet()

Returns
The basis set associated with the Hamiltonian parametrization.

Return type
BasisSet

charge()

Returns
The charge of the system.

Return type
float

checkpointHandler()

Returns
The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

Return type
CheckpointHandler

dynamicalMatrixParameters()

This method is deprecated.

fixedSpinMoment()

Get the fixed spin moment.

Returns
The fixed spin moment or `False` if the spin moment is not held fixed.

Return type
float | False

hamiltonianDerivativesParameters()

This method is deprecated.

hamiltonianParametrization()

Returns
The Hamiltonian parametrization associated with a semi-empirical calculator.

Return type
SemiEmpiricalHamiltonianParametrization

isConverged()

Returns
True when the call to update() resulted in a converged SCF loop.

Return type
bool

iterationControlParameters()

Returns
The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

Return type
IterationControlParameters

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfSpins()

Returns
The number of spins.

Return type
int

numericalAccuracyParameters()

Returns
The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

Return type
NumericalAccuracyParameters

pairPotentials()

Returns
The repulsive pair potentials used for total energy and force calculations.

Return type
PairPotential

parallelParameters()

Returns
The parameters used to control parallelization options.

Return type
ParallelParameters

poissonSolver()

Returns
The Poisson solver used to determine the electrostatic potential.

Return type
DirectSolver | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*

setBasisSet(*basis_set*)

Set the basis set.

Parameters
basis_set (*SlaterKosterTable* | *DFTBDirectory* | *HotbitDirectory*) – An object describing the basis set used for the calculation.

setCheckpointHandler(*checkpoint_handler*)

Set the the checkpoint handler.

Parameters
checkpoint_handler (*CheckpointHandler*) – The *CheckpointHandler* used for specifying the save-file and the time interval. between saving the calculation during the scf-loop.

setHamiltonianParametrization(*hamiltonian_parametrization*)

Set and check the Hamiltonian parametrization.

Parameters
hamiltonian_parametrization (HamiltonianParametrization) – An object describing the Hamiltonian parametrization for the semi-empirical calculation.

setIterationControlParameters(*iteration_control_parameters*)

Set the iteration control parameters.

Parameters
iteration_control_parameters (*IterationControlParameters*) – The *IterationControlParameters* used for a self-consistent calculation. For non-self-consistent calculations this parameter is NonSelfconsistent.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setNumericalAccuracyParameters(*numerical_accuracy_parameters*)

Set the numerical accuracy parameters.

Parameters
numerical_accuracy_parameters (*NumericalAccuracyParameters*) – The *NumericalAccuracyParameters* used for the self-consistent Huckel calculation.

setPairPotentials(*pair_potentials*)

Set the pair potentials.

Parameters
pair_potentials (*PairPotential*) – The repulsive pair potentials used for total energy and force calculations.

setParallelParameters(*parallel_parameters*)

Set the parallel paramters.

Parameters
parallel_parameters (*ParallelParameters*) – The parameters used to control parallelization options.

setPoissonSolver(*poisson_solver*)

Set the poisson solver.

Parameters
poisson_solver (*DirectSolver* | *MultigridSolver* | *FastFourierSolver* | *FastFourier2DSolver*) – The Poisson solver used to determine the electrostatic potential.

setSpinPolarization(*spin_polarization*)

Set the spin polarization.

Parameters
spin_polarization (Unpolarized | Polarized | Noncollinear | SpinOrbit) – Flag indicating if the calculation is spin-polarized or not.

spinPolarization()

Returns
Flag indicating if the calculation is spin-polarized or not.

Return type
Unpolarized | Polarized | Noncollinear | SpinOrbit

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Private method for updating the calculator from the configuration, if it is possible. @private

versionUsed()

Returns
The version of ATK used to update the calculator.

Return type
str

Attention: The *SlaterKosterCalculator* is being deprecated. Use the *SemiEmpiricalCalculator* with the *SlaterKosterHamiltonianParametrization* instead.

Usage Examples

Define a spin polarized *SlaterKosterCalculator* with a Carbon pi model

```
# Generate a basis set for a Carbon p-p Pi model
# The p-p Pi model is described using s orbitals
# Parameters from: Hancock et. al., J. Low Temp Phys. 153, 393 (2008)
carbon_onsite = SlaterKosterOnsiteParameters(
    element=Carbon,
    angular_momenta=[0],
    occupations=[1],
    ionization_potential=[0.0]*eV,
    onsite_hartree_shift=ATK_U(Carbon, ['2p']),
    onsite_spin_split=ATK_W(Carbon, ['2p'])
    )

# Specify the distances for which the matrix elements are specified
distances = [1.422, 2.463 ,2.844, 3.1]*Ang

# Specify the p-p Pi matrix elements
pp_pi = [[-2.7], [-0.2], [-0.18], [0.0]]*eV
Carbon_Basis = SlaterKosterTable(
    carbon_carbon_sss=zip(distances,pp_pi),
    carbon=carbon_onsite)
    calculator=SlaterKosterCalculator(basis_set = Carbon_Basis,
    spin_polarization=True
    )
```

Restart a Slater-Koster calculation using the self-consistent state from a previous calculation

```
# Read in the BulkConfiguration with the old SCF state
old_calculation=nlread("filename.nc",BulkConfiguration)[0]

# Define the BulkConfiguration with similar number of atoms
new_calculation=BulkConfiguration(...)

# extract the old calculator
old_calculator = old_calculation.calculator()

# make a clone of the old calculator
new_calculator = old_calculator()

# Attach the calculator and use the old initial state
new_calculation.setCalculator(new_calculator, initial_state=old_calculation)
```

Notes

For the details of the Slater-Koster model, see the chapter on *Semi Empirical*.

SlaterKosterHamiltonianParametrization

class SlaterKosterHamiltonianParametrization(*basis_set*)

A class representing the hamiltonian parametrization for a generic Slater-Koster-type Tight Binding model

Parameters
basis_set (*SlaterKosterTable* | *DFTBDirectory* | *HotbitDirectory*) – An object describing the basis set used for the Slater-Koster calculation.

basisSet()

Returns the Slater Koster basis set used to construct the environmental independent component of the Hamiltonian and Overlap.

Returns
The basis set.

Return type
SlaterKosterTable

pairPotentials()

Return the pair potentials. If no pair potentials are available, it returns None.

Returns
The pair potentials.

Return type
PairPotential

usingOrthogonalBasis()

Routine for determining if the Hamiltonian parametrization is using an orthogonal basis set.

Returns
True if the basis is orthogonal, False otherwise.

Return type
bool

Usage Examples

The following example demonstrates some of the different Slater-Koster basis sets and directories available for a Slater Koster-type Hamiltonian parametrization.

```
# Create a Slater-Koster Hamiltonian parametrization with a
# Boykin basis set for Silicon.
hamiltonian_parametrization = SlaterKosterHamiltonianParametrization(
    basis_set=Boykin.Si_Basis)

# Create a S-K parametrization with the basis parameteres defined
# in a DFTBDirectory.
directory = DFTBDirectory('dftb/mio-1-1/')
hamiltonian_parametrization = SlaterKosterHamiltonianParametrization(
```

(continues on next page)

(continued from previous page)

```
    basis_set=directory)

# Set up a basis set and pair potentials using a Hotbit directory.
basis_set = HotbitDirectory('hotbit/standard/')
pair_potentials = HotbitDirectory('hotbit/standard/')

# Set up a Slater-Koster Hamiltonian parametrization.
hamiltonian_parametrization = SlaterKosterHamiltonianParametrization(
    basis_set=basis_set)

# Set up a semi-empirical calculator using the Hotbit-based
# parametrization and pair-potentials.
calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization,
    pair_potentials=pair_potentials)
```

Notes

The Slater–Koster parameters can be provided either through the *SlaterKosterTable* class or through various 3rd-party formats. The supported 3rd-party formats are the DFTB Slater–Koster files from the DFTB consortium, and the Hotbit Slater–Koster files from the Hotbit consortium.

To use parameters in the DFTB or Hotbit format, put the parameter files in a single directory, and set up the *basis_set* using the functions *DFTBDirectory* or *HotbitDirectory*. Note that if one wishes to use the corresponding pair-potentials to calculate the atomic forces, the same directory must be given as the `pair_potentials` argument for the *SemiEmpiricalCalculator* or *DeviceSemiEmpiricalCalculator*.

Tables showing which elements are covered by the Slater-Koster parameters are given here: *Slater–Koster basis sets*.

SlaterKosterOnsiteParameters

class SlaterKosterOnsiteParameters(*element*, *angular_momenta*, *occupations=None*, *filling_method=None*, *ionization_potential=None*, *onsite_hartree_shift=None*, *onsite_spin_split=None*, *onsite_spin_orbit_split=None*, *number_of_valence_electrons=None*, *vacuum_level=None*)

Class for representing the parameters that determine the onsite parameters of a Slater-Koster basis.

Parameters

- **element** (*PeriodicTableElement*) – The element associated with the basis set.
- **angular_momenta** (*list of int*) – The angular momentum of each subshell as a list of positive integers with one entry for each subshell.
- **occupations** (*list of float*) – The initial occupation of each subshell as a list of positive floats with one entry for each subshell. Either this, or `number_of_valence_electrons` must be given.
- **filling_method** (`SphericalSymmetric` | `Anisotropic`) – The method used for setting up the initial occupation. **Default:** `SphericalSymmetric`
- **ionization_potential** (*PhysicalQuantity* of type energy) – The ionization potential with one entry for each subshell. **Default:** Must be given.
- **onsite_hartree_shift** (*PhysicalQuantity* of type energy) – The on-site Hartree shift with one entry for each subshell. **Default:** Must be given.

- **onsite_spin_split** (*PhysicalQuantity* of type energy) – The on-site spin-split values of each orbital (subshell) pair. This should be a square matrix of size *n* where *n* is the number of subshells. **Default:** No splitting
- **onsite_spin_orbit_split** (*PhysicalQuantity* of type energy) – The on-site spin-orbit split with one entry for each subshell. **Default:** No splitting
- **number_of_valence_electrons** (*int*) – The number of valence electrons used to determine the neutral state. Must be non-negative. Either this, or `occupations` must be given.
- **vacuum_level** (*PhysicalQuantity* of type energy) – The energy shift of the vacuum level. **Default:** `0.0 * Hartree`

angularMomenta()

Returns
The angular momentum of each subshell.

Return type
list of int

element()

Returns
The element of the basis set.

Return type
PeriodicTableElement

fillingMethod()

Returns
The method used for setting up the initial occupation.

Return type
`SphericalSymmetric|Anisotropic`

ionizationPotential()

Returns
The ionization potential of each subshell in the basis set.

Return type
PhysicalQuantity of type energy

nlinfo()

Returns
The nlinfo.

Return type
dict

numberOfValenceElectrons()

Returns
The sum of the occupations.

Return type
float

occupations()

Returns
The occupations associated with the orbitals.

Return type
list of float

`onsiteHartreeShift()`

Returns
The on-site Hartree shift of each subshell in the basis set.

Return type
PhysicalQuantity of type energy

`onsiteSpinOrbitSplit()`

Returns
The on-site spin-orbit split of each subshell in the basis set.

Return type
PhysicalQuantity of type energy

`onsiteSpinSplit()`

Returns
The on-site spin-split values of each subshell in the basis set.

Return type
PhysicalQuantity of type energy

`uniqueString()`

Return a unique string representing the state of the object.

`vacuumLevel()`

Returns
The energy shift of the vacuum level.

Return type
PhysicalQuantity of type energy

Usage Examples

Define the onsite parameters for an sps* tight-binding model of silicon.

```
si_onsite = SlaterKosterOnsiteParameters(
    element = Silicon,
    angular_momenta=[0,1,0],
    occupations=[2,2,0],
    ionization_potential=[-4.545, 1.715, 6.6850]*eV,
    onsite_hartree_shift=0.0*eV)
```

Notes

The *SlaterKosterOnsiteParameters* is used to define the onsite parameters in a *SlaterKosterTable*.

SlaterKosterTable

class SlaterKosterTable(***kwargs*)

Class for representing the parameters that determine the parameters of a Slater-Koster basis.

The constructor takes parameters specified as keyword arguments. These arguments follow one of two conventions, depending on wether offsite or onsite parameters are being specified:

Onsite keyword arguments are in the form `element=instance`, where `element` is the full name of an element as a string, and `instance` is a *SlaterKosterOnsiteParameters* object

Offsite keyword arguments are in the form `element1_element2_XYZ=list`, where **element1:** the first element - e.g. `carbon`. **element2:** the second element - e.g. `hydrogen`. **X:** the orbital on the first element - `s`, `p`, `d`. **Y:** the orbital on the second element - `s`, `p`, `d`. **Z:** the type of interaction - `s` for sigma, `p` for pi, `d` for delta. **list:** A list of tuples in one of two formats: Either, `[(d0,h0,s0), (d1,h1,s1), ...]` where `(d,h,s)` indicate distance, hamiltonian and overlap elements. Or, `[(d0,h0), (d1,h1), ...]` in the case the overlap matrix is diagonal.

offsiteParameters()

Returns
The offsite parameters.

Return type
dict

onsiteParameters()

Returns
The onsite parameters.

Return type
dict

Usage Examples

Define a *SlaterKosterTable* with a Vogl silicon basis

```
# Generate a basis set for Silicon using
# J. Phys. Chem. solids Vol. 44, 365-378, 1983

# Parameters, distances in Angstrom silicon
a = 5.4306*Ang
d_n1 = math.sqrt(3.0)/4.0*a
d_n2 = math.sqrt(2.0)/2.0*a

# Matrix elements from paper, energies in eV
E_s =     -4.545*eV
E_p =      1.715*eV
E_sstar =  6.6850*eV

V_ss =    -8.3000*eV
```

(continues on next page)

(continued from previous page)

```
V_xx =     1.7150*eV
V_xy =     4.5750*eV
V_sp =     5.7292*eV
V_sstarp = 5.3749*eV

# ------------------------------------------------------------------------
# The paper is nearest neighbour Slater-Koster parameterization,
# hence it has to be tabulated using a power law.

# Specify that the cutoff for the matrix elements between 1. and 2. nn
rcut = 0.5*d_n1 + 0.5*d_n2

# Generate a series of strain -0.1 from +0.1
epsilon = numpy.linspace(-0.10, 0.10, 21)

# Specify the distances for which the matrix elements are specified
distances = [ d_n1*(1.0+x) for x in epsilon ]+ [ rcut ]

# Using a power-law to get the distance dependence of the matrix elements
sss  = [ 0.25*V_ss/(1.0+x)**2             for x in epsilon ] + [0.0*eV]
sps  = [ 0.25*math.sqrt(3)*V_sp/(1.0+x)**2     for x in epsilon ] + [0.0*eV]
pps  = [ 0.25*(V_xx+2.0*V_xy)/(1.0+x)**2  for x in epsilon ] + [0.0*eV]
ppp  = [ 0.25*(V_xx-V_xy)/(1.0+x)**2      for x in epsilon ] + [0.0*eV]
ps1s = [ 0.25*math.sqrt(3)*V_sstarp/(1.0+x)**2 for x in epsilon ] + [0.0*eV]

# Create the onsite.
si_onsite = SlaterKosterOnsiteParameters(element = Silicon, # Not needed.
                                         angular_momenta = [0,1,0],
                                         occupations =     [2,2,0],
                                         ionization_potential = [E_s, E_p, E_sstar],
                                         onsite_hartree_shift = ATK_U(Silicon, ['3s','3p
↪','3s']),
                                         onsite_spin_split = ATK_W(Silicon, ['3s','3p',
↪'3s']),
)

Silicon_Basis = SlaterKosterTable(silicon_silicon_sss =  zip(distances, sss),
                            silicon_silicon_sps =  zip(distances, sps),
                            silicon_silicon_pps =  zip(distances, pps),
                            silicon_silicon_ppp =  zip(distances, ppp),
                            silicon_silicon_ps1s = zip(distances, ps1s),
                            silicon = si_onsite,
                            )
```

Notes

•The onsite and offsite arguments can be specified at the same time, and in any order.

```
carbon_onsite_settings = SlaterKosterOnsiteParameters(...)
hydrogen_onsite_settings = SlaterKosterOnsiteParameters(...)
t = SlaterKosterTable(carbon=carbon_onsite_settings,
                      hydrogen=hydrogen_onsite_settings,
                      hydrogen_hydrogen_sss=[(1.1*Ang,0.2*eV,0.2), ... ],
                      hydrogen_carbon_spp=[(0.2*Ang,0.1*eV), ... ])
```

•If there are multiple orbitals in an angular shell the shells are named, s0, s1, s2, ..

```
carbon_onsite_settings = SlaterKosterOnsiteParameters(...)
hydrogen_onsite_settings = SlaterKosterOnsiteParameters(...)
t = SlaterKosterTable(carbon=carbon_onsite_settings,
                      hydrogen=hydrogen_onsite_settings,
                      hydrogen_hydrogen_s0s0s=[(1.1*Ang,0.2*eV,0.2), ... ],
                      hydrogen_hydrogen_s1s1s=[(1.1*Ang,0.22*eV), ... ],
                      hydrogen_carbon_s0pp=[(0.2*Ang,0.1*eV), ... ])
                      hydrogen_carbon_s1pp=[(0.2*Ang,0.15*eV), ... ])
```

Note that s is equivalent with s0, see the example Vogl.py above which defines orbitals s and s1.

•In the offsite keyword arguments the order of the element and shell pairs is irrelevant. E.g., it is equivalent to define

```
t = SlaterKosterTable(carbon=carbon_onsite_settings,
                      hydrogen=hydrogen_onsite_settings,
                      hydrogen_carbon_sps=[(0.2*Ang,0.1*eV), ... ]),
                      ...
```

or

```
t = SlaterKosterTable(carbon=carbon_onsite_settings,
                      hydrogen=hydrogen_onsite_settings,
                      carbon_hydrogen_pss=[(0.2*Ang,0.1*eV), ... ]),
                      ...
```

The proper signs for even and odd parity are enforced internally, therefore each interaction should be specified only once.

•Matrix elements that are not specified in the SlaterKosterTable are set to zero.

Specifying custom tags for a Slater-Koster table

It is possible to assign different parametrizations for different atoms of the same type by using tags. The tags must be strings without the underscore (_) character, and they must also be valid Python keyword arguments. For example, `foo1`, `Hf02` and `specialtag` are all valid tag names, but `my_tag` and `1foo` are not. In this case, the last one is not a valid Python keyword argument. The tags are also treated as case-insensitive, i.e. `Hf02` and `hfo2` correspond to the same tag.

The tags can be specified for a Slater-Koster table in a similar fashion as for any periodic table element. The syntax looks as follows:

```
t = SlaterKosterTable(element1=SlaterKosterOnSiteParameters(...),
                      element2=SlaterKosterOnSiteParameters(...),
                      tag1=SlaterKosterOnSiteParameters(...),
                      tag2=SlaterKosterOnSiteParameters(...),
                      element1_element2_sss=...,
                      element1_tag1_sss=...,
                      element1_tag2_sss=...)
```

Note that here we presume that the off-site terms are only required between `element1` and both tagged basis sets. In order for the special on-site parameters to be used in the calculation, the tagged atoms must also be specified on the configuration, i.e.

```
configuration.addTags('tag1', tag1_indices)
configuration.addTags('tag2', tag2_indices)
```

Note: All atoms in the configuration must have onsite parameters specified for the element type or through tags.

The following example script demonstrates how to use tags for a Slater-Koster table to substitute the first two Gallium atoms with a simplified VCA parameter for a InGaAs alloy with a 50 % molar fraction of Indium.

```
# Set up lattice
vector_a = [5.63685, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.63685, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.63685]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Gallium, Arsenic, Gallium, Arsenic, Gallium, Arsenic, Gallium,
            Arsenic]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25],
                          [ 0.5 ,  0.5 ,  0.  ],
                          [ 0.75,  0.75,  0.25],
                          [ 0.5 ,  0.  ,  0.5 ],
                          [ 0.75,  0.25,  0.75],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.25,  0.75,  0.75]]
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Fetch Bassani InAs and GaAs basis sets.
inas_basis = Bassani.InAs_Basis
gaas_basis = Bassani.GaAs_Basis

# Set up keyword arguments for the Slater-Koster table; add
# the GaAs onsite and off-site parameters there.
```

(continues on next page)

(continued from previous page)

```
args = {}
args.update(gaas_basis.onsiteParameters())
args.update(gaas_basis.offsiteParameters())

# Add the tag 'inga' for the first two Ga atoms.
bulk_configuration.addTags('inga', [0, 2])

# Create the InGa on-site term by mixing the ionization potentials
# of In and Ga.
indium_onsite_term = inas_basis.onsiteParameters()['in']
gallium_onsite_term = gaas_basis.onsiteParameters()['ga']

ga_onsite_potential = gallium_onsite_term.ionizationPotential()
in_onsite_potential = indium_onsite_term.ionizationPotential()

inga_onsite_potential = 0.5 * (ga_onsite_potential + in_onsite_potential)
inga_onsite_term = gallium_onsite_term(ionization_potential=inga_onsite_potential)

# Use the 'inga' tag to apply the special on-site term for
# the first two Ga atoms.
args.update({'inga': inga_onsite_term})
# One should also generate inga_as_sss, inga_as_sps, ... off-site terms according
# to some interpolation between In-As and Ga-As.
# For sake of simplicity we will assume that they are all identical to the ga-as␣
↪interactions:
args.update({
    'inga_as_sss'  : gaas_basis.offsiteParameters()['ga_as_sss'],
    'inga_as_sps'  : gaas_basis.offsiteParameters()['ga_as_sps'],
    'inga_as_sds'  : gaas_basis.offsiteParameters()['ga_as_sds'],
    'inga_as_s1s1s': gaas_basis.offsiteParameters()['ga_as_s1s1s'],
    'inga_as_s1ps' : gaas_basis.offsiteParameters()['ga_as_s1ps'],
    'inga_as_s1ds' : gaas_basis.offsiteParameters()['ga_as_s1ds'],
    'inga_as_pps'  : gaas_basis.offsiteParameters()['ga_as_pps'],
    'inga_as_ppp'  : gaas_basis.offsiteParameters()['ga_as_ppp'],
    'inga_as_pds'  : gaas_basis.offsiteParameters()['ga_as_pds'],
    'inga_as_pdp'  : gaas_basis.offsiteParameters()['ga_as_pdp'],
    'inga_as_dds'  : gaas_basis.offsiteParameters()['ga_as_dds'],
    'inga_as_ddp'  : gaas_basis.offsiteParameters()['ga_as_ddp'],
    'inga_as_ddd'  : gaas_basis.offsiteParameters()['ga_as_ddd'],
    'as_inga_ss1s' : gaas_basis.offsiteParameters()['as_ga_ss1s'],
    'as_inga_sps'  : gaas_basis.offsiteParameters()['as_ga_sps'],
    'as_inga_sds'  : gaas_basis.offsiteParameters()['as_ga_sds'],
    'as_inga_s1ss' : gaas_basis.offsiteParameters()['as_ga_s1ss'],
    'as_inga_s1ps' : gaas_basis.offsiteParameters()['as_ga_s1ps'],
    'as_inga_s1ds' : gaas_basis.offsiteParameters()['as_ga_s1ds'],
    'as_inga_pds'  : gaas_basis.offsiteParameters()['as_ga_pds'],
    'as_inga_pdp'  : gaas_basis.offsiteParameters()['as_ga_pdp'],
    })

# Set up the basis and calculate band structure.
basis = SlaterKosterTable(**args)
calculator = SemiEmpiricalCalculator(
```

(continues on next page)

(continued from previous page)

```
    hamiltonian_parametrization=SlaterKosterHamiltonianParametrization(basis_set=basis))
bulk_configuration.setCalculator(calculator)
bandstructure = Bandstructure(bulk_configuration)
```

For the details of the Slater-Koster model, see the chapter on *Semi Empirical*.

SlaterOrbital

class SlaterOrbital(*principal_quantum_number*, *angular_momentum*, *slater_coefficients*, *weights*)

Constructor for the Slater orbitals.

Parameters

- **principal_quantum_number** (*positive int*) – The Principal quantum number (*n*) of the orbital.
- **angular_momentum** (*non-negative int*) – The Azimuthal quantum number (*l*) of the orbital.
- **slater_coefficients** (*PhysicalQuantity* of type inverse length) – The Slater coefficients as inverse length. Maximum two coefficients can be given. Every entry must be positive.
- **weights** (numpy.array) – The weight for each of the Slater coefficients. Each entry should be positive.

angularMomentum()

Returns
The angular momentum for the orbital.

Return type
int

principalQuantumNumber()

Returns
The principal quantum number n for the orbital.

Return type
int

slaterCoefficients()

Returns
The Slater coefficients as inverse length.

Return type
list of *PhysicalQuantity* of type inverse length

Usage Examples

Define a 1s *SlaterOrbital* from a single exponential function

```
carbon_2s = SlaterOrbital(
    principal_quantum_number=2,
    angular_momentum=0,
    slater_coefficients=[2.0249*1/Bohr],
    weights=[0.76422]
    )
```

Define a 2p *SlaterOrbital* as superposition of two exponential functions

```
carbon_2p = SlaterOrbital(
    principal_quantum_number=2,
    angular_momentum=1,
    slater_coefficients=[1.62412*1/Bohr , 2.17687*1/Bohr],
    weights=[0.27152, 0.73886]
    )
```

Notes

Within the extended Hückel model[1], the electronic structure is expanded in a basis formed by a linear combination of atomic orbitals (LCAO)

$$\phi_{nlm}(\mathbf{r}) = R_{nl}(r)Y_{lm}(\hat{r}),$$

where Y_{lm} is a spherical harmonic and R_{nl} is a Slater orbital

$$R_{nl}(r) = \frac{r^{n-1}}{\sqrt{(2n)!}}\left[C_1(2\eta_1)^{n+\frac{1}{2}}e^{-\eta_1 r} + C_2(2\eta_2)^{n+\frac{1}{2}}e^{-\eta_2 r}\right].$$

The *SlaterOrbital* is described by the adjustable parameters η_1, η_2, C_1, and C_2. These parameters must be defined for each angular shell of valence orbitals for each element.

Table 4.23: Slater orbital parameters.

Symbol	*SlaterOrbital* parameters
n	`principal_quantum_number`
l	`angular_momentum`
η	`slater_coefficients`
C	`weights`

In the current version QuantumATK comes with built-in Hoffmann and Müller parameter sets which are appropriate for organic molecules, and Cerda parameters[2] which are appropriate for crystals. The parameter set are available with the keyword `HoffmannHuckelParameters.ElementName`, `MullerHuckelParameters.ElementName` and `CerdaHuckelParameters.ElementName`, where `ElementName` is the name of the element.

Reference

SolidSolubility

class SolidSolubility(*solutes*, *solvent_components*, *solvent_density=None*, *temperature=None*, *parameters=None*)

Calculate the solubility of molecular solids using the COSMO-RS model.

Parameters

- **solutes** (*CosmoRealSpecies* | Sequence of *CosmoRealSpecies*) – The solids for which the solubility is calculated.

[1] K.ăStokbro, D.ăE. Petersen, S.ăSmidstrup, A.ăBlom, M.ăIpsen, and K.ăKaasbjerg. Semiempirical model for nanoscale device simulations. *Phys. Rev. B*, 82:075420, Aug 2010. doi:10.1103/PhysRevB.82.075420.

[2] J.ăCerda and F.ăSoria. Accurate and transferable extended Hückel-type tight-binding parameters. *Phys. Rev. B*, 61:7965–7971, Mar 2000. doi:10.1103/PhysRevB.61.7965.

- **solvent_components** (*CosmoRealSolvent* | *MoleculeConfiguration* | *CosmoRSMixture*) – The solvent in which the solid is dissolved in.
- **solvent_density** (*PhysicalQuantity* of type mass per volume) – The density of the solvent. If not specified solvent density is estimated from the components.
- **temperature** (*PhysicalQuantity* of type temperature) – The solvent temperature. **Default:** `298.15 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

parameters()

Returns
The COSMO-RS parameters

Return type
CosmoRSParameters

soluteConcentration()

Returns
The concentration of each solute in the solvent.

Return type
Sequence of *PhysicalQuantity* of molar

soluteMoleFraction()

Returns
The mole fractions of each species dissolved in the solvent.

Return type
Sequence of float

soluteWeightPercent()

Returns
The mass-mass loading of the species in the solvent.

Return type
float

solutes()

Returns
The gases the solubility is calculated for.

Return type
Sequence of *CosmoRealSpecies*

solventComponents()

Returns
The components of the solvent.

Return type
CosmoRSMixture

solventDensity()

Returns
The solvent density, if specified. If not returns None, and the solvent density is taken from the solvent components.

Return type
PhysicalQuantity of type mass per volume

`temperature()`

Returns
The solvent temperature.

Return type
PhysicalQuantity of type temperature

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

Calculate the amount of aspirin dissolved in water. Note in this example the melting point and enthalpy of fusion are provided for aspirin but not the change in specific heat capapcity Data link. This data may be set using the method `setSpecificHeatCapacityChange`.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
aspirin = database.exportSpecies('acetylsalicylic acid')

# Set the melting temperature and enthalpy of fusion.
aspirin.setMeltingPoint(409*Kelvin)
aspirin.setEnthalpyOfFusion(29.17*kiloJoulePerMol)

# Determine the solid solubility.
solid_solubility = SolidSolubility(
    aspirin,
    water,
    temperature=298*Kelvin,
    parameters=CosmoRSParameters()
)

# Retrieve solute concentration in different forms.
molar_concentration = solid_solubility.soluteConcentration()[0]
nlprint(f'The molar concentration of aspirin in water is {molar_concentration}')
mole_fraction = solid_solubility.soluteMoleFraction()[0]
nlprint(f'The mole fraction of aspirin in water is {mole_fraction}')
weight_percent = solid_solubility.soluteWeightPercent()[0]
nlprint(f'The weight percent of aspirin in water is {weight_percent}')
```

aspirin_water_example.py

Notes

The *SolidSolubility* object allows to calculate COSMO-RS solid-liquid equilibrium (SLE) of a solid substance given a liquid phase. The amount of a solid substance being dissolved, x^i, is calculated according to

$$\mu_i^{liq} - \Delta G^{fus} = \mu_i^{solv} + RT \ln x_i$$

Here i represents the given substance, μ_i^{solv} chemical potential of the substance in the solvent phase and μ_i^{liq} is the chemical potential of the pure liquid compound. The Gibbs free energy of fusion, ΔG^{fus} is approximated with the *CosmoRealSolid* object. The most accurate approximation is generally:

$$\Delta G^{fus} = \Delta H^{fus}\left(1 - \frac{T}{T_m}\right) - \Delta C_p^{fus}(T_m - T) + \Delta C_p^{fus} T \ln \frac{T_m}{T}$$

Note several pure compound data are required. T_m is the melting pont, ΔH^{fus} the heat of fusion and ΔC_p^{fus} the change in molar heat capacity upon melting, which corresponds to the difference in molar heat capacity of a compound in its pure liquid and solid state[1]. If ΔC_p^{fus} is not available the two last terms may be neglected which gives

$$\Delta G^{fus} = \Delta H^{fus}\left(1 - \frac{T}{T_m}\right)$$

If none of these data is provided the Gibbs free energy of fusion is predicted from a simple linear model at 298K.

$$\Delta G^{fus}(298K) = 12.2 V_{COSMO} - 0.76 N_{ringatoms} + 0.54 \mu_i^{water}$$

The linear coefficients are purely empirical and determined by fitting to experimental data[2].

The *SolidSolubility* does not take into account the changing nature of the solvent as more solute is dissolved. It therefore only gives an estimate of the initial solubility at low solute concentration. In cases where the temperature is above the given melting point of a substance, ΔG^{fus} is assumed to be zero, and the substance is treated as though it is in the liquid state.

SolidThermodynamicIntegrationHook

class SolidThermodynamicIntegrationHook(*configuration*, *spring_constants*, *total_steps*, *temperature=None*, *equilibration_steps=None*, *lambda_min=None*, *measurement_call_interval=None*)

Post step hook for calculating the free energy of solid

Parameters

- **configuration** (*DistributedConfiguration*) – The current configuration.
- **spring_constants** (*PhysicalQuantity* of type force per distance.) – The atomic spring constants used in the reference system. Can be one spring constant for all atoms, or an array of spring constants, one for each atom.
- **total_steps** (*int*) – The total number of simulation steps.
- **temperature** (*float*) – The temperature of the system
- **equilibration_steps** (*int*) – The number of steps to equilibrate at the reference state.
- **lambda_min** (*float*) – Minimum value of the switching parameter used during the simulation.
- **measurement_call_interval** (*int*) – The interval between calculating measurement data

[1] Zhen Song, Jingwen Wang, and Kai Sundmacher. Evaluation of COSMO-RS for solid–liquid equilibria prediction of binary eutectic solvent systems. *Green Energy & Environment*, 6(3):371–379, 2020. doi:10.1016/j.gee.2020.11.020.

[2] Andreas Klamt, Frank Eckert, Martin Hornig, MichaelăE. Beck, and Thorsten Bürger. Prediction of aqueous solubility of drugs and pesticides with COSMO-RS. *Journal of Computational Chemistry*, 23(2):275–281, 2002. doi:10.1002/jcc.1168.

`callInterval()`

Returns
The call interval of this hook function.

Return type
int

`reference()`

Returns the reference system.

Returns
The reference system

Return type
`BaseReference`

`referenceFreeEnergy()`

Returns the free energy of the reference system.

Returns
The enegy of the reference system

Return type
PhysicalQuantity of type energy

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples

In the example below, the free energy of two Co crystal phases is calculated at different temperatures. The result can then be used to determine the relative stability of the two phases.

For each phase and temperature, a number of steps are performed:

- Equilibration in the iso-baric ensemble at the given temperature and pressure. During this simulation the volume is sampled, and finally averaged.
- From the iso-baric equilibration, the force constant is calculated and the starting configuration is generated using the sampled volume.
- A production MD simulation in the canonical ensemble is performed. During this simulation a number of properties are sampled, such as the potential energy of both the actual system and the reference system.

The result of each production simulation is the absolute free energy at the given temperature and pressure.

Note that each simulation takes about 15 minutes to complete on a standard workstation. This means that simulations of the two structures at four different temperatures would take about 2 hours to run.

`ti_example_workflow.hdf5`

General

Thermodynamic integration[1] is a method used to calculate free energy differences between systems. In molecular simulations, it is often used to calculate the free energy difference between two states of a system, such as two different crystal phases of a solid. In that way, one can determine which phase is the most stable under the given conditions.

Thermodynamic integration involves running several simulations where a parameter connecting the systems is varied. This can be computationally expensive and a more efficient way is to utilize non-equilibrium simulations. In non-equilibrium thermodynamic integration[2,3] the parameter connecting the system is changed in a non-equilibrium manner in a single simulation. From this simulation the irreversible work can be estimated, which in turn enables one to calculate the free energy difference.

By calculating the free energy difference of a system with that of a reference system, with known free energy, one can calculate the absolute free energy of the system. This method is implemented in the two hook function classes *SolidThermodynamicIntegrationHook* and *LiquidThermodynamicIntegrationHook*. *SolidThermodynamicIntegrationHook* is suitable for calculating the free energy of a solid and uses the Einstein crystal as a reference system[Page 3851, 3]. When using *LiquidThermodynamicIntegrationHook* the reference system is an Uhlenbeck-Ford liquid[4].

Notes

- The Uhlenbeck-Ford model depends on two parameters. One that determines the length scale of the system, and one that determines the energy scale. Both of these parameters can usually be taken as their Lennard-Jones counterparts.
- The Einstein crystal depends on a force constant. In the supplied template this force constant is automatically calculated.

SolvationEnergy

class SolvationEnergy(*configuration*)

Create the object, calculating the solvation and gas phase energies.

Parameters
configuration (MoleculConfiguration | *SurfaceConfiguration* | *BulkConfiguration*) – The configuration for which the energy is calculated. This configuration must have an attached calculator that contains a solvation model, such as an LCAOCalculator that has COSMO enabled.

evaluate()

Returns
The total solvation energy of the system.

Return type
PhysicalQuantity of type energy

gasEnergy()

Returns
The calculated gas phase energy.

[1] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2023.

[2] Masakatsu Watanabe and WilliamăP. Reinhardt. Direct dynamical calculation of entropy and free energy by adiabatic switching. *Phys. Rev. Lett.*, 65:3301–3304, Dec 1990. URL: https://link.aps.org/doi/10.1103/PhysRevLett.65.3301, doi:10.1103/PhysRevLett.65.3301.

[3] Sarath Menon, Yury Lysogorskiy, Jutta Rogal, and Ralf Drautz. Automated free-energy calculation from atomistic simulations. *Phys. Rev. Mater.*, 5:103801, Oct 2021. URL: https://link.aps.org/doi/10.1103/PhysRevMaterials.5.103801, doi:10.1103/PhysRevMaterials.5.103801.

[4] Rodolfo PaulaăLeite, Rodrigo Freitas, Rodolfo Azevedo, and Maurice deăKoning. The Uhlenbeck-Ford model: Exact virial coefficients and application as a reference system in fluid-phase free-energy calculations. *The Journal of Chemical Physics*, 145(19):194101, 11 2016. URL: https://doi.org/10.1063/1.4967775, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4967775/13587547/194101_1_online.pdf, doi:10.1063/1.4967775.

Return type
TotalEnergy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Information about the solvation energy calculation.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

solventEnergy()

Returns
The calculated solvation energy.

Return type
TotalEnergy

totalSolvationEnergy()

Returns
Return the total solvation energy.

Return type
PhysicalQuantity of type energy

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the solvation energy of ammonia in a toluene solvent. This calculation gives a solvation energy of approximately -2.38 kcal/mol, which is in agreement with the experimental value[1].

```
solvation_parameters = CosmoSolvationParameters(
    solvent_dielectric_constant=2.3741,
    solvent_surface_tension=27.9*dyne/cm
)

calculator = LCAOCalculator(
    basis_set=basis_set,
    solvation_parameters=solvation_parameters
)

# Set the calculator
configuration.setCalculator(calculator)
configuration.update()
nlsave('ammonia_toluene.hdf5', configuration)

# Optimize the geometry
optimized_configuration = OptimizeGeometry(
    configuration=configuration,
    trajectory_filename='ammonia_toluene.hdf5'
)
nlsave('ammonia_toluene.hdf5', optimized_configuration)

# Calculate the solvation energy
solvation_energy = SolvationEnergy(
    configuration=optimized_configuration
)
nlsave('ammonia_toluene.hdf5', solvation_energy)
```

ammonia_toluene.py

Notes

The *SolvationEnergy* analysis allows the calculation of the solvation energy of a molecule or surface, including the gas phase reference energy. In the case of a surface, the solvation energy is the solvation energy for the surface area of the unit cell. The solvent and gas phase energy calculations are performed with the same calculator settings, except for the inclusion of a continuum solvation model. This removes the need for changing calculators during the calculation to obtain both energies.

In setting up the calculation, the configuration passed to the *SolvationEnergy* class must have an :class~.*LCAOCalculator* or *DeviceLCAOCalculator* with an appropriate solvation model. The solvated molecular energy is calculated including this solvation model. Once the calculation is finished, the total solvation energy can be obtained using either the `evaluate` or the `totalSolvationEnergy` methods. The total energy of both solvent and gas phases can also be obtained using the `solventEnergy` and `gasEnergy` methods respectively. These both return a *TotalEnergy* analysis object. This allows the different energy components in each calculation to be inspected.

[1] AleksandrăV. Marenich, ChristopherăJ. Cramer, and DonaldăG. Truhlar. Universal solvation model based on solute electron density and on a continuum model of the solvent defined by the bulk dielectric constant and atomic surface tensionsl. *J. Phys. Chem. B.*, 113(18):6378, 2009. doi:10.1021/jp810292n.

SolventSurface

class SolventSurface(*configuration*)

Class for storing the configuration and solvent surface for later visualization.

Parameters
configuration (*MoleculeConfiguration*) – Configuration with a calculator that has a solvent surface.

configuration()

Returns
The configuration associated with the solvent surface.

Return type
MoleculeConfiguration | *BulkConfiguration* | *SurfaceConfiguration*

evaluate()

Returns
The solvent surface

Return type
CosmoSolventSurface

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Information about the calculated solvent surface.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

surface()

Returns
The solvent surface

Return type
CosmoSolventSurface

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Get a COSMO solvent surface for a methanol molecule.

```
# Set the LCAO Calculator
solvation_parameters = CosmoSolvationParameters()
calculator = LCAOCalculator(
    solvation_parameters=solvation_parameters
)
configuration.setCalculator(calculator)
configuration.update()
nlsave('methanol_cosmo_surface.hdf5', configuration)


# Get the COSMO solvent surface
solvent_surface = SolventSurface(
    configuration=configuration
)
nlsave('methanol_cosmo_surface.hdf5', solvent_surface)

cosmo_surface = solvent_surface.evaluate()
```

methanol_solvent_surface.py

Once the *SolventSurface* is calculated, it can be opened in the Viewer to display the solvent surface.

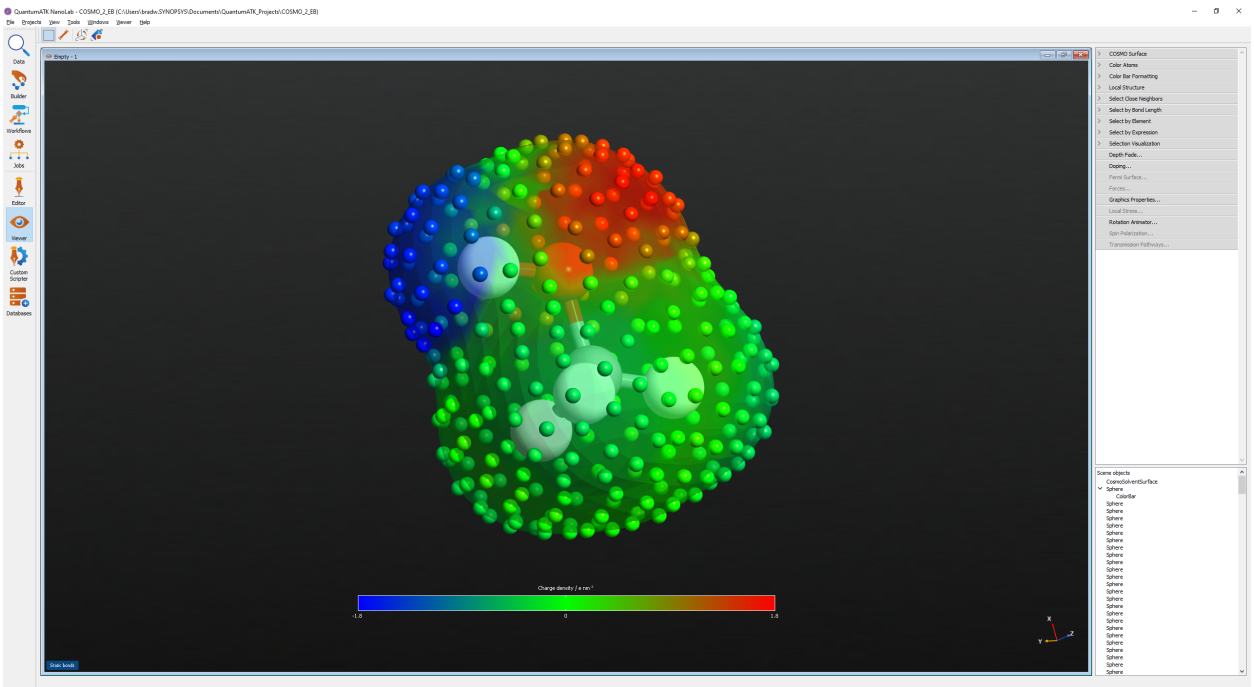


Fig. 4.37: Visualization of the methanol COSMO solvent surface.

Notes

The *SolventSurface* object saves and stores a solvent surface that is used in a continuum solvation DFT calculation. This object can be used to access the solvent surface created in a COSMO calculation. It enables manual inspection of the different properties of the solvent surface, including DFT calculated properties such as surface charge. Opening the *SolventSurface* in the **Viewer** creates a visualization of the surface which can be customized using the COSMO Surface Viewer plugin.

In order to obtain the solvent surface, an updated configuration with a suitable solvent model must be provided. This can either be a molecule or a surface configuration. In the case where a DFT-LCAO calculation with the COSMO model is performed, the *SolventSurface* returns a *CosmoSolventSurface*. The surface can be returned with either the `evaluate` or the `surface` methods.

SpaceGroupConstraint

class SpaceGroupConstraint(*tolerance=PhysicalQuantity(1e-05, Ang)*)

This constraint will symmeterize the atomic forces and cell stress according to the space group of the lattice. This will ensure that the space group does not change during geometry optimization.

Parameters
tolerance (*PhysicalQuantity of type length*) – The distance tolerance used when determining the space group. **Default:** 1e-5*Angstrom

frozenDegreesOfFreedom(*local_atoms=None*)

Parameters
local_atoms (*list of int | None*) – The group of atoms from which the frozen degrees of freedom should be calculated, e.g. a thermalized group of atoms. **Default:** All atoms.

Returns
The number of degrees of freedom that are frozen by this constraint object.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run a geometry optimization of a rutile crystal with the space group fixed.

```
constraints = [SpaceGroupConstraint()]

bulk_configuration = OptimizeGeometry(
    bulk_configuration,
    max_forces=0.05*eV/Ang,
    max_stress=0.1*GPa,
    max_steps=200,
    max_step_length=0.2*Ang,
    constraints=constraints,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
)
```

space_group_constraint.py

Notes

- *SpaceGroupConstraint* can be used in *OptimizeGeometry* to constrain the space group during optimization. This means that the forces and stress are symmetrized according to the space group that is detected.
- In addition to the atomic number, *SpaceGroupConstraint* also considers each atoms initial spin when determining the symmetry operations, so that atoms with different initial spin are not considered symmetrically equivalent.
- *SpaceGroupConstraint* constraints cannot be used together with *FixAtomConstraints* or *RigidBody*.

Sparse

class Sparse

Cache self energies in sparse matrix storage.

SparseGreensFunction

class SparseGreensFunction(*processes_per_contour_point=None*, *workspace_path=None*)

The direct sparse inversion method used to calculate the Greens function and lesser Greens function.

The method distributes the evaluation of the (lesser) Greens function for multiple contour points, i.e., energy- and k-points, over all available processes.

The evaluation of the (lesser) Greens function in each contour point, can be performed by multiple processes in parallel.

The methods required virtual memory can be reduced by allocating workspace on disk. Usage of disk workspace has a negative impact on the methods performance.

Parameters

- **processes_per_contour_point** (*int*) – The number of processes used to calculate each contour point. Must be positive. The best performance is obtained when set to 1. Increasing the number of processes per contour point reduces the required memory. **Default:** 1
- **workspace_path** (*str*) – Path to an existing directory usable as workspace. An empty string disables the disk workspace. The given string must not be longer than 255 characters. **Default:** Disk workspace disabled.

processesPerContourPoint()

Returns
The processes per contour point

Return type
int

uniqueString()

Return a unique string representing the state of the object.

workspacePath()

Returns
The workspace path

Return type
str | None

Usage Examples

Setup a device calculation that computes the Greens function and lesser Greens function via direct sparse inversion using 2 MPI processes for each contour point.

```
greens_function_method = SparseGreensFunction(
    processes_per_contour_point=2)

device_algorithm_parameters = DeviceAlgorithmParameters(
    equilibrium_method=greens_function_method,
    non_equilibrium_method=greens_function_method,
 )

calculator = DeviceHuckelCalculator(
    device_algorithm_parameters=device_algorithm_parameters,
    )
```

SparseRecursionSelfEnergy

class SparseRecursionSelfEnergy(*save_self_energies=None*, *tolerance=None*, *maximum_iteration=None*, *sparse_threshold=None*, *storage_strategy=None*)

Self-energy calculator based on the Sparse Recursion method. This method is an iterative scheme for getting self energies, with convergence $O(2^N)$

Parameters

- **storage_strategy** (*SaveInMemory* | *StoreOnDisk* | *NoStorage*) – The way self energies are stored between iterations. **Default:** *SaveInMemory()*
- **tolerance** (*float > 0*) – The tolerance of convergence of the self energies before termination of the recursion. **Default:** `1e-13`
- **maximum_iteration** (*int > 0*) – The maximum number of recursion steps allowed before termination. **Default:** `400`
- **sparse_threshold** (*float > 0*) – Self energies (in Hartree) smaller than this value treated as 0 when storing as sparse matrix. **Default:** `1e-12`
- **save_self_energies** – **Deprecated:** from v2017.0, use `storage_strategy` instead.

maximumIteration()

Returns
The maximum number of recursion steps.

Return type
int

nlinfo()

Returns
The nlinfo.

Return type
dict

saveSelfEnergies()

Deprecated: from v2017.0

sparseThreshold()

Returns
Values treated as 0.

Return type
float

storageStrategy()

Returns
The way self energies are stored between iterations.

Return type
SaveInMemory | *StoreOnDisk* | *NoStorage*

tolerance()

Returns
The tolerance of convergence.

Return type
float

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define that the self energies on the complex contour are calculated with the recursion method, and the self energies are not saved.

```
device_algorithm_parameters = DeviceAlgorithmParameters(
    self_energy_calculator_complex=SparseRecursionSelfEnergy(storage_
↪strategy=NoStorage()),
)
```

More examples on how to use the `storage_strategy` parameter can be found in the *Usage Examples* of *RecursionSelfEnergy*.

Notes

The *SparseRecursionSelfEnergy* uses the iterative scheme by Lopez, Lopez and Rubio[1] for calculating the self energy matrix. The sparse recursion method is identical to the recursion method provided by *RecursionSelfEnergy*, but exploits inherent sparsity. This provides increased performance for all but the smallest systems.

[1] M.ăP. LopezăSancho, J.ăM. LopezăSancho, and J.ăRubio. Highly convergent schemes for the calculation of bulk and surface Green functions. *J. Phys. F: Metal Physics*, 15(4):851, 1985. URL: http://stacks.iop.org/0305-4608/15/i=4/a=009.

SpecificHeatCapacity

class SpecificHeatCapacity(*md*, *start_time=None*, *end_time=None*, *atom_selection=None*, *time_resolution=None*, *min_temperature=PhysicalQuantity(0.0, K)*, *max_temperature=PhysicalQuantity(1000.0, K)*, *temperature_resolution=None*, *info_panel=None*)

Class for calculating the specific heat capacity of a system from a MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory*) – The MDtrajectory to calculate the specific heat capacity for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **min_temperature** (*PhysicalQuantity* of type temperature) – The minimum temperature. **Default:** 0*Kelvin
- **max_temperature** (*PhysicalQuantity* of type temperature) – The maximum temperature. **Default:** 1000*Kelvin
- **temperature_resolution** (*PhysicalQuantity* of type temperature) – The difference between individual temperature points. **Default:** 100 points
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

frequencies()

Returns
The frequencies associated with the vibrational density of states.

Return type
PhysicalQuantity of type 1/time

specificHeatCapacity()

Returns
The specific heat capacity calculated over the requested temperature range.

Return type
PhysicalQuantity of type energy per mass per temperature

temperatures()

Returns
The array of temperature values that the specific heat capacity has been evaluated at.

Return type
PhysicalQuantity of type temperature

vibrationalDensityOfStates()

Returns
The vibrational density-of-states, calculated from the Fourier transformation of the velocities.

Return type
PhysicalQuantity of type time

Usage Examples

This example calculates the specific heat capacity of bulk silicon using classical molecular dynamics:

```
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='si_bulk_specific_heat.hdf5',
    steps=5000,
    log_interval=1,
    method=method
)

bulk_configuration = md_trajectory.lastImage()

# Calculate specific heat capacity.
specific_heat_capacity = SpecificHeatCapacity(
    md_trajectory,
    min_temperature=0.0*Kelvin,
    max_temperature=1000.0*Kelvin,
)

# Plot specific heat capacity.
import pylab
temperatures = specific_heat_capacity.temperatures().inUnitsOf(Kelvin)
specific_heats = specific_heat_capacity.specificHeatCapacity().inUnitsOf(Joule/gram/
↪Kelvin)
pylab.plot(temperatures, specific_heats)
pylab.xlabel('Temperature (K)')
pylab.ylabel('Specific Heat Capacity (J/g/K)')
pylab.savefig('si_bulk_specific_heat.png')
```

si_bulk_specific_heat.py

The resulting plot of heat capacity versus temperature shows the correct limiting behavior and low and high temperature. This is achieved, when calculating the heat capacity, by modeling the system as a collection of quantum harmonic oscillators, whose frequencies are determined from the vibrational density of states.

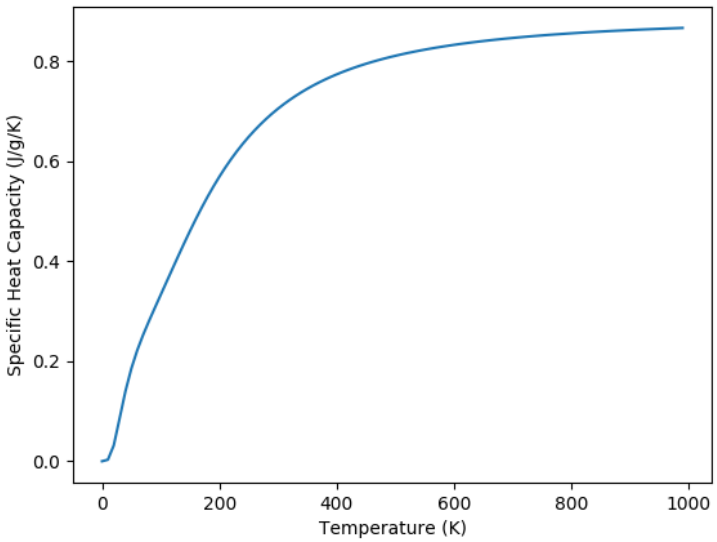
Notes

The heat capacity, is calculated by modeling the system as a collection of quantum harmonic oscillators, whose frequencies are determined using the vibrational density of states.[1] The specific heat, as a function of temperature, is

$$C_{\mathrm{v}}(T) = \frac{h^2}{mk_{\mathrm{B}}T^2}\int_0^\infty \frac{\nu^2 \exp(h\nu/k_{\mathrm{B}}T)}{(\exp(h\nu/k_{\mathrm{B}}T)-1)^2} S(\nu)\mathrm{d}\nu$$

where, T is the temperature, ν is the vibrational frequency, h is Plancks constant, m is the total mass of the atoms in the unit cell, k_{B} is Boltzmanns constant, and $S(\nu)$ is the vibrational density of states (VDOS). See the *VibrationalDensityOfStates notes* for details on how the VDOS is calculated.

[1] Jürgen Horbach, Walter Kob, and Kurt Binder. Specific Heat of Amorphous Silica within the Harmonic Approximation. *The Journal of Physical Chemistry B*, 103(20):4104–4108, 1999. doi:10.1021/jp983898b.



SphereRegion

class SphereRegion(*value*, *center*, *radius*)

Class for representing a spherical region of metallic or dielectric material.

Parameters

- **value** (float | *PhysicalQuantity* compatible with Volt) – The value that should be assigned to the sphere. Either the voltage or the dielectric constant of the region.
- **center** (*PhysicalQuantity* of type length) – Vector to the center of the sphere.
- **radius** (*PhysicalQuantity* of type length) – Radius of the sphere.

center()

Returns
The vector to the center of the sphere.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *header=True*)

Print a string containing an ASCII description of the SpatialRegion.

Parameters

- **stream** (*Stream based object*) – The io to write to.
- **header** (*bool*) – Option to emphasize the description as a header or not.

radius()

Returns
The radius of the sphere.

Return type
PhysicalQuantity of type length

toGLObject(*color*)

Create the necessary object(s) to pass back into the GLConfiguration.

Parameters
color (*GLColor*) – The region type-specific color.

Returns
The sphere.

Return type
GLSphere

uniqueString()

Return a unique string representing the state of the object.

value()

Function for asking for the value of the box region.

Returns
The value of the box region.

Return type
float | *PhysicalQuantity* compatible with Volt

Usage Examples

Define a dielectric region with dielectric constant $4\varepsilon_0$:

```
dielectric_region = SphereRegion(
    value=4.0,
    radius=2.0*Angstrom,
    center=[1.0, 2.0 ,3.0]*Angstrom
)
```

dielectric_sphereregion.py

Define two metallic regions where the second metallic region is constructed by cloning the first:

```
metallic_region1 = SphereRegion(
    value=1.0*Volt,
    radius=2.0*Angstrom,
    center=[1.0, 2.0 ,3.0]*Angstrom
)

metallic_region2 = metallic_region1(
    value=-1*Volt,
    radius=3.3*Angstrom,
)
```

metal_sphereregion.py

Notes

The following spatial regions are available:

- *BoxRegion*
- *SphereRegion*
- *TubeRegion*

SpinDynamicsCurieTemperature

class SpinDynamicsCurieTemperature(*temperatures*, *total_magnetic_moments*, *max_total_magnetic_moment*)

An object containing temperatures and related magnetizations with the purpose of estimating the Curie temperature.

Parameters

- **temperatures** (*PhysicalQuantity* of type temperature) – The list of temperatures.
- **total_magnetic_moments** (*PhysicalQuantity* of type energy/magnetic field) – The list of total magnetic moments of the entire system.
- **max_total_magnetic_moments** (*PhysicalQuantity* of type energy/magnetic field) – The maximum total magnetic moment of the entire system.

maxTotalMagneticMoment()

Get the maximum total magnetic moment. :return: The maximum total magnetic moment. :rtype: *PhysicalQuantity* of type energy/magnetic field

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
Structured information about the data.

Return type
dict

nlprint(*stream*)

Print report with the data and results to a given stream.

Parameters
stream (*stream*) – The stream that the report should be written to.

relativeMagnetizations()

Get the relative magnetizations. :return: The relative magnetizations. :rtype: ndarray

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

temperatures()

Get the temperatures. :return: The list of temperatures. :rtype: *PhysicalQuantity* of type temperature

totalMagneticMoments()

Get the total magnetic moments. :return: The list of magnetic moments. :rtype: *PhysicalQuantity* of type energy/magnetic field

uniqueString()

Return a unique string representing the state of the object.

SpinTransferTorque

class SpinTransferTorque(*configuration*, *energy=None*, *kpoints=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *z_integration_range=None*, *adaptive_directions=None*, *adaptive_method=None*)

Class for representing the spin transfer torque for a given device configuration and calculator.

Parameters

- **configuration** (*DeviceConfiguration*.) – The configuration to calculate the spin torque transfer of.
- **energy** (*PhysicalQuantity* of type energy) – The energy for which the spin transfer torque should be calculated. **Default:** `0.0 * eV`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *AdaptiveGrid*) – The k-points for which the spin transfer torque should be calculated. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **contributions** (`Left | Right`) – The density contributions to include in the spin transfer torque. **Default:** `Left`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to use. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel | AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy) – Small energy, used to move the spin transfer torque calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6 * eV`
- **z_integration_range** (*list of float*) – The range of fractional z-coordinates used for integrating the spin transfer torque. This should be given as a list of two floats. This is only used when the parameter kpoints is an *AdaptiveGrid* object. **Default:** `[0.0, 0.5]`
- **adaptive_directions** (*list of CartesianDirection.{X,Y,Z}*) – The directions used for adaptive integration. This should be given as 1, 2, or 3 `CartesianDirection` flags. **Default:** `[CartesianDirection.X, CartesianDirection.Y, CartesianDirection.Z]`
- **adaptive_method** (`RealSpace | OrbitalSpace`) – The method used for adaptive integration. **Default:** `OrbitalSpace`

adaptive()

Returns
`True` if adaptive grids are used.

Return type
bool

adaptiveIntegralValues()

Returns
The integrated spin-transfer torque in x, y, z direction obtained with adaptive integration. If adaptive grids are not used, this function returns None.

Return type
numpy.array | None

adaptiveSTTValues()

Returns
The integrated spin-transfer torque values at each k-point obtained only with adaptive integration. If adaptive grids are not used, this function returns None.

Return type
numpy.array | None

contributions()

Returns
The contributions parameter.

Return type
Left | Right

energy()

Returns
The energy parameter.

Return type
PhysicalQuantity of type energy

energyZeroParameter()

Returns
The energy zero parameter.

Return type
AverageFermiLevel | AbsoluteEnergy

evaluate(*x*, *y*, *z*, *spin=None*)

Evaluate in the point x,y,z.

Parameters

- **x** (*PhysicalQuantity* with type length) – The cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length) – The cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length) – The cartesian z coordinate.
- **spin** (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z) – The spin component to project on. **Default:** The spin that the object was constructed with.

Returns
The vector grid value at the specified point for the given spin. For Spin.All, a tuple with (Spin.Sum, Spin.X, Spin.Y, Spin.Z) components is returned.

Return type
PhysicalQuantity

getComponent(*component=None*)

Get the grid component.

Parameters
component (*int*) – The index of the component to extract; `0` for x, `1` for y, `2` for z. **Default:** Return all

Returns
The grid component.

Return type
`RealGrid3D` | list of `RealGrid3D`

gridCoordinate(*i*, *j*, *k*)

Return the grid coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coornate of the given grid index.

Return type
PhysicalQuantity of type length.

kpoints()

Returns
A list of k-points obtained with adaptive integration, each as a list of two fractional cartesian coordinates. If adaptive grids are not used, this function returns a *MonkhorstPackGrid* object.

Return type
list of list | *MonkhorstPackGrid*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

scale(*scale*)

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spinProjection(*spin=None*)

Query method to get a spin component of the VectorGridValue object.

Parameters
spin (Spin.Sum | Spin.Up | Spin.Down | Spin.X | Spin.Y | Spin.Z | Spin.RealUpDown | Spin.ImagUpDown) – The spin component for which to return the GridValues object. **Default:** Spin.Sum

Returns
A new VectorGridValues object for the specified spin.

Return type
VectorGridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array.

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

volumeElement()

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the *SpinTransferTorque* in the real-space representation of a one-dimensional carbon chain with a small gap in the middle where the electrons have to tunnel through. The spins to the left of the tunnelling gap points in a different direction than the spins in the right part.

```
# -----------------------------------------------------------
# Left electrode
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.8]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
left_electrode_elements = [Carbon, Carbon]
# Define coordinates
left_electrode_coordinates = [[ 3.  ,  3.  ,  1.45],
                              [ 3.  ,  3.  ,  4.35]]*Angstrom
# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )
# -----------------------------------------------------------
# Right electrode
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.8]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
right_electrode_elements = [Carbon, Carbon]
# Define coordinates
right_electrode_coordinates = [[ 3.  ,  3.  ,  1.45],
                               [ 3.  ,  3.  ,  4.35]]*Angstrom
# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )
# -----------------------------------------------------------
# Central region
# -----------------------------------------------------------
# Set up lattice
vector_a = [6.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 6.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 71.5]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)
# Define elements
```

(continues on next page)

(continued from previous page)

```
central_region_elements = [Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon,
                           Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon,
                           Carbon, Carbon, Carbon, Carbon, Carbon, Carbon, Carbon,␣
↪Carbon]
# Define coordinates
central_region_coordinates = [[  3.  ,   3.  ,   1.45],
                              [  3.  ,   3.  ,   4.35],
                              [  3.  ,   3.  ,   7.25],
                              [  3.  ,   3.  ,  10.15],
                              [  3.  ,   3.  ,  13.05],
                              [  3.  ,   3.  ,  15.95],
                              [  3.  ,   3.  ,  18.85],
                              [  3.  ,   3.  ,  21.75],
                              [  3.  ,   3.  ,  24.65],
                              [  3.  ,   3.  ,  27.55],
                              [  3.  ,   3.  ,  30.45],
                              [  3.  ,   3.  ,  33.35],
                              [  3.  ,   3.  ,  38.15],
                              [  3.  ,   3.  ,  41.05],
                              [  3.  ,   3.  ,  43.95],
                              [  3.  ,   3.  ,  46.85],
                              [  3.  ,   3.  ,  49.75],
                              [  3.  ,   3.  ,  52.65],
                              [  3.  ,   3.  ,  55.55],
                              [  3.  ,   3.  ,  58.45],
                              [  3.  ,   3.  ,  61.35],
                              [  3.  ,   3.  ,  64.25],
                              [  3.  ,   3.  ,  67.15],
                              [  3.  ,   3.  ,  70.05]]*Angstrom
# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )
device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode]
    )
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#----------------------------------------
# Basis Set
#----------------------------------------
basis_set = [
    LDABasis.Carbon_SingleZeta,
    ]

#----------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Exchange-Correlation
#-----------------------------------------
exchange_correlation = NCLDA.PZ

#-----------------------------------------
# Poisson Solver Settings
#-----------------------------------------
left_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                         [PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                         [PeriodicBoundaryCondition,PeriodicBoundaryCondition]]
    )
right_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                         [PeriodicBoundaryCondition,PeriodicBoundaryCondition],
                         [PeriodicBoundaryCondition,PeriodicBoundaryCondition]]
    )

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True)
    )

#-----------------------------------------
# Electrode Calculators
#-----------------------------------------
left_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    poisson_solver=left_electrode_poisson_solver,
    )
right_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    poisson_solver=right_electrode_poisson_solver,
    )

#-----------------------------------------
# Device Calculator
#-----------------------------------------
calculator = DeviceLCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    iteration_control_parameters = iteration_control_parameters,
    electrode_calculators=
        [left_electrode_calculator, right_electrode_calculator],
    )

# Define the spin rotation
theta = 90*Degrees
left_spins = [(i, 1, 0*Degrees, 0*Degrees) for i in range(12)]
right_spins = [(i, 1, theta, 0*Degrees ) for i in range(12,24)]
```

(continues on next page)

(continued from previous page)

```
spin_list = left_spins+right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

# Setup the initial state
device_configuration.setCalculator(
calculator,
    initial_spin=initial_spin,
)

# Calculate and save
device_configuration.update()
nlsave("STT.nc", device_configuration)

# ------------------------------------------------------------
# Mulliken population
# ------------------------------------------------------------
mulliken_population = MullikenPopulation(device_configuration)
nlsave("STT.nc", mulliken_population)
nlprint(mulliken_population)

#----------------------------------------
# Spin Transfer Torque
#----------------------------------------
spin_transfer_torque = SpinTransferTorque(
    configuration=device_configuration,
    energy=0.0*eV,
    kpoints=MonkhorstPackGrid(1,1),
    contributions=Left,
    self_energy_calculator=RecursionSelfEnergy(),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1.0e-6*eV
)
nlsave("STT.nc", spin_transfer_torque)
```

spin_transfer_torque.py

Spin transfer torque calculations can require a very fine k-point sampling in order to get converged results which can be very computationally demanding. An alternative to the MonkhorstPack grid, which is the default, is to use an *AdaptiveGrid* which is automatically refined in the the regions in k-space where a fine resolution is required. A calculation, with adaptive k-point grid, can be set up as follows:

```
#----------------------------------------
# Adaptive Grid.
#----------------------------------------

adaptive_grid = AdaptiveGrid(
        kA_range=[-0.5, 0.5],
        kB_range=[-0.5, 0.5],
        tolerance=1e-2,
        error_measure=Relative,
        number_of_initial_levels=2,
        maximum_number_of_levels=7)
```

(continues on next page)

(continued from previous page)

```
#----------------------------------------
# Spin Transfer Torque
#----------------------------------------
spin_transfer_torque = SpinTransferTorque(
    configuration=device_configuration,
    energy=0.0*eV,
    kpoints=adaptive_grid,
    contributions=Left,
    self_energy_calculator=RecursionSelfEnergy(),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1.0e-6*eV,
    z_integration_range=[0.0, 0.5],
    adaptive_directions=[0,1,2],
    adaptive_method=RealSpace,
)
nlsave("STT.nc", spin_transfer_torque)
```

spin_transfer_torque_adaptive.py

The last three input arguments to the *SpinTransferTorque* object are special for the use of adaptive grids. They are used to define a single function value from the three-dimensional vector grid, which represents the spin transfer torque (STT). The STT is integrated over the transverse directions, x and y, and along the z-axis in an interval defined by the `z_integration_range` such that z_{min} = `z_integration_range[0]` and z_{max} = `z_integration_range[1]`. In fractional coordinates, this results in:

$$S_i = \int_{-0.5}^{0.5} dx \int_{-0.5}^{0.5} dy \int_{z_{min}}^{z_{max}} dz \; STT_i(x, y, z),$$

where STT_i is the ith cartesian component of the STT. (0,1,2) correspond to x-, y-, and z-direcrions respectively. Which of the cartesian directions is included, is determined by the input argument `adaptive_directions`.

The last input argument, adaptive_method, can either be RealSpace or OrbitalSpace and refers to either a real-space or orbital-space evaluation of the integrated STT S_i . More information about the two approaches can be found in the tutorial *Spin Transfer Torque*.

Notes

More information is available in a technical note: *Noncollinear spins and spin transfer torque in devices*.

SplineInterpolation1D

class SplineInterpolation1D(*x_values*, *y_values*, *left_boundary_type=0*, *left_boundary_value=0.0*, *right_boundary_type=0*, *right_boundary_value=0.0*)

Constructor for the SplineInterpolation1D object.

Parameters

- **x_values** (*list(float,...)*) – A list of x values of data points.
- **y_values** (*list(float,...)*) – A list of y values of data points.
- **left_boundary_type** (*int (0 | 1 | 2)*) – Boundary condition type for the left boundary.

–0 - Parabolically terminated spline (left_boundary_value is ignored).

–1 - First derivative boundary condition.

–2 - Second derivative boundary condition.

•**left_boundary_value** (*float*) – Left boundary condition value.

•**right_boundary_type** (*int (0 | 1 | 2)*) – Boundary condition type for the right boundary.

–0 - Parabolically terminated spline (right_boundary_value is ignored).

–1 - First derivative boundary condition.

–2 - Second derivative boundary condition.

•**right_boundary_value** (*float*) – Right boundary condition value.

derivatives(*x*)

Calculate and return a tuple containing the derivatives (f, df/dx, d2f/dx2) of the spline interpolation evaluated at x.

@param x : The x value. @type : float @return The tuple (f, df/dx, d2f/dx2)

Usage Examples

Define a spline interpolation of the `sin` function:

```
x = numpy.linspace(-4,9,20)
y = numpy.sin(x)
f = SplineInterpolation1D(x,y)
```

Using the function, we obtain:

```
>>> print f(3)
0.141203196688
>>> print f.derivatives(3)
(0.14120319668785397, -0.98998913849695858, -0.14589372176098986)
>>> print numpy.array(f.derivatives(3)) - \\
numpy.array((numpy.sin(3), numpy.cos(3), -numpy.sin(3)))
[ 8.31886280e-05   3.35810349e-06  -4.77371370e-03]
```

Notes

•The data points does not need to have an increasing value of x.

•The data points does not need to have an equidistant separation.

•Function values outside the interpolation range will be extrapolated using the outermost spline function.

SplitInterstitial

class SplitInterstitial(*interstitial_element*, *site_element*, *site_index*, *unit_cell_index=None*, *cartesian_coordinates=None*, *fractional_coordinates=None*)

Data structure for a split interstitial (dumbbell) defect. This class only holds information and does not check for physical validity, which is imposed by `SplitInterstitialList` during dumbbell generation. The two sets of `cartesian_coordinates` (or `fractional_coordinates`) are symmetric with respect to `site_index`.

Parameters

- **interstitial_element** (*PeriodicTableElement*) – The element to add at the given positions.
- **site_element** (*PeriodicTableElement*) – The lattice site displaced by the an interstitial element.
- **site_index** (*int*) – The index of the atom in the bulk unit cell configuration to be substituted with a split interstitial defect. **Default:** `0`
- **unit_cell_index** (*sequence on int (size 3)*) – The index of the unit cell in the infinite crystal which the reference site belongs to. substituted with a split interstitial defect. **Default:** `(0, 0, 0)`
- **cartesian_coordinates** (*PhysicalQuantity* of type length (size 2x3) | None) – Positions at which to insert two atoms of the split interstitial defect, given in absolute coordinates. This option is mutually exclusive to `fractional_coordinates`. **Default:** None
- **fractional_coordinates** (*numpy.array (size 2x3) | None*) – Positions at which to insert two atoms of the split interstitial defect, given in fractional coordinates of the bulk unit cell. This option is mutually exclusive to `cartesian_coordinates`. **Default:** None

cartesianCoordinates()

Returns
The coordinates of the split interstitial defect as Cartesian coordinates if they have been specified, otherwise returns None.

Return type
PhysicalQuantity of type length (size 2x3) | None

defectList()

Returns
The list of constituent defects.

Return type
list of [*Vacancy* | *Substitutional* | *Interstitial*]

element()

Returns
The interstitial element used in the split-interstitial defect.

Return type
int

fractionalCoordinates()

Returns
The coordinates of the split interstitial defect as fractional coordinates if they have been specified, otherwise returns None.

Return type
numpy.array (size 2x3) | None

generateDefectConfiguration(*reference_bulk_configuration*, *supercell_repetitions*, *use_ghost=None*)

Generate the supercell with the embedded defect.

Parameters

- **reference_bulk_configuration** (*BulkConfiguration*) – The bulk unit cell configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
The supercell configuration with the embedded defect.

Return type
BulkConfiguration

siteIndex()

Returns
Index of the site being replaced by a split interstitial defect

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Notes

A split interstitial is a type of defect occurring in solids. It is created by introducing an impurity `interstitial_element` that displaces an atom of the lattice. The pair of impurity and displaced atom is commonly referred to as a *dumbbell* for its resemblance to the weight-lifting device, as seen in Fig. 4.38.

A *SplitInterstitial* can be added to a *NamedPointDefect* to define a defect for use in a *ChargedPointDefectConfiguration* calculation.

SplitInterstitialGenerator

class SplitInterstitialGenerator(*host_configuration*, *interstitial_element*, *split_direction*, *split_length=None*, *symmetry_tolerance=None*)

A class which describes the possible interstitial defects for a given host material.

Parameters

- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **interstitial_element** (*PeriodicTableElement*) – The element to add as split-interstitial defect.
- **split_direction** (*MillerIndices*) – MillerIndices object containing 3 int [hkl].
- **split_length** (*PhysicalQuantity* of type length) – Length of the split. **Default:** None.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.

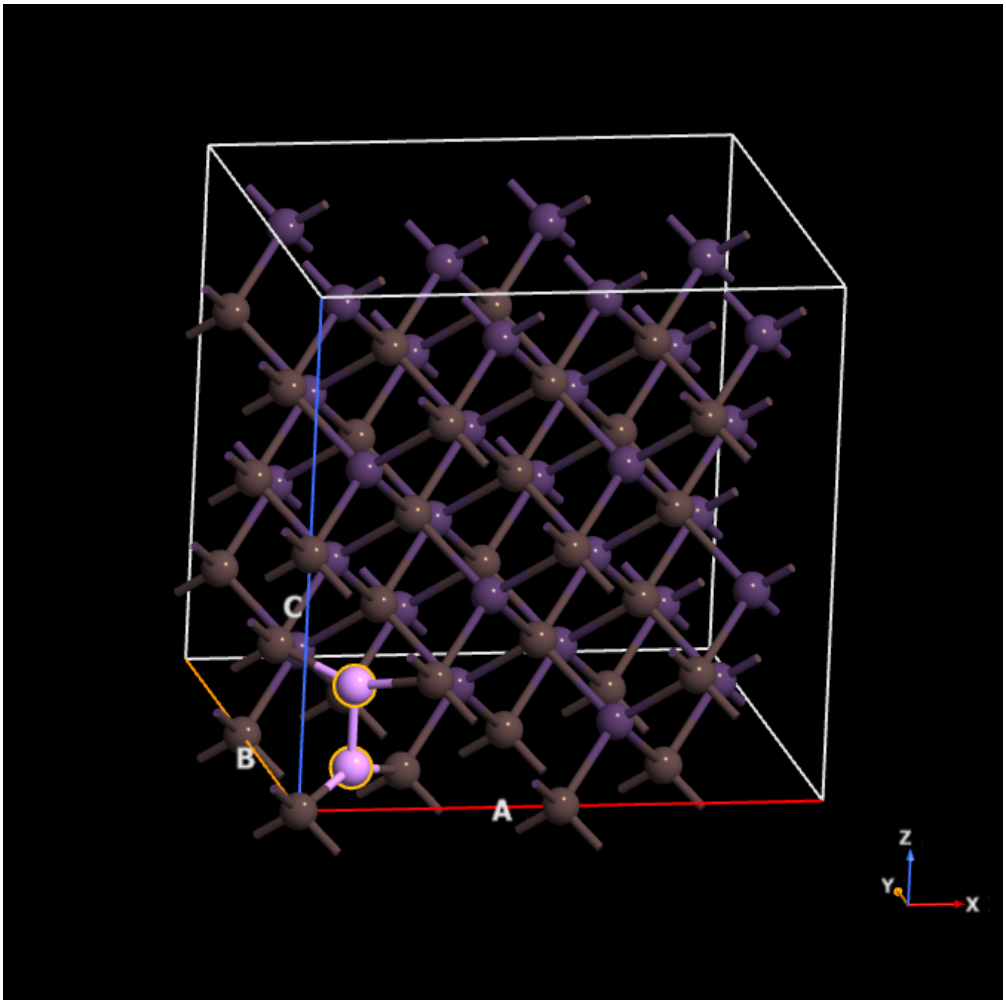


Fig. 4.38: Split-interstitial on GaAs. One As atom of the crystal is replaced with two atoms of the same species. This *dumbbell* defect is centered at the site of the original atom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters
top_n_configurations (*int*) – Number of most distinct configurations.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByLatticeSpecies(*element_list*)

Method for selecting unique defects by element. Valid only for defect types that are associated with a lattice site, e.g. Vacancies, but not Interstitials.

Parameters
element_list (*PeriodicTableElement* | list of *PeriodicTableElement*) – List of elements to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *SplitInterstitialGenerator*

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

- **point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.
- **keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`

•**select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)

Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)

Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

interstitialElement()

Returns
The element to add as interstitial defect.

Return type
PeriodicTableElement

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

splitDirection()

Returns
MillerIndices object containing 3 int [hkl].

Return type
MillerIndices

splitLength()

Returns
Length of the split.

Return type
PhysicalQuantity of type length

`symmetryTolerance()`

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

`uniqueDefects()`

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

`uniqueString()`

Return a unique string representing the state of the object.

Notes

+The *SplitInterstitialGenerator* object generates the symmetrically unique split-interstitial +defects for a given host material, by identifying the symmetrically equivalent atoms and positions. The *SplitInterstitialGenerator* object is derived from the *BaseDefectGenerator* object, and thus contains the common methods from that object.

To create a *SplitInterstitialGenerator* object, a specific host configuration must be given along with the interstitial element and a split direction. The split direction is specified as three Miller indices which give the direction relative to the unit cell. Optionally, a split length can also be given using the `split_length` argument. The lattice and interstitial atoms are then moved in opposite directions along the specified split direction, each by half the split length. Once the generator is created filters can be added so that only the desired subset of defects are generated.

The unique defects can be returned with the `uniqueDefects` method as *NamedPointDefect* objects, which can be used as input for a *ChargedPointDefectConfiguration* object. The defects and generators can also be given to a *Defects* object to create the basic defect configurations.

StaticHyperdynamics

`class StaticHyperdynamics(`*distortions*, *temperature=None*, *global_exponent=None*, *global_cut=None*, *potential_barrier=None*, *reaction_steps=None*, *optimize_new_state=None*, *measurement_frequency=None*, *tracked_atoms_tag=None*, *allow_no_variables=None*`)`

Hook function implementing static hyperdynamics.

Parameters

- **`distortions`** (Sequence of `HyperdynamicsBaseDistortion`) – The geometric distortions used to create the collective variable.
- **`temperature`** (*PhysicalQuantity* of type temperature) – The temperature of the simulation, used for calculating the hypertime. **Default:** 300 Kelvin
- **`global_exponent`** (*`float`*) – The exponent used to create the global distortion. **Default:** 6

- **global_cut** – The value of the global distortion at the transition state. This can be used to set the position of the transition state where individual distortions do not range from 0 to 1. One example of this is when using a bond distortion without a maximum bond length. **Default:** 1
- **potential_barrier** (*PhysicalQuantity* of type energy) – The barrier height at the transition state. **Default:** 0.1 eV
- **reaction_steps** (*int*) – The number of steps at the maximum collective variable before a reaction takes place. **Default:** 5000
- **optimize_new_state** (*bool*) – Optimize the configuration before setting new distortions after a reaction is detected. **Default:** True
- **measurement_frequency** (*int*) – The step frequency with which measurements are added to the trajectory. **Default:** 10
- **tracked_atoms_tag** (*str*) – Tag giving the indices of atoms for which the position is measured during the simulation. None results in no atoms being tracked. **Default:** None
- **allow_no_variables** (*bool*) – Continue in cases where no variables are found for acceleration. The simulation continues the number of reaction steps without bias before trying again to find variables to accelerate. **Default:** True

allowNoVariables()

Returns
True if the simulation continues in cases where no variables are found for acceleration.

Return type
bool

callInterval()

Returns
The call interval of this hook function.

Return type
int

distortions()

Returns
The geometric distortions used to create the collective variable.

Return type
Sequence of HyperdynamicsBaseDistortion

globalCut()

Returns
The value of the global distortion at the transition state.

Return type
float

globalExponent()

Returns
The global exponent used in the collective variable.

Return type
float

measurementFrequency()

Returns
The step frequency with which measurements are saved to the trajectory.

Return type
int

numberOfVariables()

Returns
The total number of geometric coordinates being accelerated. If the geometric coordinates are not set with a configuration None is returned.

Return type
int | None

optimizeNewState()

Returns
True if the configuration is optimized before setting new distortions.

Return type
bool

potentialBarrier()

Returns
The barrier height at the transition state.

Return type
PhysicalQuantity of type energy

reactionSteps()

Returns
The number of steps at the maximum collective variable before a reaction takes place.

Return type
int

temperature()

Returns
The simulation temperature.

Return type
PhysicalQuantity of type temperature

trackedAtomsTag()

Returns
The tag for the atoms that are tracked during the simulation.

Return type
str

uniqueString()

Return a unique string representing the state of the object.

Notes

Hyperdynamics is a method for sampling rare events using molecular dynamics. In hyperdynamics a bias potential is added to degrees of freedom separating different states. The bias potential aids in lowering the energy barrier between these states, allowing the system to sample different states more freely. In this way the biased degrees of freedom move at elevated temperature, while the rest of the system moves at the simulation temperature. The acceleration factor given to the system by the bias potential can be calculated as the ensemble average of the bias potential. When this boost factor is multiplied by the simulation time it gives the effective sampling time, known as the hypertime.

In using the hypertime to estimate reaction rates, it is important that the bias potential is not applied at the transition state. If bias is applied both at the reactant and transition state it obscures the relative free energy barrier between two states. This can lead to errors in estimations of the rate constant. This also applies to kinetic based processes such as diffusion.

StaticHyperdynamics is very similar to *CollectiveVariableHyperdynamics*[1]. Both methods construct a collective variable in the same way to which a bias potential is applied. In *StaticHyperdynamics* a simple linear bias potential is used rather than using Gaussians to build up an effective bias potential. This makes the simulation simpler at the cost of requiring some knowledge of the potential energy hypersurface around the transition state. In particular the heights of the likely transition state barrier is needed.

In *StaticHyperdynamics* the bias potential can be given as:

$$\Delta V(\eta) = \Delta V^{max} \left(1 - \eta\right)$$

Here ΔV^{max} is the maximum height of the bias potential and η is the collective variable. This bias potential places the largest bias at the equilibrium structure and decreases the bias to zero at the transition state. The maximum height of the bias potential is given with the `potential_barrier` keyword argument.

All other arguments function in the same way as they do in *CollectiveVariableHyperdynamics*. The same measurements are also added to the trajectory during the simulation.

StaticMonteCarlo

class StaticMonteCarlo(*time_step=None*, *temperature=None*, *pressure=None*)

Create a method for performing a pure Monte Carlo simulation in the Hybrid Monte Carlo framework

Parameters

- **time_step** (*PhysicalQuantity* of type time) – The nominal time step between Monte Carlo events. **Default:** 1 fs
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature of the Monte Carlo simulation **Default:** 300 Kelvin

monteCarloStep(*configuration*, *forces*, *stress*, *constraints=None*)

Placeholder for an empty dynamics step in hybrid Monte Carlo.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration on which the Monte Carlo step should be performed.
- **forces** (*PhysicalQuantity* of type energy/length) – The atomic forces for the given configuration.
- **stress** (*PhysicalQuantity* of type pressure) – The current stress in the unit cell.
- **constraints** (*list of type BaseConstraint | None*) – The list of constraints to apply.

[1] KristofăM. Bal and ErikăC. Neyts. Merging Metadynamics into Hyperdynamics: Accelerated Molecular Simulations Reaching Time Scales from Microseconds to Seconds. *Journal of Chemical Theory and Computation*, 11(10):4545–4554, 2015.

monteCarloTimeStep()

Returns
The nominal time step between Monte Carlo events.

Return type
PhysicalQuantity of type time

reservoirTemperature()

Returns
The simulation temperature.

Return type
PhysicalQuantity of type temperature

Notes

The *StaticMonteCarlo* class is used in the *HybridMonteCarlo()* method. There a dynamic method must be given that specifies the time and temperature associated with the dynamics steps. To perform a pure Monte Carlo simulation, which does not use dynamics steps, this class can be given as the method. This skips the dynamics cycle and only performs Monte Carlo cycles. This is useful in cases where the Monte Carlo method alone sufficiently samples the given ensemble and dynamics samples are not needed.

SteeredMolecularDynamics

class SteeredMolecularDynamics(*collective_variable*, *velocity*, *spring_constant*, *initial_target_cv_value=None*, *collective_variable_name=None*, *measurement_interval=None*)

Hook class to run steered MD by applying a moving spring to a given collective variable.

Parameters

- **collective_variable** (*DiffractionPeak* | object) – The collective variable (cv) on which the steered MD should act. The unit of the collective variable must be consistent with the units for `velocity` and `spring_constant`.
- **velocity** (*PhysicalQuantity* of type cv-unit / time) – The velocity at which the spring position moves.
- **spring_constant** (*PhysicalQuantity* of type energy / cv-unit ** 2) – The spring constant used to move the collective variable.
- **initial_target_cv_value** (float | *PhysicalQuantity* of type cv-unit | None) – The initial target value of the collective variable. If not specified the initial value at the first MD step is used.
- **collective_variable_name** (*str*) – The name with which the collective variable shows up in the measurements.
- **measurement_interval** (*int*) – The interval at which measurements of cv-value and bias potential are measured.

callInterval()

Returns
The call interval of this hook function.

Return type
int

measurements(*step*, *time*, *configuration*)

Measurement hook that records the collective variable value, bias, and target value.

Parameters

- **step** (*int*) – The MD step number.
- **time** (*PhysicalQuantity* of type time) – The MD time.
- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The current MD configuration.

Returns
A dictionary with the measurements.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Use the *DiffractionPeak* as collective variable in a *SteeredMolecularDynamics* to accelerate the crystallization in an amorphous cobalt silicide configuration using the MomentTensorPotential.

```
# Set up the diffraction peak intensity at q=2.1/Ang as collective variable.
# The diffraction intensity is only calculated for Cobalt atoms.
diffraction_peak = DiffractionPeak(
    cutoff_radius=10.0*Ang,
    q_value=2.1/Ang,
    included_atoms=[Cobalt],
)

# Set up the Steered MD with the diffraction peak intensity as collective variable.
# The object will be used as post-step-hook in the MD simulation.
hook_function = SteeredMolecularDynamics(
    collective_variable=diffraction_peak,
    velocity=300/1000/ps,
    spring_constant=0.5*eV,
    measurement_interval=50,
)

method = NVTNoseHoover(
    initial_velocity=MaxwellBoltzmannDistribution(
        temperature=1500.0*Kelvin
    ),
    reservoir_temperature=1500.0*Kelvin,
    thermostat_timescale=500.0*fs,
)

md_trajectory = MolecularDynamics(
    configuration=last_image_2,
    constraints=[
        FixCenterOfMass()
```

(continues on next page)

(continued from previous page)

```
    ],
    trajectory_filename='CoSi2_bulk_crystallization.hdf5',
    steps=1200000,
    post_step_hook=hook_function,
    measurement_hook=hook_function.measurements,
    log_interval=5000,
    method=method
)
```

crystallization_example.py

Notes

The *SteeredMolecularDynamics* object can be used as a `post_step_hook` in *MolecularDynamics()* or *TimeStampedForceBiasMonteCarlo()* to drive a given collective variable towards a specified value. This is done using a bias potential which mimics a harmonic spring acting on the collective variable (CV):

$$V^{bias}(\mathbf{R}) = \frac{1}{2}k(CV(\mathbf{R}) - v^{CV}t - CV_0)^2$$

where $\mathbf{R}$ is the set of atomic coordinates, k is the spring constant, $CV(\mathbf{R})$ is the collective variable as function of all cartesian coordinates, and v^{CV} is the velocity at which the spring moves. CV_0 is the initial CV value which is either taken as the value at $t = 0$ or can be specified explicitly via the keyword `initial_target_cv_value`. The values of the collective variable (actual value and target value), as well as the value of the bias potential are stored as measurements on the resulting `MDTrajectory` object and can be visualized in the **MovieTool**, along with the Movie of the simulation.

SteeredMolecularDynamics can be used with predefined or custom implementations of collective variables which implement the calculation of CV value and the derivatives with respect to atomic positions and cell vectors via the method `calculateCVAndDerivatives`. Currently, only the class *DiffractionPeak* is available as predefined collective variable.

Stiwe2Potential

class Stiwe2Potential(*particleType1*, *particleType2*, *p*, *A*, *B*, *gamma*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **p** (*float*) – Potential parameter.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type length**p*) – Potential parameter.
- **gamma** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for silicon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk silicon configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.00, 0.00, 0.00],
                          [ 0.25, 0.25, 0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name = 'StillingerWeber_Si_1985')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit))
```

(continues on next page)

(continued from previous page)

```
# Add the Stillinger-Weber two-body potential to the potential set
potentialSet.addPotential(Stiwe2Potential('Si', 'Si',
                                          r_cut=3.77118*Angstrom,
                                          p=4.0,
                                          A = 15.2855528754*eV,
                                          B = 11.6031922834*Ang**4,
                                          gamma = 2.0951*Angstrom))

# Add the Stillinger-Weber three-body potential to the potential set
potentialSet.addPotential(Stiwe3Potential('Si', 'Si', 'Si',
                                          gamma0=2.51412*Angstrom,
                                          gamma1=2.51412*Angstrom,
                                          l=45.5343*eV,
                                          cosTheta0=-0.333333333333,
                                          type=1,
                                          r_0=3.77118*Angstrom,
                                          r_1=3.77118*Angstrom,
                                          r_13=-1.0*Angstrom))

# Create the TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)
```

Notes

The *Stiwe2Potential* represents the two-body contribution of the Stillinger-Weber potential[1].

The Stillinger-Weber potential can be written as:

$$V = \sum_{i<j} v_2^{ij}(r_{ij}) + \sum_{i<j<k} \left[v_3^{ijk}(r_{ji}, r_{jk}, \theta_{ijk}) + v_3^{jik}(r_{ij}, r_{ik}, \theta_{jik}) + v_3^{ikj}(r_{ki}, r_{kj}, \theta_{ikj}) \right] .$$

The two-body part is calculated as

$$v_2^{ij}(r) = A \cdot (B \cdot r^{-p} - 1) \cdot \exp\left[\frac{\gamma}{r - r_{\text{cut}}} \right] .$$

For a more general form of the two-body term, see *GeneralStiwe2Potential*.

For the three-body part two types are available, which can be selected by the `type` argument.

Type 1:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1} \right] \cdot (\cos(\theta) - cos(\theta_0))^{\alpha} .$$

If not specified otherwise, α is set to 2, as in the original Stillinger-Weber potential[Page 3889, 1].

Type 2:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1} \right] \cdot (\cos(\theta) - cos(\theta_0)) \sin(\theta) \cos(\theta) .$$

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

Stiwe3Potential

class Stiwe3Potential(*particleType1*, *particleType2*, *particleType3*, *gamma0*, *gamma1*, *l*, *cosTheta0*, *type*, *r_0*, *r_1*, *r_13*, *alpha=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **gamma0** (*PhysicalQuantity of type length*) – Potential parameter.
- **gamma1** (*PhysicalQuantity of type length*) – Potential parameter.
- **l** (*PhysicalQuantity of type energy*) – Potential parameter.
- **cosTheta0** (*float*) – Potential parameter.
- **type** (*int*) – Potential parameter.
- **r_0** (*PhysicalQuantity of type length*) – Cutoff radius
- **r_1** (*PhysicalQuantity of type length*) – Cutoff radius
- **r_13** (*PhysicalQuantity of type length*) – Cutoff radius
- **alpha** (*float*) – Potential parameter

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for silicon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk silicon configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.00, 0.00, 0.00],
                          [ 0.25, 0.25, 0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name = 'StillingerWeber_Si_1985')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si', mass=28.0855*atomic_mass_unit))

# Add the Stillinger-Weber two-body potential to the potential set
potentialSet.addPotential(Stiwe2Potential('Si', 'Si',
                                          r_cut=3.77118*Angstrom,
                                          p=4.0,
                                          A = 15.2855528754*eV,
                                          B = 11.6031922834*Ang**4,
                                          gamma = 2.0951*Angstrom))

# Add the Stillinger-Weber three-body potential to the potential set
potentialSet.addPotential(Stiwe3Potential('Si', 'Si', 'Si',
                                          gamma0=2.51412*Angstrom,
                                          gamma1=2.51412*Angstrom,
                                          l=45.5343*eV,
                                          cosTheta0=-0.333333333333,
                                          type=1,
                                          r_0=3.77118*Angstrom,
                                          r_1=3.77118*Angstrom,
                                          r_13=-1.0*Angstrom))
```

(continues on next page)

(continued from previous page)

```
# Create the TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)
```

Notes

The *Stiwe3Potential* defines the three-body contribution of the Stillinger-Weber potential[1].

The Stillinger-Weber potential can be written as:

$$V = \sum_{i<j} v_2^{ij}(r_{ij}) + \sum_{i<j<k} [v_3^{ijk}(r_{ji}, r_{jk}, \theta_{ijk}) + v_3^{jik}(r_{ij}, r_{ik}, \theta_{jik}) + v_3^{ikj}(r_{ki}, r_{kj}, \theta_{ikj})] \,.$$

The two-body part is calculated as

$$v_2^{ij}(r) = A \cdot (B \cdot r^{-p} - 1) \cdot \exp\left[\frac{\gamma}{r - r_{\mathrm{cut}}}\right] \,.$$

For the three-body part two types are available, which can be selected by the `type` argument.

Type 1:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1}\right] \cdot (\cos(\theta) - cos(\theta_0))^{\alpha} \,.$$

If not specified otherwise, α is set to 2, as in the original Stillinger-Weber potential[Page 3892, 1].

Type 2:

$$v_3^{ijk}(r_{ji}, r_{jk}, \theta) = \lambda \exp\left[\frac{\gamma_0}{r_{ji} - r_0} + \frac{\gamma_1}{r_{jk} - r_1}\right] \cdot (\cos(\theta) - cos(\theta_0)) \sin(\theta) \cos(\theta) \,.$$

StoreOnDisk

class StoreOnDisk(*filename=None*, *mode=None*, *working_directory=None*)

Class indicating that the self energies should be cached to disk.

The given file will be used for both reading and writing self energies if mode is `Append`. If no filename is given, or mode is `ReadOnly`, newly calculated self energies are stored in a temporary cache file for the duration of the calculation (note that this file will be deleted at the end of the calculation).

Parameters

- **filename** (*str*) – The filename to read/write self energies to.
- **mode** (`Append` | `ReadOnly`) – Whether newly calculated self energies are added to the given file. With `Append`, new self energies are appended to the file. With `ReadOnly`, they are stored in a temporary file that is discarded at the end of the calculation. **Default:** `Append`
- **working_directory** (*str*) – The directory to store temporary cache files in. **Default:** The systems temporary directory.

uniqueString()

Return a unique string representing the state of the object.

[1] F.ăH. Stillinger and T.ăA. Weber. Computer simulation of local order in condensed phases of silicon. *Phys. Rev. B*, 31:5262–5271, Apr 1985. doi:10.1103/PhysRevB.31.5262.

Usage Examples

Basic examples on how to use *StoreOnDisk()* and other storage strategies can be found in the *Usage Examples* of *RecursionSelfEnergy*.

Temporarily save self energies on disk

Define that the self energies are saved on disk in a temporary file after the first iteration of the SCF loop and reused in the following iterations.

```
self_energy_calculator = RecursionSelfEnergy(
    storage_strategy=StoreOnDisk()
)
```

By default, a temporary cache file is created in the systems temporary directory. The location of the temporary cache file can also be specified.

```
self_energy_calculator = RecursionSelfEnergy(
    storage_strategy=StoreOnDisk(working_directoy='/home/user/')
)
```

Reuse self energies between calculations

Specify the path of the file that stores the calculated self energies after the first iteration of the SCF loop.

```
self_energy_calculator = RecursionSelfEnergy(
    storage_strategy=StoreOnDisk(filename='/home/user/selfenergycache.hdf5')
)
```

Note: Self energies are stored on disk using the HDF5 format. The `filename` provided to *StoreOnDisk()* can have any extension.

In subsequent calculations, the previously calculated self energies can be reused by providing the same path again.

Important: QuantumATK does not verify the validity of the self energies in the provided file. Providing a cache file containing self energies calculated for a different electrode will give incorrect results.

By default, self energies not present in the cache file are calculated and added to the cache file. This can be avoided by marking the cache file read-only.

```
self_energy_calculator = RecursionSelfEnergy(
    storage_strategy=StoreOnDisk(
        filename='/home/user/selfenergycache.hdf5'),
        mode=ReadOnly
        )
)
```

Notes

In each step of the SCF loop of a device calculation, the same self energies need to be evaluated.

The *StoreOnDisk()* storage strategy specifies that each self energy is calculated only once and stored on disk. The self energies are reused in subsequent iterations. This strategy can significantly improve performance, but requires sufficient disk space.

In calculations involving a large number of parallel processes, storing the self energies to disk is equally fast as the *SaveInMemory()* storage strategy inside a single node. However, when a custom filename is used, the performance can slightly slow down when multiple nodes are involved in the calculation. If a temporary file is used instead, there is no slowdown across nodes. After the self energies have been stored, the *StoreOnDisk()* and *SaveInMemory()* storage strategies provide a similar performance improvement regardless of the parameters used.

Note that the *StoreOnDisk()* storage strategy requires less memory than the *SaveInMemory()* strategy, and is therefore the recommended choice for memory-intensive calculations when there is plenty of disk space available.

StrainConfigurationHook

class StrainConfigurationHook(*strain_direction=None*, *strain_rate=None*, *call_interval=None*, *strain_interval=1*)

Pre-step hook apply strain to the configuration in a given cartesian direction during an MD simulation.

Parameters

- **strain_direction** (*str*) – The Cartesian direction to strain. May either be a uniaxial strain (e.g. x, y, z) or a shear strain (e.g. xy, zx, yz).
- **strain_rate** (*PhysicalQuantity* of type inverse time) – The amount of strain per unit time applied to the chosen axis.
- **call_interval** (*int*) – The number of MD steps between when the strain is applied to the configuration.
- **strain_interval** (*int*) – The number of MD steps between when the strain is applied to the configuration.

callInterval()

Returns
The call interval of this hook function.

Return type
int

strainDirection()

Returns
The strain direction.

Return type
str

strainRate()

Returns
The strain rate.

Return type
PhysicalQuantity of type inverse time

tensorIdx()

Returns
The tensor indices.

Return type
tuple

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run a 6 ps NVT MD simulation, with a strain rate of 0.25% per ps, while measuring the stress and strain at every MD step. These measurements are then used to plot a stress-strain curve.

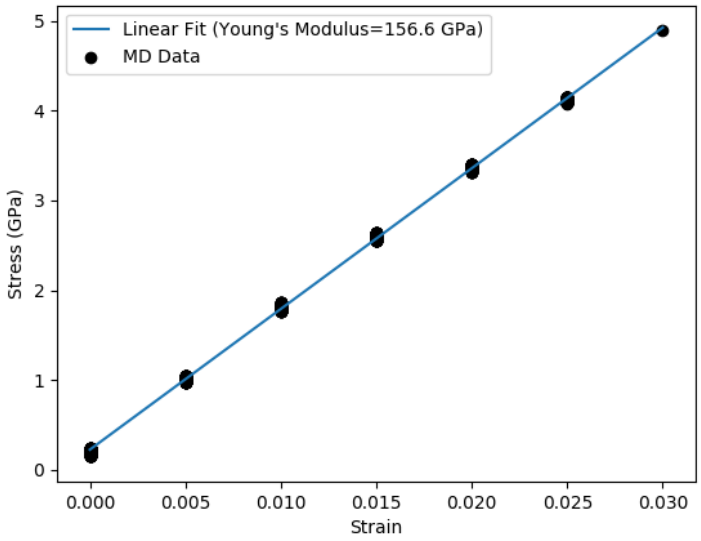
```
# Define a strain configuration hook.
strain_hook = StrainConfigurationHook(
    strain_direction='x',
    strain_rate=0.005 / ps,
    strain_interval=1000,
)

measurement_names = ['stress_xx', 'strain_xx']

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='stress_strain.hdf5',
    steps=6000,
    log_interval=1000,
    pre_step_hook=strain_hook,
    measurement_hook=MDMeasurement(measurement_names, call_interval=1),
    method=method,
)

bulk_configuration = md_trajectory.lastImage()

# Get stress strain values.
times, strains = md_trajectory.measurement('strain_xx')
times, stresses = md_trajectory.measurement('stress_xx')

# Perform linear fit.
polynomial_coefficients = numpy.polyfit(strains, stresses.inUnitsOf(GPa), 1)
youngs_modulus = polynomial_coefficients[0] * GPa

# Evaluate polynomial to plot best fit line.
stresses_fit = numpy.polyval(polynomial_coefficients, strains)

# Plot the fit.
import pylab
pylab.plot(
    strains,
```

(continues on next page)

(continued from previous page)

```
    stresses_fit,
    label="Linear Fit (Young's Modulus=%.1f GPa)" % youngs_modulus,
)

# Plot MD data.
pylab.scatter(
    strains,
    stresses.inUnitsOf(GPa),
    label='MD Data',
    color='k',
)

# Setup plot.
pylab.xlabel('Strain')
pylab.ylabel('Stress (GPa)')
pylab.legend()

# Save to file.
pylab.savefig('stress_strain.png')
```

The resulting plot will look like:



The full script can be downloaded: `stress_strain.py`

Stress Strain Calculations

In a stress-strain calculation the cell is extended or strained in one direction and the resulting stress in that direction is then recorded. In the elastic region, the stress ideally varies linearly with the strain. The derivative of the stress-strain curve is the Youngs modulus of the material. The *StrainConfigurationHook* enables you to easily calculate the Youngs modulus by extending the cell during the simulation in the specified direction. The stress during the molecular dynamics run can then be recorded using an *MDMeasurement* object. The *StrainConfigurationHook* is included in the molecular dynamics simulation by using the `pre_step_hook argument`. This argument takes functions or callable objects that are called before the coordinates are integrated in each molecular dynamics step.

Notes

- The strain is defined as the engineering strain, which is the relative change from the initial unit cell.
- For additional information on using hooks in molecular dynamics see the *notes section* of the *MolecularDynamics* reference manual entry.

StrainNiquetHamiltonianParametrization

class StrainNiquetHamiltonianParametrization(*basis_set*, *strain_parameters*)

Class representing the Hamiltonian parametrization for a Slater-Koster type tight binding model with onsite strain corrections.

Parameters

- **basis_set** (*SlaterKosterTable*) – An object describing the basis set used for the Slater-Koster calculation.
- **strain_parameters** (*StrainNiquetModelParameters*) – The parameters for the onsite strain model.

basisSet()

Returns the Slater Koster basis set used to construct the environmental independent component of the Hamiltonian and Overlap.

Returns
The basis set.

Return type
SlaterKosterTable

pairPotentials()

Return the pair potentials. If no pair potentials are available, it returns `None`.

Returns
The pair potentials.

Return type
PairPotential

strainParameters()

Returns
The parameters used to construct the strain correction.

Return type
StrainNiquetModelParameters

usingOrthogonalBasis()

Routine for determining if the Hamiltonian parametrization is using an orthogonal basis set.

Returns
True if the basis is orthogonal, False otherwise.

Return type
bool

Usage Examples

The following example shows how to setup a parametrization for Silicon and Germanium.

```
# Create a parametrization using built-in parameters for Silicon and Germanium.
hamiltonian_parametrization = StrainNiquetHamiltonianParametrization(
    basis_set=Niquet.SiGeH_Basis,
    strain_parameters=Niquet.SiGe_StrainParameters)

# Set up a semi-empirical calculator with this parametrization.
calculator = SemiEmpiricalCalculator(
    hamiltonian_parametrization=hamiltonian_parametrization)
```

Notes

This hamiltonian parametrization is an extension to a Slater-Koster nearest neighbr basis including onsite Hamiltonian matrix elements corrections for strained crystal[1].

The available parameter sets in QuantumATK can be found in the following table: *NRL style parameters for electronic structure and total energy calculations.*

StrainNiquetModelParameters

class StrainNiquetModelParameters

Container class to store parameters for Niquets onsite strain model, see StrainedNiquetHamiltonianParametrization.

addAlpha(*element*, *subshell*, *value*)

Add the volumetric parameter *alpha* for a given subshell and element.

Parameters

- **element** (*PeriodicTableElement*) – the element for which the parameter is defined.
- **subshell** (*str*) – for alpha parameters, the subshell for which the parameter is defined (s, p or d). For beta parameter, the subshell pair. (s, p or d). When multiple subshells with the same angular momentum are present, an integer suffix can be used (e.g. *s0*, *s1*).
- **value** (*PhysicalQuantity* of type energy.) – the value of the parameter.

[1] Y.ăM. Niquet, D.ăRideau, C.ăTavernier, H.ăJaouen, and X.ăBlase. Onsite matrix elements of the tight-binding Hamiltonian of a strained crystal: Application to silicon, germanium, and their alloys. *Phys. Rev. B*, 79:245201, Jun 2009. URL: https://link.aps.org/doi/10.1103/PhysRevB.79.245201, doi:10.1103/PhysRevB.79.245201.

addBeta0(*element*, *subshell_pair*, *value*)
Add the bond parameter *beta 0* for element and subshell pair.

Parameters

- **element** (*PeriodicTableElement*) – the element for which the parameter is defined.
- **subshell_pair** – the subshell pair for which the parameter is defined, for example *ss*, s0p0`, *s0s1* etc.
- **value** (*PhysicalQuantity* of type energy.) – the value of the parameter.

addBeta1(*element*, *subshell_pair*, *value*)
Add the bond parameter *beta 1* for element and subshell pair.

Parameters

- **element** (*PeriodicTableElement*) – the element for which the parameter is defined.
- **subshell_pair** – the subshell pair for which the parameter is defined, for example *ss*, s0p0`, *s0s1* etc.
- **value** (*PhysicalQuantity* of type energy.) – the value of the parameter.

addNearestNeighborDistance(*element1*, *element2*, *distance*)
Add the reference nearest-neighbor bond distance between two elements.

Parameters

- **element1** (*PeriodicTableElement*) – the element of the first atom in the bond.
- **element2** (*PeriodicTableElement*) – the element of the second atom in the bond.
- **distance** (*PhysicalQuantity* of type length.) – the reference bond distance.

parameters()
Return the parameters as a dictionary.

Stress

class Stress(*configuration*, *fix_strain=None*)
Class for calculating the stress of a configuration.

Parameters

- **configuration** (*BulkConfiguration*) – Configuration with a calculator that supports stress calculations.
- **fix_strain** (*FixStrain* | None) – Additional constraint parameter. **Default:** None

evaluate()

Returns
The stress for the given configuration. The shape of the stress tensor is (3,3).

Return type
PhysicalQuantity with the unit `eV * Angstrom**-3`

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate and print the stress tensor of a compressed GaAs crystal:

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4*Angstrom),
    elements=[Gallium, Arsenic],
    cartesian_coordinates=[[ 0.      ,  0.      ,  0.      ],
                           [ 1.413425,  1.413425,  1.413425]]*Angstrom
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=NumericalAccuracyParameters(
    k_point_sampling=(4, 4, 4))
    )

bulk_configuration.setCalculator(calculator)

#calculate the stress
stress = Stress(bulk_configuration)
nlprint(stress)
```

stress.py

Notes

Stress is given by the derivative of the total energy, E, with respect to a strain $\epsilon_{\alpha\beta}$,

$$\sigma_{\alpha\beta} = \frac{1}{V}\frac{dE}{d\epsilon_{\alpha\beta}}.$$

In this equation, V, is the volume of the system and the strain tensor displaces positions by

$$R^{\epsilon}_{\alpha} = \sum_{\beta}(\delta_{\alpha\beta} + \epsilon_{\alpha\beta})R_{\beta}.$$

Stress has unit energy per volume element, i.e. eV/Å ^{3}.

Substitutional

class Substitutional(*element*, *site_index=None*, *unit_cell_index=None*)

Defines a substitutional defect. Used as part of a *ChargedPointDefect* study.

Parameters

- **element** (*PeriodicTableElement*) – The element to substitute the atom at the selected site with.
- **site_index** (*int*) – The index of the atom in the bulk unit cell configuration to be substituted with the given element. **Default:** `0`.
- **unit_cell_index** (*sequence (size 3) of int*) – The index of the unit cell in the infinite crystal which the substitutional site belongs to. **Default:** `(0, 0, 0)`.

element()

Returns
The element used in the substitutional defect.

Return type
PeriodicTableElement

generateDefectConfiguration(*reference_bulk_configuration*, *supercell_repetitions*, *use_ghost=None*)

Generate the supercell with the embedded defect.

Parameters

- **reference_bulk_configuration** (*BulkConfiguration*) – The bulk unit cell configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.
- **use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
The supercell configuration with the embedded defect.

Return type
BulkConfiguration

siteIndex()

Returns
The index of the lattice site which the defect is referenced to.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

unitCellIndex()

Returns
The index of the unit cell in the infinite crystal which the reference site belongs to.

Return type
tuple (size 3) of int

Notes

The *Substitutional* object defines a substitution defect in a material.

A *Substitutional* can be added to a *NamedPointDefect* to define a defect for use in a *ChargedPointDefectConfiguration* calculation.

See *Defining the point defect* for a description of how to use this object as part of a *ChargedPointDefect* study.

SubstitutionalGenerator

class SubstitutionalGenerator(*host_configuration*, *substitutional_element*, *symmetry_tolerance=None*)

A class which describes the possible substitutional defects for a given host material.

Parameters

- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **substitutional_element** (*PeriodicTableElement*) – The element to set as substitutional defect.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters
top_n_configurations (*int*) – Number of most distinct configurations.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByLatticeSpecies(*element_list*)

Method for selecting unique defects by element. Valid only for defect types that are associated with a lattice site, e.g. Vacancies, but not Interstitials.

Parameters
element_list (*PeriodicTableElement* | list of *PeriodicTableElement*) – List of elements to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *SplitInterstitialGenerator*

filterByLatticeSpeciesAndDistinctConfigurations(*top_n_configurations*, *element*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters

- **top_n_configurations** (*int*) – Number of most distinct configurations.
- **element** (*PeriodicTableElement*) – Element to keep after filtering.

Returns
A filtered defect list.

Return type
SubstitutionalGenerator

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

- **point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.
- **keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.
- **resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`
- **in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`
- **select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`
- **defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** `1`

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)

Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)

Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

substitutionalElement()

Returns
The element to add as substitutional defect.

Return type
PeriodicTableElement

symmetryTolerance()

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

uniqueDefects()

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

uniqueString()
Return a unique string representing the state of the object.

Notes

The *SubstitutionalGenerator* object generates the symmetrically unique substitutional defects for a given host material, by identifying the symmetrically equivalent atoms and positions. In multi-element configurations, substitutions are ignored if the lattice element would not change. The *SubstitutionalGenerator* object is derived from the *BaseDefectGenerator* object, and thus contains the common methods from that object.

To create a *SubstitutionalGenerator* object, a specific host configuration must be given along with the substitution element. Once the generator is created, filters can be added so that only the desired subset of defects are generated.

The unique defects can be returned with the `uniqueDefects` method as *NamedPointDefect* objects, which can be used as input for a *ChargedPointDefectConfiguration* object. The defects and generators can also be given to a *Defects* object to create the basic defect configurations.

SurfaceBandstructure

class SurfaceBandstructure(*configuration*, *route=None*, *points_per_segment=None*, *kpoints=None*, *energies=None*, *projection_list=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*)

Constructor for the SurfaceBandstructure object.

Parameters

- **configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The one- or two-probe configuration with attached calculator for which the surface band structure should be calculated.
- **route** (*list of str*) – The route to take through the Brillouin-zone as a list of symmetry points of the unit cell, e.g. `['G', 'X', 'G']`. This option is mutually exclusive to `kpoints`. **Default:** Unit-cell dependent route.
- **points_per_segment** (*Positive int*) – The number of points per segment of the route. This option is mutually exclusive to `kpoints`. **Default:** `20`.
- **kpoints** (*list of lists of floats*) – A list of 3-dimensional fractional k-points at which to calculate the band structure e.g. `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. The shape is (K, 3) where K is the number of k-points. This option is mutually exclusive to `route`, and `points_per_segment`. **Default:** Unit-cell dependent route.
- **energies** (*PhysicalQuantity* of type energy.) – The energies for which the spectrum of the band structure should be calculated. **Default:** `numpy.linspace(-2.0, 2.0, 200)*eV`
- **projection_list** (*ProjectionList*) – A projection object list defining a projection. **Default:** If no projection list is specified, all orbitals will be used.
- **contributions** (`Left | Right | All`) – The density contributions to include in the spectral function. **Default:** `All`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel | AbsoluteEnergy`.) – Specifies the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small energy, used to move the density of states calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

contributions()

Returns
The choice of electrodes from which the contributions from the transmission states are used.

Return type
`Left | Right | All`

electrodeFermiLevels()

Returns the Fermi level(s) of the electrode(s). For a *DeviceConfiguration* the left and right electrode Fermi levels are returned, while for a *SurfaceConfiguration* only the Fermi level of the surface electrode is returned.

Returns
The Fermi level(s) of the electrode(s).

Return type
list of *PhysicalQuantity* of type energy

energies()

Returns
The energies of the energy spectrum of the surface band structure.

Return type
PhysicalQuantity of type energy.

energyZero()

The energy zero. It is zero if the energy zero parameter is equal to `AbsoluteEnergy`, otherwise it is determined by the electrode Fermi level(s). For a *DeviceConfiguration* it is the average of the left and right electrode Fermi levels. For a *SurfaceConfiguration* it is equal to the surface Fermi level.

Returns
The energy zero.

Return type
`PhysicalQuantity` of type energy

energyZeroParameter()

Returns
The specified choice of the energy zero.

Return type
`AverageFermiLevel | AbsoluteEnergy`

evaluate(*spin=None*)

Evaluate the spectral function.

Parameters
spin (`Spin`) – The spin component. **Default:** `Spin.Sum`

Returns
The spectral function. The shape is (*K*, *E*) where *E* is the number of energies and *K* is the number of k-points. If spin is `Spin.All` the shape will be (*S*, *K*, *E*) with *S* being the number of spin components which is 2 for a polarized calculation and 1 otherwise.

Return type
PhysicalQuantity of type inverse energy.

infinitesimal()

Returns
The infinitesimal used for calculating the surface band structure.

Return type
PhysicalQuantity of type energy.

kpoints()

Returns
The array of 3-dimensional fractional k-points at which the spectrum is calculated. The shape is (*K*, 3) where *K* is the number of k-points.

Return type
numpy.ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

projectionList()

The projections to include in the contribution to the surface band structure.

Returns
The *ProjectionList* object.

Return type
ProjectionList

route()

Returns
The route through the Brillouin-zone as a list of symmetry points of the unit cell.

Return type
list of str

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Notes

The *SurfaceBandstructure* object can be used to perform a detailed study of the projected density of states along specified k-point routes. The *SurfaceBandstructure* can be calculated for both device- and surface configurations as performed in[1] and[2], respectively.

Technically speaking the *SurfaceBandstructure* doesnt calculate the bandstructure, but rather the device density of states, $D(E, k)$ along a route of k-point in the plane perpendicular to the C-axis. As illustrated in[Page 3908, 1] and[2] the resulting figures resembles band structure plots, hence the name.

[1] J.ăM Marmolejo-Tejada, K.ăDolui, P.ăLazic, P.H. Chang, S.ăSmidstrup, D.ăStradi, K.ăStokbro, and B.ăK. Nikolic. Proximity Band Structure and Spin Textures on Both Sides of Topological-Insulator/Ferromagnetic-Metal Interface and Their Charge Transport Probes. *Nano Letters*, 17:5626–5633, 2017. doi:10.1021/acs.nanolett.7b02511.

[2] S.ăSmidstrup, D.ăStradi, J.ăWellendorff, P.ăA. Khomyakov, U.ăG. Vej-Hansen, M.-E. Lee, T.ăGhosh, E.ăJonsson, H.ăJonsson, and K.ăStokbro. First-principles Greens-function method for surface calculations: a pseudopotential localized basis set approach. *Phys. Rev. B*, 96:195309, 2017. URL: https://link.aps.org/doi/10.1103/PhysRevB.96.195309.

Usage Example

This example script shows how to perform a *SurfaceBandstructure* calculation on a silver (111) surface. This structure has a characteristic surface state around the Γ-point as also shown in[2] (Fig. 11).

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Left Electrode
# ------------------------------------------------------------

# Set up lattice
vector_a = [2.88902617589, 0.0, 0.0]*Angstrom
vector_b = [-1.44451308795, 2.50197006052, 0.0]*Angstrom
vector_c = [0.0, 0.0, 7.07663998448]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
left_electrode_elements = [Silver, Silver, Silver]

# Define coordinates
left_electrode_coordinates = [[-0.            ,  1.667980040348,  1.179540023274],
                              [-0.            ,  0.            ,  3.538420018102],
                              [ 1.444513087947,  0.833990020174,  5.89730001293␣
↪]]*Angstrom

# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )

# ------------------------------------------------------------
# Central Region
# ------------------------------------------------------------

# Set up lattice
vector_a = [2.88902617589, 0.0, 0.0]*Angstrom
vector_b = [-1.44451308795, 2.50197006052, 0.0]*Angstrom
vector_c = [0.0, 0.0, 44.7683399716]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
central_region_elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver,␣
↪Silver,
                           Silver, Silver, Silver]

# Define coordinates
central_region_coordinates = [[ -0.            ,   1.667980040348,   1.179540023274],
                              [ -0.            ,   0.            ,   3.538420018102],
                              [  1.444513087947,   0.833990020174,   5.89730001293 ],
```

(continues on next page)

(continued from previous page)

```
                                [ -0.            ,   1.667980040348,   8.256180007758],
                                [ -0.            ,   0.            ,  10.615060002586],
                                [  1.444513087947,   0.833990020174,  12.973939997414],
                                [ -0.            ,   1.667980040348,  15.332819992242],
                                [ -0.            ,   0.            ,  17.69169998707 ],
                                [  1.444513087947,   0.833990020174,  20.050579981898],
                                [ -0.            ,   1.667980040348,  22.409459976726],
                                [ -0.            ,   0.            ,  24.
↪768339971554]]*Angstrom

# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )

surface_configuration = SurfaceConfiguration(
    central_region,
    left_electrode,
    equivalent_electrode_length=7.07663998448*Angstrom,
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
#---------------------------------------
# Basis Set
#---------------------------------------
basis_set = [
    LDABasis.Silver_SingleZetaPolarized,
    ]

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = LDA.PZ

#---------------------------------------
# Numerical Accuracy Settings
#---------------------------------------
left_electrode_k_point_sampling = MonkhorstPackGrid(
    na=9,
    nb=9,
    nc=231,
    )
left_electrode_numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=45.0*Hartree,
    k_point_sampling=left_electrode_k_point_sampling,
    )

device_k_point_sampling = MonkhorstPackGrid(
```

(continues on next page)

(continued from previous page)

```
    na=9,
    nb=9,
    nc=231,
    )
device_numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=45.0*Hartree,
    k_point_sampling=device_k_point_sampling,
    )

#----------------------------------------
# Poisson Solver Settings
#----------------------------------------
left_electrode_poisson_solver = FastFourier2DSolver(
    boundary_conditions=[[PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()],
                         [PeriodicBoundaryCondition(),PeriodicBoundaryCondition()]]
    )

#----------------------------------------
# Contour Integral Settings
#----------------------------------------
equilibrium_contour = SemiCircleContour(
    integral_lower_bound=1.17597836274*Hartree,
    circle_points=30,
    )
contour_parameters = ContourParameters(
    equilibrium_contour=equilibrium_contour,
    )

#----------------------------------------
# Electrode Calculators
#----------------------------------------
left_electrode_calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=left_electrode_numerical_accuracy_parameters,
    poisson_solver=left_electrode_poisson_solver,
    )

#----------------------------------------
# Device Calculator
#----------------------------------------
calculator = DeviceLCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=device_numerical_accuracy_parameters,
    contour_parameters=contour_parameters,
    electrode_calculators=
        [left_electrode_calculator, left_electrode_calculator],
    )

surface_configuration.setCalculator(calculator)
```

(continues on next page)

(continued from previous page)

```
nlprint(surface_configuration)
surface_configuration.update()
nlsave('Surface Silver 111 LDA.hdf5', surface_configuration)

# ------------------------------------------------------------
# Surface Bandstructure
# ------------------------------------------------------------
surface_bandstructure = SurfaceBandstructure(
    configuration=surface_configuration,
    route=['X', 'G', 'Y'],
    points_per_segment=150,
    energies=numpy.linspace(-1.5, 1.5, 300)*eV,
    projection_list=ProjectionList(atoms=All),
    contributions=All,
    self_energy_calculator=RecursionSelfEnergy(storage_strategy=NoStorage()),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-03*eV,
    )
nlsave('Surface Silver 111 LDA.hdf5', surface_bandstructure)
nlprint(surface_bandstructure)
```

surface_bandstructure.py

Projections

When constructing the *SurfaceBandstructure* it is possible to specify a *ProjectionList* object with information about which atoms and orbital shells to project on. By default all atoms and orbitals are included. If only the contribution from the two outermost atoms should be included, the following should be set in the above script:

```
projection_list=ProjectionList(atoms=[9, 10])
```

when setting up the *SurfaceBandstructure* object.

SurfaceConfiguration

class SurfaceConfiguration(*central_region*, *electrode*, *equivalent_electrode_length=None*)

A one-probe configuration consisting of a central region coupled to an electrode.

Parameters

- **central_region** (*BulkConfiguration*) – The surface region.
- **electrode** (*BulkConfiguration*) – The bulk region below the surface.
- **equivalent_electrode_length** (*PhysicalQuantity* of type length **Default:** Length of the actual electrode given.) – The length to use for the equivalent electrode region in the central region.

addBonds(*bond_list*)

Add bonds.

Parameters
bond_list (*numpy.ndarray*) – The list of bonds to add.

addTags(*tags*, *indices=None*)

Add a set of tags to atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms.
- **indices** (*list | int | None*) – The list of indices to match atoms against. **Default:** All indices.

atomicMasses()

Returns
The masses of the atoms in the configuration.

Return type
PhysicalQuantity of type mass

atomicNumbers()

Returns
The list of atomic numbers associated with the elements.

Return type
list of ints

bonds()

Returns
An array with the the two atom indices for each bond in the central region along with the vector which periodic images this bond connects.

Return type
array

bravaisLattice()

Returns
The bravais lattice of the central region.

Return type
BravaisLattice

calculator()

Returns
The calculator attached to the configuration, i.e. the calculator that will be used for both simulation and analysis.

Return type
Calculator

cartesianCoordinates()

The Cartesian coordinates of the atoms in the the central region of the configuration.

Returns
The Cartesian coordinates.

Return type
PhysicalQuantity of type length

centralRegion()

The central region of the surface.

Returns
The central region.

Return type
BulkConfiguration

coarseGrainDescriptors()

Returns
The list of either elements or coarse grain particles for each element. Isotopes are returned as just their base element, UnitedAtoms and Particle are given as their specific type.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

copy()

Returns
A copy of the current configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndDeleteAtoms(*indices*)

Create a new configuration by deleting some atoms from this configuration.

Parameters
indices (*list of int*) – The indices of the atoms to delete.

Returns
The configuration with some atoms deleted.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndMerge(*other*)

Create a new configuration by merging this configuration with another configuration.

Parameters
other (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The other configuration.

Returns
The merged configuration.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

copyAndShiftAtoms(*displacement*, *indices=None*)

Create a new configuration with some atoms translated.

Parameters

- **displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.
- **indices** (*list of int*) – The indices to shift. **Default:** All.

Returns
The configuration with the translation applied.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

crossSection()

Return the cross-sectional area of the central region.

Returns
The cross-sectional area.

Return type
float

deleteAtoms(*indices*)

Delete the specified atoms.

Parameters
indices (*array of int*) – The indices to delete.

deleteBonds(*bond_list=None*, *pair_selection=None*)

Delete bonds connected to atomic indices.

Parameters

- **bond_list** (*A two-dimensional sequence*) – The pairs of bondes indices.
- **pair_selection** (*list | None*) – Specifies two groups between which bonds are delete. Selectable groups are elements, index lists, tag names, or None (all atoms).

dielectricRegions()

Returns
The dielectric regions in the central region.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

electrodeDisplacement()

The displacement of the BravaisLattice of the electrode in the C-direction in order to match the BravaisLattice of the central_region.

Returns
The electrode displacement.

Return type
PhysicalQuantity of type length

electrodes()

The electrode of the surface as a list.

Returns
The electrode belonging to this configuration.

Return type
list of *BulkConfiguration*

electrodesDisplacement()

The displacement of the BravaisLattice of the electrode in the C-direction in order to match the BravaisLattice of the central_region.

Returns
The electrode displacement.

Return type
PhysicalQuantity of type length

electrodesWithCalculator()

Returns
A copy of the electrodes of the configuration with attached electrode calculator. If the configuration is updated, the calculators include a copy of the updated state.

Return type
list of *BulkConfiguration*

elements()

Returns
The elements in configuration.

Return type
list of *PeriodicTableElement*

externalPotential()

Returns
The external potential present in the central region.

Return type
AtomicShift | *AtomicCompensationCharge*

findBonds(*fuzz_factor=1.1*, *pair_selection=None*)
Find bonds in the configuration according to the combined covalent radii of the element pairs, multiplied with a fuzz factor. Optionally, find bonds only between two specified sub-groups of atoms. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

- **fuzz_factor** (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond.
- **pair_selection** (list(2) of type *PeriodicTableElement*, list of int, or str.) – Specifies two groups between which bonds are detected. Selectable groups are elements, index lists, tag names, or None (all atoms). By default bonds between all atoms in the configuration are taken into account.

fixedSpinDirections()

Returns
The fixed spin directions for the configuration.

Return type
FixedSpin | None

fractionalCoordinates()

Returns
The fractional coordinates of the central region.

Return type
array of floats

generateShifts()
Method for generating a list of origin shifts along two lattice directions.

It will create a list of 9 shifts of the supercell origin (-1 to 1) in x and y periodic directions.

Returns
The array of shifts.

Return type
PhysicalQuantity of type length

ghostAtoms()

Returns
The list of ghost atoms of the central region.

Return type
list of ints

improperDihedralIndices()

Returns
The list of atom indices for each improper dihedral or None if no improper dihedrals are defined. Improper dihedrals are mainly used in bonded force fields.

Return type
numpy array | None

indices()

Returns
The indices of the atoms.

Return type
list of int

indicesFromIsotopes(*isotopes*)

Parameters
isotopes (*list of type PeriodicTableElement or Isotope*) – The isotopes to select.

Returns
The indices of the selected isotopes.

Return type
list of type int

indicesFromTags(*tags=None*)

List the indices associated with a given collection of tags.

Parameters
tags (*list | str*) – A list of tags for which all matching indices should be extracted.

Returns
The list of indices corresponding to the specified tag name(s).

Return type
list of ints

magneticMoments()

Returns
The magnetic moments of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type Bohr magneton.

merge(*other*)

Add all atoms from a different configuration.

NOTE: For surface this method shifts the atoms of the other configuration to the lateral center of the cell and to the upper end in z-direction.

Parameters
other (*AtomicConfiguration*) – A different AtomicConfiguration.

metallicRegions()

Returns
The metallic regions in the central region.

Return type
list of *BoxRegion* | *SphereRegion* | *TubeRegion*

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The configuration information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AtomicConfiguration object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`.

numberOfAtoms()

Returns
The total number of atoms in the configuration.

Return type
int

partialCharges(*indices=None*)

Get the list of partial atomic charges that can be used for representing electrostatic interactions in ATK-ForceField.

Parameters
indices (*list | int | None*) – The indices for which to return the partial charges. **Default:** All indices.

Returns
A PhysicalQuantity array of the atomic partial charge for each atom.

Return type
PhysicalQuantity of type charge | None

particleDescriptors()

Returns
The list of particle descriptors for each atom. This should return the exact same list as was given in the constructor argument elements.

Return type
list of type ParticleDescriptor or *PeriodicTableElement*

static periodicBoundaries()

Returns
The periodic boundary conditions of the configuration.

Return type
list

primitiveVectors()

Returns
The primitive lattice vectors.

Return type
PhysicalQuantity of type length

removeTags(*tags=None*, *indices=None*, *purge=False*)

Remove a set of tags from atoms matching a collection of indices.

Parameters

- **tags** (*list | str*) – The list of tags to add to matching atoms. **Default:** All tags.
- **indices** (*list | int*) – The list of indices to match atoms against. **Default:** All indices.
- **purge** (*bool*) – When removing tags from the configuration, delete the tag completely when not associated with any atoms anymore. **Default:** False

repeat(*na=1*, *nb=1*, *nc=1*, *stack_systems=False*)

Repeat the derived class with the integer values na, nb, and nc along the three primitive unit cell vectors. The repeated system is constructed with a cell of the type UnitCell.

Parameters

- **na** (*int*) – The repetition along the a-axis. **Default:** 1.
- **nb** (*int*) – The repetition along the b-axis. **Default:** 1.
- **nc** (*int*) – The repetition along the c-axis. **Default:** 1.
- **stack_systems** (*bool*) – If True the basis atoms are repeated as a unit, i.e. a0 and b0 are repeated as: a0,b0,a1,b1, If False the basis atom are repeated individually, i.e. a0 and b0 are repeated as: a0,a1,, b0,b1, **Default:** True.

Returns
The repeated system.

Return type
DeviceConfiguration | *SurfaceConfiguration*

scalePartialCharges(*scale_factor*, *indices=None*)

Scale the partial charges with a given scale factor. These partial charges are used with Forcefield calculators.

Parameters

- **scale_factor** (*float*) – The factor for scaling charges.
- **indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

setBonds(*bond_list*, *skip_checks=False*)

Set the bonds on the configuration. The bonds are primarily used in to set the topology of bonded potentials in the TremoloX-calculator.

Parameters

•**bond_list** (*list(n, 2) | list(n, 5) | None*) – A list which contains for each bond the indices of the two connected atoms. Optionally, three more integers can be specified for each bond, which must be between -1 and 1, and which denote to which neighboring image cell the bond is connected. Without these additional indices, the minimum image convention is obeyed.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value.

setCalculator(*calculator*, *initial_state=None*, *initial_spin=None*)

Attach a *Calculator* to the configuration which will be used in calculations involving the configuration.

Parameters

•**calculator** (*Calculator*) – The calculator object that should be attached to the configuration.

•**initial_state** (*SurfaceConfiguration* with a calculator | None) – The initial state to be used for this configuration. **Default:** No initial state.

•**initial_spin** (*InitialSpin* | None) – The initial InitialSpin object to be used for this configuration. **Default:** `InitialSpin()`, except when used with a calculator of type *HuckelCalculator*, *SlaterKosterCalculator*, *SemiEmpiricalCalculator*, *DeviceHuckelCalculator*, *DeviceSlaterKosterCalculator*, *DeviceSemiEmpiricalCalculator*. In this case `InitialSpin([0., 0., ...])` is used, i.e. the scaled spins for each atom are set to 0.

setCartesianCoordinates(*cartesian_coordinates*, *indices=None*, *skip_checks=False*)

Set the Cartesian coordinates of the atoms in the central region. Changes in the coordinates of the electrode extension will be synchronized with the corresponding coordinates in the electrodes.

Parameters

•**cartesian_coordinates** (*PhysicalQuantity* of type length) – The new coordinates of the atoms in each image.

•**indices** (*list*) – The indices of the atoms to set the positions of. **Default:** All indices.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False

setDielectricRegions(*dielectric_regions*)

Set the dielectric regions for the central region.

Parameters
dielectric_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of dielectric regions to set.

setExternalPotential(*external_potential*)

Set an external potential on the configuration that will be used in calculations involving the configuration.

Parameters
external_potential (*AtomicShift* | *AtomicCompensationCharge*) – The external potential to apply.

setImproperDihedralIndices(*improper_dihedral_indices*)

Set the list of atom indices for each improper dihedral in bonded force fields.

Parameters
improper_dihedral_indices (*list or array with shape (m, 4) | None*) – The list of the 4 indices defining the connectivity for each improper dihedral or None to delete the current dihedral connectivity.

setMagneticField(*magnetic_field*)

Set local magnetic field. The spins will be forced to point in the directions given by the magnetic_field object. The magnetic field can be defined for each atom. This only has an effect for Noncollinear or Spinorbit calculations.

Parameters
magnetic_field (*FixedSpin*) – The magnetic field to be applied.

setMagneticMoments(*magnetic_moments=None*, *skip_checks=False*)

Function to set magnetic moments on the configuration.

Parameters

•**magnetic_moments** (*PhysicalQuantity* of type Bohr magneton.) – The magnetic_moments to set on the configuration. Has the dimensionality nx3. **Default:** None.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

setMetallicRegions(*metallic_regions*)

Set the metallic regions for the central region.

Parameters
metallic_regions (list of *BoxRegion* | *SphereRegion* | *TubeRegion*) – The list of metallic regions to set.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setPartialCharges(*charge_list*, *indices=None*, *skip_checks=False*, *update_calculator=True*)

Set the partial charges on the configuration. The partial charges are used primarily to model electrostatic interactions in the TremoloX-calculator.

Parameters

•**charge_list** (*PhysicalQuantity of type charge | None*) – A list of atomic partial charges which contains a charge for each atom.

•**indices** (*list | int | None*) – The indices for which to set the partial charges. **Default:** All indices.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

•**update_calculator** (*bool*) – Whether to update and attached Forcefield calculator with the new charges. **Default:** True.

setVelocities(*velocities=None*, *skip_checks=False*)

Function to set velocities on the configuration.

Parameters

•**velocities** (*PhysicalQuantity* of type velocity | None) – The velocities to set on the configuration. Has the dimensionality nx3. **Default:** None.

•**skip_checks** (*bool*) – Skip argument type checking and just directly assign the value. **Default:** False.

shiftAtoms(*displacement*, *indices=None*, *skip_checks=False*)

Translate some atoms.

Parameters

•**displacement** (*PhysicalQuantity* of type length) – The displacement that should be applied to the atom positions.

•**indices** (*list of int*) – The indices to shift. **Default:** All.

•**skip_checks** (*bool*) – True, if all consistency checks should be skipped.

shiftPartialCharges(*total_charge*, *indices=None*)

Shift the partial charges so that their sum is the given total charge value. These partial charges are used with Forcefield calculators.

Parameters

- **total_charge** (*PhysicalQuantity of type charge*) – The new total charge.
- **indices** (*list | int | None*) – The indices for which to set the total charge. **Default:** All indices.

symbols()

Returns
The element symbols of the configuration.

Return type
list of str

tags(*indices=None*)

List the tags associated with a given collection of indices. The list returned is the set union of tags associated with the given indices. If no collection of indices is provided, then all tags on the configuration are returned.

Parameters
indices (*list | int*) – The indices to check. **Default:** All indices.

Returns
The set union of tags present on the provided indices.

Return type
set

transverseElectrodeRepetitions()

Returns
The transverse electrode repetitions; always [[1, 1]] for a surface.

Return type
list (size 1) of list (size 2) of int

uniqueElements(*ordered=False*)

Parameters
ordered (*bool*) – If the elements should be returned in ascending order by atomic number.

Returns
The unique elements contained in the configuration.

Return type
list of *PeriodicTableElement*

uniqueString()

Return a unique string representing the state of the object.

update(*force_restart=False*)

A self-consistent solution is generated, using the currently set calculator.

Parameters
force_restart (*bool*) – Force the self-consistent calculation to restart. **Default:** False

velocities()

Returns
The velocities of the atoms. Has the dimensionality nx3.

Return type
PhysicalQuantity of type velocity

Notes

ATK recognizes four types of atomic geometries:

•Molecules (*MoleculeConfiguration*)

•Bulk (*BulkConfiguration*)

•One Probe (*SurfaceConfiguration*)

•Two Probe (*DeviceConfiguration*)

SurfaceProcessSimulation

class SurfaceProcessSimulation(*substrate*, *filename*, *object_id*, *random_seed*, *temperature*, *thermostat_method=None*, *fixed_thickness=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *thermostat_thickness=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *adaptive_thermostat_layer=None*, *distance_above_surface=None*, *log_interval=None*, *trajectory_interval=None*, *log_filename_prefix=None*, *active_learning=None*, *fixed_directions=None*, *max_force_threshold=None*)

Set up a surface process simulation to study deposition, etching, or sputtering.

Parameters

•**substrate** (*BulkConfiguration*) – The initial surface (bulk slab model).

•**filename** (*str*) – The full or relative path to save the restart information to.

•**object_id** (*str*) – The object id to use when saving the restart information.

•**random_seed** (*int*) – The seed for the random number generator.

•**temperature** (*PhysicalQuantity* of type temperature) – The thermostat temperature.

•**thermostat_method** (NVTBerendsen | NVTNoseHoover | Langevin) – The class that should be used for the thermostat method in the molecular dynamics simulation. **Default:** NVTNoseHoover

•**fixed_thickness** (*PhysicalQuantity* of type length | str) – The length of the bottom fixed layer of the slab. Alternative to a length value, one can also give a tag name, which means that all atoms in the tagged group will be treated as fixed atoms. **Default:** 10 * Angstrom.

•**thermostat_thickness** (*PhysicalQuantity* of type length | str) – The length of the thermostat layer of the slab (above the fixed layer). Alternative to a length value, one can also give a tag name, which means that all atoms in the tagged group will be coupled to the thermostat. **Default:** 10 * Angstrom.

•**adaptive_thermostat_layer** (*bool*) – Whether to grow (or shrink) the length of the thermostat layer such that the width of the reactive layer remains approximately the same as new layers are deposited (or removed). The length of the reactive layer is taken from the initial configuration (i.e. total_initial_thickness - fixed_thickness - thermostat_thickness) **Default:** False

•**distance_above_surface** (*PhysicalQuantity* of type length) – The distance above the surface where new atoms are inserted into the system. **Default:** 10 * Angstrom.

•**log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. **Default:** 100.

•**trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. **Default:** 100.

•**log_filename_prefix** (*str*) – An optional prefix to the log filenames.

•**active_learning** (ActiveLearningSimulation) – Can be used to specify an ActiveLearningSimulation object, which is then used to run the MD or FBMC simulations while training an MTP. The calculator on the configuration is ignored in this case. Note, this does not support restarting from file.

•**fixed_directions** (*tuple of int*) – Can be used to choose the directions of fixed_thickness and thermostat_thickness for 2D materials. This is to specify the cell dimensions along which the edges are chosen to be fixed.

•**max_force_threshold** (*PhysicalQuantity* of type energy/length) – The maximum allowed force to run MD simulation with single time step. If maximum force in a MD simulation is larger than the threshold force, multiple time steps MD simulation is performed. **Default:** None

activeLearning()

Returns
The active learning object, if specified, otherwise None.

Return type
ActiveLearningSimulation | None

addAtom(*atom*, *md_time*, *time_step*, *kinetic_energy*, *incident_angle*, *enforce_temperature=None*, *label=None*, *md_hooks=None*, *post_md_hooks=None*, *log_interval=None*, *trajectory_interval=None*, *stability_checks=False*, *max_attempts=10*)

Add an atom to the simulation.

Parameters

•**atom** (PeriodicTableElement | *ParticleDescriptor*) – The element to add to the simulation.

•**md_time** (*PhysicalQuantity* of type time) – The total MD simulation time for this event.

•**time_step** (*PhysicalQuantity* of type time) – The time step of the MD simulation.

•**kinetic_energy** (*PhysicalQuantity* of type energy or temperature) – The kinetic energy of the molecule when added to the system. If the kinetic energy is specified in temperature units, the translational kinetic energy will be drawn from a Maxwell-Boltzmann distribution.

•**incident_angle** (*PhysicalQuantity* of type angle) – The angle between the velocity vector of the added molecule and the surface normal. An *incident_angle* of 0 degrees produces a molecule headed directly at the surface.

•**enforce_temperature** (*bool*) – Rescale the velocities of the system to ensure the instantaneous temperature is exactly the requested temperature in the *MaxwellBoltzmannDistribution*. **Default:** False.

•**label** (*str*) – A short label describing this trajectory.

•**md_hooks** (*HookFunctions* | None) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the MD forces evaluation. It can also contain measurement hooks, which are called at the end of each MD step after all constraints have been applied.

•**post_md_hooks** (*list of type SPSHook*) – Hook functions to call after the molecular dynamics run.

•**log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.

•**trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.

•**stability_checks** (*bool*) – Add check during molecular dynamics to make sure the simulation is stable and producing meaningful results. If an unstable event is encountered, the simulation will be repeated with slightly different initial conditions.

- **max_attempts** (*int*) – Deposition events can fail when the molecular dynamics becomes unstable, or in the case of 2D materials the molecule is deposited onto the fixed region. If these cases are detected then the event is re-run. This parameter specifies the maximum number of attempts at running an event before the simulation is terminated. For general MD stability checks to be performed the argument `stability_checks` must be set to `True`

addMolecule(*molecule*, *md_time*, *time_step*, *kinetic_energy*, *incident_angle*, *internal_energy=None*, *enforce_temperature=None*, *label=None*, *md_hooks=None*, *post_md_hooks=None*, *log_interval=None*, *trajectory_interval=None*, *stability_checks=False*, *max_attempts=10*)

Add a molecule to the simulation.

Parameters

- **molecule** (*MoleculeConfiguration*) – The molecule to add to the simulation.
- **md_time** (*PhysicalQuantity* of type time) – The total MD simulation time for this event.
- **time_step** (*PhysicalQuantity* of type time) – The time step of the MD simulation.
- **kinetic_energy** (*PhysicalQuantity* of type energy or temperature) – The kinetic energy of the molecule when added to the system. If the kinetic energy is specified in temperature units, it is treated as thermal kinetic energy and will be drawn from a *MaxwellBoltzmannDistribution*. Rotation and vibration degrees of freedom are discarded.
- **incident_angle** (*PhysicalQuantity* of type angle) – The angle between the velocity vector of the added molecule and the surface normal. An *incident_angle* of 0 degrees produces a molecule headed directly at the surface.
- **internal_energy** (*PhysicalQuantity* of type temperature) – The internal kinetic energy of the molecule when added to the system drawn from a *MaxwellBoltzmannDistribution*. The translational kinetic energy component is replaced by the value set for the kinetic_energy parameter.
- **enforce_temperature** (*bool*) – Rescale the velocities of the system to ensure the instantaneous temperature is exactly the requested temperature in the *MaxwellBoltzmannDistribution*. Applies to both kinetic- and internal energy. **Default:** False.
- **label** (*str*) – A short label describing this trajectory.
- **md_hooks** (*HookFunctions* | None) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the MD forces evaluation. It can also contain measurement hooks, which are called at the end of each MD step after all constraints have been applied.
- **post_md_hooks** (*list of type SPSHook*) – Hook functions to call after the molecular dynamics run.
- **log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.
- **trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.
- **stability_checks** (*bool*) – Add check during molecular dynamics to make sure the simulation is stable and producing meaningful results. If an unstable event is encountered, the simulation will be repeated with slightly different initial conditions.
- **max_attempts** (*int*) – Deposition events can fail when the molecular dynamics becomes unstable, or in the case of 2D materials the molecule is deposited onto the fixed region. If these cases are detected then the event is re-run. This parameter specifies the maximum number of attempts at running an event before the simulation is terminated. For general MD stability checks to be performed the argument `stability_checks` must be set to `True`

addSequence(*molecule*, *number_of_events*, *md_time*, *time_step*, *mean_kinetic_energy*, *mean_incident_angle*, *std_kinetic_energy=None*, *std_incident_angle=None*, *internal_energy=None*, *enforce_temperature=None*, *relaxation_time=None*, *relaxation_time_step=None*, *fbmc_time=None*, *fbmc_displacement=None*, *optimize_geometry=None*, *label=None*, *md_hooks=None*, *post_md_hooks=None*, *log_interval=None*, *trajectory_interval=None*, *stability_checks=None*, *max_attempts=10*)

Add a series of events with the given molecule or atom.

Parameters

•**molecule** (PeriodicTableElement | ParticleDescriptor | *MoleculeConfiguration*) – The molecule to add to the simulation. This may either be a MoleculeConfiguration or an element.

•**number_of_events** (*int*) – The number of cycles in which a new molecule / atom is added to the simulation.

•**md_time** (*PhysicalQuantity* of type time) – The MD simulation time for the MD simulation of the initial process, where a molecule is launched at the substrate.

•**time_step** (*PhysicalQuantity* of type time) – The time step of the MD simulation of the initial process, where a molecule is launched at the substrate.

•**mean_kinetic_energy** (*PhysicalQuantity* of type energy or temperature) – The average kinetic energy of the molecule or atom when added to the system. If the kinetic energy is specified in temperature units, it is treated as thermal kinetic energy and will be drawn from a *MaxwellBoltzmannDistribution*. For molecules, rotation and vibration degrees of freedom need to be set separately via the internal_energy parameter.

•**mean_incident_angle** (*PhysicalQuantity* of type angle) – The average angle between the velocity vector of the added molecule and the surface normal. An *incident_angle* of 0 degrees produces a molecule headed directly at the surface.

•**std_kinetic_energy** (*PhysicalQuantity* of type energy) – The standard deviation of the kinetic energy. If the mean_kinetic_energy is specified in temperature units, std_kinetic_energy is automatically set to 0 since the kinetic energy will follow a *MaxwellBoltzmannDistribution*.

•**std_incident_angle** (*PhysicalQuantity* of type angle) – The standard deviation of the incident angle.

•**internal_energy** (*PhysicalQuantity* of type temperature) – The internal kinetic energy of the molecule when added to the system. Drawn from a *MaxwellBoltzmannDistribution*. The translational kinetic energy component is replaced by the value set for the mean_kinetic_energy parameter.

•**enforce_temperature** (*bool*) – Rescale the velocities of the system to ensure the instantaneous temperature is exactly the requested temperature in the *MaxwellBoltzmannDistribution*. Applies to both kinetic- and internal energy. **Default:** False.

•**relaxation_time** (*PhysicalQuantity* of type time) – The MD simulation time of an optional subsequent MD relaxation, which is run after the initial process. This additional relaxation can be run with a slightly larger time step to extend the total simulation time.

•**relaxation_time_step** (*PhysicalQuantity* of type time) – The time step of an optional subsequent MD simulation, which is run after the initial process.

•**fbmc_time** (*PhysicalQuantity* of type time) – The MC simulation time of an optional subsequent FBMC relaxation, which is run after the initial process. This additional relaxation can be run with a slightly larger time step to extend the total simulation time. If None, no FBMC simulation will be run.

•**fbmc_displacement** (*PhysicalQuantity* of type length) – The maximum displacement of an optional subsequent FBMC simulation, which is run after the initial process.

•**optimize_geometry** (*bool*) – Flag that controls if the geometry should be optimized after the FBMC run.

•**label** (*str*) – A short label describing this trajectory.

- **md_hooks** (*HookFunctions* | None) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the MD forces evaluation. It can also contain measurement hooks, which are called at the end of each MD step after all constraints have been applied.
- **post_md_hooks** (*list of type SPSHook | None*) – Hook functions to call after each event.
- **log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.
- **trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.
- **stability_checks** (*bool*) – Add check during molecular dynamics to make sure the simulation is stable and producing meaningful results. If an unstable event is encountered, the simulation will be repeated with slightly different initial conditions.
- **max_attempts** (*int*) – Deposition events can fail when the molecular dynamics becomes unstable, or in the case of 2D materials the molecule is deposited onto the fixed region. If these cases are detected then the event is re-run. This parameter specifies the maximum number of attempts at running an event before the simulation is terminated. For general MD stability checks to be performed the argument `stability_checks` must be set to `True`

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

directionAngles()

Returns the direction angle of the atoms or molecules for each event. It returns `numpy.nan` for each event that does not add atoms or molecules. It returns `None` if this is the case for all events in the simulation.

Returns
The direction angles.

Return type
PhysicalQuantity of type angle | NoneType

filename()

Returns
The filename where the study object is stored.

Return type
str

forceBiasMonteCarloEquilibration(*mc_time*, *max_atom_displacement=None*, *optimize_geometry=False*, *log_interval=None*, *trajectory_interval=None*)

Run a time-stamped force bias Monte Carlo equilibration.

Parameters

- **mc_time** (*PhysicalQuantity* of type time) – The total elapsed Monte Carlo time to simulate.
- **max_atom_displacement** (*PhysicalQuantity* of type length) – The maximum distance an atom can move in each Cartesian direction during a single step. **Default:** 0.1*Angstrom.
- **optimize_geometry** (*bool*) – Flag controls if the geometry should be optimized after the FBMC run. **Default:** False.

- **log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.
- **trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.

incidentAngles()

Returns the incident angle of the atoms or molecules for each event. It returns numpy.nan for each event that does not add atoms or molecules. It returns None if this is the case for all events in the simulation.

Returns
The incident angles.

Return type
PhysicalQuantity of type angle | NoneType

kineticEnergies()

Returns the kinetic energy of the atoms or molecules for each event. It returns numpy.nan for each event that does not add atoms or molecules. It returns None if this is the case for all events in the simulation.

Returns
The kinetic energies.

Return type
PhysicalQuantity of type energy | NoneType

lastImage()

Get the last image from the SPS task.

Returns
The last configuration.

Return type
AtomicConfiguration

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

molecularDynamicsEquilibration(*md_time*, *time_step*, *post_md_hooks=None*, *log_interval=None*, *trajectory_interval=None*)

Run an MD simulation but without adding an atom or molecule to the system.

Parameters

- **md_time** (*PhysicalQuantity* of type time) – The total MD simulation time for this event.
- **time_step** (*PhysicalQuantity* of type time) – The time step of the MD simulation.
- **post_md_hooks** (*list of type SPSHook | None*) – Hook functions to call after the molecular dynamics run.
- **log_interval** (int | *PhysicalQuantity* of type time) – The frequency with which log messages are written. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.

•**trajectory_interval** (*int*) – The frequency with which configurations are saved in the trajectory. The value can be given either as a number of MD steps (integer) or as a time interval. If not given, the default interval specified in SurfaceProcessSimulation is used.

nlinfo()

Returns
Structured information about the SurfaceProcessSimulation.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometry(*max_forces=PhysicalQuantity(0.05, eV / Ang)*, *max_steps=200*, *max_step_length=PhysicalQuantity(0.2, Ang)*, *optimizer_method=None*)

Run geometry optimization.

Parameters

•**max_forces** (*PhysicalQuantity* of type force) – The convergence criterion for the atomic forces. **Default:** 0.05*eV/Angstrom.

•**max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200.

•**max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.

•**optimizer_method** (FIRE | LBFGS) – The optimizer to use for optimizing the structure. **Default:** LBFGS.

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

sputteringYield(*exclude_elements=None*, *start_trajectory_index=None*, *end_trajectory_index=None*, *substrate_search_radius=PhysicalQuantity(2.0, Ang)*)

Compute the average sputtering yield, that is the average number of substrate particles removed in one event. To estimate the number of events, this function only considers the ones with a label that contains deposition.

Parameters

- **exclude_elements** (*List of type PeriodicTableElement*) – List of elements that are to be excluded from substrate atoms when computing the yield, for example the ones used for sputtering. **Default:** the difference between the set of elements in the final and the initial configuration.
- **start_trajectory_index** (*int*) – Index of the first trajectory in the simulation, usually identifies the clean surface. Indexing based on Movie Tool. **Default:** the first trajectory with a deposition label available.
- **end_trajectory_index** (*int*) – Index of the last deposition trajectory in the simulation. Indexing based on Movie Tool. **Default:** the last trajectory with a deposition label available.
- **substrate_search_radius** (PhysicalQuantity of type distance | None) – Search radius for finding substrate atoms that have become unbonded from the substrate structure. Unbound atoms that have substrate atoms above them within a cylinder of the given radius are considered to be part of the substrate structure and therefore not removed. Setting this argument to None disables this additional check. **Default:** 2 Angstrom.

Returns
The average sputtering yield.

Return type
float

stickingCoefficient(*exclude_elements=None*, *start_trajectory_index=None*, *end_trajectory_index=None*, *substrate_search_radius=PhysicalQuantity(2.0, Ang)*)

Compute average sticking coefficient, that is the average number of successful deposition events. To estimate the number of events, this function only considers the ones with a label that contains deposition.

Parameters

- **exclude_elements** (*List of type PeriodicTableElement*) – List of elements that are to be excluded when computing the yield, for example substrate atoms or precursors that will be removed. **Default:** if there is any difference between the elements in the initial and final configuration, will be the elements in the initial configuration, otherwise will be an empty list.
- **start_trajectory_index** (*int*) – Index of the first trajectory in the simulation, usually identifies the clean surface. **Default:** the first trajectory with a deposition label available.
- **end_trajectory_index** (*int*) – Index of the last deposition trajectory in the simulation. **Default:** the last trajectory with a deposition label available.
- **substrate_search_radius** (PhysicalQuantity of type distance | None) – Search radius for finding substrate atoms that have become unbound from the substrate structure. Unbound atoms that have substrate atoms above them within a cylinder of the given radius are considered to be part of the substrate structure and therefore not removed. Setting this argument to None disables this additional check. **Default:** 2 Angstrom.

Returns
The average sticking coefficient.

Return type
float

classmethod thicknessesFromSubstrate(*substrate*)

Calculate the length of the bottom fixed layer of the slab from the substrate Calculate the length of the thermostat layer of the slab (above the fixed layer) from the substrate

Parameters
substrate (*BulkConfiguration*) – The initial surface (bulk slab model).

Returns
The length of the bottom fixed layer of the slab. The length of the thermostat layer of the slab (above the fixed layer). The largest Z value The smallest Z value

Return type
tuple

trajectories()

Returns
The list of all MD trajectories in the order they were run.

Return type
list of type MDTrajectory

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations.

Usage Examples

Run an etching simulation of nitrogen on a silica surface using a ReaxFF potential.

```
# Attach ReaxFF calculator.
potentialSet = ReaxFF_CHONSSi_2012(strict_bondpairs = None)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
bulk_configuration.setCalculator(calculator)

surface_process_simulation = SurfaceProcessSimulation(
    substrate=bulk_configuration,
    filename='sio2_n_etching.hdf5',
    object_id='sps_0',
    random_seed=7,
    temperature=300.0*Kelvin,
    fixed_thickness=4.0*Angstrom,
    thermostat_thickness=8*Angstrom,
    log_interval=10,
)

# Run 5 deposition events, each 20 ps long.
surface_process_simulation.addSequence(
    molecule=Nitrogen,
    number_of_events=5,
```

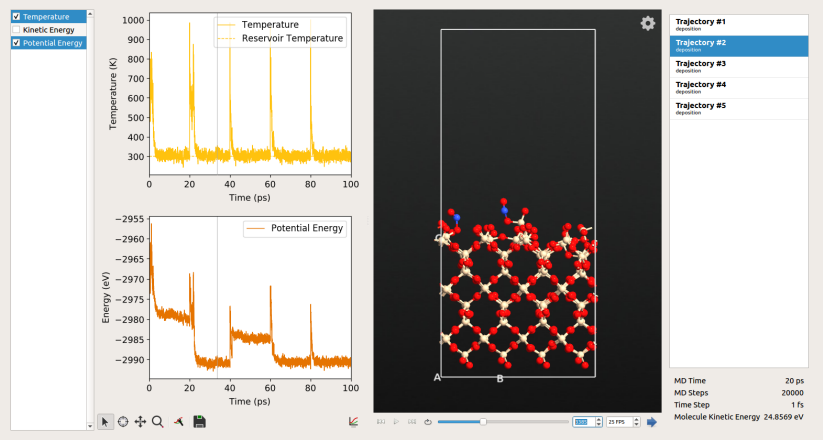
(continues on next page)

(continued from previous page)

```
    md_time=20*ps,
    time_step=1.0*fs,
    mean_kinetic_energy=25*eV,
    std_kinetic_energy=0.3*eV,
    mean_incident_angle=0*Degrees,
    std_incident_angle=2*Degrees,
)

surface_process_simulation.update()

print('Sticking coefficient:', surface_process_simulation.stickingCoefficient())
```

sio2_n_etching.py

The results from the simulation can be visualized using the Movie Tool on the QATK LabFloor.



2D Materials

Run an impact MD simulation of Argon on a 2D MoS2 surface.

```
surface_process_simulation = SurfaceProcessSimulation(
    substrate=molybdenite_0001,
    filename='MoS2_Ar_SPS_fixed_edges6.hdf5',
    object_id='sps',
    random_seed=randomSeed(1558689474, argon),
    temperature=300.0 * Kelvin,
    thermostat_method=Langevin,
    fixed_thickness=4.0 * Angstrom,
    thermostat_thickness=6.0 * Angstrom,
    log_interval=500,
    trajectory_interval=500,
    # Set the cartesian directions in which the fixed and thermostat regions
    # should be applied. If 0 or 1 (x- or y- direction) is chosen the regions are␣
↪applied both at
    # upper and lower end.
    fixed_directions=(0, 1),
```

(continues on next page)

(continued from previous page)

```
)

surface_process_simulation.molecularDynamicsEquilibration(
    md_time=5.0*ps,
    time_step=1*fs,
    log_interval=500,
    trajectory_interval=500
)

surface_process_simulation.addSequence(
    molecule=argon,
    number_of_events=5,
    md_time=2.0 * ps,
    time_step=0.5 * fs,
    mean_kinetic_energy=10.0 * eV,
    mean_incident_angle=30.0 * Degrees,
    std_incident_angle=5.0 * Degrees,
    relaxation_time=5.0 * ps,
    relaxation_time_step=1.0 * fs,
    post_md_hooks=sps_hooks,
    log_interval=500,
    trajectory_interval=500,
    stability_checks=False,
    # Max-attempts is also used for the attempts to place the molecule
    # so that it impacts only in the central free area.
    max_attempts=100,
)

surface_process_simulation.update()
```

`MoS2_Ar_etching.py`

The results from the simulation can be visualized using the Movie Tool on the QATK LabFloor as demonstrated above.

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Initial setup

Setting up the simulation requires an initial substrate, which should be a slab model with a surface normal in the positive z direction (such as those created by the cleave tool available in the **Builder**). Each simulation requires a `filename` and a unique `object_id` to save the results to, as well as a `random_seed` to control the random numbers used in the simulation. The substrate is divided into three regions, a fixed bottom layer, with thickness controlled by `fixed_thickness`, a layer of thermostat controlled atoms, with thickness controlled by `thermostat_thickness`, and the remaining atoms which are unconstrained and only coupled to the thermostat indirectly through interactions with the thermostated atoms.

Setting up simulation for a surface of a 2D material requires fixing the edges along surface vector instead of the bottom layer of the surface. Similar setting is required for choosing the `thermostat_thickness`. In this case the `fixed_thickness` and `thermostat_thickness` are chosen along the lateral directions instead of normal to the surface. This is controlled by `fixed_directions`.

Finally, the `distance_above_surface` parameter controls the height above the surface that incoming molecules are placed at.

The logging and trajectory intervals can also be controlled through the `log_interval` and `trajectory_interval` options. Both arguments take an integer number of steps between either logging to the console or writing trajectory data to the HDF5 file.

Adding events to the process

Once the object has been created, it is then possible to setup the individual *MolecularDynamics* (MD) simulations. Using the `addAtom` or `addMolecule` methods, will create a new MD simulation that adds an atom (or molecule) to the system. The initial velocity can be controlled by the `kinetic_energy` and `incident_angle` options. The `kinetic_energy` determines the magnitude of the velocity (all atoms in the molecule will be given the same velocity vector). While `incident_angle` controls the direction. The `incident_angle` is defined relative to the surface normal, so a value of 0 degrees is straight into the surface and an angle of 90 degrees is parallel to the surface.

Often, the time step must be quite small to accurately model the dynamics of high energy particles, but can be increased after a short time to equilibrate the system. This can be achieved by calling `molecularDynamicsEquilibration` after `addAtom` (or `addMolecule`). This will add a MD simulation that does not add any atoms, but allows for the selection of a different time step.

In addition to MD, it is also possible to run *ForceBiasMonteCarlo* (fbMC) simulations using the `forceBiasMonteCarloEquilibration` method.

Computing the yield of a process

It is possible to compute the average yield of a sputtering, etching or deposition process with the `sputteringYield()` and `stickingCoefficient()` functions.

In order to compute the average yield, the number of events is needed, this is estimated from the number of events with a label that contains `deposition`. The default labels for `addSequence()`, `addAtom()` and `addMolecule()` meet this requirement.

The optional parameters `start_trajectory_index` and `end_trajectory_index` expect the same indexing convention used in the **Movie Tool**.

Usually, some atoms in the system are not supposed to be considered for the yield estimate, this can be set through the `exclude_elements`, sensible default values should cover the most common use cases.

> **Attention:** In order to estimate the yield of the process, it is essential to have saved trajectories that correctly represent the initial and the final configuration. Make sure that (at least) the start and the end trajectories have more than one saved step, this can be tuned for each event with the `trajectory_interval` parameter.

SurfaceProcessSimulation Hooks

Hooks functions can be used to apply some changes on the last step of an event. After a sputtering event, it may be useful to clean the vacuum region from sputtered atoms with *CleanVacuumRegion*, or to remove the atoms of the element used for the process with *RemoveElementFromSubstrate*. In some cases the thermalization of the substrate could require several steps, it may then be convenient to decrease the kinetic energy of all the particles in one step with *ThermalizeSubstrate*.

Stabilizing the simulation

In some cases a *SurfaceProcessSimulation* can become unstable, leading to inaccurate results. This can especially occur when high energy impacts are being modeled. High initial molecular kinetic energy can lead to high energy configurations where the energy and forces are poorly estimated by an empirical potential. To help counteract this, extra stability checks can be added to the simulation. This is done by setting the keyword `stability_checks` to `True` when adding a sequence, molecule or atom to the *SurfaceProcessSimulation*. This keyword does two things to help improve the overall stability of the simulation. First, it adds a hook to the molecular dynamics that periodically checks the simulation temperature and maximum velocity. If these are deemed to be outside an acceptable range the simulation is terminated. This handles cases where the simulation would freeze because all of the energy becomes concentrated in a single interaction. Secondly, if the simulation terminates for any reason, it is re-run using a different randomized starting position of the molecule. Simulations are repeated up to 5 times to try and find a stable trajectory. If no stable trajectory is found the simulation is then ultimately terminated.

Care should be taken, when using this option, that re-running of the trajectory does not bias any statistics generated from multiple impacts.

SusceptibilityDerivatives

class SusceptibilityDerivatives(*configuration*, *filename*, *object_id*, *atomic_displacement=None*, *finite_difference_method=None*, *kpoints=None*, *bands_below_fermi_level=None*, *bands_above_fermi_level=None*, *broadening=None*, *processes_per_displacement=1*, *log_filename_prefix=None*, *use_symmetry=None*)

Constructor for the SusceptibilityDerivatives object.

Parameters

- **configuration** (*BulkConfiguration*) – The bulk configuration with an attached calculator for which to calculate the optical spectrum.
- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **atomic_displacement** (*PhysicalQuantity* of type length) – The distance the atoms are displaced in the finite difference method. **Default:** `0.01 * Angstrom`
- **finite_difference_method** (`Forward` | `Central`) – The finite difference scheme to use. **Default:** `Central`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which to calculate the transitions. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **bands_below_fermi_level** (*int*) – The maximum number of valence band states per principal spin channel to include at each k-point. **Default:** `25`
- **bands_above_fermi_level** (*int*) – The maximum number of conduction band states per principal spin channel to include at each k-point. **Default:** `25`

- **broadening** (*PhysicalQuantity* of type energy) – The broadening parameter used for the optical spectrum. **Default:** `0.1 * eV`
- **processes_per_displacement** (*int*) – The number of processes assigned to calculating a single displacement. **Default:** 1 process per displacement.
- **log_filename_prefix** (*str or None*) – Prefix for the filenames where the logging output for every displacement calculation is stored. The filenames are formed by appending a number and the file extension (.log). If a value of None is given then all logging output is done to stdout. **Default:** `"susceptibility_displacement_"`

atomicDisplacement()

Returns
The distance the atoms are displaced in the finite difference method.

Return type
PhysicalQuantity with length unit

bandsAboveFermiLevel()

Returns
The maximum number of conduction band states per principal spin channel to include at each k-point.

Return type
Int.

bandsBelowFermiLevel()

Returns
The maximum number of valence band states per principal spin channel to include at each k-point.

Return type
Int.

broadening()

Returns
The broadening parameter used for the optical spectrum.

Return type
PhysicalQuantity of type energy

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

filename()

Returns
The filename where the study object is stored.

Return type
str

finiteDifferenceMethod()

Returns
The finite difference scheme to use.

Return type
Central | Forward

kpoints()

Returns
The k-points for which optical spectrum is calculated.

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

processesPerDisplacement()

Returns
The number of processes per displacement.

Return type
int

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

susceptibilityDerivatives(*spin=None*)

Return the derivative of the susceptibility.

symmetry()

Returns
True if the use of crystal symmetry to reduce the number of displacements is enabled.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the SusceptibilityDerivatives study object.

Usage Examples

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Calculate the *SusceptibilityDerivatives* for a system with a $25 \times 25 \times 25$ kpoint-sampling used for the optical spectrum.

```
susceptibility_derivatives = SusceptibilityDerivatives(
    bulk_configuration,
    filename=filename,
    object_id='sd',
    kpoints=MonkhorstPackGrid(25, 25, 25),
    log_filename_prefix=None,
    use_symmetry=False)
susceptibility_derivatives.update()
```

Notes

The *SusceptibilityDerivatives* is calculated using the finite difference method in the basic unit cell.

In the following, we denote the Cartesian coordinates by α, β and the index of a degree of freedom by l. The high frequency dielectric constant is related to the susceptibility as $\epsilon^{\infty}_{\alpha\beta} = \delta_{\alpha,\beta} + 4\pi\chi_{\alpha\beta}(\omega = 0)$. The derivative of the dielectric susceptibility tensor, $\chi(\omega = 0)$, is therefore given by

$$\frac{\partial \chi_{\alpha\beta}}{\partial r_l}(\omega = 0) \approx \frac{\epsilon^{\infty}_{\alpha\beta}(\Delta r_l) - \epsilon^{\infty}_{\alpha\beta}(-\Delta r_l)}{2\Delta r_l},$$

and is a real tensor, i.e. a matrix for each degree of freedom. Degree of freedom l is displaced by Δr_l and $-\Delta r_l$, and the changes in the dielectric constant, $\Delta\epsilon^{\infty}_{\alpha\beta}$ are calculated to approximate the susceptibility derivative elements. The derivative is calculated by either `forward` or `central` finite differences, where we above focused on the latter.

SusceptibilityDerivatives calculations can be performed for DFT or Semi-Empirical calculators supporting the *OpticalSpectrum*. The object includes functionality to fully resume partially completed calculations by re-running the same script or reading the study object from file and calling `update()` on it. The study object will automatically detect which displacement calculations have already been carried out and only run the ones that are not yet completed.

When calculating the *SusceptibilityDerivatives* with ATK-DFT, accurate results may require a higher precision than usual by increasing the `density_mesh_cutoff` in *NumericalAccuracyParameters* and decreasing the `tolerance` in *IterationControlParameters*, e.g.

```
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=150.0*Hartree
    )
iteration_control_parameters = IterationControlParameters(
    tolerance=1e-6
    )
```

SuttonChenPotential

class SuttonChenPotential(*particleType1*, *particleType2*, *c*, *m*, *n*, *epsilon*, *sigma*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **c** (*float*) – Potential parameter.
- **m** (*float*) – Potential parameter.
- **n** (*float*) – Potential parameter.
- **epsilon** (*PhysicalQuantity of type energy*) – Potential parameter.
- **sigma** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

static fromMixingRule(*potential1*, *potential2*, *epsilon=None*, *sigma=None*, *c=None*, *m=None*, *n=None*)

Generate a Sutton-Chen potential using a mixing rule.

Parameters

- **potential1** (SuttonChenPotential) – Sutton-Chen potential with particleType1 == particleType2.
- **potential2** (SuttonChenPotential) – Sutton-Chen potential with particleType1 == particleType2.

:param epsilon Potential parameter. If this value is None, the corresponding
value in the new potential will be calculated using the mixing rule epsilon_new = sqrt(epsilon_1 * epsilon2). Otherwise the given value is used.

:param sigma Potential parameter. If this value is None, the corresponding
value in the new potential will be calculated using the mixing rule sigma_new = 0.5 * (sigma_1 + sigma_2). Otherwise the given value is used.

:param sigma Potential parameter. If this value is None, the corresponding
value in the new potential will be calculated using the mixing rule sigma_new = 0.5 * (sigma_1 + sigma_2). Otherwise the given value is used.

Returns
Sutton-Chen potential with particleType1 = potential1.particleType1 and particleType2 = potential2.particleType1. All other values are created using a mixing rule.

Return type
SuttonChenPotential

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Sutton-Chen potential for an alloy of aluminum and nickel by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk Al-Ni alloy configuration
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleCubic(2.881*Angstrom)

# Define elements
elements = [Aluminium, Nickel]

# Define coordinates
fractional_coordinates = [[0.0, 0.0, 0.0],
                          [0.5, 0.5, 0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Create a new potential set
potentialSet = TremoloXPotentialSet(name='SuttonChen_NiAl_2008')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Ni',
                                          mass=58.6934*atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='Al',
                                          mass=26.9815*atomic_mass_unit))

# Add the Sutton-Chen potentials to the potential set
potentialSet.addPotential(SuttonChenPotential('Ni', 'Ni',
                                              r_cut=7.04*Angstrom,
                                              c=39.432000,
                                              m=6.0,
                                              n=9.0,
                                              sigma=3.52*Angstrom,
                                              epsilon=0.015707*eV))
potentialSet.addPotential(SuttonChenPotential('Al', 'Al',
                                              r_cut=8.1*Angstrom,
                                              c=16.399,
                                              m=6.0,
                                              n=7.0,
                                              sigma=4.05*Angstrom,
                                              epsilon=0.033147*eV))

# Add the Sutton-Chen potential between different types to the potential set
potentialSet.addPotential(SuttonChenPotential('Ni', 'Al',
                                              r_cut=7.57*Angstrom,
                                              c=39.432000,
                                              m=6.000000,
                                              n=8.000000,
                                              sigma=3.785*Angstrom,
                                              epsilon=0.0228175355593*eV))
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

Notes

The *SuttonChenPotential* is a many-body potential primarily designed for metals and alloys[1].

The potential energy is defined as

$$V = \sum_i \left[\sum_{j>i} \left(\frac{\epsilon_{ij}\sigma_{ij}}{r_{ij}} \right)^{n_{ij}} - \epsilon_i c_i \sqrt{\rho_i} \right] ,$$

[1] A.ăP. Sutton and J.ăChen. Long-range Finnis–Sinclair potentials. *Philosophical Magazine Letters*, 61(3):139–146, 1990. doi:10.1080/09500839008206493.

with

$$\rho_i = \sum_{j \neq i} \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{m_{ij}} .$$

If the Sutton-Chen potential is supposed to act between different *ParticleType*, the parameters `epsilon`, `sigma`, `m`, and `n` should be chosen according to the native Sutton-Chen combination rules

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} ,$$

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} ,$$

$$m_{ij} = \frac{m_i + m_j}{2} ,$$

and

$$n_{ij} = \frac{n_i + n_j}{2} ,$$

as done in the example script above.

The `c` parameter does not have an effect in this case.

Table

class Table(*path=None*, *object_id=None*)

Table with data ordered into multiple columns.

A table contains data ordered in columns. Several different types of columns are supported, e.g. strings, integers, floats, PhysicalQuantity and a generic column of instances.

The table has two advantages over e.g. numpy arrays. Firstly, a Table can be immediately persisted to disk (if constructed with a path). Additionally a Table has the ability to also contain instances of any savable QuantumATK type, i.e. `MoleculeConfiguration`, `Bandstructure`, etc.

A table is constructed in the following way:

```
table = Table(path='my_filename.hdf5', object_id='MyTable')
```

Then columns are added. Here we add an index, a bandgap and a band structure:

```
table.addIntegerColumn(key='index')
table.addQuantityColumn(key='band_gap', unit=eV)
table.addInstanceColumn(key='bandstructure', types=Bandstructure)
```

Data can be added row by row:

```
table.append(0, 1.23 * meV, bandstructure=my_bandstructure)
```

Or the table can be extended with a list:

```
table.extend([
    [1, 2.34 * meV, bandstructure_1],
    [2, 3.45 * meV, bandstructure_2],
    [3, 4.56 * meV, bandstructure_3]
])
```

The table also supports normal array operations:

- `table[row-index, column-index]`, e.g. `table[1, 2]` or `table[2, 'bandstructure']`
- `table[row, column] = value`, e.g. `table[2, 1] = 12.34 * meV`
- `len(table)` returns number of rows.

Do note that the table (like numpy arrays) must be filled with values, before the cell can be addressed. This can be done using `table.fill(number_of_rows)`.

Parameters

- **path** (*str*) – The optional path. No path means that table is in-memory. **Default:** None
- **object_id** (str | None) – The optional object id. **Default:** None

add(*data*)

Add data as new columns.

Parameters

data (*Column, Table, list*) – The data.

addColumn(*column_type*, *key=None*, *title=None*, ***kwargs*)

Add a column to this table.

However, prefer to use the specific methods like `addIntegerColumn`, `addFloatColumn`, `addQuantityColumn`, `addStringColumn` or `addInstanceColumn`.

Parameters

- **column_type** (*Column type*) – The column type.
- **key** (*str*) – The key.
- **title** (*str*) – The title.
- ****kwargs** (*anything*) – The kwargs.

addFloatColumn(*key=None*, *title=None*)

Add a column of floats with specified key and title.

Parameters

- **key** (*str*) – The optional column key.
- **title** (*str*) – The optional title for this column.

addInstanceColumn(*key=None*, *title=None*, *types=None*)

Add a column of instances with specified types.

Parameters

- **key** (*str*) – The optional column key.
- **title** (*str*) – The optional title for this column.
- **types** (*type or tuple of type*) – The types of the instances in the column.

addIntegerColumn(*key=None*, *title=None*)

Add a column of integers with specified key and title.

Parameters

- **key** (*str*) – The optional column key.
- **title** (*str*) – The optional title for this column.

addQuantityColumn(*key=None*, *title=None*, *unit=None*)

Add a column of quantities with specified unit.

Parameters

- **key** (*str*) – The optional column key.
- **title** (*str*) – The optional title for this column.
- **unit** (PhysicalQuantity.Unit) – The unit of the column.

addStringColumn(*key=None*, *title=None*)

Add a column of strings with specified key and title.

Parameters

- **key** (*str*) – The optional column key.
- **title** (*str*) – The optional title for this column.

append(**values*, ***named_values*)

Append row of data to the table.

For a table with an integer column and a string column, one could do like this:

table.append(12, my_string)

Parameters

- ***values** (*anything*) – The row values.
- ****named_values** (*anything*) – the named row values.

clone(*path=None*, *object_id=None*)

Clone the table (i.e. create a new table with the same columns as this table, but no data).

Parameters

- **path** (*str*) – The filename to save the new table to.
- **object_id** (*str*) – The object id.

Returns
The cloned table.

Return type
Table

column(*index*)

Retrieve column at a given index.

Parameters
index (*int*) – The index.

Returns
The corresponding column.

Return type
Column

columns()

Returns
Retrieve all columns in the table.

Return type
List of columns

`consolidate()`

Try to embed all links into table.

`extend(data, link=False)`

Append rows to table with data from other table or a list of lists.

Parameters

- **data** (*Table | list | list of list*) – The data to extend the table with.
- **link** (*bool*) – True, if links in the data should be preserved. **Default:** False.

`extract(types=<class 'object'>)`

Extract all objects of a specific type within the table.

Parameters
types (*tuple of type*) – The types.

Returns
An iterator over all matches.

Return type
ColumnsIterator

`fill(length)`

Fill the table with default values up to a total number of rows.

Parameters
length (*int*) – The length to fill to.

`findColumn(key=None, types=None)`

Find column by force-matching types and selecting the closest key.

Parameters

- **key** (*str*) – The optional key (column name) to search for.
- **types** (*str*) – The optional types to filter for.

Returns
The best matched column.

Return type
Column | None

`findColumns(types)`

Find columns that may contain any objects of the supplied types.

Parameters
types (*tuple of types*) – The types to contain.

Returns
All matching columns.

Return type
list of Columns

classmethod fromList(*data*, *path=None*, *object_id=None*)

Create table from list.

Column types and names are automatically guessed from the contents of a list. A single list is turned into a single column table.

Passing in a list of list is interpreted as a list of rows.

Parameters

- **data** (*list or list of lists*) – The list.
- **path** (*str*) – The path to store the Table to. None means an in-memory table.
- **object_id** (*str*) – The optional object id. **Default:** None

Returns
The constructed table.

Return type
Table

metatext()

Returns
The metatext of the object or None if no metatext is set.

Return type
str | None

numberOfColumns()

Return the number of columns.

Returns
The number of columns.

Return type
str

numberOfRows()

Returns
The number of rows.

Return type
int

persist(*path*, *object_id=None*, *hidden=False*)

Persist the table to disk. If you later modify this table, all modifications are also persisted to disk.

This is a no-op if the table has already been saved to disk, even though the supplied filename or group_name may differ from the tables actual filename and group_name.

If you wish to save a copy of the table, simply save the table.

Parameters

- **path** (*str*) – The path.
- **group_name** (str | None | None) – The object id. **Default:** None

raw(**index*)

Retrieve the raw data. If using links, this will return LinkedObjects instead of the corresponding instances.

Parameters
index (*int | slice | tuple*) – The index.

Returns
The corresponding values.

Return type
Any

setMetatext(*text*)

Set a given metatext string on the object.

Parameters
text (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

signature()

Return a tuple describing the column signatures in the table. This can be used to match if two tables contain comparable data.

Returns
The signature of the table

Return type
tuple

TabulatedBondPotential

class TabulatedBondPotential(*particleType1*, *particleType2*, *fileName*, *splineType=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **fileName** (*str*) – The name of the file that holds the spline data. It must be a text file that stores the spline in the following way for not hermitian:

```
numPoints distance
<energy at 0 * distance>
<energy at 1 * distance>
...
<energy at (numPoints-1) * distance>
```

and for hermitian:

```
numPoints distance dfactor
<energy at 0 * distance>  <denergy/dr at 0 * distance>
<energy at 1 * distance>  <denergy/dr at 1 * distance>
...
<energy at (numPoints-1) * distance>  <denergy/dr at (numPoints-1) * distance>
```

where dfactor is used as a prefactor for the denergy/dr values

- **splineType** – The spline interpolation that will be used. Must be one of the following variables: TabulatedBondPotential.akima, TabulatedBondPotential.bessel, TabulatedBondPotential.fivepoint, TabulatedBondPotential.hermitian, TabulatedBondPotential.natural.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

  This potential is a tabulated potential for bond stretch interactions, that is read from a file. The tabulated file must be present in your project directory, otherwise the correct path has to be given.

  The first line of the file must contain two entries:

- The number of data points N of the tabulation.
- The distance Δr between two data points.

  The following N lines must contain the potential energy values $E_i, (i = 1...N)$, representing the energy at a particle distance $r = i\Delta r$.

  Possible spline types are

- hermitian
- bessel
- akima
- natural
- fivepoint

  If hermitian splines are used, a second column is required in the potential file, which contains the derivative values.

TaperedAnglePotential

class TaperedAnglePotential(*particleType1*, *particleType2*, *particleType3*, *k*, *theta0*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **k** (*PhysicalQuantity of type energy / angle**2*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius of this potential.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

TemperatureProfile

class TemperatureProfile(*md_trajectory*, *start_time=None*, *end_time=None*, *resolution=None*, *atom_selection=None*, *direction_index=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the spatial temperature profile of an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the temperature profile for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **resolution** (*PhysicalQuantity* of type length) – The bin size, which determines the resolution of the profile. **Default:** `2.0 * Angstrom`

- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Has to be an element of [0, 1, 2]. **Default:** 2
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the temperature profile.

zValues()

Return the positions of the bins associated with the temperature profile.

Usage Examples

Load an *MDTrajectory*, calculate the *TemperatureProfile* along the z-axis, and plot the results:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

t_profile = TemperatureProfile(md_trajectory)

# Get the bin_centers and the temperature values.
z            = t_profile.zValues().inUnitsOf(Angstrom)
temperatures = t_profile.data().inUnitsOf(Kelvin)

# Plot the data using pylab.
import pylab

pylab.plot(z, temperatures, label='Temperature profile')
pylab.xlabel('z (Ang)')
pylab.ylabel('T(z) (K)')
pylab.legend()

pylab.show()
```

temperature_profile.py

Notes

The *TemperatureProfile* is always calculated along the cartesian z-axis. It is recommended to use an orthogonal cell, if you want to calculate the *TemperatureProfile*.

The temperature values of empty bins in the profile are set to zero.

By default, all elements are taken into account.

The `atom_selection` parameter accepts an element, a tag name, or a list of indices to select atoms for the velocity distribution. This can be useful, e.g. in the presence of constraints as constrained atoms should be excluded in this analysis.

TersoffBrennerBOPairPotential

class TersoffBrennerBOPairPotential(*particleType1*, *particleType2*, *delta*, *eta*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **delta** (*float*) – Potential parameter.
- **eta** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner pair and bond-order potentials between silicon and fluorine.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    A = 1830.8*eV,
    B = 471.18*eV,
    l = 2.4799*1/Angstrom,
    mu = 1.7322*1/Angstrom,
    Re = 2.35*Angstrom,
    R1 = 2.7*Angstrom,
    R2 = 3.0*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('F', []),
```

(continues on next page)

(continued from previous page)

```
    A = 16451.97*eV,
    B = 146.8149*eV,
    l = 6.8149*1/Angstrom,
    mu = 2.8568*1/Angstrom,
    Re = 1.4119*Angstrom,
    R1 = 1.7*Angstrom,
    R2 = 2.0*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('F', []),
    A = 37412.28*eV,
    B = 925.846*eV,
    l = 5.4875*1/Angstrom,
    mu = 2.7437*1/Angstrom,
    Re = 1.6008*Angstrom,
    R1 = 1.83922*Angstrom,
    R2 = 2.13922*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerBOPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    delta = 0.63505,
    eta = 0.78734,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerBOPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('F', []),
    delta = 0.80469,
    eta = 1.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerBOPairPotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('Si', []),
    delta = 0.5,
    eta = 1.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerBOPairPotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('F', []),
    delta = 0.5,
    eta = 1.0,
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerCorrectionPotential

class TersoffBrennerCorrectionPotential(*particleType1*, *particleType2*, *activeTypes*, *L*, *U*, *x*, *z*, *f*)

Constructor of the potential.

To construct this potential it is necessary to specify a threedimensional function. This is done by passing the function values of this function at some grid points. For all other points, tricubic interpolation is used. At the border of the grid, all derivatives are assumed to be zero.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **activeTypes** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the so-called conjugated part of the potential.
- **x** (*sequence of float*) – The x-coordinates of the grid. It must have uniform spacing. The same coordinates are also used for the y-coordinates.
- **z** (*sequence of float*) – The z-coordinates of the grid. It must have uniform spacing.
- **f** (*3D numpy.array*) – The function values at the given grid points. f should be a threedimensional array of size (len(x), len(y), len(z)). f[i, j, k] should be the function value at (x[i], y[j], z[k]) and f must be symmetric with respect to its to first components, i.e. f[i, j, k] = f[j, i, k].

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner correction potentials for carbon.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerCorrectionPotential(
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('C', []),
    activeTypes = [ParticleIdentifier('C', []), ],
    L = 2.0,
```

(continues on next page)

(continued from previous page)

```
    U = 3.0,
    x = numpy.array([0., 1., 2., 3.]),
    z = numpy.array([0., 1., 2., 3.]),
    f = numpy.array([[
        [ 0.     ,  0.     ,  0.     ,  0.     ],
        [ 0.     , -0.02882,  0.     ,  0.     ],
        [ 0.     ,  0.     ,  0.     ,  0.     ],
        [ 0.     ,  0.     ,  0.     ,  0.     ]],

       [[ 0.     , -0.02882,  0.     ,  0.     ],
        [ 0.     , -0.0288 ,  0.     ,  0.     ],
        [ 0.     , -0.09   , -0.0243 , -0.0243 ],
        [ 0.     ,  0.     ,  0.     ,  0.     ]],

       [[ 0.     ,  0.     ,  0.     ,  0.     ],
        [ 0.     , -0.09   , -0.0243 , -0.0243 ],
        [ 0.     ,  0.0415 ,  0.     ,  0.     ],
        [ 0.     , -0.0363 , -0.0363 , -0.0363 ]],

       [[ 0.     ,  0.     ,  0.     ,  0.     ],
        [ 0.     ,  0.     ,  0.     ,  0.     ],
        [ 0.     , -0.0363 , -0.0363 , -0.0363 ],
        [ 0.     ,  0.     ,  0.     ,  0.     ]]]),
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerCorrectionPotential2

class TersoffBrennerCorrectionPotential2(*particleType1*, *particleType2*, *activeTypes*, *L*, *U*, *x*, *z*, *f*)

Constructor of the potential.

To construct this potential it is necessary to specify a threedimensional function. This is done by passing the function values of this function at some grid points. For all other points, tricubic interpolation is used. At the border of the grid, all derivatives are assumed to be zero.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **activeTypes** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the so-called conjugated part of the potential.
- **x** (*sequence of float*) – The x-coordinates of the grid. It must have uniform spacing. The same coordinates are also used for the y-coordinates.
- **z** (*sequence of float*) – The z-coordinates of the grid. It must have uniform spacing.
- **f** (*3D numpy.array*) – The function values at the given grid points. f should be a threedimensional array of size (len(x), len(y), len(z)). f[i, j, k] should be the function value at (x[i], y[j], z[k]) and f must be symmetric with respect to its to first components, i.e. f[i, j, k] = f[j, i, k].

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner pair potentials for carbon.

```
potentialSet = TremoloXPotentialSet(name = 'Brenner_CH_2002')
potentialSet.addParticleType(ParticleType(
        symbol = 'C',
        mass = 12.0107*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 6,
        tags = [],
))
_potential = TersoffBrennerCorrectionPotential2(
        particleType1 = ParticleIdentifier('C', []),
        particleType2 = ParticleIdentifier('C', []),
        activeTypes = [ParticleIdentifier('C', []), ],
        L = 2.0,
        U = 3.0,
        x = numpy.array([0, 1, 2, 3]),
        z = numpy.array(
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
    ),
        f = numpy.array([
        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0216935, 0.00495861, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
```

(continues on next page)

(continued from previous page)

```
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.02469883, -0.00597133, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947]],

        [[0.0, 0.0216935, 0.00495861, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0525, -0.00208875, -0.0080428, -0.0080428,
         -0.0080428, -0.0080428, -0.0080428, -0.0080428, -0.0080428,
         -0.0080428, -0.0080428, -0.0080428, -0.0080428, -0.0080428,
         -0.0080428, -0.0080428, -0.0080428, -0.0080428, -0.0080428 ],
         [0.0, 0.00482478, 0.015, -0.01, -0.01168894,
         -0.01337788, -0.01506682, -0.01506682, -0.01506682, -0.01506682,
         -0.01506682, -0.01506682, -0.01506682, -0.01506682, -0.01506682,
         -0.01506682, -0.01506682, -0.01506682, -0.01506682, -0.01506682],
         [0.0, 0.0, -0.06241838, -0.06241838, -0.06241838,
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838,
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838,
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838]],

        [[0.0, 0.02469883, -0.00597133, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.00482478, 0.015, -0.01, -0.01168894,
         -0.01337788, -0.01506682, -0.01506682, -0.01506682, -0.01506682,
         -0.01506682, -0.01506682, -0.01506682, -0.01506682, -0.01506682,
         -0.01506682, -0.01506682, -0.01506682, -0.01506682, -0.01506682],
         [0.0, 0.04722478, 0.011, 0.01985293, 0.01654411,
          0.01323529, 0.00992647, 0.00661765, 0.00330882, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469]],

        [[0.0, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947,
         -0.05989947, -0.05989947, -0.05989947, -0.05989947, -0.05989947],
         [0.0, 0.0, -0.06241838, -0.06241838, -0.06241838,
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838,
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838,
```

(continues on next page)

(continued from previous page)

```
         -0.06241838, -0.06241838, -0.06241838, -0.06241838, -0.06241838],
         [0.0, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469,
         -0.02235469, -0.02235469, -0.02235469, -0.02235469, -0.02235469],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0]]]),
)
potentialSet.addPotential(_potential)
```

tersoffbrenner_carbon.py

Notes

The correction potential that is used in the second generation REBO[1] and the AIREBO[2] potential. (In most papers it is denoted by pi_rc). It corrects the potential energy by using a tricubic spline that depends on the coordination numbers of the involved particles.

Parameters are added in a symmetric fashion, meaning that the correction will be applied to particleType1-particleType2 bonds as well as to particleType2-particleType1 bonds (if the two types are different).

TersoffBrennerMoliere Potential

class TersoffBrennerMoliere Potential(*particleType1*, *particleType2*, *a*, *b*, *c*, *d*, *c1*, *c2*, *c3*, *c4*, *d1*, *d2*, *d3*, *d4*, *f*, *Zi*, *Zj*, *s*, *ra*, *rb*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **a** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **b** (*float*) – Potential parameter.
- **c** (*PhysicalQuantity of type energy*) – Potential parameter.
- **d** (*PhysicalQuantity of type energy*) – Potential parameter.
- **c1** (*float*) – Potential parameter.
- **c2** (*float*) – Potential parameter.
- **c3** (*float*) – Potential parameter.
- **c4** (*float*) – Potential parameter.
- **d1** (*float*) – Potential parameter.

[1] D.ăW. Brenner, O.ăShenderova, J.ăHarrison, S.ăJ. Stuart, B.ăNi, and S.ăB Sinnott. A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons. *J. Phys.: Condensed Matter*, 14(4):783–802, 2002. arXiv:S0953-8984(02)31186-X, doi:10.1088/0953-8984/14/4/312.

[2] StevenăJ. Stuart, AlanăB. Tutein, and JudithăA. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *J. Chem. Phys.*, 112(14):6472–6486, 2000. URL: https://doi.org/10.1063/1.481208.

•**d2** (*float*) – Potential parameter.

•**d3** (*float*) – Potential parameter.

•**d4** (*float*) – Potential parameter.

•**f** (*PhysicalQuantity of type length*) – Potential parameter.

•**Zi** (*PhysicalQuantity of type charge*) – Potential parameter.

•**Zj** (*PhysicalQuantity of type charge*) – Potential parameter.

•**s** (*PhysicalQuantity of type energy*) – Potential parameter.

•**ra** (*PhysicalQuantity of type length*) – Potential parameter.

•**rb** (*PhysicalQuantity of type length*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff-Brenner pair potential with additional moliere-type repulsive term for silicon.

```
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    A = 1830.8*eV,
    B = 471.18*eV,
    l = 2.4799*1/Angstrom,
    mu = 1.7322*1/Angstrom,
    Re = 2.35*Angstrom,
    R1 = 2.7*Angstrom,
    R2 = 3.0*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerMoliere Potential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
```

(continues on next page)

(continued from previous page)

```
    a = -7.155376*1/Angstrom,
    b = 9.502208,
    c = 237.361562*eV,
    d = 1.0*eV,
    f = 0.10164343824*Angstrom,
    c1 = 0.35,
    c2 = 0.55,
    c3 = 0.1,
    c4 = 0.0,
    d1 = 0.3,
    d2 = 1.2,
    d3 = 6.0,
    d4 = 0.0,
    Zi = 14.0*elementary_charge,
    Zj = 14.0*elementary_charge,
    s = 296.792932*eV,
    ra = 0.286968*Angstrom,
    rb = 0.65522*Angstrom,
)
potentialSet.addPotential(_potential)
```

Notes

In some cases it is necessary to boost the repulsive term

$$U_{ij}^{rep} = f_{ij}^{TB}(r_{ij}) A_{ij} \exp(-\lambda_{ij} r_{ij})$$

of a *TersoffBrennerPairPotential* for particles that are close to each other. Therefore, a correction term was introduced in[1] that replaces the original repulsion term by a Coulomb-like term for close interactions. This behavior can be achieved by an additional correction term U_{ij}^{repCor} that is added to the U_{ij}^{rep} terms. It has the form

$$U_{ij}^{repCor} = \begin{cases} -U_{ij}^{rep} + U_{ij}^{Moliere} & r_{ij} \leq r_{ij}^{(a)} \\ -U_{ij}^{rep} + U_{ij}^{spline} & r_{ij}^{(a)} < r_{ij} < r_{ij}^{(b)} \\ 0 & r_{ij}^{(b)} \leq r_{ij} \end{cases}$$

which basically means that the repulsive term U_{ij}^{rep} is replaced by

$$\begin{cases} U_{ij}^{Moliere} & r_{ij} \leq r_{ij}^{(a)} \\ U_{ij}^{spline} & r_{ij}^{(a)} < r_{ij} < r_{ij}^{(b)} \\ U_{ij}^{rep} & r_{ij}^{(b)} \leq r_{ij} \end{cases}$$

The two new terms $U_{ij}^{Moliere}$ and U_{ij}^{spline} are given by

$$U_{ij}^{Moliere} = s_{ij} + \frac{Z_1 Z_2}{4\pi\epsilon_0 r_{ij}} \sum_{m=1}^{4} c_{ij}^{(m)} \exp\left(-\frac{d_{ij}^{(m)} r_{ij}}{f_{ij}}\right)$$

and

$$U_{ij}^{spline} = c_{ij} + d_{ij} \exp(a_{ij} r_{ij} + b_{ij}).$$

[1] CameronăF. Abrams and DavidăB. Graves. Molecular dynamics simulations of si etching by energetic cf3+. *Journal of Applied Physics*, 86(11):5938–5948, December 1999. URL: https://doi.org/10.1063/1.371637, arXiv:https://pubs.aip.org/aip/jap/article-pdf/86/11/5938/18980143/5938_1_online.pdf, doi:10.1063/1.371637.

This correction can be set up by adding a *TersoffBrennerMoliere Potential*. The potential parameters are defined as follows:

a	The parameter a_{ij}
b	The parameter b_{ij}
c	The parameter c_{ij}
d	The parameter d_{ij}
f	The parameter f_{ij}
s	The parameter s_{ij}
c1,,c4	The parameters $c_{ij}^{(1)}, \ldots, c_{ij}^{(4)}$
d1,,d4	The parameters $d_{ij}^{(1)}, \ldots, d_{ij}^{(4)}$
zi	The parameter Z_1
zj	The parameter Z_2
r_a	The parameter $r_{ij}^{(a)}$
r_b	The parameter $r_{ij}^{(b)}$

TersoffBrennerPairPotential

class TersoffBrennerPairPotential(*particleType1*, *particleType2*, *A*, *B*, *l*, *mu*, *Re*, *R1*, *R2*, *fType=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type energy*) – Potential parameter.
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).
- **mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **Re** (*PhysicalQuantity of type length*) – Potential parameter.
- **R1** (*PhysicalQuantity of type length*) – Potential parameter.
- **R2** (*PhysicalQuantity of type length*) – Potential parameter.
- **fType** – The smoothing function that will be used. Must be one of the following variables:

TwoBodyPotential.Brenner TwoBodyPotential.MurtyAtwater TwoBodyPotential.HumbridGraves

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner pair potentials between silicon and fluorine.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    A = 1830.8*eV,
    B = 471.18*eV,
    l = 2.4799*1/Angstrom,
    mu = 1.7322*1/Angstrom,
    Re = 2.35*Angstrom,
    R1 = 2.7*Angstrom,
    R2 = 3.0*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('F', []),
    A = 16451.97*eV,
    B = 146.8149*eV,
    l = 6.8149*1/Angstrom,
    mu = 2.8568*1/Angstrom,
    Re = 1.4119*Angstrom,
    R1 = 1.7*Angstrom,
    R2 = 2.0*Angstrom,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerPairPotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('F', []),
    A = 37412.28*eV,
    B = 925.846*eV,
    l = 5.4875*1/Angstrom,
    mu = 2.7437*1/Angstrom,
    Re = 1.6008*Angstrom,
    R1 = 1.83922*Angstrom,
    R2 = 2.13922*Angstrom,
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerPairPotential2

class TersoffBrennerPairPotential2(*particleType1*, *particleType2*, *A*, *l*, *Q*, *B1*, *B2*, *B3*, *mu1*, *mu2*, *mu3*, *Re*, *R1*, *R2*, *fType=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).
- **Q** (*PhysicalQuantity of type length*) – Potential parameter.
- **B1** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B2** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B3** (*PhysicalQuantity of type energy*) – Potential parameter.
- **mu1** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **mu2** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **mu3** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **Re** (*PhysicalQuantity of type length*) – Potential parameter.
- **R1** (*PhysicalQuantity of type length*) – Potential parameter.
- **R2** (*PhysicalQuantity of type length*) – Potential parameter.
- **fType** – The smoothing function that will be used. Must be one of the following variables:

  TwoBodyPotential.Brenner TwoBodyPotential.MurtyAtwater TwoBodyPotential.HumbridGraves

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner pair potentials for carbon.

```
potentialSet = TremoloXPotentialSet(name = 'Brenner_CH_2002')
potentialSet.addParticleType(ParticleType(
        symbol = 'C',
        mass = 12.0107*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 6,
        tags = [],
))

_potential = TersoffBrennerPairPotential2(
        particleType1 = ParticleIdentifier('C', []),
        particleType2 = ParticleIdentifier('C', []),
        A = 10953.5441622*eV,
        Q = 0.313460296083*Angstrom,
        l = 4.74653906066*1/Angstrom,
        B1 = 12388.791978*eV,
        B2 = 17.567406465*eV,
        B3 = 30.7149320807*eV,
        mu1 = 4.7204523127*1/Angstrom,
        mu2 = 1.4332132499*1/Angstrom,
        mu3 = 1.3826912506*1/Angstrom,
        Re = 0.0*Angstrom,
        R1 = 1.7*Angstrom,
        R2 = 2.0*Angstrom,
)
```

tersoffbrenner_carbon.py

Notes

The pair potential that is used in the second generation REBO[1] and the AIREBO[2] potential. When added to a potential set, this class activates the repulsive and attractive terms between two types of particles. Interactions are added in a symmetric fashion, which means that adding a *TersoffBrennerPairPotential2* object with particle types A and B to a potential set will activate interactions between particles of type A and B but also between B and A.

[1] D.ăW. Brenner, O.ăShenderova, J.ăHarrison, S.ăJ. Stuart, B.ăNi, and S.ăB Sinnott. A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons. *J. Phys.: Condensed Matter*, 14(4):783–802, 2002. arXiv:S0953-8984(02)31186-X, doi:10.1088/0953-8984/14/4/312.

[2] StevenăJ. Stuart, AlanăB. Tutein, and JudithăA. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *J. Chem. Phys.*, 112(14):6472–6486, 2000. URL: https://doi.org/10.1063/1.481208.

TersoffBrennerPairPotential3

class TersoffBrennerPairPotential3(*particleType1*, *particleType2*, *A*, *B*, *l*, *mu*, *Re*, *R1*, *R2*, *xRep1*, *FRep1*, *dxFRep1*, *dxxFRep1*, *xRep2*, *FRep2*, *dxFRep2*, *dxxFRep2*, *xAttr1*, *FAttr1*, *dxFAttr1*, *dxxFAttr1*, *xAttr2*, *FAttr2*, *dxFAttr2*, *dxxFAttr2*, *fType=None*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**A** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B** (*PhysicalQuantity of type energy*) – Potential parameter.

•**l** (*PhysicalQuantity of type length**-1*) – Potential parameter (lambda).

•**mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**Re** (*PhysicalQuantity of type length*) – Potential parameter.

•**R1** (*PhysicalQuantity of type length*) – Potential parameter.

•**R2** (*PhysicalQuantity of type length*) – Potential parameter.

•**xRep1** (*sequence of floats*) – Coordinates for which the values of the function F^(rep, 1) are known. Must be sorted in ascending order.

•**FRep1** (*sequence of floats*) – The function values of F^(rep, 1) at the points in xRep1.

•**dxFRep1** (*sequence of floats*) – The first derivatives of F^(rep, 1) at the points in xRep1.

•**dxxFRep1** (*sequence of floats*) – The second derivatives of F^(rep, 1) at the points in xRep1.

•**xRep2** (*sequence of floats*) – Coordinates for which the values of the function F^(rep, 1) are known. Must be sorted in ascending order.

•**FRep2** (*sequence of floats*) – The function values of F^(rep, 2) at the points in xRep2.

•**dxFRep2** (*sequence of floats*) – The first derivatives of F^(rep, 2) at the points in xRep2.

•**dxxFRep2** (*sequence of floats*) – The second derivatives of F^(rep, 2) at the points in xRep2.

•**xAttr1** (*sequence of floats*) – Coordinates for which the values of the function F^(attr, 1) are known. Must be sorted in ascending order.

•**FAttr1** (*sequence of floats*) – The function values of F^(attr, 1) at the points in xAttr1.

•**dxFAttr1** (*sequence of floats*) – The first derivatives of F^(attr, 1) at the points in xAttr1.

•**dxxFAttr1** (*sequence of floats*) – The second derivatives of F^(attr, 1) at the points in xAttr1.

•**xAttr2** (*sequence of floats*) – Coordinates for which the values of the function F^(attr, 1) are known. Must be sorted in ascending order.

•**FAttr2** (*sequence of floats*) – The function values of F^(attr, 2) at the points in xAttr2.

•**dxFAttr2** (*sequence of floats*) – The first derivatives of F^(attr, 2) at the points in xAttr2.

•**dxxFAttr2** (*sequence of floats*) – The second derivatives of F^(attr, 2) at the points in xAttr2.

•**fType** – The smoothing function that will be used. Must be one of the following variables:

TwoBodyPotential.Brenner TwoBodyPotential.MurtyAtwater TwoBodyPotential.HumbridGraves

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner pair potentials for silane.

```
potentialSet = TremoloXPotentialSet(name = 'Murty_HSi_1995')
potentialSet.addParticleType(ParticleType(
        symbol = 'Si',
        mass = 28.0855*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 14,
        tags = [],
))
potentialSet.addParticleType(ParticleType(
        symbol = 'H',
        mass = 1.00794*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 1,
        tags = [],
))

_potential = TersoffBrennerPairPotential3(
        particleType1 = ParticleIdentifier('Si', []),
        particleType2 = ParticleIdentifier('H', []),
        A = 323.54*eV,
        B = 84.18*eV,
```

(continues on next page)

(continued from previous page)

```
        l = 2.9595*1/Angstrom,
        mu = 1.6158*1/Angstrom,
        Re = 1.475*Angstrom,
        R1 = 1.7*Angstrom,
        R2 = 2.0*Angstrom,
        xRep1 = numpy.array([1, 2, 3, 4, 5, 6, 7]),
        FRep1 = numpy.array([ 2.01 ,  2.218,  1.908,  2.   ,  2.   ,  2.   ,  2.   ]),
        dxFRep1 = numpy.array([ 0.   , -0.038, -0.154,  0.   ,  0.   ,  0.   ,  0.   ]),
        dxxFRep1 = numpy.array([ 1.324, -1.4  ,  1.168,  0.   ,  0.   ,  0.   ,  0.   ]),
        xRep2 = numpy.array([ 0.,  1.]),
        FRep2 = numpy.array([ 0.,  0.]),
        dxFRep2 = numpy.array([ 0.,  0.]),
        dxxFRep2 = numpy.array([ 0.,  0.]),
        xAttr1 = numpy.array([1, 2, 3, 4, 5, 6, 7]),
        FAttr1 = numpy.array([ 1.86 ,  2.07 ,  1.868,  2.   ,  2.   ,  2.   ,  2.   ]),
        dxFAttr1 = numpy.array([-0.    ,  0.0204, -0.0576,  0.    ,  0.    ,  0.    ,  0.
↪    ]),
        dxxFAttr1 = numpy.array([ 1.2192, -1.1784,  1.0224,  0.    ,  0.    ,  0.    , ␣
↪0.    ]),
        xAttr2 = numpy.array([ 0.,  1.]),
        FAttr2 = numpy.array([ 0.,  0.]),
        dxFAttr2 = numpy.array([ 0.,  0.]),
        dxxFAttr2 = numpy.array([ 0.,  0.]),
)
potentialSet.addPotential(_potential)
```

tersoff_silane.py

Notes

The pair potential that is used in the Murty-Atwater variant of the Tersoff- Brenner potential[1]. When added to a potential set, this class activates the repulsive and attractive terms between two types of particles.

Both the repulsive and the attractive interactions depend on quintic splines. To define these splines, their function values and derivatives at some coordinates must be passed to this potential.

Interactions are added in a symmetric fashion, which means that adding a TersoffBrennerPairPotential2 object with particle types A and B to a potential set will activate interactions between particles of type A and B but also between B and A.

TersoffBrennerSplinePotential

class TersoffBrennerSplinePotential(*particleType1*, *particleType2*, *activeTypes1*, *activeTypes2*, *x*, *y*, *f*)

Constructor of the potential.

To construct this potential it is necessary to specify a twodimensional function. This is done by passing the function values of this function at some grid points, for all other points, bicubic interpolation is used. At the border of the grid, all derivatives are assumed to be zero.

Parameters

[1] M.ăV. RamanaăMurty and HarryăA. Atwater. Empirical interatomic potential for Si-H interactions. *Phys. Rev. B*, 51:4889–4893, 1995. URL: https://link.aps.org/doi/10.1103/PhysRevB.51.4889.

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **activeTypes1** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the N1 coefficient in the potential.
- **activeTypes2** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the N2 coefficient in the potential.
- **x** (*sequence of float*) – The x-coordinates of the grid. It must have uniform spacing.
- **y** (*sequence of float*) – The y-coordinates of the grid. It must have uniform spacing.
- **f** (*2D numpy.array*) – The function values at the given grid points. f should be a twodimensional array of size (len(x), len(y)). f[i, j] should be the function value at (x[i], y[j]).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner spline potentials between carbon and fluorine pairs. Note that for fluorine atoms the coordination with both carbon and silicon atoms is considered.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerSplinePotential(
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('F', []),
    activeTypes1 = [ParticleIdentifier('F', []), ],
    activeTypes2 = [ParticleIdentifier('C', []), ParticleIdentifier('Si', []), ],
    x = numpy.array([ 0.,  1.,  2.,  3.]),
    y = numpy.array([ 0.,  1.,  2.]),
    f = numpy.array([
       [ 0.     ,  0.10643,  0.1892 ],
       [ 0.06353,  0.1892 , -0.0021 ],
       [ 0.21895,  0.0104 ,  0.     ],
       [ 0.01806,  0.     ,  0.     ]]),
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerTorsionCorrectionPotential

class TersoffBrennerTorsionCorrectionPotential(*particleType1*, *particleType2*, *activeTypes*, *L*, *U*, *x*, *z*, *f*)

Constructor of the potential.

To construct this potential it is necessary to specify a threedimensional function. This is done by passing the function values of this function at some grid points. For all other points, tricubic interpolation is used. At the border of the grid, all derivatives are assumed to be zero.

Parameters

- **particleType1** (*ParticleType or ParticleIdentifier*) – Identifier of the first particle type.
- **particleType2** (*ParticleType or ParticleIdentifier*) – Identifier of the second particle type.
- **activeTypes** (*sequence of ParticleType or ParticleIdentifier*) – List of particle types that are involved in the calculation of the so-called conjugated part of the potential.
- **x** (*sequence of float*) – The x-coordinates of the grid. It must have uniform spacing. The same coordinates are also used for the y-coordinates.
- **z** (*sequence of float*) – The z-coordinates of the grid. It must have uniform spacing.
- **f** (*3D numpy.array*) – The function values at the given grid points. f should be a threedimensional array of size (len(x), len(y), len(z)). f[i, j, k] should be the function value at (x[i], y[j], z[k]) and f must be symmetric with respect to its to first components, i.e. f[i, j, k] = f[j, i, k].

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff-Brenner torsion correction potential for carbon.

```
potentialSet = TremoloXPotentialSet(name = 'Brenner_CH_2002')
potentialSet.addParticleType(ParticleType(
        symbol = 'C',
        mass = 12.0107*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 6,
        tags = [],
))

        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0]],

        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0]]]),
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTorsionCorrectionPotential(
```

(continues on next page)

(continued from previous page)

```
        particleType1 = ParticleIdentifier('C', []),
        particleType2 = ParticleIdentifier('C', []),
        activeTypes = [ParticleIdentifier('C', []), ],
        L = 2.0,
        U = 3.0,
        x = numpy.array([0, 1, 2, 3]),
        z = numpy.array(
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
    ),
        f = numpy.array([
        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0]],

        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0]],

        [[0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
         [0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0,
          0.0, 0.0, 0.0, 0.0, 0.0],
```

(continues on next page)

(continued from previous page)

```
          [0.0, -0.03514004, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, -0.00404837,
          -0.00404837, -0.00404837, -0.00404837, -0.00404837, -0.00404837,
          -0.00404837, -0.00404837, -0.00404837, -0.00404837, -0.00404837],
          [0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0]],

         [[0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0],
          [0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0],
          [0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0],
          [0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0]]]),
)
potentialSet.addPotential(_potential)
```

tersoffbrenner_carbon.py

Notes

This potential adds the torsion correction potential that is used in the second generation REBO potential[1]. It corrects the potential energy by using a tricubic spline that depends on the coordination numbers of the involved particles.

Parameters are added in a symmetric fashion, meaning that the correction will be applied to particleType1-particleType2 bonds as well as to particleType2-particleType1 bonds (if the two types are different).

TersoffBrennerTriplePotential

class TersoffBrennerTriplePotential(*particleType1*, *particleType2*, *particleType3*, *alpha*, *beta*, *g_c*, *g_d*, *g_h*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

[1] D.ăW. Brenner, O.ăShenderova, J.ăHarrison, S.ăJ. Stuart, B.ăNi, and S.ăB Sinnott. A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons. *J. Phys.: Condensed Matter*, 14(4):783–802, 2002. arXiv:S0953-8984(02)31186-X, doi:10.1088/0953-8984/14/4/312.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**alpha** (*PhysicalQuantity of type length**-beta*) – Potential parameter.

•**beta** (*int*) – Potential parameter.

•**g_c** (*float*) – Potential parameter.

•**g_d** (*float*) – Potential parameter.

•**g_h** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner triple potentials between silicon and fluorine.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 3,
    alpha = 5.197495*1/Angstrom**3,
    g_c = 0.0,
    g_d = 0.16,
    g_h = -0.59826,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('F', []),
    beta = 3,
    alpha = 4.0*1/Angstrom**3,
    g_c = 0.0216,
    g_d = 0.27,
```

(continues on next page)

(continued from previous page)

```
    g_h = -0.47,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('F', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 3,
    alpha = 4.0*1/Angstrom**3,
    g_c = 0.0216,
    g_d = 0.27,
    g_h = -0.47,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('Si', []),
    particleType2 = ParticleIdentifier('F', []),
    particleType3 = ParticleIdentifier('F', []),
    beta = 3,
    alpha = 4.0*1/Angstrom**3,
    g_c = 0.0216,
    g_d = 0.27,
    g_h = -0.47,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('F', []),
    particleType3 = ParticleIdentifier('F', []),
    beta = 1,
    alpha = 3.0*1/Angstrom,
    g_c = 14.0,
    g_d = 0.0,
    g_h = 0.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('F', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 1,
    alpha = 3.0*1/Angstrom,
    g_c = 14.0,
    g_d = 0.0,
    g_h = 0.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('F', []),
    beta = 1,
```

(continues on next page)

(continued from previous page)

```
    alpha = 3.0*1/Angstrom,
    g_c = 14.0,
    g_d = 0.0,
    g_h = 0.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential(
    particleType1 = ParticleIdentifier('F', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 1,
    alpha = 3.0*1/Angstrom,
    g_c = 14.0,
    g_d = 0.0,
    g_h = 0.0,
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerTriplePotential2

class TersoffBrennerTriplePotential2(*particleType1*, *particleType2*, *particleType3*, *alpha*, *beta*, *g_a*, *g_c*, *g_d*, *g_h*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **alpha** (*PhysicalQuantity of type length**-beta*) – Potential parameter.
- **beta** (*int*) – Potential parameter.
- **g_a** (*float*) – Potential parameter.
- **g_c** (*float*) – Potential parameter.
- **g_d** (*float*) – Potential parameter. Must not be zero.
- **g_h** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner triple potentials (2. variant) between silicon and carbon.

```
potentialSet = TremoloXPotentialSet(name = 'TersoffBrenner_CSiF_1999')
_potential = TersoffBrennerTriplePotential2(
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('C', []),
    particleType3 = ParticleIdentifier('C', []),
    beta = 0,
    alpha = 0,
    g_a = 0.00020813,
    g_c = 330.0,
    g_d = 3.5,
    g_h = -1.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential2(
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('C', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 0,
    alpha = 0,
    g_a = 0.00020813,
    g_c = 330.0,
    g_d = 3.5,
    g_h = -1.0,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential2(
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('C', []),
    beta = 0,
    alpha = 0,
    g_a = 1.5724e-07,
    g_c = 0.00038049,
    g_d = 4.384,
    g_h = -0.57058,
)
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential2(
```

(continues on next page)

(continued from previous page)

```
    particleType1 = ParticleIdentifier('C', []),
    particleType2 = ParticleIdentifier('Si', []),
    particleType3 = ParticleIdentifier('Si', []),
    beta = 0,
    alpha = 0,
    g_a = 1.5724e-07,
    g_c = 0.00038049,
    g_d = 4.384,
    g_h = -0.57058,
)
potentialSet.addPotential(_potential)
```

See also ref.tersoffbrennercommon.notes.

TersoffBrennerTriplePotential3

class TersoffBrennerTriplePotential3(*particleType1*, *particleType2*, *particleType3*, *alpha*, *beta*, *rho1*, *rho2*, *L*, *U*, *x1*, *f1*, *dfx1*, *dfxx1*, *x2*, *f2*, *dfx2*, *dfxx2*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **alpha** (*PhysicalQuantity of type length**-beta*) – Potential parameter.
- **beta** (*int*) – Potential parameter.
- **rho1** (*PhysicalQuantity of type length*) – Potential parameter.
- **rho2** (*PhysicalQuantity of type length*) – Potential parameter.
- **L** (*float*) – Potential parameter.
- **U** (*float*) – Potential parameter.
- **x1** (*sequence of float*) – The grid coordinates for the first spline.
- **f1** (*sequence of float*) – The function values for the first spline at the coordinates in x1.
- **dfx1** (*sequence of float*) – The first derivatives for the first spline at the coordinates in x1.
- **dfxx1** (*sequence of float*) – The second derivatives for the first spline at the coordinates in x1.
- **x2** (*sequence of float*) – The grid coordinates for the second spline.
- **f2** (*sequence of float*) – The function values for the second spline at the coordinates in x1.
- **dfx2** (*sequence of float*) – The first derivatives for the second spline at the coordinates in x1.
- **dfxx2** (*sequence of float*) – The second derivatives for the second spline at the coordinates in x1.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner triple potentials for carbon.

```
potentialSet = TremoloXPotentialSet(name = 'Brenner_CH_2002')
potentialSet.addParticleType(ParticleType(
        symbol = 'C',
        mass = 12.0107*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 6,
        tags = [],
))
_potential = TersoffBrennerTriplePotential3(
        particleType1 = ParticleIdentifier('C', []),
        particleType2 = ParticleIdentifier('C', []),
        particleType3 = ParticleIdentifier('C', []),
        beta = 1,
        alpha = 0.0*1/Angstrom,
        L = 3.2,
        U = 3.7,
        rho1 = 0.0*Angstrom,
        rho2 = 0.0*Angstrom,
        x1 = numpy.array([-1.0, -0.5, -0.333412, 0.0, 0.5, 1.0]),
        f1 = numpy.array([-0.01, 0.0528, 0.09733, 0.37545, 2.0014, 8.0]),
        dfx1 = numpy.array([0.104, 0.17, 0.4, 1.406776, 6.056472, 20.240234]),
        dfxx1 = numpy.array([0.0, 0.37, 1.98, 4.508755, 16.147742, 43.901021]),
        x2 = numpy.array([-1.0, -0.5, -0.333412, 0.0, 0.5, 1.0]),
        f2 = numpy.array([-0.01, 0.0528, 0.09733, 0.271856, 0.416335, 1.0]),
        dfx2 = numpy.array([0.104, 0.17, 0.4, 0.488916, 0.25958, 2.831613]),
        dfxx2 = numpy.array([0.0, 0.37, 1.98, -0.866163, 1.170791, 10.239179]),
)
potentialSet.addPotential(_potential)
```

tersoffbrenner_carbon.py

Notes

When added to a potential set, this class activates the three-body interactions that are used in the second generation REBO[1] and AIREBO[2] potential. This potential depends on two angle functions g1 and g2 that are usually specified by quintic splines. Therefore it is necessary to specify the values of these two functions (and the derivatives) at some given points.

Please note that interactions are added in a non-symmetric fashion. particleType1-particleType2 is associated to the bond-order pair i-j. (i is the central particle)

TersoffBrennerTriplePotential4

class TersoffBrennerTriplePotential4(*particleType1*, *particleType2*, *particleType3*, *alpha*, *beta*, *g_c*, *g_d*, *x*, *H*, *dxH*, *dxxH*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **alpha** (*PhysicalQuantity of type length**-beta*) – Potential parameter.
- **beta** (*int*) – Potential parameter.
- **g_c** (*float*) – Potential parameter.
- **g_d** (*float*) – Potential parameter.
- **x** (*sequence of floats*) – Coordinates for which the values of the function H are known. Must be sorted in ascending order.
- **H** (*sequence of floats*) – The function values of H at the points in x.
- **dxH** (*sequence of floats*) – The first derivatives of H at the points in x.
- **dxxH** (*sequence of floats*) – The second derivatives of H at the points in x.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

[1] D.ăW. Brenner, O.ăShenderova, J.ăHarrison, S.ăJ. Stuart, B.ăNi, and S.ăB Sinnott. A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons. *J. Phys.: Condensed Matter*, 14(4):783–802, 2002. arXiv:S0953-8984(02)31186-X, doi:10.1088/0953-8984/14/4/312.

[2] StevenăJ. Stuart, AlanăB. Tutein, and JudithăA. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *J. Chem. Phys.*, 112(14):6472–6486, 2000. URL: https://doi.org/10.1063/1.481208.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner triple potentials for a silane molecule.

```
potentialSet = TremoloXPotentialSet(name = 'Murty_HSi_1995')
potentialSet.addParticleType(ParticleType(
        symbol = 'Si',
        mass = 28.0855*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 14,
        tags = [],
))
potentialSet.addPotential(_potential)
_potential = TersoffBrennerTriplePotential4(
        particleType1 = ParticleIdentifier('Si', []),
        particleType2 = ParticleIdentifier('Si', []),
        particleType3 = ParticleIdentifier('H', []),
        beta = 3,
        alpha = 4.0*1/Angstrom**3,
        g_c = 0.0216,
        g_d = 0.27,
        x = numpy.array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.]),
        H = numpy.array([-0.04 , -0.04 , -0.276, -0.47 , -0.47 , -0.47 , -0.47 ]),
        dxH = numpy.array([-0.    , -0.1028, -0.2968,  0.    ,  0.    ,  0.    ,  0.   ␣
↪]),
        dxxH = numpy.array([ 0.2056, -0.4112,  0.0232, -0.    ,  0.    ,  0.    ,  0.   ␣
↪]),
)
potentialSet.addPotential(_potential)
```

tersoff_silane.py

Notes

When added to a potential set, this class activates the three-body interactions that are used in the Murty-Atwater variant of the Tersoff-Brenner potential :footcite:Murty1995`.

This potential depends on a function H that is represented by quintic splines. Therefore it is necessary to specify the values of this functions (and its derivatives) at some given points.

Please note that interactions are added in a non-symmetric fashion. particleType1-particleType2 is associated to the bond-order pair i-j. (i is the central particle)

TersoffBrennerTriplePotential5

class TersoffBrennerTriplePotential5(*particleType1*, *particleType2*, *particleType3*, *alpha*, *beta*, *g_a*, *g_c*, *g_d*, *activeTypes1*, *activeTypes2*, *x*, *y*, *h*)

Constructor of the potential.

To construct this potential it is necessary to specify a twodimensional function. This is done by passing the function values of this function at some grid points, for all other points, bicubic interpolation is used. At the border of the grid, all derivatives are assumed to be zero.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **alpha** (*PhysicalQuantity of type length**-beta*) – Potential parameter.
- **beta** (*int*) – Potential parameter.
- **g_a** (*float*) – Potential parameter.
- **g_c** (*float*) – Potential parameter.
- **g_d** (*float*) – Potential parameter. Must not be zero.
- **activeTypes1** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the N1 coefficient in the potential.
- **activeTypes2** (*sequence of* ParticleType *or* ParticleIdentifier) – List of particle types that are involved in the calculation of the N2 coefficient in the potential.
- **x** (*sequence of float*) – The x-coordinates of the grid. It must have uniform spacing.
- **y** (*sequence of float*) – The y-coordinates of the grid. It must have uniform spacing.
- **h** (*2D numpy.array*) – The function values at the given grid points. h should be a twodimensional array of size (len(x), len(y)). h[i, j] should be the function value at (x[i], y[j]).

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define Tersoff-Brenner triple potentials for silicon-carbon interactions.

```
potentialSet = TremoloXPotentialSet(name = 'Dyson_CHSi_1999')
potentialSet.addParticleType(ParticleType(
        symbol = 'C',
        mass = 12.0107*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 6,
        tags = [],
))
potentialSet.addParticleType(ParticleType(
        symbol = 'Si',
        mass = 28.0855*atomic_mass_unit,
        charge = None,
        sigma = None,
        sigma14 = None,
        epsilon = None,
        epsilon14 = None,
        atomicNumber = 14,
        tags = [],
))

_potential = TersoffBrennerTriplePotential5(
        particleType1 = ParticleIdentifier('Si', []),
        particleType2 = ParticleIdentifier('Si', []),
        particleType3 = ParticleIdentifier('C', []),
        beta = 1,
        alpha = 3.0*1/Angstrom,
        g_a = 0.01,
        g_c = 14.0,
        g_d = 2.1,
        activeTypes1 = [ParticleIdentifier('H', []), ],
        activeTypes2 = [ParticleIdentifier('Si', []), ParticleIdentifier('C', []), ],
        x = numpy.array([0, 1, 2, 3, 4, 5]),
        y = numpy.array([0, 1, 2, 3, 4, 5]),
        h = numpy.array([[-0.   , -0.5  , -1.   , -1.   , -0.   , -0.   ],
```

(continues on next page)

(continued from previous page)

```
        [-0.   , -0.049, -0.45 , -0.68 , -0.   , -0.   ],
        [-0.04 , -0.436, -0.75 , -0.   , -0.   , -0.   ],
        [-0.35 , -0.65 , -0.85 , -0.   , -0.   , -0.   ],
        [-0.6  , -0.8  , -0.   , -0.   , -0.   , -0.   ],
        [-0.   , -0.   , -0.   , -0.   , -0.   , -0.   ]]),
)
potentialSet.addPotential(_potential)
```

dyson_SiCH6.py

Notes

When added to a potential set, this class activates the three-body interactions that are used in the Dyson-Smith variant of the Tersoff-Brenner potential[1].

This potential depends on a function h that is represented by a bicubic splines. Therefore it is necessary to specify the values of this functions (and its derivatives) at some given points.

Please note that interactions are added in a non-symmetric fashion. particleType1-particleType2 is associated to the bond-order pair i-j. (i is the central particle)

TersoffDiag2Potential

class TersoffDiag2Potential(*particleType1*, *particleType2*, *A*, *B*, *R*, *S*, *l*, *mu*, *alpha*, *beta*, *omega*, *chi*, *chiR*, *m*, *n*, *c*, *d*, *h*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type energy*) – Potential parameter.
- **R** (*PhysicalQuantity of type length*) – Potential parameter.
- **S** (*PhysicalQuantity of type length*) – Potential parameter.
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **beta** (*float*) – Potential parameter.
- **omega** (*float*) – Potential parameter.
- **chi** (*float*) – Potential parameter.
- **chiR** (*float*) – Potential parameter.
- **m** (*int*) – Potential parameter.

[1] A.ăJ. Dyson and P.ăV. Smith. Improved empirical interatomic potential for C-Si-H systems. *Mol. Phys.*, 96:1491–1507, 1999. URL: https://doi.org/10.1080/00268979909483092.

- **n** (*float*) – Potential parameter.
- **c** (*float*) – Potential parameter.
- **d** (*float*) – Potential parameter.
- **h** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for gallium arsenide by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration for FCC GaAs
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.6537*Angstrom)

# Define elements
elements = [Gallium, Arsenic]

# Define coordinates
fractional_coordinates = [[ 0.00,  0.00,  0.00],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Calculator
# ------------------------------------------------------------

# Create the new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_Powell_2007')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='As',
                                          mass=74.9216*atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='Ga',
                                          mass=69.723*atomic_mass_unit))

# Add the single type potentials to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('Ga',
                                                     A=0.0*eV,
                                                     B=0.0*eV,
                                                     R=0.0*Angstrom,
                                                     S=0.0*Angstrom,
                                                     l=0.0/Angstrom,
                                                     mu=0.0/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=0.0,
                                                     omega=0.0,
                                                     chi=0.0,
                                                     chiR=0.0,
                                                     m=0,
                                                     n=0.0,
                                                     c=0.0,
                                                     d=0.0,
                                                     h=0.0))
potentialSet.addPotential(TersoffSingleTypePotential('As',
                                                     A=0.0*eV,
                                                     B=0.0*eV,
                                                     R=0.0*Angstrom,
                                                     S=0.0*Angstrom,
                                                     l=0.0/Angstrom,
                                                     mu=0.0/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=0.0,
                                                     omega=0.0,
                                                     chi=0.0,
                                                     chiR=0.0,
                                                     m=0,
                                                     n=0.0,
                                                     c=0.0,
                                                     d=0.0,
                                                     h=0.0))

# Add the diagonal type potentials to the potential set
potentialSet.addPotential(TersoffDiag2Potential('Ga', 'As',
                                                A=446079.331668*eV,
                                                B=16.0304697869*eV,
```

(continues on next page)

(continued from previous page)

```
                                                R=3.4*Angstrom,
                                                S=3.6*Angstrom,
                                                l=5.84464068024/Angstrom,
                                                mu=0.839717061922/Angstrom,
                                                alpha=0.244341/Angstrom,
                                                beta=0.257183,
                                                omega=1.0,
                                                chi=1.0,
                                                chiR=1.0,
                                                m=3,
                                                n=3.55586,
                                                c=2.16345,
                                                d=0.750147,
                                                h=-0.448899))

calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

TersoffDiagPotential

class TersoffDiagPotential(*particleType1*, *particleType2*, *A*, *B*, *R*, *S*, *l*, *mu*, *alpha*, *omega*, *chi*, *chiR*, *m*)
Constructor of the potential.

Note: TersoffSingleTypePotentials must be added for BOTH particle types!

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**A** (*PhysicalQuantity of type energy*) – Potential parameter.

•**B** (*PhysicalQuantity of type energy*) – Potential parameter.

•**R** (*PhysicalQuantity of type length*) – Potential parameter.

•**S** (*PhysicalQuantity of type length*) – Potential parameter.

•**l** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**omega** (*float*) – Potential parameter.

•**chi** (*float*) – Potential parameter.

•**chiR** (*float*) – Potential parameter.

•**m** (*int*) – Potential parameter.

classmethod getAllParameterNames()
Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for an ethane molecule by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration of a ethane molecule in a large supercell
# ------------------------------------------------------------

# Set up lattice
vector_a = [25.0, 0.0,  0.0]*Angstrom
vector_b = [0.0, 25.0,  0.0]*Angstrom
vector_c = [0.0,  0.0, 25.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Carbon, Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen, Hydrogen,
            Hydrogen]

# Define coordinates
fractional_coordinates = [[ 0.46713495,  0.49407606,  0.50000000],
                          [ 0.52151251,  0.52578179,  0.50000000],
                          [ 0.47593606,  0.44625342,  0.50000000],
                          [ 0.44187846,  0.50582697,  0.54086237],
                          [ 0.44187846,  0.50582697,  0.45913763],
                          [ 0.55812154,  0.49417303,  0.50000000],
                          [ 0.52406394,  0.55374658,  0.45913763],
                          [ 0.52406394,  0.55374658,  0.54086237]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_CH_2010')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='H',
                                          mass=1.00794 * atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='C',
                                          mass=12.0107 * atomic_mass_unit))

# Add the Tersoff potential components to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('H',
                                                     A=80.070348*eV,
                                                     B=31.379342*eV,
                                                     R=1.1*Angstrom,
                                                     S=1.7*Angstrom,
                                                     l=4.207524/Angstrom,
                                                     mu=1.795631/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=1.0,
                                                     omega=12.33,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=0,
                                                     n=1.0,
                                                     c=0.0,
                                                     d=1.0,
                                                     h=-1.0))
potentialSet.addPotential(TersoffSingleTypePotential('C',
                                                     A=1393.6*eV,
                                                     B=430.0*eV,
                                                     R=1.8*Angstrom,
                                                     S=2.1*Angstrom,
                                                     l=3.4879/Angstrom,
                                                     mu=2.2119/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=0.0,
                                                     omega=1.0,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=0,
                                                     n=0.72751,
                                                     c=38049.0,
                                                     d=4.3484,
                                                     h=-0.93))
potentialSet.addPotential(TersoffDiagPotential('C', 'H',
                                               A=297.485465*eV,
                                               B=96.824546*eV,
```

(continues on next page)

(continued from previous page)

```
                                                l=3.601111/Angstrom,
                                                mu=2.129864/Angstrom,
                                                R=1.3*Angstrom,
                                                S=1.8*Angstrom,
                                                chiR=1.0,
                                                chi=1.0,
                                                omega=0.000208,
                                                alpha=4.0/Angstrom,
                                                m=1))

potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'C', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'C', omega=0.000208, alpha=0.0/Angstrom, m=0)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'H', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'H', omega=2.71334, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'H', omega=7.1e-05, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'H', omega=0.000613, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'C', omega=0.000208, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'C', omega=56.02937, alpha=4.0/Angstrom, m=1)
)

calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

TersoffMixitPotential

class TersoffMixitPotential(*particleType1*, *particleType2*, *alpha*, *omega*, *chi*, *chiR*, *m*)

Constructor of the potential.

Note: TersoffSingleTypePotentials must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.

•**omega** (*float*) – Potential parameter.

•**chi** (*float*) – Potential parameter.

•**chiR** (*float*) – Potential parameter.

•**m** (*int*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for an ethane molecule by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration of a ethane molecule in a large supercell
# ------------------------------------------------------------

# Set up lattice
vector_a = [25.0, 0.0,  0.0]*Angstrom
vector_b = [0.0, 25.0,  0.0]*Angstrom
vector_c = [0.0,  0.0, 25.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Carbon, Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen, Hydrogen,
            Hydrogen]

# Define coordinates
fractional_coordinates = [[ 0.46713495,  0.49407606,  0.50000000],
                          [ 0.52151251,  0.52578179,  0.50000000],
                          [ 0.47593606,  0.44625342,  0.50000000],
                          [ 0.44187846,  0.50582697,  0.54086237],
                          [ 0.44187846,  0.50582697,  0.45913763],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.55812154,  0.49417303,  0.50000000],
                          [ 0.52406394,  0.55374658,  0.45913763],
                          [ 0.52406394,  0.55374658,  0.54086237]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_CH_2010')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='H',
                                          mass=1.00794 * atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='C',
                                          mass=12.0107 * atomic_mass_unit))

# Add the Tersoff potential components to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('H',
                                                     A=80.070348*eV,
                                                     B=31.379342*eV,
                                                     R=1.1*Angstrom,
                                                     S=1.7*Angstrom,
                                                     l=4.207524/Angstrom,
                                                     mu=1.795631/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=1.0,
                                                     omega=12.33,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=0,
                                                     n=1.0,
                                                     c=0.0,
                                                     d=1.0,
                                                     h=-1.0))
potentialSet.addPotential(TersoffSingleTypePotential('C',
                                                     A=1393.6*eV,
                                                     B=430.0*eV,
                                                     R=1.8*Angstrom,
                                                     S=2.1*Angstrom,
                                                     l=3.4879/Angstrom,
                                                     mu=2.2119/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=0.0,
                                                     omega=1.0,
```

(continues on next page)

(continued from previous page)

```
                                                    chi=1.0,
                                                    chiR=1.0,
                                                    m=0,
                                                    n=0.72751,
                                                    c=38049.0,
                                                    d=4.3484,
                                                    h=-0.93))
potentialSet.addPotential(TersoffDiagPotential('C', 'H',
                                               A=297.485465*eV,
                                               B=96.824546*eV,
                                               l=3.601111/Angstrom,
                                               mu=2.129864/Angstrom,
                                               R=1.3*Angstrom,
                                               S=1.8*Angstrom,
                                               chiR=1.0,
                                               chi=1.0,
                                               omega=0.000208,
                                               alpha=4.0/Angstrom,
                                               m=1))

potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'C', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'C', omega=0.000208, alpha=0.0/Angstrom, m=0)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'H', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'H', omega=2.71334, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'H', omega=7.1e-05, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'H', omega=0.000613, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'C', omega=0.000208, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'C', omega=56.02937, alpha=4.0/Angstrom, m=1)
)

calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

TersoffSingleTypePotential

class TersoffSingleTypePotential(*particleType*, *A*, *B*, *R*, *S*, *l*, *mu*, *alpha*, *beta*, *omega*, *chi*, *chiR*, *m*, *n*, *c*, *d*, *h*)

Constructor of the potential.

Parameters

- **particleType** (ParticleType *or* ParticleIdentifier) – Identifier of the particle type
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type energy*) – Potential parameter.
- **R** (*PhysicalQuantity of type length*) – Potential parameter.
- **S** (*PhysicalQuantity of type length*) – Potential parameter.
- **l** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **mu** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **beta** (*float*) – Potential parameter.
- **omega** (*float*) – Potential parameter.
- **chi** (*float*) – Potential parameter.
- **chiR** (*float*) – Potential parameter.
- **m** (*int*) – Potential parameter.
- **n** (*float*) – Potential parameter.
- **c** (*float*) – Potential parameter.
- **d** (*float*) – Potential parameter.
- **h** (*float*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for an ethane molecule by adding particle types and interaction functions to the *TremoloX-PotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration of a ethane molecule in a large supercell
# ------------------------------------------------------------

# Set up lattice
vector_a = [25.0, 0.0,  0.0]*Angstrom
vector_b = [0.0, 25.0,  0.0]*Angstrom
vector_c = [0.0,  0.0, 25.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Carbon, Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen, Hydrogen,
            Hydrogen]

# Define coordinates
fractional_coordinates = [[ 0.46713495,  0.49407606,  0.50000000],
                          [ 0.52151251,  0.52578179,  0.50000000],
                          [ 0.47593606,  0.44625342,  0.50000000],
                          [ 0.44187846,  0.50582697,  0.54086237],
                          [ 0.44187846,  0.50582697,  0.45913763],
                          [ 0.55812154,  0.49417303,  0.50000000],
                          [ 0.52406394,  0.55374658,  0.45913763],
                          [ 0.52406394,  0.55374658,  0.54086237]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_CH_2010')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='H',
                                          mass=1.00794 * atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='C',
                                          mass=12.0107 * atomic_mass_unit))

# Add the Tersoff potential components to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('H',
                                                     A=80.070348*eV,
                                                     B=31.379342*eV,
```

(continues on next page)

(continued from previous page)

```
                                                    R=1.1*Angstrom,
                                                    S=1.7*Angstrom,
                                                    l=4.207524/Angstrom,
                                                    mu=1.795631/Angstrom,
                                                    alpha=0.0/Angstrom,
                                                    beta=1.0,
                                                    omega=12.33,
                                                    chi=1.0,
                                                    chiR=1.0,
                                                    m=0,
                                                    n=1.0,
                                                    c=0.0,
                                                    d=1.0,
                                                    h=-1.0))
potentialSet.addPotential(TersoffSingleTypePotential('C',
                                                    A=1393.6*eV,
                                                    B=430.0*eV,
                                                    R=1.8*Angstrom,
                                                    S=2.1*Angstrom,
                                                    l=3.4879/Angstrom,
                                                    mu=2.2119/Angstrom,
                                                    alpha=0.0/Angstrom,
                                                    beta=0.0,
                                                    omega=1.0,
                                                    chi=1.0,
                                                    chiR=1.0,
                                                    m=0,
                                                    n=0.72751,
                                                    c=38049.0,
                                                    d=4.3484,
                                                    h=-0.93))
potentialSet.addPotential(TersoffDiagPotential('C', 'H',
                                                A=297.485465*eV,
                                                B=96.824546*eV,
                                                l=3.601111/Angstrom,
                                                mu=2.129864/Angstrom,
                                                R=1.3*Angstrom,
                                                S=1.8*Angstrom,
                                                chiR=1.0,
                                                chi=1.0,
                                                omega=0.000208,
                                                alpha=4.0/Angstrom,
                                                m=1))

potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'C', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'C', omega=0.000208, alpha=0.0/Angstrom, m=0)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'H', omega=12.33, alpha=4.0/Angstrom, m=1)
```

(continues on next page)

(continued from previous page)

```
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'H', omega=2.71334, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'H', omega=7.1e-05, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'H', omega=0.000613, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'C', omega=0.000208, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'C', omega=56.02937, alpha=4.0/Angstrom, m=1)
)

calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

TersoffTriplePotential

class TersoffTriplePotential(*particleType1*, *particleType2*, *particleType3*, *alpha*, *omega*, *m*, *alphaij=None*, *alphaik=None*)

Constructor of the potential.

Note: TersoffSingleTypePotentials must be added for ALL THREE particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **alpha** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **omega** (*float*) – Potential parameter.
- **m** (*int*) – Potential parameter.
- **alphaij** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **alphaik** (*PhysicalQuantity of type length**-1*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for an ethane molecule by adding particle types and interaction functions tothe *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration of a ethane molecule in a large supercell
# ------------------------------------------------------------

# Set up lattice
vector_a = [25.0, 0.0,  0.0]*Angstrom
vector_b = [0.0, 25.0,  0.0]*Angstrom
vector_c = [0.0,  0.0, 25.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Carbon, Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen, Hydrogen,
            Hydrogen]

# Define coordinates
fractional_coordinates = [[ 0.46713495,  0.49407606,  0.50000000],
                          [ 0.52151251,  0.52578179,  0.50000000],
                          [ 0.47593606,  0.44625342,  0.50000000],
                          [ 0.44187846,  0.50582697,  0.54086237],
                          [ 0.44187846,  0.50582697,  0.45913763],
                          [ 0.55812154,  0.49417303,  0.50000000],
                          [ 0.52406394,  0.55374658,  0.45913763],
                          [ 0.52406394,  0.55374658,  0.54086237]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_CH_2010')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='H',
                                          mass=1.00794 * atomic_mass_unit))
potentialSet.addParticleType(ParticleType(symbol='C',
                                          mass=12.0107 * atomic_mass_unit))

# Add the Tersoff potential components to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('H',
                                                     A=80.070348*eV,
                                                     B=31.379342*eV,
                                                     R=1.1*Angstrom,
                                                     S=1.7*Angstrom,
                                                     l=4.207524/Angstrom,
                                                     mu=1.795631/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=1.0,
                                                     omega=12.33,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=0,
                                                     n=1.0,
                                                     c=0.0,
                                                     d=1.0,
                                                     h=-1.0))
potentialSet.addPotential(TersoffSingleTypePotential('C',
                                                     A=1393.6*eV,
                                                     B=430.0*eV,
                                                     R=1.8*Angstrom,
                                                     S=2.1*Angstrom,
                                                     l=3.4879/Angstrom,
                                                     mu=2.2119/Angstrom,
                                                     alpha=0.0/Angstrom,
                                                     beta=0.0,
                                                     omega=1.0,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=0,
                                                     n=0.72751,
                                                     c=38049.0,
                                                     d=4.3484,
                                                     h=-0.93))
potentialSet.addPotential(TersoffDiagPotential('C', 'H',
                                               A=297.485465*eV,
                                               B=96.824546*eV,
                                               l=3.601111/Angstrom,
                                               mu=2.129864/Angstrom,
                                               R=1.3*Angstrom,
```

(continues on next page)

(continued from previous page)

```
                                                S=1.8*Angstrom,
                                                chiR=1.0,
                                                chi=1.0,
                                                omega=0.000208,
                                                alpha=4.0/Angstrom,
                                                m=1))

potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'C', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'C', omega=0.000208, alpha=0.0/Angstrom, m=0)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'H', omega=12.33, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'C', 'H', omega=2.71334, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'C', 'H', omega=7.1e-05, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'H', omega=0.000613, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('C', 'H', 'C', omega=0.000208, alpha=4.0/Angstrom, m=1)
)
potentialSet.addPotential(
    TersoffTriplePotential('H', 'H', 'C', omega=56.02937, alpha=4.0/Angstrom, m=1)
)

calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

TersoffZBLPotential

class TersoffZBLPotential(*particleType1*, *particleType2*, *type*, *Z1*, *Z2*, *a0*, *rf*, *bf*)

Constructor of the potential.

Note: TersoffSingleTypePotentials must be added for BOTH particle types!

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **type** (*int*) – either 1 or 2, sets the type of the actual potential
- **Z1** (*float*) – Potential parameter.

- **Z2** (*float*) – Potential parameter.
- **a0** (*PhysicalQuantity of type length*) – Potential parameter.
- **rf** (*PhysicalQuantity of type length*) – Potential parameter.
- **bf** (*PhysicalQuantity of type length**-1*) – Potential parameter.

classmethod getAllParameterNames()
Return the names of all used parameters as a list.

getAllParameters()
Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()
Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)
Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)
Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Tersoff potential for gold by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration for FCC gold
# ------------------------------------------------------------

# Set up lattice
vector_a = [4.07825, 0.0, 0.0]*Angstrom
vector_b = [0.0, 4.07825, 0.0]*Angstrom
vector_c = [0.0, 0.0, 4.07825]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Gold, Gold, Gold, Gold]

# Define coordinates
fractional_coordinates = [[ 0.0,  0.0,  0.0],
                          [ 0.5,  0.5,  0.0],
                          [ 0.5,  0.0,  0.5],
                          [ 0.0,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )

# -----------------------------------------------------------
# Calculator
# -----------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Tersoff_Au_2012')

# Add the particle type to the potential set
potentialSet.addParticleType(ParticleType(symbol='Au',
                                          mass=196.967*atomic_mass_unit))

# Add the single type potential to the potential set
potentialSet.addPotential(TersoffSingleTypePotential('Au',
                                                     A=5445.22104118*eV,
                                                     B=247.287913089*eV,
                                                     R=3.0*Angstrom,
                                                     S=3.4*Angstrom,
                                                     l=3.13294032317/Angstrom,
                                                     mu=1.60663606317/Angstrom,
                                                     alpha=2.05/Angstrom,
                                                     beta=1.0,
                                                     omega=0.0006374494,
                                                     chi=1.0,
                                                     chiR=1.0,
                                                     m=1,
                                                     n=1.0,
                                                     c=3.351525,
                                                     d=0.1649262,
                                                     h=-0.9941884))

# Add the ZBL modification to the potential set
potentialSet.addPotential(TersoffZBLPotential('Au', 'Au',
                                              type=1,
                                              Z1=79.0,
                                              Z2=79.0,
                                              a0=0.529*Angstrom,
                                              rf=1.7*Angstrom,
                                              bf=12.0/Angstrom))

# Create the TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
```

See also ref.tersoffcommon.notes.

ThermalConductivity

class ThermalConductivity(*md_trajectory*, *start_time=None*, *end_time=None*, *atom_selection=None*, *direction_index=None*, *time_resolution=None*, *info_panel=None*, *bin_size=None*, *heat_source_tag_name=None*, *heat_sink_tag_name=None*, *buffer_length=None*)

Class for calculating the thermal conductivity by means of an MD simulation.

Parameters

•**md_trajectory** (*MDTrajectory* | *BulkConfiguration*) – An MDTrajectory or single bulk configuration to calculate the temperature profile for.

•**start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`

•**end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.

•**atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.

•**direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Has to be an element of [0, 1, 2]. **Default:** `2`

•**time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

•**bin_size** (*PhysicalQuantity* of type length) – The bin size, which determines the resolution of the profile. **Default:** `2.0 * Angstrom`

•**heat_source_tag_name** (*str*) – Name of the heat source tag. **Default:** `heat_source`

•**heat_sink_tag_name** (*str*) – Name of the heat sink tag. **Default:** `heat_sink`

•**buffer_length** (*PhysicalQuantity* of type length) – A small buffer length if needed after the heat source and heat sink. **Default:** `2.0 * Angstrom`

areaOfConfiguration(*configuration*, *direction_index=None*)

This function calculates the surface area of a configuration.

Parameters

•**configuration** (*BulkConfiguration*) – The initial configuration on which the heat flow simulation will be performed.

•**direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Has to be an element of [0, 1, 2]. **Default:** `2`

Returns
The area of the configuration.

Return type
PhysicalQuantity of type area

slopeFromTemperatureProfile(*md_trajectory*, *temperature_profile*, *direction_index=None*, *heat_source_tag_name=None*, *heat_sink_tag_name=None*, *buffer_length=None*)

This function calculates the slope from the temperature profile.

Parameters

•**md_trajectory** (*MDTrajectory* | *BulkConfiguration*) – The MDTrajectory or configuration to calculate the slope for.

- **temperature_profile** (class:~.*TemperatureProfile*) – The temperature profile data as a histogram.
- **direction_index** (*int*) – The index of the cell vector along which the profile should be calculated. Has to be an element of [0, 1, 2]. **Default:** `2`.
- **heat_source_tag_name** (*str*) – Name of the heat source tag. **Default:** `heat_source`.
- **heat_sink_tag_name** (*str*) – Name of the heat sink tag. **Default:** `heat_sink`.
- **buffer_length** (*PhysicalQuantity* of type length) – If needed a small buffer length after the heat source and heat sink. **Default:** `2.0 * Angstrom`.

Returns
The slope of the temperature profile.

Return type
PhysicalQuantity of type temperature per length

Usage Examples

Load an *MDTrajectory*, calculate the *ThermalConductivity* along the z-axis, and print the results:

```
md_trajectory = nlread('md_trajectory.hdf5', MDTrajectory)[0]
thermal_conductivity=ThermalConductivity(
    md_trajectory,
    start_time=10*picosecond,
    end_time=md_trajectory.lastTime(),
    time_resolution=md_trajectory.timeInterval(),
    heat_source_tag_name='heat_source',
    heat_sink_tag_name='heat_sink',
)
```

`thermal_conductivity.py`

Notes

- The MD calculation should be done with *NonEquilibriumMomentumExchange* as a *measurement_hook*.
- Since the *TemperatureProfile* is always calculated along the specified Cartesian axis, it is recommended to use an orthogonal cell.
- The temperature values of empty bins in the profile are set to zero.
- In the presence of constraints the `atom_selection` parameter should be used to exclude constrained atoms from the analysis.
- *ThermalConductivity* can only can be calculated for bulk configurations.

ThermalizeSubstrate

class ThermalizeSubstrate(*fuzz_factor=None*)

Rescales the velocities of all atoms in the substrate (i.e. all atoms bonded to the fixed layer) so that the instantaneous kinetic temperature is the current thermostat temperature. This can be used between high energy events to prevent heating of the substrate.

Parameters
fuzz_factor (*float*) – The factor by which the covalent radii are multiplied to determine the cutoff distance for a bond. **Default:** 1.5

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Run an etching simulation of nitrogen on a silica surface using a ReaxFF potential, with an additional ThermalizeSubstrate hook function.

```
# Attach ReaxFF calculator.
potentialSet = ReaxFF_CHONSSi_2012(strict_bondpairs = None)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
bulk_configuration.setCalculator(calculator)

surface_process_simulation = SurfaceProcessSimulation(
    substrate=bulk_configuration,
    filename='sio2_n_etching.hdf5',
    object_id='sps_0',
    random_seed=7,
    temperature=300.0*Kelvin,
    fixed_thickness=4.0*Angstrom,
    thermostat_thickness=8*Angstrom,
    log_interval=10,
)

# Run 5 deposition events, each 20 ps long.
surface_process_simulation.addSequence(
    molecule=Nitrogen,
    number_of_events=5,
    md_time=20*ps,
    time_step=1.0*fs,
    mean_kinetic_energy=25*eV,
    std_kinetic_energy=0.3*eV,
    mean_incident_angle=0*Degrees,
    std_incident_angle=2*Degrees,
    post_md_hooks=[ThermalizeSubstrate()],
)

surface_process_simulation.update()
```

sio2_n_etching.py

The results from the simulation can be visualized using the Movie Tool on the QATK LabFloor.

See also ref.spshookscommon.notes.

TorchXPotential

class TorchXPotential(*dtype=None*, *device=None*, *file=None*, *mapping=None*, *enforceLTX=None*)

Constructor of the potential.

Parameters

- **dtype** (*str*) – The dtype of the involved torch tensors [TorchXPotential.float32, TorchXPotential.float64]. Default is TorchXPotential.float32.
- **device** (*str*) – The device to evaluate the torchscript model. E.g.: cpu, cuda, cuda:0, cuda:1. Default is to choose available cuda devices.
- **file** (*str*) – The name of the file that contains the torchscript model parameters.
- **mapping** (*dict with (ParticleIdentifier : Number)*) – A mapping from ParticleIdentifiers to TorchX-Model Particle Numbers
- **enforceLTX** (*bool*) – Enforce to use also in sequential case the LTX version if available LTX versions support local stress calculation. Note that, in parallel MPI decomposition mode LTX (if available) is used be default anyway.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a *TorchXPotential* and add it to the *TremoloXPotentialSet*. In this example the `dtype` is switched from float32 to float64.

```
potential_set = TremoloXPotential_set(name='TorchX-example')

# Add particle types for the needed elements, in this case Si and H.
potential_set.addParticleType(
    ParticleType(symbol='H')
)
potential_set.addParticleType(
```

(continues on next page)

(continued from previous page)

```
    ParticleType(symbol='Si')
)

# Add the TorchXPotential with the torch-script file trained via QuantumATK.
_potential = TorchXPotential(
    file='MACE_training_example.qatkpt',
    dtype='float64',
)
potential_set.addPotential(_potential)
calculator = TremoloXCalculator(parameters=potential_set)
```

Set up a pre-trained MACE model and switch the dtype to float64. Enforce LTX-mode (even on single GPU) to enable calculation of local stress.

```
# Set up a pre-trained MACE potential with float64 precision.
# Enable LTX-mode to calculate local stress.
potential_set = TorchX_MACE_MP_0_L0_2023(dtype='float64', enforceLTX=True)
calculator = TremoloXCalculator(parameters=potential_set)

bulk_configuration.setCalculator(calculator)

# Calculate and print the local stress with the MACE ML-FF.
local_stress = LocalStress(bulk_configuration)
nlprint(local_stress)
```

Notes

The *TorchXPotential* class can be used to include torch-based machine-learned FF models, such as MACE[1]. The primary use case is to run simulations with user trained ML-FF models, after training the model using the *MachineLearnedForceFieldTrainer*. The `.qatkpt` file which results from the training workflow can be loaded in the *TorchXPotential*, either in script or using the **MachineLearnedForceField** block in the **Workflow Builder**. The path to the torch-script should be passed via the `file` argument, when constructing the *TorchXPotential* object.

For external user-trained models, currently only models based on the MACE architecture are supported. A given model can be converted using the function *convertMACEModelToQATKFormat*.

The `dtype` keyword can be used to switch the floating point precision that is used for calculating the energy, forces, and stress. The default `float32` value is normally sufficient for MD simulations, as it provides a better performance. However, for calculations that require a higher accuracy, e.g. *DynamicalMatrix*, *OptimizeNudgedElasticBand*, etc. it is recommended to use `float64` instead.

The `enforceLTX` keyword can be used to manually switch between standard serial mode (`enforceLTX=False`) and multi-GPU mode (`enforceLTX=True`), which enables MPI communication of node features between message-passing layers of the ML-FF (currently supported only for MACE and MatterSim models). By default, the modes are selected automatically, depending on whether the calculation is run with GPU and multiple MPI processes or not. One can use this flag to enforce the multi-GPU mode even when running serial by setting `enforceLTX=True`. This enables calculation of *LocalStress*, which is otherwise not supported for *TorchXPotential*.

The `device` keyword can be used to manually disable or enable GPU acceleration. However, the recommended approach is to leave this parameter at the default value `None`, and enable the GPU acceleration as described in the *technical notes*.

[1] Ilyes Batatia, Simon Batzner, DávidăPéter Kovács, Albert Musaelian, Gregor N.ăC. Simm, Ralf Drautz, Christoph Ortner, Boris Kozinsky, and Gábor Csányi. The design space of e(3)-equivariant atom-centered interatomic potentials. 2022. arXiv:2205.06643, doi:10.48550/arXiv.2205.06643.

The mapping keyword can be used to specify a dict between atomic symbols and atomic numbers used to encode the elements in most ML-FF backends. The mapping is not needed for models trained with QuantumATK, or converted using *convertMACEModelToQATKFormat*.

TosiFumiPotential

class TosiFumiPotential(*particleType1*, *particleType2*, *A*, *B*, *C*, *D*, *sigma*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType) – Identifier of the first particle type.
- **particleType2** (ParticleType) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **C** (*PhysicalQuantity of type length**6 * energy*) – Potential parameter.
- **D** (*PhysicalQuantity of type length**8 * energy*) – Potential parameter.
- **sigma** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a potential for perovskite MgSiO3 by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# ------------------------------------------------------------
# Bulk configuration for the MgSiO3 perovskite structure
# ------------------------------------------------------------

# Set up lattice
lattice = SimpleOrthorhombic(4.7786*Angstrom, 4.9293*Angstrom, 6.9003*Angstrom)

# Define elements
elements = [Magnesium, Magnesium, Magnesium, Magnesium, Silicon, Silicon,
            Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.9856,  0.0564,  0.2500],
                          [ 0.0144,  0.9436,  0.7500],
                          [ 0.4856,  0.4436,  0.7500],
                          [ 0.5144,  0.5564,  0.2500],
                          [ 0.0000,  0.500 ,  0.0000],
                          [ 0.0000,  0.500 ,  0.5000],
                          [ 0.5000,  0.0000,  0.5000],
                          [ 0.5000,  0.0000,  0.0000],
                          [ 0.1015,  0.4673,  0.2500],
                          [ 0.8985,  0.5327,  0.7500],
                          [ 0.6015,  0.0327,  0.7500],
                          [ 0.3985,  0.9673,  0.2500],
                          [ 0.6962,  0.2983,  0.0524],
                          [ 0.6962,  0.2983,  0.4476],
                          [ 0.3038,  0.7017,  0.5524],
                          [ 0.1962,  0.2017,  0.5524],
                          [ 0.8038,  0.7983,  0.4476],
                          [ 0.8038,  0.7983,  0.0524],
                          [ 0.1962,  0.2017,  0.9476],
                          [ 0.3038,  0.7017,  0.9476]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create a new potential set
potentialSet = TremoloXPotentialSet(name='Matsui_MgOSi_1987')
```

(continues on next page)

(continued from previous page)

```
# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Mg',
                                          mass=24.305*atomic_mass_unit,
                                          charge=1.565))
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
                                          charge=2.329))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.298))

# Add the Tosi-Fumi type pair potentials to the potential set
potentialSet.addPotential(TosiFumiPotential('Mg', 'Mg',
                                            A=0.0045098664*eV,
                                            B=9.6153846*1/Ang,
                                            C=0.0*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=2.0534*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential('Si', 'Si',
                                            A=0.003469128*eV,
                                            B=12.5*1/Ang,
                                            C=0.0*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=1.7058*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential('O', 'O',
                                            A=0.01300923*eV,
                                            B=3.3333333*1/Ang,
                                            C=30.222207*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=3.5304*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential('Mg', 'Si',
                                            A=0.0039894972*eV,
                                            B=10.869565*1/Ang,
                                            C=0.0*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=1.8796*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))
potentialSet.addPotential(TosiFumiPotential('Mg', 'O',
                                            A=0.0087595482*eV,
                                            B=4.950495*1/Ang,
                                            C=0.0*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=2.7919*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))
```

(continues on next page)

(continued from previous page)

```
potentialSet.addPotential(TosiFumiPotential('Si', 'O',
                                            A=0.008239179*eV,
                                            B=5.2631579*1/Ang,
                                            C=0.0*Ang**6*eV,
                                            D=0.0*Ang**8*eV,
                                            sigma=2.6181*Angstrom,
                                            r_i=6.0*Angstrom,
                                            r_cut=7.5*Angstrom))

# Add the Coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.200000))

# Create a TremoloX calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *TosiFumiPotential*[1] defines a pair potential of the form

$$V_{ij}(r) = Ae^{B(\sigma - r)} - \frac{C}{r^6} - \frac{D}{r^8}\,.$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

TotalEnergy

class TotalEnergy(*configuration*)

Class for calculating the total energy

Parameters

configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – Configuration with a calculator that supports total energy calculations.

[1] M.ăP. Tosi and F.ăG. Fumi. Ionic sizes and born repulsive parameters in the NaCl-type alkali halides-II. *J. Phys. Chem. Solids*, 25(1):45–52, jan 1964. doi:10.1016/0022-3697(64)90160-X.

`alternativeEnergies`()

Returns
A dictionary containing the values of other types of energies if available. If none are available it returns an empty dictionary.

Return type
dict

`components`()

Returns
A dictionary containing the energy components of the total free energy.

Return type
dict

`evaluate`()

Returns
The total free energy of the system.

Return type
PhysicalQuantity of type energy

`metatext`()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo`()

Returns
The total energy information.

Return type
dict

`nlprint`(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`setMetatext`(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

`uniqueString`()

Return a unique string representing the state of the object.

Usage Examples

Calculate the total energy of a hydrogen molecule:

```
#setup H-H geometry
elements = [ Hydrogen, Hydrogen]
cartesian_coordinates = [[ 0.0,0.0,0.0],
                         [ 0.0,0.0,0.75]]*Angstrom
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

#define calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)

#calculate total energy and individual components
total_energy = TotalEnergy(molecule_configuration)
components = total_energy.components()

#print total energy and individual components
print()
print('-------------------------------------------------')
for term in components.keys():
    print('%-30s = %12.4f eV' % (term,components[term].inUnitsOf(eV)))
print('-------------------------------------------------')
energy = total_energy.evaluate().inUnitsOf(eV)
print('Total energy                   = %12.4f eV' % energy)
```

h2_energy.py

Calculate the equilibrium bond length and dissociation energy of a hydrogen molecule:

```
#define calculator
calculator = LCAOCalculator()

#initialize arrays
energies = numpy.array([])
old_configuration = None

#setup distances where h2 energy is evaluated
distances = numpy.linspace(0.7,0.8,6)

#calculate total energy as function of distance
for d in distances:
    #setup H-H geometry
    elements = [ Hydrogen, Hydrogen]
    cartesian_coordinates = [[ 0.0,0.0,0.0],
                         [ 0.0,0.0,d]]*Angstrom
    molecule_configuration = MoleculeConfiguration(
        elements=elements,
        cartesian_coordinates=cartesian_coordinates
        )
```

(continues on next page)

(continued from previous page)

```
    # set the calculator
    if old_configuration == None:
        molecule_configuration.setCalculator(calculator())
    else:
        molecule_configuration.setCalculator(calculator(),
                initial_state=old_configuration)
    #store the scf state
    old_configuration = molecule_configuration
    #calculate total energy
    total_energy = TotalEnergy(molecule_configuration)
    #append the total energy to energies
    energies = numpy.append(energies,total_energy.evaluate().inUnitsOf(eV))

#get the energy of atomic hydrogen
elements = [ Hydrogen]
cartesian_coordinates = [[ 0.0,0.0,0.0]]*Ang
molecule_configuration = MoleculeConfiguration(
        elements=elements,
        cartesian_coordinates=cartesian_coordinates
        )
molecule_configuration.setCalculator(calculator())
h_energy = TotalEnergy(molecule_configuration)

#calculate energies relative to atomic hydrogen
energies = energies - 2.*h_energy.evaluate().inUnitsOf(eV)

#fit polynomium to data
(a,b,c) = numpy.polyfit(distances,energies,2)

#print out the results
print()
print('distance(Angstrom) Energy (eV) polynomial_fit (eV)')
for i in range(len(distances)):
    d=distances[i]
    energy_fit = a*d*d+b*d+c
    print(' %8.2f         %8.4f         %8.4f '% (d,energies[i],energy_fit))

#calculate the bonding distance (experimental 0.742* Angstrom)
d_min = -b/(2.*a)
print('bonding distance     = %8.3f Angstrom'% d_min)

# calculate vibrational frequency (experimental 4395 cm-1)
k = 2.*a*eV/(Ang*Ang)
effective_mass = Hydrogen.atomicMass()/math.sqrt(2)
omega = (k/effective_mass)**(0.5)
wave_number = (omega/speed_of_light/2./math.pi).inUnitsOf(Meter**-1)/100.
print('vibrational frequency = %8.f cm-1 '% (wave_number))

#calculate H-H bond dissociation energy (experimental 436 kJ/mol)
e_min = (a*d_min*d_min+b*d_min+c)*eV
#add zero point energy
e_min = e_min + 0.5 * hbar * omega
```

(continues on next page)

(continued from previous page)

```
#convert to kj/mol
e_min_kjmol = (e_min*avogadro_number).inUnitsOf(Joule/Units.Mol)/1000.
print('dissociation energy   = %8.f kJ/mol'% e_min_kjmol)
```

h2.py

Note: To obtain a more accurate dissociation energy of the hydrogen molecule, we must perform a spin polarized calculation of the reference energy of the hydrogen atom.

Notes LCAOCalculator

In *ATK-DFT*, the *TotalEnergy* object returns the free energy given by

$$F[n] = T[n] + E^{\mathrm{xc}}[n] + E^{H}[n] + E^{\mathrm{ext}}[n] - \sigma S,$$

where n is the electron density.

The terms in this equation are:

- $T[n]$ is the kinetic energy of the Kohn-Sham orbitals and obtained with the method `E.components()['Kinetic']`.
- $E^{\mathrm{xc}}[n]$ is the exchange-correlation energy and obtained with the method `E.components()['Exchange-Correlation']`.
- $E^{H}[n]$ includes all the electrostatic terms, i.e. the Hartree energy and the interaction energy with the pseudo potential ions. The sum of these terms is obtained with the method `E.components()['Electrostatic']`.
- $E^{\mathrm{ext}}[n]$ is the interaction energy with an external field. The value of this term is obtained with the method `E.components()['External-Field']`.
- $-\sigma S$ is the entropy contribution due to smearing of the occupation function. Here σ is the broadening of the occupation function and S is the generalized entropy, see *Occupation Methods*. The value of this term is obtained with the method `E.components()['Entropy-Term']`.

It is possible to approximately correct for the effects introduced by using a smeared occupation function with a finite broadening, σ, by the formula

$$E_{\sigma\to 0}(\sigma) = F_{\sigma\to 0}(\sigma) = E(\sigma) + \Delta E_{\sigma\to 0},$$

where the correction term, $\Delta E_{\sigma\to 0}$ is determined by the specific occupation method used. This extrapolated zero-broadening energy is also written to output, but can also be obtained with the expression `E.alternativeEnergies()['Zero-Broadening-Energy']`.

To print the data of the total energy, use the method *nlprint*.

Notes DeviceLCAOCalculator

For a *DeviceConfiguration*, the *TotalEnergy* object will, in a DFT calculation, contain the free energy functional given by

$$F[n] = T[n] + E^{\mathrm{xc}}[n] + E^{H}[n] + E^{\mathrm{ext}}[n] - e\,\Delta N_L\,\mu_L - e\,\Delta N_R\,\mu_R,$$

where n is the electron density. Compared to the bulk system, the electronic entropy term is missing, $-\sigma S$. The entropy term is not included since it usually is very insignificant and very difficult and time consuming to calculate for a *DeviceConfiguration*.

The last two terms in the above equation arise from charge flow between the central region and the electrode reservoirs[1]. The symbol μ_L (μ_R) is the electro-chemical potential of the left (right) electrode and ΔN is the excess number of electrons from the electrode. If the central region is charge neutral, we will have $\Delta N_L + \Delta N_R = 0$. As detailed below, for finite bias systems, these reservoir terms are calculated approximately and the use of the total energy object is therefore approximate for a *DeviceConfiguration* at finite bias. Therefore, the calculated force may in this case not be fully consistent with the derivative of the total energy.

The terms in the equation are:

- $T[n]$ is the kinetic energy of the Kohn-Sham orbitals and obtained with the method `E.components()['Kinetic']`.
- $E^{\mathrm{xc}}[n]$ is the exchange-correlation energy and obtained with the method `E.components()['Exchange-Correlation']`.
- $E^{H}[n]$ includes all the electrostatic terms, i.e. the Hartree energy and the interaction energy with the pseudo potential ions. The sum of these terms is obtained with the method `E.components()['Electrostatic']`.
- $E^{\mathrm{ext}}[n]$ is the interaction energy with an external field. This term is obtained with the method `E.components()['External-Field']`.
- $-e\,\Delta N_L\,\mu_L$ is the energy of the electrons which have entered the central region from the left reservoir. The number of electrons, ΔN_L, is calculated approximately as the Mulliken charge of the atoms closest to the left electrode. This term is obtained with the method `E.components()['Reservoir-Left']`.
- $-e\,\Delta N_R\,\mu_R$ is the energy of the electrons which have entered the central region from the right reservoir. The number of electrons, ΔN_R, is calculated approximately as the Mulliken charge of the atoms closest to the right electrode. This term is obtained with the method `E.components()['Reservoir-Right']`.

To print the data of the *TotalEnergy*, use the method *nlprint*.

Notes HuckelCalculator

In *ATK-SE*, the *TotalEnergy* object returns a free energy containing only attractive total energy terms,

$$F[n] = E^{0}_{\mathrm{1el}} + E^{H}[n] + E^{\mathrm{ext}}[n] - \sigma S,$$

where n is the electron density. The total energy is missing a repulsive pair potential term. The energy can therefore only be used to compare the total energy of systems with the same distances between atoms. This, is for instance the case for the calculation of charging energies, i.e. the difference in the total energy as function of the charge on a molecule[2].

The terms in the equation are:

- $E^{0}_{\mathrm{1el}} = E_{\mathrm{1el}} - \int V^{H}(\mathbf{r}) n(\mathbf{r}) d\mathbf{r}$ is the one-electron energy, subtracted electrostatic double counting terms. The term may also be written as $E^{0}_{\mathrm{1el}} = \mathrm{Tr}[H^{0}D]$, where H^{0} is the non-self-consistent part of the Hückel Hamiltonian and D is the density matrix. The value of the term is obtained with the method `E.components()['One-Electron']`.
- $E^{H}[n]$ is the Hartree energy and obtained with the method `E.components()['Hartree']`.
- $E^{\mathrm{ext}}[n]$ is the interaction energy with an external field. The value of this term is obtained with the method `E.components()['External-Field']`.
- $-\sigma S$ is the entropy contribution due to smearing of the occupation function. Here σ is the broadening of the occupation function and S is the generalized entropy, see *Occupation Methods*. The value of this term is obtained with the method `E.components()['Entropy-Term']`.

[1] A.ăP. Sutton and T.ăN. Todorov. A Maxwell relation for current-induced forces. *Mol. Phys.*, 102(9-10):919–925, 2004. doi:10.1080/00268970410001703354.

[2] K.ăKaasbjerg and K.ăFlensberg. Strong Polarization-Induced Reduction of Addition Energies in Single-Molecule Nanojunctions. *Nano Letters*, 8(11):3809–3814, 2008. doi:10.1021/nl8021708.

It is possible to approximately correct for the effects introduced by using a smeared occupation function with a finite broadening, σ, by the formula

$$E_{\sigma\to 0}(\sigma) = F_{\sigma\to 0}(\sigma) = E(\sigma) + \Delta E_{\sigma\to 0},$$

where the correction term, $\Delta E_{\sigma\to 0}$ is determined by the specific occupation method used. This extrapolated zero-broadening energy is also written to output, but can also be obtained with the expression `E.alternativeEnergies()['Zero-Broadening-Energy']`.

To print the data of the *TotalEnergy*, use the method *nlprint*.

Notes DeviceHuckelCalculator

For a *DeviceConfiguration*, the *TotalEnergy* object in a Hückel calculation will contain the terms

$$E[n] = E^0_{\rm 1el} + E^H[n] + E^{\rm ext}[n] - e\,\Delta N_L\,\mu_L - e\,\Delta N_R\,\mu_R,$$

where n is the electron density. Compared to the bulk system, the *DeviceConfiguration* total energy object is missing the electronic free energy term, $-\sigma S$. This term is omitted for convenience, since it is very difficult and time consuming to calculate for a *DeviceConfiguration*. For most systems this term is insignificant.

There are two new terms arising from charge flow between the central region and the electrode reservoirs. The reservoir terms are calculated approximately and the use of the total energy object is therefore approximate for a *DeviceConfiguration*.

The terms in the equation are:

- $E^0_{\rm 1el} = E_{\rm 1el} - \int V^H(\mathbf{r}) n(\mathbf{r}) d\mathbf{r}$ is the one-electron energy, subtracted electrostatic double counting terms. The term may also be written as $E^0_{\rm 1el} = {\rm Tr}[H^0 D]$, where H^0 is the non-self-consistent part of the Hückel Hamiltonian and D is the density matrix. The value of the term is obtained with the method `E.components()['One-Electron']`.
- $E^H[n]$ is the Hartree energy and obtained with the method `E.components()['Hartree']`.
- $E^{\rm ext}[n]$ is the interaction energy with an external field. The value of this term is obtained with the method `E.components()['External-Field']`.
- $-e\,\Delta N_L\,\mu_L$ is the energy of the electrons which have entered the central region from the left reservoir. The number of electrons, ΔN_L, is calculated approximately as the Mulliken charge of the atoms closest to the left electrode. This term is obtained with the method `E.components()['Reservoir-Left']`.
- $-e\,\Delta N_R\,\mu_R$ is the energy of the electrons which have entered the central region from the right reservoir. The number of electrons, ΔN_R, is calculated approximately as the Mulliken charge of the atoms closest to the right electrode. This term is obtained with the method `E.components()['Reservoir-Right']`.

To print the data of the total energy, use the method *nlprint*.

TrainingSet

class TrainingSet(*configurations*, *sample_size=None*, *calculator=None*, *recalculate_training_data=None*, *log_filename_prefix=None*, *data_tag=None*)

Class for storing a set of training configurations.

Parameters

- **configurations** (*MoleculeConfiguration* | *BulkConfiguration* | sequence of[*MoleculeConfiguration* | *BulkConfiguration*] | *Trajectory* | *MDTrajectory* | *ConfigurationDataContainer* | *MomentTensorPotentialTraining* | *SurfaceProcessSimulation* | *NudgedElasticBand* | *Table*) – The training set, either as a sequence of individual configurations or stored within a trajectory object. When using a trajectory object, energy, force and stress data may also be included.

- **sample_size** (*int*) – The number of configurations to use in the list provided, spaced out evenly. **Default:** All configurations.
- **calculator** (*Calculator*) – The calculator that was used to calculate the energy, force and stress data on the trajectory object, if present. **Default:** energy, force and stress data is ignored.
- **recalculate_training_data** (*bool | None*) – Flag to enforce or avoid recalculation of training data. If set, this flag will take precedence over the calculator. If not set, the data will be automatically recalculated if the specified calculator is different from the reference calculator in the MomentTensorPotentialTraining object. **Default:** None
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting.

calculator()

Returns
The calculator that was used to calculate the energy, force and stress data on the trajectory object, or None if energy, force and stress data is not present or should be ignored.

Return type
Calculator | None

configurations()

Returns
The configurations in the training set. If the configurations have the same elements they are returned as an *MDTrajectory*, otherwise the configurations are returned as a *ConfigurationDataContainer*.

Return type
ConfigurationDataContainer | *MDTrajectory*

dataTag()

Returns
The selection tag added to the data in the training set.

Return type
str

logFilenameIdentifier()

Returns
Filename identifier for the logging output of the tasks associated with this set, or None if it hasnt been set yet.

Return type
str | None

logFilenamePrefix()

Returns
Filename prefix for the logging output of the tasks associated with this set, or None if it is to be defined by the *MomentTensorPotentialTraining* object.

Return type
str | LogToStdOut | None

nlinfo()

Returns
The training set information.

Return type
dict

recalculateTrainingData()

Returns
A flag signaling whether the training data should be recalculated or not.

Return type
bool

referenceConfigurations()

Returns
The list of reference configurations that identify the training set.

Return type
list of [*MoleculeConfiguration* | *BulkConfiguration*]

sampleSize()

Returns
The number of training configurations for each combination of list parameters.

Return type
int

classmethod supportedConfigurationTypes()
Return a list of supported configurations.

Returns
List of supported configurations.

Return type
list

uniqueString()
Return a unique string representing the state of the object.

Usage Examples

Setup of an MTP training by reading pre-calculated training data and passing it as *TrainingSet*:

```
# Import a pre-calculated training dataset.
training_data = nlread('training_data_precalculated.hdf5')[-1]
training_set_precalc = TrainingSet(training_data, recalculate_training_data=False)

# Import a training dataset with a calculator to check for calculation consistency.
# In this case the calculator is different to the one given in the␣
↪MomentTensorPotentialTraining
# object, and so the training data will be re-calculated.
calculator = LCAOCalculator(exchange_correlation=GGA.PBE)
training_data = nlread('training_data_recalculate.hdf5')[-1]
training_set_recalc = TrainingSet(training_data, calculator=calculator)

# Import a training dataset with a calculator to check for calculation consistency.
# In this case the calculator is the same as the one given in the␣
↪MomentTensorPotentialTraining
```

(continues on next page)

(continued from previous page)

```
# object, and so the training data will not be re-calculated.
calculator = LCAOCalculator(exchange_correlation=HybridGGA.HSE06)
training_data = nlread('training_data_same_calculator.hdf5')[-1]
training_set_same_calc = TrainingSet(training_data, calculator=calculator)

# Set up MTP training and run the training data calculation and MTP training.
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_study.hdf5',
    object_id='training',
    training_sets=[training_set_precalc, training_set_recalc, training_set_same_calc],
    calculator=LCAOCalculator(exchange_correlation=HybridGGA.HSE06),
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters,
)
moment_tensor_potential_training.update()
```

TrainingSet_example.py

Here, the first training dataset is included as is, without recalculating any of the energy, force or stress values. For the second training dataset, a calculator is given. As this calculator is different to the one given in the *MomentTensorPotentialTraining*, the training data will be recalculated. In the third training set the same calculator is given as in the *MomentTensorPotentialTraining* object. Here, if training data is given for each configuration, the training data will not be re-calculated.

The following example shows how to save the training data from a completed *MomentTensorPotentialTraining* object in a *TrainingSet* and read it again to be used in another training.

```
# Run the training calculations using the MomentTensorPotentialTraining
moment_tensor_potential_training = MomentTensorPotentialTraining(
    filename='mtp_study',
    object_id='training',
    training_sets=training_sets,
    calculator=calculator,
    calculate_stress=True,
)
moment_tensor_potential_training.update()

# Pack the completed MomentTensorPotentialTraining object in a TrainingSet and save it.
training_set = TrainingSet(
   moment_tensor_potential_training,
   calculator=calculator,
   recalculate_training_data=False,
)

nlsave('mtp_training_set1.hdf5', training_set)

# Read it again and combine with other training sets from disc to run a new fit.
training_sets = [
   nlread('mtp_training_set1.hdf5', TrainingSet)[0],
   nlread('mtp_training_set2.hdf5', TrainingSet)[0],
   nlread('mtp_training_set3.hdf5', TrainingSet)[0],
]
new_mtp_training = MomentTensorPotentialTraining(
```

(continues on next page)

(continued from previous page)

```
    filename='mtp_study_from_training_sets.hdf5',
    object_id='training',
    training_sets=training_sets,
    calculator=calculator,
    calculate_stress=True,
    fitting_parameters=MomentTensorPotentialFittingParameters(),
    train_test_split=0.95,
)
new_mtp_training.update()
nlprint(new_mtp_training)
```

Notes

The *TrainingSet* class can be used to include existing configurations, MD, optimization, *ConfigurationDataContainer*, or *SurfaceProcessSimulation* trajectories, as well as pre-calculated *MomentTensorPotentialTraining* objects in the training data of another *MomentTensorPotentialTraining*.

This can be useful when combining existing training data from different sources or projects to fit a new Moment Tensor Potential. It can also be used to efficiently re-calculate DFT data for existing un-labelled configurations, i.e. configurations without DFT energy, forces, and stress.

By default training data is re-calculated using the calculator in the *MomentTensorPotentialTraining* object. To keep the original data in the training set, the argument `recalculate_training_data` can be set to `False`. This stops re-calculation of data regardless of calculator settings. In this case if any training data is missing, an error will be raised in the *MomentTensorPotentialTraining* object. It is also possible to re-calculate data based on the consistency of calculator settings. The argument `calculator` takes the calculator used to generate the energy, force and stress data. If this calculator is the same as the calculator given in the *MomentTensorPotentialTraining* object, the original data is kept and not re-calculated. If the calculators differ or there is missing training data then the dataset is re-calculated using the *MomentTensorPotentialTraining* calculator.

It is generally recommended to save training data as *TrainingSet* objects, which can easily be read and combined with other TrainingSet objects in a list or *Table* to be re-used in other training scenarios (as shown in the example above).

Trajectory

class Trajectory(*path=None*, *object_id=None*)

Construct a Trajectory object that contains the configurations from an optimization run.

Parameters

- **path** (*str*) – The path to the HDF5 file to store the results in.
- **object_id** (*str | None*) – The object id to use when saving to HDF5. **Default:** None

image(*image_index*)

Parameters
image_index (*int*) – The index of the desired image.

Returns
The configuration for this image.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

imageConstraints(*image_index*)

Parameters
image_index (*int*) – The index of the desired image.

Returns
The constraint objects for the image.

Return type
list of type BaseConstraint

imageEnergy(*image_index*)

Parameters
image_index (*int*) – The index of the desired image.

Returns
The energy of the image.

Return type
PhysicalQuantity of type energy

imageForces(*image_index*, *include_constraints=True*)

Parameters

- **image_index** (*int*) – The index of the desired image.
- **include_constraints** (*bool*) – If this object was generated by an geometry optimization with constraints, determine whether the output forces should include constraint corrections or not.

Returns
The forces of the image.

Return type
PhysicalQuantity of type force (e.g. eV/Angstrom)

imageOptimizerMethod(*image_index*)

Parameters
image_index (*int*) – The index of the desired image.

Returns
The optimizer method object for the image.

Return type
OptimizerMethod

imageStress(*image_index*, *include_constraints=True*)

Parameters

- **image_index** (*int*) – The index of the desired image.
- **include_constraints** (*bool*) – If this object was generated by an geometry optimization with constraints, determine whether the output stress should include constraint corrections or not.

Returns
The stress of the image.

Return type
PhysicalQuantity of type pressure (e.g. GPa)

images()

Returns
All the configurations in the trajectory.

Return type
list of *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

lastImage()

Returns
The last configuration in the trajectory.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

length()

Returns
The number of configurations in the trajectory.

Return type
int

metatext()

Returns
The metatext of the object or None if no metatext is set.

Return type
str | None

setMetatext(*text*)
Set a given metatext string on the object.

Parameters
text (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

Usage Examples

Optimize the geometry of a water molecule and read the result into a trajectory object

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,  -1.70000000e-05,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# define a calculator
```

(continues on next page)

(continued from previous page)

```
molecule_configuration.setCalculator(LCAOCalculator())

# perform optimization and read the relaxation trajectory
OptimizeGeometry(molecule_configuration, trajectory_filename='opt.nc')
trajectory = nlread('opt.nc', Trajectory)[0]
```

trajectory.py

TransmissionEigenstate

class TransmissionEigenstate(*configuration=None*, *energy=None*, *k_point=None*, *quantum_number=None*, *spin=None*, *contributions=None*, *energy_zero_parameter=None*)

Class for representing the transmission eigenstate for a given configuration and calculator. Noncollinear and Spin-Orbit spin is currently not supported by TransmissionEigenstate.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which the transmission eigenstate should be calculated.
- **energy** (*PhysicalQuantity* of type energy) – The energy for which the transmission eigenstate should be calculated. **Default:** `0.0*eV`
- **k_point** (*list of two floats.*) – The 2-dimensional k-point in fractional coordinates for which the transmission eigenstate should be calculated (x,y), e.g. [0.8, 0.2]. **Default:** `[0.0, 0.0]`
- **quantum_number** (*integer*) – The quantum number of the desired eigenstate. **Default:** `0`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin to be used for the transmission eigenstate calculation. **Default:** `Spin.All`
- **contributions** (`Left` | `Right`) – The initial state electrode of the transmission eigenstate. **Default:** `Left`
- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`.) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`

absolute()

Returns
A new grid containing the absolute values (or modulus) of the current field.

Return type
GridValues

axisProjection(*projection_type='sum'*, *axis='c'*, *spin=None*, *projection_point=None*, *coordinate_type=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Fractional'>*)

Get the values projected on one of the grid axes.

Parameters

- **projection_type** (*str*) –

The type of projection to perform. Should be either

– sum for the sum over the plane spanned by the two other axes.

– average or avg for the average value over the plane spanned by the two other axes.

–line for the value along a line parallel to the axis and through a point specified by the *projection_point* parameter.

Default: sum

- **axis** (*str*) – The axis to project the data onto. Should be either a, b or c. **Default:** c
- **spin** (Spin.Sum | Spin.Z | Spin.X | Spin.Y | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – Which spin component to project on. **Default:** Spin.All
- **projection_point** (sequence, PhysicalQuantity) – Axis coordinates of the point through which to take a line if projection_type is projection_point. Must be given as a sequence of three coordinates *[a, b, c]*. It the numbers have units of length, they are first divided by the length of the respective primitive vectors [A, B, C], and then interpreted as fractional coordinates. Unitless coordinates are immidiately interpreted as fractional.
- **coordinate_type** (Fractional | Cartesian) – Flag to toggle if the returned axis values should be given in units of Angstrom (NLFlag.Cartesian) or in units of the norm of the axis primitive vector (NLFlag.Fractional). **Default:** Fractional

Returns
A 2-tuple of 1D numpy.arrays containing the axis values and the projected data. For Cartesian coordinate type the grid offset is added to the axis values.

Return type
tuple.

contributions()

Returns
The contributions used in the transmission eigenstate calculation.

Return type
Left | Right

derivatives(*x*, *y*, *z*, *spin=None*)

Calculate the derivative of the transmission eigenstate wave function in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length.) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length.) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length.) – The Cartesian z coordinate.
- **spin** (Spin.All | Spin.Up | Spin.Down) – The spin component to project on. **Default:** The spin this TransmissionEigenstate object was calculated for.

Returns
The gradient at the specified point for the given spin. For Spin.All, a tuple with (Spin.Up, Spin.Down) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-1/2}$ Œ length$^{-5/2}$

downsample(*downsampling_a=None*, *downsampling_b=None*, *downsampling_c=None*)

Generate a new GridValues object where the grid is downsampled. Along periodic directions an FFT downsampling is performed. Along non-periodic directions antialiasing and downsampling is performed.

Parameters

- **downsampling_a** (*int*) – The new number of grid points along the A direction. **Default:** No downsampling.
- **downsampling_b** (*int*) – The new number of grid points along the B direction. **Default:** No downsampling.
- **downsampling_c** (*int*) – The new number of grid points along the C direction. **Default:** No downsampling.

`electrodeFermiLevels()`

Returns
The left and right electrodes Fermi levels in absolute energies.

Return type
PhysicalQuantity

`energy()`

Returns
The energy used in this transmission eigenstate calculation.

Return type
PhysicalQuantity of type energy

`energyZero()`

Returns
The energy zero value.

Return type
PhysicalQuantity of type energy

`energyZeroParameter()`

Returns
The specified choice for the energy zero.

Return type
AverageFermiLevel | AbsoluteEnergy

`evaluate(x, y, z, spin=None)`

Evaluate the transmission eigenstate wave function in the point (x, y, z).

Parameters

- **x** (*PhysicalQuantity* with type length.) – The Cartesian x coordinate.
- **y** (*PhysicalQuantity* with type length.) – The Cartesian y coordinate.
- **z** (*PhysicalQuantity* with type length.) – The Cartesian z coordinate.
- **spin** (`Spin.All` | `Spin.Up` | `Spin.Down`) – The spin component to project on. **Default:** The spin this TransmissionEigenstate object was calculated for.

Returns
The value at the specified point for the given spin. For `Spin.All`, a tuple with (`Spin.Up`, `Spin.Down`) components is returned if the calculation is not unpolarized.

Return type
PhysicalQuantity of type energy$^{-1/2}$ Œ length$^{-3/2}$

`gridCoordinate(i, j, k)`

Return the coordinate for a given grid index.

Parameters

- **i** (*int*) – The grid index in the A direction.
- **j** (*int*) – The grid index in the B direction.
- **k** (*int*) – The grid index in the C direction.

Returns
The Cartesian coordinate of the given grid index.

Return type
PhysicalQuantity of type length.

`kPoint()`

Returns
The two-dimensional fractional k-point used in this transmission eigenstate calculation.

Return type
list of two floats

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`

Returns
The information.

Return type
dict

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`primitiveVectors()`

Returns
The primitive vectors of the grid.

Return type
PhysicalQuantity of type length.

`quantumNumber()`

Returns
The quantum number - the requested transmission state.

Return type
int

`scale(scale)`

Scale the field with a float.

Parameters
scale (*float*) – The parameter to scale with.

`setMetatext(metatext)`

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

shape()

Returns
The number of grid points in each direction.

Return type
tuple of three int.

spin()

Returns
The spin the transmission eigenstate is calculated for.

Return type
Spin.Up | Spin.Down | Spin.All

spinProjection(*spin=None*)

Construct a new *GridValues* object with the values of this object projected on a given spin component.

Parameters
spin (Spin.All | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.Up | Spin.Down | Spin.RealUpDown | Spin.ImagUpDown) – The spin component to project on. **Default:** The spin the object was created with. If the spin was Spin.All, Spin.Sum will be used for the projection.

Returns
A new *GridValues* object for the specified spin.

Return type
GridValues

toArray()

Returns
The values of the grid as a numpy array slicing off any units.

Return type
numpy.array

uniqueString()

Return a unique string representing the state of the object.

unit()

Returns
The unit of the data in the grid.

Return type
A physical unit.

unitCell()

Returns
The unit cell of the grid.

Return type
PhysicalQuantity of type length.

`volumeElement()`

Returns
The volume element of the grid represented by three vectors.

Return type
PhysicalQuantity of type length.

Usage Examples

Calculate the *TransmissionEigenstate* for a *DeviceConfiguration* and save the results for visualization with QuantumATK:

```
eigenstates = TransmissionEigenstate(device_configuration, 0.0*eV, quantum_number=0)
nlsave('eigenstate.nc',eigenstates)
```

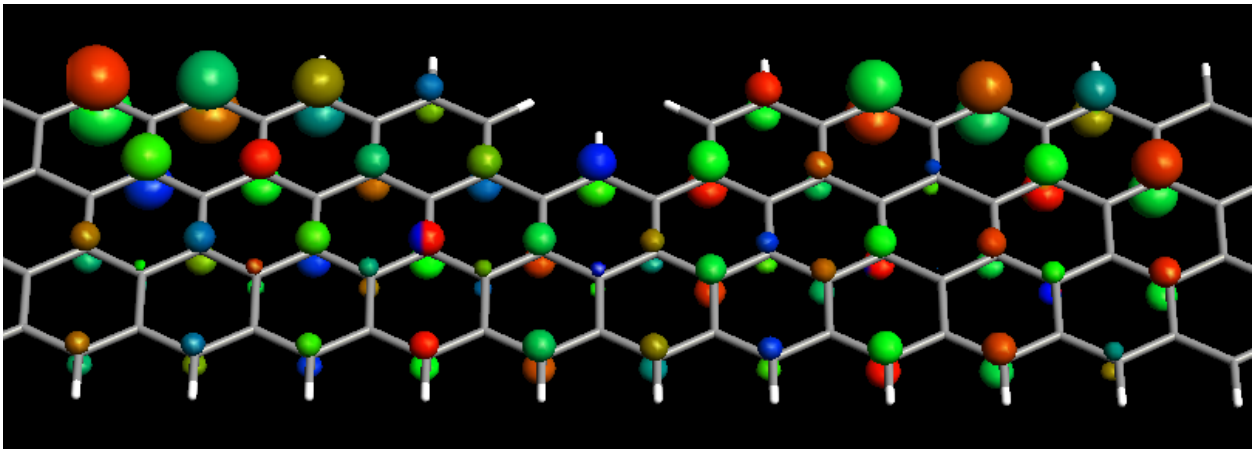


Fig. 4.39: Spin up component of the transmission eigenstates of a graphene ribbon with a wedge. The transmission eigenstate is a complex 3D wave function and is illustrated by an isosurface with isovalue given by the magnitude of the eigenstate and the color maps the phase of the eigenstate.

For examples on working with 3D grids, see *HartreePotential* and *ElectronDensity*.

Notes

The *TransmissionEigenstate* is an analysis option which finds the eigenstates of the transmission matrix. The transmission matrix is given by

$$T_{nm} = \sum_k t_{nk} t^{\dagger}_{km},$$

where t_{nk} is the transmission amplitude from Bloch state ψ_n in the left electrode to Bloch state ψ_k in the right electrode. The transmission coefficient is given by the trace of the transmission matrix,

$$T = \sum_n T_{nn}.$$

The transmission eigenstates are obtained by propagating the linear combination of the Bloch states, $\sum_n e_{\alpha,n}\psi_n$, where $e_{\alpha,n}$ diagonalize the transmission matrix,

$$\sum_m T_{nm} e_{\alpha,m} = \lambda_\alpha e_{\alpha,n},$$

with transmission eigenvalue λ_α.

See also, *TransmissionEigenvalues* and *LocalDeviceDensityOfStates*.

TransmissionEigenvalues

class TransmissionEigenvalues(*configuration=None*, *energy=None*, *k_point=None*, *energy_zero_parameter=None*)

Class for representing the transmission eigenvalues for a given configuration and calculator.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which the transmission eigenvalues should be calculated.
- **energy** (*PhysicalQuantity* of type energy) – The energy for which the transmission eigenvalues should be calculated. **Default:** `0.0*eV`
- **k_point** (*list of two floats.*) – The 2-dimensional k-point in fractional coordinates for which the transmission eigenvalues should be calculated (x,y), e.g. [0.8, 0.2]. **Default:** `[0.0, 0.0]`
- **energy_zero_parameter** (`AverageFermiLevel | AbsoluteEnergy.`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`

electrodeFermiLevels()

Returns
The left and right electrodes Fermi levels in absolute energies.

Return type
PhysicalQuantity

energy()

Returns
The energy used in this transmission eigenvalue calculation.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero value.

Return type
PhysicalQuantity of type energy

energyZeroParameter()

Returns
The specified choice for the energy zero.

Return type
AverageFermiLevel | AbsoluteEnergy

evaluate(*spin=None*)

Obtain the calculated transmission eigenvalues.

Parameters
spin (`Spin.Up | Spin.Down`) – The spin component to evaluate. **Default:** `Spin.Up`

Returns
The requested transmission eigenvalues.

Return type
numpy.array

kPoint()

Returns
The two-dimensional fractional k-point used in this transmission eigenvalue calculation.

Return type
list of two floats

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *TransmissionEigenvalues* for a *DeviceConfiguration* and print out the result:

```
eigenvalues = TransmissionEigenvalues(device_configuration,0.0*eV )
nlprint(eigenvalues)
```

Notes

The *TransmissionEigenvalues* is an analysis option which finds the eigenvalues of the transmission matrix. The transmission matrix, for a given energy, E, and k-point, $\mathbf{k}$, is given by

$$T_{nm}(E,\mathbf{k}) = \sum_{\ell} t_{n\ell}(E,\mathbf{k}) t^{\dagger}_{\ell m}(E,\mathbf{k}),$$

where $t_{n\ell}(\mathbf{k})$ is the transmission amplitude from Bloch state $\psi_n(\mathbf{k})$ in the left electrode to Bloch state $\psi_\ell(\mathbf{k})$ in the right electrode.

The transmission coefficient is given by the trace of the transmission matrix,

$$T(E, \mathbf{k}) = \sum_n T_{nn}(E, \mathbf{k}).$$

Below, we will suppress the indices E and $\mathbf{k}$ in most cases, but keep in mind that all quantities depend on these quantum numbers parametrically.

The transmission eigenvalues λ_α are the eigenvalues of the transmission matrix T_{nm}.

It follows from the invariance of the trace of a matrix, that the transmission eigenvalues sum up to the transmission coefficient,

$$T = \sum_\alpha \lambda_\alpha.$$

The transmission eigenvalues are, in QuantumATK, in the range [0,1] for each spin channel.

See also, *TransmissionEigenstate*.

TransmissionPathways

class TransmissionPathways(*configuration=None*, *energy=None*, *kpoints=None*, *contributions=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*)

Class for representing the Transmission Pathways for a given device configuration and calculator.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which the transmission pathways should be calculated.
- **energy** (*PhysicalQuantity* of type energy) – The energy for which the transmission pathway should be calculated. **Default:** `0.0*eV`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which the transmission pathways should be calculated. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.
- **contributions** (`Left` | `Right`) – The transmission pathways contributions from the left or right electrode. **Default:** `Left`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to be used for the transmission pathways. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`.) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the transmission calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`

electrodeFermiLevels()

Returns
The left and right electrodes Fermi levels in absolute energies.

Return type
PhysicalQuantity

energy()

Returns
The energy used in this transmission pathways.

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero value.

Return type
PhysicalQuantity of type energy

evaluate(*spin=None*)

Return the atom transmission pathways.

Parameters
spin (Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown) – The spin used in the analysis **Default:** Spin.Sum

Returns
A list with transmission pathway data for the give spin type. Each entry in the list contains the data (atom index i, neighbour atom index j, repetitions, t_ij), where repetitions=(n_A, n_B, n_C) is the repetitions of the neighbour atom in directions A, B, C, and t_ij is the local bond transmission from i to j.

Return type
list

infinitesimal()

Returns
The infinitesimal used for calculating the transmission pathways.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *TransmissionPathways* for a device configuration, print out the result, and save them for visualization with QuantumATK:

```
pathways = TransmissionPathways(device_configuration)
nlprint(pathways)
nlsave('pathways.nc',pathways)
```

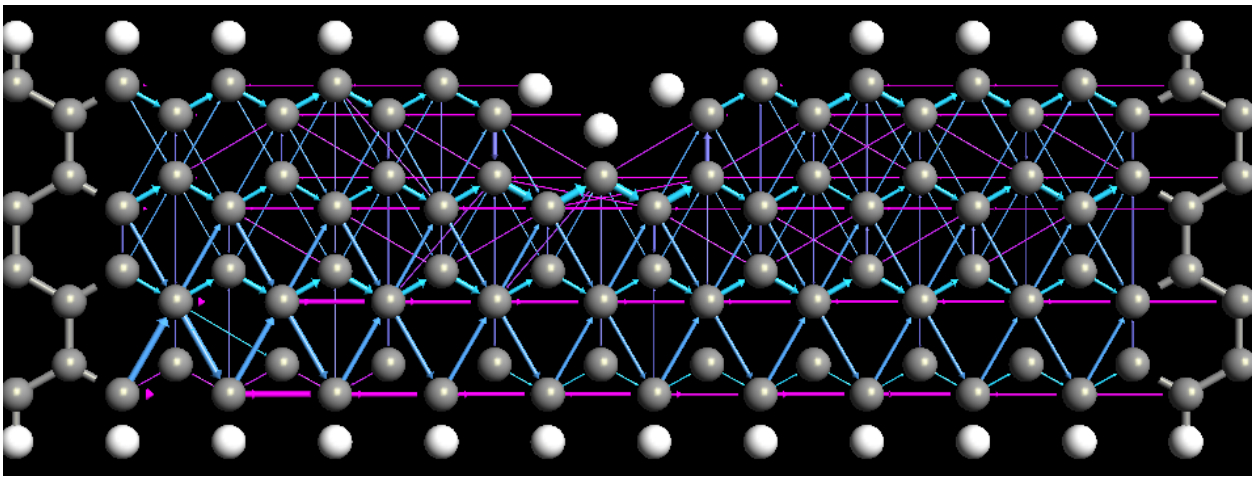


Fig. 4.40: Spin up component of the transmission pathways of a graphene ribbon with a wedge. Arrows point in the direction of the pathways.The magnitude of the pathway is illustrated by the volume of the arrow and the color shows the direction, i.e. right point arrows are blue and left pointing arrows violet.

Check, that the pathways sum up to the correct transmission spectrum:

```
pathways = TransmissionPathways(device_configuration)
border_atom = len(device_configuration.elements())/2
print 'transmission coefficient ',
print pathways._transmissionCoefficient(border_atom=border_atom)
```

Notes

The *TransmissionPathways* is an analysis option which splits the transmission coefficient into local bond contributions, T_{ij}. The pathways have the property that if the system is divided into 2 parts (A, B), then the pathways across the boundary between A and B sum up to the total transmission coefficient

$$T(E) = \sum_{i \in A, j \in B} T_{ij}(E).$$

The local bond contributions, T_{ij} , can be both positive and negative. A negative value correspond to that the electron is back scattered along the bond. A positive value of T_{ij} is visualized as an arrow from i to j while a negative value is visualized as an arrow from j to i, i.e. the direction of the arrows shows the flow of the electrons.

For further details, see[1].

[1] G.ăC. Solomon, C.ăHerrmann, T.ăHansen, V.ăMujica, and M.ăA. Ratner. Exploring local currents in molecular junctions. *Nat. Chem.*, 2(3):223–228, mar 2010. doi:10.1038/nchem.546.

TransmissionSpectrum

class TransmissionSpectrum(*configuration*, *energies=None*, *kpoints=None*, *kpoints_weights=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*, *device_transmission_method=None*, *enforce_zero_in_band_gap=None*)

Constructor for the TransmissionSpectrum object.

Parameters

- **configuration** (*DeviceConfiguration* | *BulkConfiguration*) – The configuration with an attached calculator for which to calculate the transmission spectrum. A BulkConfiguration must be constructed as an electrode, i.e. its unit cell vectors A and B are orthogonal to C, and C is in the z-direction.
- **energies** (*PhysicalQuantity* of type energy.) – The energies for which to calculate the transmission spectrum. **Default:** Energy range that covers the bias window, plus $30k_BT$ for both the upper and lower limits.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | *AdaptiveGrid* | sequence of sequence (size 3) of float) – The k-points for which to calculate the transmission spectrum. This can either be given as a k-point grid or a list of fractional k-points, e.g., `[[0.0, 0.0, 0.0], [0.0, 0.0, 0.1], ...]`. Note that the k-points must be in the xy-plane. **Default:** `MonkhorstPackGrid(na, nb)` where `(na, nb)` is the sampling used for the self-consistent calculation.
- **kpoints_weights** (*sequence of float*) – The weight of each k-point. **Default:** The weights corresponding to the MonkhorstPackGrid, or a list of [1.0, 1.0,] if k-points are specified as floats.
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The SelfEnergyCalculator to be used for the transmission spectrum. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel` | `AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy.) – Small positive energy, used to move the transmission calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6*eV`
- **device_transmission_method** (*GreensFunction* | *SparseGreensFunction*.) – The method employed in a device transmission spectrum calculation. This parameter is only used if `configuration` is a *DeviceConfiguration*. Note that *GreensFunction* in *TransmissionSpectrum* is only supported for a maximum of 2 (1) `processes_per_contour_point` for Unpolarized, Polarized (Noncollinear, SpinOrbit) calculations. **Default:** `GreensFunction(processes_per_contour_point=1)`.
- **enforce_zero_in_band_gap** (*bool*) – Flag which specifies whether the transmission values are enforced to zero inside the band gap. This parameter is only used if `configuration` is a *DeviceConfiguration*. **Default:** True

adaptive()

Returns
True if adaptive grids are used, otherwise False.

Return type
bool

adaptiveGrid(*spin_index*, *energy_index*, *tolerance=None*, *error_measure=None*, *maximum_number_of_levels=None*)

Method for construction of a new *AdaptiveGrid*, which can be used for restarting from a previous step.

Parameters

- **spin_index** (*Non-negative int.*) – The spin index for which to extract the transmission values, k-points and triangles for the new adaptive grid. **Default:** 0

•**energy_index** (*Non-negative int.*) – The energy index for which to extract the transmission values, k-points and triangles for the new adaptive grid. **Default:** 0

•**tolerance** (*Positive float.*) – Tolerance for adaptive grid integration. **Default:** 1e-6

•**error_measure** (`Absolute` | `Relative`) – Method to calculate error in adaptive grid integration. Either Absolute error) or Relative error. **Default:** `Absolute`

•**maximum_number_of_levels** (*Non-negative int.*) – Maximum number of refinement steps in the adaptive grid. **Default:** 10

Returns
The adaptive grid object constructed with the transmission values, k-points and triangles corresponding to the spin and energy index.

Return type
AdaptiveGrid

bias(*positive_current_convention=None*)

Parameters
positive_current_convention (`LeftToRight` | `RightToLeft`) – The convention for the direction of the positive current. If `LeftToRight` the bias is defined as `V(left) - V(right)`, whereas for `RightToLeft` the bias is defined as `V(right) - V(left)`. **Default:** `LeftToRight`

Returns
The applied bias.

Return type
PhysicalQuantity of type voltage.

conductance(*electrode_voltages=None*, *electrode_temperatures=None*, *coupling_rates=None*, *spin=None*)

Calculate the differential conductance.

Parameters

•**electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the current calculation. If no voltage is specified it will use the voltages in the self-consistent calculation. If voltages are specified the current will be obtained non-self-consistent. **Default:** The electrode voltages from the calculator.

•**electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.

•**coupling_rates** (*list of two floats*) – Two fractions `(alpha_L, alpha_R)` with `alpha_L + alpha_R = 1`, which specifies the fraction of the differential voltage, `alpha_L*dV`, added to the left electrode voltage, and `alpha_R*dV`, added to the right electrode voltage. **Default:** `[0.5, 0.5]`

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z` | `Spin.UpUp` | `Spin.DownDown` | `Spin.RealUpDown` | `Spin.ImagUpDown` | `Spin.All`) – The spin of the current. **Default:** `Spin.Sum`

Returns
The differential conductance for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity of type conductance.

current(*electrode_voltages=None*, *electrode_temperatures=None*, *spin=None*, *positive_current_convention=None*)

Calculate the current with positive direction from left to right.

Parameters

- **electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the current calculation. If no voltage is specified it will use the voltages in the self-consistent calculation. If voltages are specified the current will be obtained non-self-consistent. **Default:** The electrode voltages from the calculator.
- **electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.
- **spin** (`Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown | Spin.All`) – The spin of the current. **Default:** `Spin.Sum`
- **positive_current_convention** (`LeftToRight | RightToLeft`) – The convention for the direction of the positive current. **Default:** `LeftToRight`

Returns
The current for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity of type electric current.

electrodeFermiLevels()

Returns
The Fermi levels of the left and right electrodes in absolute energies.

Return type
PhysicalQuantity of type energy.

electrodeFermiTemperatures()

Returns
The Fermi temperature of the left and right electrodes used in this transmission spectrum.

Return type
PhysicalQuantity of type temperature.

electrodeVoltages()

Returns
The left and right electrode voltages used in this transmission spectrum.

Return type
PhysicalQuantity of type voltage.

energies()

Returns
The energies used in this transmission spectrum

Return type
PhysicalQuantity of type energy

energyZero()

Returns
The energy zero used for the energy scale in this transmission spectrum.

Return type
PhysicalQuantity of type energy.

energyZeroParameter()

Returns
The specified choice for the energy zero.

Return type
AverageFermiLevel | AbsoluteEnergy

enforceZeroInBandGap()

Returns
Whether the transmission values are enforced to zero inside the band gap.

Return type
bool

evaluate(*kpoints=None*, *spin=None*)

Evaluates the k-point averaged transmission spectrum.

Parameters

- **kpoints** (*MonkhorstPackGrid* | *RegularKpointGrid* | list(n) of list(3) of float.) – The set of k-points to average over. For calculations with *AdaptiveGrid* this parameter cannot be set and the average is over all k-points. **Default:** Average over all k-points.
- **spin** (Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown | Spin.All) – The spin component to query result for. **Default:** Spin.Sum

Returns
The k-point averaged transmission spectrum for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
list(4) of array(n_energies) | array(n_energies)

infinitesimal()

Returns
The infinitesimal used for calculating the transmission.

Return type
PhysicalQuantity of type energy

kpoints()

Returns
The k-points used in this transmission spectrum.

Return type
array(n_kpoints, 3) of float | dict(n_spins, n_energies)

kpointsWeights()

Returns
The weights of the k-points used in this transmission spectrum.

Return type
list(n_kpoints) of float | None

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Returns
The transmissions spectrum information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

peltierCoefficient(*temperature=None*, *fermi_level_shifts=None*, *spin=None*)

Calculate the Peltier coefficient.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature.) – The average temperature of the left and right electrodes. **Default:** The temperatures from the calculator.
- **fermi_level_shifts** (*PhysicalQuantity* of type energy.) – Non-self-consistent shifts of the Fermi level. Can be a list of energies. **Default:** 0*eV.
- **spin** (Spin.Up | Spin.Down | Spin.Sum) – The spin of the conductance. **Default:** Spin.Sum

Returns
The Peltier coefficient for the given spin type at each Fermi level shift.

Return type
PhysicalQuantity of type Volt.

seebeckCoefficient(*temperature=None*, *fermi_level_shifts=None*, *spin=None*)

Calculate the Seebeck coefficient.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature.) – The average temperature of the left and right electrodes. **Default:** The temperatures from the calculator.
- **fermi_level_shifts** (*PhysicalQuantity* of type energy.) – Non-self-consistent shifts of the Fermi level. Can be a list of energies. **Default:** 0*eV.
- **spin** (Spin.Up | Spin.Down | Spin.Sum) – The spin of the conductance. **Default:** Spin.Sum

Returns
The Seebeck coefficient for the given spin type at each Fermi level shift.

Return type
PhysicalQuantity of type Volt/Kelvin.

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

spectralConductivity(*electrode_voltages=None*, *electrode_temperatures=None*, *coupling_rates=None*, *spin=None*, *positive_current_convention=None*)

Calculate the spectral conductivity.

Parameters

- **electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the calculation. **Default:** The electrode voltages from the calculator.
- **electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The left and right electrode temperatures to be used in the spectral current calculation. **Default:** The temperatures from the calculator.
- **coupling_rates** (*list of two non-negative float.*) – Two fractions (alpha_L, alpha_R) with alpha_L + alpha_R = 1, which specifies the fraction of the differential voltage, alpha_L*dV, added to the left electrode voltage, and alpha_R*dV, added to the right electrode voltage. **Default:** [0.5, 0.5]
- **spin** (Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown | Spin.All) – The spin of the current. **Default:** Spin.Sum

Returns
The spectral conductivity for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity with the unit e**2/hplanck | list of four *PhysicalQuantity* with the unit e**2/hplanck

spectralCurrent(*electrode_voltages=None*, *electrode_temperatures=None*, *spin=None*, *positive_current_convention=None*)

Calculate the spectral current.

Parameters

- **electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the current calculation. If no voltage is specified it will use the voltages in the self-consistent calculation. If voltages are specified the current will be obtained non-self-consistent. **Default:** The electrode voltages from the calculator.
- **electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The electrode temperatures to be used in the current calculation. **Default:** The temperatures from the calculator.
- **spin** (Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown | Spin.All) – The spin of the current. **Default:** Spin.Sum
- **positive_current_convention** (LeftToRight | RightToLeft) – The convention for the direction of the positive current. **Default:** LeftToRight

Returns
The spectral current for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity with the unit e/hplanck | list of *PhysicalQuantity* with the unit e/hplanck

spectralThermalCurrent(*electrode_voltages=None*, *electrode_temperatures=None*, *spin=None*, *positive_current_convention=None*)

Calculate the spectral Thermal current.

Parameters

- **electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the calculation. **Default:** The electrode voltages from the calculator.
- **electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The left and right electrode temperatures to be used in the spectral current calculation. **Default:** The temperatures from the calculator.
- **spin** (Spin.Up | Spin.Down | Spin.Sum | Spin.X | Spin.Y | Spin.Z | Spin.UpUp | Spin.DownDown | Spin.RealUpDown | Spin.ImagUpDown | Spin.All) – The spin of the current. **Default:** Spin.Sum

•**positive_current_convention** (`LeftToRight` | `RightToLeft`) – The convention for the direction of the current. When `RightToLeft` the thermal current will from the Right electrode. **Default:** `LeftToRight`

Returns
The spectral current for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity with the unit e/hplanck | list of *PhysicalQuantity* with the unit e/hplanck

spins()

Returns
The spins used in this transmission spectrum.

Return type
tuple(n_spins) of spin component flags.

thermalConductance(*temperature=None*, *fermi_level_shifts=None*, *spin=None*)

Calculate the thermal conductance.

Parameters

•**temperature** (*PhysicalQuantity* of type temperature.) – The average temperature of the left and right electrodes. **Default:** The temperatures from the calculator.

•**fermi_level_shifts** (*PhysicalQuantity* of type energy.) – Non-self-consistent shifts of the Fermi level. Can be a list of energies. **Default:** 0*eV.

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin of the conductance. **Default:** `Spin.Sum`

Returns
The thermal conductance for the given spin type at each Fermi level shift.

Return type
PhysicalQuantity of type Watt/Kelvin.

thermalCurrent(*electrode_voltages=None*, *electrode_temperatures=None*, *spin=None*, *positive_current_convention=None*)

Calculate the Thermal current.

Parameters

•**electrode_voltages** (*PhysicalQuantity* of type voltage.) – The left and right electrode voltages to be used for defining the bias window for the calculation. **Default:** The electrode voltages from the calculator.

•**electrode_temperatures** (*PhysicalQuantity* of type temperature.) – The left and right electrode temperatures to be used in the spectral current calculation. **Default:** The temperatures from the calculator.

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z` | `Spin.UpUp` | `Spin.DownDown` | `Spin.RealUpDown` | `Spin.ImagUpDown` | `Spin.All`) – The spin of the current. **Default:** `Spin.Sum`

•**positive_current_convention** (`LeftToRight` | `RightToLeft`) – The convention for the direction of the positive current. **Default:** `RightToLeft`

Returns
The spectral current for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
PhysicalQuantity with the unit e/hplanck | list of *PhysicalQuantity* with the unit e/hplanck

thermoelectricZT(*temperature=None*, *fermi_level_shifts=None*, *spin=None*, *phonon_thermal_conductance=None*)

Calculate the thermoelectric figure of merit, ZT.

Parameters

- **temperature** (*PhysicalQuantity* of type temperature.) – The average temperature of the left and right electrodes. **Default:** The temperatures from the calculator.
- **fermi_level_shifts** (*PhysicalQuantity* of type energy.) – Non-self-consistent shifts of the Fermi level. Can be a list of energies. **Default:** 0*eV.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin of the conductance. **Default:** `Spin.Sum`
- **phonon_thermal_conductance** (*PhysicalQuantity* of type Watt/Kelvin.) – Optional phonon contribution to the thermal conductance. **Default:** 0.0 * Watt/Kelvin

Returns
The ZT for the given spin type at each Fermi level shift.

Return type
float or numpy.array

transmission(*spin=None*)

Determine the transmission coefficients of the transmission spectrum.

Parameters
spin (`Spin.Up` | `Spin.Down` | `Spin.Sum` | `Spin.X` | `Spin.Y` | `Spin.Z` | `Spin.UpUp` | `Spin.DownDown` | `Spin.RealUpDown` | `Spin.ImagUpDown` | `Spin.All`) – The spin component to get the transmission coefficients for. **Default:** `Spin.All`

Returns
The transmission coefficients for each energy and k-point for the given spin type. For Spin.All a list of (Spin.Sum, Spin.X, Spin.Y, Spin.Z) will be returned.

Return type
list(4) of array(n_energies, n_kpoints) | array(n_energies, n_kpoints) | list(4) of dict(n_energies, n_kpoints) | dict(n_energies, n_kpoints)

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *TransmissionSpectrum* and for each energy print out all k-dependent coefficients:

```
# Define A,B directions of lattice
vector_a = [5.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.0, 0.0]*Angstrom

# Setup a device configuration
electrode = BulkConfiguration(
    bravais_lattice=UnitCell(vector_a, vector_b,
                             [0.0, 0.0, 9.0]*Angstrom),
    elements=[Lithium, Lithium, Lithium],
    cartesian_coordinates=[[ 2.5,  2.5,  1.5],
                           [ 2.5,  2.5,  4.5],
                           [ 2.5,  2.5,  7.5]]*Angstrom
    )
central_region = BulkConfiguration(
    bravais_lattice= UnitCell(vector_a, vector_b,
```

(continues on next page)

(continued from previous page)

```
                              [0.0, 0.0, 22.0]*Angstrom),
    elements=[Lithium, Lithium, Lithium, Hydrogen, Hydrogen,
              Lithium, Lithium,  Lithium],
    cartesian_coordinates=[[  2.5,   2.5,   1.5],
                           [  2.5,   2.5,   4.5],
                           [  2.5,   2.5,   7.5],
                           [  2.5,   2.5,  10.5],
                           [  2.5,   2.5,  11.5],
                           [  2.5,   2.5,  14.5],
                           [  2.5,   2.5,  17.5],
                           [  2.5,   2.5,  20.5]]*Angstrom
     )
device_configuration = DeviceConfiguration(
    central_region,
    [electrode, electrode]
    )

# Setup calculators
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(1, 1, 100))

electrode_calculator = HuckelCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters)

calculator = DeviceHuckelCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    electrode_calculators=[electrode_calculator, electrode_calculator])

device_configuration.setCalculator(calculator)

# Calculate the transmission coefficient
transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(-0.5,0.5,10)*eV,
    kpoints=MonkhorstPackGrid(3,3,1),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*Units.eV
    )

# Print out all k-dependent transmission coefficients
data = transmission_spectrum.transmission()
energies = transmission_spectrum.energies()
kpoints = transmission_spectrum.kpoints()
spins = ["Up","Down"]
#print data
for s in range(len(spins)):
    print("Transmission coefficient of spin component ", spins[s])
    for i in range(data[s].shape[0]):
        print('Transmission at energy = %12.6f eV' % (energies[i].inUnitsOf(eV)))
        print('      kx         ky      transmission ')
        for j in range(data[s].shape[1]):
            print('%10.4f %10.4f %16.6e' % \
```

(continues on next page)

(continued from previous page)

```
                (kpoints[j][0],kpoints[j][1],data[s][i][j]))
        print()
```

li_h2_li_trans.py

Alternatively, using the *MonkhorstPackGrid* class, we can also specify the k-points and weights directly. The k-points are specified in units of the reciprocal lattice vectors. Note, that except for the Gamma point, the k-points have weight 2 due to use of time-reversal symmetry, i.e $T(k) = T(-k)$.

```
transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(-0.5,0.5,10)*eV,
    kpoints=[[0.0,0.0,0.0],[0.0,1./3.,0.0],[1./3.,-1./3.,0.0],
    [1./3.,0.0,0.0],[1./3.,1./3.,0.0]],
    kpoints_weights = [1.,2.,2.,2.,2.],
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*Units.eV
    )
```

li_h2_li_trans1.py

Finally, the *TransmissionSpectrum* can be calculated using an *AdaptiveGrid* of k-points which is automatically refined in the regions of k-space where a fine k-point sampling is needed:

```
#----------------------------------------
# Adaptive Grid.
#----------------------------------------

adaptive_grid = AdaptiveGrid(
        kA_range=[-0.5, 0.5],
        kB_range=[-0.5, 0.5],
        tolerance=1e-2,
        error_measure=Relative,
        number_of_initial_levels=2,
        maximum_number_of_levels=7)

#----------------------------------------
# Transmission Spectrum.
#----------------------------------------

transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(-0.5,0.5,10)*eV,
    kpoints=adaptive_grid,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*Units.eV
    )
```

trans_adaptive.py

The following lines can be appended to the script to evaluate the current and differential conductance for different electron temperatures in the electrodes:

```
temp = numpy.linspace(0,1000,21)
currents = []
conductances = []

for t in temp:
    current = transmission_spectrum.current(
                  electrode_temperatures=[200.,t]*Kelvin
                  )
    currents.append(current)

    conductance = transmission_spectrum.conductance(
                        electrode_temperatures=[200.,t]*Kelvin
                        )
    conductances.append(conductance)

print("Temperature[Left]   Temperature[Right]"\
      + "     Current     Differential-Conductance ")
output_format = "      %6.0f K          %6.0f K     %s  %s"
for i in range(len(temp)):
    print(output_format%(200., temp[i], currents[i], conductances[i]))
```

current.py

A *TransmissionSpectrum* can also be calculated for a *BulkConfiguration*. This example show the calculation of the *TransmissionSpectrum* for a Bulk boron wire:

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [10.0, 0.0, 0.0]*Angstrom
vector_b = [0.0, 10.0, 0.0]*Angstrom
vector_c = [0.0, 0.0, 10.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Boron, Boron, Boron, Boron, Boron]

# Define coordinates
fractional_coordinates = [[ 0.5,  0.5,  0. ],
                          [ 0.5,  0.5,  0.2],
                          [ 0.5,  0.5,  0.4],
                          [ 0.5,  0.5,  0.6],
                          [ 0.5,  0.5,  0.8]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
calculator = HuckelCalculator()

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('boron_wire_transmission_spectrum.nc', bulk_configuration)

# ------------------------------------------------------------
# Transmission Spectrum
# ------------------------------------------------------------
transmission_spectrum = TransmissionSpectrum(
    configuration=bulk_configuration,
    energies=numpy.linspace(-2,2,101)*eV,
    kpoints=MonkhorstPackGrid(1,1),
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )
nlsave('boron_wire_transmission_spectrum.nc', transmission_spectrum)
nlprint(transmission_spectrum)
```

boron_wire_transmission_spectrum.py

Notes

To export the data of a *TransmissionSpectrum*, use the method *nlsave*.

Unit of transmission

A fully transmitting channel contributes 1 per spin.

Note: If you sum the spin channels, the total transmission of an ideal system is 2. Often, it is the energy dependent conductance which is reported in articles,

$$G(E) = \frac{e^2}{h} T(E).$$

If $G(E)$ is plotted in units of the conductance quantum $G_0 = 2\frac{e^2}{h}$ for an ideal system,

$$G(E) = 1G_0.$$

In QuantumATK we usually report the transmission coefficient per spin channel, even for unpolarized systems. Thus, the reported transmission coefficient per spin channel for an ideal system is 1.

See also, *TransmissionEigenvalues*.

Current

The current is calculated from the transmission coefficient using

$$I(V_L, V_R, T_L, T_R) = \frac{e}{h} \sum_{\sigma} \int T_{\sigma}(E) \left[f\left(\frac{E - \mu_R}{k_B T_R} \right) - f\left(\frac{E - \mu_L}{k_B T_L} \right) \right] dE,$$

where f is the Fermi function, $T_{L/R}$ is the electron temperatures of the left/right electrode, and $T_{\sigma}(E)$ is the transmission coefficient for the spin component σ.

The chemical potentials of the left electrode, $\mu_L = E_F^L - eV_L$, and the right electrode, $\mu_R = E_F^L - eV_R$, are defined relative to the Fermi level of the left electrode and related to the applied bias through

$$\mu_R - \mu_L = eV_{bias},$$

thus

$$V_{bias} = V_L - V_R.$$

Note: The Fermi levels of the left and right electrodes are not necessarily the same *a priori*, if for instance the electrodes consist of different materials or are doped differently. However, when the device calculation is set up, the right electrode Fermi level is shifted (and all energy eigenvalues along with it) such that the Fermi levels coincide at zero bias.

The current is independent of how the voltages are applied, it only depends on their difference (i.e. on V_{bias}).

A positive current flows when the applied voltage for the left electrode is higher than that of the right one, thus the positive current direction is left to right.

The *TransmissionSpectrum* itself is usually rather insensitive to the temperatures T_L and T_R used in the self-consistent calculation, but often depends strongly on the electrode voltages V_L and V_R. Thus, for an accurate estimate of the current, the *TransmissionSpectrum* should be calculated self-consistently for each desired bias. It is, however, possible to obtain an approximation for the current at a different bias than that used in the self-consistent calculation by giving a new set of voltages to the `current()` method. This will simply modify the integration range on the existing transmission spectrum. The temperature-dependence of the current can, on the other hand, in most cases be rather accurately estimated by only changing the electrode temperatures in the current calculation.

Differential conductance

The differential conductance is calculated from the transmission spectrum using:

$$\sigma(V_L, V_R, T_L, T_R, \alpha_L, \alpha_R) = \lim_{\delta V \to 0} \frac{I(V_L + \alpha_L \delta V, V_R - \alpha_R \delta V, T_L, T_R)}{\delta V}.$$

The coupling constants $\alpha_L + \alpha_R = 1.0$ models how the transmission spectrum couples with the left and right electrode. For a molecule strongly bound to the left electrode, we must have $\alpha_L = 0$ and $\alpha_R = 1$.

A more accurate estimate of the differential conductance which avoid introducing α_L and α_R is obtained by calculating the self-consistent current at a number of applied biases, $V_{bias}^1, V_{bias}^2, \ldots$, and performing numerical differentiation:

$$\sigma(V_{bias}, T_L, T_R) = \frac{I(V_{bias}^1, T_L, T_R) - I(V_{bias}^2, T_L, T_R))}{V_{bias}^1 - V_{bias}^2}.$$

Default automatic energy window

If no energies are provided, *TransmissionSpectrum* attempts to estimate an energy range such that the whole bias window is covered with an accurate enough spacing between the energy points for the integration.

The lower and upper bounds for the automatic energy window are defined as follows. First, we define an energy padding on each side of the bias window as $E_L^p = 30k_BT_L$ and $E_R^p = 30k_BT_R$, for the left and right electrodes, respectively. This is to ensure that the tails of the Fermi distributions are included in the energy window with a very tight accuracy. Then, we use the following formula to calculate the minimum and maximum energies of the energy window:

$$E_{\text{lower}} = \min(E_{F,L} - E_L^p, E_{F,R} - E_R^p) - E_0,$$
$$E_{\text{upper}} = \max(E_{F,L} + E_L^p, E_{F,R} + E_R^p) - E_0,$$

where E_0 denotes the energy zero as specified by the `energy_zero_parameter`, and $E_{F,L}$ and $E_{F,R}$ denote the Fermi levels of the left and right electrodes, respectively. The spacing between the energy points in the energy window is determined by the real-axis point density of the non-equilibrium contour (δE_{neq}) used for the calculation (see *Complex contour integration* for more details about the contour integration). However, we always ensure that there are at least three points per each k_BT value for the lower electrode temperature, i.e. the energy spacing can be written as

$$\delta E = \min(\delta E_{\text{neq}}, \frac{\min(k_BT_L, k_BT_R)}{3}).$$

As an example, for electrode temperatures $T_L = 300$ K, $T_R = 400$ K and a default *RealAxisContour* as the non-equilibrium contour, we obtain the (absolute) energy window [-1.084 eV, 0.984 eV] with 241 equally spaced energy points.

TremoloXCalculator

class TremoloXCalculator(*parameters*, *dynamical_matrix_parameters=None*)

Set up a force field calculator with the given potential set.

Parameters

- **parameters** (TremoloXPotentialSet | *list*) – A TremoloXPotentialSet object/list/tuple that contains potential and particle information. If a sequence of potential sets if given, the potential sets will be merged into a single potential set.
- **dynamical_matrix_parameters** (*Deprecated*) – Deprecated. This parameter is not used anymore.

clearPartialCharges()

Clear all stored partial charges.

dynamicalMatrixParameters()

This method is deprecated.

Returns
None always.

Return type
None

static getGPUDevices()

Returns
The available GPU devices on this node as a list of (device id, device description) tuples..

Return type
list

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

queryResult(*query*)

Get the result of a query.

Parameters
query (*TremoloXQuery*) – The query to get the result for.

Returns
The result of the query.

setAllGPUAcceleration(*enable*)

Enable or disable GPU acceleration for all supported potential types.

Parameters
enable (*bool*) – Enable or disable GPU acceleration for all supported potentials.

setDoInitialLoadBalancing(*do_initial_load_balancing*)

Method that sets to do initial load balancing

Parameters
do_initial_load_balancing (*bool*) – Flag to set to do initial load balancing

setGPUAcceleration(*pairPotentials=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *fft=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *mtp=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*)

Enable GPU acceleration for certain operations.

Parameters

- **pairPotentials** (*bool*) – Enable GPU acceleration for some selected pair potentials.
- **fft** (*bool*) – Enable GPU acceleration for fast fourier transforms (which are mainly used in SPME).
- **mtp** (*bool*) – Enable GPU acceleration for MTP potential.

setGPUDevice(*device*)

Set the GPU that should be used by this process.

Parameters
device (*int*) – The device index that should be used.

setGPUDevices(*indices*)

Set the GPUs that should be used by the processes on this computer/node. This call must be performed by all processes simultaneously!

Parameters
indices (*sequence of int or None.*) – List of device indices that should be used. If empty or None, all available devices are used.

setInternalOrdering(*ordering*)

Method that sets the internal ordering of the particles

Parameters
ordering (*TremoloXCalculator.DefaultOrdering | TremoloXCalculator.HilbertCurveOrdering |*

TremoloXCalculator.CellOrdering) – The ordering that will be used for all future calculations. Valid choices are default, hilbertcurve and cellorder

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

setPartialCharges(*charges*)

Stores a number of <particle ID / charge> pairs that will override the charges in the potential set for these particles. Please note that these stored charges will be applied to all configurations that are passed to this calculator. To clear the fixed charges, please call the clearPartialCharges function.

Parameters
charges (*dict*) – A dictionary that holds the <index / charge> pairs. The index is represented by an integer and is 0-based. The charge is expected to be a PhysicalQuantity of type charge.

setTabulationDelta(*delta*)

Method that sets the delta used to tabulate all tabulatable potentials (experimental status)

Parameters
delta (*PhysicalQuantity of type length*) – The delta used during tabulation. If delta <= 0, no tabulation is done.

setVerletListsDelta(*delta*)

Method that sets the delta used to create Verlet-Lists

Parameters
delta (*PhysicalQuantity of type length*) – The delta used for the creation. It has to be >= 0.

uniqueString()

Return a unique string representing the state of the object.

upgrade(*configuration*)

Method to upgrade the current calculator based on the given configuration.

Parameters
configuration (AtomicConfiguration sub-class) – The configuration the calculator will be used for.

Usage Examples

Set up a *TremoloXCalculator* and add a Tersoff potential for a silicon crystal.

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[0.0,  0.0,  0.0 ],
                          [0.25, 0.25, 0.25]]

# Set up configuration
```

(continues on next page)

(continued from previous page)

```
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = Tersoff_Si_1988()
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

The *TremoloXCalculator* is the main calculation engine in the **ATK-ForceField** module. It is being developed by the *Fraunhofer Institute for Algorithms and Scientific Computing* (SCAI). For more details see the TremoloX homepage.

The *TremoloXCalculator* provides a broad range of different classical potentials. It requires a *TremoloXPotentialSet* object, which defines the parameters and settings of the potential. This object can either be a pre- defined potential from the list of available potentials, or a user-defined combination of different potential functions, collected in a *TremoloXPotentialSet*.

GPU Acceleration

The TremoloXCalculator supports GPU acceleration for selected types of potentials. The default way to activate GPU acceleration is by checking the Enable proprietary GPU acceleration option in the **Job Manager** or by using the `atkpython_gpu` command instead of `atkpython`, as described in the *technical notes*. This will enable GPU acceleration for all parts of the potential set which support it.

The following potential classes support GPU acceleration:

- Long-range pair potentials:
  - *LennardJonesPotential*
  - *LennardJonesSplinePotential*
  - *LennardJonesSpline2Potential*
  - *LennardJonesMNPotential*
  - *LennardJonesShiftPotential*
  - *Repulsive12Potential*
  - *MorsePotential*
  - *TosiFumiPotential*
  - *BuckinghamPotential*
- All Coulomb solver types (real- and fourier-space parts can be accelerated).

•Machine-learned force fields:

–*MTPPotential*

–*QuantumATKMTPPotential*

–*TorchXPotential*

The same holds for pre-defined potential sets which contain one or more of these potential classes. GPU acceleration is meant to be used in simulations that run a series of energy, forces, and stress calculations, such as Molecular Dynamics, Geometry or NEB Optimization, Force Bias Monte Carlo, etc.

As primary application, *MolecularDynamics* provides some more specific support for GPU acceleration:

•In the header of the MD section it is reported if GPU acceleration is used for this simulation or not.

•Multi-GPU acceleration as described in the *technical notes* is supported for almost all of the above potential types. This can give additional speedup for large configurations. Note, that the configuration needs to have a minimal size, so that domain decomposition can be enabled. In particular for ML-FF foundation models based on the MACE framework this can give access to simulating larger configurations for longer times.

TremoloXPotentialSet

class TremoloXPotentialSet(*name='DefaultPotential'*)

Create an empty potential set.

Parameters
name (*str*) – The name of the potential set.

actOnlyBetweenTaggedRegions(*tag1*, *tag2*)

If this potential set consists only of twobody interactions, calling this function makes all twobody interactions act between particles of the two given, different tags. If this potential contains other interactions than twobody interactions, calling this function will raise a RuntimeError. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters

•**tag1** (*str*) – The first tag.

•**tag2** (*str*) – The second tag.

actOnlyOnTaggedRegion(*tag*)

Calling this function makes this potential set act only on particles that have the given tag. Particle types and potentials that are added to this potential set after this function call are not affected. The Coulomb interactions are also not affected.

Parameters
tag (*str*) – The tag on which this potential set should act.

addAnalyzer(*analyzer*)

Adds an analyzer to this potential set.

Parameters
analyzer (*TremoloBaseAnalyzer*) – The analyzer to should be added.

addOption(*option*)

Add an option to the potential.

Parameters
option (*TremoloOption*) – The option that will be added.

addParticleType(*particle*)

Add a particle type definition to the potential set.

Parameters
particle (ParticleType) – Instance of ParticleType.

addPotential(*potential*)

Add a potential to this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential that will be added.

findPotentials(*potentialType=None*, *particleTypes=None*)

Return a list of potentials in this potential set, that fullfil certain conditions.

Parameters

- **potentialType** (*TremoloBasePotential*) – A potential class (e.g. LennardJonesPotential).
- **particleTypes** (*str | ParticleType | ParticleIdentifier | sequence of type str/ ParticleType/ParticleIdentifier*) – A string or a sequence of strings that determine the particle types which the potential should act on.

Returns
The list of potentials that are of type potentialType and act on the particle types that are given by particleTypes.

Return type
list of type TremoloBasePotential

getCoulombSolver()

Returns
The current Coulomb potential or None.

Return type
CoulombSolver

getOptions()

Return a list of all options that are stored in this potential set.

getParticleType(*symbol*)

Return the particle definition with the given symbol

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

Returns
The ParticleType that belongs to the given identifier or None if no such particle was found.

Return type
ParticleType | None

getParticleTypes()

Return a list of all particle types that are stored in this potential set.

getPotentials()

Return a list of all potentials that are stored in this potential set.

getUsedTags()

Returns
The tags this potential set acts on.

Return type
set of type str

merge(*otherPotentialSet*)

Merges the current potential set with another potential set.

Parameters
otherPotentialSet (*PotentialSetBase*) – The potential set, the current potential should be merged with.

static mergePotentials(*potentials*)

Merges a list of potential sets and returns the merged potential set.

Parameters
potentials (*list of type PotentialSetBase*) – List of potential sets to be merged.

Returns
The potential set that was created by merging all potentials in the list.

Return type
PotentialSetBase

removeParticleType(*symbol*)

Remove a given particle type from the potential set.

Parameters
symbol (ParticleType, ParticleIdentifier *or str (deprecated)*) – The identifier of the particle type.

removePotential(*potential*)

Remove a potential from this potential set.

Parameters
potential (*TremoloBasePotential*) – The potential to be removed, this potential must have been retrieved by the findPotentials method.

setCoulombSolver(*coulomb*)

Sets the coulomb potential. Previously set Coulomb solvers will be overwritten.

Parameters
coulomb (CoulombSolver | *None*) – The new Coulomb solver. Must either be None or an object that was derived from the CoulombSolver class.

setPreferredUnitSystem(*unitSystem*)

Set the preferred unit system for this potential set. The choice here may be overridden if some potentials require a specific unit system.

Parameters
unitSystem (*TremoloUnitSystem*) – The unit system, e.g. (eVolt, kCalPerMole, SI).

setTags(*tags*)

Makes this potential set act only on particles with a given tag Note: This operation can not be reverted easily and will change the names of your particles. Use it only when you are finished building the potential set!

Parameters
tags (*str | list of type str*) – If tags is a string, the potential set acts only on particles with the given tag, if tag is list of strings of length 2, the potential acts only on particles with different tags.

supportedElements()

Return the chemical elements that are supported by this potential set. The elements are returned as a list that contains the element symbols.

Returns
The list of the symbols of the supported elements.

Return type
list of type str

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Define a potential for Quartz by adding particle types and interaction functions to the *TremoloXPotentialSet* .

```
# ------------------------------------------------------------
# Set up a SiO2 Quartz crystal
# ------------------------------------------------------------

# Set up lattice
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

# Define elements
elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.4697,  0.0000,  0.0000    ],
                          [ 0.0000,  0.4697,  0.66666667],
                          [ 0.5303,  0.5303,  0.33333333],
                          [ 0.4135,  0.2669,  0.1191    ],
                          [ 0.2669,  0.4135,  0.547567  ],
                          [ 0.7331,  0.1466,  0.785767  ],
                          [ 0.5865,  0.8534,  0.214233  ],
                          [ 0.8534,  0.5865,  0.452433  ],
                          [ 0.1466,  0.7331,  0.8809    ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

# Create the Pedone_2006Fe2 potential by hand, by adding the individual components

potentialSet = TremoloXPotentialSet(name='Pedone_2006Fe2')

# Add the particle types to the potential set
potentialSet.addParticleType(ParticleType(symbol='Si',
                                          mass=28.0855*atomic_mass_unit,
```

(continues on next page)

(continued from previous page)

```
                                          charge=2.4))
potentialSet.addParticleType(ParticleType(symbol='O',
                                          mass=15.9994*atomic_mass_unit,
                                          charge=-1.2))

# Add the pair potentials to the potential set
potentialSet.addPotential(MorsePotential('Si', 'O',
                                         r_0=2.1*Angstrom,
                                         k=2.0067*1/Ang,
                                         E_0=0.340554*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('Si', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=1.0*Ang**12*eV))
potentialSet.addPotential(MorsePotential('O', 'O',
                                         r_0=3.618701*Angstrom,
                                         k=1.379316*1/Ang,
                                         E_0=0.042395*eV,
                                         r_i=6.0*Angstrom,
                                         r_cut=7.5*Angstrom))
potentialSet.addPotential(Repulsive12Potential('O', 'O',
                                               r_cut=7.5*Angstrom,
                                               c=22.0*Ang**12*eV))

# Add the coulomb solver to the potential set
potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))

# Create the calculator from the potential set
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This class is used to combine a set of particles and interaction functions into a new potential. It can be used to define a new potential, based on the implemented potential classes, such as *LennardJonesPotential*, *MorsePotential*, or *TersoffSingleTypePotential*.

To display the composition of a predefined parameter set, which has been selected in the **Script Generator**, select *Show default* in the Global IO settings of the Script Generator, and send the script to the **Editor**. This allows you to edit individual parameter values.

TubeRegion

class TubeRegion(*value*, *start_point*, *end_point*, *inner_radius*, *thickness*)

Class for representing a hollow cylinder region of metallic or dielectric material.

Parameters

- **value** (float | *PhysicalQuantity* compatible with Volt) – The value that should be assigned to the tube. Either the voltage or the dielectric constant of the region.
- **start_point** (*PhysicalQuantity* of type length) – Vector to the first point defining the central axis of the tube.
- **end_point** (*PhysicalQuantity* of type length) – Vector to the second point defining the central axis of the tube.
- **inner_radius** (*PhysicalQuantity* of type length) – Inner radius of the tube.
- **thickness** (*PhysicalQuantity* of type length) – Thickness of the tube wall.

endPoint()

Returns
The end point of the tube.

Return type
PhysicalQuantity of type length

innerRadius()

Returns
The inner radius of the tube.

Return type
PhysicalQuantity of type length

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *header=True*)

Print a string containing an ASCII description of the SpatialRegion.

Parameters

- **stream** (*Stream based object*) – The io to write to.
- **header** (*bool*) – Option to emphasize the description as a header or not.

startPoint()

Returns
The start point of the tube.

Return type
PhysicalQuantity of type length

thickness()

Returns
The wall thickness of the tube.

Return type
PhysicalQuantity of type length

`toGLObject`(*color*)

Create the necessary object(s) to pass back into the GLConfiguration.

Parameters
color (*GLColor*) – The region type-specific color.

Returns
The inner and outer cylinders and the two end-caps.

Return type
list

`uniqueString`()

Return a unique string representing the state of the object.

`value`()

Function for asking for the value of the box region.

Returns
The value of the box region.

Return type
float | *PhysicalQuantity* compatible with Volt

Usage Examples

Define a dielectric region with dielectric constant $4\varepsilon_0$:

```
dielectric_region = TubeRegion(
    value=4.0,
    start_point=[1.0, 2.0, 3.0]*Angstrom, end_point=[4.0, 5.0, 6.0]*Angstrom,
    inner_radius=3.5*Angstrom,
    thickness=1.2*Angstrom
)
```

dielectric_tuberegion.py

Define two metallic regions where the second metallic region is constructed by cloning the first:

```
metallic_region1 = TubeRegion(
    value=1*Volt,
    start_point=[1.0, 2.0, 3.0]*Angstrom, end_point=[4.0, 5.0, 6.0]*Angstrom,
    inner_radius=3.5*Angstrom,
    thickness=1.2*Angstrom
)

metallic_region2 = metallic_region1(
    value=-1*Volt,
    inner_radius=1.1*Angstrom,
    thickness=0.5*Angstrom
)
```

metal_tuberegion.py

Notes

The following spatial regions are available:

- *BoxRegion*
- *SphereRegion*
- *TubeRegion*

TwoParticleCoulombInteractionFFTMethod

class TwoParticleCoulombInteractionFFTMethod(*grid_sampling=None*)

Calculate two-particle integrals by projecting single-particle orbitals on the real space grid and then evaluating the integral via Fast Fourier Transform as:

$\langle\psi_a\psi_b|V|\psi_c\psi_d\rangle = \frac{\Omega}{N^2}\sum_{\mathbf{G}} V(\mathbf{G})\mathrm{FFT}[\rho_{da}(\mathbf{r})]^*(\mathbf{G})\mathrm{FFT}[\rho_{bc}(\mathbf{r})](\mathbf{G})$

where Ω is the supercell volume.

Parameters

grid_sampling (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The grid sampling used to perform the FFT. For the best performance, use *OptimizedFFTGridSampling*. Note that the Coulomb integrals are sensitive to the grid sampling and might require a finer grid than the default values. **Default:** The grid sampling set on the calculator. If the

static eigenstatesRequireNormalization(*calculator*)

Check if the eigenstates require a normalization before calculating the two particle integrals.

TwoParticleCoulombInteractionPointChargeMethod

class TwoParticleCoulombInteractionPointChargeMethod

Calculate the two-particle integrals in orbital space as:

$\langle\psi_a\psi_b|V|\psi_c\psi_d\rangle = \sum_{\mathbf{R}_1,\mathbf{R}_2} V(\mathbf{R}_1-\mathbf{R}_2)\sum_{\alpha\in U(\mathbf{R}_1)} c^*_{a,\alpha}c_{d,\alpha}\sum_{\beta\in U(\mathbf{R}_2)} c^*_{b,\beta}c_{c,\beta}$

where c are the wavefunction coefficients in orbital space and $U(\mathbf{R})$ the orbitals centered on $\mathbf{R}$.

The divergent term $\mathbf{R}_1 = \mathbf{R}_2$ is approximated by averaging on a sphere. This method is computationally very efficient, but it is a somewhat crude approximation only suitable for large systems (several thousands of atoms).

UFFPotentialBuilder

class UFFPotentialBuilder(*use_bonds=None*, *include_bond_stretching=None*, *include_bond_angles=None*, *include_torsions=None*, *include_inversions=None*, *include_lennard_jones=None*, *include_electrostatic=None*, *lennard_jones_cutoff=None*, *lennard_jones_smoothing_length=None*, *electrostatic_pairwise_cutoff=None*, *electrostatic_smoothing_length=None*, *electrostatic_spme_cutoff=None*, *electrostatic_accuracy=None*, *electrostatic_scale=None*, *use_electrostatic_direct_sum=None*, *ignore_missing_types=None*)

This object builds a Universal potential using TremoloX.

Parameters

- **use_bonds** – Controls if any bonding (stretching, angle, torsion, inversion and hydrogen bond) terms are to be included. **Default:** True.

•**include_bond_stretching** (*bool*) – Controls if bonding stretching terms are included. **Default:** use_bonds.

•**include_bond_angles** (*bool*) – Controls if angle bending terms are included. **Default:** use_bonds.

•**include_torsions** (*bool*) – Controls if bond torsion terms are included. **Default:** use_bonds.

•**include_inversions** (*bool*) – Controls if bond inversion torsion terms are included. **Default:** use_bonds.

•**include_lennard_jones** (*bool*) – Controls if Lennard-Jones terms are included. **Default:** use_bonds.

•**include_electrostatic** (*bool*) – Controls if electrostatic terms are included. **Default:** use_bonds.

•**lennard_jones_cutoff** (*PhysicalQuantity* of type distance) – Distance beyond which the Lennard-Jones terms are truncated. **Default:** 10.0 * Angstrom.

•**lennard_jones_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the Lennard-Jones terms are brought to zero at the cutoff. **Default:** 2.0 * Angstrom.

•**electrostatic_pairwise_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in pairwise summation. **Default:** 12 * Angstrom

•**electrostatic_smoothing_length** (*PhysicalQuantity* of type distance) – Distance over which the electrostatic terms are brought to zero at the cutoff when using the direct electrostatic sum. **Default:** 2.0 * Angstrom

•**electrostatic_spme_cutoff** (*PhysicalQuantity* of type distance) – Distance at which the pairwise electrostatic terms are brought to zero in SPME summation. **Default:** 7.5 * Angstrom

•**electrostatic_accuracy** (*float*) – Sets the relative accuracy of the SPME summation. **Default:** 0.0001

•**electrostatic_scale** (*float*) – Set the relative strength of the electrostatic terms. **Default:** 1.0

•**use_electrostatic_direct_sum** (*bool*) – Use the direct Coulomb sum to calculate the electrostatic terms in MoleculeConfigurations. **Default:** False

•**ignore_missing_types** (*bool*) – Whether or not molecules missing Universal types are ignored. **Default:** False

assignAtomTypes(*configuration*, *overwrite=True*)

Function that gets the atom types based on the bonding pattern.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration of the system that is having types assigned. **Default:** True

•**overwrite** (*bool*) – Whether or not the existing Universal types are overwritten.

checkAtomTypes(*configuration*, *add_missing_terms=False*)

Function that tests that each atom has only one valid Universal type.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration whose types are being checked.

•**add_missing_terms** (*bool*) – Whether or not to add terms for missing UFF types. **Default:** False

createCalculator(*configuration*, *check_types=True*, *atomic_charges=None*, *charge_averaging=None*)

Create and return a calculator with the Universal potential.

Parameters

•**configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration the potential is to be used with.

•**check_types** (*bool*) – Flag as to whether or not the correctness of the types are checked. **Default:** True.

•**atomic_charges** (*PhysicalQuantity* of type charge) – Custom atomic charges to be used in the potential.

•**charge_averaging** (*NLFlag | None*) – The type of averaging of charges to be used, if any. Accepted values are None, AtomConnectivity to average over atoms bonded to the same elements and AtomTypes to average over atoms of the same forcefield type. **Default:** None.

Returns
The calculator containing the potential.

Return type
TremoloXCalculator

electrostaticAccuracy()

Returns
The relative accuracy of the electrostatic summation for SPME.

Return type
float

electrostaticPairwiseCutoff()

Returns
The cutoff used in pairwise electrostatic summation.

Return type
PhysicalQuantity (length)

electrostaticScale()

Returns
Scaling factor for the electrostatic terms.

Return type
float

electrostaticSmoothingLength()

Returns
Distance at which electrostatic terms are splined when using direct summation.

Return type
PhysicalQuantity (length)

electrostaticSpmeCutoff()

Returns
The cutoff used in SPME electrostatic summation.

Return type
PhysicalQuantity (length)

ignoreMissingTypes()

Returns
Whether or not missing types are ignored on molecules.

Return type
bool

includeBondAngles()

Returns
Whether or not bond angle terms are included in the potential set.

Return type
bool

includeBondStretching()

Returns
Whether or not bond stretching terms are included in the potential set.

Return type
bool

includeElectrostatic()

Returns
Whether or not electrostatic terms are included in the potential set.

Return type
bool

includeInversions()

Returns
Whether or not bond inversion terms are included in the potential set.

Return type
bool

includeLennardJones()

Returns
Whether or not Lennard Jones terms are included in the potential set.

Return type
bool

includeTorsions()

Returns
Whether or not bond torsion terms are included in the potential set.

Return type
bool

isUniversalType(*type_label*)
Function that tests if a type is in the potential

Parameters
type_label (*str*) – The type being tested.

Returns
Whether or not the tag follows the convention.

Return type
bool

lennardJonesCutoff()

Returns
Distance at which Lennard Jones terms are ignored.

Return type
PhysicalQuantity (length)

lennardJonesSmoothingLength()

Returns
Distance at which Lennard Jones terms are splined.

Return type
PhysicalQuantity (length)

nlinfo()

Returns
The nlinfo.

Return type
dict

uniqueString()

Return a unique string representing the state of the object.

universalType(*index*, *configuration*, *add_missing_terms=False*)

Gives the Universal type on an atom in a configuration, ignoring other tags.

Parameters

- **index** (*int*) – Index of the atom for which the type is sought.
- **configuration** (*AtomicConfiguration*) – The configuration containing the atom.
- **add_missing_terms** (*bool*) – Whether or not to add terms for missing UFF types. **Default:** False

Returns
The type of the atom.

Return type
str

useBonds()

Returns
Whether or not bond terms are included in the potential set.

Return type
bool

useElectrostaticDirectSum()

Returns
Whether or not direct Coulomb sums are used for molecules.

Return type
bool

Usage Examples

Investigate the diffusion of chloroform molecules in LTA zeolites using the UFF potential.

```
# -----------------------------------------------------------
# Bulk Configuration
# -----------------------------------------------------------
bulk_configuration = nlread('LTA_Chloroform.hdf5')[-1]

# -----------------------------------------------------------
# UFF Calculator
# -----------------------------------------------------------
potential_builder = UFFPotentialBuilder()
potential_builder.assignAtomTypes(bulk_configuration)
calculator = potential_builder.createCalculator(bulk_configuration)
bulk_configuration.setCalculator(calculator)

# -----------------------------------------------------------
# Molecular Dynamics
# -----------------------------------------------------------
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)

method = NVTNoseHoover(
    time_step=1*femtoSecond,
    reservoir_temperature=300*Kelvin,
    thermostat_timescale=100*femtoSecond,
    heating_rate=0*Kelvin/picoSecond,
    chain_length=3,
    initial_velocity=initial_velocity,
)

constraints = [FixCenterOfMass()]

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='LTA_Chloroform_Trajectory.hdf5',
    steps=100000,
    log_interval=500,
    method=method
)

bulk_configuration = md_trajectory.lastImage()
```

LTA_Chloroform.py LTA_Chloroform.hdf5

Notes

The UFFPotentialBuilder class enables the creation of a *TremoloXCalculator* that implements the UFF potential of Rappe, Casewit, Colwell, Goddard and Skiff[1]. This is a bonded valence forcefield that represents the potential energy of a molecule as a collection of simple functions based on bond lengths, bond angles, torsion angles, inversion angles and inter-atomic distances. One of the main advantages of the UFF potential is that it contains data for every atom in the periodic table, so that appropriate potentials can be approximated for any system. The overall mathematical form of the UFF potential can be given as:

$$E(\mathbf{x}) = \sum_{i}^{bonds} k_i(r_i - r_0)^2 + \sum_{i}^{angles} k_i \sum_{n}^{m} C_n \cos(n\theta_i) + \sum_{i}^{torsions} k_i(1 - \cos(n_i\phi_i - \delta_i)) + \sum_{i}^{inversions} k_i \sum_{n}^{m} C_n \cos(n\chi_i) + \sum_{ij}^{atoms} 4\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \sum_{ij}^{atoms} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Here r, θ, ϕ and χ represent inter-atomic and bond distances, angles, torsions and inversions respectively in the configuration.

Note: While the UFF potential contains data for atoms across the periodic table, in many cases specific types of atoms are assumed by the potential. This means that the generated potentials may not be suitable for specific cases. For instance transition metals are often parameterized based on complexed metal ions, and therefore the generated potential may not be suitable for simulating a pure metal or metal alloy. As in all simulations, care should be taken to ensure that the potential is an appropriate choice for the system at hand by reading the original paper.

To create a UFF calculator, an instance of the *UFFPotentialBuilder* must first be created. The constructor for this class has a number of options that controls the way in which the calculator is built. Individual types of interactions can be excluded by setting the `include_` options to `False`. By default all possible terms are included. All valence terms can also be excluded by setting the argument `use_bonds` to False. This is appropriate for configurations consisting of single atoms that are not bonded to each other.

As the *UFFPotentialBuilder* class uses the bonding in the configuration to determine appropriate potentials, bonds should be set on the configuration. This can be done with the `findBonds()` method in either the *MoleculeConfiguration* or *BulkConfiguration* class. This automatically assigns bonds between atoms that are within the covalent radii of each other. Specific bonds can also be assigned with the `setBonds()`method, which takes a list of the indices of pairs of atoms that are to be considered bonded.

There are then a number of options that control the non-bonding interactions. The parameter `lennard_jones_cutoff` sets the distance beyond which Lennard-Jones interactions are truncated. To avoid integration problems in molecular dynamics, Lennard-Jones interactions are brought to zero at the cutoff using a spline function. The width of this spline can be set using the `lennard_jones_smoothing_length` parameter.

Electrostatic interactions are a little more complicated. By default the *UFFPotentialBuilder* adds a *CoulombSPME* electrostatic solver for calculators to be used with a bulk configuration, and a *CoulombDSF* electrostatic solver to calculators to be used with a molecule configuration. When adding an *CoulombSPME* solver the cutoff used for calculating the real-space interactions can be set using the `electrostatic_spme_cutoff` argument. Likewise the accuracy of the smoothed particle mesh Ewald solver can be set using the `electrostatic_accuracy` argument. In the case where a *CoulombDSF* solver is added to the potential, the cutoff used can be set using the `electrostatic_pairwise_cutoff` argument. In cases where molecular electrostatics are to be compared with values calculated in a bulk configuration,

[1] A.ăK. Rappe, C.ăJ. Casewit, K.ăS. Colwell, W.ăA. GoddardăIII, and W.ăM. Skiff. UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations. *J. Am. Chem. Soc.*, 114(25):10024–10035, 1992. URL: https://pubs.acs.org/doi/abs/10.1021/ja00051a040, doi:10.1021/ja00051a040.

an explicit splined Coulomb potential can be used for molecules instead of the damped shifted force potential. In this case a *CoulombN2Spline* electrostatic solver can be added by using `use_electrostatic_direct_sum` argument. As with the damped shifted force potential the cutoff is controlled using the `electrostatic_pairwise_cutoff` argument, and like the Lennard-Jones potential is brought to zero with a spline whose length is defined by the `electrostatic_smoothing_length` argument. For all Coulomb solvers the total electrostatic energies and forces can be scaled up or down using the `electrostatic_scale` argument.

Once the *UFFPotentialBuilder* object is instantiated, a calculator is created using the `createCalculator` method. This takes a configuration and returns the appropriate UFF calculator for that configuration. To use electrostatic potentials in the calculator, atomic partial charges have to be assigned to each atom. As the UFF forcefield does not define specific charges, by default charges are calculated with the *QEqAtomicCharges* class. This implements the QEq charge equilibration method of Rappe and Goddard[2]. These charges are then added to the calculator. It is also possible to add pre-calculated charges from different sources using the `atomic_charges` argument. This argument takes a list of charges that have the same ordering as the atoms in the configuration. The partial atomic charges can also be averaged according to a few different schemes. This is controlled by the `charge_averaging` parameter, which takes an NLFlag which indicates the type of averaging. The flag `AtomConnectivity` averages charges on each atom based on the element and the other elements it is bonded to. The flag `AtomType` averages according to the assigned UFF type.

When creating a UFF calculator for a configuration, the configuration must have the appropriate tags for each atom type. Tags used by the UFF forcefield all begin with the prefix `UFF_`. The next two characters in the tag are for the atomic symbol, followed by a possible character to indicate the hybridization of the atom. In cases where there are atom types with the same geometry, further characters can indicate the oxidation state of the atoms. Tags can be assigned manually, or they can be assigned automatically. Automatic typing can be done with the `assignAtomTypes` method. This takes the configuration to have types added, as well as a possible flag to overwrite existing types. The atom types are determined by the element and in some cases the bonding of the atom. The validity of the existing types on a configuration can be checked by calling the `checkAtomTypes` method. This method raises a `NLValueError` if a problem is found with the tags on the given configuration. The UFF type of a specific atom can also be found using the `universalType` method.

Two additional types from the set originally specified in the UFF reference have been added to allow the implementation of some special cases. As amides in the UFF potential are treated as a special case, the type `UFF_C_R_a` has been added to specifically refer to resonant carbons in an amide group. A similar addition has been made to correctly model aromatic nitrogens. Aromatic nitrogens can be bonded to either two or three other atoms, as in the case of imidazole. The original `UFF_N_R` type for aromatic nitrogens has therefore been split into the two types `UFF_N_R2` and `UFF_N_R3` which represent aromatic nitrogens bonded to two and three other atoms respectively.

UpdateElectricFieldCorrection

class UpdateElectricFieldCorrection(*max_displacement=None*)

A class to control the update of Polarization and Born Effective Charges analysis during optimization and molecular dynamics with electric field. The update is controlled by the maximum displacement between two molecular dynamics or optimization steps.

Parameters
max_displacement (*PhysicalQuantity* of type length.) – The maximum displacement required to consider a configuration equivalent to a cached one. For the most accurate results (i.e., update at every change in configuration), select `0.0 * Ang`. For less frequent update, increase this parameter. **Default:** `0.0 * Ang`

maxDisplacement()

Returns
the maximum atomic displacement to consider two configuration different.

[2] AnthonyăK. Rappe and WillianăA. GoddardăIII. Charge equilibration for molecular dynamics simulations. *J. Phys. Chem.*, 95:3358, April 1991. URL: https://pubs.acs.org/doi/10.1021/j100161a070, doi:10.1021/j100161a070.

Return type
PhysicalQuantity of type length.

UserDefinedTabulatedPotential

class UserDefinedTabulatedPotential(*particleType1*, *particleType2*, *fileName*, *splineType=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **fileName** (*str*) – The name of the file that holds the spline data. It must be a text file that stores the spline in the following way for not hermitian:

```
numPoints distance
<energy at 0 * distance>
<energy at 1 * distance>
...
<energy at (numPoints-1) * distance>
```

and for hermitian:

```
numPoints distance dfactor
<energy at 0 * distance>  <denergy/dr at 0 * distance>
<energy at 1 * distance>  <denergy/dr at 1 * distance>
...
<energy at (numPoints-1) * distance>  <denergy/dr at (numPoints-1) * distance>
```

where dfactor is used as a prefactor for the denergy/dr values

- **splineType** – The spline interpolation that will be used. Must be one of the following variables: UserDefinedTabulatedPotential.akima, UserDefinedTabulatedPotential.bessel, UserDefinedTabulatedPotential.fivepoint, UserDefinedTabulatedPotential.hermitian, UserDefinedTabulatedPotential.natural.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Read a tabulated potential from a file `tab_potential_argon.dat` and add it between argon particles.

```
# Setup an argon crystal.

# Set up lattice
vector_a = [5.316, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.316, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.316]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Argon, Argon, Argon, Argon]

# Define coordinates
fractional_coordinates = [[ 0. ,  0. ,  0. ],
                          [ 0.5,  0.5,  0. ],
                          [ 0.5,  0. ,  0.5],
                          [ 0. ,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

potential_set = TremoloXPotentialSet(name = "argon_tabulated")

# Add default argon particle.
potential_set.addParticleType(ParticleType.fromElement(Argon))

# Add the tabulated potential form the file 'tab_potential_argon.dat'.
potential_set.addPotential(UserDefinedTabulatedPotential(
    particleType1='Ar',
    particleType2='Ar',
    fileName='tab_potential_argon.dat',
    splineType='fivepoint'
))

calculator = TremoloXCalculator(potential_set)

bulk_configuration.setCalculator(calculator)
```

The top of the file `tab_potential_argon.dat` should look like the following example.

```
65000 0.0001
0
4e+48
9.765625e+44
```

(continues on next page)

(continued from previous page)

```
7.52670569264e+42
2.38418579102e+41
1.6384e+40
1.83757463199e+39
2.88990463236e+38
5.82076609135e+37
1.41628246459e+37
4e+36
1.27452327084e+36
4.48626619138e+35
1.71687929672e+35
7.05543123135e+34
3.0829386517e+34
# .. continue until the specified number of points is reached.
```

Notes

Define a tabulated potential that is read from a file. The tabulated file must be present in your project directory, otherwise the correct path has to be given.

The first line of the file must contain two entries:

- The number of data points N of the tabulation.
- The distance Δr between two data points.

The following N lines must contain the potential energy values $E_i, (i = 1...N)$, representing the energy at a particle distance $r = i\Delta r$.

Possible spline types are

- hermitian
- bessel
- akima
- natural
- fivepoint

If hermitian splines are used, a second column is required in the potential file, which contains the derivative values.

VFFBondBendingPotential

class VFFBondBendingPotential(*particleType1*, *particleType2*, *particleType3*, *alpha*, *delta*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.

- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.
- **delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a valence force field for silicon:

```
potentialSet = TremoloXPotentialSet(name='VFF_silicon')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[]
))

# Add a VFFBondBendingPotential for Si-Si-Si angles.
potential = VFFBondBendingPotential(
    particleType1=ParticleIdentifier('Si'),
    particleType2=ParticleIdentifier('Si',),
    particleType3=ParticleIdentifier('Si',),
    alpha=0.0584121324987*eV/Angstrom**4,
    delta=1.84321352251*Angstrom**2,
)
potentialSet.addPotential(potential)
```

Here, only the VFFBondBendingPotential block of the script is shown. The full script can be found found in the file `silicon_vff1.py`.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential is calculated as

$$V(\mathbf{r}_{i,j}, \mathbf{r}_{j,k}) = \alpha \left(\cos(\theta_{i,j,k}) r_{i,j} r_{j,k} - \delta\right)^2 = \alpha \left(\mathbf{r}_{j,i} \cdot \mathbf{r}_{j,k} - \delta\right)^2$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ is the corresponding distance.

VFFBondBendingStretchingPotential

class VFFBondBendingStretchingPotential(*particleType1*, *particleType2*, *particleType3*, *alpha*, *r0*, *r1*, *delta*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**r0** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**r1** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential is calculated as

$$V(\mathbf{r}_{i,j}, \mathbf{r}_{j,k}) = \frac{\alpha}{2} \left(\mathbf{r}_{j,i} \cdot \mathbf{r}_{j,k} - \delta\right) \left[(r_{i,j}^2 - r_0) + (r_{j,k} - r_1)\right]$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ is the corresponding distance.

VFFBondStretchingPotential

class VFFBondStretchingPotential(*particleType1*, *particleType2*, *alpha*, *delta*, *A*, *epsilon*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** – Identifier of the second particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**A** (*PhysicalQuantity of type length**-2*) – Potential parameter.

•**epsilon** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Set up a valence force field for silicon:

```
potentialSet = TremoloXPotentialSet(name='VFF_silicon')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    charge=None,
    sigma=None,
    sigma14=None,
    epsilon=None,
    epsilon14=None,
    atomicNumber=14,
    tags=[]
))

# Add a VFFBondStrechingPotential for Si-Si bonds.
potential = VFFBondStretchingPotential(
    particleType1=ParticleIdentifier('Si'),
    particleType2=ParticleIdentifier('Si'),
    alpha=0.205289014474*eV/Angstrom**4,
    delta=5.52964056752*Angstrom**2,
    A=0.0*1/Angstrom**2,
    epsilon=0.0*Angstrom**2,
)
potentialSet.addPotential(potential)
```

Here, only the VFFBondStretchingPotential block of the script is shown. The full script can be found found in the file `silicon_vff1.py`.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential is calculated as

$$V(r) = \alpha \left[1 + A(r^2 - \epsilon)\right] \left(r^2 - \delta\right)^2$$

where r is the bond distance.

VFFCoplanarBondBendingPotential

class VFFCoplanarBondBendingPotential(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *alpha*, *delta*, *epsilon*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**epsilon** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

Notes

- This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.
- This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].
- This potential acts on a quadruple of atoms *(i,j,k,l)* and it is calculated as

$$V(\mathbf{r}_{i,j}, \mathbf{r}_{j,k}, \mathbf{r}_{k,l}) = \alpha \left(\mathbf{r}_{j,i} \cdot \mathbf{r}_{j,k} - \delta\right) \left(\mathbf{r}_{h,j} \cdot \mathbf{r}_{k,l} - \epsilon\right)$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ is the corresponding distance.

VFFCrossBondStretchingPotential

class VFFCrossBondStretchingPotential(*particleType1*, *particleType2*, *particleType3*, *alpha*, *r0*, *r1*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.
- **r0** (*PhysicalQuantity of type length**2*) – Potential parameter.
- **r1** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential is calculated as

$$V(r_{i,j}, r_{j,k}) = \alpha(r_{i,j}^2 - r_0)(r_{j,k} - r_1)$$

where $r_{i,j}$ is the bond distance between atoms *i* and *j*.

VFFModifiedBondBendingPotential1

class VFFModifiedBondBendingPotential1(*particleType1*, *particleType2*, *particleType3*, *alpha*, *delta*, *A*, *epsilon*, *B*, *mu*)

Constructor of the potential.

Parameters

•**particleType1** (*ParticleType or ParticleIdentifier*) – Identifier of the first particle type.

•**particleType2** (*ParticleType or ParticleIdentifier*) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.

•**particleType3** (*ParticleType or ParticleIdentifier*) – Identifier of the third particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**A** (*float*) – Potential parameter.

•**epsilon** (*float*) – Potential parameter.

•**B** (*PhysicalQuantity of type length**-2*) – Potential parameter.

•**mu** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2]. It is a variant of the *VFFBondBendingPotential*.

•This potential is calculated as

$$V(\mathbf{r}_{i,j}, \mathbf{r}_{j,k}) = \alpha \left[1 + A(\cos(\theta_{i,j,k}) - \epsilon)\right] \left[1 + B(r_{i,j} r_{j,k} -_m u)\right] \left(\mathbf{r}_{j,i} \cdot \mathbf{r}_{j,k} - \delta\right)^2$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, $r_{i,j}$ is the corresponding distance, and $\theta_{i,j,k}$ is the angle between atoms *i*, *j*, and *k*.

VFFModifiedBondBendingStretchingPotential1

class VFFModifiedBondBendingStretchingPotential1(*particleType1*, *particleType2*, *particleType3*, *alpha*, *r0*, *r1*, *delta*, *A*, *epsilon*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**r0** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**r1** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**A** (*PhysicalQuantity of type length**-2*) – Potential parameter.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

•**epsilon** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential is a modified version of the *VFFBondBendingStretchingPotential*. It is calculated as

$$V(\mathbf{r}_{i,j}, \mathbf{r}_{j,k}) = \frac{\alpha}{2}\left[1 + A(r_{i,j}r_{j,k} - \epsilon)\right](\mathbf{r}_{j,i} \cdot \mathbf{r}_{j,k} - \delta)\left[(r_{i,j}^2 - r_0) + (r_{j,k} - r_1)\right]$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ is the corresponding distance.

VFFModifiedCoplanarBondBendingPotential1

class VFFModifiedCoplanarBondBendingPotential1(*particleType1*, *particleType2*, *particleType3*, *particleType4*, *alpha*, *delta*, *epsilon*, *A*, *mu*, *B*, *nu*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

•**particleType4** (ParticleType *or* ParticleIdentifier) – Identifier of the fourth particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**epsilon** – Potential parameter.

•**A** (*PhysicalQuantity of type length**-2*) – Potential parameter.

•**mu** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**B** (*PhysicalQuantity of type length**-2*) – Potential parameter.

•**nu** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

•This potential acts on a quadruple of atoms *(i,j,k,l)* and it is calculated as

$$V(\mathbf{r}_{i,j},\mathbf{r}_{j,k},\mathbf{r}_{k,l}) = \alpha\left[1 + A(r_{i,j}r_{j,k} - \mu) + B(r_{j,k}r_{k,l} - \nu)\right](\mathbf{r}_{j,i}\cdot\mathbf{r}_{j,k} - \delta)(\mathbf{r}_{h,j}\cdot\mathbf{r}_{k,l} - \epsilon)$$

where $\mathbf{r}_{i,j}$ is the distance vector from atom *i* to atom *j*, and $r_{i,j}$ is the corresponding distance.

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

VFFModifiedCrossBondStretchingPotential1

class VFFModifiedCrossBondStretchingPotential1(*particleType1*, *particleType2*, *particleType3*, *alpha*, *r0*, *r1*, *A*, *delta*)

Constructor of the potential.

Parameters

•**particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.

•**particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.

•**particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.

•**alpha** (*PhysicalQuantity of type energy * length**-4*) – Potential parameter.

•**r0** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**r1** (*PhysicalQuantity of type length**2*) – Potential parameter.

•**A** (*PhysicalQuantity of type length**-2*) – Potential parameter.

•**delta** (*PhysicalQuantity of type length**2*) – Potential parameter.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

•This potential class is a bonded potential. This means it requires a bond topology to be specified on the configuration which the potential should be used for. Bonds can be set using the `findBonds()` method on configurations (*MoleculeConfiguration*, *BulkConfiguration*, *DeviceConfiguration*, or *SurfaceConfiguration*), which automatically adds bonds for atoms which are closer than their combined covalent radii (multiplied by a `fuzz_factor` of 1.1 by default). Alternatively bonds can be set manually by passing a list of the atom index pairs that form the bonds to the `setBonds()` method on the configuration. If no bonds are specified on the configuration bonded potentials have no effect.

•This potential is part of the potential functions that are used in *valence force fields (VFF)*[1][2].

[1] P.ăN. Keating. Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure. *Phys. Rev.*, 145:637–645, 1966. doi:10.1103/PhysRev.145.637.

[2] OlgaăL. Lazarenkova, Paul von Allmen, Fabiano Oyafuso, Seungwon Lee, and Gerhard Klimeck. Effect of anharmonicity of the strain energy on band offsets in semiconductor nanostructures. *Appl. Phys. Lett.*, 85(18):4193–4195, 2004. doi:10.1063/1.1814810.

•This potential is a variant of the *VFFCrossBondStretchingPotential*. It is calculated as

$$V(r_{i,j}, r_{j,k}) = \alpha \left[1 + A(r_{i,j} r_{j,k} - \delta)\right] (r_{i,j}^2 - r_0)(r_{j,k} - r_1)$$

where $r_{i,j}$ is the bond distance between atoms *i* and *j*.

Vacancy

class Vacancy(*site_index=None*, *unit_cell_index=None*)

Defines a vacancy defect. Used as part of a *ChargedPointDefect* study.

Parameters

•**site_index** (*int*) – The index of the atom in the bulk unit cell configuration to be removed to create the vacancy. **Default:** `0`

•**unit_cell_index** (*sequence (size 3) of int*) – The index of the unit cell in the infinite crystal which the vacancy site belongs to. **Default:** `(0, 0, 0)`

generateDefectConfiguration(*reference_bulk_configuration*, *supercell_repetitions*, *use_ghost=None*)

Generate the supercell with the embedded defect.

Parameters

•**reference_bulk_configuration** (*BulkConfiguration*) – The bulk unit cell configuration.

•**supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the bulk unit cell along the (a, b, c) directions.

•**use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
The supercell configuration with the embedded defect.

Return type
BulkConfiguration

siteIndex()

Returns
The index of the lattice site which the defect is referenced to.

Return type
int

uniqueString()

Return a unique string representing the state of the object.

unitCellIndex()

Returns
The index of the unit cell in the infinite crystal which the reference site belongs to.

Return type
tuple (size 3) of int

Notes

The *Vacancy* object defines a vacancy defect in a material.

A *Vacancy* can be added to a *NamedPointDefect* to define a defect for use in a *ChargedPointDefectConfiguration* calculation.

See *Defining the point defect* for a description of how to use this object as part of a *ChargedPointDefect* study.

VacancyGenerator

class VacancyGenerator(*host_configuration*, *symmetry_tolerance=None*)

A class which describes the possible vacancy defects for a given host material.

Parameters

- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **symmetry_tolerance** (*PhysicalQuantity* of type length) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom.

filterByDefectGenerators(*defect_generators*)

Method for filtering a defect generator by keeping the defects that are in one or more other defect generators.

Parameters
defect_generators (list of *BaseDefectGenerator*) – The list of defect generators containing the defects to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByDistinctConfigurations(*top_n_configurations*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters
top_n_configurations (*int*) – Number of most distinct configurations.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByLatticeSpecies(*element_list*)

Method for selecting unique defects by element. Valid only for defect types that are associated with a lattice site, e.g. Vacancies, but not Interstitials.

Parameters
element_list (*PeriodicTableElement* | list of *PeriodicTableElement*) – List of elements to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *SplitInterstitialGenerator*

filterByLatticeSpeciesAndDistinctConfigurations(*top_n_configurations*, *element*)

Method for selecting a number of distinct unique defects using the MTP structural descriptor.

Parameters

- **top_n_configurations** (*int*) – Number of most distinct configurations.
- **element** (*PeriodicTableElement*) – Element to keep after filtering.

Returns
A filtered defect list.

Return type
VacancyGenerator

filterByPointDefect(*point_defects*, *keep_defects=True*)

Method for filtering the generator of unique point defects by selecting a specific defects.

Parameters

- **point_defects** (*Sequence of BasePointDefect*) – The sequence of point defects to filter.
- **keep_defects** (*bool*) – Whether or not the given defects are to be kept and all other removed (True) or removed from the generator and all other defect kept (False). **Default:** True

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByPositionInZSlices(*start=None*, *end=None*, *resolution=None*, *in_plane_position=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

- **start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.
- **end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.
- **resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`
- **in_plane_position** (*Sequence (2) of type float*) – The selected defect in each slice is as close as possible to this fractional coordinate. Mutually exclusive with parameter *select_distinct*. **Default:** `[0.5, 0.5]`
- **select_distinct** (*bool*) – Select the most distinct defects per slice. This can be used to sample a diverse configuration space with a minimal number of defects. Mutually exclusive with parameter *in_plane_position*. **Default:** `False`
- **defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** `1`

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySimilarityInZSlices(*start=None*, *end=None*, *resolution=None*, *defects_per_slice=None*)

Select defects along the Z-axis of the cell. This method is intended to be used for configurations that are anisotropic along one direction, such as interfaces or surfaces.

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

•**resolution** (*PhysicalQuantity* of type length) – The size of each slice of the interface profile. **Default:** `2.0 * Angstrom`

•**defects_per_slice** (*int*) – The number of defects to select per slice. This parameter is only used together with the parameter *select_distinct*. **Default:** 1

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySphere(*center*, *radius*)

Method for filtering unique defects by their position in a sphere.

Parameters

•**center** (*PhysicalQuantity* of type position) – The center of the sphere in Cartesian coordinates.

•**radius** (*PhysicalQuantity* of type length) – The radius of the sphere.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterBySymmetryIndex(*symmetry_indices*)

Method for filtering the list of unique point defects by selecting specific symmetry indices.

Parameters
symmetry_indices (*int | list of int*) – The list of symmetry indices to keep after filtering.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

filterByZPositionInterval(*start=None*, *end=None*)

Method for selecting unique defects by their position in an interval of the z axis. Note: The

Parameters

•**start** (*PhysicalQuantity* of type length) – The start of the position interval. **Default:** 0 * Angstrom.

•**end** (*PhysicalQuantity* of type length) – The end of the position interval. **Default:** length of the cell in z direction.

Returns
A filtered defect generator.

Return type
VacancyGenerator | *SubstitutionalGenerator* | *InterstitialGenerator* *SplitInterstitialGenerator*

hostConfiguration()

Returns
The host configuration.

Return type
BulkConfiguration

indices()

Returns
The defect indices for the current defect generator.

Return type
list of int

pointDefects()

Returns
The list of unique point defects.

Return type
Vacancy | *Substitutional* | *Interstitial* | *DefectCluster* | *SplitInterstitial*

symmetryTolerance()

Returns
The symmetry tolerance.

Return type
PhysicalQuantity of type length

uniqueDefects()

Returns
The list of symmetrically unique point defects with their names and symmetry indices, ordered by increasing index.

Return type
list of *NamedPointDefect*

uniqueString()

Return a unique string representing the state of the object.

Notes

The *VacancyGenerator* object generates the symmetrically unique substitutional defects for a given host material, by identifying the symmetrically equivalent atoms and positions. The *VacancyGenerator* object is derived from the *BaseDefectGenerator* object, and thus contains the common methods from that object.

To create a *VacancyGenerator* object, a specific host configuration must be given. Once the generator is created, filters can be added so that only the desired subset of defects are generated.

The unique defects can be returned with the `uniqueDefects` method as *NamedPointDefect* objects, which can be used as input for a *ChargedPointDefectConfiguration* object. The defects and generators can also be given to a *Defects* object to create the basic defect configurations.

VampireCMCAnisotropySimulation

class VampireCMCAnisotropySimulation(*temperature_range=None*, *constraint_theta_sampling=None*, *constraint_phi_sampling=None*, *equilibration_time_steps=None*, *time_steps_increment=None*, *loop_time_steps=None*, *atoms_output_rate=None*)

Parameters

- **temperature_range** (*list | tuple*) – The temperature range specified as a list of equidistantly spaced temperatures like `numpy.linspace(0, 1500, 61)*Kelvin` or as a tuple like `(0, 1500, 61, Kelvin)`. **Default:** `numpy.linspace(0, 1500, 61)*Kelvin`
- **constraint_theta_sampling** (*list | tuple*) – The contraint angle theta sampling specified as a list of equidistantly spaced angles like `numpy.linspace(0, 90, 11)*Degrees` or as a tuple like `(0, 90, 11, Degrees)`. **Default:** `numpy.linspace(0, 90, 11)*Degrees`
- **constraint_phi_sampling** (*list | tuple*) – The contraint phi theta sampling specified as a list of equidistantly spaced angles like `numpy.linspace(0, 90, 11)*Degrees` or as a tuple like `(0, 90, 11, Degrees)`. **Default:** `numpy.linspace(0, 90, 11)*Degrees`
- **equilibration_time_steps** (*int*) – Number of time steps used for equilibration. **Default:** 10,000
- **time_steps_increment** (*int*) – Statistics are calculated once every time_steps_increment number of time steps. **Default:** 1
- **loop_time_steps** (*int*) – The number of time steps that statistics are taken over. This takes place after equilibration. **Default:** 10,000
- **atoms_output_rate** (*int*) – Determines the rate configuration files are outputted as a multiple of time_steps_increment. If 0, output of configuration data is disabled. **Default:** 1

atomsOutputRate()

Returns
The atoms output rate.

Return type
int

constraintAnglePhiIncrement()

Returns
The phi angle increment

Return type
PhysicalQuantity of type Degrees.

constraintAnglePhiMaximum()

Returns
The maximum phi angle

Return type
PhysicalQuantity of type Degrees.

constraintAnglePhiMinimum()

Returns
The minimum phi angle

Return type
PhysicalQuantity of type Degrees.

`constraintAngleThetaIncrement()`

Returns
The theta angle increment

Return type
PhysicalQuantity of type Degrees.

`constraintAngleThetaMaximum()`

Returns
The maximum theta angle

Return type
PhysicalQuantity of type Degrees.

`constraintAngleThetaMinimum()`

Returns
The minimum theta angle.

Return type
PhysicalQuantity of type Degrees

`equilibrationTimeSteps()`

Returns
The number of equilibration time steps

Return type
int

`integrator()`

Returns
The integrator

Return type
str

`loopTimeSteps()`

Returns
The number of time steps used for statistics.

Return type
int

`maximumTemperature()`

Returns
The maximum temperature.

Return type
PhysicalQuantity of type temperature

`minimumTemperature()`

Returns
The minimum temperature.

Return type
PhysicalQuantity of type temperature

`temperatureIncrement()`

Returns
The temperature increment.

Return type
PhysicalQuantity of type temperature

`timeStep()`

Returns
The time step.

Return type
PhysicalQuantity of type time

`timeStepsIncrement()`

Returns
The time step increment.

Return type
int

`writeToFile(fd)`

Write simulation parameters.

Parameters
fd (*File object.*) – File object to write to.

VampireConfigurationParameters

`class VampireConfigurationParameters(configuration_type=None, periodic_boundary_conditions=None, system_size=None, particle_size=None, particle_shape_factor=None)`

Parameters

- **configuration_type** (*'full' | 'cylinder' | 'cube' | 'sphere'*) – The type of configuration. **Default:** full
- **periodic_boundary_conditions** (*list*) – Tuple of bool indicating if periodic boundary conditions should be used in the x,y,z directions. **Default:** [False, False, False]
- **system_size** (*PhysicalQuantity* of type length) – List of system size in x,y,z directions. **Default:** [10,10,10]*nm
- **particle_size** (*PhysicalQuantity* of type length) – Used as radius of cylinder and sphere, and size of cube. **Default:** 10*nm
- **particle_shape_factor** (*list*) – List of floats. Modifies the default particle shape to create elongated particles. **Default:** [1,1,1]

`configurationType()`

Returns
The configuration type

Return type
str

particleShapeFactor()

Returns
List of floats between 0.001 and 1.0.

Return type
list

particleSize()

Returns
Particle size

Return type
PhysicalQuantity of type length.

periodicBoundaryConditions()

Returns
List of bool indicating if periodic boundary conditions are used in x, y, z directions.

Return type
list

systemSize()

Returns
List of system size in x,y,z directions.

Return type
PhysicalQuantity of type length.

writeToFile(*fd*)

Write the configuration information to the file object.

Parameters
fd (*File object.*) – File object to write to.

VampireCurieTemperatureSimulation

class VampireCurieTemperatureSimulation(*temperature_range=None*, *equilibration_time_steps=None*, *time_steps_increment=None*, *loop_time_steps=None*, *atoms_output_rate=None*)

Parameters

- **temperature_range** (*list*) – The temperature range specified as a list of equidistantly spaced temperatures like `numpy.linspace(0, 1500, 61)*Kelvin` or as a tuple like `(0, 1500, 61, Kelvin)`. **Default:** `numpy.linspace(0, 1500, 61)*Kelvin`
- **equilibration_time_steps** (*int*) – Number of time steps used for equilibration. **Default:** 10,000
- **time_steps_increment** (*int*) – Statistics are calculated once every time_steps_increment number of time steps. **Default:** 1
- **loop_time_steps** (*int*) – The number of time steps that statistics are taken over. This takes place after equilibration. **Default:** 10,000
- **atoms_output_rate** (*int*) – Determines the rate configuration files are outputted as a multiple of time_steps_increment. If 0, output of configuration data is disabled. **Default:** 1

atomsOutputRate()

Returns
The rate at which configuration files are outputted as a multiple of time_steps_increment.

Return type
int

equilibrationTimeSteps()

Returns
The number of equilibration time steps

Return type
int

integrator()

Returns
The integrator

Return type
str

loopTimeSteps()

Returns
The number of time steps used for statistics.

Return type
int

maximumTemperature()

Returns
The maximum temperature

Return type
PhysicalQuantity of type temperature.

minimumTemperature()

Returns
The minimum temperature

Return type
PhysicalQuantity of type temperature.

temperatureIncrement()

Returns
The temperature increment

Return type
PhysicalQuantity of type temperature.

temperatureRange()

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

`timeStepsIncrement()`

Returns
The time step increment.

Return type
int

`writeToFile(fd)`

Write simulation parameters.

Parameters
fd (*File object.*) – File object to write to.

VampireFieldCoolSimulation

`class VampireFieldCoolSimulation(minimum_temperature=None, maximum_temperature=None, cooling_time=None, cooling_function=None, applied_field_strength=None, equilibration_time_steps=None, time_steps_increment=None, loop_time_steps=None, time_step=None, output_rate=None, atoms_output_rate=None)`

Parameters

•**minimum_temperature** (*PhysicalQuantity* of type temperature) – The minimum temperature **Default:** 0 Kelvin

•**maximum_temperature** (*PhysicalQuantity* of type temperature) – The maximum temperature **Default:** 1000 Kelvin

•**cooling_time** (*PhysicalQuantity* of type time.) – The cooling time. **Default:** 100 ps

•**cooling_function** (*str*) – The cooling function. This determines the shape of the temperature vs time profile. This can either be gaussian, exponential, double-gaussian or linear. **Default:** gaussian

•**applied_field_strength** (*PhysicalQuantity* of type Tesla) – The applied field strength. **Default:** 4 Tesla

•**equilibration_time_steps** (*int*) – Number of time steps used for equilibration. **Default:** 10,000

•**time_steps_increment** (*int*) – Statistics are calculated once every time_steps_increment number of time steps. **Default:** 1

•**time_step** (*PhysicalQuantity* of type time.) – The time step used in the simulations **Default:** 0.1 fs

•**output_rate** (*int*) – Determines the rate output files are outputted as a multiple of time_steps_increment. **Default:** 10000

•**atoms_output_rate** (*int*) – Determines the rate configuration files are outputted as a multiple of time_steps_increment. If 0, output of configuration data is disabled. **Default:** 100000

`appliedFieldStrength()`

Returns
The applied field strength

Return type
PhysicalQuantity of type Tesla.

atomsOutputRate()

Returns
The rate at which configuration files are outputted as a multiple of time_steps_increment.

Return type
int

coolingFunction()

Returns
The cooling function

Return type
str

coolingTime()

Returns
The cooling time

Return type
PhysicalQuantity of type time.

equilibrationTimeSteps()

Returns
The number of equilibration time steps

Return type
int

integrator()

Returns
The integrator

Return type
str

loopTimeSteps()

Returns
The number of time steps used for statistics.

Return type
int

maximumTemperature()

Returns
The maximum temperature

Return type
PhysicalQuantity of type temperature.

minimumTemperature()

Returns
The minimum temperature

Return type
PhysicalQuantity of type temperature.

outputRate()

Returns
The rate configuration files are outputted as a multiple of time_steps_increment.

Return type
int

temperatureIncrement()

Returns
The temperature increment

Return type
PhysicalQuantity of type temperature.

temperatureRange()

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

timeStepsIncrement()

Returns
The time step increment.

Return type
int

totalTimeSteps()

Returns
The total number of time steps

Return type
int

writeToFile(*fd*)
Write simulation parameters.

Parameters
fd (*File object.*) – File object to write to.

VampireHysteresisLoopSimulation

class VampireHysteresisLoopSimulation(*temperature=None*, *maximum_applied_field_strength=None*, *applied_field_strength_increment=None*, *applied_field_angle_phi=None*, *equilibration_time_steps=None*, *time_steps_increment=None*, *loop_time_steps=None*, *time_step=None*, *atoms_output_rate=None*)

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature **Default:** 5 Kelvin

- **maximum_applied_field_strength** (*PhysicalQuantity* of type Tesla) – The maximum strength of the applied external field acting on the system. In this simulation, this maximum is the maximum magnitude, and dictates both the maximum and minimum (+/-) magnetisation in the target direction. **Default:** 2.0 T
- **applied_field_strength_increment** (*PhysicalQuantity* of type Tesla) – The increment in the strength of the applied external field acting on the system. **Default:** 0.01 T
- **applied_field_angle_phi** – The angle rotated away from z-axis **Default:** 0.1*Degrees
- **equilibration_time_steps** (*int*) – Number of time steps used for equilibration. **Default:** 10,000
- **time_steps_increment** (*int*) – Statistics are calculated once every time_steps_increment number of time steps. **Default:** 1
- **loop_time_steps** (*int*) – The number of time steps that statistics are taken over. This takes place after equilibration. **Default:** 10,000
- **time_step** (*PhysicalQuantity* of type time.) – The time step used in the simulations **Default:** 1 fs
- **atoms_output_rate** (*int*) – Determines the rate configuration files are outputted as a multiple of time_steps_increment. If 0, output of configuration data is disabled. **Default:** 1

appliedFieldAnglePhi()

Returns
The angle of the applied fiels away from the z-axis.

Return type
PhysicalQuantity of type Degrees.

appliedFieldStrengthIncrement()

Returns
The increment in the strength of the applied external field.

Return type
PhysicalQuantity of type Tesla.

atomsOutputRate()

Returns
The rate at which configuration files are outputted as a multiple of time_steps_increment.

Return type
int

equilibrationTimeSteps()

Returns
The number of equilibration time steps

Return type
int

integrator()

Returns
The integrator

Return type
str

loopTimeSteps()

Returns
The number of time steps used for statistics.

Return type
int

maximumAppliedFieldStrength()

Returns
The maximum strength of the applied external field.

Return type
PhysicalQuantity of type Tesla.

temperature()

Returns
The temperature

Return type
PhysicalQuantity of type temperature.

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

timeStepsIncrement()

Returns
The time step increment.

Return type
int

writeToFile(*fd*)
Write simulation parameters.

Parameters
fd (*File object.*) – File object to write to.

VampireStaticHysteresisLoopSimulation

class VampireStaticHysteresisLoopSimulation(*maximum_applied_field_strength=None*, *applied_field_strength_increment=None*, *applied_field_angle_phi=None*, *equilibration_time_steps=None*, *time_steps_increment=None*, *loop_time_steps=None*, *time_step=None*, *atoms_output_rate=None*)

Parameters

- **temperature** (*PhysicalQuantity* of type temperature) – The temperature **Default:** 0 Kelvin

- **maximum_applied_field_strength** (*PhysicalQuantity* of type Tesla) – The maximum strength of the applied external field acting on the system. In this simulation, this maximum is the maximum magnitude, and dictates both the maximum and minimum (+/-) magnetisation in the target direction. **Default:** 2.0 T
- **applied_field_strength_increment** (*PhysicalQuantity* of type Tesla) – The increment in the strength of the applied external field acting on the system. **Default:** 0.01 T
- **applied_field_angle_phi** – The angle rotated away from z-axis **Default:** 0.1*Degrees
- **equilibration_time_steps** (*int*) – Number of time steps used for equilibration. **Default:** 10,000
- **time_steps_increment** (*int*) – Statistics are calculated once every time_steps_increment number of time steps. **Default:** 1
- **loop_time_steps** (*int*) – The number of time steps that statistics are taken over. This takes place after equilibration. **Default:** 10,000
- **time_step** (*PhysicalQuantity* of type time.) – The time step used in the simulations **Default:** 1 fs
- **atoms_output_rate** (*int*) – Determines the rate configuration files are outputted as a multiple of time_steps_increment. If 0, output of configuration data is disabled. **Default:** 1

appliedFieldAnglePhi()

Returns
The angle of the applied fiels away from the z-axis.

Return type
PhysicalQuantity of type Degrees.

appliedFieldStrengthIncrement()

Returns
The increment in the strength of the applied external field.

Return type
PhysicalQuantity of type Tesla.

atomsOutputRate()

Returns
The rate at which configuration files are outputted as a multiple of time_steps_increment.

Return type
int

equilibrationTimeSteps()

Returns
The number of equilibration time steps

Return type
int

integrator()

Returns
The integrator

Return type
str

loopTimeSteps()

Returns
The number of time steps used for statistics.

Return type
int

maximumAppliedFieldStrength()

Returns
The maximum strength of the applied external field.

Return type
PhysicalQuantity of type Tesla.

temperature()

Returns
The temperature

Return type
PhysicalQuantity of type temperature.

timeStep()

Returns
The time step.

Return type
PhysicalQuantity of type time

timeStepsIncrement()

Returns
The time step increment.

Return type
int

writeToFile(*fd*)
Write simulation parameters.

Parameters
fd (*File object.*) – File object to write to.

VaporPressure

class VaporPressure(*solvent*, *temperature=None*, *parameters=None*)
Calculate the vapor pressure of a liquid.

Parameters

- **solvent_components** (*MoleculeConfiguration* | *CosmoRealSpecies* | *CosmoRSMixture*) – The solvent components.
- **temperature** (*PhysicalQuantity* of type temperature) – The solvent and vapor temperature. **Default:** `298 * Kelvin`
- **parameters** (*CosmoRSParameters*) – The COSMO-RS parameters

parameters()

Returns
The COSMO real solvent parameters

Return type
CosmoRSParameters

temperature()

Returns
The solvent temperature.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

vaporPressure()

Returns
The vapor pressure of each solvent component, ordered by their input order.

Return type
PhysicalQuantity of type pressure

Usage Examples

Calculate the vapor pressure of acetonitrile in a water mixture.

```
# Load the COSMO species
database = CosmoRSSpeciesDatabase()
water = database.exportSpecies('water')
acetonitrile = database.exportSpecies('acetonitrile')
acetonitrile.setAntoineA(5.93296 * bar)
acetonitrile.setAntoineB(2345.829 * Kelvin)
acetonitrile.setAntoineC(43.815 * Kelvin)

# Create a mixture
mixture = CosmoRSMixture([(water, 0.5), (acetonitrile, 0.5)])

# Determine the vapor pressure.
vapor_pressure = VaporPressure(
    mixture,
    298*Kelvin,
    parameters=CosmoRSParameters(),
)

# Retrieve the vapor pressure of acetonitrile.
p_vap = vapor_pressure.vaporPressure()[1].convertTo(bar)
nlprint(f'The vapor pressure of acetonitrile is {p_vap}')
```

water_acn_vapor_pressure_example.py

Notes

The *VaporPressure* object allows calculation of the partial vapor pressure of a substance above its condensed phase. The partial vapor pressure p_{vap} of a species i is calculated according to

$$p_i^{vap} = x_i p_0 \exp\left(\frac{\mu_i^{solv} - \mu_i^{gas}}{RT}\right)$$

Here μ_i^{solv} is the pseudochemical potential of solvation of species i and μ_i^{gas} the pseudochemical potential of the gas-phase with respect to the perfect conductor condensed phase. p_0 is a predefined reference pressure of 1 bar and x_i the mole fraction.

If the *Antoine* parameters are known for the pure substances then the partial pressure, p^{pure} can be estimated as:

$$\log_{10}(p^{pure}) = A - \frac{B}{C + T}$$

Here T is the temperature and A, B and C are the Antoine equation parameters. Since the partial pressure of a pure substance in COSMO-RS is defined as

$$p^{pure} = p_0 \exp\left(\frac{\mu^{liq} - \mu^{gas}}{RT}\right)$$

Rearranging the expression then yields

$$\mu^{gas} = \mu^{liq} - RT \ln \frac{p^{pure}}{p_0}$$

This can be used to simplify the calculation of a mixture vapor pressure to the difference in chemical potential between the solvated and pure liquid states, μ_i^{liq}. The vapor pressure is then

$$p_i^{vap} = x_i p_i^{pure} \exp\left(\frac{\mu_i^{solv} - \mu_i^{liq}}{RT}\right)$$

Using this expression avoids approximating the gas chemical potential and simply estimates the partial pressure based on the difference between the solvated and pure liquid chemical potentials.

VelocityAutocorrelation

class VelocityAutocorrelation(*md*, *start_time=None*, *end_time=None*, *atom_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the velocity auto-correlation function of an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfigution) – The MDtrajectory or configuration to calculate the velocity autocorrelation for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Returns
The normalized velocity autocorrelation values.

Return type
array

times()
Return the time values.

Usage Examples

Load an *MDTrajectory* and calculate the *VelocityAutocorrelation* function of the oxygen atoms in the simulation:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

# Calculate the velocity autocorrelation function of the oxygen atoms.
vacf = VelocityAutocorrelation(md_trajectory, atom_selection=Oxygen)

# Get the time and autocorrelation values.
times     = vacf.times().inUnitsOf(fs)
vacf_data = vacf.data()

# Plot the data using pylab.
import pylab

pylab.plot(times, vacf_data,
           label='Velocity autocorrelation of the oxygen atoms')
pylab.xlabel('t (fs)')
pylab.ylabel('VACF(t)')
pylab.legend()

pylab.show()
```

velocity_autocorrelation.py

Notes

The *VelocityAutocorrelation* function is normalized to one at $t = 0$ and weighted with the atomic masses, i.e.

$$vacf(t) = \frac{\sum_i \langle m_i \mathbf{v}_i(t) \cdot \mathbf{v}_i(0) \rangle}{\sum_i \langle m_i \mathbf{v}_i^2 \rangle} .$$

In practice, the average $< ... >$ runs over all selected atoms i in the trajectory, and an additional average over simulation time is carried out to improve the statistical sampling. That means for a given time difference t all image pairs that are separated by t are taken into account in the average, as

$$\langle m_i \mathbf{v}_i(t) \cdot \mathbf{v}_i(0) \rangle = \frac{1}{(t_{max} - t)} \sum_{t'=0}^{t_{max}-t} \mathbf{v}_i(t' + t) \cdot \mathbf{v}_i(t')$$

By default, all elements are taken into account, but a specified selection can be given as well. The `atom_selection` parameter accepts an element, a tag name, or a list of indices to select atoms for the velocity distribution. This can be useful, e.g. in the presence of constraints as constrained atoms should be excluded in this analysis.

VelocityDistribution

class VelocityDistribution(*md_trajectory*, *start_time=None*, *end_time=None*, *maximum_velocity=None*, *resolution=None*, *atom_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the velocity distribution of an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the velocity distribution.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **maximum_velocity** (*PhysicalQuantity* of type length/time) – Upper limit of considered velocities. **Default:** the maximum velocity of the trajectory
- **resolution** (*PhysicalQuantity* of type length/time.) – The bin size, which determines the resolution of the histogram. **Default:** `0.001 * Angstrom / fs`
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

data()

Return the velocity distribution histogram.

velocities()

Return the velocity values associated with the histogram.

Usage Examples

Load an *MDTrajectory* and calculate the *VelocityDistribution*:

```
md_trajectory = nlread('alumina_trajectory.nc')[-1]

v_distribution = VelocityDistribution(md_trajectory)

# Get the bin_centers and the histogram of the velocity distribution.
velocities = v_distribution.velocities().inUnitsOf(Ang/fs)
histogram  = v_distribution.data()

# Plot the data using pylab.
import pylab

pylab.bar(velocities, histogram, label='Velocity distribution', width=0.001)
pylab.xlabel('v (Ang/fs)')
pylab.ylabel('Histogram')
pylab.legend()

pylab.show()
```

velocity_distribution.py

Notes

By default, all elements are taken into account.

The atom_selection parameter accepts an element, a tag name, or a list of indices to select atoms for the velocity distribution. This can be useful, e.g. in the presence of constraints as constrained atoms should be excluded in this analysis.

VelocityProjectionOptimization

class VelocityProjectionOptimization(*time_step=None*, *initial_direction=None*)

Optimization method based on a Verlet integration scheme.

Parameters

- **time_step** (*PhysicalQuantity* with time units or None) – The time step used in the optimization scheme. A smaller time step will result in a more accurate integration, but will also require more computational time. A larger time step will require less time, but may result in a less accurate integration.
- **initial_direction** (numpy.ndarray of *PhysicalQuantity* with length units per time units or None) – The initial direction (velocities) for the optimizer. If None, the initial velocities are set to zero.

scfInitialGuess()

Returns
The SCF initial guess used for the Verlet optimizer; always the previous configuration.

Return type
UsePreviousConfiguration

uniqueString()

Return a unique string representing the state of the object.

VessalPotential

class VessalPotential(*particleType1*, *particleType2*, *particleType3*, *k*, *theta0*, *rho1*, *rho2*, *rmax1*, *rmax2*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type. This is the central particle (the vertex) when calculating the angle.
- **particleType3** (ParticleType *or* ParticleIdentifier) – Identifier of the third particle type.
- **k** (*PhysicalQuantity of type energy*) – Potential parameter.
- **theta0** (*PhysicalQuantity of type angle*) – Potential parameter
- **rho1** (*PhysicalQuantity of type length*) – Potential parameter.
- **rho2** (*PhysicalQuantity of type length*) – Potential parameter.

•**rmax1** (*PhysicalQuantity of type length*) – Potential parameter.

•**rmax2** (*PhysicalQuantity of type length*) – Potential parameter.

•**r_cut** – Deprecated. This parameter is not used anymore. Please use rmax1 and rmax2 to set the cutoff radius.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

•**parameterName** (*str*) – The name of the parameter that will be modified.

•**value** – The new value that will be assigned to the parameter parameterName.

Notes

The *VessalPotential*[1] defines a screened three-body potential of the form

$$V_{ijk}(r_{ij}, r_{ik}, \theta_{ijk}) = \frac{k}{8(\theta_{ijk} - \pi)^2} \left[(\theta_0 - \pi)^2 - (\theta_{ijk} - \pi)^2\right]^2 \exp\left[-(r_{ij}/\rho_1 + r_{ik}/\rho_2)\right] .$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

•1 when $r \leq r_i$

•In the range $[0, 1]$ when $r_i < r < r_{cut}$

•0 when $r \geq r_{cut}$

[1] B.ăVessal. Simulation studies of silicates and phosphates. *J. Non-Crystal. Solids*, 177:103–124, nov 1994. doi:10.1016/0022-3093(94)90520-7.

VibrationalDensityOfStates

class VibrationalDensityOfStates(*md*, *start_time=None*, *end_time=None*, *atom_selection=None*, *time_resolution=None*, *info_panel=None*)

Class for calculating the vibrational density of states of an MD simulation.

Parameters

- **md_trajectory** (*MDTrajectory*) – The MDtrajectory to calculate the vibrational density of states for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame
- **atom_selection** (*PeriodicTableElement* | str | list of ints) – Only include contributions from this selection. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

frequencies()

Returns
The frequencies associated with the vibrational density of states.

Return type
PhysicalQuantity of type 1/time

vibrationalDensityOfStates()

Returns
The vibrational density-of-states, calculated from the Fourier transformation of the velocities.

Return type
PhysicalQuantity of type time

Usage Examples

Load an *MDTrajectory* and calculate the vibrational density of states from the simulation:

```
md_trajectory = nlread('ag-md.hdf5', class_type=MDTrajectory)[-1]

# Get the vibrational density of states.
vdos = VibrationalDensityOfStates(md_trajectory)

# Get the frequencies associated with the DOS.
frequencies = vdos.frequencies()

# Convert the frequencies to energies in meV.
energies = (frequencies*planck_constant).inUnitsOf(meV)

# Plot the data using pylab.
import pylab

pylab.plot(energies, vdos.vibrationalDensityOfStates(), label='Vibrational density of␣
↪states')
```

(continues on next page)

(continued from previous page)

```
pylab.xlabel('E (meV)')
pylab.ylabel('S(E)')
pylab.xlim(0,50)
pylab.legend()

pylab.show()
```

vibrational_dos.py

Notes

The vibrational density of states is calculated from the Fourier transform of the atomic velocities (cf.[1]):

$$S(\nu) = \frac{2}{k_B T} \sum_{i=1}^{N} \sum_{\alpha=x,y,z} m_i s_i^{\alpha}(\nu),$$

where

$$s_i^{\alpha}(\nu) = \frac{\left| \int_0^{t_{max}} dt' v_i^{\alpha}(t') e^{-i2\pi\nu t} \right|^2}{t_{max}}.$$

The temperature entering the first expression is calculated directly from the velocities during the simulation.

The vibrational DOS are normalized in such a way that the integration over the frequencies (if given in fs^{-1}) gives the total number of degrees of freedom in the system

$$\int_0^{\infty} d\nu S(\nu) = 3N\,.$$

Apart from the normalization, the vibrational density of states should give similar results as the *PhononDensityOfStates* with the important distinction, that in the former, anharmonic effects are seamlessly included. You also have the possibility to extract the contribution of only a part of the system, specified via the `atom_selection` parameter. Moreover, in contrast to the *PhononDensityOfStates* analysis object, the vibrational density of states can be calculated for soft systems or even for liquid systems.

In order to sample all relevant vibrations, the system size must be chosen large enough to accommodate vibrational modes with long wavelengths. The log interval for saving snapshots to the *MDTrajectory* must be chosen such that the sampling rate (the reciprocal of the time step times the logging interval) is twice the highest vibrational frequency that should be included in the density of states. For example, if the time step is 1 femtosecond and the logging interval is 10 steps, then the sampling rate is 100 THz and the maximum frequency in the density of states is 50 THz.

VibrationalInverseParticipationRatio

class VibrationalInverseParticipationRatio(*configuration*, *dynamical_matrix*, *qpoints=None*)

Class for calculating the vibrational inverse participation ratio.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration for which to calculate the vibrational inverse participation ratio.

[1] S.-T. Lin, M.ăBlanco, and W.ăA GoddardăIII. The two-phase model for calculating thermodynamic properties of liquids from molecular dynamics: Validation for the phase diagram of Lennard-Jones fluids. *J. Chem. Phys.*, 2003. doi:10.1063/1.1624057.

- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix from which to calculate the vibrational inverse participation ratio.
- **qpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity* | *RegularKpointGrid* | sequence of sequence (size 3) of float) – The q-points on which to calculate the vibrational inverse participation ratio. **Default:** The Gamma point (0.0, 0.0, 0.0).

energies()

Returns
The energies for each vibrational mode.

Return type
PhysicalQuantity of type energy

inverseParticipationRatio()

Returns
The vibrational inverse participation ratio for each vibrational mode and q-point. The array has size (n_qpoints, n_modes)

Return type
ndarray

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlprint(*stream=None*)
Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

printTable(*stream=None*, *energy_min=None*, *energy_max=None*)
Print a string containing an ASCII table with the vibrational inverse participation ratio data, restricted in an energy window.

Parameters

- **stream** (*A stream that supports strings being written to using 'write'.*) – The stream to write to.
- **energy_min** (*PhysicalQuantity* of type energy) – Lower limit of the energy window. **Default:** 0 * meV
- **energy_max** – Upper limit of the energy window. **Default:** 20 * meV

qpoints()

Returns
The q-points on which to calculate the vibrational inverse participation ratio.

Return type
ndarray

setMetatext(*metatext*)
Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Calculate the *VibrationalInverseParticipationRatio* at the Gamma point for silicon bulk with a vacancy defect and print out information for the most localized state:

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[0.,   0.,   0.  ],
                          [0.25, 0.25, 0.25]]

# Set up configuration
unit_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
reps = 3
bulk_configuration = unit_configuration.repeat(reps, reps, reps)
bulk_configuration = bulk_configuration.copyAndDeleteAtoms(5)

# Set up calculator
potentialSet = Tersoff_Si_2005()
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(1.0*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# Set up density matrix
dynamical_matrix = DynamicalMatrix(
    bulk_configuration,
    filename='Silicon_defect.hdf5',
    object_id='dynamical_matrix',
    repetitions=Automatic,
    atomic_displacement=0.01*Angstrom,
    acoustic_sum_rule=True,
    finite_difference_method=Central,
    force_tolerance=1e-08*Hartree/Bohr**2,
    processes_per_displacement=None,
    log_filename_prefix='forces_displacement_',
    use_wigner_seitz_scheme=False,
    )
dynamical_matrix.update()
```

(continues on next page)

(continued from previous page)

```
# Compute the vibrational inverse participation ratio
vibrational_ipr = VibrationalInverseParticipationRatio(
    configuration=bulk_configuration,
    dynamical_matrix=dynamical_matrix
    )

print('\nNumber of atoms N:', bulk_configuration.numberOfAtoms())
print(f'Expected IPR for delocalized states 1/N: {1 / bulk_configuration.
↪numberOfAtoms():.3f}')

# Obtain results
energies = vibrational_ipr.energies().inUnitsOf(meV)
ipr = vibrational_ipr.inverseParticipationRatio()
idx = numpy.argmax(ipr)
print(f'Mode at energy {energies.ravel()[idx]:.2f} meV',
      f'has the highest IPR: {ipr.ravel()[idx]:.3f}')
```

vibrational_ipr.py

Notes

The following definition is used to calculate the vibrational inverse participation ratio for a system of N atoms[1]

$$p_i^{(n)} = \frac{\sum_{i=1}^{N} |\mathbf{u}_i^{(n)}|^4}{\left(\sum_{i=1}^{N} |\mathbf{u}_i^{(n)}|^2\right)^2}$$

where $\mathbf{u}_i^{(n)}$ are the displacements eigenvectors for the n-th mode from the *DynamicalMatrix*. The inverse participation ratio is 1 for completely localized states and $1/N$ for completely delocalized states.

VibrationalMode

class VibrationalMode(*dynamical_matrix*, *configuration=None*, *kpoint_fractional=None*, *mode_indices=None*)

Class for calculating the vibrational modes of a configuration.

Parameters

- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use for the calculation.
- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration*) – The configuration for which the vibrational modes should be calculated. If no calculator is attached to the configuration, the calculator from dynamical_matrix is used. Currently, configuration must be the same as the configuration from dynamical_matrix and is therefore not needed. **Default:** The configuration from `dynamical_matrix`
- **kpoint_fractional** (*tuple of float*) – The kpoint (in fractional reciprocal space coordinates) for which the vibrational modes should be calculated. **Default:** The gamma point `(0.0, 0.0, 0.0)`
- **mode_indices** (*list of int*) – The indices of the modes (bands) to include. **Default:** All modes.

[1] S.ăI. Simdyankin, S.ăR. Elliott, Z.ăHajnal, T.ăA. Niehaus, and Th. Frauenheim. Simulation of physical properties of the chalcogenide glass As2S3 using a density-functional-based tight-binding method. *Physical Review B*, Apr 2004. URL: http://dx.doi.org/10.1103/PhysRevB.69.144202, doi:10.1103/physrevb.69.144202.

eigenvectors(*mode_index=None*)

Return the vibration eigenvectors, u, for the selected k-point and mode indices.

The eigenvectors are solutions to $Du = \omega^2 u$, where D is the dynamical matrix, and ω is the frequency. If `mode_index` is not specified all calculated modes are returned.

Note, the mode `u[:,i]` is the eigenvector corresponding to mode index `modeIndices()[i]`.

Parameters
mode_index (*int*) – Mode index. **Default:** All calculated modes.

Returns
Eigenvectors of the vibrational modes

Return type
`numpy.array`

evaluate(*mode_index=None*)

Return the vibrational energies at the selected mode index. If `mode_index` is not specified all mode energies are returned.

Parameters
mode_index (*int*) – Mode index. **Default:** All modes.

Returns
Energies of the vibrational modes.

Return type
PhysicalQuantity of type energy

kPoint()

Returns
The kpoint (in fractional reciprocal space coordinates).

Return type
tuple of float

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

mode(*mode_index*, *temperature=None*)

Return the vibrational mode with the specified mode_index. The eigenmode, v, is obtained from the eigenvectors, u, as $v = \frac{u}{\sqrt{M}} s$, where the scaling factor, $s = \frac{\sqrt{k_B T}}{\omega}$, gives the amplitude of a classical oscillator with frequency ω.

Parameters

- **mode_index** (*int*) – The mode index. Must be contained in the list returned by *modeIndices()*.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature used in scaling the amplitude. **Default:** `300 * Kelvin`

Returns
(*N*, 3) complex array

Return type
PhysicalQuantity of type length

modeIndices()

Returns
The mode indices.

Return type
list of int

movie(*mode_index*, *mode=None*, *temperature=None*, *number_of_frames=None*, *repeat=None*)

Method for generating a trajectory showing the vibrations with the specified `mode_index`.

Parameters

- **mode_index** (*int*) – The mode index. Must be contained in the list returned by *modeIndices()*.
- **mode** (*PhysicalQuantity* of type length) – The mode (*N*, 3) complex array.
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature used in scaling the amplitude. **Default:** `300 * Kelvin`
- **number_of_frames** (*int*) – The number of frames in the trajectory. **Default:** `20`
- **repeat** (*tuple of integers*) – The number of repetitions along A, B, and C axes. For molecules this must be `(1, 1, 1)`; for devices `(n, m, 1)`. **Default:** `(1, 1, 1)`

Returns
A trajectory.

Return type
Trajectory

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the AnalysisSpin object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

wavenumbers(*mode_index=None*)

Return vibrational wavenumbers. If `mode_index` is not specified all wavenumbers are returned.

Parameters
mode_index (*int*) – Mode index. **Default:** All modes.

Returns
Wavenumbers of the vibrational modes

Return type
PhysicalQuantity of type inverse length

Usage Examples

Calculate the vibrational eigenmodes of a methane molecule using the *BrennerCalculator*:

```
# ------------------------------------------------------------
# Molecule configuration, CH4
# ------------------------------------------------------------

# Define elements
elements = [Hydrogen, Carbon, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[ 11.61454603,  11.09244804,   9.99999993],
                         [ 10.67862603,  10.93144595,  10.52879495],
                         [  9.99999997,  11.75542099,  10.3236729 ],
                         [ 10.22904002,  10.00000001,  10.1942581 ],
                         [ 10.87074303,  10.87768505,  11.59731907]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
calculator = BrennerCalculator()

molecule_configuration.setCalculator(calculator)
molecule_configuration.update()

molecule_configuration = OptimizeGeometry(
        molecule_configuration,
        max_forces=0.001*eV/Ang,
        max_steps=200,
        max_step_length=0.5*Ang,
        trajectory_filename=None,
        optimize_cell=False,
        optimizer_method=LBFGS(),
        )

# ------------------------------------------------------------
# Vibrational modes
# ------------------------------------------------------------
# Compute Hessian by finite difference of forces.
dynamical_matrix = DynamicalMatrix(molecule_configuration, 'ch4.hdf5', object_id=
↪'dynamical_matrix')

# Vibrational analysis.
vib_mode = VibrationalMode(dynamical_matrix, mode_indices=range(6,15))
vib_mode.nlprint()
```

(continues on next page)

(continued from previous page)

```
vibrationalmode.py
```

Notes

- For a *MoleculeConfiguration*, the six lowest vibrational modes have zero-frequency. In the above example, these modes are not calculated since `mode_indices=range(6,15)`. The six zero-frequency modes correspond to three rigid translational modes and three rigid rotational modes.
- It is important to do a fine structual relaxation before calculating vibrational (or phonon) modes. This may also require specifying a higher `density_mesh_cutoff` in *NumericalAccuracyParameters*.

VirtualCrystalBasisSet

class VirtualCrystalBasisSet(*basis_sets*, *x=None*)

Class providing the virtual crystal approximation basis set.

Parameters

- **basis_sets** (tuple of two *BasisSet* instances) – The two basis sets to combine. The basis sets must have pseudopotentials with the same number of valence electrons.
- **x** (*float*) – The relative weight of the first element (between zero and one). **Default:** 0.5

angularMomenta()

Return the angular momenta of the basis set.

Returns
The angular momentum (azimuthal quantum number) for each orbital.

Return type
list

basisSets()

The constituent basis sets.

Returns
The two basis sets that make up the virtual crystal basis.

Return type
list of two *BasisSet* instances

dftHalfParameters()

The DFT-1/2 parameters.

Returns
The DFT-1/2 parameters.

Return type
DFTHalfParameters | Automatic | Disabled

element()

The element for this basis set.

Returns
The atomic element.

Return type
PeriodicTableElement

fillingMethod()

The method using for filling the orbitals.

Returns
The method for filling the orbitals.

Return type
SphericalSymmetric | Anisotropic

filterMeshCutoff()

The cutoff for the filter function.

Returns
The cutoff.

Return type
PhysicalQuantity of type energy

gramSchmidtOrthonormalization()

Whether the basis set has been orthonormalized.

Returns
True if the basis set is orthonormal.

Return type
bool

hubbardU()

The Hubbard U associated with the orbitals.

Returns
The Hubbard U energies either as a *PhysicalQuantity* with one element for each orbital.

Return type
PhysicalQuantity of type energy.

nlinfo()

Returns
Structured information about the virtual crystal basis set.

Return type
dict

numberOfValenceElectrons()

Function to get the number of electrons in total.

Returns
The total number of valence electrons.

Return type
int

occupations()

The occupations associated with the orbitals.

Returns
The occupations of the orbitals given as a list of non-negative floats.

Return type
list

`onsiteSpinOrbitSplit()`

List of spin-orbit splits per orbital.

Returns
Spin-orbit splittings with one element for each orbital.

Return type
PhysicalQuantity of type energy

`orbitalPermutationList()`

Get the orbital permutation list.

Returns
The orbital permutation list.

Return type
list

`orbitals()`

The orbitals that forms this basis set.

Returns
A list of orbital objects.

Return type
list

`projectorShift()`

A query method for the pseudopotential projector shift.

Returns
The projector shift energies specified for each angular momentum.

Return type
PseudoPotentialProjectorShift

`pseudopotential()`

The pseudopotential used in generating the basis set.

Returns
The pseudopotential.

Return type
NormConservingPseudoPotential

`uniqueString()`

Return a unique string representing the state of the object.

`x()`

The weight parameter for mixing the elements.

Returns
The weight parameter x.

Return type
float

Usage Examples

$Si_x Ge_{1-x}$ with a fixed value for x

```
# Set x for the alloy Si_x Ge_(1-x)
x = 0.39

# Lattice constant, from http://www.ioffe.ru/SVA/NSM/Semicond/SiGe/basic.html
a = 5.431 + 0.20*(1-x) + 0.027*(1-x)**2

# Setup a BulkConfiguration
lattice = FaceCenteredCubic(a*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define a basis set for Si.
SiliconBasis = LDABasis.Silicon_DoubleZetaPolarized

# Define a basis set for Ge.
GermaniumBasis = LDABasis.Germanium_DoubleZetaPolarized

# Define the virtual crystal basis set for the Si_x Ge_1-x system.
SiliconVirtualCrystalBasis = VirtualCrystalBasisSet(
    basis_sets=[SiliconBasis, GermaniumBasis],
    x=x)

basis_set = [SiliconVirtualCrystalBasis]

# Define a calculator and use the virtual crystal basis set.
k_point_sampling = MonkhorstPackGrid(
    na=5,
    nb=5,
    nc=5,
    )

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
```

(continues on next page)

(continued from previous page)

```
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

# Attach the calculator to the configuration.
bulk_configuration.setCalculator(calculator)

# Calculate the bandstructure.
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'X'],
    points_per_segment=40,
    bands_above_fermi_level=4)

# Calculate the direct and indirect gap.
direct_gap = bandstructure.directBandGap()
indirect_gap = bandstructure.indirectBandGap()

# Print the results.
print('%f\t%f\t%f' % (x, direct_gap.inUnitsOf(eV), indirect_gap.inUnitsOf(eV)))
```

$Si_x Ge_{1-x}$ with x ranging from 0 (pure Ge) to 1 (pure Si)

```
# Define a basis set for Si
SiliconBasis = LDABasis.Silicon_DoubleZetaPolarized
# Define a basis set for Ge
GermaniumBasis = LDABasis.Germanium_DoubleZetaPolarized

# Loop over Si concentration.
x_values = numpy.linspace(0.0, 1.0, 21)

for x in x_values:
    # Lattice constant for specific x. From http://www.ioffe.ru/SVA/NSM/Semicond/SiGe/
↪basic.html
    a = 5.431 + 0.20*(1-x) + 0.027*(1-x)**2

    # Setup a BulkConfiguration
    lattice = FaceCenteredCubic(a*Angstrom)

    # Define elements
    elements = [Silicon, Silicon]

    # Define coordinates
    fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                              [ 0.25,  0.25,  0.25]]

    # Set up configuration
    bulk_configuration = BulkConfiguration(
        bravais_lattice=lattice,
        elements=elements,
```

(continues on next page)

(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )

# Define the virtual crystal basis set for the Si_x Ge_1-x with the current x.
SiliconVirtualCrystalBasis = VirtualCrystalBasisSet(
    basis_sets=[SiliconBasis, GermaniumBasis],
    x=x)

basis_set = [SiliconVirtualCrystalBasis]

# Define a calculator.
k_point_sampling = MonkhorstPackGrid(
    na=5,
    nb=5,
    nc=5,
    )

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

# Attach the calculator to the configuration.
bulk_configuration.setCalculator(calculator)

# Calculate the bandstructure.
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'X'],
    points_per_segment=40,
    bands_above_fermi_level=4)

# Calculate the direct and indirect gap.
direct_gap = bandstructure.directBandGap()
indirect_gap = bandstructure.indirectBandGap()

# Print the results.
print('%f\t%f\t%f' % (x, direct_gap.inUnitsOf(eV), indirect_gap.inUnitsOf(eV)))
```

$Ga_x In_{1-x} As$ with x from 0 to 1

The following script sets up a calculation of the bandstructure of Ga(x) In(1-x) As as a function of the Ga concentration x. In order to obtain physically meaningful results, the spin-orbit interaction is included and the TB09 meta-GGA exchange-correlation potential is used. The spin-orbit coupling terms are contained in the pseudopotentials. Note that the pseudopotentials used in this script are not provided as part of QuantumATK. To run this script, the pseudopotentials have to be downloaded separately, e.g., from http://www.abinit.org/downloads/psp-links/hgh. Save the downloaded pseudopotentials in the same directory as the script.

```
# Setup a GaInAs calculation with spin-orbit coupling and TB09 meta-GGA.

#-------------------------------------------------------------------------------
# HGH Basis Set with manually set pseudopotential to include spin-orbit terms.
#-------------------------------------------------------------------------------
ga_basis = BasisLDAPZHGH.Gallium_13_Tier_4(pseudopotential=NormConservingPseudoPotential(
↪'31ga.13.hgh'))
in_basis = BasisLDAPZHGH.Indium_13_Tier_4(pseudopotential=NormConservingPseudoPotential(
↪'49in.13.hgh'))
as_basis = BasisLDAPZHGH.Arsenic_5_Tier_4(pseudopotential=NormConservingPseudoPotential(
↪'33as.5.hgh'))

#---------------------------------------
# Exchange-Correlation
#---------------------------------------
exchange_correlation = SOMGGA.TB09LDA

numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_cutoff=1e-06,
    k_point_sampling=(9, 9, 9),
    density_mesh_cutoff=300.0*Hartree)

iteration_control_parameters = IterationControlParameters(
    max_steps=100,
    tolerance=1e-06)

# Loop over Ga concentration x.
for x in numpy.linspace(0.0, 1.0, 21):
    # Lattice constant. From http://www.ioffe.rssi.ru/SVA/NSM/Semicond/GaInAs/basic.html
    a = 6.0583 - 0.405*x

    # ---------------------------------------------------------------
    # Bulk configuration
    # ---------------------------------------------------------------

    # Set up lattice
    lattice = FaceCenteredCubic(a*Angstrom)

    # Define elements
    elements = [Gallium, Arsenic]

    # Define coordinates
    fractional_coordinates = [[ 0.   ,  0.   ,  0.   ],
                              [ 0.25 ,  0.25 ,  0.25 ]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates)

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
# Define the virtual crystal basis set for the Ga/In system.
GalliumVirtualCrystalBasis = VirtualCrystalBasisSet(
    basis_sets=[ga_basis, in_basis],
    x=x)

# Define the complete basis set consisting of the Ga/In virtual crystal basis
# and the ordinary As basis.
basis_set = [GalliumVirtualCrystalBasis, as_basis]

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

nlsave('GaInAs_VCA.nc', bulk_configuration)

# ------------------------------------------------------------
# Bandstructure
# ------------------------------------------------------------
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['X', 'G', 'L'],
    points_per_segment=101,
    bands_above_fermi_level=All)

nlsave('GaInAs_VCA.nc', bandstructure)
```

Notes

When constructing a virtual crystal basis set in QuantumATK the following procedure is applied. The virtual crystal pseudopotential is obtained as a mixture of the two original pseudopotentials provided through the basis sets. The basis set orbitals will be the sum of the two original basis sets, each evaluated with their original pseudopotential. The total number of basis orbitals in a virtual crystal basis set will therefore always be larger than the individual basis sets, and calculations will be more time consuming.

Viscosity

class Viscosity(*md_trajectories*, *start_time=None*, *end_time=None*, *measurement_name='pressure_tensor'*, *info_panel=None*)

Class for calculating the shear viscosity from an *MDTrajectory*.

Parameters

- **md_trajectories** (*MDTrajectory* | list of type *MDTrajectory*) – The MDTrajectory the viscosity should be calculated for. This may be given as a single MDTrajectory or as a list of MDTrajectory objects.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last frame time.
- **measurement_name** (*str*) – The name of the measurement in the MDTrajectory that contains the pressure tensor data.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel

correlationFunction()

Returns
The shear pressure autocorrelation function.

Return type
PhysicalQuantity of type pressure**2

lagTimes()

Returns
The lag times of the shear pressure autocorrelation function.

Return type
PhysicalQuantity of type time

temperature()

Returns
The temperature of the MD simulation.

Return type
PhysicalQuantity of type temperature

uniqueString()

Return a unique string representing the state of the object.

viscosities()

Returns
An estimate of the viscosity using the Green-Kubo formalism as a function of lag time.

Return type
PhysicalQuantity of type time * pressure

viscositiesStandardError()

Returns
The standard error of the mean for the estimated viscosity.

Return type
PhysicalQuantity of type time * pressure

Usage Examples

This example runs a NVT molecular dynamics (MD) simulation of a Lennard-Jones fluid. The pressure tensor is recorded at each MD step, using the *MDMeasurement* measurement hook.

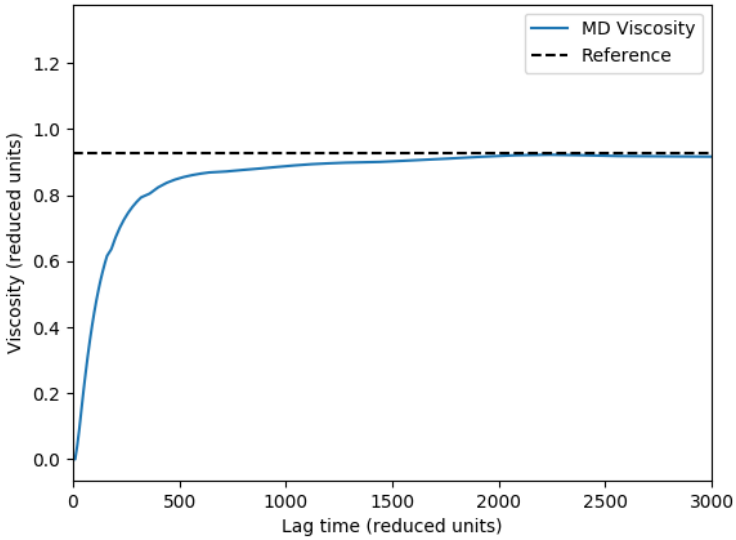
```
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=T,
    enforce_temperature=True,
    remove_center_of_mass_momentum=True
)

method = NVTNoseHoover(
    time_step=10.0*femtoSecond,
    initial_velocity=initial_velocity,
    thermostat_timescale=1*ps,
    reservoir_temperature=T,
)
constraints = [FixCenterOfMass()]

measurement_names = ['pressure_tensor']

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=constraints,
    trajectory_filename='ljmd.hdf5',
    steps=100000,
    log_interval=1000,
    measurement_hook=MDMeasurement(measurement_names, call_interval=1),
    method=method
)
```

The shear viscosity is then calculated from the MD trajectory.

```
viscosity = Viscosity(
    md_trajectory,
    start_time=10*ps,
    measurement_name='pressure_tensor',
)
nlsave('viscosity.hdf5', viscosity)
```

The script will produce a plot of the estimated viscosity as a function of the maximum lag time as well as a reference value taken from[1].

[1] R.ăKhordad. Viscosity of lennard–jones fluid: integral equation method. *Physica A: Statistical Mechanics and its Applications*, 387(18):4519 – 4530, 2008. URL: http://www.sciencedirect.com/science/article/pii/S0378437108003270, doi:https://doi.org/10.1016/j.physa.2008.03.025.



Notes

The viscosity is calculated from the shear pressure using the Green-Kubo relations.[2] In the Green-Kubo formalism, the viscosity is defined as,

$$\eta = \frac{V}{k_{\mathrm{B}}T}\int_0^\infty \langle P_{\alpha\beta}(\tau_0)P_{\alpha\beta}(\tau_0+\tau)\rangle \mathrm{d}\tau,$$

where V is the volume of the unit cell, k_{B} is the Boltzmann constant, T is the temperature, P is the pressure tensor with $P_{\alpha\beta}$ indicating the off-diagonal elements (shear), the angle brackets represent an ensemble average over trajectories with time origin τ_0, and τ is the lag time between two pressure tensors. In practice the MD simulations will be of finite length and the integral must be truncated at finite time.

- As the pressure tensor needs to be recorded with a high frequency (ideally every MD time step) and the trajectories are often on the time scale of 1 ns, the multiple-tau correlation technique is used to calculate the shear pressure autocorrelation function.[3]
- This analysis class is designed to be used with the *MDMeasurement* hook function. This allows for the pressure tensor to be recorded at each time step, without saving the full configuration to the *MDTrajectory*.
- The analysis class can analyze a single trajectory or a list of multiple trajectories. If multiple trajectories are given, then it is assumed that each trajectory is independent. Additionally, the standard error of the estimated viscosity, as a function of lag time, can be obtained by calling `viscositiesStandardError`.

[2] Dario Alfè and MichaelăJ. Gillan. First-principles calculation of transport coefficients. *Phys. Rev. Lett.*, 81:5161–5164, Dec 1998. URL: https://link.aps.org/doi/10.1103/PhysRevLett.81.5161, doi:10.1103/PhysRevLett.81.5161.

[3] Jorge Ramírez, SathishăK. Sukumaran, Bart Vorselaars, and AlexeiăE. Likhtman. Efficient on the fly calculation of time correlation functions in computer simulations. *The Journal of Chemical Physics*, 133(15):154103, 2010. URL: https://doi.org/10.1063/1.3491098, doi:10.1063/1.3491098.

VoidsizeDistribution

class VoidsizeDistribution(*md_trajectory*, *start_time=None*, *end_time=None*, *atom_selection=None*, *resolution=None*, *atomic_radii=None*, *time_resolution=None*, *info_panel=None*)

Calculate the voidsize distribution of an MD trajectory or a configuration.

Parameters

- **md_trajectory** (*MDTrajectory* | AtomicConfiguration) – The MDTrajectory or configuration to calculate the coordination number for.
- **start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`
- **end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.
- **atom_selection** (*PeriodicTableElement* | str | list | None) – Only include contributions between this selection of atoms. The atoms can be selected by element i.e. *PeriodicTableElement*, tag or a list of atomic indices. **Default:** All atoms.
- **resolution** (*PhysicalQuantity* of type length) – The bin size, which determines the resolution of the voidsize-histogram. **Default:** `2.0 * Angstrom`
- **atomic_radii** (*PhysicalQuantity* of type length) – The atomic radii used for calculating the atomic volumes, which do not contribute to th void volumes. **Default:** Covalent radii of the elements.
- **time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.
- **info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No InfoPanel.

data()

Returns
The voidsize-distribution as a histogram.

Return type
array

voidsizes()

Returns
The voidsize-diameters associated with the histogram.

Return type
PhysicalQuantity of type length

Usage Examples

Load an *MDTrajectory* and calculate the *VoidsizeDistribution*:

```
md_trajectory = nlread('alumina_trajectory.hdf5')[-1]

# Calculate the voidsize distribution of the trajectory.
vd = VoidsizeDistribution(md_trajectory)

# Get the time and autocorrelation values.
voidsizes = vd.voidsizes().inUnitsOf(Ang)
histogram = vd.data()
```

(continues on next page)

(continued from previous page)

```
# Plot the data using pylab.
import pylab

pylab.plot(
    voidsizes,
    histogram,
    label='Voidsize distribution'
)
pylab.xlabel('Voidsizes (Ang)')
pylab.ylabel('Distribution')
pylab.legend()

pylab.show()
```

voidsize_distribution.py

Notes

The voidsize distribution analysis calculates the average diameters of the volume of voids between atoms, primarily in disordered solid like glasses. The underlying algorithm first creates a Delaunay-triangulation of the selected coordinates, and then calculates the tetrahedral volume of each simplex. This volume is reduced by the overlap of the volumes of the adjacent vertices with the tetrahedron. The volumes of the vertices are calculated as spheres whose radii can either be specified as parameters, or are taken from the atomic covalent radii. Finally, the resulting void volumes are approximated by spheres with equivalent volume, and a histogram of the diameters of these spheres is used as a measure for the voidsize distribution.

This feature requires the **ATK-Classical** license feature.

WavefunctionOverlapsAndEnergyDifferences

class WavefunctionOverlapsAndEnergyDifferences(*wavefunction_overlaps*, *energy_differences*)

Constructor for WavefunctionOverlapAndEnergyDifferences.

Parameters

- **wavefunction_overlap** (*list of float*) – The wave function overlap.
- **energy_difference** (list of *PhysicalQuantity* of type energy) – The energy difference between the states in the overlap.

energyDifferences()

Returns
The energy difference between the states in the overlap.

Return type
list of *PhysicalQuantity* of type energy

uniqueString()

Return a unique string representing the state of the object.

wavefunctionOverlaps()

Returns
The wave function overlaps.

Return type
list of float

XRayScattering

class XRayScattering(*md_trajectory*, *start_time=None*, *end_time=None*, *pair_selection=None*, *maximum_q_value=None*, *q_resolution=None*, *cutoff_radius=None*, *resolution=None*, *time_resolution=None*, *info_panel=None*, *charges=None*, *parameters=None*)

Class for calculating the total X-ray scattering structure factor from an MD simulation.

Parameters

•**md_trajectory** (*MDTrajectory* | *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The MDTrajectory or configuration to calculate the X-ray scattering for.

•**start_time** (*PhysicalQuantity* of type time) – The start time. **Default:** `0.0 * fs`

•**end_time** (*PhysicalQuantity* of type time) – The end time. **Default:** The last time frame.

•**pair_selection** (*sequence*) – Only include contributions between this selection of atoms. A sequence has to contain two of the following types: Element, tag name, list of indices, or None. **Default:** all atoms pairs.

•**maximum_q_value** (*PhysicalQuantity* of type inverse length) – The maximum scattering vector length. **Default:** `15.0 / Angstrom`

•**q_resolution** (*PhysicalQuantity* of type inverse length) – The resolution of the X-ray structure factor curve. **Default:** `0.05/Angstrom`.

•**cutoff_radius** (*PhysicalQuantity* of type length) – Upper limit on sampled distances. **Default:** Half the diagonal of the unit.

•**resolution** (*PhysicalQuantity* of type length) – The bin size determining the resolution of the underlying radial-distribution-function. **Default:** `0.05 * Angstrom`

•**time_resolution** (*PhysicalQuantity* of type time) – The time interval between snapshots in the MD trajectory that are included in the analysis.

•**info_panel** (*InfoPanel (Plot2D)*) – Info panel to show the calculation progress. **Default:** No info panel.

•**charges** (*dictionary*) – Which charge state to use for the elements. **Default:** Charge state 0 for all elements.

•**parameters** (*nested dictionary*) – Parameters for the analytic approximation of scattering factors. **Default:** Parameters taken from P. J. Brown, A. G. Fox, E. N. Maslen, M. A. OKeefe and B. T. M. Willis. International Tables for Crystallography (2006). Vol. C, ch. 6.1, pp. 554-595 [doi:10.1107/97809553602060000600]

data()

Returns
The X-ray scattering structure factor.

Return type
numpy.ndarray

qRange()

Returns
The list of scattering vector magnitudes.

Return type
PhysicalQuantity of type inverse length

Usage Examples

Load an *MDTrajectory* and calculate the X-ray scattering structure factor using theoretically determined parameters approximating atomic scattering factors of the different elements (see Ref.[1]):

```
parameters = {
    'O': {
        '0': [3.0, 13.2, 2.2, 5.7, 1.5, 0.3, 0.8, 32.9, 0.2],
        '1-': [4.1, 12.8, 1.6, 4.1, 1.53, 47.0, -20.3, -0.0, 21.9],
    },
    'Si': {
        '0': [5.6, 2.6, 3.0, 38.6, 2.6, 0.9, 1.3, 93.5, 1.2],
        '4+': [4.4, 1.6, 3.2, 3.4, 1.1, 0.2, 0.4, 6.6, 0.7],
    },
}

charges = {
    'O': '0',
    'Si': '0',
}

md_trajectory = nlread('SiO2simulation_traj.hdf5', MDTrajectory,)[0]

xray_scattering = XRayScattering(
    md_trajectory,
    maximum_q_value=15.0 * Angstrom**-1,
    q_resolution=0.05 * Angstrom**-1,
    cutoff_radius=10 * Angstrom,
    time_resolution=399*fs,
    charges=charges,
    parameters=parameters
)

# Get the q-values and the X-ray structure function.
q_values = xray_scattering.qRange()
s_q      = xray_scattering.data()

model = Plot.PlotModel(x_unit=Angstrom**-1)
model.title().setText('X-ray scattering')
model.xAxis().setLabel('q')
model.yAxis().setLabel('S(q)')
model.legend().setVisible(True)

line = Plot.Line(q_values, s_q)
line.setLabel('Amorphous hafnia')
model.addItem(line)

# Set the ranges and show the plot
model.setLimits()
Plot.show(model)
```

[1] P.ăJ. Brown, A.ăG. Fox, E.ăN. Maslen, M.ăA. OKeefe, and B.ăT.ăM. Willis. Intensity of diffracted intensities. In *International Tables for Crystallography, Set, Volumes A - G, OnlineMRW*, pages 554–595. John Wiley & Sons, Ltd, 2006. URL: https://onlinelibrary.wiley.com/doi/abs/10.1107/97809553602060000600 (visited on 2023-07-27), doi:10.1107/97809553602060000600.

`xray_scattering.py SiO2simulation_traj.hdf5`

Notes

This object calculates the isotropic (i.e. averaged over all scattering angles) X-ray scattering structure factor. It is therefore predominantly aimed at the characterization of amorphous materials.

The X-ray scattering structure factor is calculated as described in Ref.[2]:

$$S(q) = \frac{\sum_{\alpha,\beta} f_\alpha(q) f_\beta(q) (c_\alpha c_\beta)^{1/2} S_{\alpha,\beta}(q)}{\sum_\alpha f(q)_\alpha^2 c_\alpha},$$

where c_α and $f_\alpha(q)$ are the mole fraction and scattering factor of element α.

$$f_\alpha(q) = \sum_{i=1}^{4} a_{\alpha,i} \exp\left[-b_{\alpha,i}(q/4\pi)^2\right] + c_\alpha,$$

where $a_{\alpha,i}$, $b_{\alpha,i}$ and c_α are scattering parameters.

The *PartialStructureFactor* is calculated from the partial radial distribution functions, $g_{\alpha,\beta}(r)$, as follows,

$$S_{\alpha,\beta}(q) = \delta_{\alpha,\beta} + 4\pi\rho(c_\alpha c_\beta)^{1/2} \int_0^{r_{max}} r^2 [g_{\alpha,\beta}(r) - 1] \frac{\sin(qr)}{qr} \frac{\sin(\pi r_{max} r)}{\pi r_{max} r} dr.$$

If no scattering parameters are provided by the user, the (default) parameters are taken from[Page 4124, 1]. This parameter set contains several charge states for some elements. By default all elements are considered to be charge neutral. The user may select a different charge state using a dictionary as given in the example above.

If a single pair of elements, or a selection via indices or tags, is specified in the `pair_selection` parameter, the *PartialStructureFactor* of the selected pair is returned.

The `cutoff_radius` parameter determines the range of the radial distribution function. This parameter might have an influence on the calculated structure factor at small scattering vectors.

ZBLPotential

class ZBLPotential(*particleType1*, *particleType2*, *A*, *b*, *r_i=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **A** (*PhysicalQuantity of type energy * length*) – Potential parameter.
- **b** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_i** (*PhysicalQuantity of type length*) – The inner cutoff radius where the smoothing of the potential starts. r_i must be smaller than r_cut.
- **r_cut** (*PhysicalQuantity of type length*) – The cutoff radius of this potential.

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

[2] G.ăGutierrez and B.ăJohansson. Molecular dynamics study of structural properties of amorphous Al2O3. *Phys. Rev. B*, 65:104202, Feb 2002. doi:10.1103/PhysRevB.65.104202.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Buckingham-type potential for titanium oxide with additional ZBL repulsive term between titanium and oxygen atoms by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# Set up lattice
lattice = SimpleTetragonal(4.593*Angstrom, 2.959*Angstrom)

# Define elements
elements = [Titanium, Titanium, Oxygen, Oxygen, Oxygen, Oxygen]

# Define coordinates
fractional_coordinates = [[ 0.    ,  0.    ,  0.    ],
                          [ 0.5   ,  0.5   ,  0.5   ],
                          [ 0.3051,  0.3051,  0.    ],
                          [ 0.6949,  0.6949,  0.    ],
                          [ 0.8051,  0.1949,  0.5   ],
                          [ 0.1949,  0.8051,  0.5   ]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ----------------------------------------------------------------
# Calculator
# ----------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='Trinastic_HfOSiTaTi_2013')
potentialSet.addParticleType(ParticleType(
    symbol='O',
    mass=15.9994*atomic_mass_unit,
    charge=-1.2
```

(continues on next page)

(continued from previous page)

```
))
potentialSet.addParticleType(ParticleType(
    symbol='Ti',
    mass=47.867*atomic_mass_unit,
    charge=2.4
))

potential = BuckinghamPotential(
    particleType1='O',
    particleType2='O',
    A=1388.77*eV,
    rho=0.3623*Angstrom,
    r_i=6.0*Angstrom,
    r_cut=7.5*Angstrom
)
potentialSet.addPotential(potential)
potential = LennardJonesMNPotential(
    particleType1='O',
    particleType2='O',
    A=0.0*Angstrom**12*eV,
    B=175.0*eV*Angstrom**6,
    m=12.0,
    n=6.0,
    r_cut=10.0*Angstrom
)
potentialSet.addPotential(potential)
potential = BuckinghamPotential(
    particleType1='O',
    particleType2='Ti',
    A=5105.12*eV,
    rho=0.2253*Angstrom,
    r_i=6.0*Angstrom,
    r_cut=7.5*Angstrom
)
potentialSet.addPotential(potential)
potential = LennardJonesMNPotential(
    particleType1='O',
    particleType2='Ti',
    A=0.0*Angstrom**12*eV,
    B=20.0*eV*Angstrom**6,
    m=12.0,
    n=6.0,
    r_cut=10.0*Angstrom
)
potentialSet.addPotential(potential)
potential = MorsePotential(
    particleType1='O',
    particleType2='Ti',
    r_0=1.96*Angstrom,
    k=1.9*1/Angstrom,
    E_0=0.3478*eV,
    r_i=6.5*Angstrom,
```

(continues on next page)

(continued from previous page)

```
    r_cut=7.0*Angstrom
)
potentialSet.addPotential(potential)

# Add the ZBL-potential
# Calculate the ZBL-prefactor A=e**2/(4*pi*eps_0)*Zi*Zj
ZBL_prefactor = elementary_charge**2/(4.0*numpy.pi*vacuum_permitivity)*Titanium.
↪atomicNumber()*Oxygen.atomicNumber()
ZBL_length = 0.8854*Bohr/(Titanium.atomicNumber()**0.23 + Oxygen.atomicNumber()**0.23)
potential = ZBLPotential(
    particleType1='Ti',
    particleType2='O',
    A=ZBL_prefactor,
    b=ZBL_length,
    r_i=0.7*Angstrom,
    r_cut=2.0*Angstrom
)
potentialSet.addPotential(potential)

potentialSet.setCoulombSolver(CoulombDSF(r_cut=9.0*Angstrom, alpha=0.2))
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This potential implements the universal *Ziegler-Biersack-Littmark (ZBL)* repulsive potential[1]

$$V^{\mathrm{ZBL}} = \sum_i \sum_{j>i} \frac{1}{4\pi\epsilon_0} \frac{Z_i Z_j}{r_{ij}} \phi\left(\frac{r_{ij}}{a_{ij}}\right)$$

where

$$\phi(x) = 0.1818e^{-3.2x} + 0.5099e^{-0.9423x} + 0.2802e^{-0.4029x} + 0.02817e^{-0.2016x}$$

and

$$a_{ij} = \frac{0.8854a_0}{Z_i^{0.23} + Z_j^{0.23}}\,.$$

In TremoloX, the ZBL-potential between a pair of particle types *i* and *j* is implemented as follows

$$V_{ij}^{\mathrm{ZBL}}(r_{ij}) = \frac{A_{ij}}{r_{ij}} \phi\left(\frac{r_{ij}}{b_{ij}}\right)$$

where the constructor arguments **A** and **b** correspond to A_{ij} and b_{ij}.

[1] JamesăF. Ziegler and JochenăP. Biersack. *Treatise on Heavy-Ion Science: Volume 6: Astrophysics, Chemistry, and Condensed Matter*, chapter The Stopping and Range of Ions in Matter, pages 93–129. Springer US, Boston, MA, 1985. doi:10.1007/978-1-4615-8103-1_3.

To obtain the original ZBL-potential, set

$$A_{ij} = \frac{Z_i Z_j}{4\pi\epsilon_0}$$

and

$$b_{ij} = a_{ij} = \frac{0.8854 a_0}{Z_i^{0.23} + Z_j^{0.23}} .$$

At long distances this potential is brought smoothly to zero between the inner cutoff `r_i` and the outer cutoff `r_cut`, using a 5th order spline function. This ensures that there are no discontinuites in the forces as atoms are brought closer together, and that the energy is properly conserved. The applied potential $U(r)$ is given as:

$$U(r) = V_{ij}(r) \times S(r)$$

where $V_{ij}(r)$ is the pair potential and $S(r)$ is the spline function. The values of the spline function are:

- 1 when $r \leq r_i$
- In the range $[0, 1]$ when $r_i < r < r_{cut}$
- 0 when $r \geq r_{cut}$

ZBLStiwePotential

class ZBLStiwePotential(*particleType1*, *particleType2*, *f*, *A*, *B*, *p*, *gamma*, *C*, *d*, *r_f=None*, *r_cut=None*)

Constructor of the potential.

Parameters

- **particleType1** (ParticleType *or* ParticleIdentifier) – Identifier of the first particle type.
- **particleType2** (ParticleType *or* ParticleIdentifier) – Identifier of the second particle type.
- **f** (*PhysicalQuantity of type length**-1*) – Potential parameter.
- **A** (*PhysicalQuantity of type energy*) – Potential parameter.
- **B** (*PhysicalQuantity of type length**p*) – Potential parameter.
- **p** (*int*) – Potential parameter.
- **gamma** (*PhysicalQuantity of type length*) – Potential parameter.
- **C** (*PhysicalQuantity of type energy * length*) – Potential parameter.
- **d** (*PhysicalQuantity of type length*) – Potential parameter.
- **r_f** (*PhysicalQuantity of type length*) – Radius in the Fermi function.
- **r_cut** (*PhysicalQuantity of type length*) – Cutoff radius

classmethod getAllParameterNames()

Return the names of all used parameters as a list.

getAllParameters()

Return all parameters of this potential and their current values as a <parameterName / parameterValue> dictionary.

static getDefaults()

Get the default parameters of this potential and return them in form of a dictionary of <parameter name, default value> key-value pairs.

getParameter(*parameterName*)

Get the current value of the parameter parameterName.

setParameter(*parameterName*, *value*)

Set the parameter parameterName to the given value.

Parameters

- **parameterName** (*str*) – The name of the parameter that will be modified.
- **value** – The new value that will be assigned to the parameter parameterName.

Usage Examples

Define a Stillinger-Weber potential with additional ZBL repulsive term for silicon by adding particle types and interaction functions to the *TremoloXPotentialSet*.

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = TremoloXPotentialSet(name='StillingerWeber_Si_1985')
potentialSet.addParticleType(ParticleType(
    symbol='Si',
    mass=28.0855*atomic_mass_unit,
    atomicNumber=14
))

# Calculate the ZBL-prefactor A=e**2/(4*pi*eps_0)*Zi*Zj
ZBL_prefactor = elementary_charge**2/(4.0*numpy.pi*vacuum_permitivity)*14**2

# Calculate the ZBL-length 0.8854*a_0/(Zi*'0.23 + Zj*0.23)
```

(continues on next page)

(continued from previous page)

```
ZBL_length = 0.8854*Bohr/(14**0.23 + 14**0.23)

potential = ZBLStiwePotential(
    particleType1='Si',
    particleType2='Si',
    p=4.0,
    A=15.2855528754*eV,
    B=11.6031922834*Angstrom**4,
    gamma=2.0951*Angstrom,
    C=ZBL_prefactor,
    d=ZBL_length,
    f=14.0*Angstrom**-1,
    r_f=0.95*Angstrom,
    r_cut=3.77118*Angstrom
)
potentialSet.addPotential(potential)
potential = Stiwe3Potential(
    particleType1='Si',
    particleType2='Si',
    particleType3='Si',
    gamma0=2.51412*Angstrom,
    gamma1=2.51412*Angstrom,
    l=45.5343*eV,
    cosTheta0=-0.333333333333,
    type=1,
    r_0=3.77118*Angstrom,
    r_1=3.77118*Angstrom,
    r_13=-1.0*Angstrom
)
potentialSet.addPotential(potential)
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setInternalOrdering("default")
calculator.setVerletListsDelta(0.25*Angstrom)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
```

Notes

This potential can be used to combine a Stillinger-Weber potential with the universal *Ziegler-Biersack-Littmark (ZBL)* repulsive potential V^{ZBL}[1].

To do so, the *ZBLStiwePotential* must be used instead of the normal *Stiwe2Potential*.

The ZBL-potential and the two-body Stillinger-Weber potential v_2 are then mixed as follows

$$\overline{v}_2(r) = (1 - F(r))V^{\mathrm{ZBL}}(r) + F(r)v_2(r)$$

where $F(r)$ is the Fermi function:

$$F_{ij}(r) = \frac{1}{1 + \exp(-f_{ij}(r - r_f))} .$$

[1] JamesăF. Ziegler and JochenăP. Biersack. *Treatise on Heavy-Ion Science: Volume 6: Astrophysics, Chemistry, and Condensed Matter*, chapter The Stopping and Range of Ions in Matter, pages 93–129. Springer US, Boston, MA, 1985. doi:10.1007/978-1-4615-8103-1_3.

Note, that in this class the constructor arguments **C** and **d** correspond to A_{ij} and b_{ij}.

ZeroVelocities

class ZeroVelocities

Utility for using zero velocities as the starting value in a *MolecularDynamics()* simulation.

uniqueString()

Return a unique string representing the state of the object.

Usage Examples

Perform a molecular dynamics run of a silicon crystal, with the initial velocities of all particles set to zero.

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[-0.009580233862,  0.002314591627,  0.010466101485],
                          [ 0.259580233862,  0.247685408373,  0.239533898515]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

potentialSet = Tersoff_Si_1988b()
calculator = TremoloXCalculator(parameters=potentialSet)
bulk_configuration.setCalculator(calculator)

# The MD method will use initial velocities corresponding to zero.
method = NVEVelocityVerlet(initial_velocity=ZeroVelocities())

md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=500,
    log_interval=100,
    method=method
)
```

zero_velocities.py

Notes

Use this object as `initial_velocity` parameter in the MD method (e.g. *NVEVelocityVerlet*), when you want to run an MD simulation with zero initial velocities. This has the same effect as setting `initial_velocity` to *None* in the MD method.

Functions

DoubleContourIntegralParameters

DoubleContourIntegralParameters(*integral_lower_bound=None*, *circle_eccentricity=None*, *logarithmic_bunching=None*, *circle_points=None*, *real_axis_point_density=None*, *real_axis_infinitesimal=None*, *real_axis_kbt_padding_factor=None*, *fermi_line_points=None*, *fermi_function_poles=None*)

Deprecated: Use ContourParameters instead.

Representation of the contour parameters when using a double contour.

Deprecated: from v2016.0, use *ContourParameters* instead.

Parameters

- **integral_lower_bound** (*PhysicalQuantity* of type energy) – The distance between the lowest Fermi-level to the lowest energy circle contour point. **Default:** `1.5 * Hartree`
- **circle_eccentricity** (*float*) – The eccentricity of the circle contour. This should be a float between 0 and 1. 0 is a circle, 1 is a line. **Default:** `0.3`
- **logarithmic_bunching** (*float*) – Logarithmic bunching of the circle contour around the Fermi level. This should be a float between 0 and 1. 0 means no bunching, equidistant points, 1 means all bunched, centred on the Fermi Level. **Default:** `0.3`
- **circle_points** (*int*) – The number of points on the circle contour. Must be at least 3. **Default:** `30`
- **real_axis_point_density** (*PhysicalQuantity* of type energy) – Spacing between the points on bias window integration line close to the real axis. **Default:** `0.001 * Hartree`
- **real_axis_infinitesimal** (*PhysicalQuantity* of type energy) – Small imaginary shift of the bias window integration line from the real axis. **Default:** `0.001 * Hartree`
- **real_axis_kbt_padding_factor** (*float*) – The integration line is extended with the amount fk_bT below and above the electrode Fermi levels where f is this value. Must be larger than 0.0. **Default:** `5.0`
- **fermi_line_points** (*int*) – The number of points on the straight line from the Fermi energy level up to infinity. This should be an integer in the range 1 to 11 (inclusive). **Default:** `10`
- **fermi_function_poles** (*int*) – The number of poles of the Fermi function to include. Determines the imaginary shift of the straight line from the Fermi level to infinity. Must be positive. **Default:** `8`

Returns
A *ContourParameters* object consisting of the *SemiCircleContour* equilibrium contour and the *RealAxisContour* Non-equilibrium contour. Both left and right are set.

Return type
ContourParameters

Notes

This class is deprecated and only included for backwards compatibility.

A *DoubleContourIntegralParameters* object:

```
contour_parameters = DoubleContourIntegralParameters(
    integral_lower_bound=5.0*Hartree,
    circle_eccentricity=0.5,
    logarithmic_bunching=0.4,
    circle_points=30,
    real_axis_point_density=0.002*eV,
    real_axis_infinitesimal=1e-05*eV,
    real_axis_kbt_padding_factor=0.6,
    fermi_line_points=10,
    fermi_function_poles=8)
```

is equivalent to the following *ContourParameters* object

```
eq_contour = SemiCircleContour(
    integral_lower_bound=5.0*Hartree,
    circle_eccentricity=0.5,
    logarithmic_bunching=0.4,
    circle_points=30
    fermi_line_points=10,
    fermi_function_poles=8)

neq_contour = RealAxisContour(
    real_axis_point_density=0.002*eV,
    real_axis_infinitesimal=1e-05*eV
    real_axis_kbt_padding_factor=0.6)

contour_parameters = ContourParameters(
    equilibrium_contour=eq_contour,
    non_equilibrium_contour=neq_contour,
    method=DoubleContour())
```

EstimateBasisSetMeshCutoff

EstimateBasisSetMeshCutoff(*basis_set*, *exchange_correlation=None*, *kinetic_energy_tolerance=None*)

Function for estimating the mesh cutoff required for a basis set

Parameters

- **basis_set** (*a BasisSet object*) – The BasisSet to be analyzed
- **exchange_correlation** (*ExchangeCorrelation*) – The choice of Exchange-Correlation for this calculation. **Default:** The exchange-correlation of the given basis set.
- **kinetic_energy_tolerance** (*PhysicalQuantity* of type energy) – Specify how much the kinetic energy may change for the calculated cutoff. This energy must be positive. **Default:** `0.0015 * Hartree`

Returns

Estimated mesh cutoff for each basis orbital

Return type
PhysicalQuantity of type energy

EstimatePseudoPotentialMeshCutoff

EstimatePseudoPotentialMeshCutoff(*pseudopotential*, *exchange_correlation=None*, *kinetic_energy_tolerance=None*)

Function for estimating the mesh cutoff required for a pseudopotential

Parameters

- **pseudopotential** (*NormConservingPseudoPotential*) – The pseudopotential to be analyzed
- **exchange_correlation** (*ExchangeCorrelation*) – The choice of Exchange-Correlation for this calculation. **Default:** The exchange-correlation of the given pseudopotential
- **kinetic_energy_tolerance** (*PhysicalQuantity* of type energy) – Specify how much the kinetic energy may change for the calculated cutoff. This energy must be positive. **Default:** `0.0015 * Hartree`

Returns
Estimated mesh cutoff for various pseudopotential quantities. The keys in this dictionary are strings and all values are *PhysicalQuantity* objects of type energy.

Return type
dict

HybridMonteCarlo

HybridMonteCarlo(*configuration*, *trajectory_filename=None*, *steps=None*, *log_interval=None*, *mc_method=None*, *md_method=None*, *hook_functions=None*, *write_velocities=True*, *write_forces=True*, *write_stresses=None*, *trajectory_interval=None*, *trajectory_object_id=None*, *constraints=None*, *pre_step_hook=None*, *post_step_hook=None*, *measurement_hook=None*)

Function for performing a hybrid Monte Carlo molecular dynamics simulation.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The initial configuration.
- **trajectory_filename** (*str | None*) – The filename of the file to be used for storing the trajectory, or None if no trajectory should be written. A trajectory filename should not be given if *configuration* is a trajectory. **Default:** `None`.
- **steps** (*int*) – The number of time-steps to take in simulation. **Default:** `50`.
- **log_interval** (*int*) – The interval at which information, such as time, energy, temperature, etc. is written to the log output. **Default:** 1.
- **mc_method** (`MCMethod`) – The Monte Carlo method used for the simulation. **Default:** `MCMethod`.
- **md_method** (`BaseMDmethod` | *ForceBiasMonteCarlo*) – The molecular dynamics or FBMC method used for the simulation. **Default:** *NVEVelocityVerlet*.
- **hook_functions** (*HookFunctions*) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the forces evaluation. It can also contain measurement hooks, which are called at the end of each configuration move step. **Default:** `None`.
- **write_velocities** (*bool*) – Write the velocities to the trajectory file every `log_interval` steps. **Default:** True.
- **write_forces** (*bool*) – Write the forces to the trajectory file every `log_interval` steps. **Default:** True.

- **write_stresses** (*bool*) – Write the stress to the trajectory file every `log_interval` steps. A value of None means that `write_stresses` is True by default for NPT methods and False otherwise. This is to avoid the additional work of calculating the stress when it is not needed. **Default:** None.
- **trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If `configuration` is a trajectory (i.e. this is a restart MD simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** The same value as `log_interval`.
- **trajectory_object_id** (*str | None*) – The object id of the trajectory written to `trajectory_filename`. If a value of `None` is given, then an object id will be chosen automatically. **Default:** `None`.
- **constraints** (list of ints | list of `BaseConstraint`) – The constraints in the system.
- **pre_step_hook** (*function | list of functions | None*) – A function to be called before each simulation step.
- **post_step_hook** (*function | list of functions | None*) – A function to be called after each simulation step.
- **measurement_hook** (*function | list of functions | None*) – A function to be called at the end of each simulation step, after all constraints have been applied.

Returns
The molecular dynamics trajectory.

Return type
MDTrajectory

Notes

The *HybridMonteCarlo()* method implements the hybrid Monte Carlo simulation method. In this method the equilibrium properties of a system in a thermodynamic ensemble are sampled. Samples are generated using alternating dynamical (molecular dynamics or force-bias Monte Carlo) and stochastic (Monte Carlo) cycles. The method works similarly to *MolecularDynamics()* in that it takes a starting configuration as input and then generates a trajectory of samples in the given thermodynamic ensemble. As in some ensembles used in Monte Carlo the number or type of atoms can change, HybridMonteCarlo generates a *ConfigurationDataContainer* rather than a *MDTrajectory* as an output trajectory. In hybrid Monte Carlo simulations the properties being sampled are written to the trajectory as measurements.

The dynamics method is specified using the `method` keyword. This works in the same way as in *MolecularDynamics()* or *TimeStampedForceBiasMonteCarlo()*. For pure Monte Carlo simulations where no dynamics are performed a *StaticMonteCarlo* object can also be given. In hybrid Monte Carlo it is assumed that the dynamics method is simply sampling the same ensemble as the Monte Carlo method. Dynamics steps can therefore be uniformly accepted because the ensemble is maintained.

The Monte Carlo method is specified as a pre-step hook method. Currently only the semi-grand canonical ensemble can be sampled using the *SemiGrandCanonicalMonteCarloHook* and *SemiGrandCanonicalTransitionMatrixHook* methods.

Hybrid Monte Carlo is more limited in features than regular molecular dynamics. Hybrid Monte Carlo does not support distributed parallel calculations with MPI. It also does not support adding constraints to the dynamics of the system.

MakeTrajectory

MakeTrajectory(*configuration_list*, *analysis=None*, ***kwargs*)

Construct a trajectory sequence of configurations, from a given nudged elastic band configuration, or a list of configurations.

Parameters

- **configuration_list** (list of configurations | *NudgedElasticBand*.) – Either an explicit list of configurations (each of type *MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*), or a *NudgedElasticBand* configuration object.
- **analysis** (*class*) – The analysis class that will be applied across the configurations, e.g. `Forces`. **Default:** No analysis is performed.
- ****kwargs** – Any additional keyword arguments will be passed to the constructor of the class given in the `analysis` argument.

Usage Examples

Create a Trajectory object from a list of configurations:

```
# Define elements.
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates.
c0 = [[  0.00,   0.00,   0.12],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

c1 = [[  0.00,   0.00,   0.20],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

c2 = [[  0.00,   0.00,   0.28],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

c3 = [[  0.00,   0.00,   0.36],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

# Define the coordinate list.
coordinates = [c0, c1, c2, c3]

# Set up a list of configurations.
configuration_list = [ MoleculeConfiguration(elements,c) for c in coordinates ]

# Create a Trajectory only - no analysis.
t = MakeTrajectory(configuration_list)

# Define calculators on each configuration - for analysis.
for c in configuration_list:
    c.setCalculator(LCAOCalculator())
```

(continues on next page)

(continued from previous page)

```
# Create a Trajectory with a TotalEnergy analysis carried out on each configuration.
t = MakeTrajectory(configuration_list, TotalEnergy)
```

maketrajectory1.py

Create a *Trajectory* object from a *NudgedElasticBand* object:

```
# Define elements.
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates.
c0 = [[  0.00,   0.00,   0.12],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

c1 = [[  0.00,   0.00,   1.20],
      [  0.00,   0.76,  -0.48],
      [  0.00,  -0.76,  -0.48]]*Angstrom

# Set up a list of configurations.
configuration_list = [ MoleculeConfiguration(elements,c) for c in [c0,c1] ]

# Create the NudgedElasticBand object.
neb = NudgedElasticBand(configuration_list)

# Set a calculator on the configurations in the NEB - for analysis.
neb.setCalculator(LCAOCalculator())

# Create a Trajectory with a TotalEnergy analysis carried out on each configuration.
t = MakeTrajectory(neb, TotalEnergy)
```

maketrajectory2.py

MemoryUsage

MemoryUsage(*configuration*, *stream=None*)

Routine for printing an estimate of the memory usage of a calculation.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration*) – The configuration with attached calculator of type Huckel, SlaterKoster or LCAO.

Returns
A dictionary with the different terms.

Return type
dict

MetropolisMonteCarlo

MetropolisMonteCarlo(*configuration*, *constraints=None*, *trajectory_filename=None*, *steps=None*, *max_trial_moves=None*, *log_interval=None*, *method=None*, *random_seed=None*, *trajectory_interval=None*, *trajectory_object_id=None*)

Perform a Metropolis Monte Carlo simulation on a configuration using a specified method for carrying out each Monte Carlo trial move.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | *MDTrajectory* | *Trajectory*) – The initial configuration or a trajectory. If a trajectory is given, the last configuration will be used as the initial configuration for this MC run and the trajectory will be appended to. When a trajectory is used as input, the arguments `trajectory_interval` and `trajectory_object_id` should be *None* `.
- **constraints** (int | *FixAtomConstraints* | list of int | list of *FixAtomConstraints*) – The list of atomic constraints. Note that for the *ContinuousRandomNetwork* these constraints are used only for fixing the bonding network (i.e., any atom listed here will not be allowed to take part in a bond switch move, hence fixing its neighbours). The coordinates of the atoms are however not constrained in the geometry optimization following a bond switch. If this further constraint is desired, it should be introduced in the `optimize_geometry_parameters` input of the *ContinuousRandomNetwork* object. **Default:** No constraints.
- **trajectory_filename** (*str*) – The filename of the file to be used for storing the trajectory. This should not be given if *configuration* is a trajectory. **Default:** No trajectory will be written, unless `configuration` is a trajectory.
- **steps** (*int*) – The number of steps to take in the simulation. A smaller number might be taken if *max_trial_moves* is reached before the target number of steps. **Default:** `50`
- **max_trial_moves** (*int*) – The maximum number of trial moves (accepted or rejected) to take in the simulation. **Default:** *100 * steps*
- **log_interval** (*int*) – The resolution used in writing the log file, where the given number represents the number of steps performed between each each log file entry. **Default:** `1`
- **method** (`BaseMetropolisMonteCarloMethod`) – The method used to take the Monte Carlo step. **Default:** `ContinuousRandomNetwork()`
- **random_seed** (*int*) – The random seed used for the Metropolis algorithm. **Default:** Generated automatically.
- **trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If *configuration* is a trajectory (i.e., this is a restart MC simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** Equal to *log_interval*.
- **trajectory_object_id** (*str*) – The object ID of the trajectory written to *trajectory_filename*. This should not be given if `configuration` is a trajectory. **Default:** Chosen automatically.

Returns
The Monte Carlo trajectory.

Return type
MDTrajectory | *Trajectory*

Notes

The *MetropolisMonteCarlo()* method implements the Metropolis Monte Carlo scheme for sampling thermodynamic or structural properties. The method takes a class that specifies a set of moves that define the type of Monte Carlo simulation. These moves are randomly trialed, with each move being accepted or rejected stochastically based on the change in energy relative to the given temperature. In this way a non-autocorrelated sampling of the initial configuration, *Trajectory* or *MDTrajectory* is generated.

One example of a Monte Carlo model that can be generated with the *MetropolisMonteCarlo()* is the *ContinuousRandomNetwork* model. This model defines moves that alter the bonding of an input configuration, allowing for the generation of amorphous materials.

MolecularDynamics

MolecularDynamics(*configuration*, *constraints=None*, *trajectory_filename=None*, *steps=None*, *log_interval=None*, *method=None*, *xyz_filename=None*, *hook_functions=None*, *write_velocities=True*, *write_forces=True*, *write_stresses=None*, *domain_decomposition_pattern=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *trajectory_interval=None*, *trajectory_object_id=None*, *scf_initial_guess=None*, *pre_step_hook=None*, *post_step_hook=None*, *measurement_hook=None*)

Function for performing a molecular dynamics simulation.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | *MDTrajectory*) – The initial configuration or a trajectory. If a trajectory is given, the last configuration will be used as the initial configuration for this MD run and the trajectory will be appended to. If a trajectory is given, then `trajectory_interval` and `trajectory_object_id` should be `None`.
- **constraints** (list of ints | list of `BaseConstraint`) – The list of atomic indices, denoting fixed atoms, or constraint objects, such as *RigidBody*. **Default:** `[]`.
- **trajectory_filename** (*str | None*) – The filename of the file to be used for storing the trajectory, or None if no trajectory should be written. A trajectory filename should not be given if *configuration* is a trajectory. **Default:** `None`.
- **steps** (*int*) – The number of time-steps to take in simulation. **Default:** `50`.
- **log_interval** (*int*) – The interval at which information, such as time, energy, temperature, etc. is written to the log output. **Default:** 1.
- **method** (`BaseMDmethod`) – The molecular dynamics method used for the simulation. **Default:** *NVEVelocityVerlet*.
- **xyz_filename** (*str*) – The name of the file to be used for storing the xyz trajectory, or None if no xyz-trajectory should be written. **Default:** `None`.
- **hook_functions** (*HookFunctions*) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the forces evaluation. It can also contain measurement hooks, which are called at the end of each MD step after all constraints have been applied. **Default:** `None`.
- **write_velocities** (*bool*) – Write the velocities to the trajectory file every `log_interval` steps. If `configuration` is a trajectory (i.e. this is a restart MD calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** True.
- **write_forces** (*bool*) – Write the forces to the trajectory file every `log_interval` steps.

  If `configuration` is a trajectory (i.e. this is a restart MD calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** True.

- **write_stresses** (*bool*) – Write the stress to the trajectory file every `log_interval` steps. A value of None means that `write_stresses` is True by default for NPT methods and False otherwise. This is to avoid the additional work of calculating the stress when it is not needed.

  If `configuration` is a trajectory (i.e. this is a restart MD calculation) this parameter will be the same value as it was in the previous trajectory. **Default:** None.

- **domain_decomposition_pattern** (list of type int | `Automatic` | None) – The pattern how the domains should be arranged in a parallel simulation. E.g. [1, 2, 4] means 1 domain in A-, 2 in B-, and 4 in C-direction. If `Automatic` domain decomposition is used, then the simulation cell will be divided into domains whose edges are as close together in length as possible. If `None` is given then domain decomposition will be disabled. **Default:** `Automatic`.

- **trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If `configuration` is a trajectory (i.e. this is a restart MD simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** The same value as `log_interval`.

- **trajectory_object_id** (*str | None*) – The object id of the trajectory written to `trajectory_filename`. If a value of `None` is given, then an object id will be chosen automatically. **Default:** `None`.

- **scf_initial_guess** (`UsePreviousConfiguration` | `DIISExtrapolation`) – Determines how the state of the system at each MD step is used as an initial guess for the next MD step, in the case when a self-consistent calculation is performed at each step. `UsePreviousConfiguration` means that the density matrix (or valence density/wave functions in a plane-wave calculation) of the previous configuration will be used as an initial guess for the next SCF calculation. For LCAO and semi-empirical calculations, it is possible and recommended to use `DIISExtrapolation`, which calculates an improved initial guess using the DIIS extrapolation method for the previous density matrices.

  This parameter has no effect when the calculator attached to `configuration` is a force-field calculator. **Default:** `UsePreviousConfiguration` for *PlaneWaveCalculator*, `DIISExtrapolation(4)` for *LCAOCalculator* and *SemiEmpiricalCalculator*.

- **pre_step_hook** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.

- **post_step_hook** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return status is ignored. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.

- **measurement_hook** (*callable | list of callable | None*) – A user-defined function or a list of functions which will be called at the end of each MD step after all constraints have been applied. The signature of the function requires the arguments `step, time, configuration`. Optional arguments include forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and can be accessed using the measurement method. Unhandled exceptions will terminate a Molecular Dynamics evaluation. If a list is given, the functions will be called in the given order. **Default:** `None`.

Usage Examples

Perform a molecular dynamics run of 500 steps of a bulk silicon crystal, using the Stillinger-Weber potential:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]

# Define coordinates
fractional_coordinates = [[0.0 ,  0.0 ,  0.0 ],
                          [0.25,  0.25,  0.25],
                          [0.5 ,  0.5 ,  0.0 ],
                          [0.75,  0.75,  0.25],
                          [0.5 ,  0.0 ,  0.5 ],
                          [0.75,  0.25,  0.75],
                          [0.0 ,  0.5 ,  0.5 ],
                          [0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# Molecular Dynamics
# ------------------------------------------------------------

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=300.0*Kelvin,
    remove_center_of_mass_momentum=True
)
```

(continues on next page)

(continued from previous page)

```
method = NVEVelocityVerlet(
    time_step=1*femtoSecond,
    initial_velocity=initial_velocity
)

md_trajectory = MolecularDynamics(
    bulk_configuration,
    trajectory_filename='trajectory.nc',
    steps=500,
    log_interval=50,
    method=method
)

bulk_configuration = md_trajectory.lastImage()

# Save the final configuration
nlsave('final_configuration.nc', bulk_configuration)

# Get the list of the kinetic energies of the snapshots
kinetic_energies = md_trajectory.kineticEnergies()

# Get the list of the potential energies of the snapshots
potential_energies = md_trajectory.potentialEnergies()
```

md_example1.py

Usage example of the measurement hook functionality to print the pressure and the cell volume to the log file at each step (e.g. in an NPT simulation).

```
# Define a hook function that prints the instantaneous cell volume and the pressure at␣
↪each MD step
def print_cell_and_stress(step, time, configuration, forces, stress, volume):

    # Convert volume to units of Ang**3.
    volume = volume.inUnitsOf(Ang**3)

    # Calculate the hydrostatic pressure from the stress tensor
    pressure = -(stress[0, 0] + stress[1, 1] + stress [2, 2]) / 3.0

    # Print the output to the log file
    print("| MD step %i:  Pressure : %f bar    Volume:  %f Angstrom**3" % (step,␣
↪pressure, volume))

# Run the MD simulation
md_trajectory = MolecularDynamics(
    bulk_configuration,
    constraints=[],
    trajectory_filename='trajectory.nc',
    steps=200,
    measurement_hook=print_cell_and_stress,
```

(continues on next page)

(continued from previous page)

```
    log_interval=50,
    method=NPTMartynaTobiasKlein()
)
```

md_example2-full.py

Example of a post-step hook function for MD simulations using a domain decomposition (see Notes below). Here the z-coordinate of the first atom (index 0) is tethered to a fixed position using a harmonic spring potential. Note, that this example also works in serial simulations.

```
def parallelBiasForceHook(step, time, configuration, local_forces):
    # Add a bias force to the force acting on the first atom
    # in the z-direction.
    # First, check if the first atom is in the local domain.
    if 0 in configuration.localIndices():
        # Get the local index of the atom with first atom.
        local_index = numpy.where(configuration.localIndices() == 0)[0]

        # Get the local coordinates and calculate the bias force.
        local_positions = configuration.localCartesianCoordinates()
        bias_force = -0.5*eV/Ang**2*(local_positions[local_index, 2] - 2.0*Ang)

        # Add the bias force to the local forces vector.
        local_forces[local_index, 2] += bias_force
```

md_parallel_hook_example.py

Notes

- The *MolecularDynamics* function returns an *MDTrajectory* object. After the simulation has completed, you can extract various properties of the stored snapshots, such as coordinates, forces, or kinetic energies, by using the *MDTrajectory* class methods.
- When continuing a simulation, an existing MD trajectory object is used to initialize the configuration of the next simulation. In this case, the `initial_velocities` argument of the MD method should be set to *ConfigurationVelocities*.
- You can perform custom operations on the configuration during the simulation, by using the `pre_step_hook` or `post_step_hook` functionality.

  The `pre_step_hook` one is invoked immediately *before* the force and stress calculations, which allows you to modify atomic positions or the cell vectors.

  The `post_step_hook` is invoked *after* the force and stress calculation, which means that you can modify the forces and stress, e.g. by adding a bias potential.

  The `measurement_hook` is invoked last. It may return a dictionary that describes the data that should be saved to the *MDTrajectory*. The keys of the dictionary should be strings and the values should be numeric (e.g. numpy array, PhysicalQuantity, integer, or float).

  Suitable functions must at least have the signature `(step, time, configuration)` where `step` denotes the integration step number, `time` is the current simulation time, and `configuration` is the current configuration. Optionally, additional arguments `forces`, `local_forces`, and `stress` can be passed. `forces` is the PhysicalQuantity array holding the current (global) forces, and `stress` is the stress tensor. `local_forces` are the forces on the atoms in local

domain when run in distributed mode (see notes below), otherwise the array is identical to `forces`. Note, that you need to modify these two arrays *inplace*, as the return status of the function is ignored. You can use the class methods of the `configuration` object (see e.g. *BulkConfiguration*) to perform operations on the atoms.

Measurement hooks may accept additional arguments beyond what is available to pre- and post-step hook functions:

–`temperature`, the instantaneous kinetic temperature

–`pressure`, the instantaneous pressure tensor

–`potential_energy`, the current total potential energy of the system

–`kinetic_energy`, the current total kinetic energy of the system

Moreover, it is possible to give a list of hook functions, which will be called sequentially at each MD step.

•Constraints are specified by adding constraint objects, such as *FixAtomConstraints*, *RigidBody*, or *FixCenterOfMass*, to the `constraints` list.

As an alternative to *FixAtomConstraints*, you can still use the old way of specifying the atomic indices of the fixed atoms in the `constraints` list. When using a thermostat to control the temperature, the reduced number of degrees of freedom due to the constraints is automatically taken into account.

•The inclusion of a homogeneous external electric field in bulk periodic DFT calculations with periodic boundary conditions is supported via the object *ElectricFieldConstraint*. For other geometries (i.e. slab or molecules), it is also possible to add an electric field by means of metallic regions in the Poisson solver (see *Poisson solvers*).

Notes for Parallel MD Simulations

MolecularDynamics behaves differently when run in parallel with different types of calculators.

For DFT calculators (e.g. *LCAOCalculator*) running a MD simulation using more than one MPI process will parallelize over the energy, force and stress calculations as in a normal DFT calculation on a single configuration. The MD integration will be carried using an identical copy of the configuration on each process. In this case, when running MD with custom scripts (e.g. hook functions), you need to ensure that every MPI process follows the exact same behavior (e.g. when using random numbers, make sure to use a fixed seed).

For force field calculations (i.e. *TremoloXCalculator*), it is also possible to parallelize over multiple MPI processes. When more than one MPI process is used (and distributed mode is enabled, see below), the MD simulation will be executed using a domain decomposition. This means that each MPI process will only contain a sub-region of the configuration (a domain). The number of atoms in these local domains changes as atoms move across the domain boundaries during the MD simulation. This mode makes most sense for large configurations (> 10 000 atoms) as the domain size must be larger than the largest cutoff in the potential. The distributed mode will automatically be triggered if all of the following conditions are fulfilled:

•There is more than one MPI process

•The number of atoms in the configuration is larger than 1000 atoms per MPI process

•All potential functions in the TremoloXCalculator support MPI

When these conditions are not met, the simulation will fall back to the serial mode. This means that each MPI process will run the same simulation and no parallel speedup will occur. For smaller system sizes, it may therefore be more efficient to use a single MPI process but many OpenMP threads instead.

When using *hook functions* in MD simulations, which are run with a domain decomposition, it is important to note that there are a few pitfalls, which can decrease the performance:

•Using the `forces` argument in the signature will trigger gathering of all forces on all processes each time the hook function is called. Instead you should use `local_forces` if possible.

•Calling `configuration.cartesianCoordinates()` or `configuration.velocities()` will trigger gathering of all positions respectively velocities on all processes. Instead, you should try to use `configuration.localCartesianCoordinates()` or `configuration.localVelocities()` if possible. These methods will return the arrays with the only the positions or velocities of the current atoms in the local domain (see the third usage example above).

NanoRibbon

NanoRibbon(*n*, *m*, *atom_1=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *atom_2=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *bond_length=PhysicalQuantity(1.42086, Ang)*, *vacuum=PhysicalQuantity(5.0, Ang)*)

Generate a nano ribbon.

Parameters

•**n** (*positive int*) – The first coefficient of the chiral vector.

•**m** (non-negative int, `m > n`) – The second coefficient of the chiral vector.

•**atom_1** (*PeriodicTableElement*) – The first element of the lattice cell. **Default:** `Carbon`

•**atom_2** (*PeriodicTableElement*) – The second element of the lattice cell. **Default:** `Carbon`

•**bond_length** (*PhysicalQuantity* of type length) – The bond length between neighboring atoms in the sheet. **Default:** 1.42086*Angstrom

•**vacuum** (*PhysicalQuantity* of type length) – The amount of vacuum that will be added on top and below the sheet. **Default:** 5.0*Angstrom

Returns
The created NanoRibbon

Return type
BulkConfiguration

Usage Examples

Define the geometry of a (3,0) Carbon nano-ribbon and repeat it ten times along z-axis:

```
ribbon = NanoRibbon(3,0)
ribbon = ribbon.repeat(1,1,10)
```

Notes

The *NanoRibbon* function generates a *BulkConfiguration* with a nano-ribbon geometry as specified by the input parameters.

NanoSheet

NanoSheet(*n*, *m*, *atom_1=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *atom_2=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *bond_length=PhysicalQuantity(1.42086, Ang)*, *vacuum=PhysicalQuantity(5.0, Ang)*)

Generate a nano sheet.

Parameters

•**n** (*positive int*) – The first coefficient of the chiral vector.

•**m** (non-negative int, m > n) – The second coefficient of the chiral vector.

•**atom_1** (*PeriodicTableElement*) – The first element of the lattice cell. **Default:** Carbon

•**atom_2** (*PeriodicTableElement*) – The second element of the lattice cell. **Default:** Carbon

•**bond_length** (*PhysicalQuantity* of type length) – The sheet bond length between neighboring atoms. **Default:** 1.42086*Angstrom

•**vacuum** (*PhysicalQuantity* of type length) – The amount of vacuum that will be added on top and below the sheet. **Default:** 5.0*Angstrom

Returns
The created NanoSheet

Return type
BulkConfiguration

Usage Examples

Define the geometry of a (3,0) Carbon-Nitride nanosheet and repeat it ten times along the z-axis:

```
sheet = NanoSheet(3,0,Carbon,Nitrogen)
sheet = sheet.repeat(1,1,10)
```

Notes

The *NanoSheet* function generates a *BulkConfiguration* with a nano-sheet geometry as specified by the input parameters.

NanoTube

NanoTube(*n*, *m*, *atom_1=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *atom_2=<class 'NL.CommonConcepts.PeriodicTable.Carbon'>*, *bond_length=PhysicalQuantity(1.42086, Ang)*, *vacuum=PhysicalQuantity(5.0, Ang)*)

Generate a nano tube.

Parameters

•**n** (*positive int*) – The first coefficient of the chiral vector.

•**m** (non-negative int, m > n) – The second coefficient of the chiral vector.

•**atom_1** (*PeriodicTableElement*) – The first element of the lattice cell. **Default:** Carbon

•**atom_2** (*PeriodicTableElement*) – The second element of the lattice cell. **Default:** Carbon

•**bond_length** (*PhysicalQuantity* of type length) – The bond length between neighboring atoms in the sheet. **Default:** 1.42086*Angstrom

•**vacuum** (*PhysicalQuantity* of type length) – The amount of vacuum that will be added on top and below the tube **Default:** 5.0*Angstrom

Returns
The created NanoTube

Return type
BulkConfiguration

Usage Examples

Define the geometry of a (3,0) Carbon nano-tube and repeat it ten times along z-axis:

```
cnt = NanoTube(3,0)
cnt = cnt.repeat(1,1,10)
```

Notes

The *NanoTube* function generates a *BulkConfiguration* with a nano-tube geometry as specified by the input parameters.

OptimizeBulkFreeEnergy

OptimizeBulkFreeEnergy(*configuration*, *temperature=None*, *pressure=None*, *max_steps=None*, *opt_tolerance=None*, *bounding_factor=None*, *optimize_geometry_parameters=None*, *dynamical_matrix_parameters=None*, *qpoint_density=None*, *preserve_bravais_lattice=None*, *optimize_volume=None*, *trajectory_filename=None*, *trajectory_object_id=None*)

Optimize the free energy of a bulk configuration. The vibrational free energy is calculated used a quasi-harmonic approach. The result of the optimization is returned as a *Table* class. Its columns contain the optimized REVIEW: This sentence is missing something configuration, the component and total energies and whether the optimization converged.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration to be optimized.

•**temperature** (*PhysicalQuantity* of type temperature.) – The temperature the configuration is optimized at. **Default:** 300 Kelvin

•**pressure** (*PhysicalQuantity* of type pressure.) – The pressure that the configuration is optimized at. Specifying a pressure of zero corresponds to optimizing the Helmholtz free energy. A non-negative value will result in optimizing the Gibbs free energy. **Default:** 1 bar

•**max_steps** (*int*) – The total number of cell optimization steps. This is independent of the number of atom relation steps. **Default:** 200

•**opt_tolerance** (*float*) – Sets the convergence criteria of the optimization. This corresponds roughly to the maximum change in the cell parameters being optimized when expressed in either Angstroms or Radians. **Default:** 0.01

•**bounding_factor** (*float*) – The range of the parameters to search in. Optimized parameters are sought within a range of +/- the bounds times the original value **Default:** 0.1

- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The optimization parameters used to define how the atom positions are optimized in each cycle. This optimization does not support optimizing the cell, and so all options relating optimizing the cell are ignored.
- **dynamical_matrix_parameters** (*DynamicalMatrixParameters*) – The dynamical matrix parameters used to define how the vibrational free energy is calculated at each step. As the configuration is a bulk configuration all options that do not apply to bulk configurations are ignored.
- **qpoint_density** – The density of q-points for which the phonon density of states should be calculated. A value of *None* uses the Monkhorst-Pack grid for the self-consistent calculation. **Default:** Monkhorst-Pack grid
- **preserve_bravais_lattice** (*bool*) – Whether to optimize the existing Bravais lattice of the input configuration or to perform the optimization on a triclinic configuration, allowing all cell parameters to be optimized. **Default:** True
- **optimize_volume** (*bool*) – Optimize the total cell volume rather than the individual unit cell parameters. **Default:** False
- **trajectory_filename** (*str*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. **Default:** Trajectory is not written to file
- **trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. If the object-id already exists in the given file, it will be overwritten. **Default:** None

Returns
The optimized configuration, energies, and success status of the optimization.

Return type
Table

Usage Examples

In this example, free energy optimization is used to model the phase change between the HCP and FCC phases of cobalt. The bulk free energy is computed in a range of 300 to 1000 Kelvin, and the resulting energies are plotted as a function of temperature. This shows a transition from HCP to FCC at around 650 Kelvin.

A workflow for he calculation can be downloaded `here` with a corresponding python script `Cobalt_FCC_HCP_Example.py`.

Notes

This method minimizes the free energy of a bulk configuration by including vibrational free energy in the total energy optimized. At each step of the optimization the atom positions are first optimized to determine the total electronic energy. The vibrational free energy is then calculated through a *DynamicalMatrix* calculation. Finally an additional energy is added for the compression work at the given pressure. These terms then combine together to define the free energy that is optimized.

Cell parameters can be optimized individually. By default, the Bravais lattice type is preserved during the optimization. If it is not preserved then the existing Bravais lattice is replaced with a triclinic Bravais lattice, which is then optimized. It is also possible to select to optimize just the cell volume. In this case the total volume is uniformly scaled to give different cell sizes.

When performing this optimization it is recommended to first fully optimize the configuration using the chosen calculator to get the minimum configuration at absolute zero temperature. The dynamical matrix calculation can produce negative vibrational modes if the configuration is too far from the minimum.

The optimization returns a *Table* with a single row containing the optimized configuration, the three components of the free energy, the total free energy and a boolean flag which indicates if the optimization converged.

OptimizeGeometry

OptimizeGeometry(*configuration*, *max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *optimize_cell=None*, *disable_stress=None*, *optimizer_method=None*, *target_stress=None*, *pre_step_hook=None*, *post_step_hook=None*, *constrain_bravais_lattice=None*, *trajectory_interval=None*, *restart_strategy=None*, *remove_drift=None*, *enable_optimization_stop_file=None*, *write_raw_stress=None*, *write_raw_forces=None*, *compute_stress=None*)

Function for optimizing the geometry and/or the unit cell of the given configuration.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to be optimized.
- **max_forces** (*PhysicalQuantity* of type force) – The convergence criterion for the atomic forces. **Default:** 0.05*eV/Angstrom
- **max_stress** (*PhysicalQuantity* of type pressure) – The convergence criterion for the maximum difference between the internal stress and the target stress. **Default:** 0.1*GPa
- **max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang
- **constraints** (*list of integers and Constraints objects*) – A list of indices of the atom with fixed positions and Constraints objects. **Default:** []
- **trajectory_filename** (*str | None*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. **Default:** None
- **trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. If the object-id already exists in the given file, it will be overwritten. **Default:** None
- **disable_stress** – **Deprecated:** from v2022.03, use `optimize_cell` parameter instead.
- **optimize_cell** (*bool*) – If enabled, allow the lattice vectors to change during the optimization. This also enables stress calculation for bulk configurations. Cell optimization is disabled if the cell is constrained in all directions (see Constraints). **Default:** True for BulkConfigurations otherwise False
- **optimizer_method** (*FIRE* | *LBFGS*) – The optimization algorithm to use. **Default:** *LBFGS*
- **target_stress** (*PhysicalQuantity* of type pressure) – The target internal stress (tensor) of the system. Can be given as a single value in case of isotropic pressure, or as an internal stress vector in Voigt notation or as a 3x3-matrix. **Default:** 0*GPa
- **pre_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just before each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** None
- **post_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just after each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** None
- **constrain_bravais_lattice** (*bool*) – Enable preserving the Bravais lattice symmetry of the configuration. **Default:** True if the `target_stress` is commensurate with the lattice symmetries
- **trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1

- **restart_strategy** (NoRestart | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** RestartFromTrajectory()
- **remove_drift** (*bool*) – In ab-initio calculations, the sum of the forces along each Cartesian direction does not necessarily sum to zero due to numerical inaccuracies. This option controls if the drift in the forces should be removed by subtracting the average force along each Cartesian direction from all atoms. **Default:** True
- **enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the geometry optimization. If True, creation of the stop file will stop the optimization at the next step. The name of the stop file will be shown in the log output; it will be stop-geometry-optimization-uniqueID, where uniqueID is a randomly generated identifier for this optimization. The file must be created in the current working directory. **Default:** True
- **write_raw_stress** – **Deprecated:** from v2025.06, use compute_stress parameter instead.
- **write_raw_forces** – **Deprecated:** from v2025.06, forces are always written to trajectory.
- **compute_stress** (*Automatic | bool*) – Compute and write stress to trajectory regardless if cell optimization is enabled or not. If cell optimization is enabled stress will automatically be computed and written. If cell optimization is disabled and you wish to compute and write stress to the trajectory, you need to explicitly enable compute stress. **Default:** Automatic meaning; the same as optimize cell flag.

Returns
The optimized configuration. The returned configuration is assigned the self-consistent calculation associated with the final step of the optimization loop.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*

Usage Examples

Optimize the geometry of a water molecule:

```
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
cartesian_coordinates = [[  0.0,  -1.70000000e-05,   1.20198000e-01],
                         [  0.0,   7.59572000e-01,  -4.86714000e-01],
                         [  0.0,  -7.59606000e-01,  -4.86721000e-01]]*Angstrom

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=elements,
    cartesian_coordinates=cartesian_coordinates
    )

# Define a calculator
molecule_configuration.setCalculator(LCAOCalculator())

# Perform optimization and store the relaxation trajectory
optimized_configuration = OptimizeGeometry(molecule_configuration,
                                           trajectory_filename='opt.nc'
                                           )

# Save the relaxed configuration
nlsave('optimized_configuration.nc', optimized_configuration)
```

(continues on next page)

(continued from previous page)

```
```

optimize.py

Optimize a silicon unit cell under a uniaxial tensile stress:

```
# Set up lattice
vector_a = [0.0, 2.7153, 2.7153]*Angstrom
vector_b = [2.7153, 0.0, 2.7153]*Angstrom
vector_c = [2.7153, 2.7153, 0.0]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------

bulk_configuration = OptimizeGeometry(
        bulk_configuration,
        max_forces=0.001*eV/Ang,
        max_stress=0.0001*eV/Ang**3,
        max_steps=200,
        max_step_length=0.2*Ang,
        constraints=[],
        trajectory_filename='opt2.nc',
        optimizer_method=LBFGS(),
        target_stress=[2.0, 0.0, 0.0, 0.0, 0.0, 0.0]*GPa,
        )

# Save the relaxed configuration
```

(continues on next page)

(continued from previous page)

```
nlsave('optimized_configuration2.nc', bulk_configuration)
```

optimize2.py

Notes

By default, all cell optimizations are performed under zero stress.

The definition of target stress is that if a single value is given, it will be interpreted as an *external* pressure value from which the *internal* target stress tensor is calculated as

$$\sigma = \begin{pmatrix} -p & 0 & 0 \\ 0 & -p & 0 \\ 0 & 0 & -p \end{pmatrix} .$$

If a target stress tensor is given, it will be interpreted as *internal* stress of the system, meaning that a negative entry on the diagonal will result in a compression in the corresponding direction and vice versa. The target stress may also be specified in Voigt notation.

By default `constrain_bravais_lattice` is `True` when the `target_stress` is commensurate with the lattice symmetries. This means that the Bravais lattice type will not change unless a symmetry breaking internal stress is applied. If a symmetry breaking `target_stress` is used then the Bravais Lattice of the configuration is automatically converted to a UnitCell, to enable changes in the shape of the cell.

Note: The last line of the logging output of OptimizeGeometry provides the tag `OPT` which can be used to grep for the current step, volume, max. force, max. stress error and energy / enthalpy.

Optimization simulations can be restarted from a previous run, e.g. if the simulation was interrupted or the optimization did not converge in the maximum number of steps. The detailed restart behavior is specified and documented in *RestartFromTrajectory*.

External electric field

The inclusion of a homogeneous external electric field in bulk periodic DFT calculations with periodic boundary conditions is supported via the object *ElectricFieldConstraint*. For other geometries (i.e. slab or molecules), it is also possible to add an electric field by means of metallic regions in the Poisson solver (see *Poisson solvers*).

OptimizeNudgedElasticBand

OptimizeNudgedElasticBand(*neb*, *max_forces=None*, *max_stress=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *log_interval=None*, *spring_constant=None*, *spring_skew=None*, *climbing_image=None*, *climbing_image_tolerance=None*, *preoptimization=None*, *optimizer_method=None*, *log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *pre_step_hook=None*, *post_step_hook=None*, *optimize_cell=None*, *disable_stress=None*, *target_stress=None*, *trajectory_interval=None*, *restart_strategy=None*, *write_raw_stress=None*, *write_raw_forces=None*, *surrogate_model=None*)

Function for performing a NEB optimization.

Parameters

- **neb** (*NudgedElasticBand*) – The nudged elastic band configuration to optimize.
- **max_forces** (*PhysicalQuantity* of type eV/Ang) – The maximum forces when the optimization should stop. **Default:** 0.05*eV/Ang.
- **max_stress** (*PhysicalQuantity* of type pressure) – The convergence criterion for the maximum difference between the internal stress and the target stress. **Default:** 0.1*GPa.
- **max_steps** (*int*) – The maximum number of steps to take before the NEB relaxation stops. When restarting, the max step are added to the current step. **Default:** 200.
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang.
- **constraints** (*list of ints*) – List of atom indices that are kept fixed during optimization. **Default:** [].
- **trajectory_filename** (*str | None*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. **Default:** None.
- **trajectory_object_id** (*str | None*) – The object id to use when saving the trajectory to HDF5. **Default:** None.
- **log_interval** – **Deprecated:** from v2019.12, use `trajectory_interval` instead.
- **spring_constant** (*PhysicalQuantity* of type force/length) – The spring constant used for the NEB relaxation. **Default:** 5.0*(eV/Ang**2).
- **spring_skew** (*float*) – Skew the spring constant linearly depending on the image energies. For a value larger than 1.0, higher energy images are connected by a stronger spring. As a result images are spaced closer around the transition state, which can improve convergence of the NEB optimization. This parameter determines the maximum factor by which the spring constant is increased or decreased. The current transition state estimate and the lower of the endpoints are used as references. **Default:** 1.0.
- **climbing_image** (bool | *PhysicalQuantity* of type eV/Ang | Automatic | None) – Flag indicating if the climbing image algorithm should be used to find a transition state. If a force quantity is given, the climbing image algorithm is initially disabled and will be activated after the force threshold is reached. If `Automatic` is given, the threshold is set to 5 times the `max_forces` tolerance. **Default:** False.
- **climbing_image_tolerance** (*PhysicalQuantity* of type eV/Ang) – The tolerance with which to converge the climbing image. This can be different to the forces convergence tolerance of the overall NEB. If `climbing_image` is `False` this tolerance is ignored. **Default:** Same as the `max_forces` tolerance.
- **preoptimization** (*bool*) – Flag indicating if the endpoints should be optimized before the NEB optimization. Preoptimization is disabled when restarting. Note that the NEB must be re-interpolated after endpoint preoptimization. The interpolation is performed with the method specified when creating the NEB object. **Default:** False.
- **optimizer_method** (*FIRE* | *LBFGS* | *VelocityProjectionOptimization*) – The optimizer to use for optimizing the structure. **Default:** *LBFGS*.
- **log_filename_prefix** (`Automatic` | str | None) – The logging output from each image will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** `Automatic`.
- **pre_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just before each optimization step. The signature of the function requires the arguments (step, neb_configuration). The return status is ignored. Unhandled exceptions will terminate the optimization.
- **post_step_hook** (*A callable function or method*) – An optional user-defined function which will be called just after each optimization step. The signature of the function requires the arguments (step, neb_configuration). The return status is ignored. Unhandled exceptions will terminate the optimization.

- **optimize_cell** (*bool*) – If it is *None*, it will be set automatically according to the NEB configuration. If the NEB configuration is composed of configurations with the type *MoleculeConfiguration* | *SurfaceConfiguration* | *DeviceConfiguration*, it is set False. If the NEB configuration is composed of configurations of the type *BulkConfiguration* and the lattice vectors of the configurations are different, it is set to True; otherwise it is False.
- **disable_stress** – **Deprecated:** from v2022.03, use `optimize_cell` parameter instead.
- **target_stress** (*PhysicalQuantity* of type pressure) – The target internal stress (tensor) of the system. Can be given as a single value in case of isotropic pressure, or as an internal stress vector in Voigt notation or as a 3x3-matrix. **Default:** 0*GPa.
- **trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1.
- **restart_strategy** (`NoRestart` | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** `RestartFromTrajectory()`.
- **write_raw_stress** – **Deprecated:** from v2025.06.
- **write_raw_forces** – **Deprecated:** from v2025.06.
- **surrogate_model** (*GaussianProcessNEBParameters* | None) – Parameters for the surrogate model used to estimate the potential energy surface. The surrogate model is not compatible with pre_step_hook, and optimize_cell or target_stress arguments. Furthermore, it cannot be restarted.

Usage Examples

Find the reaction path for conversion of ethane to ethene, i.e.

$$C_2H_6 \rightarrow C_2H_4 + H_2$$

```
# Find the reaction pathway
optimized_neb = OptimizeNudgedElasticBand(
        neb_configuration,
        max_forces=0.05*eV/Ang)
```

`neb_c6h6.py`

The two following examples demonstrate how to calculate the reaction path for rock-salt to wurtzite in CdSe. The first example is with a very small unit cell that results in a fully-concerted reaction mechanism, while the second is in a larger unit cell that first forms a line defect.

```
# Perform a VC-NEB optimization
neb_configuration = OptimizeNudgedElasticBand(
    neb_configuration,
    max_forces=0.01*eV/Ang,
    max_stress=0.01*GPa,
    max_steps=500,
    climbing_image=True,
    optimize_cell=True,
)
```

`cdse_neb_concerted.py` `cdse_neb_line_defect.py`

Variable Cell NEB

It is possible to study phase transitions using `OptimizeNudgedElasticBand`. When a *NudgedElasticBand* contains images that are bulk configurations with different lattice vectors, the generalized solid-state nudged elastic band method (GSS-NEB)[1] algorithm is used. The algorithm optimizes both the atomic and lattice degrees of freedom along the reaction pathway and can be used to determine the reaction mechanism and energy barrier of a phase change. The `target_stress` parameter can be set to apply pressure or arbitrary stress.

There are two important points to be aware of when using this technique:

1. In order to get an accurate estimate of the energy barrier one must use very large unit cells. This is because if a small unit cell is used, then all atoms will move during the phase transition. This is almost certainly not the favorable pathway in real systems. A large enough unit cell should permit the formation of a local nucleation process that occurs in either the initial or final phase of the material. After the local nucleation event the new crystal structure should then propagate across the unit cell. For more details, see for instance Sec. III.C. and Fig. 9 in Ref.[Page 4156, 1].

2. Great care must be taken when constructing the initial and final images. Just as in the regular NEB algorithm, the atom indices in the initial and final structure must match. Atom #1 in the initial structure will be moved to atom #1 in the final structure. This means that you cannot simply take two different crystal structures with the same number of atoms and expect the resulting pathway to be correct.

Optimization with surrogate potential energy surface

Optimization of the NEB can be accelerated by using a surrogate potential energy surface (PES). The surrogate model is trained on-the-fly during the optimization based on the known *true* energies and forces determined by the reference calculator. The model is then used to predict the the next step in the optimization, thus significantly reducing the number of calls to the reference calculator. The surrogate model is updated periodically to ensure that the predictions are accurate. The present implementation is based on Gaussian process regression, a machine learning model that is well-suited for this purpose[2][3]. The surrogate optimization loop is carried out with the *All Images Evaluated* algorithm, which means that all images are optimized in parallel.

The surrogate model is enabled by passing a *GaussianProcessNEBParameters* object to the `surrogate_model` argument of the `OptimizeNudgedElasticBand` function.

Notes on Parallelization

For non-force field calculators, the number of processes used to calculate the energy and forces on each image is determined by the *ParallelParameters* object that has been set on the attached calculator. By default, the update is parallelized over images and then over the per-image energy/force calculation.

For force field calculators, which do not support *ParallelParameters*, the number of processes per image (PPI) is automatically determined. In the case where the total number of available MPI processes is equal to or smaller than the number of interior images the PPI is set to 1. This means that the number of images optimized in parallel corresponds to the number of available processes. If there are more available processes than interior images, the processes are distributed across those images. It is generally recommended not to use more MPI processes than interior images but rely on threading instead.

[1] Daniel Sheppard, Penghao Xiao, William Chemelewski, DuaneăD. Johnson, and Graeme Henkelman. A generalized solid-state nudged elastic band method. *The Journal of Chemical Physics*, 136(7):074103, 2012. doi:10.1063/1.3684549.

[2] Olli-Pekka Koistinen, FreyjaăB. Dagbjartsdóttir, Vilhjálmur Ásgeirsson, Aki Vehtari, and Hannes Jónsson. Nudged elastic band calculations accelerated with Gaussian process regression. *The Journal of Chemical Physics*, 147(15):152720, 09 2017. URL: https://doi.org/10.1063/1.4986787, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4986787/15536366/152720_1_online.pdf, doi:10.1063/1.4986787.

[3] Olli-Pekka Koistinen, Vilhjálmur Ásgeirsson, Aki Vehtari, and Hannes Jónsson. Nudged Elastic Band Calculations Accelerated with Gaussian Process Regression Based on Inverse Interatomic Distances. *Journal of Chemical Theory and Computation*, 15(12):6738–6751, 2019. URL: https://doi.org/10.1021/acs.jctc.9b00692, arXiv:https://doi.org/10.1021/acs.jctc.9b00692, doi:10.1021/acs.jctc.9b00692.

Care should be taken to ensure that the total number of MPI processes is divisible by the number of interior (moving) NEB images to achieve good load balancing. For example, if there are 7 images total, then 5 of them are interior images that move during the simulation and the total number of MPI processes should be a multiple of 5. The calculation will still run if the number of MPI processes is not a multiple of the number of interior images, however, a warning will be printed and the calculation of some images will be slower than others reducing the parallel efficiency of the overall calculation.

The output from the calculation will be logged to a separate file for each image whose name is the specified by the `log_filename_prefix` parameter with the image number appended to it. If the log files already exist, they will be appended to.

If `preoptimization` is selected and there are enough MPI processes for at least two images to be calculated in parallel, then the `preoptimization` will be performed in parallel and its output will be logged to files starting with the `log_filename_prefix` followed by _preoptimization.

NEB optimizations can be restarted from a previous run, e.g. if the simulation was interrupted or the optimization did not converge in the maximum number of steps. The detailed restart behavior is specified and documented in *RestartFromTrajectory*.

•See also *NudgedElasticBand*.

External Electric Field

The inclusion of a homogeneous external electric field in bulk periodic DFT calculations with periodic boundary conditions is supported via the object *ElectricFieldConstraint*. For other geometries (i.e. slab or molecules), it is also possible to add an electric field by means of metallic regions in the Poisson solver (see *Poisson solvers*).

Troubleshooting non-convergence

Unfortunately, NEB optimizations can run into convergence issues which cannot be completely avoided even when following best practice guidelines[4]. However, there are several strategies one can try to improve convergence:

•The convergence is very sensitive to the initial guess of the path. The simplest interpolation scheme is a *LinearInterpolation* between the provided initial and final states. While it is guaranteed to generate a path, it can lead to unphysical configurations when multiple atoms rotate around a common center. In such cases, it is recommended to choose a more sophisticated interpolation scheme, such as the *ImageDependentPairPotential* (IDPP) method and its *sequential variant*

•Related to the previous point, the endpoints should not only be well converged, but also chosen in such a way that they minimize the path length by avoiding unnecessary rotations and translations. A failed NEB optimization can often be salvaged by taking the two images that are closest to the transition state and optimizing them to the closest local minimum. Afterwards, the NEB can be restarted with those minima as the new endpoints.

•Increasing the number of images can also aid convergence. The more images are used the more accurate the forces will be that ensure the images stay evenly spaced and on the minimum energy path. However, one must be aware that the computational cost scales linearly with the number of images (see *Notes on Parallelization*).

•In addition to increasing the number of images one can also enable the spring skewing option. This will push the images towards higher energy regions, which will improve convergence as these regions are usually associated with more complex atomic rearrangements. In practice, one can achieve convergence faster and with fewer images by using spring skewing[5].

[4] AndrewăW. Ruttinger, Divya Sharma, and Paulette Clancy. Protocol for directing nudged elastic band calculations to the minimum energy pathway: nurturing errant calculations back to convergence. *Journal of Chemical Theory and Computation*, 18(5):2993–3005, 2022. URL: https://doi.org/10.1021/acs.jctc.1c00926, doi:10.1021/acs.jctc.1c00926.

[5] G.ăHenkelman, B.ăP. Uberuaga, and H.ăJonsson. A climbing image nudged elastic band method for finding saddle points and minimum energy paths. *J. Chem. Phys.*, 2000.

•In the NEB method, the spring constant is relatively arbitrary since it only serves to keep the images equidistant. However, the spring constant can have some impact on the convergence of the NEB calculation. Both too high and too low spring constants can lead to slower convergence. The default value preforms well in most cases.

•The *LBFGS* optimizer is the default for NEB optimizations since it is the most robust optimizer. If the initial path is far from the minimum energy path (MEP), convergence can be slow. Another optimizer, such as *FIRE* or *VPO* might be more efficient. One might also consider starting out with FIRE and then switching to LBFGS once the path has closed in on the MEP.

ParallelEnergyAndForces

ParallelEnergyAndForces(*configurations*, *processes_per_configuration*, *filenames=None*, *master_only=True*)

Evaluates the energy and forces for a list of configurations in parallel.

Parameters

•**configurations** (list of configurations (*MoleculeConfiguration* | *BulkConfiguration* | *SurfaceConfiguration* | *DeviceConfiguration*)) – A list of configurations with attached calculators.

•**processes_per_configuration** (*int*) – The number of MPI processes to use for calculating the energy and forces on each configuration.

•**filenames** (*list of strings*) – A list of filenames to log each calculation to. It must be the same length as the configurations argument. If None is given, then logging will be performed to stdout. **Default:** `All calculations are logged to stdout`

•**master_only** (*bool*) – Controls if the master process should be the only rank allowed to write to the log. **Default:** `True`

Returns
A tuple of energies and forces as *PhysicalQuantity* arrays.

Return type
tuple

Usage Examples

Calculate the potential energy curve and forces for a hydrogen molecule in parallel. At the end of the script, a table of internuclear distances, energies, and the force is printed to the screen.

```
# Make a list to hold the configurations.
configurations = []

# Loop over a list of distances between 0.3 and 5.0 Angstrom.
distances = numpy.linspace(0.3, 5.0, 20)
for distance in distances:
    # Define elements
    elements = [Hydrogen, Hydrogen]

    # Define coordinates
    cartesian_coordinates = [[ 0.0, 0.0, 0.0 ],
                             [ distance, 0.0, 0.0 ]]*Angstrom

    # Set up configuration
    molecule_configuration = MoleculeConfiguration(
```

(continues on next page)

(continued from previous page)

```
        elements=elements,
        cartesian_coordinates=cartesian_coordinates
        )

    # Define a calculator
    molecule_configuration.setCalculator(LCAOCalculator())

    # Add the configuration to the list of configurations.
    configurations.append(molecule_configuration)

def task_function(configuration, index):
    # Compute the total energy.
    total_energy = TotalEnergy(configuration)
    # Save the result to a file.
    nlsave('total_energy_%i.hdf5'%index, configuration)
    # Return the calculated total energy.
    return total_energy.evaluate()

# Define a list of filenames to save the logging output from each calculation to.
filenames = [ 'total_energy_%i.log' % i for i in range(len(configurations)) ]

# Calculate the energy of each configuration. Each calculation will use 2 MPI processes.
energies = ParallelMapConfigurations(
    task_function,
    configurations,
    processes_per_configuration=2,
    filenames=filenames,
)

# Only print on the master process. This prevents the table from being printed multiple␣
↪times.
if processIsMaster():
    print('%10s %12s' % ('distance', 'energy'))
    for i in range(len(configurations)):
        print('%10.4f %12.3e' % (distances[i], energies[i].inUnitsOf(eV)))
```

parallelmapconfigurations.py

Notes

It is important to properly coordinate the total number of MPI processes, the `processes_per_configuration` argument, and the number of configurations. When `ParallelEnergyAndForces` is called, the MPI processes are divided up into `NLEngine.numberOfProcesses()/processes_per_configuration` sized groups. For example, if there are 8 MPI processes and `processes_per_configuration=2`, then 4 groups will be made. However, if there are only 2 configurations in the `configurations` list then 2 of those groups will be idle.

Ideally, one would pick `processes_per_configuration` to be the largest number of processes that a single DFT calculation runs efficiently on. This generally depends on a number of variables including the number of atoms, basis set size, computer hardware, etc. Then, the total number of MPI processes should be an integer multiple of `processes_per_configuration`.

This function can be used with *ATK-ForceFiled* calculators as well. However, *ATK-ForceField* does not currently make use of MPI. This means that `processes_per_configuration` should always be set to 1 in order not to have idle processes.

See also, *Notes*.

ParallelMapConfigurations

ParallelMapConfigurations(*task_function*, *configurations*, *processes_per_configuration*, *log_filenames=None*, *interleaved_split=False*, *master_only=True*, *synchronize_configurations=False*)

Applies a function on a list of configurations in parallel.

Parameters

- **task_function** (*function*) – The function to apply to each configuration. It takes an **AtomicConfiguration** and the index of the configuration in the configurations array as its arguments. The function can, optionally, return data that will be saved and send to all processes upon return from **ParallelMapConfigurations**. Note: The return type needs to be able to be sent over **MPIself**. This currently includes most basic datatypes (e.g. numbers, lists, dicts, numpy arrays) as well as ATK configurations. A pickling error will be raised if the type is not picklable.
- **configurations** (list of configurations (*MoleculeConfiguration* | *BulkConfiguration* | *SurfaceConfiguration* | *DeviceConfiguration*)) – list of configurations with attached calculators that is passed to the task_function.
- **processes_per_configuration** (*int*) – The number of MPI processes that will be used to call task_function for each configuration.
- **log_filenames** (*list of strings*) – A list of filenames to log each calculation to. It must be the same length as the configurations argument. If None is given, then all logging will be performed to stdout. **Default:** `All calculations are logged to stdout`
- **interleaved_split** (*bool*) – When True, the tasks are distributed amongst the processes groups in an interleaved way. If False, the tasks are distributed in a contiguous manner. **Default:** `False`
- **master_only** (*bool*) – Controls if the master process should be the only rank allowed to write to the log. **Default:** `True`
- **synchronize_configurations** (*bool*) – Ensure that the state of the configurations at the end of the task is synchronized and distributed across all processes. **Default:** `False`

Returns
A list containing the results of applying the task_function to each configuration.

Return type
list

Usage Examples

Calculate the potential energy curve for a hydrogen molecule in parallel. This example uses *TotalEnergy* to calculate the potential energy. The `task_function` function is passed to `ParallelMapConfigurations` to perform each individual energy calculation. Each time `task_function` is called, it calculates the total energy, saves the TotalEnergy analysis object to a file, and then returns the total energy. `ParallelMapConfigurations` collects the returned energies into an array that is returned when the calculations complete. At the end of the script, a table of internuclear distances and energies is printed to the screen.

```
# Make a list to hold the configurations.
configurations = []

# Loop over a list of distances between 0.3 and 5.0 Angstrom.
distances = numpy.linspace(0.3, 5.0, 20)
for distance in distances:
    # Define elements
    elements = [Hydrogen, Hydrogen]

    # Define coordinates
    cartesian_coordinates = [[ 0.0, 0.0, 0.0 ],
                             [ distance, 0.0, 0.0 ]]*Angstrom

    # Set up configuration
    molecule_configuration = MoleculeConfiguration(
        elements=elements,
        cartesian_coordinates=cartesian_coordinates
        )

    # Define a calculator
    molecule_configuration.setCalculator(LCAOCalculator())

    # Add the configuration to the list of configurations.
    configurations.append(molecule_configuration)

def task_function(configuration, index):
    # Compute the total energy.
    total_energy = TotalEnergy(configuration)
    # Save the result to a file.
    nlsave('total_energy_%i.hdf5'%index, configuration)
    # Return the calculated total energy.
    return total_energy.evaluate()

# Define a list of filenames to save the logging output from each calculation to.
filenames = [ 'total_energy_%i.log' % i for i in range(len(configurations)) ]

# Calculate the energy of each configuration. Each calculation will use 2 MPI processes.
energies = ParallelMapConfigurations(
    task_function,
    configurations,
    processes_per_configuration=2,
    filenames=filenames,
)

# Only print on the master process. This prevents the table from being printed multiple␣
↪times.
if processIsMaster():
    print('%10s %12s' % ('distance', 'energy'))
    for i in range(len(configurations)):
        print('%10.4f %12.3e' % (distances[i], energies[i].inUnitsOf(eV)))
```

parallelmapconfigurations.py

Notes

It is important to properly coordinate the total number of MPI processes, the `processes_per_configuration` argument, and the number of configurations. When `ParallelMapConfigurations` is calledm the MPI processes are divided up into groups of MPI processes no larger than `processes_per_configuration`. For example, if there are 8 MPI processes and `processes_per_configuration=2`, then 4 groups will be made. If there are 8 MPI Processes and `processes_per_configuration=3` then two groups of 3 process and one goup of 2 processes will be made.

The caller is responsible for choosing settings that guarantee that all groups of MPI processes handle approximately the same number of configurations. Otherwise, the parallel performance will not be ideal due to load balancing issues. This typically means that the total number of groups should divide evenly into the total number of configurations.

Ideally, one would pick `processes_per_configuration` to be the largest number of processes that a single DFT calculation runs efficiently on. This generally depends on a number of variables including the number of atoms, basis set size, computer hardware, etc. Then, the total number of MPI processes should be an integer multiple of `processes_per_configuration`.

This function can be used with *ATK-ForceField* calculators as well. However, *ATK-ForceField* does not currently make use of MPI. This means that `processes_per_configuration` should always be set to 1 in order not to have idle processes.

See also, *Notes*.

RelaxDimer

RelaxDimer(*dimer*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *finite_difference_method=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *log_filename_prefix=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Automatic'>*, *trajectory_interval=None*, *restart_strategy=None*, *post_step_hook=None*, *remove_drift=None*, *enable_optimization_stop_file=None*, *trial_step_length=None*, *min_rot_force=None*, *max_rot_force=None*, *trial_angle=None*, *max_number_of_rotations=None*, *extrapolate_rot_force=None*)

Run dimer saddle point relaxation.

Parameters

- **dimer** (*DimerConfiguration*) – The dimer to be optimized.
- **max_forces** (*PhysicalQuantity* with force units) – The maximum force convergence criterion. **Default:** 0.05 * eV / Angstrom
- **max_steps** (*int*) – The maximum number of optimization steps allowed. **Default:** 200
- **max_step_length** (*PhysicalQuantity* with length units) – The maximum step length the optimizer may take. **Default:** 0.2 * Angstrom
- **finite_difference_method** (`NLFlag` (`Forward` | `Central`)) – The finite difference scheme to use. `Forward` uses only one configuration for the force calculation, while `Central` uses two configurations. Thus, `Forward` is about a third faster than `Central`. However, `Central` is more accurate and stable which can result in overall fewer steps to convergence. If `Forward` is used, it is recommended to adjust the `dimer_separation` parameter in the *class: `~.DimerConfiguration* object to a smaller value. **Default:** `Central`
- **constraints** (*list of ints | None*) – List of atom indices that are kept fixed during optimization. **Default:** [].
- **trajectory_filename** (*str | None*) – The filename to write the trajectory to. **Default:** The trajectory is not saved.

- **trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. If the object-id already exists in the given file, it will be overwritten. **Default:** None
- **log_filename_prefix** (Automatic | str | None) – The logging output from each dimer endpoint will be written to filenames starting with this value. If it is set to Automatic then the prefix will be the name of the calling python script. If it is set to None, then all output will be written to stdout. **Default:** Automatic.
- **trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1
- **restart_strategy** (NoRestart | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** RestartFromTrajectory()
- **remove_drift** (*bool*) – Controls if the drift is removed from the forces. **Default:** True
- **post_step_hook** (*A callable method, function or instance*) – An optional user-defined function which will be called just after each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** None
- **enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the dimer relaxation. If True, creation of the stop file will stop the relaxation at the next step. The name of the stop file will be shown in the log output; it will be stop-dimer-relaxation-uniqueID, where uniqueID is a randomly generated identifier for this relaxation. The file must be created in the current working directory. **Default:** True
- **trial_step_length** (*PhysicalQuantity* of type length) – The trial step to determine the ideal step. **Default:** 0.001 * Angstrom
- **min_rot_force** (*PhysicalQuantity* of type force) – The minimum rotational force necessary to attempt a rotation. **Default:** 0.1 * eV/Angstrom
- **max_rot_force** (*PhysicalQuantity* of type force) – The maximum rotational force that terminates the eigenmode optimization. A rotation will always be performed before checking this parameter. **Default:** 1.0 * eV/Angstrom
- **trial_angle** (*float*) – The trial angle in radians. **Default:** pi / 4.0 (45 degrees)
- **max_number_of_rotations** (*Positive int*) – The maximum number of rotations. **Default:** 1
- **extrapolate_rot_force** (*bool*) – Use an extrapolation scheme to estimate the rotational force. This reduces the number of force evaluations per rotation from 2 to 1. This parameter only has an effect if the maximum number of rotations is greater than 1. **Default:** False

Returns
The optimized dimer.

Return type
DimerConfiguration

Usage Examples

The Dimer method[1] [2] is a saddle search method that can be used to relax a configuration into a nearby transition state following the smallest eigenmode. Unlike other methods, it does so without requiring the calculation of second derivatives. Instead, the Dimer method uses a finite difference approximation to the gradient of the potential energy surface to

[1] Graeme Henkelman and Hannes Jónsson. A dimer method for finding saddle points on high dimensional potential surfaces using only first derivatives. *The Journal of Chemical Physics*, 111(15):7010–7022, 10 1999. URL: https://doi.org/10.1063/1.480097, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/111/15/7010/19179946/7010_1_online.pdf, doi:10.1063/1.480097.

[2] Andreas Heyden, AlexisăT. Bell, and FrerichăJ. Keil. Efficient methods for finding transition states in chemical reactions: Comparison of improved dimer method and partitioned rational function optimization method. *The Journal of Chemical Physics*, 123(22):224101, 12 2005. URL: https://doi.org/10.1063/1.2104507, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.2104507/13124124/224101_1_online.pdf, doi:10.1063/1.2104507.

find the smallest eigenmode. This makes the dimer method computationally very efficient[3], especially when compared to the climbing-image *NudgedElasticBand* method, which requires more images.

The initial configuration and search direction are supplied to the *RelaxDimer()* function through a *DimerConfiguration* object. The search direction can either be randomly generated or correspond to an initial guess of the minimum eigenmode.

```
# Get the coordinates of the transition state estimate and a local minimum.
ts_coordinates = ts.cartesianCoordinates()
minimum_coordinates = minimum.cartesianCoordinates()

# Calculate an initial displacement.
norm = (ts_coordinates - minimum_coordinates).norm()
# Normalize the displacement vector and divide by 10 to get a small displacement.
displacement_vector = ((ts_coordinates - minimum_coordinates) / norm) / 10

dimer = DimerConfiguration(
    configuration=ts,
    displacement=displacement_vector * Angstrom
)

relaxed_dimer = RelaxDimer(
    dimer=dimer,
    trajectory_filename='dimer_relaxation.hdf5',
    # Translation.
    max_forces=0.05 * eV / Angstrom,
    max_steps=50,
    max_step_length=0.2 * Angstrom,
    constraints=None,
    trial_step_length=0.001 * Angstrom,
    # Rotation.
    min_rot_force=0.1 * eV/Angstrom,
    max_rot_force=1.0 * eV/Angstrom,
    trial_angle=pi / 4.0,
    max_number_of_rotations=5
)
relaxed_ts = relaxed_dimer.configuration()
```

However, the simplest way to generate the initial guess is to use the *dimerFromNEB()* function. This way, one can combine the robustness of the NEB method with the efficiency of the Dimer method.

```
dimer = dimerFromNEB(nudged_elastic_band)
```

The log output of the Dimer method can give some insight into how the relaxation is proceeding.

- The rotational force indicates how well the dimer is aligned with the smallest eigenmode. If it is too large multiple rotations might be necessary. Likewise, the angle should decrease during subsequent rotations.
- The curvature must become negative to indicate that the dimer is moving towards a saddle point.

[3] R.ăA. Olsen, G.ăJ. Kroes, G.ăHenkelman, A.ăArnaldsson, and H.ăJónsson. Comparison of methods for finding saddle points without knowledge of the final states. *The Journal of Chemical Physics*, 121(20):9776–9792, 11 2004. URL: https://doi.org/10.1063/1.1809574, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/121/20/9776/19220457/9776_1_online.pdf, doi:10.1063/1.1809574.

RelaxLanczos

RelaxLanczos(*configuration*, *max_forces=None*, *max_steps=None*, *max_step_length=None*, *constraints=None*, *trajectory_filename=None*, *trajectory_object_id=None*, *initial_direction=None*, *lanczos_relative_error=None*, *lanczos_maximum_iterations=None*, *finite_difference=None*, *finite_difference_method=None*, *reference_energy=None*, *optimizer_method=None*, *trajectory_interval=None*, *restart_strategy=None*, *post_step_hook=None*, *enable_optimization_stop_file=None*, *reorthogonalize=None*)

Function for optimizing the geometry of the given configuration to a first order saddle point.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to be optimized.
- **max_forces** (*PhysicalQuantity* of type force) – The convergence criterion for the atomic forces. **Default:** 0.01*eV/Angstrom
- **max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200
- **max_step_length** (*PhysicalQuantity* of type length) – The maximum step length the optimizer may take. **Default:** 0.2*Ang
- **constraints** (*list of integers and Constraints objects*) – A list of indices of the atom with fixed positions and Constraints objects. **Default:** []
- **trajectory_filename** (*str | None*) – The filename used to store the trajectory. If the value is None then no trajectory file will be written. **Default:** None
- **trajectory_object_id** (*str | None*) – The object id to use when saving to HDF5. If the object-id already exists in the given file, it will be overwritten. **Default:** None
- **initial_direction** (*numpy array (num_atoms, 3)*) – The initial guess for minimum mode. **Default:** random direction
- **lanczos_relative_error** (*float*) – Stop the calculation when the relative change in the eigenvalue is less than this number. **Default:** 0.01
- **lanczos_maximum_iterations** (*int*) – The maximum number of iterations in the Lanczos algorithm at each step. **Default:** 20
- **finite_difference** (*PhysicalQuantity* of type distance) – The step size for the finite difference estimation of the second derivatives. **Default:** 0.01*Angstrom
- **finite_difference_method** (*Forward | Central*) – The type of finite difference used to estimate the second derivatives. **Default:** Forward
- **reference_energy** (*PhysicalQuantity* of type energy | None) – If specified, the energy displayed during the optimization is relative to the reference energy. **Default:** None
- **optimizer_method** (*OptimizerParameters*) – The optimizer to use for optimizing the structure. **Default:** LBFGS
- **trajectory_interval** (int | *PhysicalQuantity* of type time) – The resolution used in saving steps to a trajectory file. This can either be given as an integer (a value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc.) or as a time interval. **Default:** 1
- **restart_strategy** (NoRestart | *RestartFromTrajectory*) – The restart mechanism based on trajectory data saved in a given file. **Default:** RestartFromTrajectory()
- **post_step_hook** (*A callable method, function or instance*) – An optional user-defined function which will be called just after each optimization step. The signature of the function requires the arguments (step, configuration). The return status is ignored. Unhandled exceptions will terminate the optimization. **Default:** None

•**enable_optimization_stop_file** (*bool*) – Determines whether to enable a file for stopping the geometry optimization. If `True`, creation of the stop file will stop the optimization at the next step. The name of the stop file will be shown in the log output; it will be `stop-lanczos-relaxation-uniqueID`, where `uniqueID` is a randomly generated identifier for this optimization. The file must be created in the current working directory. **Default:** `True`

Returns
A tuple of the optimized saddle point configuration and the estimated lowest eigenvector at the saddle.

Return type
MoleculeConfiguration | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*, numpy.array

SingleContourIntegralParameters

SingleContourIntegralParameters(*integral_lower_bound=None*, *circle_eccentricity=None*, *logarithmic_bunching=None*, *circle_points=None*, *real_axis_point_density=None*, *real_axis_infinitesimal=None*, *real_axis_kbt_padding_factor=None*, *fermi_line_points=None*, *fermi_function_poles=None*, *contour_type=None*)

Deprecated: Use ContourParameters instead.

Representation of the contour parameters when using a single contour.

Deprecated: from v2016.0, use *ContourParameters* instead.

Parameters

•**integral_lower_bound** (*PhysicalQuantity* of type energy) – The distance between the lowest Fermi-level to the lowest energy circle contour point. **Default:** `1.5 * Hartree`

•**circle_eccentricity** (*float*) – The eccentricity of the circle contour. This should be a float between 0 and 1. 0 is a circle, 1 is a line. **Default:** `0.3`

•**logarithmic_bunching** (*float*) – Logarithmic bunching of the circle contour around the Fermi level. This should be a float between 0 and 1. 0 means no bunching, equidistant points, 1 means all bunched, centred on the Fermi Level. **Default:** `0.3`

•**circle_points** (*int*) – The number of points on the circle contour. Must be at least 3. **Default:** `30`

•**real_axis_point_density** (*PhysicalQuantity* of type energy) – Spacing between the points on bias window integration line close to the real axis. **Default:** `0.001 * Hartree`

•**real_axis_infinitesimal** (*PhysicalQuantity* of type energy) – Small imaginary shift of the bias window integration line from the real axis. **Default:** `0.001 * Hartree`

•**real_axis_kbt_padding_factor** (*float*) – The integration line is extended with the amount fk_bT below and above the electrode Fermi levels where f is this value. Must be larger than 0.0. **Default:** `5.0`

•**fermi_line_points** (*int*) – The number of points on the straight line from the Fermi energy level up to infinity. This should be an integer in the range 1 to 11 (inclusive). **Default:** `10`

•**fermi_function_poles** (*int*) – The number of poles of the Fermi function to include. Determines the imaginary shift of the straight line from the Fermi level to infinity. Must be positive. **Default:** `8`

•**contour_type** (`Left` | `Right`) – The side this contour is used for. **Default:** `Left`

Returns
A *ContourParameters* object consisting of the *SemiCircleContour* equilibrium contour and the *RealAxisContour* Non-equilibrium contour.

Return type
ContourParameters

Notes

This class is deprecated and only included for backwards compatibility.

A *SingleContourIntegralParameters* object

```
contour_parameters = SingleContourIntegralParameters(
    integral_lower_bound=5.0*Hartree,
    circle_eccentricity=0.5,
    logarithmic_bunching=0.4,
    circle_points=30,
    real_axis_point_density=0.002*eV,
    real_axis_infinitesimal=1e-05*eV,
    real_axis_kbt_padding_factor=0.6,
    fermi_line_points=10,
    fermi_function_poles=8,
    contour_type=Right)
```

is equivalent to the following *ContourParameters* object

```
eq_contour = SemiCircleContour(
    integral_lower_bound=5.0*Hartree,
    circle_eccentricity=0.5,
    logarithmic_bunching=0.4,
    circle_points=30
    fermi_line_points=10,
    fermi_function_poles=8)

neq_contour = RealAxisContour(
    real_axis_point_density=0.002*eV,
    real_axis_infinitesimal=1e-05*eV
    real_axis_kbt_padding_factor=0.6)

contour_parameters = ContourParameters(
    equilibrium_contour=eq_contour,
    non_equilibrium_contour=neq_contour,
    method=SingleContour(direction=Right))
```

SpecialThermalDisplacement

SpecialThermalDisplacement(*configuration*, *dynamical_matrix=None*, *temperature=None*, *minimum_phonon_energy=None*, *maximum_phonon_energy=None*, *configuration_index=None*, *fix_electrode_atoms=None*)

Generates a temperature dependent displacement and returns the configuration or list of configurations with displaced atomic coordinates.

Parameters

- **configuration** (*DeviceConfiguration* | *BulkConfiguration*) – The configuration to displace.

- **dynamical_matrix** (*DynamicalMatrix*) – A *DynamicalMatrix* object constructed for the same configuration given as input. **Default:** A default *DynamicalMatrix* object
- **temperature** (*PhysicalQuantity* of type temperature) – The temperature for which to generate the displacements. **Default:** `300*Kelvin`
- **minimum_phonon_energy** (*PhysicalQuantity* of type energy) – Include only phonon modes with energies above the minimum phonon energy. **Default:** `1e-3*eV`
- **maximum_phonon_energy** – Include only phonon modes with energies below the minimum phonon energy. **Default:** `1e-3*eV`
- **configuration_index** (*int*) – The method can be extended by using configurational averaging over a small number (four) of different displacement configurations specified by the configuration index which must be either 0, 1, 2, or 3. **Default:** `0`
- **fix_electrode_atoms** (*bool*) – Boolean controling whether the electrodes and electrode extensions should be kept fixed for a device configuration. **Default:** `True`

Returns
The displaced configuration or list of displaced configurations.

Return type
DeviceConfiguration | *BulkConfiguration*

Usage Examples

Calculate the *SpecialThermalDisplacement* configuration for bulk silicon and save it to a file:

```
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------
# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration0 = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = Tersoff_Si_1988b()
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25*Angstrom)
```

(continues on next page)

(continued from previous page)

```
bulk_configuration0.setCalculator(calculator)
bulk_configuration0.update()

bulk_configuration0 = OptimizeGeometry(
    bulk_configuration0,
    max_forces=0.001*eV/Ang,
    max_stress=0.01*GPa,
    max_steps=200,
    max_step_length=0.2*Ang,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
    constrain_bravais_lattice=True,
)

# Repeat the configuration 3 times in each direction
bulk_configuration = bulk_configuration0.repeat(3, 3, 3)

# ------------------------------------------------------------
# Dynamical Matrix
# ------------------------------------------------------------
dynamical_matrix = DynamicalMatrix(
    configuration=bulk_configuration,
    repetitions=(3, 3, 3),
    atomic_displacement=0.01*Angstrom,
    acoustic_sum_rule=True,
    symmetrize=True,
    finite_difference_method=Central,
    processes_per_displacement=1,
    )

# ------------------------------------------------------------
# Special Thermal Displacement
# ------------------------------------------------------------
new_configuration = SpecialThermalDisplacement(
    bulk_configuration,
    dynamical_matrix,
    temperature=300*Kelvin,
    phonon_energy_interval=[1e-3, 1]*eV,
    )

nlsave('si_special_thermal_displacement.hdf5', new_configuration)
```

si_special_thermal_displacement.py

Notes

The special thermal displacement (STD) method by Zacharias and Giustino[1] provides a simple and efficient way of calculating electron-phonon coupling by using a single (one-shot) frozen-phonon calculation in a supercell of the material. The method can for instance be used for optical absorbtion spectra including phonon-induced indirect optical transitions, band-gap renormalization at finite temperature for semiconductors and insulators, phonon-limited mobilities, and electron-phonon coupling effects in device calculations[2] . The method can be extended by using configurational averaging over a small number of different displacement configurations in the same supercell to improve the accuracy of the calculated properties. The scheme for generating the optimal displaced configurations and its formal justification are given in[Page 4170, 1].

The *SpecialThermalDisplacement* function requires a bulk configuration or a device configuration with a dynamical matrix from which the phonons of the system can be calculated. The function returns the displaced configuration or list of displaced configurations which can then be used to calculate e.g. the *OpticalSpectrum* at the specified temperature.

TimeStampedForceBiasMonteCarlo

TimeStampedForceBiasMonteCarlo(*configuration*, *constraints=None*, *trajectory_filename=None*, *steps=None*, *log_interval=None*, *method=None*, *hook_functions=None*, *write_velocities=None*, *write_forces=None*, *write_stresses=None*, *trajectory_interval=None*, *trajectory_object_id=None*, *pre_step_hook=None*, *post_step_hook=None*, *measurement_hook=None*)

Function for performing a time-stamped force-bias Monte Carlo simulation.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration* | *MDTrajectory*) – The initial configuration or a trajectory. If a trajectory is given the last configuration will be used as the initial configuration for this MC run and the trajectory will be appended to. If a trajectory is given, then `trajectory_interval` and `trajectory_object_id` should be `None`.
- **constraints** (*list of int | list of constraint objects.*) – The list of atomic indices, denoting fixed atoms, or constraint objects. *FixAtomConstraints*, *FixCenterOfMass*, *RigidBody* and *FixStrain* constraint objects are supported **Default:** `[]`.
- **trajectory_filename** (*str | None*) – The filename of the file to be used for storing the trajectory, or None if no trajectory should be written. **Default:** `None`.
- **steps** (*int*) – The number of steps to take in the simulation. **Default:** `50`.
- **log_interval** (*int*) – The resolution used in saving steps to a trajectory file ad writing the log file, where a value of 1 results in all steps being saved, and e.g. a value of 2 resulting in every other step being discarded. **Default:** 1.
- **method** (`ForceBiasMontCarlo` | *ForceBiasMonteCarloNPTBerendsen*) – The Monte Carlo method used for the simulation. **Default:** *ForceBiasMonteCarlo*.
- **hook_functions** (*HookFunctions*) – An optional HookFunctions object, which contains user-defined functions or a list of functions which will be called just before (pre-step) or just after (post-step) the forces evaluation. **Default:** `None`.
- **write_velocities** (*bool*) – Write the velocities to the trajectory file every `log_interval` steps. Since, the time-stamped force-bias Monte Carlo algorithm does not use velocities explicitly, zero velocities will be written. **Default:** False.

[1] Marios Zacharias and Feliciano Giustino. One-shot calculation of temperature-dependent optical spectra and phonon-induced band-gap renormalization. *Phys. Rev. B*, 94:075125, Aug 2016. doi:10.1103/PhysRevB.94.075125.

[2] Tue Gunst, Troels Markussen, Mattias L.ăN. Palsgaard, Kurt Stokbro, and Mads Brandbyge. First-principles electron transport with phonon coupling: Large scale at low cost. *Phys. Rev. B*, 96(16):161404, oct 2017. arXiv:1706.09290, doi:10.1103/PhysRevB.96.161404.

- **write_forces** (*bool*) – Write the forces to the trajectory file every `log_interval` steps. **Default:** True.
- **write_stresses** (*bool*) – Write the stress to the trajectory file every `log_interval` steps. **Default:** False.
- **trajectory_interval** (*int*) – The resolution used in saving steps to a trajectory file. A value of 1 results in all steps being saved; a value of 2 results in every second step being saved; etc. If `configuration` is a trajectory (i.e. this is a restart MC simulation) this parameter will use the same value as was used in the previous trajectory. **Default:** The same value as `log_interval`.
- **trajectory_object_id** (*str | None*) – The object id of the trajectory written to `trajectory_filename`. If a value of `None` is given, then an object id will be chosen automatically. **Default:** `None`.
- **pre_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just before the forces evaluation. The signature of the function requires the arguments (step, time, configuration, forces, stress). The return status is ignored. Unhandled exceptions will terminate the simulation. If a list is given the functions will be called in the given order. **Default:** `None`.
- **post_step_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called just after the forces evaluation. The signature of the function requires the arguments (step, time, configuration, forces, stress). The return status is ignored. Unhandled exceptions will terminate the simulation. If a list is given the functions will be called in the given order. **Default:** `None`.
- **measurement_hook** (*function | list of functions | None*) – An optional user-defined function or a list of functions which will be called at the end of each MD step after all constraints have been applied. The signature of the function requires the arguments (step, time, configuration). Optional arguments include, forces, local_forces (for distributed MD), stress, trajectory (the MD trajectory), temperature, pressure, potential_energy, and/or kinetic_energy. The return value should be a dictionary that maps string keys to values. The values may be numbers, numpy arrays, or PhysicalQuantities. These values are stored on the MDTrajectory and may be accessed using the measurement method. Unhandled exceptions will terminate the Molecular Dynamics evaluation. If a list is given the functions will be called in the given order. **Default:** None.

Returns
The Monte Carlo trajectory.

Return type
MDTrajectory

Usage Examples

Perform a time-stamped force-bias Monte Carlo run of 500 steps of a bulk silicon crystal, using the Stillinger-Weber potential:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 5.4306]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
fractional_coordinates = [[0.0 ,  0.0 ,  0.0 ],
                          [0.25,  0.25,  0.25],
                          [0.5 ,  0.5 ,  0.0 ],
                          [0.75,  0.75,  0.25],
                          [0.5 ,  0.0 ,  0.5 ],
                          [0.75,  0.25,  0.75],
                          [0.0 ,  0.5 ,  0.5 ],
                          [0.25,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------

potentialSet = StillingerWeber_Si_1985()
calculator = TremoloXCalculator(parameters=potentialSet)

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# ------------------------------------------------------------
# Time-Stamped Force-Bias Monte Carlo
# ------------------------------------------------------------

method = ForceBiasMonteCarlo(
    reservoir_temperature=500.0*Kelvin,
    max_atom_displacement=0.3*Ang,
)

mc_trajectory = TimeStampedForceBiasMonteCarlo(
    bulk_configuration,
    constraints=[],
    trajectory_filename='tfmc_trajectory.hdf5',
    steps=500,
    log_interval=50,
    method=method
)

bulk_configuration = mc_trajectory.lastImage()
```

tfmc_example.py

Notes

This function performs a time-stamped force-bias MonteCarlo (TFMC) simulation on the given configuration (see[1] for a summary on the TFMC method). The TFMC algorithm essentially consists of a Monte Carlo sampling of the energy landscape at a given temperature.

For details about the Monte Carlo algorithm and how the simulation parameters are specified see *ForceBiasMonteCarlo*.

Each Monte Carlo step in the TFMC algorithm is a possible realization of how the atoms might move in the potential energy landscape, starting from a given configuration. Therefore the simulation bears some similarity to Molecular Dynamics, and it is even possible to estimate a time step by which subsequent images are separated on average[2]. The time step in TFMC is typically a factor of 2-50 larger than in Molecular Dynamics simulations, however, one must keep in mind that it is not very rigorously defined and primarily suitable for long-timescale-kinetics.

The *TimeStampedForceBiasMonteCarlo* function returns an *MDTrajectory* object. After the simulation has completed, you can extract various properties of the stored snapshots, such as coordinates, or forces, by using the *MDTrajectory* class methods.

Similar to *MolecularDynamics*, you can perform custom operations on the configuration during the simulation by using the `pre_step_hook` or `post_step_hook` functionality.

The `pre-step-hook` one is invoked immediately *before* the force and stress calculations, which allows you to modify atomic positions or the cell vectors.

The `post-step-hook` is invoked *after* the force and stress calculation, which means that you can modify the forces and stress, e.g. by adding a bias potential.

Suitable functions must have the signature `(step, time, configuration, forces, stress)` where `step` denotes the integration step number, `time` is the current simulation time, and `configuration` is the current configuration, `forces` is the PhysicalQuantity array holding the current forces, and `stress` is the stress tensor. Note, that you need to modify these two arrays *in-place*, as the return status of the function is ignored. You can use the class methods of the `configuration` object (see e.g. *BulkConfiguration*) to perform operations on the atoms.

Moreover, it is possible to give a list of hook functions, which will be called sequentially at each TFMC step.

Only *FixAtomConstraints* are supported in TFMC simulations.

addDoping

addDoping(*configuration*, *indices*, *charge*, *doping_type*, *tag=None*)

Add doping to selected atoms in a configuration.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration to affect.
- **indices** (*ndarray*) – The indices of a selection of atoms in the configuration to which doping should be added.
- **charge** (*float*) – The doping charge to apply. This value is interpreted as being in the unit e/atom.
- **doping_type** (*NType | PType*) – The new doping type to apply to the tagged atoms.
- **tag** (*str | None*) – The tag to assign to this doping selection.

[1] ErikăC. Neyts and Annemie Bogaerts. Combining molecular dynamics with Monte Carlo simulations: implementations and applications. *Theor. Chem. Acc.*, 132(2):1320, 2012. doi:10.1007/s00214-012-1320-x.

[2] MaartenăJ. Mees, Geoffrey Pourtois, ErikăC. Neyts, BarendăJ. Thijsse, and Andre Stesmans. Uniform-acceptance force-bias Monte Carlo method with time scale to study solid-state diffusion. *Phys. Rev. B*, 85:134301, 2012. doi:10.1103/PhysRevB.85.134301.

Usage Examples

Add n-type doping to a selection of atoms in a Si crystal.

```
# --- Setup a configuration ---
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 16.291800000000002]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.           ,  0.           ,  0.           ],
                          [ 0.           ,  0.           ,  0.333333333333],
                          [ 0.           ,  0.           ,  0.666666666667],
                          [ 0.25         ,  0.25         ,  0.083333333333],
                          [ 0.25         ,  0.25         ,  0.416666666667],
                          [ 0.25         ,  0.25         ,  0.75          ],
                          [ 0.5          ,  0.5          ,  0.            ],
                          [ 0.5          ,  0.5          ,  0.333333333333],
                          [ 0.5          ,  0.5          ,  0.666666666667],
                          [ 0.75         ,  0.75         ,  0.083333333333],
                          [ 0.75         ,  0.75         ,  0.416666666667],
                          [ 0.75         ,  0.75         ,  0.75          ],
                          [ 0.5          ,  0.           ,  0.166666666667],
                          [ 0.5          ,  0.           ,  0.5           ],
                          [ 0.5          ,  0.           ,  0.833333333333],
                          [ 0.75         ,  0.25         ,  0.25          ],
                          [ 0.75         ,  0.25         ,  0.583333333333],
                          [ 0.75         ,  0.25         ,  0.916666666667],
                          [ 0.           ,  0.5          ,  0.166666666667],
                          [ 0.           ,  0.5          ,  0.5           ],
                          [ 0.           ,  0.5          ,  0.833333333333],
                          [ 0.25         ,  0.75         ,  0.25          ],
                          [ 0.25         ,  0.75         ,  0.583333333333],
                          [ 0.25         ,  0.75         ,  0.916666666667]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Add external potential
external_potential = AtomicCompensationCharge()
bulk_configuration.setExternalPotential(external_potential)
```

(continues on next page)

(continued from previous page)

```
# Make a selection of atoms to add doping to
selected_indices = numpy.array([2, 5, 8, 11, 14, 17, 20, 23])

# --- Add doping ---
addDoping(
    configuration=bulk_configuration,
    indices=selected_indices,
    charge=0.000200195107106,
    doping_type=N_type,
    tag='doping_0'
)
```

addDoping_example.py

See also ref.dopingcommon.notes.

allElements

allElements()

Method for obtaining a list of all the elements.

Usage Examples

Loop over all elements and print the element names and masses:

```
for element in PeriodicTable.allElements():
    # Print the element name
    print "Element name: ", element.name()
    # Print the element mass
    print "Element mass: ", element.atomicMass().inUnitsOf(atomic_mass_unit)
```

Notes

The function is part of the *PeriodicTableElement* module and can be called via

```
PeriodicTable.allElements()
```

amorphousPrebuilder

amorphousPrebuilder(*configuration*, *density=None*, *sigma=0.1*)

Parameters

- **configuration** (*BulkConfiguration*) – The base configuration to use when creating the amorphous structure. The final structure will have the same name of each element and the same shape unit cell.
- **density** (*PhysicalQuantity* of type mass per volume**3) – The density of the amorphous material. **Default:** same density as *configuration*

- **sigma** (*float*) – The position of each atom has Gaussian noise added to it. The standard deviation is sigma times the average volume per atom. **Default:** 0.1

Returns
The amorphous configuration.

Return type
BulkConfiguration

Usage Examples

Use the amorphous prebuilder to create an initial configuration for amorphous silica from an initial quartz unit cell:

```
# Setup a quartz cell.
lattice = Hexagonal(4.916*Angstrom, 5.4054*Angstrom)

elements = [Silicon, Silicon, Silicon, Oxygen, Oxygen, Oxygen, Oxygen, Oxygen,
            Oxygen]

fractional_coordinates = [[ 0.4697        ,  0.            ,  0.            ],
                          [ 0.            ,  0.4697        ,  0.666666666667],
                          [ 0.5303        ,  0.5303        ,  0.333333333333],
                          [ 0.4135        ,  0.2669        ,  0.1191        ],
                          [ 0.2669        ,  0.4135        ,  0.547567      ],
                          [ 0.7331        ,  0.1466        ,  0.785767      ],
                          [ 0.5865        ,  0.8534        ,  0.214233      ],
                          [ 0.8534        ,  0.5865        ,  0.452433      ],
                          [ 0.1466        ,  0.7331        ,  0.8809        ]]

bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Repeat the primitive cell to make a larger configuration.
bulk_configuration = bulk_configuration.repeat(3, 3, 3)

# Make the amorphous SiO2 cell with a density of 2.2 gram/cm**3.
bulk_configuration = amorphousPrebuilder(bulk_configuration, density=2.2*gram/cm**3)
```

In the full script, this configuration is then equilibrated by a 10 ps NVT molecular dynamics simulation at 2000 K and a 50 ps NPT molecular dynamics simulation at 1500 K.

`amorphous_sio2.py`

Notes

The amorphousPrebuilder function creates a rough initial guess for an amorphous structure. The function will return a new configuration with the same number and type of atoms as the original, but with the unit cell vectors scaled to give the requested density. Each atom is repositioned along a FCC grid with Gaussian noise added to its position.

The resulting configuration requires additional optimization and equilibration in order to create a realistic amorphous structure.

automaticTransitionState

automaticTransitionState(*neb*, *final_max_forces=None*, *initial_max_forces=None*, *neb_preoptimization_calculator=None*, *climbing_image=None*, *dimer=None*, *trajectory_filename_prefix=None*, *log_filename_prefix=None*, *spring_constant=None*, *spring_skew=None*, *preoptimization=None*, *optimizer_method=None*, *relax_dimer_parameters=None*, *dimer_separation=None*)

Automatically determine the transition state using NEB and Dimer methods. Up to a total of four optimization steps are performed in following order:

1. OptimizeNudgedElasticBand with a pre-optimization calculator
2. OptimizeNudgedElasticBand without climbing image
3. OptimizeNudgedElasticBand with climbing image
4. DimerRelaxation

The NEB pre-optimization is optional and can be used to prepare the NEB for the subsequent optimization steps. The NEB optimization without climbing image is always performed and can be followed by a climbing image optimization. The Dimer relaxation can be used to further optimize the transition state.

Parameters

- **neb** (*NudgedElasticBand*) – The NudgedElasticBand object serving as the initial guess for the transition state.
- **final_max_forces** (*PhysicalQuantity* of type force) – The maximum forces for the final optimization steps. **Default:** *0.05 eV/Ang*
- **initial_max_forces** (*PhysicalQuantity* of type force) – The maximum forces for the initial optimization steps. **Default:** Five times larger than the final max forces
- **neb_preoptimization_calculator** (Calculator) – The calculator to use for pre-optimizing the complete NEB. This parameter is mutually exclusive with the *preoptimization* parameter used to optimize the endpoints. **Default:** No pre-optimization
- **climbing_image** (*bool*) – Whether to use the climbing image method for the NEB optimization. A regular NEB optimization is always performed before the climbing image optimization. **Default:** False
- **dimer** (*bool*) – Whether to perform a Dimer relaxation after the NEB optimization. The Dimer relaxation is performed on the transition state image obtained from the NEB optimization. **Default:** False
- **trajectory_filename_prefix** (*str*) – The prefix used for the trajectory filenames. **Default:** No trajectory files are written
- **log_filename_prefix** (*str*) – The prefix used for the log filenames. **Default:** Log is written to the standard output
- **spring_constant** (*PhysicalQuantity* of type force/length) – The spring constant used for the NEB optimizations. **Default:** 5.0*(eV/Ang**2)

•**spring_skew** (*float*) – Ratio between the distances of the optimized lower and higher energy neighbor. A value larger than 1.0 will skew an image towards the neighbor with the higher energy. As a result images are spaced closer around transition states, which can improve convergence of the NEB optimization. Works with climbing image NEB.

•**preoptimization** (*bool*) – Flag indicating if the endpoints should be optimized before the NEB optimization. Pre-optimization is disabled when restarting. **Default:** False.

•**optimizer_method** (*FIRE* | *LBFGS* | *VelocityProjectionOptimization*) – The optimizer method used for the NEB optimizations. **Default:** *LBFGS*.

•**relax_dimer_parameters** (*RelaxDimerParameters*) – The parameters used for the Dimer relaxation.

The following parameters are also applied to the NEB optimizations: * *max_steps* * *max_step_length* * *constraints* * *trajectory_interval* * *restart_strategy*

The following parameters are ignored: * *max_forces* * *trajectory_filename* * *log_filename_prefix* * *trajectory_object_id*

•**dimer_separation** (*PhysicalQuantity* of type length) – The distance separating the displaced dimer images from the central image. **Default:** *1e-4 Angstrom*

Returns
An NEB object and a Dimer configuration.

Return type
(*NudgedElasticBand*, *DimerConfiguration*)

calculateBlochStates

calculateBlochStates(*configuration*, *quantum_numbers=None*, *quantum_number_reference=None*, *spin=None*, *k_point=None*, *diagonalization_method=None*, *density_mesh_cutoff=None*)

A method for calculating the real space wave function of multiple Bloch states.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration for which the eigenstate should be calculated.

•**quantum_numbers** (*list of int*) – A list of quantum numbers of the desired eigenstate. **Default:** `[0]`

•**quantum_number_reference** (`Absolute` | `Lumo`) – Whether the quantum numbers are referred to the lowest available energy state (`Absolute`), or to the lowest unoccupied state (`Lumo`). **Default:** `Absolute`

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin to calculate the states for. **Default:** `Spin.All`

•**k_point** (*list(3) of floats*) – The k-point in fractional coordinates that the Bloch state should be calculated for. **Default:** `[0.0, 0.0, 0.0]`

•**diagonalization_method** – Method used for diagonalizing the hamiltonian.

This parameter allows to choose between a full diagonalization solver and an iterative subspace solver. The full diagonalization solver evaluates all bands from the lowest energy one to a given number of bands above fermi level. The iterative subspace solver allows to evaluate a given number of bands around fermi level, or around an energy of choice.

The full diagonalization solver is more robust, but can be proibitively expensive for very large systems.

The iterative solver can deal with very large systems (tens of thousands atoms and beyond) and greatly outperforms when calculating a small number of eigenvalues, but it is also inherently less robust.

Note: the exact method used when selecting *FullDiagonalizationSolver* is defined by the calculator (see *AlgorithmParameters*). *IterativeDiagonalizationSolver* is not supported for *PlaneWaveCalculator*

Default: `FullDiagonalizationSolver`

•**density_mesh_cutoff** (*PhysicalQuantity* of type energy | *GridSampling* | *OptimizedFFTGridSampling*) – The mesh cutoff to be used to determine the grid sampling. The mesh cutoff must be a positive energy or a *GridSampling* object. **Default:** Specific for each calculator.

Type
FullDiagonalizationSolver | *IterativeDiagonalizationSolver*

Returns
The Bloch states.

Return type
A generator of instances of *BlochState*

Usage Examples

Create a `calculateBlochStates` generator for a set of given `quantum_numbers` and save the results.

```
# %% BlochStatesGenerator

bloch_states_generator = calculateBlochStates(
    configuration=silicon_alpha,
    quantum_numbers=(0, 2, 3)
)

for bloch_state in bloch_states_generator:
    quantum_number = bloch_state.quantumNumber()
    nlsave('results.hdf5', bloch_state, object_id=f'bloch_state_{quantum_number}')
```

bloch_states_silicon.py

Notes

This method enables to create a generator for multiple *BlochState* objects according to a set of `quantum_numbers`. These `quantum_numbers` are all specified relative to a `quantum_number_reference` which can either be the lowest unoccupied energy state, `Lumo`, or the lowest available energy state, `Absolute`. As demonstrated in the **Usage Example**, the individual *BlochState* objects are generated and calculated once looped over and afterwards saved. *calculateBlochStates()* is also applicable via the GUI using the analysis **BlochStateGenerator**.

calculateDefaultContourIntegralLowerBound

calculateDefaultContourIntegralLowerBound(*calculator_type*, *configuration*, *basis_set=None*)

Find the best default density mesh cutoff based on a configuration and basis set.

Parameters

•**calculator_type** (*LCAOCalculator* | *PlaneWaveCalculator* | *SemiEmpiricalCalculator* | class:~.*DeviceLCAOCalculator* | *DeviceSemiEmpiricalCalculator*) – The type of calculator.

•**configuration** (`AtomicConfiguration`) – The configuration.

•**basis_set** (*BasisSet*) – The basis set.

Returns
The maximum default mesh cut-off for all the elements in the configuration.

Return type
PhysicalQuantity of type energy

calculateDefaultDensityMeshCutoff

calculateDefaultDensityMeshCutoff(*calculator_type*, *configuration=None*, *basis_set=None*, *wave_function_cutoff=None*)

Find the best default density mesh cutoff based on a configuration and basis set. If no configuration is provided the default will be None, unless the wave_function_cutoff is provided, in which case the default density mesh cutoff will be calculated from the wave_function_cutoff.

Parameters

- **calculator_type** (*LCAOCalculator* | *PlaneWaveCalculator* | *SemiEmpiricalCalculator* | *DeviceLCAOCalculator* | *DeviceSemiEmpiricalCalculator*) – The type of calculator.
- **configuration** (AtomicConfiguration | None) – The configuration.
- **basis_set** (*BasisSet*) – The basis set.
- **wave_function_cutoff** (*PhysicalQuantity* of type energy) – The wave function cutoff to use.

Returns
The maximum default mesh cut-off for all the elements in the configuration.

Return type
PhysicalQuantity of type energy

calculateDefaultExactExchangeGridCutoff

calculateDefaultExactExchangeGridCutoff(*calculator_type*, *configuration=None*, *basis_set=None*, *wave_function_cutoff=None*)

Find the best default exx_grid_cutoff based on a configuration and basis set.

Parameters

- **configuration** (AtomicConfiguration | None) – The configuration.
- **basis_set** (*BasisSet*) – The basis set.
- **wave_function_cutoff** (*PhysicalQuantity* of type energy) – The wave function cutoff to use.

Returns
The maximum default mesh cut-off for all the elements in the configuration.

Return type
PhysicalQuantity of type energy

calculateDefaultWavefunctionCutoff

calculateDefaultWavefunctionCutoff(*configuration*, *basis_set=None*)

Find the best default density mesh cutoff based on a configuration and basis set.

Parameters

•**configuration** (AtomicConfiguration) – The configuration.

•**basis_set** (*BasisSet*) – The basis set.

Returns
The maximum default mesh cut-off for all the elements in the configuration.

Return type
PhysicalQuantity of type energy

calculateDensityMatrix

calculateDensityMatrix(*configuration*, *kpoint=None*, *spin=None*)

Calculate the the Fourier component of the Density Matrix.

Parameters

•**configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**spin** (Spin.Up | Spin.Down | Spin.All) – The spin component for which to perform the calculation. **Default:** Spin.All

Returns
The density matrix.

Return type
numpy.array

Usage Examples

Evaluate the density matrix of a *BulkConfiguration*.

```
density_matrix = calculateDensityMatrix(bulk_configuration)
```

Notes

•Extract the Fourier transformed density matrix of a *BulkConfiguration*.

$$D(\mathbf{k})_{\mu,\nu} = \sum_n f_n c^\dagger_\mu(n,\mathbf{k}) c_\nu(n,\mathbf{k}),$$

where f_n are Fermi occupation factors and $c_\nu(n,\mathbf{k})$ are basis function coefficients for eigenstate $n,\mathbf{k}$, i.e

$$H(\mathbf{k})\mathbf{c}(n,\mathbf{k}) = E_n S(\mathbf{k})\mathbf{c}(n,\mathbf{k})$$

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

•Consider *Spin* for details on how to handle the spin parameter.

calculateDynamicalMatrixAndOverlap

calculateDynamicalMatrixAndOverlap(*configuration*, *dynamical_matrix=None*, *kpoint=None*)

Calculate the Dynamical (D) and the overlap (S) matrices.

Parameters

•**device_configuration** (AtomicConfiguration) – The configuration to use for the calculation.

•**dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use in the calculation.

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point (0.0, 0.0, 0.0)

Returns
D as a *PhysicalQuantity* array in units of squared energy and S as a numpy.array.

Return type
tuple

Usage Examples

Evaluate on a *DeviceConfiguration*:

```
dynamical_matrix, overlap = calculateDynamicalMatrixAndOverlap(device_configuration)
```

Notes

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

calculateEffectiveMassTensor

calculateEffectiveMassTensor(*configuration*, *kpoint=None*, *spin=None*, *band_indices=None*)

Calculates the effective mass tensor $\frac{d^2E}{dk_i dk_j}$ at the given kpoint, spin, and bands.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration for which to calculate the effective mass tensor.

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point (0.0, 0.0, 0.0)

•**spin** (Spin.Up | Spin.Down | Spin.All) – The spin component for which to perform the calculation. **Default:** Spin.All

•**band_indices** (*list of non-negative int*) – Indices of the bands for which to calculate the effective mass tensor. **Default:** None

Returns
List of effective mass tensors in units of the electron mass m_e for each requested band.

Return type
list of *PhysicalQuantity*

Usage Examples

Evaluate the effective mass tensor for a bulk configuration:

```
effective_mass_tensor = calculateEffectiveMassTensor(
    configuration=bulk_configuration,
    kpoint=[0.5, 0.5, 0.0],
    spin=Spin.Up,
    band_indices=[14],
    )
effective_mass_xy = effective_mass_tensor[0,1]
```

Notes

The effective mass tensor is calculated through the second order derivative of the energy band $\epsilon_{n\mathbf{k}}$ with respect to the $\mathbf{k}$ vector,

$$M_{\alpha\beta} = \frac{\partial^2 \epsilon_{n\mathbf{k}}}{\partial k_\alpha \partial k_\beta},$$

which is a 3x3 tensor. The effective mass tensor α, β is then given by the inverse of $\mathbf{M}$,

$$m^*_{\alpha\beta} = \hbar^2 M^{-1}_{\alpha\beta}.$$

The second order derivative of the energy band is obtained through an expansion of the Kohn-Sham equation in terms of a momentum displacement $\delta\mathbf{k}$. To second order of this expansion, one obtains

$$\frac{\partial^2 \epsilon_{n\mathbf{k}}}{\partial k_\alpha \partial k_\beta} = \langle \psi_{n\mathbf{k}} \left| \frac{\partial^2 H_\mathbf{k}}{\partial k_\alpha \partial k_\beta} - \epsilon_{n\mathbf{k}} \frac{\partial^2 S_\mathbf{k}}{\partial k_\alpha k_\beta} - \frac{\partial \epsilon_{n\mathbf{k}}}{\partial k_\alpha} \frac{\partial S_\mathbf{k}}{\partial k_\beta} - \frac{\partial \epsilon_{n\mathbf{k}}}{\partial k_\beta} \frac{\partial S_\mathbf{k}}{\partial k_\alpha} \right| \psi_{n\mathbf{k}} \rangle + 2 \sum_{m \neq n} \frac{\langle \psi_{n\mathbf{k}} \left| \frac{\partial H_\mathbf{k}}{\partial k_\alpha} - \epsilon_{n\mathbf{k}} \frac{\partial S_\mathbf{k}}{\partial k_\alpha} \right| \psi_{m\mathbf{k}} \rangle \langle \psi_{m\mathbf{k}} \left| \frac{\partial H_\mathbf{k}}{\partial k_\beta} - \epsilon_{n\mathbf{k}} \frac{\partial S_\mathbf{k}}{\partial k_\beta} \right| \psi_{n\mathbf{k}} \rangle}{\epsilon_{n\mathbf{k}} - \epsilon_{m\mathbf{k}}}$$

where $H_\mathbf{k}$ and $S_\mathbf{k}$ are the Fourier components of the Hamiltonian and overlap matrices, respectively.

Special care has is taken to deal with degenerate bands. The degenerate states are first rotated such that the perturbation becomes diagonal in these states. The sum in above equation then extends only over those states $|m\rangle$ that do not belong to the set of degenerate states.

calculateEigensolutions

calculateEigensolutions(*configuration*, *kpoint=None*, *diagonalization_method=None*)

Calculates eigenvalues and eigenvectors (in orbital space) for an LCAO or Semi-Empirical calculation.

Parameters

- **configuration** (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration with attached calculator for which the eigenstate should be calculated.
- **kpoint** (*list(3) of floats*) – The k-point in fractional coordinates that the Bloch state should be calculated for. **Default:** `[0.0, 0.0, 0.0]`

•**diagonalization_method** – Method used for diagonalizing the hamiltonian.

This parameter allows to choose between a full diagonalization solver and an iterative subspace solver. The full diagonalization solver evaluates all bands from the lowest energy one to a given number of bands above fermi level. The iterative subspace solver allows to evaluate a given number of bands around fermi level, or around an energy of choice.

The full diagonalization solver is more robust, but can be proibitively expensive for very large systems.

The iterative solver can deal with very large systems (tens of thousands atoms and beyond) and greatly outperforms when calculating a small number of eigenvalues, but it is also inherently less robust.

Note: if *FullDiagonalizationSolver* all eigenvalues are calculated, regardless the value of `bands_above_fermi_level`.

Default: `FullDiagonalizationSolver`

Type
FullDiagonalizationSolver | *IterativeDiagonalizationSolver*

Returns
The eigensolutions.

Return type
Eigensolutions

calculateElectrodeHamiltonianAndOverlap

calculateElectrodeHamiltonianAndOverlap(*configuration*, *kpoint=None*, *spin=None*, *contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*)

Calculate the Fourier transformed Hamiltonian (H) and the overlap (S) matrices of the electrode.

Parameters

•**configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.

•**kpoint** (*tuple of floats*) – The kpoint to use for the evaluation as three floats representing fractional reciprocal space coordinates. Can only be specified for a bulk system. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**spin** (`Spin`) – The spin flag.

•**contribution** (`Left | Right`) – Which electrode (side) should be considered. Only `Left` is allowed for a *SurfaceConfiguration*. **Default:** `Left`

Returns
H as a *PhysicalQuantity* in units of energy and S as a `numpy.array`.

Return type
tuple

calculateFiniteDifferenceLinearResponseHubbardU

calculateFiniteDifferenceLinearResponseHubbardU(*configuration*, *shell_indices*, *hubbard_term*, *supercell_repetitions*, *potential_displacement=PhysicalQuantity(0.02, eV)*)

Calculate Hubbard Us for a selection of shells using the finite difference linear response method.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration to calculate the Us for.

- **shell_indices** (*dict*) – A map between elements and the list of shell indices to calculate the Us for.
- **hubbard_term** (Onsite | Dual) – The projection method to use for calculating shell occupations.
- **supercell_repetitions** (*tuple of three integers*) – The supercell size.
- **potential_displacement** (PhysicalQuantity of type energy) – The finite difference potential displacement to use.

Returns
A list of calculated Us for each atom in the configuration.

Return type
list

Notes

The Hubbard energy contribution is given by

$$E_{\mathrm{Hubbard-U}} = \sum_{a,i} \frac{U_{a,i}}{2} \sum_{m,\sigma} \left\{ [n_{a,i}]^{\sigma\sigma}_{mm} - \sum_{m',\sigma'} [n_{a,i}]^{\sigma\sigma'}_{mm'} [n_{a,i}]^{\sigma\sigma'}_{m'm} \right\},$$

where $[n_{a,i}]^{\sigma\sigma'}_{mm'} = \sum_\mu f_\mu \langle \psi_\mu | [\hat{P}_{a,i}]^{\sigma\sigma'}_{mm'} | \psi_\mu \rangle$ with $[\hat{P}_{a,i}]^{\sigma\sigma'}_{mm'}$ the specific projection operator for a shell i on an atom a.

This function calculates the Hubbard Us, $U_{a,i}$, for selected atoms and shells in a system using linear response theory based on constrained DFT[1]. In practice this is done by introducing a term in the Hamiltonian given by

$$\hat{V}_{\mathrm{lr-U}} = \sum_{a,i} \alpha_{a,i} \hat{P}_{a,i},$$

where $\hat{P}_{a,i} = \sum_{mm'} \sum_{\sigma\sigma'} [\hat{P}_{a,i}]^{\sigma\sigma'}_{mm'}$ is the projection operator for a given atom + shell (*Dual* or *Onsite*) and $\alpha_{a,i}$ is an arbitrary potential displacement. One then calculates the shell occupation response function corresponding to this potential displacement

$$\chi_{ai;bi} = \frac{\partial n_{a,i}}{\partial \alpha_{b,i}} \approx \frac{n_{a,i}(\Delta\alpha_{b,i}) - n_{a,i}(-\Delta\alpha_{b,i})}{2\Delta\alpha_{b,i}}$$

where $n_{a,i}(\Delta\alpha_{b,i}) = \sum_m n^{a,i}_{mm}(\Delta\alpha_{b,i})$ is the occupation of shell i on atom a calculated for a self-consistent DFT calculation with a small projection potential displacement on shell i on atom b of size $\Delta\alpha_{b,i}$. The Hubbard U can then be shown to be given by

$$U_{a,i} = (\chi_0^{-1} - \chi^{-1})_{ai,ai}$$

where χ_0 is the response matrix as given above, but where the Hamiltonian is not updated, in practice found by calculating the shell occupations at the first SCF iteration.

For systems with periodic boundary conditions the pertubation potential will get artificically repeated. In theory one should apply the perturbative potential to a single atom/shell atom a time for the entire infinite system. This cannot of course be done in practice with this method, so instead one in practive performs the perturbative calculations in a supercell of a size big enough that the spatial effect of the perturbation has decayed so as to not overlap with that of the neighboring supercells. One will often have to perform calculations of increasing supercell size to check for convergence.

[1] M.ăCococcioni and S.ădeăGironcoli. Linear response approach to the calculation of the effective interaction parameters in the LDA+U method. *Phys. Rev. B*, 71:035105, Jan 2005. doi:10.1103/PhysRevB.71.035105.

Note: For semiconductors and insulators one should use an occupation function smearing of practically zero. This is because the desired shell may be part of the valence band and when introducing the finite perturbation, this may lead to isolated states above/below the valence band for the two finite perturbations. These states will get a perturbation dependent occupation due to the non-zero smearing, and this artificial occupation difference can become a significant contribution to the shell occupation response function.

Note: It can also be required to decrease the SCF convergence threshold as the small finite perturbation may only lead to small changes in the density and Hamiltonian. As the function calculates the finite difference derivatives of the shell occupations it is important that the shell occupations are converged within a tolerance that is mcuh smaller than the finite difference.

Usage Examples

The following script calculates the U values for anti-ferromagnetic NiO using the Dual projection and a supercell of size $3 \times 3 \times 3$, prints out the values and saves them to a pickle file for later use.

```
# Set up anti-ferromagnetic NiO
configuration = BulkConfiguration(
    bravais_lattice=Rhombohedral(5.10522 * Angstrom, 34.12513143396482 * Degrees),
    elements=[Nickel, Nickel, Oxygen, Oxygen],
    fractional_coordinates=[
        [ 0.125,  0.125,  0.125],
        [ 0.625,  0.625,  0.625],
        [ 0.375,  0.375,  0.375],
        [ 0.875,  0.875,  0.875]
    ]
)

calculator = LCAOCalculator(
    exchange_correlation=SGGA.PBE,
    basis_set=BasisGGAPseudoDojo.Medium,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
        k_point_sampling=KpointDensity(2.0 * Angstrom),
        occupation_method=FermiDirac(1e-6 * eV) # We need a low smearing to avoid␣
↪artificial occupations.
    ),
    iteration_control_parameters=IterationControlParameters(
        tolerance=1e-5, # We need a low tolerance to ensure convergence with a small␣
↪perturbation
        non_convergence_behavior=StopCalculation(),
    ),
    checkpoint_handler=NoCheckpointHandler
)

# Set calculator and antiferromagnetic spin ordering.
configuration.setCalculator(
    calculator,
    initial_spin=InitialSpin(scaled_spins=[1.0, -1.0, 0.0, 0.0]))
```

(continues on next page)

(continued from previous page)

```
# Calculate the standard DFT ground state and save for later use.
configuration.update()
nlsave('NiO_PBE.hdf5', configuration)

# Calculate the Hubbard U's for the 3d shell.
fdlr_us_per_atom = calculateFiniteDifferenceLinearResponseHubbardU(
    configuration=configuration,
    shell_indices={Nickel: [3]}, # Calculate U for the 3d shell of Nickel = 3rd basis␣
↪orbital
    hubbard_term=Dual, # Dual projection is most local
    supercell_repetitions=(1, 1, 1),
    potential_displacement=0.1 * eV
)

for atom_index, u_values_per_shell in fdlr_us_per_atom:
    nlprint('{:d}: U for 3d shell: {:.3f} eV'.format(atom_index, u_values_per_shell[0].
↪inUnitsOf(eV)))

# Save the calculated U's in a "pickle" file for later use.
import pickle
with open('NiO_fdlr_us_per_atom.pckl', 'wb') as fd:
    pickle.dump(fdlr_us_per_atom, fd)
```

The following script loads the calculated U values from the pickle file and uses the function *createLocalDFTUBasisSet()* to perfom a density of states calculation based on DFT+U with the Dual projection and the calculates U values.

```

# Get the configuration with PBE ground state
pbe_configuration = nlread('NiO_PBE.hdf5', BulkConfiguration)[0]

# Perform reference DOS for PBE
pbe_dos = DensityOfStates(pbe_configuration)
nlsave('NiO_dos.hdf5', pbe_dos, object_id='NiO DOS PBE')

# Get the previously calculated U values.
import pickle
with open('NiO_fdlr_us_per_atom.pckl', 'rb') as fd:
    fdlr_u_per_atom = pickle.load(fd)

pbe_calculator = pbe_configuration.calculator()

# Create a new configuration and basis set with the calculated U values.
configuration_with_tags, basis_set_with_u = createLocalDFTUBasisSet(
    configuration=pbe_configuration,
    basis_set=pbe_calculator.basisSet(),
    shell_indices={Nickel: [3]},
    atom_u_values=fdlr_u_per_atom
)

# Set up spin-polarized PBE with Hubbard U using Dual projection.
xc_with_dftu = ExchangeCorrelation(
```

(continues on next page)

(continued from previous page)

```
    exchange=PerdewBurkeErnzerhofExchange,
    correlation=PerdewBurkeErnzerhofCorrelation,
    hubbard_term=Dual,
    number_of_spins=2
)

# Create a copy of the original calculator but change xc to DFT+U and use the basis set
# with U values.
calculator_with_u = pbe_calculator(
    exchange_correlation=xc_with_dftu,
    basis_set=basis_set_with_u)

configuration_with_tags.setCalculator(
    calculator_with_u,
    initial_spin=InitialSpin(scaled_spins=[1.0, -1.0, 0.0, 0.0]))

# Perform the DFT+U calculation
configuration_with_tags.update()

# Calculate density of states
density_of_states_with_u = DensityOfStates(configuration=configuration_with_tags)
nlsave('NiO_dos.hdf5', density_of_states_with_u, object_id='NiO DOS PBE+U')
```

calculateGreenFunctionComponent

calculateGreenFunctionComponent(*device_configuration*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.All'>*, *contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the Green Function component, $G\Gamma G^{\dagger}$.

Parameters

- **device_configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.
- **energy** (*PhysicalQuantity* of type energy.) – Absolute energy for the calculation **Default:** `0.0 * eV`
- **kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin flag. **Default:** `Spin.All`
- **contribution** (`Left` | `Right`) – Which electrode (side) should be considered. Only the `Left` contribution is allowed for a *SurfaceConfiguration*. **Default:** `Left`
- **eta** (*PhysicalQuantity* of type energy.) – Complex energy for electrode transform matrices, imag part. **Default:** `1.0e-6 * Hartree`
- **self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
The Green Function Component as an array

Return type
PhysicalQuantity of type inverse energy

Usage Examples

Calculate the left Greens function component of a *DeviceConfiguration*:

```
energy=1.0*eV
G_L = calculateGreenFunctionComponent(device_configuration, energy, contribution=Left)
```

Equivalent calculation from the basic H, S variables:

```
energy=1.0*eV
H,S = calculateHamiltonianAndOverlap(device_configuration)
S_L = calculateSelfEnergy(device_configuration, energy, contribution=Left)
S_L = S_L.inUnitsOf(Hartree)
S_R = calculateSelfEnergy(device_configuration, energy, contribution=Right)
S_R = S_R.inUnitsOf(Hartree)

# Setup E*S-H-S_L-S_R
lhs = energy.inUnitsOf(Hartree)*S-H.inUnitsOf(Hartree)
n=S_L.shape[0]
lhs[:n,:n] -= S_L
n=S_R.shape[0]
lhs[-n:,-n:] -= S_R
G = numpy.linalg.inv(lhs)

# Calculate Gamma_L
g_L = 1j*(S_L-numpy.conj(S_L.transpose()))

# Do the G gamma G^dagger
n = g_L.shape[0]
G_L = numpy.dot(numpy.dot(G[:,:n],g_L), numpy.conj(G.transpose())[:n,:])*Hartree
```

Notes

•Calculates the spectral density matrix components

$$G^{L,R}(E,\mathbf{k}) = G^{R}(E,\mathbf{k})\Gamma^{L,R}(E,\mathbf{k})G^{R}(E,\mathbf{k})^{\dagger},$$

where G^R is the retarded Green function and $\Gamma = i(\Sigma - \Sigma^{\dagger})$ is the broadening function.

$G^{L,R}$ are closely related to $G^{<}$.

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

•Consider *Spin* for details on how to handle the spin parameter.

calculateHamiltonianAndOverlap

calculateHamiltonianAndOverlap(*configuration*, *kpoint=None*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.All'>*)

Calculate the Fourier transformed Hamiltonian (H) and the overlap (S) matrices.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.
- **kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component for which to perform the calculation. **Default:** `Spin.All`

Returns
H as a *PhysicalQuantity* array and S as a `numpy.array`.

Return type
tuple

Usage Examples

Evaluate a simple molecular system:

```
hamiltonian, overlap = calculateHamiltonianAndOverlap(molecule_configuration)
```

Evaluate a bulk system in a non-Gamma kpoint:

```
hamiltonian, overlap = calculateHamiltonianAndOverlap(bulk_configuration, kpoint=(0.1, 0.
↪0, 0.0))
```

Notes

- Calculates Fourier transformed Hamiltonian, $H(\mathbf{k})$ and overlap matrix $S(\mathbf{k})$. The real space Hamiltonian is defined by $H_{ij} = \langle\phi_i|\hat{H}_{1el}|\phi_j\rangle$, and the overlap matrix $S_{ij} = \langle\phi_i|\phi_j\rangle$, where ϕ_i are the basis functions. The Fourier transform is obtained by summing over all the cells in the periodic system

$$M(\mathbf{k})_{\mu\nu} = \sum_{\mathbf{R}} M_{(\mu,\mathbf{0}),(\nu,\mathbf{R})} e^{i\mathbf{k}\cdot\mathbf{R}},$$

where $\mathbf{R}$ is given relative to the central cell, $\mathbf{R} = \mathbf{0}$.

- The calculator must be density matrix-based for this function to perform successfully.
- Consider *Spin* for details on how to handle the spin parameter.

calculateLCAOBasisSpilling

calculateLCAOBasisSpilling(*configuration*, *k_point_sampling=None*, *basis=None*, *bands_above_fermi_level=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.All'>*, *processes_per_kpoint=None*)

Method for calculating the spilling factors per band (and spin) for a given configuration with an attached plane wave calculator and a given LCAO basis.

param configuration
The configuration with an attached plane wave calculator.

type configuration
BulkConfiguration

param k_point_sampling
The k-point sampling to use in the sum over k. By default, the k-points of the ground state configuration are used. The eigensystem is solved again, if different k-points are requested. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.

type k_point_sampling
sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*

param basis
The LCAO basis to project onto. It must contain basis functions for the elements in the configuration. If it is set to None, the LCAO basis attached to the calculator of the configuration is used. **Default:** None.

type basis
None | *BasisSet*

param bands_above_fermi_level
The number of bands above the Fermi level to include in the sum. By default all unoccupied bands present in the ground state calculation are used. The eigensystem is solved again, if more unoccupied bands are requested. **Default:** `All`

type bands_above_fermi_level
`All` | int

param processes_per_kpoint
The number of processes used for one k-point. **Default:** None.

type processes_per_kpoint
None | int | `Automatic`

returns
The spilling factors per band as a numpy array of floats. In the case of a spin-polarized calculation, the shape is (2, N), otherwise its shape is (1, N).

rtype
numpy.ndarray

Usage Examples

Calculate the average spilling factor for a Medium and High silicon basis set.

```
# Set up lattice
lattice = FaceCenteredCubic(5.4306 * Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates,
    )

k_point_sampling = MonkhorstPackGrid(
    na=7,
    nb=7,
    nc=7,
    )

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

# PlaneWaveCalculator with a high wave function cutoff to get a complete plane wave␣
↪basis.
calculator = PlaneWaveCalculator(
    wave_function_cutoff=50*Hartree,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()

# For each band get the LCAO basis spilling factor for a Medium basis.
# Only include the occupied states.
spilling_per_band_medium = calculateLCAOBasisSpilling(
    bulk_configuration,
    basis=BasisGGAPseudoDojo.Medium,
    bands_above_fermi_level=0,
)

# Total averaged spilling.
spilling_medium = spilling_per_band_medium.sum() / spilling_per_band_medium.size

# For each band get the LCAO basis spilling factor for a High basis.
```

(continues on next page)

(continued from previous page)

```
# Only include the occupied states.
spilling_per_band_high = calculateLCAOBasisSpilling(
    bulk_configuration,
    basis=BasisGGAPseudoDojo.High,
    bands_above_fermi_level=0,
)

# Total averaged spilling.
spilling_high = spilling_per_band_high.sum() / spilling_per_band_high.size

print('Spilling per band, Medium (%) : ', spilling_per_band_medium * 100)
print('Spilling per band, High   (%) : ', spilling_per_band_high * 100)
print('')
# Print the average spilling factors
print('Average spilling, Medium (%) : ', spilling_medium * 100)
print('Average spilling, High   (%) : ', spilling_high * 100)
```

`lcao_basis_spilling.py`

The final output of the script is:

```
Spilling per band, Medium (%) :  [[ 0.07159806  0.04691684  0.04034596  0.04223707]]
Spilling per band, High   (%) :  [[ 0.03255014  0.01255693  0.02256928  0.0250443 ]]

Average spilling, Medium (%) :  0.0502744832443
Average spilling, High   (%) :  0.0231801630042
```

showing that the Medium basis set has a spilling factor of 0.050 % while the High basis has a smaller spilling of 0.023 %. The printed spillings per band show that the lowest band (first index) has the largest spilling factor for both the Medium and High basis sets.

Notes

Following[1], we define the spilling factor for band i as

$$S_i = \frac{1}{N_k} \sum_k^{N_k} w_k \langle \psi_i(k) | 1 - P(k) | \psi_i(k) \rangle,$$

where w_k is the weight of k-point k and the projection operator P is defined as

$$P(k) = \sum_{\mu\nu} |\phi_\mu(k)\rangle (O(k)^{-1})_{\mu\nu} \langle \phi_\nu(k)|$$

with $|\phi_\mu(k)\rangle$ being the Fourier transform of the basis orbital μ and O being the basis orbital overlap matrix defined as

$$O_{\mu\nu}(k) = \langle \phi_\mu(k) | \phi_\nu(k) \rangle.$$

[1] D.ăSanchez-Portal, E.ăArtacho, and J.ăM. Soler. Projection of plane-wave calculations into atomic orbitals. *Solid State Communications*, 95(10):685–690, 1995. URL: http://www.sciencedirect.com/science/article/pii/003810989500341X, doi:https://doi.org/10.1016/0038-1098(95)00341-X.

The total spilling factor is given by the average over the considered bands, i.e.

$$S = \frac{1}{N_b} \sum_{i}^{N_b} S_i.$$

The function `calculateLCAOBasisSpilling` can be used to calculate the completeness of a a particular basis set. The smaller the spilling factor, the more complete is the basis set.

It also allows for a per band analysis since `calculateLCAOBasisSpilling` returns the spilling factor for each band. The number of unoccupied bands included in the calculation can be set in the function call as:

```
spilling_per_band_high = calculateLCAOBasisSpilling(
    configuration,
    basis=BasisGGAPseudoDojo.Medium,
    bands_above_fermi_level=5,
)
```

where five unoccupied bands above the Fermi level would be included.

calculateLinearResponseSpinTransferTorkance

calculateLinearResponseSpinTransferTorkance(*configuration*, *energy=None*, *kpoints=None*, *self_energy_calculator=None*, *energy_zero_parameter=None*, *infinitesimal=None*)

Calculate the atom-resolved spin transfer torkance in the linear response approximation.

Parameters

- **configuration** (*DeviceConfiguration*) – The configuration to calculate the spin torque transfer torkance. The configuration must have a calculator with non-collinear spin.
- **energy** (*PhysicalQuantity* of type energy) – The energy for which the spin transfer torque should be calculated. **Default:** `0.0 * eV`
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which the spin transfer torque should be calculated. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The self energy calculator to use. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **energy_zero_parameter** (`AverageFermiLevel | AbsoluteEnergy`) – Specifies the choice for the energy zero. **Default:** `AverageFermiLevel`
- **infinitesimal** (*PhysicalQuantity* of type energy) – Small energy, used to move the spin transfer torque calculation away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6 * eV`

Returns
A tuple with the total torkance, the torkance per k-point, the k-points and weights used for the integration.

Return type
A tuple `(torkance, torkance_per_kpoint, kpoints, weights)` where `torkance` is a `(3, natoms)numpy.array`, torkance_per_kpoint is a `(3, natoms, nk)` `numpy.array`.

Usage Examples

A usage example is shown together with a finite bias calculation in *calculateNonSelfConsistentFiniteBiasSpinTransferTorque* documentation.

Notes

The linear response spin transfer torkance is evaluated as[1]:

$$\tau_a = \sum_{\alpha \in a} \sum_{\beta} \Re[\frac{\partial \rho_{CD,\alpha\beta}}{\partial V_b} \sigma \times \mathbf{B}_{XC,\alpha\beta}]$$

where $\rho_{CD} = \rho_{neq} - \rho_{eq}$ is the current driven density matrix[2]. Assuming a left-right symmetric junction resulting in a small bias drop $\pm V_b/2$, and slowly varying spectral density around the Fermi energy E_F, we use:

$$\frac{\partial \rho_{CD,\alpha\beta}}{\partial V_b} = \frac{1}{4\pi} \mathbf{G}_0(E_F)[\mathbf{\Gamma}_R(E_F) - \mathbf{\Gamma}_L(E_F)]\mathbf{G}_0^\dagger(E_F)$$

This is similar, but not identical, to the implementation in the analysis object *SpinTransferTorque*. Moreover this method returns the atom-resolved torkance, while *SpinTransferTorque* evaluates the torkance in real space.

calculateLocalScreenedCoulombCorrectionHubbardU

calculateLocalScreenedCoulombCorrectionHubbardU(*configuration*, *shell_indices*)

Calculate Hubbard U values using the Local Screened Coulomb Correction method.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The system to calculate the Us for.
- **shell_indices** (*dict*) – A map between elements and the list of shell indices to calculate the Us for.

Returns
A list of tuples *(atom_index, u_values)* with the atom index and the corresponding calculated Us.

Return type
list

Notes

The DFT+U method attempts to include the exact onsite interaction energy between electrons in certain selected shells, that is poorly captured by mean-field theories like DFT. This interaction energy is proportional to the term

$$\langle nm_1, nm_2|\hat{V}_{\mathrm{sc}}|nm_3, n_4\rangle = \iint \phi^*_{nm_1}(\mathbf{r})\phi^*_{nm_2}(\mathbf{r}')v_{\mathrm{sc}}(\mathbf{r},\mathbf{r}')\phi_{nm_3}(\mathbf{r})\phi_{nm_4}(\mathbf{r}'')d\mathbf{r}d\mathbf{r}',$$

where $v_{\mathrm{sc}}(\mathbf{r},\mathbf{r}')$ is the effective screened Coulomb potential.

[1] Ivan Rungger, Andrea Droghetti, and Maria Stamenova. *Non-equilibrium Greens Function Methods for Spin Transport and Dynamics*, pages 1–27. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-42913-7_75-1, doi:10.1007/978-3-319-42913-7_75-1.

[2] BranislavăK. Nikolic, Kapildeb Dolui, MarkoăD. Petrovic, Petr Plechac, Troels Markussen, and Kurt Stokbro. *First-Principles Quantum Transport Modeling of Spin-Transfer and Spin-Orbit Torques in Magnetic Multilayers*, pages 1–35. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-50257-1_112-1, doi:10.1007/978-3-319-50257-1_112-1.

In the Local Screened Coulomb Correction method[1] one approximates the screened Coulomb potential with a simple isotropic Yukawa potential

$$v_{\mathrm{sc}}(\mathbf{r},\mathbf{r}') = \frac{e^{-\lambda|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|},$$

where one takes the screening parameter, λ, from the Thomas-Fermi screening model

$$\lambda = 2\left(\frac{3}{\pi}\rho\right)^{1/6}$$

with ρ being the valence electron density.

In QuantumATK the U for a given orbital on an atom is calculated using the same screening parameter for the entire atom calculated by averaging the local screening parameter over a small sphere around the atom, weighting it by the local valence density:

$$\bar{\lambda}_i = \frac{\int_{\Omega_i}\lambda(\mathbf{r})\rho(\mathbf{r})d\mathbf{r}}{\int_{\Omega_i}\rho(\mathbf{r})d\mathbf{r}}$$

The radius of the integration spheres are chosen to be the covalent radius for the specific atom.

Note: The Hubbard U values obtained with the LSCC method are not directly comparable to those obtained with other methods and codes as they strongly depend on the chosen pseudopotential and the shape of the basis set orbitals. For instance for the 3d shell of Mn in MnO a U value of 7.3 eV is obtained with the Medium basis set for the PseudoDojo pseudopotential, while with the Double Zeta Polarized basis set for the FHI pseudopotentials a value of 4.9 eV is obtained. The validity and utility of the calculated Hubbard Us should should mostly be based on their ability to reproduce physical properties in an actual DFT+U calculation.

Attention: The LSCC assumes that Hubbard energy term is based on projection on the local basis set orbitals. Such a projection method is not implemented in QuantumATK, but one can use the *Dual* which is the closest to such a projection. We do not recommend the *Onsite* projection with the Us obtained with the LSCC method, as the projection orbitals do not resemble those the Us were calculated for.

Usage Examples

The following script snippet calculates and prints the LSCC U value for the 3d shell of Manganese in MnO.

```
# Calculate LSCC U values for Mn 3d shell

shell_indices = {Manganese: [3]} # 3d orbital has index 3 in the PseudoDojo Medium basis␣
↪set

lscc_u_per_atom = calculateLocalScreenedCoulombCorrectionHubbardU(
    configuration=manganosite,
    shell_indices=shell_indices
)

for atom_index, u_values_per_shell in lscc_u_per_atom:
```

(continues on next page)

[1] Litora Mattis. Lorem ipsum odor amet, consectetuer adipiscing elit. Proin donec duis donec aliquet congue dictumst eu. *Ad magnis nam*, :, 2019. doi:.

(continued from previous page)

```
    nlprint('{:d}: U for 3d shell: {:.3f} eV'.format(atom_index, u_values_per_shell[0].
↪inUnitsOf(eV)))
```

For non-isotropic systems with many atoms, one will obtain different U values for different atomic sites. In order to give different U values to different atoms in a DFT calculation one has to use different basis set objects for each different atom, which in practice is done by tagging the atom and adding a specific basis set for that particular tag. This can conveniently be done programmatically with the utility function *createLocalDFTUBasisSet()* as seen in the following:

```
# Create a new configuration where the atoms with U values are tagged and create basis␣
↪sets for
# these tags which sets the U value to the ones calculated above.
configuration_with_tags, basis_set_with_u = createLocalDFTUBasisSet(
    configuration=manganosite,
    basis_set=calculator.basisSet(),
    shell_indices=shell_indices,
    atom_u_values=lscc_u_per_atom
)

# Create the XC functional with +U using the 'Dual' projection method.
xc_with_dftu = ExchangeCorrelation(
    exchange=PerdewBurkeErnzerhofExchange,
    correlation=PerdewBurkeErnzerhofCorrelation,
    hubbard_term=Dual,
    number_of_spins=2
)

# Create a copy of the original calculator but change xc to DFT+U and use the basis set
# with U values.
calculator_with_u = calculator(
    exchange_correlation=xc_with_dftu,
    basis_set=basis_set_with_u)

configuration_with_tags.setCalculator(calculator_with_u)

# Perform the DFT+U calculation
configuration_with_tags.update()
```

For full calculation example see the script `MnO_LSCC_U.py`.

calculateNonSelfConsistentFiniteBiasSpinTransferTorque

calculateNonSelfConsistentFiniteBiasSpinTransferTorque(*configuration*, *biases*, *potential_drop_region=None*, *potential_profile=None*, *kpoints=None*)

Calculate the atom-resolved spin transfer torque at finite bias, within a non-selfconsistent approximation for the non equilibrium potential.

The non-equilibrium potential is modelled as a ramp as specified in the parameter *potential_drop_region*, or as a user-defined piecewise function as defined in *potential_profile*.

Parameters

- **configuration** (*DeviceConfiguration*.) – The configuration to calculate the spin torque transfer torkance. The configuration must have a calculator with non-collinear spin.
- **biases** – A list of the bias voltages for which the torque should be calculated. **Default:** `[0.1] * Volt`
- **potential_drop_region** (A list of integers or a list of *PhysicalQuantity* of type position.) – Defines the potential model as a linear ramp, where the potential drop occurs in a spatial region specified by the input parameters. The parameters can be either the first and last atom where the potential drop occurs, or the initial and final coordinates along the C axis for the potential ramp. In an Magnetic Tunneling Junction geometry, the indices or the z coordinates of the metal atoms at the interface with the insulating barrier should be used. This argument is mutually exclusive to *potential_profile*.
- **potential_profile** (List of (*PhysicalQuantity* of type length, float)) – Defines the potential model as a piecewise linear function, defined as a list of (coordinate, value) pairs, where the values define the potential drop in an interval $(+V_b/2, V_b/2)$ with V_b the applied bias.

  For example, a list *[(0*Ang, 1), (5*Ang, 1), (10*Ang, 0.), (15*Ang, 0.), (20*Ang, -1.), (25*Ang, -1.)]* represents a potential ramp from $V_b/2$ to $-V_b/2$ between coordinates 5 and 20 Angstrom, with a flat 0 potential region in the middle. It is required that the leftmost and rightmost values are 1 and -1 respectively.

  This parameter is mutually exclusive to *potential_drop_region* and can be used to define more complex potential drop profiles than a single ramp, e.g. in presence multiple tunneling barriers.
- **kpoints** (sequence (size 3) of int | *MonkhorstPackGrid* | *KpointDensity*) – The k-points for which the spin transfer torque should be calculated. **Default:** The Monkhorst-Pack grid used for the self-consistent calculation.

Returns
A tuple with the total torque per bias point, the torque per k-point and bias point, the k-points and weights used for the integration.

Return type
A tuple `(torque, torque_per_kpoint, kpoints, weights)` where `torque` and `torque_per_kpoint` are a list of *PhysicalQuantity* of type energy of size `(3, natoms)` and `(3, natoms, nk)` per each bias point.

Usage Examples

In this example we will calculate the spin transfer torque (STT) in a a Fe-MgO-Fe magnetic tunneling junction (MTJ) for an angle of 90 degree between the magnetization in the fixed (left) and free (right) magnetic regions.

As starting point, we define the configuration and calculate the solution for a parallel spin configuration.

```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Two-probe Configuration
# ------------------------------------------------------------
# ------------------------------------------------------------
# Left Electrode
# ------------------------------------------------------------

# Set up lattice
vector_a = [2.866, 0.0, 0.0]*Angstrom
vector_b = [0.0, 2.866, 0.0]*Angstrom
vector_c = [0.0, 0.0, 2.866]*Angstrom
left_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
left_electrode_elements = [Iron, Iron]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
left_electrode_coordinates = [[ 0.7165,  0.7165,  0.7165],
                              [ 2.1495,  2.1495,  2.1495]]*Angstrom

# Set up configuration
left_electrode = BulkConfiguration(
    bravais_lattice=left_electrode_lattice,
    elements=left_electrode_elements,
    cartesian_coordinates=left_electrode_coordinates
    )

# -----------------------------------------------------------
# Right Electrode
# -----------------------------------------------------------

# Set up lattice
vector_a = [2.866, 0.0, 0.0]*Angstrom
vector_b = [0.0, 2.866, 0.0]*Angstrom
vector_c = [0.0, 0.0, 2.866]*Angstrom
right_electrode_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
right_electrode_elements = [Iron, Iron]

# Define coordinates
right_electrode_coordinates = [[ 2.1495,  2.1495,  0.7165],
                               [ 0.7165,  0.7165,  2.1495]]*Angstrom

# Set up configuration
right_electrode = BulkConfiguration(
    bravais_lattice=right_electrode_lattice,
    elements=right_electrode_elements,
    cartesian_coordinates=right_electrode_coordinates
    )

# -----------------------------------------------------------
# Central Region
# -----------------------------------------------------------

# Set up lattice
vector_a = [2.866, 0.0, 0.0]*Angstrom
vector_b = [0.0, 2.866, 0.0]*Angstrom
vector_c = [0.0, 0.0, 39.06710000000001]*Angstrom
central_region_lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
central_region_elements = [Iron, Iron, Iron, Iron, Iron, Iron, Iron, Iron, Magnesium,␣
↪Oxygen,
                           Oxygen, Magnesium, Magnesium, Oxygen, Oxygen, Magnesium,␣
↪Magnesium,
                           Oxygen, Oxygen, Magnesium, Magnesium, Oxygen, Iron, Iron,␣
```

(continues on next page)

(continued from previous page)

```
↪Iron,
                            Iron, Iron, Iron, Iron, Iron]

# Define coordinates
central_region_coordinates = [[  0.7165 ,   0.7165 ,   0.7165 ],
                              [  2.1495 ,   2.1495 ,   2.1495 ],
                              [  0.7165 ,   0.7165 ,   3.5825 ],
                              [  2.1495 ,   2.1495 ,   5.0155 ],
                              [  0.7165 ,   0.7165 ,   6.4485 ],
                              [  2.1495 ,   2.1495 ,   7.8815 ],
                              [  0.7165 ,   0.7165 ,   9.3145 ],
                              [  2.1495 ,   2.1495 ,  10.7475 ],
                              [  0.7165 ,   0.7165 ,  12.9475 ],
                              [  2.1495 ,   2.1495 ,  12.9475 ],
                              [  0.7165 ,   0.7165 ,  15.14285],
                              [  2.1495 ,   2.1495 ,  15.14285],
                              [  0.7165 ,   0.7165 ,  17.3382 ],
                              [  2.1495 ,   2.1495 ,  17.3382 ],
                              [  0.7165 ,   0.7165 ,  19.53355],
                              [  2.1495 ,   2.1495 ,  19.53355],
                              [  0.7165 ,   0.7165 ,  21.7289 ],
                              [  2.1495 ,   2.1495 ,  21.7289 ],
                              [  0.7165 ,   0.7165 ,  23.92425],
                              [  2.1495 ,   2.1495 ,  23.92425],
                              [  0.7165 ,   0.7165 ,  26.1196 ],
                              [  2.1495 ,   2.1495 ,  26.1196 ],
                              [  2.1495 ,   2.1495 ,  28.3196 ],
                              [  0.7165 ,   0.7165 ,  29.7526 ],
                              [  2.1495 ,   2.1495 ,  31.1856 ],
                              [  0.7165 ,   0.7165 ,  32.6186 ],
                              [  2.1495 ,   2.1495 ,  34.0516 ],
                              [  0.7165 ,   0.7165 ,  35.4846 ],
                              [  2.1495 ,   2.1495 ,  36.9176 ],
                              [  0.7165 ,   0.7165 ,  38.3506 ]]*Angstrom

# Set up configuration
central_region = BulkConfiguration(
    bravais_lattice=central_region_lattice,
    elements=central_region_elements,
    cartesian_coordinates=central_region_coordinates
    )

device_configuration = DeviceConfiguration(
    central_region,
    [left_electrode, right_electrode],
    equivalent_electrode_lengths=[5.732, 5.732]*Angstrom,
    transverse_electrode_repetitions=[[1, 1], [1, 1]],
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
#------------------------------------
# Basis Set
#------------------------------------
basis_set = [
    LDABasis.Oxygen_SingleZeta,
    LDABasis.Iron_SingleZetaPolarized,
    LDABasis.Magnesium_SingleZetaPolarized,
    ]

#------------------------------------
# Exchange-Correlation
#------------------------------------
exchange_correlation = LSDA.PZ

#------------------------------------
# Numerical Accuracy Settings
#------------------------------------
device_k_point_sampling = MonkhorstPackGrid(
    na=21,
    nb=21,
    nc=300,)
device_numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=device_k_point_sampling,
    )

#------------------------------------
# Contour Integral Settings
#------------------------------------
equilibrium_contour = SemiCircleContour(
    integral_lower_bound=2.55775293896*Hartree,
    )

contour_parameters = ContourParameters(
    equilibrium_contour=equilibrium_contour,
    )

device_algorithm_parameters = DeviceAlgorithmParameters(
    initial_density_type=EquivalentBulk(electrode_constraint_length=3.*Ang)
    )

#------------------------------------
# Device Calculator
#------------------------------------
calculator = DeviceLCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=device_numerical_accuracy_parameters,
    contour_parameters=contour_parameters,
    device_algorithm_parameters=device_algorithm_parameters,
    )

device_configuration.setCalculator(calculator)
```

(continues on next page)

(continued from previous page)

```
# -----------------------------------------------------------
# Initial State
# -----------------------------------------------------------
scaled_spins = [1] * 8 + [0] * 14 + [1] * 8
initial_spin = InitialSpin(scaled_spins=scaled_spins)

device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
)
device_configuration.update()
filename = 'polarized-0.hdf5'
nlsave(filename, device_configuration)
```

0-polarized.py

In order to calculate the STT, we need to rotate the spin in the free layer to obtain the desired magnetization direction. We apply a spin rotation to the previously calculated solution for the parallel configuration. Since we are only changing the magnetization direction, we can perform a non-selfconsistent update.

Note: For numerical reasons, the torque components are best determined in the layer with spin orientation along z, therefore it is suggested to rotate the spin in the fixed layer as in this example.

With this configuration we calculate the finite bias STT in a bias interval from -1 to 1 Volt, and for sake of comparison we also calculate the atom-resolved torkance. These steps are performed in the script below.

```
# Apply a spin rotation on top of the Polarized collinear solution and
# update non-selfconsistently.
device_configuration = nlread('polarized-0.hdf5', DeviceConfiguration)[-1]
calculator = device_configuration.calculator()
exchange_correlation = NCLDA.PZ
calculator = calculator(
    exchange_correlation=exchange_correlation,
    iteration_control_parameters=NonSelfconsistent
    )

# -----------------------------------------------------------
# Initial State
# -----------------------------------------------------------
# Define the spin rotation in polar coordinates
theta = 90
left_spins  = [(i, 1, theta * Degrees, 0 * Degrees) for i in range(8)]
center_spins  = [(i, 0, 0 * Degrees, 0 * Degrees) for i in range(8, 22)]
right_spins = [(i, 1, 0 * Degrees, 0 * Degrees) for i in range(22, 30)]
spin_list   = left_spins + center_spins + right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)
device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
```

(continues on next page)

(continued from previous page)

```
    initial_state=device_configuration
)

# Update and save.
device_configuration.update()
filename = 'theta-{}.hdf5'.format(theta)
nlsave(filename, device_configuration)

# Calculate and save the atom-resolved torkance.
torkance, torkance_per_kpoint, kpoints, weights =␣
↪calculateLinearResponseSpinTransferTorkance(
    device_configuration)
nlsave('stt-90.hdf5', torkance, object_id='torkance')

# Calculate and save the finite bias torque in a bias range of interest.
biases = numpy.linspace(-1.0, 1.0, 21) * Volt
torque, torque_per_kpoint, kpoints, weights =␣
↪calculateNonSelfConsistentFiniteBiasSpinTransferTorque(
    configuration=device_configuration,
    biases=biases,
    potential_drop_region=(7, 22))
nlsave('stt-90.hdf5', torque, object_id='torque')
nlsave('stt-90.hdf5', biases, object_id='biases')
```

finite_bias_stt.py

With the script below we can visualize the total torque integrated in the free layer and compare the atom-resolved torque with the linear response torkance, evaluated using *calculateLinearResponseSpinTransferTorkance*.

```
from matplotlib import pyplot as plt

# Unit conversion from eV to uJ/m^2
surface = 2.866 * 2.866 * 1e-20 # in m * m
conversion = (1 * eV).inUnitsOf(Joule) * 1e6 / surface

filename = 'stt-90.hdf5'
initial_free_layer_site = 22

# Retrieve the saved quantities
torkance_linear_response = nlread(filename, object_id='torkance')[0]
torque = nlread(filename, object_id='torque')[0]
biases = nlread(filename, object_id='biases')[0]

# Plot the torque obtained from the linear repsonse torkance.
total_torkance = numpy.sum(torkance_linear_response[:, initial_free_layer_site:], axis=1)
torque_from_torkance_x = [- total_torkance[0] * bias * conversion for bias in biases.
↪inUnitsOf(Volt)]
plt.plot(biases, torque_from_torkance_x, '--', label='In-plane torque (linear response)')

total_torque_x = [numpy.sum(x[0, initial_free_layer_site:]) * conversion for x in torque.
↪inUnitsOf(eV)]
plt.plot(biases, total_torque_x, 'o-', linewidth=2, label='In-plane torque (finite bias)
```

(continues on next page)

(continued from previous page)

```
↪')

total_torque_y = [numpy.sum(x[1, initial_free_layer_site:]) * conversion for x in torque.
↪inUnitsOf(eV)]
plt.plot(biases, total_torque_y, 'o-', linewidth=2, label='Out-of-plane torque (finite␣
↪bias)')

plt.xlabel('Voltage [V]')
plt.ylabel('Torque [uJ/m^2]')
plt.title(r'$\theta = \pi/2$')
plt.legend()
plt.grid()
plt.show()

# Plot the torque per atomic site.
single_bias_point = - 0.6
idx = 4

plt.plot(- single_bias_point * torkance_linear_response[0, :] * conversion, 'o--', label=
↪'In-plane torque (linear response)')
plt.plot(torque.inUnitsOf(eV)[idx][0, :] * conversion, '*-', label='In-plane torque␣
↪(finite bias)')

plt.title('Torque at {} V'.format(single_bias_point))
plt.ylabel('Torque [uJ/m^2]')
plt.xlabel('Atom no.')
plt.legend()
plt.grid()
plt.show()
```

plot.py

Results are shown in the figures below.

The non-linear behavior of the in-plane component for large bias and the quadratic behavior of the out-of-plane component are consistent with literature results[1],[2].

This example runs in about 240 core-hours on a modern HPC cluster, with quite loose numerical parameters. Wallclock time can be dramatically reduced by massive parallelization, since this kind of calculation scales well.

Note: Results are very sensitive to numerical parameters such as k-point sampling or real axis contour accuracy. Convergence with respect to those parameters should be carefully checked.

Note that the non-equilibrium contour parameters used to evaluate the density matrix (see *ContourParameters*) are taken from the calculator attached to the input configuration.

[1] Xingtao Jia, KeăXia, Youqi Ke, and Hong Guo. Nonlinear bias dependence of spin-transfer torque from atomic first principles. *Phys. Rev. B*, 84:014401, Jul 2011. URL: https://link.aps.org/doi/10.1103/PhysRevB.84.014401, doi:10.1103/PhysRevB.84.014401.

[2] Ivan Rungger, Andrea Droghetti, and Maria Stamenova. *Non-equilibrium Greens Function Methods for Spin Transport and Dynamics*, pages 1–27. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-42913-7_75-1, doi:10.1007/978-3-319-42913-7_75-1.

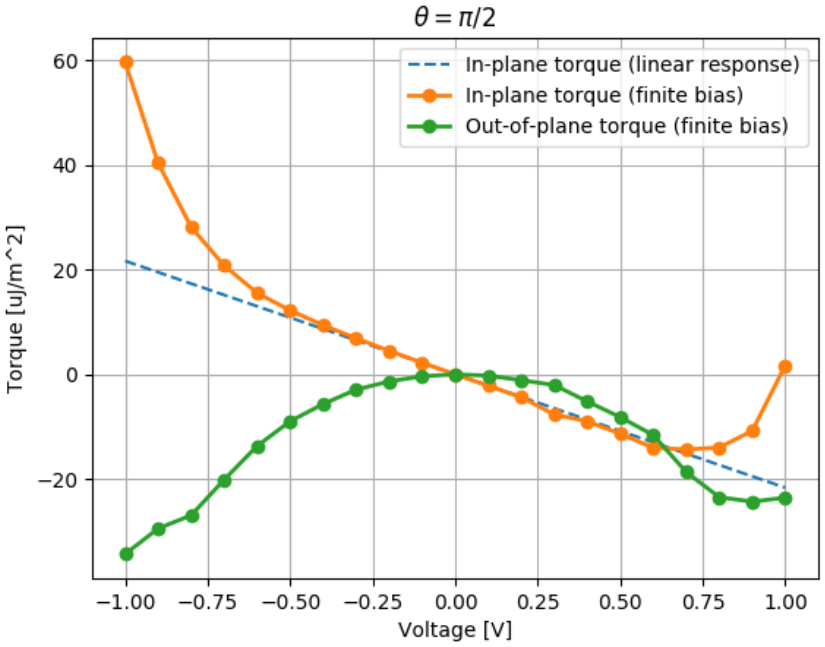


Fig. 4.41: Total Spin Transfer Torque integrated in the right metallic layer.

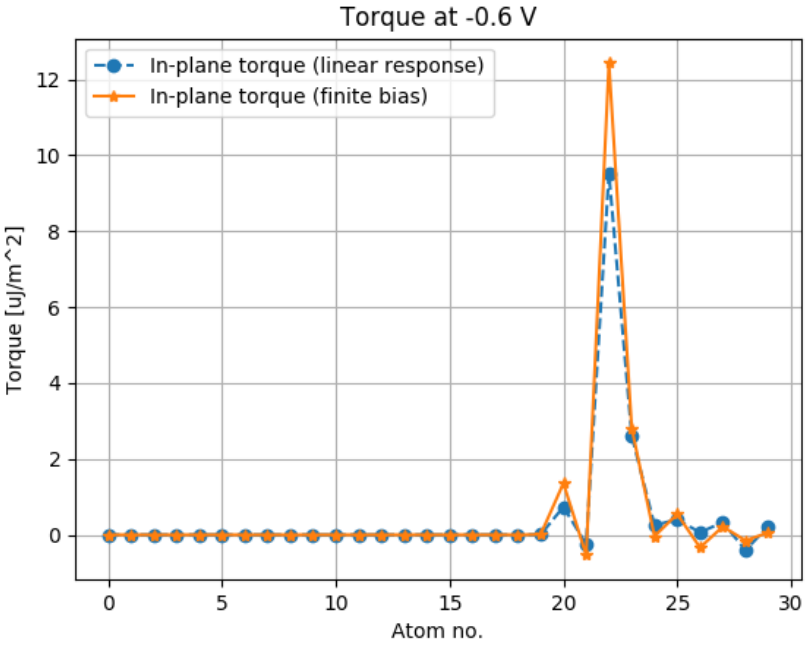


Fig. 4.42: Atom-resolved Spin Transfer Torque at -0.6 Volt evaluated with the finite bias and with the linear response methods.

Notes

The spin transfer torque is evaluated in a non-selfconsistent finite bias approximation. The potential drop across the barrier is represented as a ramp across the tunneling barrier[Page 4204, 2], avoiding time consuming and often hard to converge non-collinear calculations at finite bias. To best approximate the finite-bias potential, the potential ramp must be defined in between the metal atoms at the interface (included).

The atom-resolved torque on an atom a is then calculated as[3] :

$$\mathbf{T}_{CD,a} = \sum_{\alpha \in a} \sum_{\beta} \Re[\rho_{CD,\alpha\beta} \sigma \times \mathbf{B}_{XC,\alpha\beta}]$$

where $\rho_{CD} = \rho_{neq} - \rho_{eq}$ is the current driven density matrix, given by the difference between the density matrix at finite bias and at equilibrium.

calculatePhononGreenFunctionComponent

calculatePhononGreenFunctionComponent(*device_configuration*, *dynamical_matrix=None*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the Phonon Greens Function Component, $G\Gamma G^{\dagger}$.

Parameters

- **device_configuration** (*DeviceConfiguration*) – The configuration to use for the calculation.
- **dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use in the calculation.
- **energy** (*PhysicalQuantity* of type energy.) – Absolute energy for the calculation **Default:** `0.0 * eV`
- **kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`
- **contribution** (`Left` | `Right`) – Which electrode to consider for the evaluation. **Default:** `Left`
- **eta** (*PhysicalQuantity* of type energy.) – Complex energy for electrode transform matrices, imaginary part. **Default:** `1.0e-6 * Hartree`
- **self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
Phonon Greens Function Component as an array

Return type
PhysicalQuantity of type energy squared

[3] BranislavăK. Nikolic, Kapildeb Dolui, MarkoăD. Petrovic, Petr Plechac, Troels Markussen, and Kurt Stokbro. *First-Principles Quantum Transport Modeling of Spin-Transfer and Spin-Orbit Torques in Magnetic Multilayers*, pages 1–35. Springer International Publishing, Cham, 2018. URL: https://doi.org/10.1007/978-3-319-50257-1_112-1, doi:10.1007/978-3-319-50257-1_112-1.

Usage Examples

Evaluate on a *DeviceConfiguration*:

```
phonon_green_function_component = calculatePhononGreenFunctionComponent(device_
↪configuration)
```

Notes

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

calculatePhononRetardedGreenFunction

calculatePhononRetardedGreenFunction(*device_configuration*, *dynamical_matrix=None*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the Phonon Retarded Greens Function.

Parameters

•**device_configuration** (*DeviceConfiguration*) – The configuration to use for the calculation.

•**dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use in the calculation.

•**energy** (*PhysicalQuantity* of type energy) – Absolute energy for the calculation **Default:** `0.0 * eV`

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**eta** (*PhysicalQuantity* of type energy) – Complex energy for electrode transform matrices, imaginary part. **Default:** `1.0e-6 * Hartree`

•**self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
The Phonons Greens Function as an array

Return type
PhysicalQuantity of type energy squared

Usage Examples

Evaluate on a *DeviceConfiguration*:

```
phonon_green_function = calculatePhononRetardedGreenFunction(device_configuration)
```

Notes

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

calculatePhononSelfEnergy

calculatePhononSelfEnergy(*device_configuration*, *dynamical_matrix=None*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the Phonon Self Energy.

Parameters

•**device_configuration** (*DeviceConfiguration*) – The configuration to use for the calculation.

•**dynamical_matrix** (*DynamicalMatrix*) – The dynamical matrix to use in the calculation.

•**energy** (*PhysicalQuantity* of type energy) – Absolute energy for the calculation **Default:** `0.0 * eV`

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**contribution** (`Left` | `Right`) – Which electrode to consider for the evaluation. **Default:** `Left`

•**eta** (*PhysicalQuantity* of type energy) – Complex energy for electrode transform matrices, imaginary part. **Default:** `1.0e-6 * Hartree`

•**self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
The Phonon Self Energy as an array

Return type
PhysicalQuantity of type energy squared

Usage Examples

Evaluate on a *DeviceConfiguration*:

```
phonon_self_energy = calculatePhononSelfEnergy(device_configuration)
```

Notes

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

calculateRetardedGreenFunction

calculateRetardedGreenFunction(*device_configuration*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *spin=None*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the Retarded Greens Function.

Parameters

- **device_configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.
- **energy** (*PhysicalQuantity* of type energy) – Absolute energy for the calculation **Default:** `0.0 * eV`
- **kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component for which to perform the calculation. **Default:** `Spin.All`
- **eta** (*PhysicalQuantity* of type energy) – Complex energy for electrode transform matrices, imaginary part. **Default:** `1.0e-6 * Hartree`
- **self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
The Greens Function as an array

Return type
PhysicalQuantity in units of inverse energy

Usage Examples

Calculate the retarded Greens function for a *DeviceConfiguration*:

```
energy=1.0*eV
green_function = calculateRetardedGreenFunction(device_configuration, energy)
```

Equivalent calculation using basic variables, H, S

```
energy=1.0*eV
H,S = calculateHamiltonianAndOverlap(device_configuration)
S_L = calculateSelfEnergy(device_configuration, energy, contribution=Left)
S_L = S_L.inUnitsOf(Hartree)
S_R = calculateSelfEnergy(device_configuration, energy, contribution=Right)
S_R = S_R.inUnitsOf(Hartree)

# Setup E*S-H-S_L-S_R
lhs = energy.inUnitsOf(Hartree)*S-H.inUnitsOf(Hartree)
n = S_L.shape[0]
lhs[:n,:n] -= S_L
n = S_R.shape[0]
lhs[-n:,-n:] -= S_R
```

(continues on next page)

(continued from previous page)

```
# Do the inverse
green_function = numpy.linalg.inv(lhs)*Hartree
```

Notes

•Calculates the Fourier transformed retarded Greens function of a device. The retarded Greens function is given by

$$G^r(E, \mathbf{k}) = [(ES(\mathbf{k}) - H(\mathbf{k}) - \Sigma^L(E + i\eta, \mathbf{k}) - \Sigma^R(E + i\eta, \mathbf{k})]^{-1},$$

where S is the overlap matrix, H Hamiltonian, and Σ^L, Σ^R are the left and right self energies.

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

•Consider *Spin* for details on how to handle the spin parameter.

calculateSelfEnergy

calculateSelfEnergy(*device_configuration*, *energy=PhysicalQuantity(0.0, eV)*, *kpoint=None*, *spin=None*, *contribution=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.Left'>*, *eta=PhysicalQuantity(1e-06, Hartree)*, *self_energy_calculator=None*)

Calculate the self energy as a numpy array.

Parameters

•**device_configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.

•**energy** (*PhysicalQuantity* of type energy) – Absolute energy for self energy calculation. **Default:** `0.0 * eV`

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component for which to perform the calculation. **Default:** `Spin.All`

•**contribution** (`Left` | `Right`) – Which electrode (side) should be considered. Only the `Left` contribution is allowed for a *SurfaceConfiguration*. **Default:** `Left`

•**eta** (*PhysicalQuantity* of type energy) – Complex energy for electrode transform matrices, imaginary part. **Default:** `1.0e-6 * Hartree`

•**self_energy_calculator** (*RecursionSelfEnergy* | *DirectSelfEnergy* | *KrylovSelfEnergy* | *SparseRecursionSelfEnergy*) – The self energy evaluation strategy to use. **Default:** A default *RecursionSelfEnergy* instance.

Returns
The self energy as an array.

Return type
PhysicalQuantity of type energy

Usage Examples

Evaluate the self energy of a *DeviceConfiguration*:

```
self_energy = calculateSelfEnergy(device_configuration)
```

Notes

•Evaluate the Fourier transformed self energy of an electrode in a *DeviceConfiguration*.

•The calculator assigned to the device must be density matrix-based for this function to perform successfully.

•Consider *Spin* for details on how to handle the spin parameter.

calculateTB09C

calculateTB09C(*configuration*)

Calculate the c-parameter used in a calculation with a TB09 meta-GGA functional.

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration*) – The configuration with attached calculator using the TB09 meta-GGA functional for which the c-parameter should be calculated.

Returns
The c-parameter.

Return type
float

calculateVelocity

calculateVelocity(*configuration*, *kpoint=None*, *spin=None*, *band_indices=None*)

Calculates the Bloch-state velocity $\frac{1}{\hbar}\frac{dE}{dk}$ using first order perturbation theory.

Parameters

•**configuration** (*BulkConfiguration*) – The configuration for which to calculate the velocity.

•**kpoint** (*tuple of floats*) – The kpoint as three floats representing fractional reciprocal space coordinates. **Default:** The Gamma point `(0.0, 0.0, 0.0)`

•**spin** (`Spin.Up` | `Spin.Down` | `Spin.All`) – The spin component for which to perform the calculation. **Default:** `Spin.All`

•**band_indices** (*list of non-negative int*) – Indices of the bands for which to calculate the velocity. **Default:** All bands (`range(number_of_bands)`)

Returns
The velocities for each band.

Return type
`numpy.array`

Usage Examples

Evaluate the valence band velocity for graphene at the Dirac point:

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=Hexagonal(2.4612*Angstrom, 6.709*Angstrom),
    elements=[Carbon, Carbon],
    fractional_coordinates=[[ 0.333333333333,  0.166666666667,  0.5 ],
                            [ 0.666666666667,  0.833333333333,  0.5 ]],
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
calculator = LCAOCalculator()

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# Fractional k-point, slighly displaced away from the Dirac point.
k = [1./3+0.001, 1./3+0.001, 0  ]

# Calculate the velocity of the band with band-index 3 (valence band)
velocity = calculateVelocity(bulk_configuration, kpoint=k, spin=Spin.Up, band_
↪indices=[3])
# Take the velocity component along the X cartesian direction.
velocity = velocity[0][0][0]

# Print the result.
print('Fermi velocity of graphene:')
print('v = %.2e m/s' %abs(velocity.inUnitsOf(Meter/Second)))
```

`graphene_velocity.py`

Running the script you will get a Fermi velocity of $v_F = 8.4 \cdot 10^5$ m/s, which is close to literature values, genereally reported to be $\approx 10^6$ m/s.

Notes

- The band velocity, usually defined as $v_n(\mathbf{k}) = \nabla_{\mathbf{k}} E_n(\mathbf{k})$ is evaluated using first-order perturbation theory. The velocity in Cartesian direction α of band n at wavevector $\mathbf{k}$ is:

$$v_n^{\alpha}(\mathbf{k}) = \langle \psi_n(\mathbf{k}) | \partial_{k_\alpha} H(\mathbf{k}) - \epsilon_n(\mathbf{k}) \partial_{k_\alpha} S(\mathbf{k}) | \psi_n(\mathbf{k}) \rangle,$$

where $|\psi_n(\mathbf{k}\rangle$ is the Bloch eigenstate in band n with energy $\epsilon_n(\mathbf{k})$. $\partial_{k_\alpha} H(\mathbf{k})$ and $\partial_{k_\alpha} S(\mathbf{k})$ are the derivative of the Fourier transformed Hamiltonian and overlap matrices with respect to Cartesian component k_α.

center

center(*configuration*, *indices=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.All'>*, *axes=None*)
Return a copy of configuration where atoms at some indices have been centered along some axes.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration to work on.
- **indices** (*None | list of ints*) – List of indices to center. Supplying no indices means that all indices are centered.
- **axes** (*tuple of bools*) – The axes to center along. For molecules this refer to the (x, y, z) directions. For periodic structures it refer to the three lattice directions (A, B, C).

Returns
A configuration with atoms on indices centered along axes directions.

Return type
AtomicConfiguration based.

changeDeviceLength

changeDeviceLength(*device_configuration*, *length*, *extreme*)
Method to alter the device length. Handles devices with transverse electrode repetitions.

Parameters

- **device_configuration** (*DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to shrink.
- **length** (*PhysicalQuantity* of type length) – The length by which the central region should be altered.
- **extreme** (*Left | Right*) – Which side of the configuration to alter.

Returns
The altered device configuration.

Return type
DeviceConfiguration | *SurfaceConfiguration*

Usage Examples

Change the device length to the left by 2.0391 Ang.

```
# --- Create input configuration ---
# Set up lattice
vector_a = [4.07825, 0.0, 0.0] * Angstrom
vector_b = [0.0, 4.07825, 0.0] * Angstrom
vector_c = [0.0, 0.0, 32.626] * Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold]

# Define coordinates
```

(continues on next page)

(continued from previous page)

```
fractional_coordinates = [[ 0.    ,  0.    ,  0.    ],
                          [ 0.    ,  0.    ,  0.125 ],
                          [ 0.    ,  0.    ,  0.25  ],
                          [ 0.    ,  0.    ,  0.375 ],
                          [ 0.    ,  0.    ,  0.5   ],
                          [ 0.    ,  0.    ,  0.625 ],
                          [ 0.    ,  0.    ,  0.75  ],
                          [ 0.    ,  0.    ,  0.875 ],
                          [ 0.5   ,  0.5   ,  0.    ],
                          [ 0.5   ,  0.5   ,  0.125 ],
                          [ 0.5   ,  0.5   ,  0.25  ],
                          [ 0.5   ,  0.5   ,  0.375 ],
                          [ 0.5   ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.5   ,  0.625 ],
                          [ 0.5   ,  0.5   ,  0.75  ],
                          [ 0.5   ,  0.5   ,  0.875 ],
                          [ 0.5   ,  0.    ,  0.0625],
                          [ 0.5   ,  0.    ,  0.1875],
                          [ 0.5   ,  0.    ,  0.3125],
                          [ 0.5   ,  0.    ,  0.4375],
                          [ 0.5   ,  0.    ,  0.5625],
                          [ 0.5   ,  0.    ,  0.6875],
                          [ 0.5   ,  0.    ,  0.8125],
                          [ 0.5   ,  0.    ,  0.9375],
                          [ 0.    ,  0.5   ,  0.0625],
                          [ 0.    ,  0.5   ,  0.1875],
                          [ 0.    ,  0.5   ,  0.3125],
                          [ 0.    ,  0.5   ,  0.4375],
                          [ 0.    ,  0.5   ,  0.5625],
                          [ 0.    ,  0.5   ,  0.6875],
                          [ 0.    ,  0.5   ,  0.8125],
                          [ 0.    ,  0.5   ,  0.9375]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Create a basic device configuration.
configuration = deviceFromBulk(
    bulk_configuration=bulk_configuration,
    electrode_extension_lengths=Automatic,
    adjust_cell=True,
    use_minimal_electrodes=All
)

# --- Change the device length ---
# Apply the desired extension/retraction to the left side of the central region.
configuration = changeDeviceLength(
    device_configuration=configuration,
```

(continues on next page)

(continued from previous page)

```
    length=2.0391 * Angstrom,
    extreme=Left
)
```

changeDeviceLength_example.py

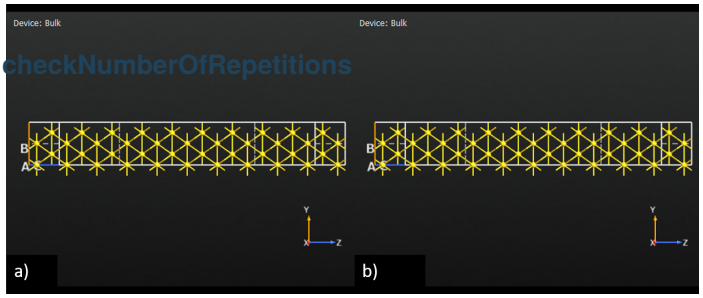


Fig. 4.43: a) The input configuration used in the above example. b) The DeviceConfiguration generated from the example.

checkNumberOfRepetitions(*configuration*, *max_interaction_range=None*)

Method for automatic detection of the number of repetitions.

@param configuration : The configuration for which the repetitions should be calculated. @type : BulkConfiguration | MoleculeConfiguration | DeviceConfiguration.

@param max_interaction_range : The maximum range of the interactions. @type : PhysicalQuantity of type length. @default : None (pre-defined interaction range)

@return Tuple with the number of repetitions.

Usage Examples

Calculate and print the automatically detected repetition for a bulk configuration.

```
# ------------------------------------------------------------
# Bulk configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )
```

(continues on next page)

(continued from previous page)

```
# Calculate the automatically detected repetitions.
repetitions = checkNumberOfRepetitions(bulk_configuration)

# Print the result.
print(repetitions)


# Calculate the automatically detected repetitions, when a maximum interaction
# range is specified.
repetitions = checkNumberOfRepetitions(bulk_configuration,
                                       max_interaction_range=10*Angstrom)

# Print the result.
print(repetitions)
```

Note that the repetition can also be calculated, when a maximum interaction range is specified as shown in the last two lines.

convertMACEModelToQATKFormat

convertMACEModelToQATKFormat(*model_path*, *model_head_number=-1*)

Convert a MACE model to a QATK compatible format.

Parameters

- **model_path** (*str | Path*) – The full path to a MACE model file.
- **model_head_number** (*int*) – The head number of the model to use for the QATK compatible model. **Default:** The last head of the input model.

Usage Examples

MACE models set up via *MACEFittingParameters* and trained via *MachineLearnedForceFieldTrainer* are automatically converted into QATK-compatible formats. For MACE models acquired from elsewhere, it is possible to convert them into a TremoloX-native format by using this method. The conversion is required in order to use MACE models with QuantumATKs native force field framework.

Assume that we have a MACE model file named `mace_model.model` with only a single model head in our current working directory. Using the following script, the mace MACE model can be converted into a QATK-compatible format with a `.qatkpt` extension.

```
# Convert the MACE model to QATK format
convertMACEModelToQATKFormat('mace_model.model')
```

After the conversion has saved the model in a QuantumATK compatible format, the MACE model can be loaded as a force field potential and used to set up a *TremoloXCalculator* that can be used as usual in QuantumATK.

```
# Load the interatomic potential into a TremoloXCalculator that can be
# attached to and used with QATK configurations...
model_path = 'mace_model.qatkpt'

potentialSet = TremoloXPotentialSet(name='my_mace_potential')
potential = TorchXPotential(
    dtype='float64',
    device='cuda',
    file=model_path,
)
for symbol in potential.get_symbols(model_path):
    potentialSet.addParticleType(ParticleType(symbol))
potentialSet.addPotential(potential)

calculator = TremoloXCalculator(parameters=potentialSet)
```

This example script is available for download: `conversion_example.py`

Notes

- The conversion script is available for `.model` files containing one or more model heads generated with the MACE framework[1][2].
- The output model file will have the same name as the input file path but replace the extension to a `.qatkpt` format.
- It is only possible to convert one head from a `.model` file into a `.qatkpt` file. If several heads from a `.model` file are required, the conversion needs to be repeated with different `model_head_number` values.

counterpoiseCorrected

counterpoiseCorrected(*calculator_cls*, *tag_list*)

Create a new class extended with a basis set superposition error (counterpoise) corrector.

Parameters

- **calculator_cls** (subclass of `Calculator`) – The class to use as base class for the new extended calculator.
- **tag_list** (*list of strings*) – The list of at least two unique tags to use to split the configuration.

[1] Ilyes Batatia, DavidăPeter Kovacs, Gregor N.ăC. Simm, Christoph Ortner, and Gabor Csanyi. MACE: higher order equivariant message passing neural networks for fast and accurate force fields. In AliceăH. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*. 2022. URL: https://openreview.net/forum?id=YPpSngE-ZU.

[2] Ilyes Batatia, Simon Batzner, DávidăPéter Kovács, Albert Musaelian, Gregor N.ăC. Simm, Ralf Drautz, Christoph Ortner, Boris Kozinsky, and Gábor Csányi. The design space of e(3)-equivariant atom-centered interatomic potentials. 2022. arXiv:2205.06643, doi:10.48550/arXiv.2205.06643.

Usage Examples

Define a counterpoise corrected LCAO calculator

```
CorrectedLCAOCalculator = counterpoiseCorrected(
    LCAOCalculator,
    tag_list=['atomgroup_tag_1', 'atomgroup_tag_2'])
 calculator = CorrectedLCAOCalculator()
```

Notes

The calculator must support ghost atoms in order for the counterpoise correction to be applicable.

createGridSampling

createGridSampling(*calculator_type*, *configuration*, *basis_set=None*, *poisson_solver=None*, *density_mesh_cutoff=None*, *exchange_correlation=None*)

Create a GridSampling object from the configuration and basis_set, optional the poisson_solver and density_mesh_cutoff.

Parameters

- **calculator_type** (*LCAOCalculator* | *PlaneWaveCalculator* | *SemiEmpiricalCalculator* | class:~.*DeviceLCAOCalculator* | *DeviceSemiEmpiricalCalculator*) – The type of calculator.
- **configuration** (AtomicConfiguration) – The configuration
- **basis_set** (*BasisSet*) – The basis set.
- **poisson_solver** (PoissonSolver) – The Poisson solver
- **density_mesh_cutoff** (*PhysicalQuantity* of type energy.) – The default mesh cut-off for all the elements in the configuration.

Returns
The grid sampling.

Return type
GridSampling

createLocalDFTUBasisSet

createLocalDFTUBasisSet(*configuration*, *basis_set*, *shell_indices*, *atom_u_values*)

Create a new configuration with tags for the atoms with U values and a new basis set with the U values for the tagged atoms.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration.
- **basis_set** (list of *BasisSet* objects) – The basis set.
- **shell_indices** (dict of *PeriodicTableElement* to list of int) – The map of elements to shell indices the U values refer to.

- **atom_u_values** (list of tuples of int and PhysicalQuantity) – The U values for the atoms: a list of tuples with the atom index and the U values for the selected shells for that atom.

Returns
The new configuration with tagged atoms and the new basis set list with basis sets for the tagged atoms.

Return type
BulkConfiguration and list of *BasisSet*

Usage Examples

The following example sets up a small iron slab, provides different U values for the different atom sites, and uses *createLocalDFTUBasisSet()* to create a configuration with tags and different basis sets for the different sites with the provided U values. It then calculates the Mulliken population for the system with the provided Us:

```
setVerbosity(MinimalLog)

# Set up a simple slab of iron.
iron_slab = BulkConfiguration(
    bravais_lattice=UnitCell(
        [2.8665, 0.0, 0.0] * Angstrom,
        [0.0, 2.8665, 0.0] * Angstrom,
        [0.0, 0.0, 30.0] * Angstrom
    ),
    elements=[Iron, Iron, Iron, Iron, Iron, Iron],
    fractional_coordinates=[
        [ 0.25     ,  0.25     ,  0.3805625],
        [ 0.25     ,  0.25     ,  0.4761125],
        [ 0.25     ,  0.25     ,  0.5716625],
        [ 0.75     ,  0.75     ,  0.4283375],
        [ 0.75     ,  0.75     ,  0.5238875],
        [ 0.75     ,  0.75     ,  0.6194375]
    ]
)

# For this example, we choose a simple Double Zeta Polarized basis set.
basis_set = [GGABasis.Iron_DoubleZetaPolarized]

# The indices of the basis set orbitals that the U values should be applied to.
# Here: For the (FHI) GGA Double Zeta Polarized basis set, the first orbital
# corresponds to the 3d orbital.
shell_indices = {Iron: [0]}

# Provide different 3d U values for the different atom sites.
u_list = [
    (0, [2.44869703] * eV),
    (1, [1.29706963] * eV),
    (2, [1.31089013] * eV),
    (3, [1.31088757] * eV),
    (4, [1.29707183] * eV),
    (5, [2.44867516] * eV),
]
```

(continues on next page)

(continued from previous page)

```
# Create a new configuration and basis set with the above U values:
iron_slab_with_us, basis_set_with_us = createLocalDFTUBasisSet(
    configuration=iron_slab,
    basis_set=basis_set,
    shell_indices=shell_indices,
    atom_u_values=u_list
)

# Set up an PBE functional with +U using the 'Dual' projection method.
xc_dftu = ExchangeCorrelation(
    exchange=PerdewBurkeErnzerhofExchange,
    correlation=PerdewBurkeErnzerhofCorrelation,
    hubbard_term=Dual,
    number_of_spins=2
)

# Set up the calculator with the U basis set.
calculator = LCAOCalculator(
    basis_set=basis_set_with_us,
    exchange_correlation=xc_dftu,
    numerical_accuracy_parameters=NumericalAccuracyParameters(
        k_point_sampling=KpointDensity(4.0 * Angstrom),
    ),
    checkpoint_handler=NoCheckpointHandler,
)

# Calculate Mulliken population
iron_slab_with_us.setCalculator(calculator)
iron_slab_with_us.update()

nlsave('Fe-slab-Hubbard-U.hdf5', iron_slab_with_us)

mulliken = MullikenPopulation(iron_slab_with_us)
nlsave('Fe-slab-Hubbard-U.hdf5', mulliken)

nlprint(mulliken)
```

crystalTrainingRandomDisplacements

crystalTrainingRandomDisplacements(*reference_configuration*, *supercell_repetitions_list=None*, *target_sample_size=200*, *base_atomic_rattling_amplitude=PhysicalQuantity(0.15, Ang)*, *max_atomic_rattling_amplitude=PhysicalQuantity(0.4, Ang)*, *max_cell_rattling_amplitude=0.05*, *random_seed=None*, *strain_included=True*, *sample_size_per_stage=None*, *volume_strain_sample_size=None*, *system_sizes=None*, *log_filename_prefix=None*, *data_tag=None*)

Preset protocol for a random displacement series for training crystalline materials. It contains stages of small displacements, small displacements with anisotropic strain, small displacements with volume strain, larger displacements, and larger displacements with anisotropic strain.

Parameters

- **reference_configuration** (*BulkConfiguration* | sequence of [*BulkConfiguration*] | *Table*) – One or more reference configurations to be used to generate the training set. For bulk configurations, this is the unit cell from which a supercell can be defined and additional strain can be applied.
- **supercell_repetitions_list** (*sequence (size 3) of int | sequence of sequence (size 3) of int*) – The list of supercells to construct for each configuration, given as the number of repetitions of the bulk unit cell along the (a, b, c) directions. Cannot be specified if there are molecules in the list of reference configurations. If None: No repetitions. **Default:** `None`
- **target_sample_size** (*int*) – The approximate number of training configurations to generate. The actual number of configurations can be slightly larger than the target. If `target_sample_size` is specified volume_strain_sample_size is determined and set automatically. **Default:** `200`
- **base_atomic_rattling_amplitude** (*PhysicalQuantity* of type length) – The minimum random displacement amplitude of the atomic positions used in the protocol. **Default:** `0.15 * Angstrom`
- **max_atomic_rattling_amplitude** (*PhysicalQuantity* of type length) – The maximum random displacement amplitude of the atomic positions used in the protocol. **Default:** `0.4 * Angstrom`
- **max_cell_rattling_amplitude** (*float*) – The maximum value of the strain tensor components in the stages that use anisotropic strains. Only applies to bulk configurations. **Default:** 0.05.
- **random_seed** (*int*) – The random seed used for generating the displacements. If None: Generated automatically. **Default:** `None`.
- **strain_included** (*bool*) – If strained configurations will be included. **Default:** True.
- **sample_size_per_stage** (*int*) – The number of training configurations to generate for each stage of random displacements. It is recommended to use target_sample_size instead. If `sample_size_per_stage` is specified, target_sample_size is disabled. **Default:** `None`
- **volume_strain_sample_size** (*int*) – The number of training configurations to generate for each of the 6 parts of volume strain stage of random displacements in the strained configurations. If not provided then it depends on `sample_size_per_stage`. The protocol is used in one stage which has six different volume strain. The total number of training configurations will be around `volume_strain_sample` size times six, multiplied with the number of configurations and supercell repetitions. **Default:** `None`
- **system_sizes** (*int | sequence of int*) – Target number of atoms in the system. If None then `supercell_repetitions_list` will be used to do the repetition of the cell. If set, `supercell_repetitions_list` will be ignored. **Default:** `None`
- **log_filename_prefix** (*str*) – Filename prefix for the logging output of the tasks associated with this set. **Default:** Defined by the *MomentTensorPotentialTraining* object.
- **data_tag** (*str*) – Label for this training set to enable selection of different data in MTP fitting. **Default:** `None`.

Returns
The *RandomDisplacementsParameters* objects for generating the training data and a single *TrainingSet* for the reference configurations. Hence, this method does not generate the configurations but returns the parameters/specifications for generating the training data configurations.

Return type
A list of *RandomDisplacementsParameters* objects and a last element of the list being a *TrainingSet*.

Usage Examples

Set up a random displacement protocol and generate training data for a Silicon crystal. This will generate 181 training configurations (including the undisplaced configuration), which provides a basic training dataset for a MTP potential that can simulate the main properties of the crystal.

```
# Automatically create a protocol of random displacements and strain for different␣
↪supercell sizes.
parameters = crystalTrainingRandomDisplacements(
    bulk_configuration,
    supercell_repetitions_list=[(2, 2, 2), (3, 3, 3)],
    sample_size_per_stage=10,
)

# Generate the displaced structures and calculate DFT training data.
mtp_training = MomentTensorPotentialTraining(
    filename='Silicon_crystal_mtp_training_testtest.hdf5',
    object_id='mtp_training',
    training_sets=parameters,
    calculator=calculator,
    calculate_stress=True,
)

mtp_training.update()
```

crystal_training.py

Notes

The *crystalTrainingRandomDisplacements* provides a pre-set list of *RandomDisplacementsParameters* to set up a basic training protocol for crystal structures. These parameters can be used in the *MomentTensorPotentialTraining* class to generate a training dataset for the machine-learned Moment Tensor Potential (MTP).

The protocol includes different stages of small atomic displacements with and without volume and anisotropic strain, followed by increasingly larger displacements magnitudes up to `max_atomic_rattling_amplitude`, with and without anisotropic strain. It is recommended to use various supercell sizes to improve the training of the total energy. When run in the *MomentTensorPotentialTraining*, this protocol typically provides a good basic training dataset that enables stable simulation of the crystal up to moderate temperatures, and a good description of crystal properties, such as lattice constants, phonons, and elastic constants.

densityToNumberOfKpoints

densityToNumberOfKpoints(*kpoint_density*, *configuration*, *parity=None*)

Convert a k-point density to numbers of k-points in the kA, kB, and kC directions.

Parameters

- **kpoint_density** (Non-negative *PhysicalQuantity* of type length | Non-negative *PhysicalQuantity* (3) of type length) – The density of k-points for the kA, kB, and kC directions specified collectively by one value or individually by three values, e.g., 12*Angstrom or [10.0, 12.0, 11.0]*Angstrom. A value of 0 ensures that a single grid point will always be used for the given direction (i.e., Γ).
- **configuration** (*BulkConfiguration*) – The configuration whose Brillouin zone should be resolved.

- **parity** (*None | Odd | Even | list(3) of Odd, Even, and None*) – Forces the number of k-points to be odd or even. It can be specified collectively by one parity flag or individually by a list of three parity flags for the kA, kB, and kC directions. **Default:** `None` (No favored parity)

Returns
A list with the numbers of k-points in the kA, kB, and kC directions.

Return type
array(3) of int

Usage Examples

Calculate the number of k-points in the A, B, and C direction corresponding to a k-point density

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=FaceCenteredCubic(5.4306*Angstrom),
    elements=[Silicon, Silicon],
    cartesian_coordinates=[[0.0    , 0.0    , 0.0     ],
                           [1.35765, 1.35765, 1.35765]]*Angstrom
    )

# Get the number of k-points for a given k-point density.
[n_kx, n_ky, n_kz] = densityToNumberOfKpoints(9.5*Ang, bulk_configuration)

# Yields [20.0, 20.0, 20.0]
```

Calculate the odd numbers of k-points corresponding to a k-point density:

```
[n_kx, n_ky, n_kz] = densityToNumberOfKpoints(9.5*Ang, bulk_configuration, Odd)

# Yields [ 21.0,  21.0,  21.0]
```

Calculate the numbers of k-points corresponding to an anisotropic k-point density:

```
[n_kx, n_ky, n_kz] = densityToNumberOfKpoints([7.5, 10.0, 10.0]*Ang, bulk_configuration)

# Yields [ 16.0,  21.0,  21.0]
```

Calculate the even numbers of k-points corresponding to an anisotropic k-point density:

```
[n_kx, n_ky, n_kz] = densityToNumberOfKpoints([7.5, 10.0, 10.0]*Ang, bulk_configuration,␣
↪Even)

# Yields [ 16.,  22.,  22.]
```

Calculate the odd numbers of k-points in the A direction, and even numbers of k-points in the B and C directions corresponding to an anisotropic k-point density:

```
[n_kx, n_ky, n_kz] = densityToNumberOfKpoints([7.5, 10.0, 10.0]*Ang, bulk_configuration,␣
↪[Odd, Even, Even])

# Yields [17.0, 22.0, 22.0]
```

Notes

The number of k-points are determined by ceiling to the nearest integer, or in the Even/Odd case rounding up to the nearest even/odd integer.

deviceFromBulk

deviceFromBulk(*bulk_configuration*, *electrode_extension_lengths*, *adjust_cell=True*, *use_minimal_electrodes=<class 'NL.ComputerScienceUtilities.NLFlag._NLFlag.All'>*)

Method that takes a bulk configuration and makes a device configuration.

Parameters

- **bulk_configuration** (*BulkConfiguration*) – The configuration to create the device from.
- **electrode_extension_lengths** (*PhysicalQuantity* of type length | Automatic) – The desired equivalent electrode extension length of each electrode. If only one length is given, a *SurfaceConfiguration* will be created, if two lengths or the NLFlag Automatic is given a *DeviceConfiguration* will be created. If use_minimal_electrodes is False or Transverse, these values also set the electrode lengths in the transport (C-)direction.
- **adjust_cell** (*bool*) – Whether or not to adjust the configuration in the C-direction such that that the electrodes are centered in their own unit cells. **Default:** True.
- **use_minimal_electrodes** (*All | Transport | Transverse | None*) – Try to use the minimal electrodes in: all directions (All), the electron transport (C-) direction (Transport), the transverse (A- and B-) directions (Transverse) or no directions (None). The electrode extension length (i.e. within the central region) will still be set according to the values of electrode_extension_lengths. **Default:** All.

Returns
The created device or surface configuration.

Return type
DeviceConfiguration | *SurfaceConfiguration*

Usage Examples

Create a device from a bulk bar of gold, using minimal electrodes in both the (charge) transport and the transverse directions. The device is extended on the left side by 2.0391 Ang, then retracted on the right side by 2.0391 Ang. The electrode lengths are also adjusted to be 12.23475 Ang and 16.313 Ang on the left and right sides, respectively.

```
# --- Create input configuration ---
# Set up lattice
vector_a = [4.07825, 0.0, 0.0] * Angstrom
vector_b = [0.0, 4.07825, 0.0] * Angstrom
vector_c = [0.0, 0.0, 32.626] * Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold]

# Define coordinates
fractional_coordinates = [[ 0.    ,  0.    ,  0.    ],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.    ,  0.    ,  0.125 ],
                          [ 0.    ,  0.    ,  0.25  ],
                          [ 0.    ,  0.    ,  0.375 ],
                          [ 0.    ,  0.    ,  0.5   ],
                          [ 0.    ,  0.    ,  0.625 ],
                          [ 0.    ,  0.    ,  0.75  ],
                          [ 0.    ,  0.    ,  0.875 ],
                          [ 0.5   ,  0.5   ,  0.    ],
                          [ 0.5   ,  0.5   ,  0.125 ],
                          [ 0.5   ,  0.5   ,  0.25  ],
                          [ 0.5   ,  0.5   ,  0.375 ],
                          [ 0.5   ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.5   ,  0.625 ],
                          [ 0.5   ,  0.5   ,  0.75  ],
                          [ 0.5   ,  0.5   ,  0.875 ],
                          [ 0.5   ,  0.    ,  0.0625],
                          [ 0.5   ,  0.    ,  0.1875],
                          [ 0.5   ,  0.    ,  0.3125],
                          [ 0.5   ,  0.    ,  0.4375],
                          [ 0.5   ,  0.    ,  0.5625],
                          [ 0.5   ,  0.    ,  0.6875],
                          [ 0.5   ,  0.    ,  0.8125],
                          [ 0.5   ,  0.    ,  0.9375],
                          [ 0.    ,  0.5   ,  0.0625],
                          [ 0.    ,  0.5   ,  0.1875],
                          [ 0.    ,  0.5   ,  0.3125],
                          [ 0.    ,  0.5   ,  0.4375],
                          [ 0.    ,  0.5   ,  0.5625],
                          [ 0.    ,  0.5   ,  0.6875],
                          [ 0.    ,  0.5   ,  0.8125],
                          [ 0.    ,  0.5   ,  0.9375]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# --- Create Device from bulk ---
# Create the device configuration.
configuration = deviceFromBulk(
    bulk_configuration=bulk_configuration,
    electrode_extension_lengths=[12.23475, 16.313] * Angstrom,
    adjust_cell=True,
    use_minimal_electrodes=All
)
# Apply the desired extension/retraction to the left side of the central region.
configuration = changeDeviceLength(
    device_configuration=configuration,
    length=2.0391 * Angstrom,
    extreme=Left
```

(continues on next page)

(continued from previous page)

```
)
# Apply the desired extension/retraction to the right side of the central region.
configuration = changeDeviceLength(
    device_configuration=configuration,
    length=-2.0391 * Angstrom,
    extreme=Right
)
```

deviceFromBulk_example.py

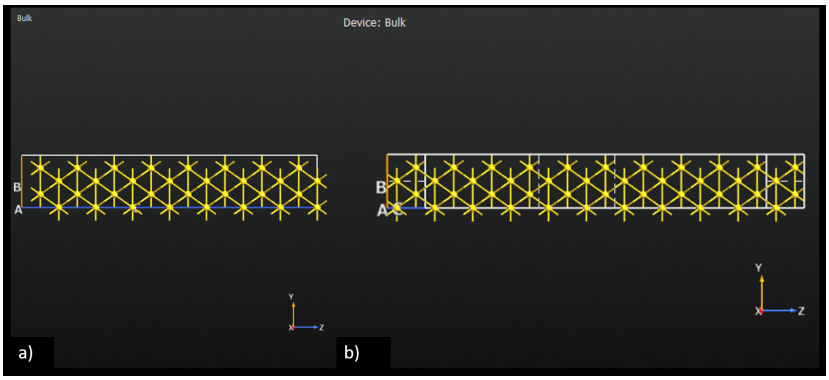


Fig. 4.44: a) The input configuration used in the above example. b) The DeviceConfiguration generated from the example.

To create a device with minimal electrodes in only the (charge) transport or transverse directions, use the flags Transport or Transverse, respectively. Example:

```
# Create a device configuration with minimal electrodes in the transport direction only.
configuration = deviceFromBulk(
    bulk_configuration=bulk_configuration,
    electrode_extension_lengths=[12.23475, 16.313] * Angstrom,
    adjust_cell=True,
    use_minimal_electrodes=Transport
)

# Create a device configuration with minimal electrodes in the transverse direction only.
configuration = deviceFromBulk(
    bulk_configuration=bulk_configuration,
    electrode_extension_lengths=[12.23475, 16.313] * Angstrom,
    adjust_cell=True,
    use_minimal_electrodes=Transverse
)
```

Notes

- When using Automatic to find suitable electrode lengths, the function will raise an NLValueError if no suitable electrode lengths could be found.
- The function will always raise an NLValueError if the sum of electrode lengths exceeds the length of the central region.
- The function automatically adjusts the tags of the original `BulkConfiguration` to only apply within the central region of the newly created `DeviceConfiguration` or `SurfaceConfiguraton`.
- This function can also be used to create a `SurfaceConfiguraton` if only one electrode length is given, but will not add the region of vacuum at the surface. To properly create a `SurfaceConfiguraton`, use the function *surfaceFromBulk()*.

dimerFromNEB

dimerFromNEB(*neb*, *dimer_separation=None*, *ts_index=None*)

Function to automatically set up a dimer configuration from a nudged elastic band. The four images (if available) around the interpolated transition state are used to generate the initial eigenmode.

Parameters

- **neb** (*NudgedElasticBand*) – The nudged elastic band configuration to generate the dimer from.
- **dimer_separation** (*PhysicalQuantity* of type length) – The distance separating the displaced dimer images. **Default:** *1e-4 Angstrom*
- **ts_index** (*int*) – The image index to use as the transition state guess. **Default:** Automatically determined if the energy is available.

Returns
The dimer configuration used for saddle point search.

Return type
DimerConfiguration

explicitHydrogenConfiguration

explicitHydrogenConfiguration(*configuration*)

Return a configuration with explicit hydrogens converted from united atoms.

Parameters
configuration (`MoleculeConfiguration` | `BulkConfiguration`) – Configuration with united atoms.

Returns
The new configuration with explicit hydrogens.

Return type
MoleculeConfiguration | *BulkConfiguration*

Notes

This method converts a united atom configuration to an all-atom or explicit hydrogen configuration. Hydrogen atoms are placed back onto the molecule by using hydrogen passivation on the united atom configuration.

The *explicitHydrogenConfiguration()* function takes a configuration with united atoms as input. All united atoms in the configuration are returned to their explicit hydrogen representation. The configuration can be returned to the united atom representation by using the *unitedAtomsConfiguration()* function.

fitCell

fitCell(*configuration*, *axes=None*)

Adjust the size of the cell to fit the atoms.

Parameters

- **configuration** (BulkConfiguration) – The configuration.
- **axes** (*3-tuple of bool*) – The axes, x, y, z.

Returns
The configuration with the adjusted cell.

Return type
BulkConfiguration

generateDefectsConfigurations

generateDefectsConfigurations(*defects*, *host_configuration*, *supercell_repetitions=(1, 1, 1)*, *use_ghost=None*)

Utility function to get the configurations associated with the defects. For use with the store framework.

Parameters

- **defects** (*Defects*) – The Defects object.
- **host_configuration** (*BulkConfiguration*) – The host configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the reference bulk unit cell along the (a, b, c) directions.
- **use_ghost** (*bool*) – Whether to use a ghost atom at the vacancy defect site (not referenced for other types of defects).

Returns
A table of defect configurations.

Return type
Table

Notes

The *generateDefectsConfigurations* is a utility method that converts a *Defects* object to a table of defect configurations. The table has one column containing the configurations. A host *BulkConfiguration* must be supplied from which the defects are generated. Optionally, supercell repetitions may be given. Ghost atoms can be added to vacancy defects by setting the `use_ghost` argument to `True`.

generateHalfHeuslerAlloy

generateHalfHeuslerAlloy(*elements*, *lattice_constant*, *atomic_ordering=1*)

Algorithm to generate a *BulkConfiguration* with the structure of a half ternary Heusler alloy with a known lattice constant. The elements should be given in the order X, Y and Z. The type of atomic ordering should also be specified.

Parameters

- **elements** (*PeriodicTableElement*) – A sequence containing the elements of the configuration, in the order [X, Y, Z].
- **lattice_constant** (*PhysicalQuantity* of type length) – The lattice constant of the chosen Heusler alloy/compound.
- **atomic_ordering** (*int*) – The type of atomic ordering for the half-Heusler, 1=(XYZ) 2=(ZXY) and 3=(YZX).(see HeuslerAlloyBuilderNotes). **Default:** 1

Returns
The primitive cell of the Heusler alloy.

Return type
BulkConfiguration

Usage Examples

Generate the half Heusler alloy NiFeSi:

```
elements = [Nickel, Iron, Silicon]
lattice_constant = 5.44*Angstrom

configuration = generateHalfHeuslerAlloy(elements, lattice_constant, atomic_ordering=1)
```

`HeuslerAlloy_Half_example.py`

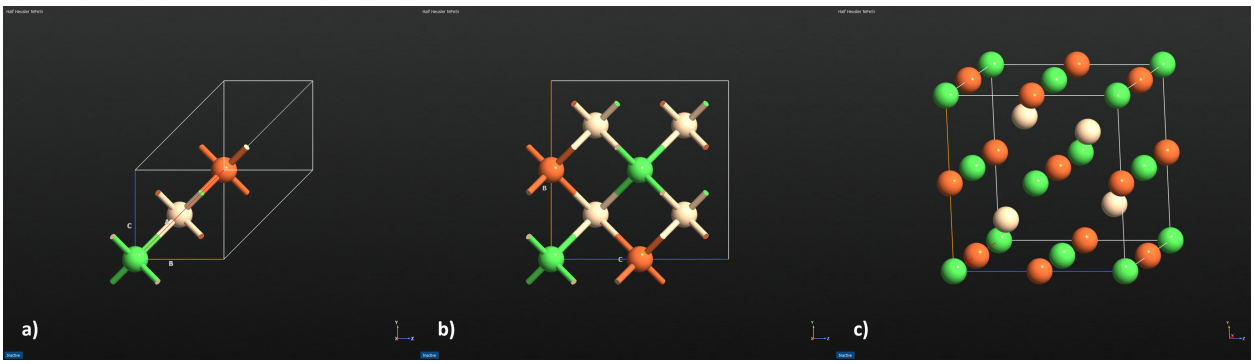


Fig. 4.45: Example of a half Heusler configuration. X: Nickel (green), Y: Iron (orange), Z: Silicon (cream). a) primitive cell, b) conventional cell, c) conventional cell (with equivalent atoms)

See also ref.heusler-al-loy-common.notes.

generateHeuslerAlloy

generateHeus

Algorithm to generate a *BulkConfiguration* with the structure of a full ternary Heusler alloy with a known lattice constant. The elements should be given in the order X, Y and Z. Optionally the inverse Heusler configuration can also be generated by setting the keyword argument `inverse=True`.

Parameters

- **elements** (*PeriodicTableElement*) – A sequence containing the elements of the configuration, in the order [X, Y, Z].
- **lattice_constant** (*PhysicalQuantity* of type length) – The lattice constant of the chosen Heusler alloy/compound.
- **inverse** (*bool*) – Create the inverse Heusler configuration **Default:** False

Returns
The primitive cell of the Heusler alloy.

Return type
BulkConfiguration

Usage Examples

Generate the full Heusler alloy Fe_2MnAl:

```
elements = [Iron, Manganese, Aluminium]
lattice_constant = 5.6711*Angstrom

configuration = generateHeuslerAlloy(elements, lattice_constant)
```

HeuslerAlloy_Full_example.py

Generate the inverse Heusler alloy MnV_2Al:

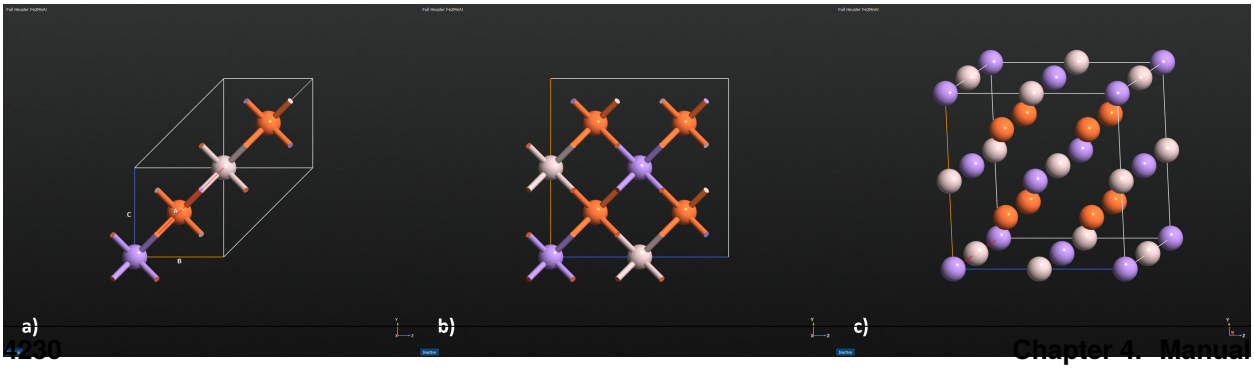


Fig. 4.46: Example of a full Heusler configuration. X: Iron (orange), Y: Manganese (purple), Z: Aluminium (pink). a) primitive cell, b) conventional cell, c) conventional cell (with equivalent atoms)

```
elements␣
↪=␣
↪[Manganese,
↪␣
↪Vanadium,
↪␣
↪Aluminium]
lattice_
↪constant␣
↪=␣
↪5.
↪9218*Angstr

configuration
↪=␣
↪generateHeu
↪␣
↪lattice_
↪constant,
↪␣
↪inverse=Tru
```

HeuslerAlloy_Inverse_example.py

See also ref.heusler-al-loy-common.notes.

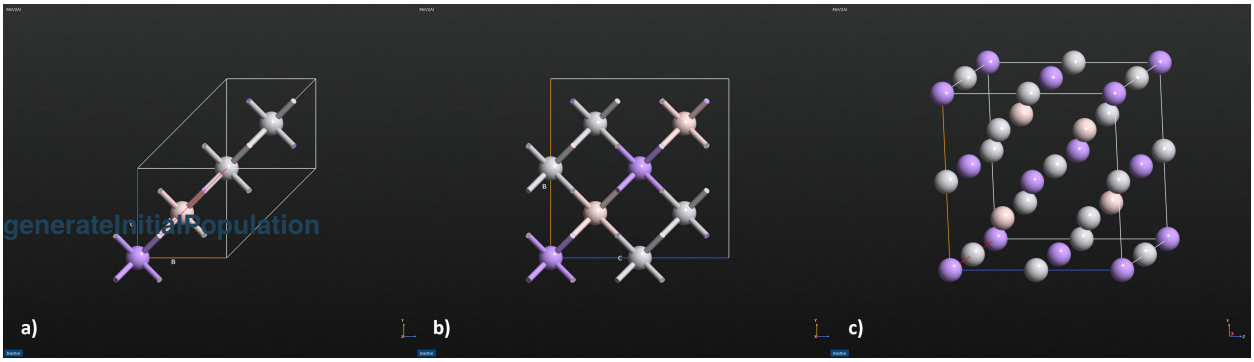


Fig. 4.47: Example of an inverse Heusler configuration. X: Manganese (purple), X: Vanadium (grey), Z: Aluminium (pink). a) primitive cell, b) conventional cell, c) conventional cell (with equivalent atoms)

generateInit

Generates an initial population to be used with *CrystalStructurePrediction*. Each structure is generated by placing the atoms in a random unit cell and performing a local geometry optimization (*OptimizeGeometry*) at the specified pressure. The initial volume of the unit cell (the volume before optimization) is either given explicitly or roughly estimated from the covalent radii of the atoms.

Parameters

- **elements** (*list*) – A list of elements. Each entry in this list corresponds to one atom.
- **population_size** (*int*) – The number of configurations to generate.
- **calculator** (Calculator) – The calculator used to perform the geometry optimization calculations.
- **volume** (*PhysicalQuantity* of type volume) – An initial guess for the volume of the unit cell. If None, an initial guess will be made based on the covalent radii of the atoms. **Default:** None.
- **max_forces** (*PhysicalQuantity* of type force) – The geometry optimization uses this parameter to determine when the optimization has converged. The max_forces parameter represents the maximum force on an atom. **Default:** 0.01*eV/Angstrom
- **max_stress** (*PhysicalQuantity* of type pressure) – The geometry optimization uses this parameter to determine when the optimization has converged. The max_stress represents the maximum deviation from the target stress. **Default:**

0.1*GPa

- **max_steps** (*int*) – The maximum number of optimization steps. **Default:** 200
- **external_pressure** (*PhysicalQuantity* of type pressure) – The target external pressure of the system. **Default:** 0.0*GPa
- **rng** (*numpy.random.RandomState*) – The random number generator used to generate the random number stream. **Default:** A new random number generator (seeded from OS entropy pool)
- **log_filename_prefix** (*str | None*) – The filename prefix to use when running in parallel. If None is specified then all logging will be done to stdout. **Default:** initial_population_
- **max_iterations** (*int*) – The maximum number of iterations used to generate a random configuration that meets the hard constraints. **Default:** 100

Returns
A Population object which contains a list of BulkConfiguration objects.

Return type
Population

generateRingConfigurations

generateRingConfigurations(*initial_defect_pair*, *initial_defect_pair_generator*, *final_defect_pair=None*, *final_defect_pair_generator=None*)

Form a ring of defects describing the migration mechanism of a substitutional-vacancy pair. The ring is created by moving the vacancy between positions connecting the initial and final defect pairs. Note that in the definition of the defect pairs and their generators, the substitutional must come first and the vacancy second.

Parameters

- **initial_defect_pair** (*NamedPointDefect*) – Starting point for the ring mechanism of a vacancy substitutional pair.
- **initial_defect_pair_generator** (*DefectPairGenerator*) – The defect list that produced the starting point of the pair.
- **final_defect_pair** (*NamedPointDefect*) – Final point for the ring mechanism of a vacancy substitutional pair. **Default:** Same as the initial pair.
- **final_defect_pair_generator** (*DefectPairGenerator*) – The defect list that produced the final point of the pair. **Default:** Same as the initial generator.

Returns
List of defects containing the initial, intermediate and final states of the ring migration and the BulkConfigurations created by them.

Return type
list of *NamedPointDefect*, list of *BulkConfiguration*

Usage Examples

Use the *generateRingConfigurations* function to generate a ring migration path between a substitutional and vacancy pair. The vacancy is obtained from a *VacancyGenerator* and the substitutional from a *SubstitutionalGenerator*. Both form a pair through the *DefectPairGenerator*.

```
substitutionals = SubstitutionalGenerator(host_configuration, Germanium)
vacancies = VacancyGenerator(host_configuration)
pairs = DefectPairGenerator(substitutionals, vacancies, 2 * Ang, 3 * Ang)

defects, configurations = generateRingConfigurations(
   initial_defect_pair=pairs.uniqueDefects()[0],
   initial_defect_pair_generator=pairs,
   final_defect_pair=pairs.uniqueDefects()[1],
   final_defect_pair_generator=pairs
)
```

Notes

In some materials, the most favorable diffusion pathway of a closely associated vacancy impurity (substitutional) pair is through a ring mechanism. Instead of the substitutional atom directly swapping places with a lattice atom (high barrier), the vacancy moves along a *ring* of lattice sites (low barrier) until the substitutional itself can move into the vacancy, located at a previously inaccessible site (see Fig. 4.48 b).

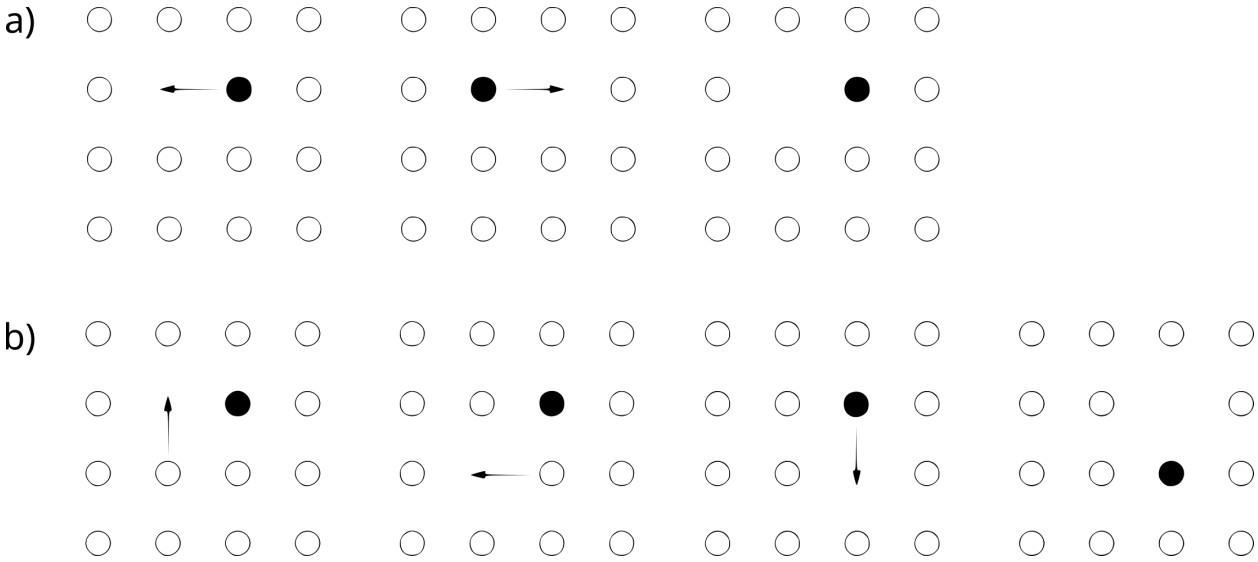


Fig. 4.48: Substitutional movement without (a) and with (b) ring mechanism.

Without considering the correlated substitutional and vacancy movement, the substitutional would only be able to oscillate between two positions (see Fig. 4.48 a). As a result, the diffusivity would be severely underestimated.

The *generateRingConfigurations* function returns a list of consecutive defect pairs as *NamedPointDefect* representing the steps of the ring. These defects can be used to calculate formation energies through the *ChargedPointDefectConfiguration* framework and diffusivities via *DefectDiffusivity*. For simple visualization and *NudgedElasticBand* generation, the defect pairs are also returned as *BulkConfiguration*.

Note that the host configuration used in the initial defect generation should be a supercell with enough repetitions to create room for the ring.

genericAlloy

genericAlloy(*configurations*, *percentages*, *algorithm*, *repetitions*, *rng=None*)

Create alloy by creating a super cell and mixing the given configurations.

Parameters

- **configurations** (*List of BulkConfigurations*) – The configurations to use in the alloy.
- **percentages** (*List of floats*) – The percentages.
- **algorithm** (*FixedFraction | NormalDistribution*) – The algorithm to use.
- **repetitions** (*tuple (length 3)*) – The repetitions in the A, B, and C directions.
- **rng** (*numpy.random.RandomState*) – The random number generator used to generate the random number stream. If None, a new random number generator stream will be made with a seed obtained from the operating systems entropy pool. **Default:** numpy.random.RandomState().

Returns
The created alloy.

Return type
BulkConfiguration

Usage Examples

Generate a AuAg alloy with 50/50 composition and repeat the alloy 4x in all dimensions.

```
# -- Setup initial crystal configuration ---
# Set up lattice
lattice = FaceCenteredCubic(4.07825*Angstrom)

# Define elements
elements = [Gold]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
Au_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Set up lattice
lattice = FaceCenteredCubic(4.0857*Angstrom)

# Define elements
elements = [Silver]
```

(continues on next page)

(continued from previous page)

```
# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
Ag_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# --- Create an alloy ---
alloy_configuration = genericAlloy(
    configurations=[Au_configuration, Ag_configuration],
    percentages=[50.0, 50.0],
    algorithm=Normal_distribution,
    repetitions=(4, 4, 4)
)
```

genericAlloy_example.py

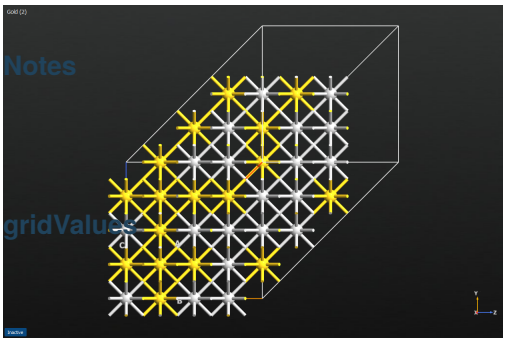


Fig. 4.49: Example alloy generated by genericAlloy. Yellow: Au, Grey/White: Ag. Note that the atom distribution is random each time genericAlloy is run.

•The percentages must sum up to exactly 100.0.

gridValues(*data*, *cell=None*, *cell_vectors=None*, *cell_origin=None*, *unit=None*)

Routine to convert a grid of values contained in a numpy array to a GridValues object that can be opened in QuantumATK. The grid data values can be given either as a physical quantity array or as a plain values (no unit) numpy array.

Parameters

•**data** (*PhysicalQuantity array or numpy.array*) – Either a PhysicalQuantity array or a plain (no unit) numpy.array containing the data set as a 3D grid. If a plain numpy array is passed, the unit parameter must be specified.

•**cell** (*BravaisLattice*) – A unit cell object describing the cell dimension and origin. This option is mutually exclusive with a specification made through the arguments cell_vectors and cell_origin. The cell must be specified either through this argument or cell_vectors/cell_origin.

•**cell_vectors** (*list*) – Unit cell vectors, as a list of three vectors. This option is mutually exclusive with a specification through the cell argument.

•**cell_origin** (*PhysicalQuantity array*) – Cell origin. If not specified, the default is (0.0,0.0,0.0)*Bohr, but only if the cell_vec-

tors argument was specified. If the cell argument was specified instead, an error will be raised. **Default:** (0.0,0.0,0.0)*Bohr

- **unit** (Unit) – The unit to assign the grid. This parameter is only evaluated, and becomes mandatory, if the data parameter contains a plain (no unit) array. If the data parameter already has a unit, this parameter is ignored.

Usage Examples

As the *GridValues* class is made for internal use, constructing a generic grid must be done using the function *gridValues()*:

```
# Setup generic grid values object with random data.
data = numpy.random.random([30, 30, 30]) * Angstrom**-3
cell = SimpleCubic(4.0*Angstrom)
grid_values = gridValues(data, cell)
```

isMainProcess

isMainProcess()

Function for querying if a process is the delegator or worker process.

Returns
True if the process is the delegator process in the parallel computation. False if it is a worker process.

Return type
bool

localExactExchangeFraction

localExactExchangeFraction(*configuration*)

Parameters
configuration (*BulkConfiguration* | *MoleculeConfiguration*) – The configuration of the system.

Returns
The local exact exchange fraction as a GridValues object. This is primarily useful for calculations using hybrid functional and in particular when using HSE06LocalDDH or HSE06MetallicLocalDDH functionals, where the exact exchange fraction is a local property and varies over the unit cell. For HSE06 or HSE06DDH functionals a grid with a constant value will be returned. For non-hybrid functional a zero-valued grid is returned.

Return type
GridValues

Usage Examples

In this example we perform a calculation for a Ag-Si interface using a small SZP basis set. The calculation is performed using the HSE06MetallicLocalDDH functional.

```
# Set minimal log verbosity
setVerbosity(MinimalLog)

# %% bulk_configuration

# Set up lattice
vector_a = [3.8400140859116654, 0.0, 0.0]*Angstrom
vector_b = [-2.3513304794962353e-16, 3.8400140859116654, 0.0]*Angstrom
vector_c = [0.0, 0.0, 29.888932833784352]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.25          ,  0.75          ,  0.033457199879],
                          [ 0.75          ,  0.25          ,  0.033457199879],
                          [ 0.25          ,  0.25          ,  0.101805240653],
                          [ 0.75          ,  0.75          ,  0.101805240653],
                          [ 0.25          ,  0.75          ,  0.170153281426],
                          [ 0.75          ,  0.25          ,  0.170153281426],
                          [ 0.25          ,  0.25          ,  0.2385013222  ],
                          [ 0.75          ,  0.75          ,  0.2385013222  ],
                          [ 0.25          ,  0.75          ,  0.306849362973],
                          [ 0.75          ,  0.25          ,  0.306849362973],
                          [ 0.25          ,  0.25          ,  0.375197403747],
                          [ 0.75          ,  0.75          ,  0.375197403747],
                          [ 0.25          ,  0.75          ,  0.421464791126],
                          [ 0.75          ,  0.75          ,  0.466887958543],
                          [ 0.75          ,  0.25          ,  0.512311125959],
                          [ 0.25          ,  0.25          ,  0.557734293375],
                          [ 0.25          ,  0.75          ,  0.603157460791],
                          [ 0.75          ,  0.75          ,  0.648580628207],
                          [ 0.75          ,  0.25          ,  0.694003795624],
                          [ 0.25          ,  0.25          ,  0.73942696304 ],
                          [ 0.25          ,  0.75          ,  0.784850130456],
                          [ 0.75          ,  0.75          ,  0.830273297872],
                          [ 0.75          ,  0.25          ,  0.875696465288],
                          [ 0.25          ,  0.25          ,  0.921119632704],
                          [ 0.25          ,  0.75          ,  0.966542800121]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
```

(continues on next page)

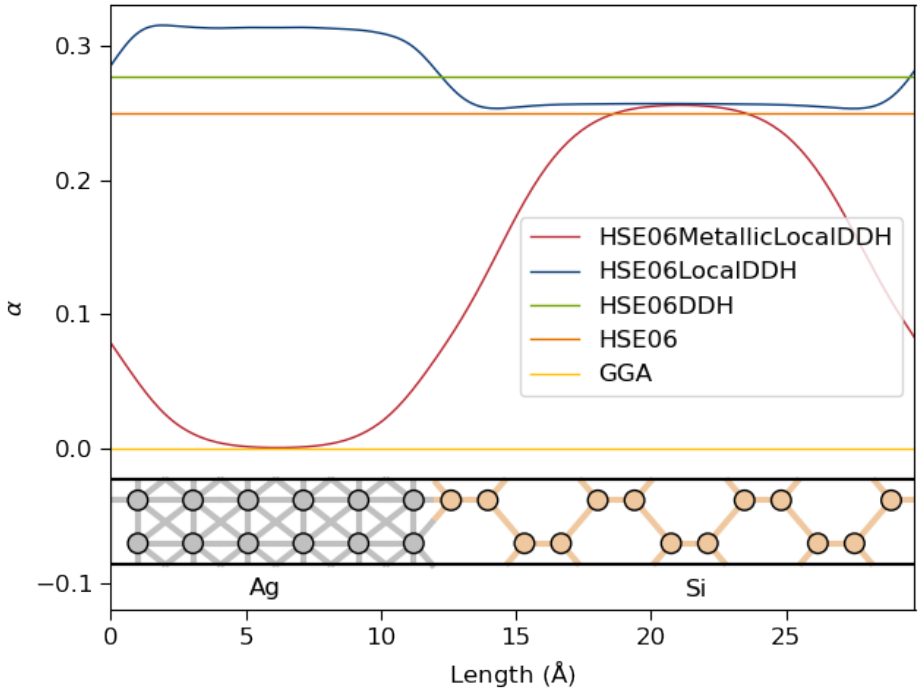
(continued from previous page)

```
    fractional_coordinates=fractional_coordinates
    )



# LCAOCalculator

#----------------------------------------
# Exchange-Correlation
#----------------------------------------
exchange_correlation = HybridGGA.HSE06MetallicLocalDDH

k_point_sampling = KpointDensity(
    density_a=4.0*Angstrom,
    density_b=4.0*Angstrom,
    density_c=4.0*Angstrom
)

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling
)

#----------------------------------------
# Basis Set
#----------------------------------------
basis_set = [
    GGABasis.Silicon_SingleZetaPolarized,
    GGABasis.Silver_SingleZetaPolarized,
    ]

calculator = LCAOCalculator(
    basis_set=basis_set,
    exchange_correlation=exchange_correlation,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    checkpoint_handler=NoCheckpointHandler
)


# Set Calculator
bulk_configuration.setCalculator(calculator)

bulk_configuration.update()
nlsave('Ag-Si-interface.hdf5', bulk_configuration)


# Calculate local exact exchange fraction
local_exchange_fraction = localExactExchangeFraction(bulk_configuration)
nlsave('Ag-Si-interface.hdf5', local_exchange_fraction)
```

`Ag-Si-interface-HSE06MetallicLocalDDH.py`

Results from similar scripts, but with different exchange correlation functionals are shown in the figure below

The HSE06MetallicLocalDDH gives an exchange fraction around 0.257 in the center of the Si part, while it decays to



zero in the center of the metallic Ag part. This means that the (central) Ag is calculated using the PBE functional. Using HSE06LocalDDH leads to an increased exchange fraction in the Ag region, while HSE06DDH results in a constant exchange fraction of 0.277, slightly larger than the normal HSE06 value of 0.25. For a GGA calculation the exact exchange fraction is zero.

maximumClusterDiameter

maximumClusterDiameter(*configuration*, *selection*, *maximum_expansion_order=None*)

Calculate the starting/default values for the maximum cluster diameters for each expansion order [2, 9] for the given configuration and atom selection.

Parameters

- **configuration** (BulkConfiguration) – The configuration.
- **selection** (*sequence of type int*) – The selected atoms.
- **maximum_expansion_order** (*int*) – If given, return the maximum cluster diameters up to this order.

Returns
All or a particular maximum cluster diameter.

Return type
dict

mergeCells

mergeCells(*first_configuration*, *second_configuration*, *merge_axis='A'*, *vacuum=PhysicalQuantity(0.0, Ang)*)

Merge two BulkConfigurations into a single BulkConfiguration along a given merge axis and separated by a chosen amount of vacuum. The order is fixed such that the first configuration is always before the second configuration. To reverse the order, simply call this function with the two configurations swapped.

Parameters

- **first_configuration** (BulkConfiguration) – The first configuration, which is merged before the second.
- **second_configuration** (BulkConfiguration) – The second configuration, which is merged after the first.
- **merge_axis** (*str*) – The axis along which to merge. Valid values: A, B or C.
- **vacuum** (*PhysicalQuantity* of type length) – The amount of vacuum to put between the two configurations.

Returns
The merged configuration.

Return type
BulkConfiguration

Usage Examples

Merge a silver (Ag) *BulkConfiguration* with a palladium (Pd) *BulkConfiguration*.

```
# --- Create the two configurations ---
# Set up lattice
vector_a = [3.8903, 0.0, 0.0] * Angstrom
vector_b = [0.0, 3.8903, 0.0] * Angstrom
vector_c = [0.0, 0.0, 3.8903] * Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Palladium, Palladium, Palladium, Palladium]

# Define coordinates
fractional_coordinates = [[0. ,  0. ,  0. ],
                          [0.5,  0.5,  0. ],
                          [0.5,  0. ,  0.5],
                          [0. ,  0.5,  0.5]]

# Set up configuration
first_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Set up lattice
vector_a = [4.0857, 0.0, 0.0] * Angstrom
vector_b = [0.0, 4.0857, 0.0] * Angstrom
vector_c = [0.0, 0.0, 4.0857] * Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)
```

(continues on next page)

(continued from previous page)

```
# Define elements
elements = [Silver, Silver, Silver, Silver]

# Define coordinates
fractional_coordinates = [[0. ,  0. ,  0. ],
                          [0.5,  0.5,  0. ],
                          [0.5,  0. ,  0.5],
                          [0. ,  0.5,  0.5]]

# Set up configuration
second_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Merge the first configuration with the second configuration, in that order, along the 'C
↪' axis, separated by 0.5 Ang.
merged_configuration = mergeCells(
    first_configuration=first_configuration,
    second_configuration=second_configuration,
    merge_axis='C',
    vacuum=0.5 * Angstrom
)
```

mergeCells_example.py

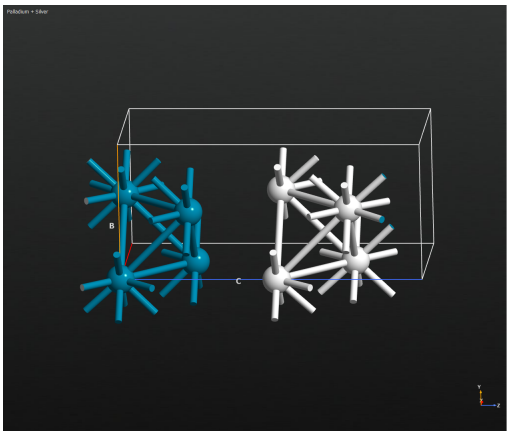


Fig. 4.50: Example output configuration from the mergeCells usage example, where the blue molecules are Pd and the grey molecules are Ag.

General

The function *mergeCells* is a simple utility which can be used to put two *BulkConfiguration* next to each other, when there is no need to consider atomic bonding at the interface or periodic boundary conditions not in the merge direction. This can be useful in cases where a structure such as a nanotube needs to be placed on the surface of a metal slab.

Notes

•In the MergeCells Builder Plugin, there is the option to set the *merge order* such that the configuration dropped on the plugin is placed either Before or After the active configuration in the Builder stash. This option has been omitted in the functon mergeCells, as changing the merge order is equivalent to changing the order of the input.

minimalSurfaceWire

minimalSurfaceWire(*bulk_configuration*, *radius*, *direction*, *energy_100*, *energy_110*, *energy_111*, *cut_perpendicular*)

Generate a nanowire by cutting the bulk along the direction.

Parameters

•**bulk_configuration** (BulkConfiguration) – The configuration to create the nanowire from.

•**radius** (*PhysicalQuantity of type length*) – The radius of the wire.

•**direction** (*str*) – The wire direction.

•**energy_100** (*PhysicalQuantity of type energy*) – The energy of the 100 plane.

•**energy_110** (*PhysicalQuantity of type energy*) – The energy of the 110 plane.

•**energy_111** (*PhysicalQuantity of type energy*) – The energy of the 111 plane.

•**cut_perpendicular** (*bool*) – Only cut perpendicular to the wire direction.

Returns
The nanowire.

Return type
BulkConfiguration

Usage Examples

Create a Ag nanowire with a radius of 7.5 Angstrom in the (100) direction. Repeat the wire in the c-direction 8x to make the wire more visually obvious.

```
# --- Create the input configuration
# Set up lattice
lattice = FaceCenteredCubic(4.0857*Angstrom)

# Define elements
elements = [Silver]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# --- Create a nanowire ---
configuration = minimalSurfaceWire(
    bulk_configuration=bulk_configuration,
    radius=7.5 * Angstrom,
    direction='100',
    energy_100=1.0 * eV,
    energy_110=1.0 * eV,
    energy_111=1.0 * eV,
    cut_perpendicular=True
)

# Repeat the configuration in the c-direction
configuration = repeat(configuration, na=1, nb=1, nc=8, stack_systems=False)
```

minimalSurfaceWire_example.py

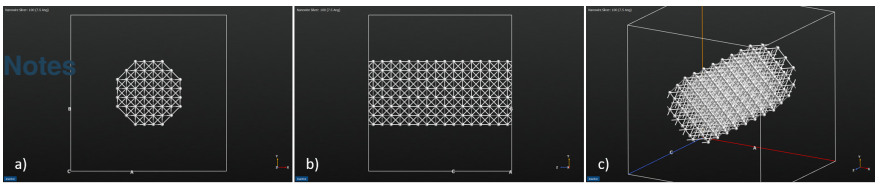


Fig. 4.51: Example nanowire generated by minimalSurfaceWire. a) Cross-section, b) side-view, c) perspective.

Notes

- The required input configuration must be a BulkConfiguration and have one of the following symmetries: Simple cubic, Face-centered cubic or Body-centered cubic.
- Wire direction options: 100, 110, 111

mirror

mirror(*configuration*, *indices*, *normal*, *u0*)

Mirror the elements in rows through the plane given by the surface normal containing the point u0.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration containing the atoms.
- **indices** (*All | sequence of integers*) – The indices of the atoms to change.
- **normal** (*sequence*) – The plane normal.
- **u0** (*sequence*) – A point in the plane.

Returns

The mirrored configuration.

Return type
AtomicConfiguration

modifyDoping

modifyDoping(*configuration*, *tag*, *new_charge=None*, *new_doping_type=None*, *new_tag=None*)
Modify the doping on a configuration.

Parameters

•**configuration** (*AtomicConfiguration*) – The configuration to dope.

•**tag** (*str*) – The tag of the doping.

•**new_charge** (*float | None*) – The new doping charge to apply. This value is interpreted as being in the unit e/atom.

•**new_doping_type** (*NType | PType | None*) – The new doping type to apply to the tagged atoms.

•**new_tag** (*str | None*) – The new tag to assign to the tagged atoms.

Usage Examples

Modify the doping charge, type and tag on the selection of atoms tagged doping_0.

```
# --- Setup a configuration ---
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 16.291800000000002]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.           ,  0.           ,  0.           ],
                          [ 0.           ,  0.           ,  0.333333333333],
                          [ 0.           ,  0.           ,  0.666666666667],
                          [ 0.25         ,  0.25         ,  0.083333333333],
                          [ 0.25         ,  0.25         ,  0.416666666667],
                          [ 0.25         ,  0.25         ,  0.75          ],
                          [ 0.5          ,  0.5          ,  0.            ],
                          [ 0.5          ,  0.5          ,  0.333333333333],
                          [ 0.5          ,  0.5          ,  0.666666666667],
                          [ 0.75         ,  0.75         ,  0.083333333333],
                          [ 0.75         ,  0.75         ,  0.416666666667],
                          [ 0.75         ,  0.75         ,  0.75          ],
                          [ 0.5          ,  0.           ,  0.166666666667],
                          [ 0.5          ,  0.           ,  0.5           ],
                          [ 0.5          ,  0.           ,  0.833333333333],
                          [ 0.75         ,  0.25         ,  0.25          ],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.75          ,  0.25          ,  0.583333333333],
                          [ 0.75          ,  0.25          ,  0.916666666667],
                          [ 0.            ,  0.5           ,  0.166666666667],
                          [ 0.            ,  0.5           ,  0.5           ],
                          [ 0.            ,  0.5           ,  0.833333333333],
                          [ 0.25          ,  0.75          ,  0.25          ],
                          [ 0.25          ,  0.75          ,  0.583333333333],
                          [ 0.25          ,  0.75          ,  0.916666666667]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Add external potential
external_potential = AtomicCompensationCharge()
bulk_configuration.setExternalPotential(external_potential)

# Make a selection of atoms to add doping to
selected_indices = numpy.array([2, 5, 8, 11, 14, 17, 20, 23])

# Add doping
addDoping(
    configuration=bulk_configuration,
    indices=selected_indices,
    charge=0.000200195107106,
    doping_type=N_type,
    tag='doping_0'
)

# --- Modify all aspects of the doping ---
modifyDoping(
    configuration=bulk_configuration,
    tag='doping_0',
    new_charge=0.000400390214212,
    new_doping_type=P_type,
    new_tag='my_doping'
)
```

modifyDoping_example.py

See also ref.dopingcommon.notes.

moleculeFromSmiles

moleculeFromSmiles(*smiles*, *set_bonds=False*)

Convert a SMILES string to a molecule configuration.

Parameters

- **smiles** (*str*) – The SMILES string to convert to a molecule.
- **set_bonds** (*bool*) – Whether static bonds are set on the configuration. **Default:** False

Returns
The MoleculeConfiguration represented by the SMILES string.

Return type
MoleculeConfiguration

Usage Examples

In this example a SMILES string for 4-methylbenzoic acid is converted into a molecule. This conversion is implemented in a custom workflow block. The SMILES string can be given as input to the block, allowing it to be easily re-used for other molecules. The molecule is then optimized with the OPLS-AA potential, and the total energy is calculated.

The workflow for the calculation can be downloaded `here` with a corresponding python script `moleculeFromSmiles_Example.py`.

Notes

This function converts a Simplified Molecular Input Line Entry System (SMILES) string to a molecule. SMILES strings can be used to represent the essential atom and bonding information of molecules. The SMILES string consists of the atomic symbols of non-hydrogen elements of the molecule arranged in the order in which they are bonded together. Hydrogen atoms are assumed to complete the bond number of each atom. In organic molecules, aromatic and aliphatic atoms can be distinguished by using lower-case symbols for aromatic atoms and upper-case for aliphatic atoms. Double and triple bonds can also be specified by placing a = symbol for double bonds and a # symbol for triple bonds. Branches can be added by encasing each branch in parenthesis. Rings can also be created by tagging the start and end atoms of the ring with an identifying number.

The `moleculeFromSmiles()` function takes the SMILES string to be converted and an optional boolean argument `set_bonds`. Setting this to `True` adds to the returned configuration the bonds specified by the SMILES string. By default no bonds are added to the configuration.

nldelete

nldelete(*filename*, *object_id=None*)

Function for deleting NLObjects from HDF5 files.

Parameters

- **filename** (*string*) – The file that an object should be removed from.
- **object_id** (*string*) – Object_id of the object that should be deleted.

Usage Example

To delete a *QuantumATK* object with the `object_id = 'my_obj'` from the file `my_file.hdf5` (for reclaiming the freed space, see *nlrepack*).

```
nldelete('my_file.hdf5', object_id='my_obj')
```

nlprint

nlprint(*object*, *stream=None*)

Print an ASCII representation of the object to the stream pointed to by stream. A typical usage is to store the data as an ASCII file and then use a third party application for generating a visualization of the data.

Parameters

- **object** (*QuantumATK object*) – The object whose data should be printed.
- **stream** (*python stream*) – The stream the data should be printed to. **Default:** `NLPrintLogger()`

Usage Examples

Print the data of a *Bandstructure* to the screen

```
# Set up a bandstructure object
bandstructure = Bandstructure(bulk_configuration)
nlprint(bandstructure)
```

Store the data of an *ElectronDifferenceDensity* in the file `grid.dat`

```
# Set up an electron difference density
electron_difference_density = ElectronDifferenceDensity(molecule_configuration)

# Store grid data on disk
f = open('grid.dat', 'w')
electron_difference_density.nlprint(f)
f.close()
```

nlread

nlread(*filename*, *class_type=None*, *object_id=None*, *read_state=None*)

Function for reading configurations and results from files. The following file types are supported: HDF5 (only from ATK 2017 and later), NetCDF (only from ATK 10.8 and later), GPAW, ASE, XYZ, Python scripts (also from ATK 2008.10 and earlier), CAR and TRAJ (ASE trajectories).

Parameters

- **filename** (*str*) – The file that objects should be loaded from.
- **class_type** (*class*) – Only load objects of this type (HDF5, NetCDF and GPAW only).
- **object_id** (*str*) – Only load objects that match object_id.
- **read_state** (*bool*) – When `read_state` is `True` the self-consistent calculation data is read from configurations when present. Otherwise, configurations will still have a calculator, but the self-consistent data will not be available.

Returns
A list of the objects loaded from the file.

Return type
list

Usage Examples

Restore the self-consistent state of a previous calculation (to save the state, see the *Usage Examples* for the *nlsave* function).

```
# Read self-consistent calculations
molecule_configuration_list = nlread(
    'nh3_scf.hdf5',
    MoleculeConfiguration
)

# Choose the first configuration in the list
molecule_configuration = molecule_configuration_list[0]
```

nh3_read.py

Restore the object with `object_id = 'id0'` from the NetCDF file `bandstructure.hdf5` (to save the state, consult the *nlsave* function).

```
molecular_energy_spectrum = nlread(filename='spectrum.nc', object_id='id0')
```

Restore all *MolecularEnergySpectrum* objects from the file `spectrum.hdf5` and select the first object.

```
molecular_energy_spectrums = nlread(
    'spectrum.hdf5',
    MolecularEnergySpectrum
    )
molecular_energy_spectrum = molecular_energy_spectrums[0]
```

More details regarding file IO in QuantumATK can be found *here*.

nlrepack

nlrepack(*path*, *fail_silently=False*)

Repacks the specified file(s). If path is a directory, it will be walked recursively and all contained hdf5 files will be repacked. If the reclaimed free space is smaller than 1 MB, the operation is discarded.

Parameters

- **path** (*str*) – The filename or directory.
- **fail_silently** (*bool*) – If True, the function will not raise an exception if h5repack is not found. Instead, a warning will be written to the log.

Returns
The freed disk space in MB.

Return type
float

Usage examples

To repack the hdf5 file `my_file.hdf5` and reclaim unused space (for deleting *QuantumATK* objects from hdf5 files see *nldelete*)

```
nlrepack('my_file.hdf5')
```

or to repack all files in a given directory `/my/directory` and its subdirectorys.

```
nlrepack('/my/directory')
```

nlsave

nlsave(*filename*, *nl_object*, *object_id=''*, *labels=None*, *comment=None*, *hidden=False*)

Function for saving the given object to a NetCDF or HDF5 file. This function is collective over all MPI processes.

Parameters

- **filename** (*string*) – The full or relative filename path that the QuantumATK object should be saved to. If no extension is given, then .hdf5 will be appended to the filename.
- **nl_object** (*A QuantumATK object*) – The QuantumATK object that should be saved to file.
- **object_id** (*string*) – A unique identifier that should be attached to the saved object. If the object_id already exists in the file, the old data will be automatically deleted prior to saving the new object. **Default:** `''` (an empty string)
- **labels** – **Deprecated:** from v2017.0.
- **comment** – **Deprecated:** from v2017.0.
- **hidden** (*bool*) – This flag controls if the object is hidden. Hidden objects do not show up on the Lab Floor. **Default:** False

Usage Examples

Save the self-consistent state of a calculation:

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements = [Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates = [[ 0.      ,  0.      ,  0.124001],
                             [ 0.      ,  0.941173, -0.289336],
                             [ 0.81508, -0.470587, -0.289336],
                             [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Perform SCF loop
molecule_configuration.update()

# Save state
nlsave('nh3_scf.hdf5', molecule_configuration)
```

nh3_save.py

Restore the self-consistent state from a saved file, calculate the *MolecularEnergySpectrum*, and save it to the file `spectrum.hdf5` with `object_id = 'id0'`. This will overwrite any data in the file `spectrum.hdf5` associated with this `object_id`.

```
# Read self-consistent calculation
molecule_configurations = nlread(
    'nh3_scf.hdf5',
    MoleculeConfiguration
    )
molecule_configuration = molecule_configurations[0]

# Calculate the molecular energy spectrum
spectrum = MolecularEnergySpectrum(molecule_configuration)

# Save the molecular energy spectrum
nlsave('spectrum.hdf5', spectrum, 'id0')
```

nh3_spectrum.py

More details regarding file IO in QuantumATK can be found *here*.

optimizeBasisSet

optimizeBasisSet(*pseudopotential=None*, *configuration_list=None*, *configuration_weights=None*, *calculator=None*, *tiers=None*, *initial_pressure=None*, *accuracy=None*, *tmp_directory=None*, *filter_mesh_cutoff=None*)

Function for optimizing an N tiered basis set

Parameters

- **pseudopotential** (*str*) – The name of the pseudopotential to use for calculating the basis set
- **configuration_list** (*list*) – Configurations to use for the optimization. Each entry should be a *MoleculeConfiguration* or a *BulkConfiguration*.
- **configuration_weights** (*list of floats*) – list of weights for weighting the total energies of each configuration in the `configuration_list` **Default:** `[1.0] * len(configuration_list)`
- **calculator** (*LCAOCalculator*) – The calculator to use for the calculation **Default:** A default *LCAOCalculator* instance.
- **tiers** (*non-negative int*) – Number of tiers to include in the basis set **Default:** 8
- **initial_pressure** (*PhysicalQuantity* of type pressure) – Pressure for confining the basis set **Default:** `1.0e8 * Newton * (Meter**-2)`
- **accuracy** (*PhysicalQuantity* of type energy) – The criterion for stopping the optimization of the basis set **Default:** `1.0e-3 * Hartree`
- **tmp_directory** (*string*) – Temporary directory for holding the basis sets **Default:** `'basisset_tmp'`
- **filter_mesh_cutoff** (*PhysicalQuantity* of type energy) – The cutoff of the basis set filter **Default:** `0.0 * Hartree`

Usage Examples

Create a *Trajectory* object from a list of configurations or from a *NudgedElasticBand* object.

orbitalInfo

orbitalInfo(*configuration*)

Calculate quantumnumbers of the orbitals

Parameters
configuration (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration to use for the calculation.

Returns
list with `[(i, l, m), ...]` for all the orbitals, where *i* is site, and (*l*, *m*) are angular quantum numbers.

Return type
list of tuples

Usage Examples

Evaluate on a molecule configuration

```
orbital_info = orbitalInfo(molecule_configuration)
```

Notes

- Get a list of quantum numbers (i, l, m) for each basis orbital, where i is site index and l, m are solid harmonics angular momenta.
- The calculator assigned to the device must be density matrix-based for this function to perform successfully.

processIsMaster

processIsMaster()

Function for querying if a process is a delegator or worker process.

Returns
True if the process is the delegator process in the parallel computation. False if it is a worker process.

Return type
bool

Usage Examples

Printing the total energy to the terminal window. If the program is run in a parallel fashion, the `processIsMaster` function can be used to ensure that only the master process prints to the screen.

```
# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Define the calculator
calculator = HuckelCalculator()
molecule_configuration.setCalculator(calculator)

# Print the total energy to the screen
# and avoid writing this for each process.

energy = TotalEnergy(molecule_configuration).evaluate()

if processIsMaster():
   print('Total Energy =', energy)
```

process_is_master.py

Notes

Care should be taken when using the `processIsMaster` function. Misuse can cause the program to freeze. We recommend that the this function is used only for printing to the terminal while debugging.

randomSeed

randomSeed(**args*)

Generate a random seed.

If any arguments are specified, they are converted to a valid seed for a random number generator. Identical seeds will always produce identical random number generators.

If no arguments are specified, an automatically generated unique random seed is returned.

Parameters
args (*ints | floats | strings*) – The optional arguments.

Returns
The random seed.

Return type
int | numpy.ndarrary

readMetatext

readMetatext(*path*, *object_id*)

Read the metatext string from a particular object in a file.

Parameters

- **path** (*str*) – The path of the file from which the metatext should be read. Only HDF5 is supported.
- **object_id** (*str*) – The object_id to read the metatext from.

Returns
The metatext string if it exists, otherwise None.

Return type
str | None

readTrainingData

readTrainingData(*data_files*, *calculator*, *data_classes=None*)

Read the training data from file.

Parameters

- **data_files** (*str | list*) – Name of the training data file.
- **calculator** (*Calculator*) – Calculator to add to the training sets.
- **data_classes** (list | *MomentTensorPotentialTraining* | *Trajectory* | *ConfigurationDataContainer* | *MDTrajectory* | OptimizationTrajectory) – The type or types of data objects used to construct training sets.

Returns
A list of *TrainingSet* objects containing the configurations in the files.

Return type
list

Usage Examples

Read in initial training data from two HDF5 files that are used as input for an active learning simulation.

```
# Read the initial pre-calculated training data for crystalline and amorphous silicon.
calculator = LCAOCalculator()
training_data = readTrainingData(
    ['Silicon_Crystal_Training_Data.hdf5', 'Silicon_Amorphous_Training_Data.hdf5'],
    calculator,
)
```

read_training_data.py

Notes

The *readTrainingData* utility function provides an automated method to read in various forms of training data intended for MTP training. The output is in the form of a list of *TrainingSet* objects that can be passed as a parameter to an *MomentTensorPotentialTraining* or *ActiveLearningSimulation* class. The first argument `data_files` takes a HDF5 file name or list of HDF5 file names from which to read the data. The second argument `calculator` takes the calculator that was used to generate the energy, forces and stresses in the training set. Where there is a calculator already set in the HDF5 file, this will be added in preference to the one specified in `calculator`.

When the training data is passed to an *MomentTensorPotentialTraining* the added calculator is checked to see if it is consistent with the training calculator to determine if the contained energy, force and stress data needs to be recalculated. When passed to an *ActiveLearningSimulation* the training data is included without checking for calculator consistency. Here configurations without energy, force or stress are not included.

readVampireCurieTemperature

readVampireCurieTemperature(*dir_path*)

Parse Vampire input and output files from a Curie temperature simulation and gather results in a Curie temperature data object.

Parameters
dir_path (*str*) – The path to the Vampire calculation directory.

Returns
The Curie temperature data object.

Return type
SpinDynamicsCurieTemperature

readVampireHysteresisLoop

readVampireHysteresisLoop(*dir_path*)

Parse Vampire input and output files from a Curie temperature simulation and gather results in a Curie temperature data object.

Parameters
dir_path (*str*) – The path to the Vampire calculation directory.

Returns
The Curie temperature data object.

Return type
HysteresisLoop

readVampireTrajectory

readVampireTrajectory(*dir_path*, *trajectory_file_path*)

Read and parse Vampire output files and create a spin dynamics trajectory object containing the data.

Parameters

- **dir_path** (*str*) – The Vampire output directory.
- **trajectory_file_path** (*str*) – The path/filename of the HDF5 file to store the trajectory in.

:return The spin dynamics trajectory. :rtype: SpinDynamicsTrajectory

removeDoping

removeDoping(*configuration*, *tag*)

Remove doping from the tagged atoms of the configuration.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration to affect.
- **tag** (*str*) – The tag of the doped atoms.

Usage Examples

Remove doping on the selection of atoms tagged doping_0.

```
# --- Setup a configuration ---
# Set up lattice
vector_a = [5.4306, 0.0, 0.0]*Angstrom
vector_b = [0.0, 5.4306, 0.0]*Angstrom
vector_c = [0.0, 0.0, 16.291800000000002]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon, Silicon, Silicon, Silicon, Silicon,
            Silicon, Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.          ,  0.          ,  0.          ],
                          [ 0.          ,  0.          ,  0.333333333333],
                          [ 0.          ,  0.          ,  0.666666666667],
                          [ 0.25        ,  0.25        ,  0.083333333333],
                          [ 0.25        ,  0.25        ,  0.416666666667],
                          [ 0.25        ,  0.25        ,  0.75          ],
                          [ 0.5         ,  0.5         ,  0.            ],
                          [ 0.5         ,  0.5         ,  0.333333333333],
                          [ 0.5         ,  0.5         ,  0.666666666667],
                          [ 0.75        ,  0.75        ,  0.083333333333],
                          [ 0.75        ,  0.75        ,  0.416666666667],
                          [ 0.75        ,  0.75        ,  0.75          ],
                          [ 0.5         ,  0.          ,  0.166666666667],
                          [ 0.5         ,  0.          ,  0.5           ],
                          [ 0.5         ,  0.          ,  0.833333333333],
                          [ 0.75        ,  0.25        ,  0.25          ],
                          [ 0.75        ,  0.25        ,  0.583333333333],
                          [ 0.75        ,  0.25        ,  0.916666666667],
                          [ 0.          ,  0.5         ,  0.166666666667],
                          [ 0.          ,  0.5         ,  0.5           ],
                          [ 0.          ,  0.5         ,  0.833333333333],
                          [ 0.25        ,  0.75        ,  0.25          ],
                          [ 0.25        ,  0.75        ,  0.583333333333],
                          [ 0.25        ,  0.75        ,  0.916666666667]]
```

(continues on next page)

(continued from previous page)

```
# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# Add external potential
external_potential = AtomicCompensationCharge()
bulk_configuration.setExternalPotential(external_potential)

# Make a selection of atoms to add doping to
selected_indices = numpy.array([2, 5, 8, 11, 14, 17, 20, 23])

# Add doping
addDoping(
    configuration=bulk_configuration,
    indices=selected_indices,
    charge=0.000200195107106,
    doping_type=N_type,
    tag='doping_0'
)

# --- Remove doping
removeDoping(
    configuration=bulk_configuration,
    tag=['doping_0']
)
```

removeDoping_example.py

See also ref.dopingcommon.notes.

repeat

repeat(*configuration*, *na=1*, *nb=1*, *nc=1*, *stack_systems=None*)

Repeat the *BulkConfiguration* or BaseProbeConfiguration with the integer values na, nb, and nc along the three primitive unit cell vectors.

The repeated system is constructed with a cell of the type UnitCell.

Parameters

- **configuration** (*BulkConfiguration* | BaseProbeConfiguration) – The configuration.
- **na** (*int*) – The repetition along the a-axis. **Default:** 1.
- **nb** (*int*) – The repetition along the b-axis. **Default:** 1.
- **nc** (*int*) – The repetition along the c-axis. **Default:** 1.
- **stack_systems** (*bool*) – If True the basis atoms are repeated as a unit, i.e. a0 and b0 are repeated as: a0,b0,a1,b1, If False the basis atom are repeated individually, i.e. a0 and b0 are repeated as: a0,a1,, b0,b1, **Default:** See the docstring for the configurations repeat method.

Returns
The repeated bulk system.

Return type
BulkConfiguration | BaseProbeConfiguration

rotate

rotate(*configuration*, *indices*, *angle*, *axis*, *u0*, *rotate_lattice=None*)

Rotate the elements in rows the angle theta around the axis (x,y,z) centered at (x0,y0,z0).

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration* | *DeviceConfiguration* | *SurfaceConfiguration*) – The configuration containing the atoms.
- **indices** (*All | sequence of integers*) – The indices of the atoms to change.
- **angle** (*float*) – The rotation angle in degrees.
- **axis** (*sequence*) – The direction of the rotation axis.
- **u0** (*sequence*) – Point the rotation axis goes through.
- **rotate_lattice** (*bool*) – If the lattice should be rotated with the coordinates. Only applies to BulkConfigurations.

Returns
The rotated configuration.

Return type
AtomicConfiguration

runPackmol

runPackmol(*molecules*, *cell_or_host*, *tolerance=None*, *buffer_size=None*, *maximum_loops=None*, *random_seed=None*, *terminal_output=False*)

Run Packmol in a subprocess, interpret the output and return the result as a *BulkConfiguration*.

Parameters

- **molecules** (list of (*MoleculeConfiguration*, int)) – A list of tuples, (Molecules, Molecules count) to include in the packing.
- **cell_or_host** (BravaisLattice | *BulkConfiguration*) – The packing volume or host configuration.
- **tolerance** (*PhysicalQuantity* of type length) – The target minimum intermolecular distance after packing. Corresponds to the tolerance keyword in packmol.
- **buffer_size** (*PhysicalQuantity* of type length) – A buffer of this size is added to the cell vectors after packing to facilitate periodic boundary conditions.
- **maximum_loops** (*int*) – The maximum number of loops in the packing optimization. Corresponds to the nloop keyword in packmol.
- **random_seed** (*int*) – The random seed to give Packmol. Use None to generate a seed automatically.
- **terminal_output** (*bool*) – If True, print Packmol message to the terminal.

Returns
Tuple containing the packed configuration, the success flag from reading the Packmol output and the Packmol output message.

Return type
tuple(*BulkConfiguration*, bool, str)

Usage Examples

The first example demonstrates the packing of 10 CH_4 molecules together with 40 H_2O in a cube with side-length 40 Angstrom, using a maximum of 30 packing loops, a tolerance of 2 Angstrom and a buffer region size, also of 2 Angstrom (see *Notes* for further details on these input parameters):

```
# ------------------------------------------------------------------------
# Methane
# ------------------------------------------------------------------------
# Define elements
elements = [Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
coordinates = [[ -3.67767153e-08,  -5.21170217e-07,  -1.77696852e-05],
               [  6.30525881e-01,  -8.91103633e-01,   2.66535938e-02],
               [ -6.29878738e-01,   2.68953440e-02,   8.91606540e-01],
               [ -6.30920686e-01,  -2.70480029e-02,  -8.90758789e-01],
               [  6.30273980e-01,   8.91262502e-01,  -2.72895999e-02]] * Angstrom

# Set up configuration
methane = MoleculeConfiguration(
    elements,
    coordinates
)

# ------------------------------------------------------------------------
# H2O
# ------------------------------------------------------------------------
# Define elements
elements = [Oxygen, Hydrogen, Hydrogen]

# Define coordinates
coordinates = [[  0.0,        4.0,  0.0],
               [  0.76923955, 4.0, -0.59357141],
               [ -0.76923955, 4.0, -0.59357141]] * Angstrom

# Set up configuration
water = MoleculeConfiguration(
    elements,
    coordinates
)

# Create a list of molecules and their numbers
molecules = [
    (methane, 10),
    (water, 40)
]

# Define the volume in which the molecules should be packed
```

(continues on next page)

(continued from previous page)

```
u1, u2, u3 = numpy.diag([40.0] * 3) * Angstrom
cell = UnitCell(u1, u2, u3)

# Set the size of the buffer region
buffer_size = 2.0 * Angstrom

# Set Packmol parameters
tolerance = 2.0 * Angstrom
maximum_loops = 30

# Run the Packmol script to generate the packed configuration
packed_configuration, packing_successful, packmol_message = runPackmol(
    molecules,
    cell,
    tolerance,
    buffer_size,
    maximum_loops,
)

print(f'Packing successful: {packing_successful}')
print(f'Packmol message: {packmol_message}')
print(f'Returned configuration: {packed_configuration}')
```

runPackmol_molecules_example.py

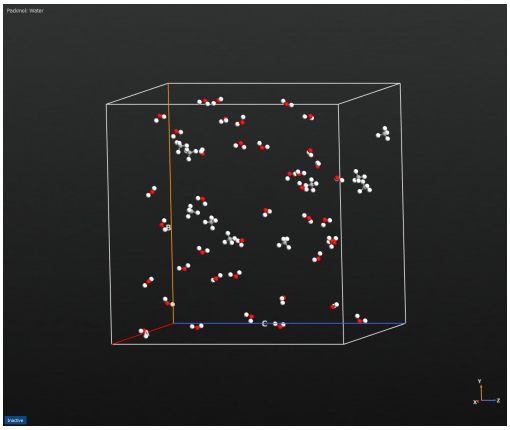


Fig. 4.52: Example output configuration from runPackmol, where the red-white molecules are H_2O and the grey-white molecules are CH_4.

In the code above, the cell is defined as a `Unitcell`, though any valid `BravaisLattice` will work, also those that are non-orthogonal. As an example, try replacing the lines

```
# Define the volume in which the molecules should be packed
u1, u2, u3 = numpy.diag([40.0] * 3) * Angstrom
cell = UnitCell(u1, u2, u3)
```

with

```
# Define the volume in which the molecules should be packed
```

(continues on next page)

(continued from previous page)

```
cell = Triclinic(14.0 * Angstrom, 17.9999 * Angstrom, 22.0 * Angstrom,
                 45.0 * Degrees, 90.0 * Degrees, 90.0 * Degrees)
```

The second example demonstrates the packing of four CH_4 into a zeolite host configuration (loaded from zeolite_LIT.py, which can be found as a separate download below), using a maximum of 30 packing loops, a tolerance of 2 Angstrom and a buffer region size also of 2 Angstrom:

```
# Import the host configuration
from zeolite_LIT import zeolite_LIT

# ------------------------------------------------------------------------
# Methane
# ------------------------------------------------------------------------
# Define elements
elements = [Carbon, Hydrogen, Hydrogen, Hydrogen, Hydrogen]

# Define coordinates
coordinates = [[ -3.67767153e-08,  -5.21170217e-07,  -1.77696852e-05],
               [  6.30525881e-01,  -8.91103633e-01,   2.66535938e-02],
               [ -6.29878738e-01,   2.68953440e-02,   8.91606540e-01],
               [ -6.30920686e-01,  -2.70480029e-02,  -8.90758789e-01],
               [  6.30273980e-01,   8.91262502e-01,  -2.72895999e-02]] * Angstrom

# Set up configuration
methane = MoleculeConfiguration(
    elements,
    coordinates
)

# Create a list of molecules and their numbers
molecules = [(methane, 4)]

# Set the size of the buffer region
buffer_size = 2.0 * Angstrom

# Set Packmol parameters
tolerance = 2.0 * Angstrom
maximum_loops = 30

# Run the Packmol script to generate the packed configuration
packed_configuration, packing_successful, packmol_message = runPackmol(
    molecules,
    zeolite_LIT,
    tolerance,
    buffer_size,
    maximum_loops,
)

print(f'packing successful: {packing_successful}')
print(f'packmol message: {packmol_message}')
print(f'Returned configuration: {packed_configuration}')
```

runPackmol_molecules_with_host_example.py zeolite_LIT.py

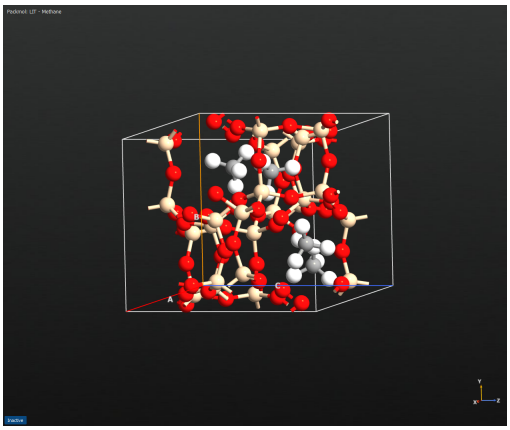


Fig. 4.53: Example output configuration from runPackmol, where the grey-white molecules are CH_4. and the surrounding atoms in red and tan belong to the host configuration.

General

Packmol is a utility used in QuantumATK to generate configurations of packed molecules.

Normally, Packmol is available through the Packmol Builder Plugin.

The function runPackmol is meant to be an easy way to add Packmol functionality to a script written within the QuantumATK framework. It takes the same inputs as the Packmol Builder Plugin, which are:

- Arbitrary types and numbers of molecules to be packed.
- The cell into which the molecules should be packed, as a `BravaisLattice` or a host configuration into which the molecules should be packed.
- Packing density tolerances, i.e. the minimum acceptable distance between molecules.
- A buffer size to avoid faulty packing at periphery of the cell. (See *Notes* for details)
- The maximum number of optimization loops performed by Packmol.

For the original article presenting Packmol, see[1].

The official homepage of Packmol is (as of 2019) http://m3g.iqm.unicamp.br/packmol/home.shtml

Notes

- Molecules must be given as a list of tuples, in which each tuple contains a pair of values: a molecule and the number of this molecule.
- Natively, Packmol does not consider packing tolerances across the boundaries of the packing volume (i.e. it does not support periodic boundary conditions). To overcome this, the original authors suggest to use a buffer region around the packing volume which is subtracted from the total packing volume. This region will be left empty after the packing, and it is suggested to use energy minimization and equilibration on the packing volume *with mirror images* as a follow-up procedure to remove the empty spaces and complete the packing. In general, a buffer_size of 2 Angstroms for a cell length of 100 Angstroms is sufficient, though this may need to be increased for large molecules.

[1] Birgin E.ăG. MartinezăL., AndradeăR. and MartinezăJ. M. PACKMOL: A package for building initial configurations for molecular dynamics simulations. *Journal of Computational Chemistry*, 30(13):2157–2164, February 2009. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/jcc.21224, doi:10.1002/jcc.21224.

•In general, a tolerance of 2 Angstroms is sufficient. Special cases may arise for cyclic systems such as toluene, which may become linked during packing. In such cases, a larger tolerance is more suitable.

runVampireInSubprocess

runVampireInSubprocess(*stream=None*, *stderr_stream=None*)

Run vampire in a subprocess

Parameters

•**stream** (*stream*) – The stream to write the Vampire output to.

•**stderr_stream** (*stream*) – The stream to write stderr to, only for testing.

saveVerbositySettings

saveVerbositySettings(*filename*)

Save the verbosity settings for the current session to a file.

Parameters
filename (*str*) – File where to save the verbosity settings to. If the file exists it will be overwritten.

Usage Examples

Create a custom dictionary of verbosity settings, apply them to the current QuantumATK Python session, and save the settings in a JSON file:

```
# Set custom verbosity settings.
verbosity = {
    'TIME_STAMP': False,
    'MATRIX_SIZES': False,
    'CHECKPOINT_FILE': False,
}
setVerbosity(verbosity)

# Save the verbosity settings in a JSON file.
# Note that this file must be named 'verbosity.json'.
saveVerbositySettings('verbosity.json')
```

Important: The saved file **must** be named `verbosity.json`. Otherwise, QuantumATK will not recognize it as a file containing verbosity settings. See also the *Notes* section below.

You can also save the pre-defined `MinimalLog` verbosity settings:

```
# Set the MinimalLog verbosity settings.
setVerbosity(MinimalLog)

# Save the MinimalLog verbosity settings in a JSON file.
# Note that this file must be named 'verbosity.json'.
saveVerbositySettings('verbosity.json')
```

See the section below to learn how to apply the saved verbosity settings to QuantumATK jobs.

Notes

Most parts of the QuantumATK logging output is associated with a logging region, see *showVerbosityRegions()*. The default QuantumATK verbosity mode is `True` for all logging regions, i.e., to print out all available logging output. However, the verbosity can be manually changed, such that the logging output becomes more condensed, see *setVerbosity()*.

Whenever running QuantumATK Python, the logging verbosity is set in the following manner:

1. The default verbosity mode is print everything; all logging regions are initially verbose.
2. Next, if the file `$HOME/.quantumatk/verbosity.json` exists, QuantumATK will automatically attempt to load verbosity settings from it.
3. Next, if a file `verbosity.json` exists in the current working directory (the directory from which QuantumATK Python is launched), the verbosity settings in that file are automatically loaded. Any verbosity settings already loaded in step 2 will be overridden.
4. Finally, if the *setVerbosity()* function is used in the executed QuantumATK Python script or interactive session, those verbosity settings are the last ones to be applied. Any verbosity settings already loaded in step 2 or 3 will be overridden.

Note: Steps 2 and 3 in the list above require no user action, except for placing a suitable `verbosity.json` file in one or more directories. No need for using the *setVerbosity()* function in QuantumATK Python scripts, except for additional tweaking of the verbosity settings.

In both cases, use the *saveVerbositySettings()* function to create the `verbosity.json` file.

Tip: Step 2 should be used to automatically apply custom **global** verbosity settings, while step 3 is used for automatically applying custom **local** verbosity settings.

---

scanOverNonLinearCoefficients

scanOverNonLinearCoefficients(*number_of_initial_guesses=30*, *perform_optimization=None*, *max_force_rmse_change=None*, *max_steps=None*, *random_seed=None*, *regularization=None*, *energy_only=None*, *radial_function_order=None*, *zero_interaction_pairs=None*, *basis_size=None*, *inner_cutoff_radii=None*, *tapering_cutoff_radii=None*, *outer_cutoff_radii=None*, *mtp_filename_suffix=None*, *cutoff_function=None*, *load_energy=None*, *load_forces=None*, *load_stress=None*, *ridge_regression_regularization=None*, *ridge_regression_cross_validation=None*, *constant_terms=None*, *weights=None*, *forces_cap=None*, *data_tags=None*, *use_element_specific_coefficients=None*, *solver=None*, *use_gpu_acceleration=None*)

Preset protocol for generating a series of MomentTensorPotentialFittingParameters to scan over different initial non-linear coefficients.

Parameters

- **number_of_initial_guesses** (*int*) – The number of initial guesses used to fit an MTP.
- **perform_optimization** (*bool*) – Whether to optimize the non-linear coefficients during the fitting. **Default:** `False`.

•**max_force_rmse_change** (*int*) – The convergence criterion for the change in the RMSE on the force values between subsequent steps in the optimization of the non-linear coefficients. Ignored if *perform_optimization* is False. **Default:** `0.05 * eV / Angstrom`.

•**max_steps** (*int*) – The maximum number of optimization steps to take. Ignored if *perform_optimization* is False. **Default:** `100`.

•**random_seed** (*non-negative int or sequence of non-negative ints*) – The random seed used for generating the initial non-linear coefficients. **Default:** Generated automatically.

•**regularization** (*float*) – The regularization strength for the optimization. Ignored if *perform_optimization* is False. **Default:** `1.0e-2`.

•**energy_only** (*bool*) – Whether to only optimize on the energy values. Ignored if *perform_optimization* is False. **Default:** `True`.

•**radial_function_order** (*int*) – The order of Chebychev polynomials in the expansion of each radial function. Only used when CutoffFunctionChebyshevExpansion is selected as cutoff function. **Default:** 5.

•**zero_interaction_pairs** (None | Sequence of tuples of `PeriodicTableElement`) – If not None, the interactions between the given element pairs will be set to zero.

•**basis_size** (*PredefinedBasisSmall | PredefinedBasisBig | int | str*) – The basis set size to use. If a filename is given the basis is read for this file. **Default:** `PredefinedBasisSmall`.

•**inner_cutoff_radii** (*PhysicalQuantity* of type length | dict of{tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The inner cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `0.5 * Angstrom`.

•**tapering_cutoff_radii** (*PhysicalQuantity* of type length | dict of{tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The tapering cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `0.7 * Angstrom`.

•**outer_cutoff_radii** (*PhysicalQuantity* of type length | dict of{tuple (size 2) of `Element`: *PhysicalQuantity* of type length}) – The outer cutoff radius for each element pair. If a single value is given, this will be used for all pairs. **Default:** `5.0 * Angstrom`.

•**mtp_filename_suffix** (*str*) – The filename suffix for the fitted MTP parameter set. **Default:** Unique name set by *MomentTensorPotentialTraining*.

•**cutoff_function** (*CutoffFunctionChebyshevOriginal | CutoffFunctionChebyshevExpansion*) – The cutoff function to use. **Default:** `CutoffFunctionChebyshevOriginal`.

•**load_energy** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the energy during fitting. **Default:** `LoadQuantityWhereAvailable`.

•**load_forces** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the forces during fitting. **Default:** `LoadQuantityWhereAvailable`.

•**load_stress** (*LoadQuantityAlways | LoadQuantityNever | LoadQuantityWhereAvailable*) – Whether to use the stress during fitting. **Default:** `LoadQuantityWhereAvailable`.

•**ridge_regression_regularization** (*float*) – The regularization strength for the ridge regression model. **Default:** `1.0e-3`.

•**ridge_regression_cross_validation** (*int*) – The number of folds for enabling cross-validation in the ridge regression model. **Default:** No cross-validation.

•**constant_terms** (dict of {`Element`: *PhysicalQuantity* of type energy}) – The energy of an isolated atom for each species. **Default:** Calculated with the fitting calculator. by *MomentTensorPotentialTraining*.

- **weights** (Sequence of *PhysicalQuantity* of type (1/energy, length/energy, length**3/energy)) – The weights used for weighting energy, forces and stress [E, F, S] of every training configuration. If the quantity is not used for fitting or absent from the data, the corresponding weight is ignored. **Default:** No weighting.
- **forces_cap** (*PhysicalQuantity* of type energy / length) – Configurations with max. force magnitudes on an atom larger than this value will be discarded from the training data. **Default:** 100.0 * eV / Angstrom.
- **data_tags** (*None | str | list*) – A single tag or a list of tags identifying the training sets to be used in the fitting. None results in all available training sets being included. **Default:** None.
- **use_element_specific_coefficients** (*bool*) – True, if element-specific coefficients should be used, otherwise False. **Default:** True.
- **solver** (*SVDSolver | ProjectionSolver*) – The linear solver algorithm to determine the linear coefficients. **Default:** ProjectionSolver
- **use_gpu_acceleration** (*bool*) – Switch GPU acceleration using CUDA on or off. This accelerates the singular value decomposition (SVD or Projection) solver when determining the linear coefficients. This requires running the training on a CUDA-enabled GPU, otherwise it has no effect. **Default:** False.

Returns
The non-linear coefficients initial guesses.

Return type
list of *MomentTensorPotentialFittingParameters*:

Usage Examples

Generate training data for a Silicon crystal and set up a random scan over non-linear coefficients. This will generate 30 separate fits sampling initial guesses for the non-linear coefficients.

```
# -*- coding: utf-8 -*-
setVerbosity(MinimalLog)

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(5.4306*Angstrom)

# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.25,  0.25,  0.25]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Calculator
# ------------------------------------------------------------
k_point_sampling = KpointDensity(
    density_a=7.0*Angstrom,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=30.0*Hartree,
    k_point_sampling=k_point_sampling,
    occupation_method=FermiDirac(25.0*meV),
    )

iteration_control_parameters = IterationControlParameters(
    tolerance=5e-05,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    )

# Generate a list of initial guesses for non-linear coefficients.
fitting_parameters_list = scanOverNonLinearCoefficients(
    number_of_initial_guesses=30,
    basis_size=PredefinedBasisSmall,
    mtp_filename_suffix='MTP_fit.mtp',
    random_seed=42,
    perform_optimization=False,

)

# Define RandomDisplacementsParameters used to generate the training sets.
training_sets = RandomDisplacementsParameters(
    reference_configurations=bulk_configuration,
    supercell_repetitions_list=[(1, 1, 1), (2, 2, 2)],
    sample_size=10,
    atomic_rattling_amplitudes=0.15*Angstrom,
    cell_rattling_amplitudes=0.07,
)

# Generate the displaced structures and calculate DFT training data.
mtp_training = MomentTensorPotentialTraining(
    filename='Silicon_crystal_mtp_training_testtest.hdf5',
    object_id='mtp_training',
    training_sets=training_sets,
    calculator=calculator,
    calculate_stress=True,
    fitting_parameters_list=fitting_parameters_list,
)
mtp_training.update()

# Determine the best fit and extract its parameters.
best_fit_index = mtp_training.rankFits(
```

(continues on next page)

(continued from previous page)

```
    data_tags=None,
    weights=[[1, 1, 1], [1, 1, 1]],
    statistical_measure=R2Score
)[0][0]

best_fitting_parameters = mtp_training.fittingParametersList()[best_fit_index]
```

random_scan.py

Notes

The *scanOverNonLinearCoefficients* function provides a list of *MomentTensorPotentialFittingParameters* to set up a sampling of initial guesses. These parameters can be used in the *MomentTensorPotentialTraining* class to generate different fits within the machine-learned Moment Tensor Potential (MTP) framework.

The function combines the APIs of *NonLinearCoefficientsParameters* and *MomentTensorPotentialFittingParameters* while providing a simple way of generating a fitting parameters list without the need for scripting a loop. The names of the MTP potential files will be prepended by an index beginning with the number 0.

After the training has concluded, the fits can be ranked calling the``rankFits()`` method on the *MomentTensorPotential-Training* instance. By default the R2 score between reference data and predicted data is used for ranking. Optionally, weights can be supplied for energies, forces and stresses of training and testing sets. The fitting parameters can be re-obtained by calling `fittingParametersList()` and filtering for the desired fit index.

The parameters can also be further optimized by using the corresponding MTP files as input for a second run or be passed to an *ActiveLearningSimulation*.

```
nl_parameters = NonLinearCoefficientsParameters(
    perform_optimization=True,
    energy_only=True,
    regularization=1.0e-3,
    initial_coefficients='0_MTP_fit.mtp',
)

fitting_parameters = MomentTensorPotentialFittingParameters(
    basis_size=PredefinedBasisSmall,
    mtp_filename='MTP_0_optimization.mtp',
    non_linear_coefficients_parameters=nl_parameters
)
```

The fitting of the linear coefficients can be accelerated by running on a GPU. This is particularly beneficial when the non-linear optimization is switched off via setting `perform_optimization=False`. To enable GPU acceleration, the flag `use_gpu_acceleration` needs to be set to `True`. To make use of this feature, the calculation needs to be run on a GPU that supports CUDA-11.8, otherwise it will fall back to the default CPU mode.

selectNEBByBarrier

selectNEBByBarrier(*nebs*, *forward_barrier=True*, *validate=True*)

Out of a given list of NEBs this function will find and return the one with the smallest barrier.

Parameters

- **nebs** (sequence of *NudgedElasticBand*) – The nudged elastic band configurations to select from.
- **forward_barrier** (*bool*) – True if the forward barrier is to be calculated from the first image and transition state. False for the reverse barrier using the last image. **Default:** True.
- **validate** (*bool*) – If the list of NEBs should be validated to have a transition state before selection.

Returns
The NEB with the lowest barrier.

Return type
NudgedElasticBand

Notes

The *selectNEBByBarrier* utility method can be used to select the *NudgedElasticBand* that has the lowest barrier from a sequence of NEBs. The option `forward_barrier` controls whether the barrier of forward reaction or the reverse reaction is considered. Validation of the NEBs can also be performed before selection. The returned *NudgedElasticBand* is the one that has the lowest barrier.

setVerbosity

setVerbosity(*verbosity_settings=None*, ***kwargs*)

Set the verbosity of logging output in ATK Python.

Both pre-defined verbosity settings (`MinimalLog` or `SilentLog`) or the verbosity of specific logging regions can be specified.

Information on the logging region that each line in the logging output belongs to can be obtained with *showVerbosityRegions()*.

Parameters
verbosity_settings (*dict*) – Dictionary of verbosity settings.

Usage Examples

Pre-defined verbosity settings

Two pre-defined dictionaries of verbosity settings are available. The `MinimalLog` provides a minimalistic but useful QuantumATK logging output:

```
# QuantumATK verbosity
setVerbosity(MinimalLog)

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
```

(continues on next page)

(continued from previous page)

```
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)
molecule_configuration.update()
```

Below is shown a section of the resulting QuantumATK logging output:

```
+-------------------------------------------------------------------------------+
| DiagonalizationSolver parallelization report.                                 |
+-------------------------------------------------------------------------------+
| Total number of processes: 1                                                  |
| Total number of k-points: 1                                                   |
| Processes per k-point: 1                                                      |
| Number of process groups: 1                                                   |
+-------------------------------------------------------------------------------+

+-------------------------------------------------------------------------------+
| Checkpoint Handler                                                            |
| Filename : /tmp/checkpoint84760295.nc                                         |
| Interval : 0.5 h                                                              |
+-------------------------------------------------------------------------------+
|   0 E = -4.24374 dE =  1.000000e+00 dH =  3.517309e-01                        |
|   1 E =  -3.5362 dE =  7.075433e-01 dH =  2.714105e-01                        |
|   2 E = -3.59261 dE =  5.641406e-02 dH =  1.945382e-02                        |
|   3 E = -3.46562 dE =  1.269879e-01 dH =  4.965302e-02                        |
|   4 E = -3.46231 dE =  3.316244e-03 dH =  3.647576e-03                        |
|   5 E = -3.45492 dE =  7.386256e-03 dH =  2.002716e-03                        |
|   6 E = -3.45729 dE =  2.373353e-03 dH =  8.340864e-04                        |
|   7 E = -3.45728 dE =  1.267347e-05 dH =  8.863979e-05                        |
| Calculation Converged in 7 steps                                              |
|                                                                               |
| Fermi Level  = -1.544221 eV                                                   |
+-------------------------------------------------------------------------------+
```

The pre-defined `SilentLog` dictionary completely silences all QuantumATK logging output (verbosity for all logging regions is `False`).

```
# QuantumATK verbosity
setVerbosity(SilentLog)

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
```

(continues on next page)

(continued from previous page)

```
                            [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)
molecule_configuration.update()
```

which results in nothing but a minimal header in the logging output:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| Atomistix ToolKit 2016.0                                                      |
|                                                                               |
+-------------------------------------------------------------------------------+
```

Advanced usage

Instruct QuantumATK Python to avoid printing time stamps, the size of density matrices, progress bars, and checkpoint file information in the logging output:

```
# QuantumATK verbosity
verbosity = {
    'TIME_STAMP': False,
    'MATRIX_SIZES': False,
    'PROGRESS_BARS': False,
    'CHECKPOINT_FILE': False,
}
setVerbosity(verbosity)

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.     ,  0.      ,  0.124001],
                           [ 0.     ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)
molecule_configuration.update()
```

Note that the verbosity settings are given as a Python dictionary (key–value pairs). Below is shown a section of the resulting QuantumATK logging output:

```
+-------------------------------------------------------------------------------+
| DiagonalizationSolver parallelization report.                                 |
+-------------------------------------------------------------------------------+
| Total number of processes: 1                                                  |
```

(continues on next page)

(continued from previous page)

```
| Total number of k-points: 1                                                    |
| Processes per k-point: 1                                                       |
| Number of process groups: 1                                                    |
+--------------------------------------------------------------------------------+
+--------------------------------------------------------------------------------+
| Density Matrix Report                          DM        DD                    |
+--------------------------------------------------------------------------------+
|   0   N   [  0.000 ,  0.000 ,  0.124 ]    5.35681   0.35681                    |
|   1   H   [  0.000 ,  0.941 , -0.289 ]    0.88105  -0.11895                    |
|   2   H   [  0.815 , -0.471 , -0.289 ]    0.88107  -0.11893                    |
|   3   H   [ -0.815 , -0.471 , -0.289 ]    0.88107  -0.11893                    |
+--------------------------------------------------------------------------------+
|   0 E = -4.24374 dE =  1.000000e+00 dH =  3.517309e-01                         |
+--------------------------------------------------------------------------------+
+--------------------------------------------------------------------------------+
| Density Matrix Report                          DM        DD                    |
+--------------------------------------------------------------------------------+
|   0   N   [  0.000 ,  0.000 ,  0.124 ]    5.25696   0.25696                    |
|   1   H   [  0.000 ,  0.941 , -0.289 ]    0.91429  -0.08571                    |
|   2   H   [  0.815 , -0.471 , -0.289 ]    0.91437  -0.08563                    |
|   3   H   [ -0.815 , -0.471 , -0.289 ]    0.91437  -0.08563                    |
+--------------------------------------------------------------------------------+
|   1 E =  -3.5362 dE =  7.075433e-01 dH =  2.714105e-01                         |
+--------------------------------------------------------------------------------+
```

Pre-defined dictionaries of verbosity settings can be combined with keywords, e.g to produce a minimalistic logging output, but keeping the otherwise silenced `SCF_HEADER` verbose:

```
# QuantumATK verbosity
setVerbosity(MinimalLog, SCF_HEADER=True)
```

It is also possible to specify the verbosity dictionary from scratch, e.g. to produce the equivalent of `MinimalLog`:

```
# QuantumATK verbosity
verbosity = {
    'TIME_STAMP': True,
    'MULLIKEN_REPORT': False,
    'PROGRESS_BARS': False,
    'SCF_HEADER': False,
    'GRID_INFO': False,
    'MATRIX_SIZES': False,
    'SCF_PROGRESS_END': False,
    'GRID_TOOL_INFO': False,
    'MEMORY_INFO': False,
    'INITIALIZE_HAMILTONIAN': False,
}
setVerbosity(verbosity)
```

Notes

Most parts of the QuantumATK logging output is associated with a logging region, see *showVerbosityRegions()*. The default QuantumATK verbosity mode is `True` for all logging regions, i.e., to print out all available logging output. Use the *setVerbosity()* function to enable/disable verbosity for specific logging regions.

Information about which logging region each line in the logging output belongs to can be obtained using *showVerbosityRegions()*.

See also *saveVerbositySettings()* for saving custom verbosity settings for later use.

The dictionary of the current verbosity settings can be retrieved with `getVerbosity()`.

MinimalLog

The `MinimalLog` dictionary of verbosity settings is reproduced below:

```
verbosity = {
    'TIME_STAMP': True,
    'MULLIKEN_REPORT': False,
    'PROGRESS_BARS': False,
    'SCF_HEADER': False,
    'GRID_INFO': False,
    'MATRIX_SIZES': False,
    'SCF_PROGRESS_END': False,
    'GRID_TOOL_INFO': False,
    'MEMORY_INFO': False,
    'INITIALIZE_HAMILTONIAN': False,
}
```

All available logging regions

The following is a list all currently available logging regions:

•ALL

•ATOMIC_CALCULATOR

•ATOMIC_CONFIGURATION

•BANDSTRUCTURE

•BONDS_WARNING

•CALCULATOR_UPDATE

•CHECKPOINT_FILE

•CHEMICAL_POTENTIAL

•CONTOUR_INFO

•COUNTERPOISE_CORRECTION

•CRYSTAL_STRUCTURE_PREDICTION

•DEATH_DATE

•DELTATEST

•DENSITYOFSTATES
•DEPRECATION_WARNING
•DEVICEDENSITYOFSTATES
•DIAGONALIZATION_INFO
•DYNAMICAL_MATRIX
•EFFECTIVEBANDSTRUCTURE
•ELASTICCONSTANTS
•ELECTRON_PHONON
•FEAST_DIAGONALIZER
•FERMI_LEVEL
•FORCES_CONVERGED
•GRID_INFO
•GRID_TOOL_INFO
•GRIMME2
•GRIMME3
•HAMILTONIAN_DERIVATIVES
•HARMONIC_CALCULATOR
•HTST
•INELASTICTRANSMISSION
•INITIAL_STATE
•INITIALIZE_HAMILTONIAN
•LBFGS_MESSAGE
•LCAO_INITILIZATION
•MATRIX_SIZES
•MD_PROGRESS
•MD_RESTART
•MD_STEP
•MEMORY_INFO
•MEMORY_USAGE
•MOBILITY
•MULLIKEN_REPORT
•NEB_CONVERGED
•NEB_INFO
•NEB_PARALLEL_INFO
•NEB_PREOPTIMIZATION_INFO
•NEB_STEP

•NEB_WARNING
•NELDER_MEAD
•NLPRINT
•NLREAD_ERROR
•NLREPACK
•NLSAVE_WARNING
•OPTIMIZATION_CELL
•OPTIMIZATION_FORCES_CLASSICAL
•OPTIMIZATION_FORCES_QUANTUM
•OPTIMIZATION_STRESS
•PARALLEL_INFO
•PDOS_WARNING
•PROGRESS_BARS
•RESTART_INFO
•REUSE_ELECTRODE
•SCF_CONVERGED
•SCF_HEADER
•SCF_PROGRESS
•SCF_PROGRESS_END
•SCF_WARNING
•SCFLOOP_INFO
•SLATERORBITAL
•SPINTRANSFERTORQUE
•SPLINE_INTERPOLATION
•STM
•STRESS_CONVERGED
•TIME_STAMP
•TIMING_REPORT
•TRANSMISSIONPATHWAYS
•TRANSMISSIONSPECTRUM

showVerbosityRegions

showVerbosityRegions()

After calling this function, all logging output will be prepended with the tag of the logging region that the outputted line belongs to.

These tags can be used to partially enable or disable logging output. See *setVerbosity()* for more information.

Usage Examples

For all QuantumATK logging output lines, prepend the logging region tag if one exists:

```
# QuantumATK verbosity
showVerbosityRegions()

# Set up configuration
molecule_configuration = MoleculeConfiguration(
    elements=[Nitrogen, Hydrogen, Hydrogen, Hydrogen],
    cartesian_coordinates=[[ 0.      ,  0.      ,  0.124001],
                           [ 0.      ,  0.941173, -0.289336],
                           [ 0.81508, -0.470587, -0.289336],
                           [-0.81508, -0.470587, -0.289336]]*Angstrom
    )

# Calculator
calculator = LCAOCalculator()
molecule_configuration.setCalculator(calculator)
molecule_configuration.update()
```

As seen in the example below, the region tags are shown in capital letters at the beginning of lines in the logging output:

```
CHECKPOINT_FILE          ==> +-------------------------------------------------------------
↪-------------------+
CHECKPOINT_FILE          ==> | Checkpoint Handler                                          ␣
↪                   |
CHECKPOINT_FILE          ==> | Filename : /tmp/checkpoint22107724.nc                       ␣
↪                   |
CHECKPOINT_FILE          ==> | Interval : 0.5 h                                            ␣
↪                   |
CHECKPOINT_FILE          ==> +-------------------------------------------------------------
↪-------------------+

                           |-----------------------------------------------------|
Calculating Eigenvalues    : ====================================================
Calculating Density Matrix : ====================================================

SCF_HEADER               ==> +-------------------------------------------------------------
↪-------------------+
SCF_HEADER               ==> | Density Matrix Report                        DM        DD ␣
↪                   |
SCF_HEADER               ==> +-------------------------------------------------------------
↪-------------------+
MULLIKEN_REPORT          ==> |   0   N   [  0.000 ,  0.000 ,  0.124 ]    5.35681   0.
```

(continues on next page)

(continued from previous page)

```
↪35681                  |
MULLIKEN_REPORT         ==> |   1   H   [  0.000 ,  0.941 , -0.289 ]    0.88105  -0.
↪11895                  |
MULLIKEN_REPORT         ==> |   2   H   [  0.815 , -0.471 , -0.289 ]    0.88107  -0.
↪11893                  |
MULLIKEN_REPORT         ==> |   3   H   [ -0.815 , -0.471 , -0.289 ]    0.88107  -0.
↪11893                  |
MULLIKEN_REPORT         ==> +-------------------------------------------------------------
↪--------------------+
SCF_PROGRESS            ==> |   0 E = -4.24374 dE =  1.000000e+00 dH =  3.517309e-01   ␣
↪                    |
SCF_PROGRESS_END        ==> +-------------------------------------------------------------
↪--------------------+
```

Note: The *showVerbosityRegions()* function does not show the logging region tag for QuantumATK progress bars. However, verbosity for these can still be disabled; the region tag is PROGRESS_BARS.

Notes

See *setVerbosity()* for more information about how to use logging region tags to partially enable or disable QuantumATK logging output.

sort

sort(*configuration*, *indices*, *priority*)

Method to sort the atoms in a configuration according to the fractional coordinates.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration to be sorted(in place).
- **indices** (*All | sequence of integers*) – The indices.
- **priority** (*List with the entries 0,1,2 in some order*) – The priority of the different coordinates in the sorting.

Returns
The sorted configuration.

Return type
AtomicConfiguration

substitutionalAlloy

substitutionalAlloy(*configuration*, *indices*, *algorithm*, *percentage*, *new_element*, *rng=None*)

Create alloy by substituting a percentage of the specified indices with a new element.

Parameters

- **configuration** (BulkConfiguration) – The configuration to create the alloy from.
- **indices** (*List of integers*) – The indices of the atoms to change.
- **algorithm** (*FixedFraction | NormalDistribution*) – The algorithm to randomly change a percentage of the atoms. NormalDistribution means you assign a number [0,1] from a normal distribution across all atoms and only pick those that have a value below the given percentage. FixedFraction means if you set 25%, you get 25% of the total number of atoms changed. The size of the random sample are the percentage of atoms provided, rounded to nearest integer.
- **percentage** (*float*) – The percentage of the atoms that should be changed.
- **new_element** (*PeriodicTable.Element or None (vacancy)*) – The element to change to.
- **rng** (*numpy.random.RandomState*) – The random number generator used to generate the random number stream. If None, a new random number generator stream will be made with a seed obtained from the operating systems entropy pool. **Default:** numpy.random.RandomState().

Returns
The created alloy.

Return type
BulkConfiguration

Usage Examples

Generate a AuAg alloy with 50/50 composition, by first creating a Ag crystal of the desired size. To affect the entire crystal, define a selection that contains all Ag atoms for passing to the substitutionalAlloy function.

```
# --- Create the initial configuration ---
# Set up lattice
vector_a = [8.1714, 0.0, 0.0]*Angstrom
vector_b = [0.0, 8.1714, 0.0]*Angstrom
vector_c = [0.0, 0.0, 8.1714]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver,
            Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver]

# Define coordinates
fractional_coordinates = [[ 0.  ,  0.  ,  0.  ],
                          [ 0.  ,  0.  ,  0.5 ],
                          [ 0.  ,  0.5 ,  0.  ],
                          [ 0.  ,  0.5 ,  0.5 ],
                          [ 0.5 ,  0.  ,  0.  ],
                          [ 0.5 ,  0.  ,  0.5 ],
```

(continues on next page)

(continued from previous page)

```
                           [ 0.5 ,  0.5 ,  0.  ],
                           [ 0.5 ,  0.5 ,  0.5 ],
                           [ 0.25,  0.25,  0.  ],
                           [ 0.25,  0.25,  0.5 ],
                           [ 0.25,  0.75,  0.  ],
                           [ 0.25,  0.75,  0.5 ],
                           [ 0.75,  0.25,  0.  ],
                           [ 0.75,  0.25,  0.5 ],
                           [ 0.75,  0.75,  0.  ],
                           [ 0.75,  0.75,  0.5 ],
                           [ 0.25,  0.  ,  0.25],
                           [ 0.25,  0.  ,  0.75],
                           [ 0.25,  0.5 ,  0.25],
                           [ 0.25,  0.5 ,  0.75],
                           [ 0.75,  0.  ,  0.25],
                           [ 0.75,  0.  ,  0.75],
                           [ 0.75,  0.5 ,  0.25],
                           [ 0.75,  0.5 ,  0.75],
                           [ 0.  ,  0.25,  0.25],
                           [ 0.  ,  0.25,  0.75],
                           [ 0.  ,  0.75,  0.25],
                           [ 0.  ,  0.75,  0.75],
                           [ 0.5 ,  0.25,  0.25],
                           [ 0.5 ,  0.25,  0.75],
                           [ 0.5 ,  0.75,  0.25],
                           [ 0.5 ,  0.75,  0.75]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# --- Generate a substitutional alloy ---
# Create a list of selected atoms. A percentage of these will be
# randomly substituted by the new element.
selection = [0, 1, 2, 3, 4, 5,
             6, 7, 8, 9, 10, 11,
             12, 13, 14, 15, 16,
             17, 18, 19, 20, 21,
             22, 23, 24, 25, 26,
             27, 28, 29, 30, 31]

configuration = substitutionalAlloy(
    configuration=bulk_configuration,
    indices=numpy.array(selection),
    algorithm=NormalDistribution,
    percentage=50.0,
    new_element=Gold
)
```

substitutionalAlloy_example.py

Fig. 4.54: Example alloy generated by substitutionalAlloy. Yellow: Au, Grey/White: Ag. Note that the atom distribution is random each time substitutionalAlloy is run.

- The percentage must be a number between 0.0 and 100.0

supercell(*configuration*, *cell_type=1*, *remove_overlaps=True*)

Create a supercell by expressing the new cell vectors A, B and C as a linear combination of the current cell vectors A, B, and C from the given configuration. Atoms are added as needed to reflect the new cell size.

Parameters

- **configuration** (*BulkConfiguration*) – The original bulk configuration which is to be transformed.
- **cell_type** (*SUPERCELL_TYPE | sequence*) – The type of cell to transform into, which can either be an option from SUPERCELL_TYPE or a 3x3 matrix for a custom transformation matrix.
- **remove_overlaps** (*bool*) – Whether to remove overlapping atoms in the new configuration.

Returns
The supercell configuration.

Return type
BulkConfiguration

surfaceFromBulk

surfaceFromBulk(*bulk_configuration*, *electrode_extension_length*, *vacuum_length*)

Method that takes a bulk configuration and makes a surface configuration with some vacuum at the surface.

Parameters

- **bulk_configuration** (*BulkConfiguration*) – The configuration to modify.
- **electrode_extension_length** (*PhysicalQuantity* of type length | Automatic) – The desired equivalent electrode extension length of the left electrode.
- **vacuum_length** (*PhysicalQuantity* of type length) – The desired vacuum length.

Returns
The surface configuration with adjusted vacuum.

Return type
SurfaceConfiguration

Usage Examples

Create a surface from a bulk bar of gold, with 10 Ang of vacuum at the surface. The surface is also retracted by 4.0782 Ang (always on the left side for `SurfaceConfiguration`) and the electrode length is adjusted to 12.23475 Ang.

```
# --- Create input configuration ---
# Set up lattice
vector_a = [4.07825, 0.0, 0.0] * Angstrom
vector_b = [0.0, 4.07825, 0.0] * Angstrom
vector_c = [0.0, 0.0, 32.626] * Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)

# Define elements
elements = [Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold,
            Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold, Gold]

# Define coordinates
fractional_coordinates = [[ 0.    ,  0.    ,  0.    ],
                          [ 0.    ,  0.    ,  0.125 ],
                          [ 0.    ,  0.    ,  0.25  ],
                          [ 0.    ,  0.    ,  0.375 ],
                          [ 0.    ,  0.    ,  0.5   ],
                          [ 0.    ,  0.    ,  0.625 ],
                          [ 0.    ,  0.    ,  0.75  ],
                          [ 0.    ,  0.    ,  0.875 ],
                          [ 0.5   ,  0.5   ,  0.    ],
                          [ 0.5   ,  0.5   ,  0.125 ],
                          [ 0.5   ,  0.5   ,  0.25  ],
                          [ 0.5   ,  0.5   ,  0.375 ],
                          [ 0.5   ,  0.5   ,  0.5   ],
                          [ 0.5   ,  0.5   ,  0.625 ],
                          [ 0.5   ,  0.5   ,  0.75  ],
                          [ 0.5   ,  0.5   ,  0.875 ],
                          [ 0.5   ,  0.    ,  0.0625],
                          [ 0.5   ,  0.    ,  0.1875],
                          [ 0.5   ,  0.    ,  0.3125],
                          [ 0.5   ,  0.    ,  0.4375],
                          [ 0.5   ,  0.    ,  0.5625],
                          [ 0.5   ,  0.    ,  0.6875],
                          [ 0.5   ,  0.    ,  0.8125],
                          [ 0.5   ,  0.    ,  0.9375],
                          [ 0.    ,  0.5   ,  0.0625],
                          [ 0.    ,  0.5   ,  0.1875],
                          [ 0.    ,  0.5   ,  0.3125],
                          [ 0.    ,  0.5   ,  0.4375],
                          [ 0.    ,  0.5   ,  0.5625],
                          [ 0.    ,  0.5   ,  0.6875],
                          [ 0.    ,  0.5   ,  0.8125],
                          [ 0.    ,  0.5   ,  0.9375]]

# Set up configuration
bulk_configuration = BulkConfiguration(
```

(continues on next page)

(continued from previous page)

```
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
)

# --- Create Surface from bulk ---
# Create a surface configuration with some vacuum.
configuration = surfaceFromBulk(
    bulk_configuration=bulk_configuration,
    electrode_extension_length=12.23475 * Angstrom,
    vacuum_length=10 * Angstrom
)
# Apply the desired extension/retraction to the left side of the central region.
configuration = changeDeviceLength(
    device_configuration=configuration,
    length=-4.0782 * Angstrom,
    extreme=Left
)
```

surfaceFromBulk_example.py

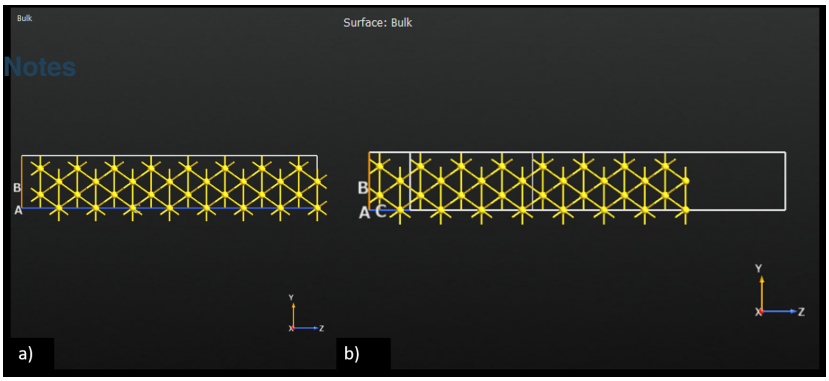


Fig. 4.55: a) The input configuration used in the above example. b) The SurfaceConfiguration generated from the example.

- This function facilitates scripting the creation of a surface, mainly by setting up the vacuum at the surface of the input bulk configuration, though *deviceFromBulk()* is called internally to generate the actual `SurfaceConfiguration`.
- When using Automatic to find suitable left electrode length, the function will raise an NLValueError if no suitable length could be found.

symmetrizeConfiguration

symmetrizeConfiguration(*configuration*, *tolerance*, *primitive=True*, *maximize_symmetry=False*, *use_detect=False*, *angle_tolerance=None*)

Generates a new bulk configuration whose atomic positions and lattice vectors exactly obey the detected space group symmetries.

Parameters

- **configuration** (BulkConfiguration) – The configuation to symmetrize.
- **tolerance** (*PhysicalQuantity of type length*) – The symmetry tolerance.
- **primitive** (*bool*) – Whether to return a primitive or a standard configuration. **Default:** True (primitive).
- **maximize_symmetry** (*bool*) – Explicitly use the highest symmetry Bravais lattice for the symmetrized configuration, which may not match the original Bravais lattice of the input configuration. **Default:** False.
- **use_detect** (*bool*) – If spglib should be used to detect the symmetry. This parameter is mutually exclusive with maximize_symmetry. **Default:** False.
- **angle_tolerance** (*int*) – The tolerance of the lattice angles.

Returns
The symmetrized configuration. The number of atoms might change from the original. If primitive=True this configuration will be the primitive cell.

Return type
BulkConfiguration

Usage Examples

Symmetrize a silicon structure that is close to FCC symmetry.

```
# Set up lattice
vector_a = [0.02, 2.69, 2.7153]*Angstrom
vector_b = [2.7153, 0.0, 2.7153]*Angstrom
vector_c = [2.7153, 2.7153, 0.01]*Angstrom
lattice = UnitCell(vector_a, vector_b, vector_c)
```

(continues on next page)

(continued from previous page)

```
# Define elements
elements = [Silicon, Silicon]

# Define coordinates
fractional_coordinates = [[ 0.007025069439,  0.006459590918, -0.015020491416],
                          [ 0.248715281446,  0.28949668641 ,  0.238503614316]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# The current bulk_configuration has lattice vectors and atomic positions
# that do not obey FCC symetry. This function will produce a new configuration
# that is perfectly symmetric.
bulk_configuration = symmetrizeConfiguration(bulk_configuration, 0.5*Ang)
```

symmetrize.py

Notes

The returned configuration will always be the primitive crystal. This means that the number of atoms may change.

If the atomic positions or lattice vectors are significantly far away from their ideal positions, then the detected space group may not be correct. Increasing the tolerance to a large value (e.g. `tolerance=0.5*Angstrom`) may be useful.

The Bravais lattice type of the returned configuration will be the type with the highest possible symmetry. This means that if the initial Bravais lattice is the generic `UnitCell` the symmetrized configuration might be for example, `FaceCenteredCubic`.

tagPolymerMolecules

tagPolymerMolecules(*configuration*, *minimum_size=0*)

Add tags for polymer molecules to a configuration. Molecules above the minimum size will be labeled with the tag POLYMER_MOLECULE_# where # is an index starting from 0. Molecules below the minimum size are labeled with tag INCLUDED_MOLECULE_# where # is again an index starting from 0. Structures that are periodic in any dimension are not tagged.

Parameters

- **configuration** (*BulkConfiguration*) – The configuration the tags are being added to.
- **minimum_size** (*int*) – The minimum size of what can be considered a polymer molecule. **Default:** 0

Usage Examples

Add tags to a bulk configuration containing polypropylene polymer chains and carbon dioxide molecules to identify the polymers and molecules in the configuration.

```
# Load the initial configuration
configuration = nlread('Polypropylene_CO2.hdf5')[-1]

# Count the number of tags currently on the configuration
number_of_tags = len(configuration.tags())
print(f'There are currently {number_of_tags} on the configuration')

# Set polymer tags on the configuration, setting the minimum size to exclude CO2␣
↪molecules
tagPolymerMolecules(configuration, minimum_size=4)

# Print out the tags and the atoms included in each one
tag_list = sorted(configuration.tags())
for tag in tag_list:
    indices = configuration.indicesFromTags(tag)
    print(tag)
    print(indices)

# Save the tagged configuration
nlsave('Polypropylene_CO2_tagged.hdf5', configuration)
```

`tagPolymerMolecules_Example.py Polypropylene_CO2.hdf5`

Notes

When analyzing polymer melt structures, identifying the polymer chains and other components of the system becomes increasingly difficult as the size of the system grows. The `tagPolymerMolecules` function assists in this by automatically adding tags identifying the different molecular components of a configuration. This is done by analyzing the bonding graph and taking each unconnected subgraph to be a different molecule. Components that have periodicity in any direction, such as a surface are not tagged by this function. A size criteria can also be added to distinguish polymer chains from other molecular components. This is controlled by the parameter `minimum_size`. Molecules with fewer atoms than this size are deemed to be molecules rather than polymer chains. By default this is set to zero, meaning that all molecules are considered to be polymer chains.

On identifying polymers and molecules the tag `POLYMER_MOLECULE_#` is placed on each chain, where `#` is an index starting from 0. Likewise the tag `INCLUDED_MOLECULE_#` is placed on molecules. The polymer tag is the default tag recognized by polymer analysis functions for identifying polymer chains in a configuration.

translate

translate(*configuration*, *indices*, *du*)

Translate the elements with indices by the vector du.

Parameters

- **configuration** (*AtomicConfiguration*) – The configuration containing the atoms.
- **indices** (*All | sequence of integers*) – The indices of the atoms to change.
- **du** (*PhysicalQuantity of type length*) – The translation vector.

Returns
The translated configuration.

Return type
AtomicConfiguration

unitedAtomsConfiguration

unitedAtomsConfiguration(*configuration*, *atom_list=None*, *min_hydrogen=1*, *max_hydrogen=4*)

Return a configuration with united atoms.

Parameters

- **configuration** (*MoleculeConfiguration* | *BulkConfiguration*) – Configuration to be converted.
- **atom_list** (list of *PeriodicTableElement* | All) – List of elements of atoms to be united. **Default:** Carbon
- **min_hydrogen** (*int*) – Minimum number of hydrogen atoms to be united.
- **max_hydrogen** (*int*) – Maximum number of hydrogen atoms to be united.

Returns
The new configuration with united atoms.

Return type
MoleculeConfiguration | *BulkConfiguration*

Usage Examples

In this example an oligomer of polyvinyl alcohol is converted to a united atoms representation. That configuration is then optimized using the DREIDING potential before being converted back to an all-atom configuration. In this conversion only the carbon atoms are united in order to preserve the alcohol hydrogen bonding.

A workflow for the calculation can be downloaded here with a corresponding python script UnitedAtom_Example.py.

Notes

This function converts a configuration from an all-atom representation to a united atoms representation. In united atoms configurations hydrogen atoms are folded into the heavy atom they are attached to, with both the heavy atom and its hydrogen atoms being treated as a single point. This representation of the molecule works mostly with empirical potentials, where atoms are treated as points. It does not with with semi-empirical or density functional theory, where the energy is calculated from the electrons of each atom. The DREIDING potential contains support for carbon united atoms.

Reducing the configuration in this way can make simulations such as molecular dynamics much faster. Here, reducing the number of atoms reduces the number of interactions that must be taken into account by the potential. Removing the light hydrogen atoms also allows longer time steps to be used in integrating the equations of motion. This comes at the expense of maintaining details of the specific hydrogen interactions. In cases where hydrogen atoms do not contribute specifically to the energy, such as in a carbon chain, this approximation works well. In other cases where hydrogen atoms play a significant role, such as in hydrogen bonding, this approximation may significantly reduce the quality of results.

The method takes either a *MoleculeConfiguration* or *BulkConfiguration* as input to convert to a united atoms representation. Optionally ,a list of elements that are to be united with their hydrogen atoms can be given, along with a range of atoms to unite. This allows control over which types of atoms in the configuration are united, allowing important hydrogen contributions to be preserved.

Once a configuration is converted to united atoms, it can be converted back to an all-atom representation using the *explicitHydrogenConfiguration()* function.

wrap

wrap(*configuration*, *indices*, *axes=None*, *min_length=None*)

Wrap the atoms with indices into the cell.

Parameters

- **configuration** (*~.AtomicConfiguration*) – The configuration containing the atoms.
- **indices** (*All | sequence of integers*) – The indices of atoms that should be wrapped.
- **axes** (*tuple*) – A Boolean triple specifying the axes that are wrapped.
- **min_length** (*~.PhysicalQuantity*) – The smallest length in order for two atoms position to be different.

Returns
The wrapped configuration.

Return type
BulkConfiguration

writeVampireFiles

writeVampireFiles(*heisenberg_exchange*, *configuration_parameters*, *simulation_parameters*, *magnetic_anisotropy_energy=None*, *gilbert_damping=None*, *stt_parameters=None*, *path=None*, *materials_filename=None*, *unit_cell_filename=None*, *include_dmi=None*, *unique_magnetic_moment_decimals=2*, *unique_anisotropy_energy_decimals=2*, *non_magnetic_tolerance=None*, *exchange_coupling_tolerance=None*, *dipole_parameters=None*)

Function to write Vampire input files. The function will create three files: The input file named input file, a materials file (default name input.mat), and a unit cell file (default name input.ucf).

Parameters

- **heisenberg_exchange** (*HeisenbergExchange*) – The HeisenbergExchange object containing information about Jij, magnetic moments and possible DMI.
- **configuration_parameters** (*VampireConfigurationParameters*) – Parameter class describing the configuration to be simulated in Vampire.
- **simulation_parameters** (*VampireCurieTemperatureSimulation* | *VampireCurieTemperatureSimulation* |) – Parameter class describing the simulation to be done in Vampire.
- **magnetic_anisotropy_energy** (*MagneticAnisotropyEnergy*) – The MagneticAnisotropyEnergy object containing information about anisotropy constants. **Default:** None, i.e. zero anisotropy.
- **gilbert_damping** (*GilbertDamping*) – The GilbertDamping object containing information about damping constant. **Default:** None, i.e. default damping.
- **stt_parameters** (*Not implemented yet.*) – Data in some format describing spin-transfer-torque. Not implemented yet. **Default:** None, i.e. zero stt..
- **path** (*str*) – Directory in which to write the files. If the directory does not exist, it will be created.
- **materials_filename** (*str*) – Filename for materials file. **Default:** input.mat
- **unit_cell_filename** (*str*) – Filename for unit cell file. **Default:** input.mat
- **include_dmi** (*bool*) – Boolean controling if DMI should be included, if present in the heisenberg_exchange input. **Default:** True
- **unique_magnetic_moment_decimals** (*int | None*) – The number of decimals to round the magnetic moment to in order to group almost identical atoms.
- **unique_anisotropy_energy_decimals** (*int | None*) – The number of decimals to round the anisotropy energy to in order to group almost identical atoms.
- **non_magnetic_tolerance** (*float*) – Tolerance for when atoms are considered non-magnetic. The value corresponds to the magnetic moment in units of Bohr magneton. **Default:** 0.1
- **exchange_coupling_tolerance** (*PhysicalQuantity* of type energy.) – Only exchange interactions larger than this tolerance will be included. **Default:** 0.0 * meV

4.5 Technical Notes

4.5.1 Poisson solvers

The Hartree potential

The self-consistent solution of *DFT: LCAO*, *DFT: Plane Wave* and *Semi Empirical* calculators requires the evaluation of the Hartree potential obtained from a given charge density via the solution of the Poisson equation:

$$\nabla^2 V^H[n](\mathbf{r}) = -\frac{e^2}{4\pi\epsilon_0} n(\mathbf{r})$$

Note that the Hartree potential has units of energy and represent the potential energy of an electron in the corresponding electrostatic potential.

The Poisson equation is a second-order differential equation and a boundary condition (BC) is required in order to fix the solution. The charge density which enters the Poisson equation is evaluate differently in Semi-Empirical and DFT methods (see *The Hartree potential and the electrostatic potential* and references therein). However, the Poisson equation and the BCs formulation are general and valid both for Semi-Empirical and DFT calculators.

In the following sections an overview of the boundary conditions and Poisson solvers available in QuantumATK is provided.

Boundary conditions

A simulated configuration is enclosed in a bounding box, and the Hartree potential is defined on a regular grid within this box. At the facets of the bounding box four basic types of BC can be applied: multipole, periodic, Dirichlet and Neumann.

The following BCs can be imposed on the six facets of the simulation box, with some limitations for specific solvers or configurations which will be listed in the following sections.

- *PeriodicBoundaryCondition* BCs enforce periodicity of the solution along opposite facets.
- *MultipoleBoundaryCondition* BCs are determined by calculating the monopole, dipole and quadrupole moments of the charge distribution inside the simulation cell, and that these moments are used to extrapolate the value of the potential at the boundary. They can be used to simulate charged molecules or system with a large dipole moment which would otherwise require a very large simulation box to allow the potential to go asymptotically to zero.
- *DirichletBoundaryCondition* BCs constrain the potential on the exterior boundary points of a facet S such that:

$$V^H(\mathbf{r}) = V_0(\mathbf{r}), \mathbf{r} \in S$$

V_0 is set to 0 for all configurations except for the electrode facets in a *SurfaceConfiguration* or a *DeviceConfiguration* (see *Boundary Conditions in NEGF*).

In the vacuum facet of a *SurfaceConfiguration* an arbitrary value can be assigned.

- *NeumannBoundaryCondition* BCs set a constraint on the normal derivative of the potential on a facet:

$$\mathbf{n} \cdot \nabla V^H(\mathbf{r}) = V_0'(\mathbf{r}), \mathbf{r} \in S$$

Also in this case $V_0'(\mathbf{r}) = 0$, but it is possible to specify a given value for the electric field on the vacuum side of a *SurfaceConfiguration*. For the electrode facets in a *SurfaceConfiguration* or a *DeviceConfiguration* the value of the derivative is calculate from the electrode bulk potential (see *Boundary Conditions in NEGF*).

The BCs are imposed by using the keyword `boundary_conditions`, supported by any Poisson solver:

```
poisson_solver = MultigridSolver(
boundary_conditions=[[PeriodicBoundaryCondition(),  PeriodicBoundaryCondition()],
                     [DirichletBoundaryCondition(), DirichletBoundaryCondition()],
                     [PeriodicBoundaryCondition(),  PeriodicBoundaryCondition()]]
)
```

Note: Periodic or Neumann boundary conditions only determine the Hartree potential up to an additive constant, which reflects the physics that the bulk electrostatic potential does not have a fixed value relative to the vacuum level. In this case the potential is internally shifted such that its average is zero.

Boundary Conditions in NEGF

When running NEGF calculations (i.e., *DeviceConfiguration* or a *SurfaceConfiguration*), the BCs at the facets where the electrodes are defined are treated in a special way.

Dirichlet BCs on the left surface of the simulation grid at equilibrium are set as:

$$V^H(\mathbf{r}) = V^H_{bulk,left}(\mathbf{r}) - |e|V_{applied,left}, \mathbf{r} \in \Omega_{left}$$

where $V_{bulk,left}(\mathbf{r})$ is the electrode bulk Hartree potential, $V_{applied,left}$ the voltage applied on the left contact, e is the electron charge, and Ω_{left} the left boundary surface. The electode bulk Hartree potential is pre-calculated at the beginning of a NEGF calculation.

On the right surface (for devices) we have:

$$V^H(\mathbf{r}) = V^H_{bulk,right}(\mathbf{r}) - |e|V_{built-in} - |e|V_{applied,right}, \mathbf{r} \in \Omega_{right}$$

where $-|e|V_{built-in} = \mu_{bulk,right} - \mu_{bulk,left}$ is the built-in potential due to the difference of chemical potential in the isolated electrodes at equilibrium. This ensures continuity of the potential between the central region and the electrode at the boundary.

Neumann BCs can also be specified in the transport direction. In this case the derivative of the potential on the facet used as constraint is calculated on each point from the electrode bulk potential:

$$V^{H\prime}(\mathbf{r}) = V^{H\prime}_{bulk,left}(\mathbf{r}), \mathbf{r} \in \Omega_{left}$$

In doing so we enforce continuity of the potential in the central region and in the electrode, apart from a floating rigid shift.

For surfaces, where only the left electrode is defined, a constant value Neumann BC can be applied on the right facet of the grid (vacuum region).

Additional information about NEGF calculations can be found in *NEGF: Device Calculators*.

Dielectric and metallic regions

A user can define regions within the simulation box to simulate dielectric regions or metallic regions. This feature is typically used to simulate gated devices. For a metallic region Ω the Hartree potential is fixed to a constrained value V_0 within this region:

$$V^H(\mathbf{r}) = V_0, \mathbf{r} \in \boldsymbol{\Omega}$$

For a dielectric region Ω with a given relative dielectric constant ϵ_r, the right hand side of the Poisson equation is modified as:

$$\nabla^2 V^H(\mathbf{r}) = -\frac{e^2}{4\pi\epsilon_r\epsilon_0} n(\mathbf{r}), \mathbf{r} \in \Omega$$

Moreover, QuantumATK allows to specify a relative dielectric constant to be used in the whole simulation box around the atoms to model an implicit solvent.

Poisson solvers

The following Poisson solvers are supported by QuantumATK:

- *FastFourierSolver* uses a Fourier Transform with Periodic boundary conditions in all directions. This is a very fast method and the default when simulating *BulkConfiguration* with any calculator.
- *FastFourier2DSolver* can be utilized to simulate *DeviceConfiguration* with Device calculator. It uses a Fourier transform in the directions perpendicular to the transport direction, and a real space multigrid method in the transport direction. It does not support metallic or dielectric region. However, it is fast and accurate and therefore it is the default method for devices without metallic and dielectric regions.
- *MultigridSolver* uses a real space multigrid method. It allows for any kind of boundary conditions in all directions and supports metallic and dielectric regions. It is the default solver for devices with metallic and dielectric regions. It is an iterative method, therefore it is slighlty less accurate than the *FastFourier2DSolver*. It requires a moderate amount of memory, however it is not parallelized and it can be significantly slower than *ParallelConjugateGradientSolver* and *NonuniformGridConjugateGradientSolver* on a parallel environment.
- *ParallelConjugateGradientSolver* uses an iterative solver based on the conjugate gradient method. Similar to the MultigridSolver, ìt allows for different types of boundary conditions, implicit solvents, and the use of metallic and dielectric spatial regions to simulate gates. This method is similar to *MultigridSolver* in terms of accuracy and outperforms it when running calculations in parallel. The solver utilizes PETSc[1] as backengine and it is parallelized.
- *NonuniformGridConjugateGradientSolver* is an efficient method for *BulkConfiguration* and *DeviceConfiguration* with a large vacuum in the A and B directions, and as such is particularly suitable for the simulations of low-dimensional systems (i.e. nanowires, nanotubes, 2D materials). It evaluates the Poisson equation on an auxiliary non-uniform grid in order to reduce the number of degree of freedoms in the vacuum regions, and it is significantly faster and less memory consuming than other solvers in such cases. The solver utilizes the conjugate gradient solver from PETSc[Page 4291, 1] as backengine and it is parallelized.
- *DirectSolver* uses a direct real space solver method. It supports all different types of boundary conditions, implicit solvent, and the use of metallic and dielectric spatial regions. It requires significantly more memory than the other solver methods and it is slower than the Multigrid and the iterative solvers, but it is exact and can be used as a robust reference for small systems. The solver utilizes MUMPS[2] as backengine and it is parallelized.

Note: A tight convergence criteria is set internally for all iterative solvers. Sometimes the iterative solver might fail to converge and a warning is displayed, with an indication of the reached tolerance where applicable. If the error is small and the selfconsistent loop reaches convergence, the calculation is likely converged to a valid solution.

In the figure below support for the various BCs, spatial regions and implicit solvent for the different Poisson solvers is summarized.

In figures below a benchmark on a `Silicon slab` is shown, where all available solvers are compared. The structure is repeated in the A, B direction 1, 2 or 3 times to vary system size. The calculation parameters are chosen such that the runtime is dominated by the calculation of the Hartree potential and a single selfconsistent step is performed. The reported time considers only the time spent assembling and solving the Poisson equation, while the memory is the total system peak memory.

FastFourierSolver and *FastFourier2DSolver* are the fastest available methods. Nevertheless, since they suffer from some limitation in the allowed BCs and lack of support for metallic and dielectric regions, they cannot be used for

[1] Satish Balay, Shrirang Abhyankar, MarkăF. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, WilliamăD. Gropp, Dmitry Karpeyev, Dinesh Kaushik, MatthewăG. Knepley, DaveăA. May, LoisăCurfman McInnes, RichardăTran Mills, Todd Munson, Karl Rupp, Patrick Sanan, BarryăF. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.15, Argonne National Laboratory, 2021. URL: https://www.mcs.anl.gov/petsc.

[2] PatrickăR. Amestoy, Abdou Guermouche, Jean-Yves LExcellent, and Stéphane Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006. Parallel Matrix Algorithms and Applications (PMAA04). URL: https://www.sciencedirect.com/science/article/pii/S0167819105001328, doi:https://doi.org/10.1016/j.parco.2005.07.004.

	Dielectric and metallic regions, implicit solvent	Periodic BC	Neumann BC	Dirichlet BC	Multipole BC	Parallel
FastFourierSolver	✗	✓	✗	✗	✗	✗
FastFourier2DSolver	✗	✓	✓ along C for surfaces	✓ along C for devices and surfaces	✗	✗
MultigridSolver	✓	✓	✓	✓	✓	✗
ParallelConjugateGradientSolver	✓	✓	✓	✓	✓	✓
NonuniformGridConjugateGradientSolver	✓	✓	✓	✓	✓ along A, B	✓
DirectSolver	✓	✓	✓	✓	✓	✓

Fig. 4.56: Support for different BCs, dielectric/metallic regions and MPI parallelization for the available Poisson solvers.

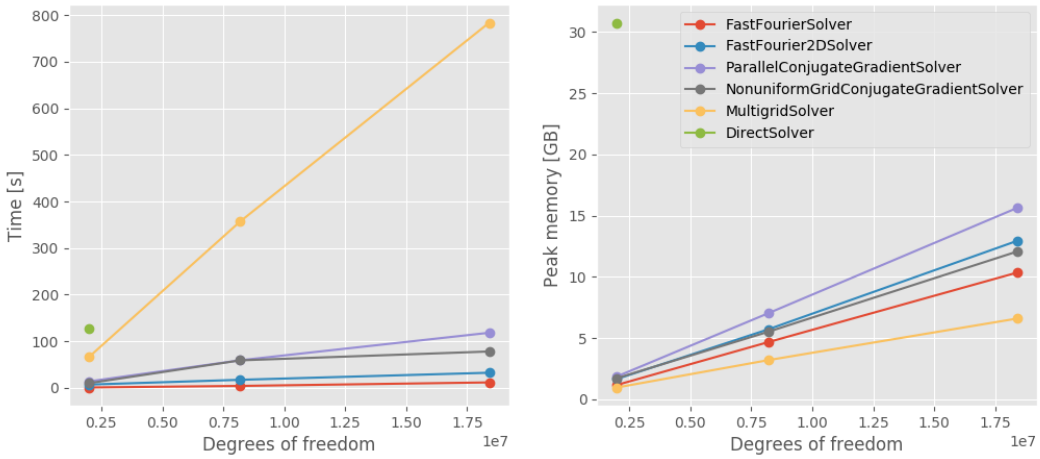


Fig. 4.57: Wallclok time and peak memory consumption as a function of the system size, expressed as number of degrees of freedom of the real-space grid. All calculations are performed on a single Intel(R) Xeon(R) CPU E5-2650 node using 8 MPI processes. For the DirectSolver only one data point could be computed, due to memory limitation.

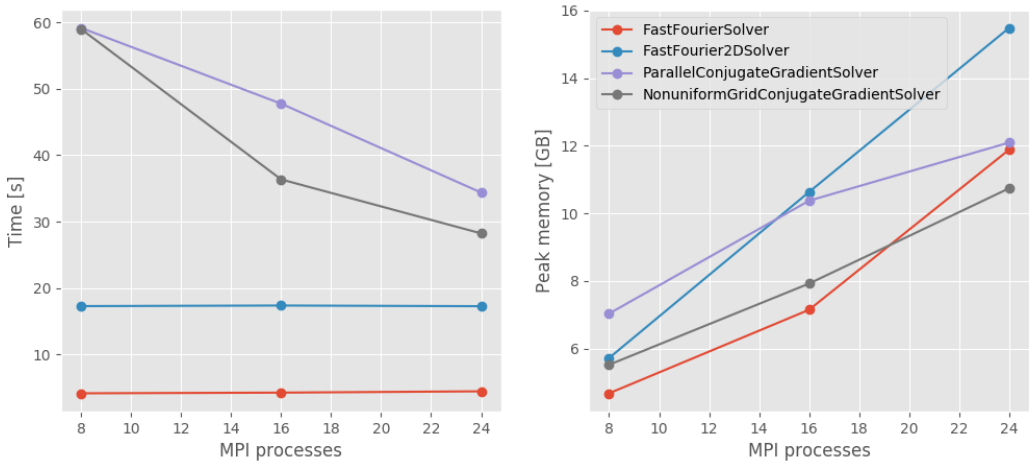


Fig. 4.58: Wallclok time and peak memory consumption as a function of MPI processes. All calculations are performed on a single Intel(R) Xeon(R) CPU E5-2650 node for the slab system repeated twice along A and B (8e6 degrees of freedom). The MultigridSolver is significantly slower and exhibit no parallel scaling, therefore it is not shown.

every possible scenario. For example, they are not suitable to simulate devices with a gate. Furthermore they are not parallelized and can be outperformed by the iterative solvers if enough processes are available.

The iterative solvers are very flexible and can be used for any kind of system. The *MultigridSolver* is not parallelized, but has a very low memory footprint. Therefore it can be preferred for serial calculations, or when memory consumption is a concern. The *ParallelConjugateGradientSolver* requires more memory, but supports parallelization and should be preferred when running on multiple processes. The *NonuniformGridConjugateGradientSolver*) scales similarly to the *ParallelConjugateGradientSolver* since it utilizes the same backengine, and should be preferred when there is vacuum in A and/or B direction.

References

4.5.2 Occupation Methods

This document explains and compares the different occupation methods available in **ATK**. We suggest these guidelines for choosing the occupation method, depending on the system of interest:

- **Systems with a band-gap** (semiconductors, insulators, molecules): Use either *Fermi-Dirac* or *Gaussian* smearing with a low broadening, e.g. around 0.01 eV.
- **Metals** : Use either *Methfessel-Paxton* or *cold* smearing with as large a broadening as possible as long as the entropy contribution to the free energy remains small.

If you are interested in understanding the rationale behind these suggestions, you can read the *Background* and the *Comparison of smearing methods* sections below.

Note: The smearing width of the *Fermi-Dirac* distribution is roughly a factor of two larger than for the other functions. Therefore, in order to obtain the same **k**-point convergence using one of these methods as obtained with the *Fermi-Dirac* method one has to use a broadening of twice the size.

Background

Introduction to smearing methods

In **ATK-DFT** and **ATK-SE** the central object is the electron density $n(\mathbf{r})$ which is calculated from the Kohn-Sham eigenvectors $\psi_n(\mathbf{r})$ by the expression

$$n(\mathbf{r}) = \sum_i f_i |\psi_i(\mathbf{r})|^2,$$

where the index i runs over all states and f_i are the occupation numbers. The latter can be either 1 if the given state is occupied or 0 if the state is unoccupied. In periodic systems such as bulk materials or surfaces, the sum over states involves an integration over the Brillouin zone (BZ) of the system:

$$n(\mathbf{r}) = \sum_i \int_{\mathrm{BZ}} f_{i\mathbf{k}} |\psi_{i\mathbf{k}}(\mathbf{r})|^2 d\mathbf{k}.$$

In practice this integration is carried out numerically by summing over a finite set of $\mathbf{k}$-points. For gapped systems the density and derived quantities, like the total energy, converge quickly with the number of $\mathbf{k}$-points used in the integration. However, for metals the bands crossing the Fermi level are only partially occupied, and a discontinuity exists at the Fermi surface, where the occupancies suddenly jump from 1 to 0. In this case, one will often need a prohibitively large amount of $\mathbf{k}$-points in order to make calculations converge.

The number of $\mathbf{k}$-points needed to make the calculations converge can be drastically reduced by replacing the integer occupation numbers $f_{i\mathbf{k}}$ by a function that varies smoothly from 1 to 0 close to the Fermi level. The most natural choice is the *Fermi-Dirac* distribution,

$$f_{i\mathbf{k}} = \frac{1}{e^{(\epsilon_{i\mathbf{k}} - \mu)/\sigma} + 1},$$

where $\epsilon_{i\mathbf{k}}$ is the energy, μ is the chemical potential and $\sigma = k_{\mathrm{B}}T$ is the broadening.

Fig. 4.59 shows the convergence of the total energy of bulk Aluminum, a typical simple metal.

We see that using σ = 0.03 eV one needs a $25 \times 25 \times 25$ $\mathbf{k}$-point sampling grid (a total of 15625 $\mathbf{k}$-points) in order to converge the total energy within 1 meV, whereas using σ = 0.43 eV the total energy is converged to within 1 meV using only a $13 \times 13 \times 13$ grid (a total of 2197 $\mathbf{k}$-points). This results in a calculation which roughly is a factor of 7 faster.

Free energy functional

When introducing the *Fermi-Dirac* distribution one effectively considers an equivalent system of non-interacting electrons at a temperature T. This also means that the variational internal energy functional $E[n]$ that is minimized is replaced by the free energy functional[1]

$$F[n] = E[n] - TS,$$

where S is the electronic entropy. All derived quantities such as the density, total energy, forces, etc., will therefore depend on the electron temperature T. If one is actually interested in simulating a system at finite temperature, then the free energy is the relevant functional. If that is not the case, the zero-temperature internal energy $E(\sigma = 0)$ can still be extrapolated from the free energy $F(\sigma)$, due to the quadratic dependence (to the lowest order) of both $E(\sigma)$ and $F(\sigma)$ on σ by the formula[2]

$$E_{\sigma \to 0}(\sigma) = \frac{1}{2}[E(\sigma) + F(\sigma)].$$

[1] N.ăD. Mermin. Thermal Properties of the Inhomogeneous Electron Gas. *Phys. Rev.*, 137:A1441–A1443, Mar 1965. doi:10.1103/PhysRev.137.A1441.

[2] M.ăJ. Gillan. Calculation of the vacancy formation energy in aluminium. *J. Phys.: Condens. Matter*, 1(4):689, 1989. doi:10.1088/0953-8984/1/4/005.

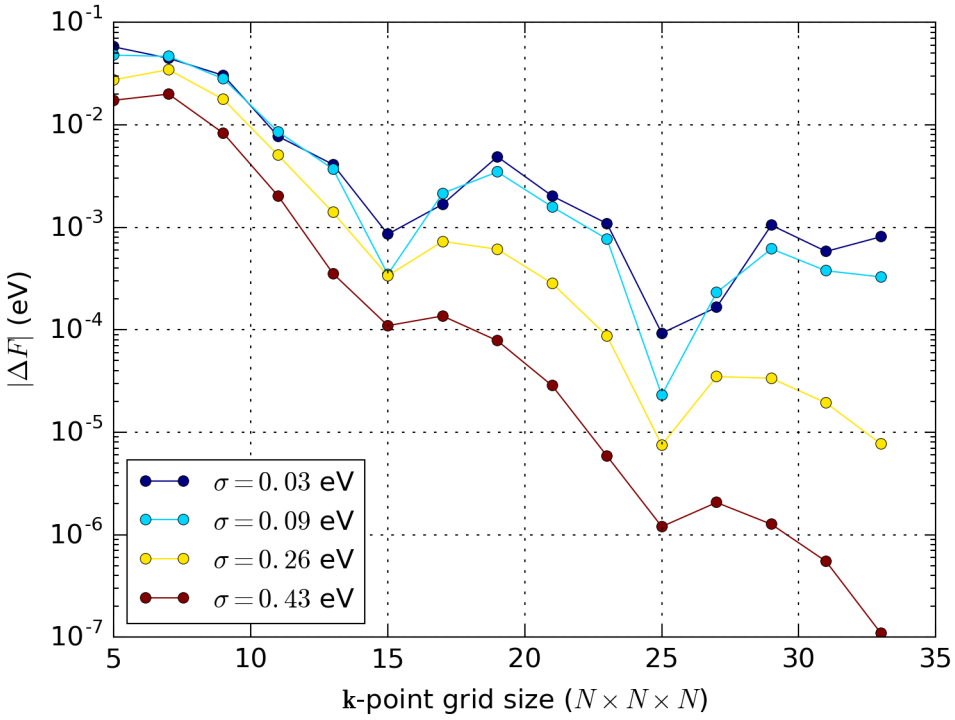


Fig. 4.59: Convergence of the free energy of bulk Aluminum with respect to the k-point sampling using the *Fermi-Dirac* occupation function with different broadenings. The free energy difference, ΔF, is calculated as the difference between the calculation at the given **k**-point sampling and one at $35 \times 35 \times 35$.

Fig. 4.60 shows that, for all the values of the broadening σ considered, the value of the energy extrapolated to $\sigma \to 0$ is basically spot on the actual value. Using this extrapolation method it is thus possible to do calculations with very high broadenings, necessary to converge metallic systems, with a reasonable number of **k**-points and still get an accurate ground state energy. The extrapolated energy is by default shown in the output when doing a total energy calculation using QuantumATK. In order to get the value using QuantumATK see `TotalEnergy`.

Unfortunately a similar extrapolation method does not exist for forces and stress. Thus these properties will be those that correspond to the free energy and will be directly dependent on the chosen broadening. In order to minimize the errors introduced by the broadening, alternative occupation functions for which the entropic contribution to the free energy is smaller than for the *Fermi-Dirac* distribution have been developed.

The different occupation functions are introduced on the basis of considering the density of states given by the expression

$$\rho(\epsilon) = \sum_i \int_{\mathrm{BZ}} \delta(\epsilon_{i\mathbf{k}} - \epsilon) d\mathbf{k}.$$

Since the **k**-point integration is in practice carried out as a sum over a finite number of points, one has to replace the δ-function by a *smeared* function $\tilde{\delta}(x)$, whose width will be determined by a broadening σ. With a choice of smearing function the occupation function is given by

$$f(\epsilon) = \int_{-\infty}^{\mu} \tilde{\delta}(x - \epsilon) dx,$$

where μ is the Fermi level. Even without directly introducing temperature, one can then show that the functional that has to be minimized is the *generalized* free energy[3][4]. The generalized temperature is given by the broadening σ and

[3] M.ăWeinert and J.ăW. Davenport. Fractional occupations and density-functional energies and forces. *Phys. Rev. B*, 45(23):13709–13712, June 1992. doi:10.1103/PhysRevB.45.13709.

[4] Allesandro DeăVita. *The Energetics of Defects and Impurities in Metals and Ionic Materials from First Principles*. PhD thesis, University of

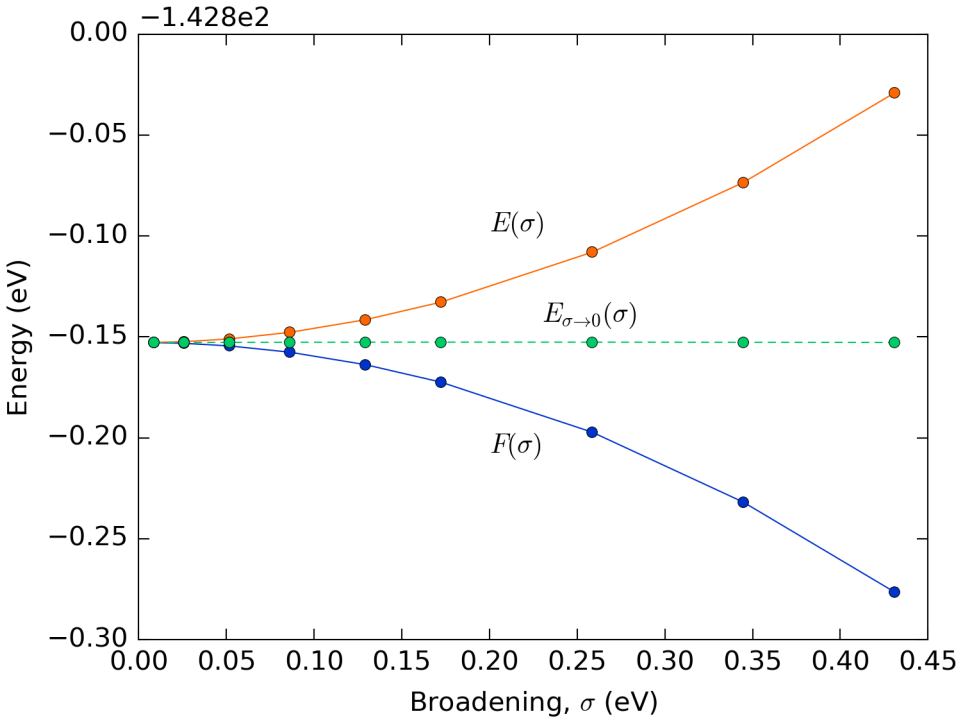


Fig. 4.60: Total free and internal energy of bulk aluminum as a function of the broadening of the Fermi-Dirac distribution. Calculated with a $35 \times 35 \times 35$ **k**-point sampling.

smearing method also directly determines the expression for the generalized entropy.

Comparison of smearing methods

In **ATK** four different smearing methods are available:

- *Fermi-Dirac* distribution (`FermiDirac`)
- *Gaussian* smearing (`GaussianSmearing`)[5]
- *Methfessel-Paxton* smearing (`MethfesselPaxton`)[6]
- *Cold* smearing (`ColdSmearing`)[7]

Warning: While the broadening parameter of the *Fermi-Dirac* distribution has a real physical meaning and can actually be associated with an electronic temperature, this is not true for the other smearing methods, for which the broadening is simply a parameter without a well defined physical meaning!

Fig. 4.61 shows plots of the smeared δ-functions and occupation functions for the different methods.

From the figure we note a few things:

Keele, sep 1992.

[5] C.ă-L. Fu and K.ă-M. Ho. First-principles calculation of the equilibrium ground-state properties of transition metals: Applications to Nb and Mo. *Phys. Rev. B*, 28(10):5480–5486, November 1983. doi:10.1103/PhysRevB.28.5480.

[6] M.ăMethfessel and A.ăT. Paxton. High-precision sampling for Brillouin-zone integration in metals. *Phys. Rev. B*, 40(6):3616–3621, aug 1989. doi:10.1103/PhysRevB.40.3616.

[7] Nicola Marzari, David Vanderbilt, Alessandro DeăVita, and M.ăC. Payne. Thermal Contraction and Disordering of the Al(110) Surface. *Phys. Rev. Lett.*, 82(16):3296–3299, April 1999. doi:10.1103/PhysRevLett.82.3296.

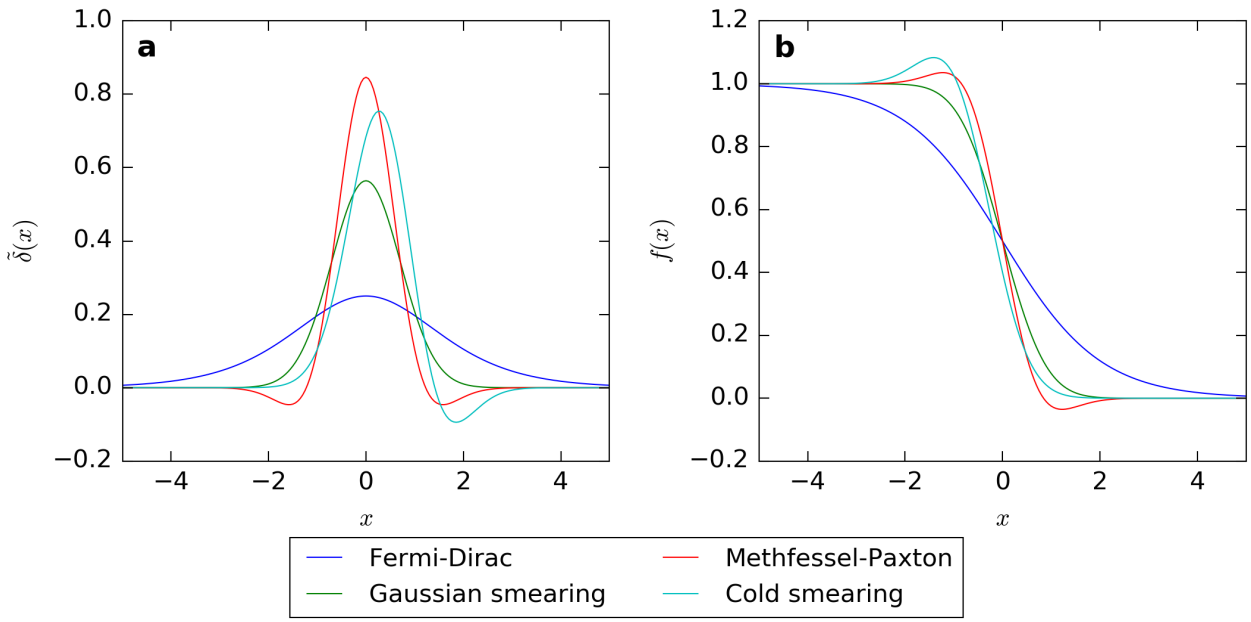


Fig. 4.61: (**a**) Plots of the different smeared delta functions, $\tilde{\delta}(x)$, and (**b**) their corresponding occupation functions $f(x)$, shown as functions of $x = \frac{\epsilon-\mu}{\sigma}$.

i)The width of the *Fermi-Dirac* smearing function is larger than all the others. The ratio of the full width at half maximum between the *Fermi-Dirac* and the *Gaussian* smeared δ-function is

$$\alpha = \frac{\text{FWHM(Fermi-Dirac)}}{\text{FWHM(Gaussian)}} = \frac{2\cosh^{-1}(\sqrt{2})}{\sqrt{\ln(2)}} \approx 2.117.$$

This means that in order to get similar **k**-point convergence as for the *Fermi-Dirac* method one has to use a broadening which is a factor of ~2 larger when using one of the other methods.

ii)The *Methfessel-Paxton* function is special in that the occupations may take unphysical negative values and values larger than one. For insulators and semiconductors as well as too coarsely sampled metals this may lead to negative density of states and a negative density, which may cause computational problems.

iii)The *cold* smearing function is asymmetric but does not attain negative values and problems with negative density are therefore avoided.

From Fig. 4.62 it can be seen that for the *Methfessel-Paxton* and *cold* smearing the free energy, $F(\sigma)$, hardly varies with σ. In fact it can be shown that for these two methods $F(\sigma)$ only has 3rd and higher order dependences on σ.

The low σ dependence on the free energy for *Methfessel-Paxton* and *cold* smearing should be carried over in derived quantities like forces and stress. This is indeed the case as illustrated in Fig. 4.63, which shows the force on the uppermost atom in a 6 layer Aluminum 111 slab as a function of the used broadening.

We see that for small values of the broadening the outer layers seek to contract, whereas this effect is reversed for the *Fermi-Dirac* distribution at a broadening of about 0.75 eV due to the introduced electron gas pressure. For *Methfessel-Paxton* and *cold* smearing the error is neglible for a large range of values for the broadening. This means that one can efficiently calculate accurate forces (for example, during structural optimizations, ab-initio molecular dyanmics and phonons calculations) for metals using sizeable broadenings and relatively low **k**-point samplings.

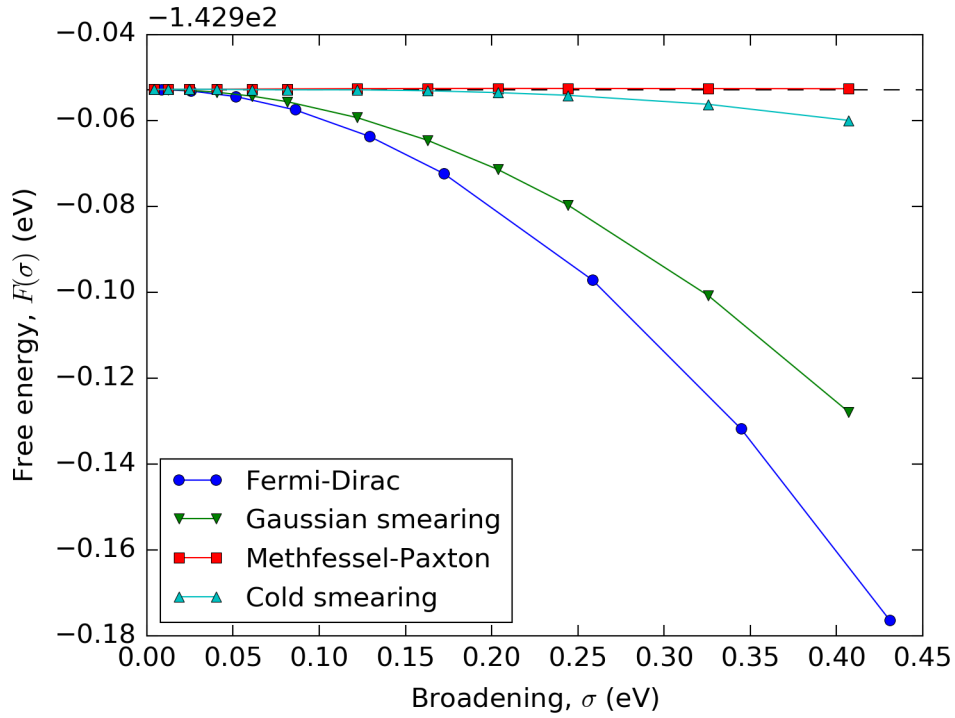


Fig. 4.62: Free energy of bulk Aluminum calculated with different occupation methods and values of the broadening using a **k**-point sampling of $35 \times 35 \times 35$. In order to keep the different methods comparable, the broadening has been multiplied by 2.117 for all but the Fermi-Dirac distribution.

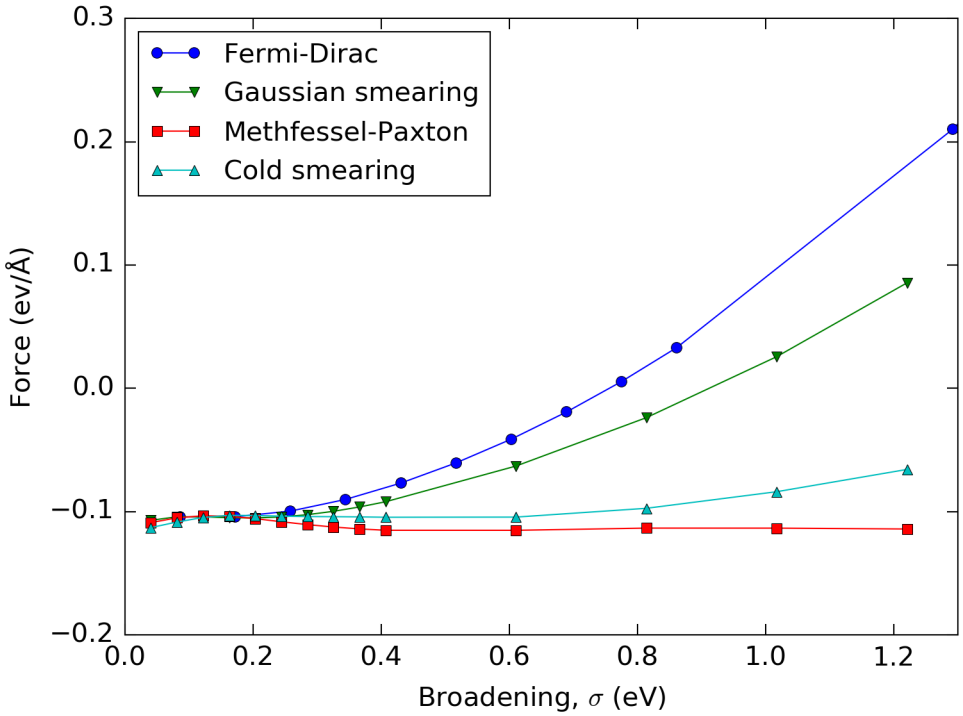


Fig. 4.63: Force on the outermost atom in a 6 layer Aluminum 111 slab as a function of the broadening using the different occupation methods. In order to keep the different methods comparable, the broadening has been multiplied by 2.117 for all but the Fermi-Dirac distribution.

References

4.5.3 Pseudopotentials and basis sets available in QuantumATK

The accuracy of the calculations with the LCAO model in QuantumATK depends on the pseudopotential and basis sets used for the calculations. In recent years we have put a large effort into providing a robust set which can provide good accuracy at reasonably computational resources. Thus, there is an evolution in quality of the pseudopotentials we provide, with the best sets being the *SG15* pseudopotentials available from the ATK-2016 release and PseudoDojo pseudopotentials available from the O-2018.06 release. The theoretical background of the basis sets and pseudopotentials are described in the *QuantumATK Manual*, and the purpose of this note is to guide the user towards the best setting.

Pseudopotentials

The pseudopotentials available for the QuantumATK software are listed in the below table. We see that it shows an evolution with quality of the pseudopotentials for each release. However, not all exchange correlation functionals are available for each type of pseudopotential. For *GGA* and *SOGGA* we recommend to use the *PseudoDojo* pseudopotential. For *LDA* we recommend the *FHI* pseudopotential. For SOLDA we recommend the *OMX*, it can also give higher accuracy for LDA, however, the required mesh cutoff needs to be tested carefully.

Table 4.24: Pseudo- and PAW potential types available for the DFT calculators.

Name	Notes
FHI	Trouiller–Martins type.
HGH	Hartwigsen–Goedecker–Hutter type.
OMX	Fully relativistic. OMX webpage.
SG15	ONCV type, scalar-relativistic (SG15) and fully relativistic (SG15-SO). SG15 webpage.
PseudoDojo	ONCV type, scalar-relativistic (PseudoDojo) and fully relativistic (PseudoDojo-SO). PseudoDojo webpage.
GPAW	PAW potentials. GPAW webpage.
JTH	PAW potentials. JTH webpage.
Suggested	A mixture of GPAW and JTH, chosen to give optimal results for each element.

Through the use of the keyword *NormConservingPseudoPotential*, it is also possible to specify a pseudopotential from a user-specified file. Not all pseudopotentials are available for all QuantumATK DFT calculators and exchange-correlation functional types. Please see the following table for details on each of the pseudopotential types:

Table 4.25: Details about the pseudo- and PAW potential types available for the QuantumATK DFT calculators.

Name	Introduced	Basis types	XC-types	Cutoff [Ha]	Accuracy	Efficiency
FHI	ATK-2008	LCAO, PW	LDA, GGA, MGGA, HGGA	30-100	Low	High
HGH	ATK-2011	LCAO, PW	LDA, GGA, MGGA, HGGA	75-500	Medium	Low
OMX	ATK-2015	LCAO	LDA, GGA, MGGA	75-500	High	Low
OMX-SO	ATK-2015	LCAO	SOLDA, SOGGA, SOMGGA	75-500	High	Low
SG15	ATK-2016	LCAO, PW	GGA, HGGA	30-220	High	Medium
SG15-SO	ATK-2016	LCAO, PW	GGA, SOGGA, HSOGGA	30-220	High	Medium
PseudoDojo	O-2018.06	LCAO, PW	GGA, HGGA	30-125	High	Medium
PseudoDojo-SC	P-2019.03	LCAO, PW	GGA, SOGGA, HSOGGA	30-125	High	Medium
PAW-GPAW	Q-2019.12	PW	LDA, GGA	N/A	High	Medium
PAW-JTH	Q-2019.12	PW	LDA, GGA	N/A	High	Medium
PAW-Suggested	Q-2019.12	PW	GGA	N/A	High	Medium

Warning: The higher values for OMX pseudopotentials, about 300 Hartree or more, are especially relevant for pseudopotentials with semi-core states. They often need these very high mesh cutoffs to give accurate forces.

LCAO basis sets

QuantumATK mainly uses a numerical LCAO *basis sets*. The accuracy depends on the number of orbitals and the range of the orbitals. Increasing the number of orbitals and their range decrease the efficiency and increase the memory requirement. For the latest *SG15* and *PseudoDojo* pseudopotentials we provide basis sets which systematically improve the accuracy. For the other pseudopotentials the improvement in accuracy is less systematic.

SG15/PseudoDojo

Three different basis sets are available for each element when using the SG15/PseudoDojo pseudopotentials; **Medium**, **High**, and **Ultra**. All three derive from the numerical atom-centered basis sets of the FHI-aims package, but have been significantly modified and optimized with respect to computational speed with the *DFT: LCAO* calculator.

The Medium basis set is default for the SG15/PseudoDojo pseudopotentials, and should be sufficient for most applications. However, if extreme accuracy is needed, the High and Ultra basis sets add more basis functions at the expense of increased computational load.

OMX

The OMX basis sets were developed in Refs.[1] and[2], and are optimized towards maximum computational speed and accuracy with the OMX pseudopotentials. The *DFT: LCAO* calculator offers three different OMX basis set sizes for most elements; **Low**, **Medium**, and **High**, which include increasingly more pseudo-atomic orbitals in the basis set. More details are available on the website for the OpenMX database (2013).

FHI

Four different types of basis functions can be used for the FHI pseudopotentials:

•*ConfinedOrbital*

•*AnalyticalSplit*

•*PolarizationOrbital*

•*HydrogenOrbital*

The basis sets available for FHI pseudopotentials are listed below. They are ordered with increasing number of basis orbitals – those with more orbitals are often more accurate, at the expense of increased computational load. The DZP basis set is default.

•SingleZeta: One *ConfinedOrbital* for each occupied valence orbital in the atom.

•DoubleZeta: One *ConfinedOrbital* and one *AnalyticalSplit* for each occupied valence orbital in the atom.

•SingleZetaPolarized: One *ConfinedOrbital* for each occupied valence orbital in the atom, and one *PolarizationOrbital* for the first unoccupied shell in the atom.

•DoubleZetaPolarized: One *ConfinedOrbital* and one *AnalyticalSplit* for each occupied valence orbital in the atom. One *PolarizationOrbital* for the first unoccupied shell in the atom.

•DoubleZetaDoublePolarized: One *ConfinedOrbital* and one *AnalyticalSplit* for each occupied valence orbital in the atom. One *PolarizationOrbital* and one *AnalyticalSplit* for the first unoccupied shell in the atom.

Visualizing the basis functions

To see the type of the basis sets it is most easy to plot the basis functions, like this:

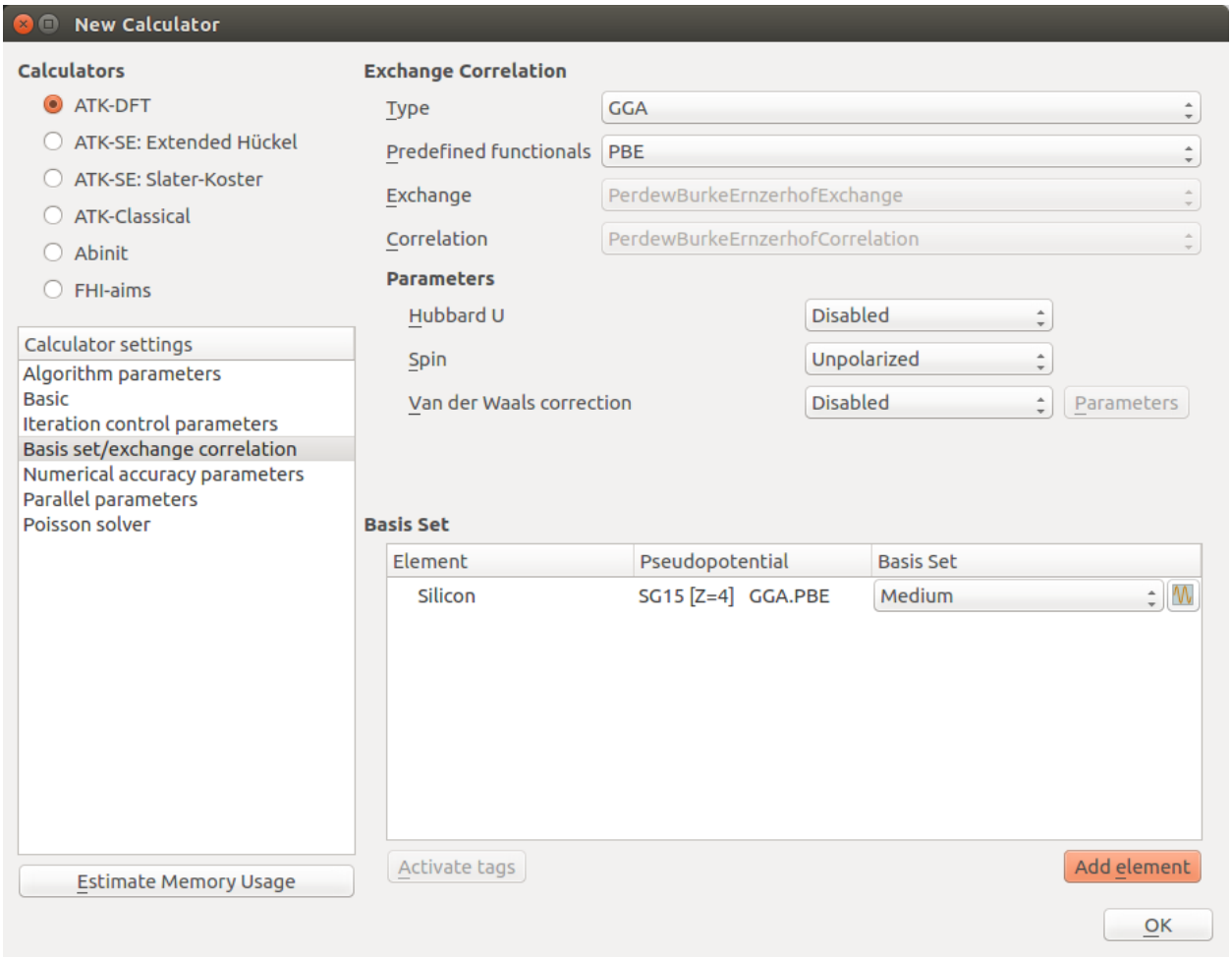
•Open the scripter

•Add an LCAOcalculator block from the Calculators section

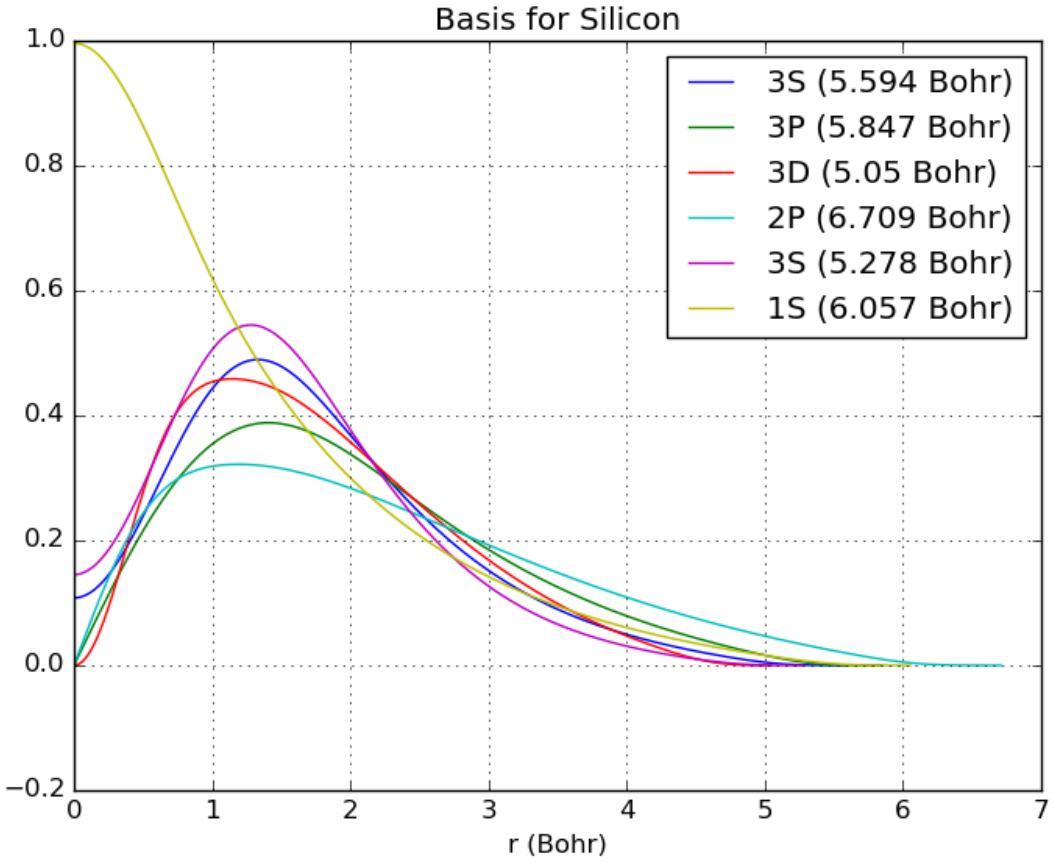
•Go to the Basis set tab at click the plot widget in the line where you define the basis set.

[1] T.ăOzaki. Variationally optimized atomic orbitals for large-scale electronic structures. *Phys. Rev. B*, 67:155108, 2003. doi:10.1103/PhysRevB.67.155108.

[2] T.ăOzaki and H.ăKino. Numerical atomic basis orbitals from H to Kr. *Phys. Rev. B*, 69:195113, 2004. doi:10.1103/PhysRevB.69.195113.



You will then get the following plot.



Accuracy tests for elemental solids

To give an overall idea of the accuracy of each pseudopotential and basis sets we have performed so-called Δ-tests[3] (see the web-site Comparing Solid State DFT Codes, Basis Sets and Potentials). These tests checks accuracy for the pseudopotential and basis set compared with state of the art all-electron calculations. The tests check the accuracy of the pseudopotential and basis sets in reproducing the equation of state of all the elemental solids in the periodic system using the GGA-PBE exchange-correlation functional.

[3] Kurt Lejaeghere, Gustav Bihlmayer, Torbjorn Bjorkman, Peter Blaha, Stefan Blugel, Volker Blum, Damien Caliste, IvanoăE. Castelli, StewartăJ. Clark, Andrea DalăCorso, Stefano deăGironcoli, Thierry Deutsch, JohnăKay Dewhurst, Igor DiăMarco, Claudia Draxl, Marcin Dulak, Olle Eriksson, JoseăA. Flores-Livas, KevinăF. Garrity, Luigi Genovese, Paolo Giannozzi, Matteo Giantomassi, Stefan Goedecker, Xavier Gonze, Oscar Granas, E.ăK.ăU. Gross, Andris Gulans, Francois Gygi, D.ăR. Hamann, PhilăJ. Hasnip, N.ăA.ăW. Holzwarth, Diana Iusan, DominikăB. Jochym, Francois Jollet, Daniel Jones, Georg Kresse, Klaus Koepernik, Emine Kucukbenli, YaroslavăO. Kvashnin, Inka L.ăM. Locht, Sven Lubeck, Martijn Marsman, Nicola Marzari, Ulrike Nitzsche, Lars Nordstrom, Taisuke Ozaki, Lorenzo Paulatto, ChrisăJ. Pickard, Ward Poelmans, Matt I.ăJ. Probert, Keith Refson, Manuel Richter, Gian-Marco Rignanese, Santanu Saha, Matthias Scheffler, Martin Schlipf, Karlheinz Schwarz, Sangeeta Sharma, Francesca Tavazza, Patrik Thunstrom, Alexandre Tkatchenko, Marc Torrent, David Vanderbilt, MichielăJ. van Setten, Veronique VanăSpeybroeck, JohnăM. Wills, JonathanăR. Yates, Guo-Xu Zhang, and Stefaan Cottenier. Reproducibility in density functional theory calculations of solids. *Science*, 2016. doi:10.1126/science.aad3000.

Table 4.26: Accuracy and performance of PBE pseudopotentials and basis types available for the DFT calculators. Performance is an approximate measure of time spent, relative to FHI-DZP. Lower numbers are faster.

Pseudo	Basis	Δ (meV)	Performance
FHI	SZP	39.4	0.5
FHI	DZP	18.2	1
FHI	DZDP	19.1	2
HGH	Tier4	12.8	6
OMX	Medium	8.0	3
OMX	High	2.2	10
SG15	Medium	3.45	2
SG15	High	1.88	6
SG15	Ultra	2.03	25
SG15	PW	1.32	N/A
PseudoDojo	Medium	4.53	2
PseudoDojo	High	1.52	6
PseudoDojo	Ultra	1.40	25
PseudoDojo	PW	1.04	N/A
PAW-GPAW	PW	1.44	N/A
PAW-JTH	PW	0.86	N/A
PAW-Suggested	PW	0.64	N/A

A Δ value below 2 meV indicates a state of the art DFT calculation. *PAW-Suggested* is a combination of the two PAW sets where we have selected the best PAW potential for each element. The performance number given in the Table is an average indication of how much time a calculation of a 64 atom supercell takes, using the different models relative to the FHI-DZP model. Lower numbers are faster. There are large variations among the elements and the performance number should only be used as a rough guide.

Accuracy tests for mixed solids

The Δ-tests are done for ideal systems, in order to illustrate that these results are transferable to general systems we have performed a number of additional accuracy tests. The first test is for a number Rock Salt (RS) and Perovskite(P) structures provided in Ref.[4]. The reference points are FHI-aims all electron calculations. We see the deviations for the a (lattice constant), B (Bulk modulus) and E_f (formation energy) compared to the Ref.[Page 4304, 4] in the below Table.

[4] KevinăF. Garrity, JosephăW. Bennett, KarinăM. Rabe, and David Vanderbilt. Pseudopotentials for high-throughput DFT calculations. *Computational Materials Science*, 81:446–452, 2014. doi:10.1016/j.commatsci.2013.08.053.

Table 4.27: Accuracy of selected methods in QuantumATK on RockSalt (RS) geometries[4]. The formation energy is not available for the Ultra-soft and PAW plane-wave calculations. The reference is a fully converged all-electron calculation using FHI-aims.

Pseudo	Basis	a (%)	B (%)	E_f (%)	Source
Ultra-Soft	PW	0.13	5.0	N/A	Page 4304, 4
PAW	PW	0.15	4.5	N/A	Page 4304, 4
PseudoDojo	Ultra	0.15	1.90	1.83	QuantumATK
PseudoDojo	High	0.18	2.12	2.79	QuantumATK
PseudoDojo	Medium	0.5	5.32	15.59	QuantumATK
PseudoDojo	PW (40 Ha)	0.09	1.86	0.85	QuantumATK
SG15	High	0.3	8.6	4.6	QuantumATK
SG15	Medium	0.6	13.0	12.5	QuantumATK
FHI	DZP	3.0	23.2	15.5	QuantumATK

Table 4.28: Accuracy of selected methods in QuantumATK on Perovskites (P) geometries from Ref.[Page 4304, 4]

Pseudo	Basis	a (%)	B (%)	E_f (%)	Source
Ultra-Soft	PW	0.08	5.5	N/A	Page 4304, 4
PAW	PW	0.13	6.1	N/A	Page 4304, 4
PseudoDojo	Ultra	0.13	1.87	0.91	QuantumATK
PseudoDojo	High	0.21	3.04	1.18	QuantumATK
PseudoDojo	Medium	0.35	3.68	2.73	QuantumATK
PseudoDojo	PW (40 Ha)	0.06	0.66	0.23	QuantumATK
SG15	High	0.3	5.7	1.4	QuantumATK
SG15	Medium	0.4	6.4	2.6	QuantumATK
FHI	DZP	3.5	18.3	12.8	QuantumATK

Notes for each pseudopotential type

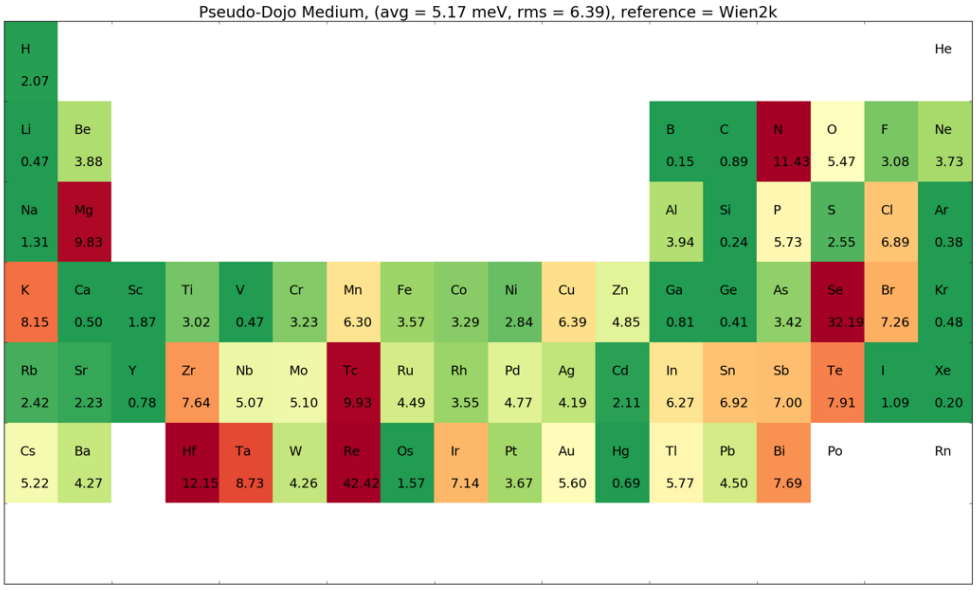
PseudoDojo Pseudopotentials and basis sets

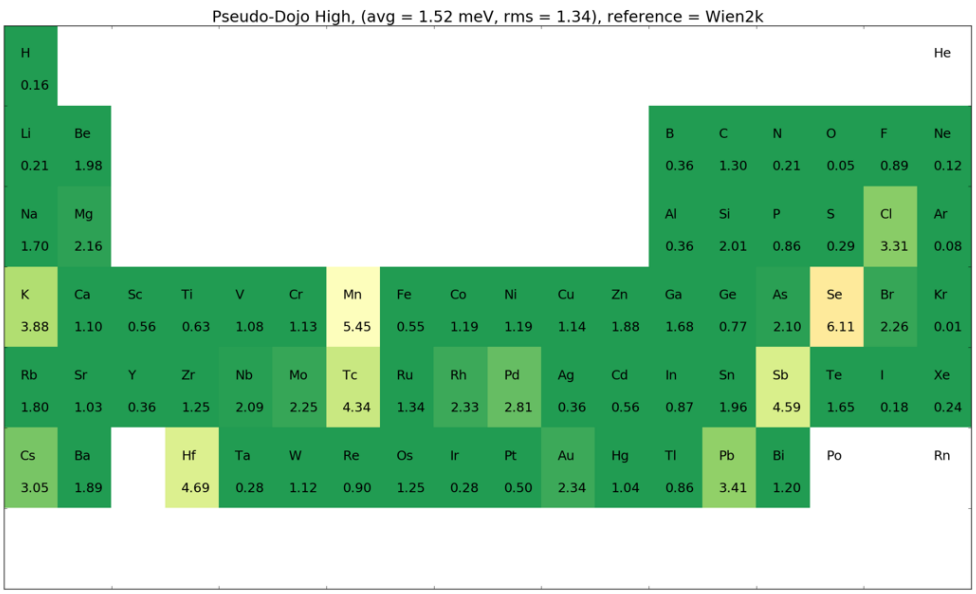
The PseudoDojo are state of the art norm-conserving pseudopotentials including multiple projectors, semi-core states and non-linear core correction[5]. They are only provided for the GGA-PBE functional, and comes in both a scalar relativistic and a fully relativistic version. The latter is used for calculations including spin-orbit. The pseudopotentials are smooth and all elements converge with a mesh cutoff below 125 Ha.

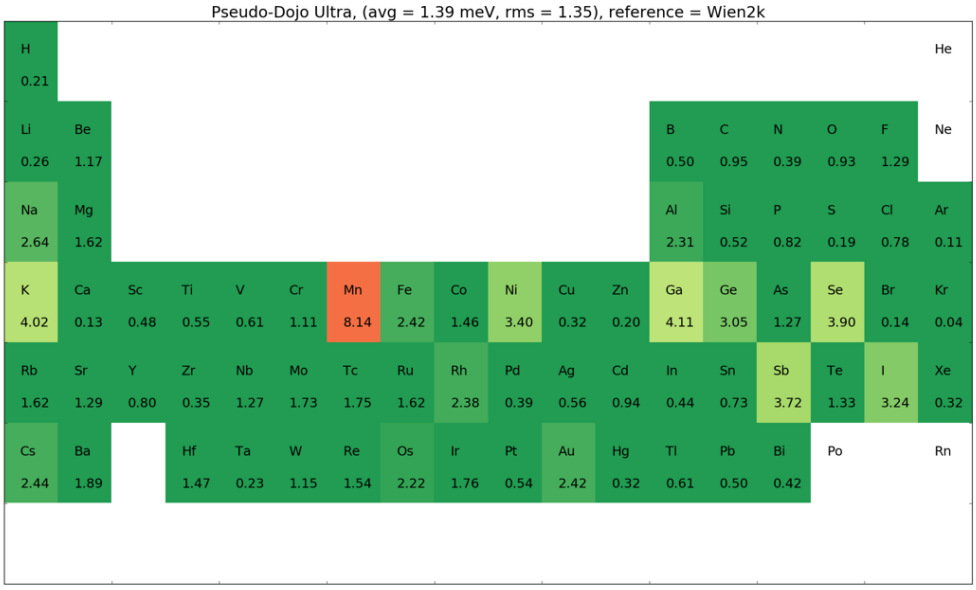
A large effort has been put into providing high accuracy basis sets. Three sets are provided: Ultra, High and Medium. The basis sets have been optimized to describe bulk systems, dimers and trimer systems.

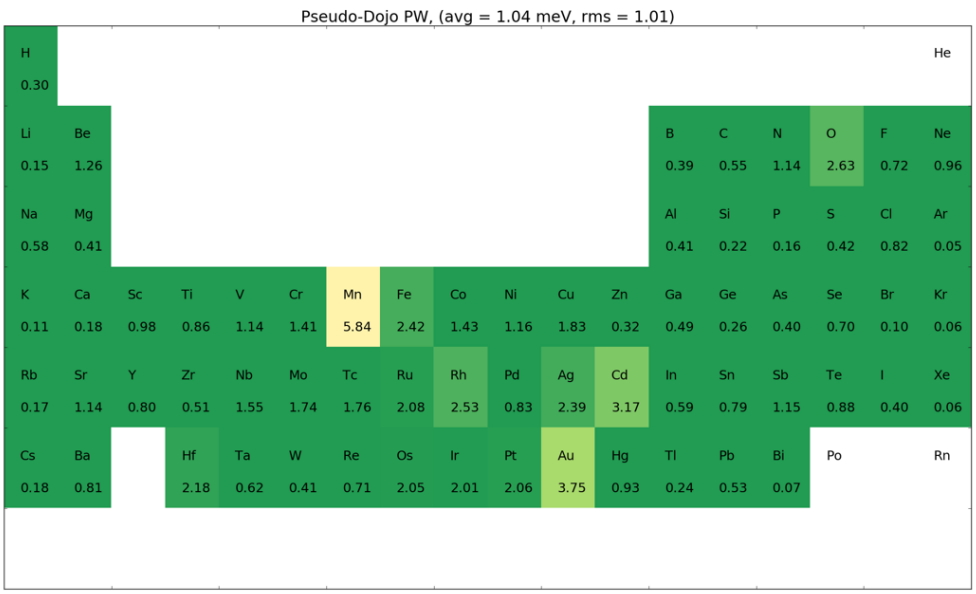
Below are given the Δ-tests for the different elements.

[5] M.J. van Setten, M.ăGiantomassi, E.ăBousquet, M.J. Verstraete, D.R. Hamann, X.ăGonze, and G.-M. Rignanese. The PseudoDojo: Training and grading a 85 element optimized norm-conserving pseudopotential table. *Computer Physics Communications*, 226:39–54, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0010465518300250, doi:https://doi.org/10.1016/j.cpc.2018.01.012.


Pseudo-Dojo Medium, (avg = 5.17 meV, rms = 6.39), reference = Wien2k
H 2.07
He
Li 0.47
Be 3.88
B 0.15
C 0.89
N 11.43
O 5.47
F 3.08
Ne 3.73
Na 1.31
Mg 9.83
Al 3.94
Si 0.24
P 5.73
S 2.55
Cl 6.89
Ar 0.38
K 8.15
Ca 0.50
Sc 1.87
Ti 3.02
V 0.47
Cr 3.23
Mn 6.30
Fe 3.57
Co 3.29
Ni 2.84
Cu 6.39
Zn 4.85
Ga 0.81
Ge 0.41
As 3.42
Se 32.19
Br 7.26
Kr 0.48
Rb 2.42
Sr 2.23
Y 0.78
Zr 7.64
Nb 5.07
Mo 5.10
Tc 9.93
Ru 4.49
Rh 3.55
Pd 4.77
Ag 4.19
Cd 2.11
In 6.27
Sn 6.92
Sb 7.00
Te 7.91
I 1.09
Xe 0.20
Cs 5.22
Ba 4.27
Hf 12.15
Ta 8.73
W 4.26
Re 42.42
Os 1.57
Ir 7.14
Pt 3.67
Au 5.60
Hg 0.69
Tl 5.77
Pb 4.50
Bi 7.69
Po
Rn


Pseudo-Dojo High, (avg = 1.52 meV, rms = 1.34), reference = Wien2k
H 0.16
He
Li 0.21
Be 1.98
B 0.36
C 1.30
N 0.21
O 0.05
F 0.89
Ne 0.12
Na 1.70
Mg 2.16
Al 0.36
Si 2.01
P 0.86
S 0.29
Cl 3.31
Ar 0.08
K 3.88
Ca 1.10
Sc 0.56
Ti 0.63
V 1.08
Cr 1.13
Mn 5.45
Fe 0.55
Co 1.19
Ni 1.19
Cu 1.14
Zn 1.88
Ga 1.68
Ge 0.77
As 2.10
Se 6.11
Br 2.26
Kr 0.01
Rb 1.80
Sr 1.03
Y 0.36
Zr 1.25
Nb 2.09
Mo 2.25
Tc 4.34
Ru 1.34
Rh 2.33
Pd 2.81
Ag 0.36
Cd 0.56
In 0.87
Sn 1.96
Sb 4.59
Te 1.65
I 0.18
Xe 0.24
Cs 3.05
Ba 1.89
Hf 4.69
Ta 0.28
W 1.12
Re 0.90
Os 1.25
Ir 0.28
Pt 0.50
Au 2.34
Hg 1.04
Tl 0.86
Pb 3.41
Bi 1.20
Po
Rn


Pseudo-Dojo Ultra, (avg = 1.39 meV, rms = 1.35), reference = Wien2k
H 0.21
He
Li 0.26
Be 1.17
B 0.50
C 0.95
N 0.39
O 0.93
F 1.29
Ne
Na 2.64
Mg 1.62
Al 2.31
Si 0.52
P 0.82
S 0.19
Cl 0.78
Ar 0.11
K 4.02
Ca 0.13
Sc 0.48
Ti 0.55
V 0.61
Cr 1.11
Mn 8.14
Fe 2.42
Co 1.46
Ni 3.40
Cu 0.32
Zn 0.20
Ga 4.11
Ge 3.05
As 1.27
Se 3.90
Br 0.14
Kr 0.04
Rb 1.62
Sr 1.29
Y 0.80
Zr 0.35
Nb 1.27
Mo 1.73
Tc 1.75
Ru 1.62
Rh 2.38
Pd 0.39
Ag 0.56
Cd 0.94
In 0.44
Sn 0.73
Sb 3.72
Te 1.33
I 3.24
Xe 0.32
Cs 2.44
Ba 1.89
Hf 1.47
Ta 0.23
W 1.15
Re 1.54
Os 2.22
Ir 1.76
Pt 0.54
Au 2.42
Hg 0.32
Tl 0.61
Pb 0.50
Bi 0.42
Po
Rn

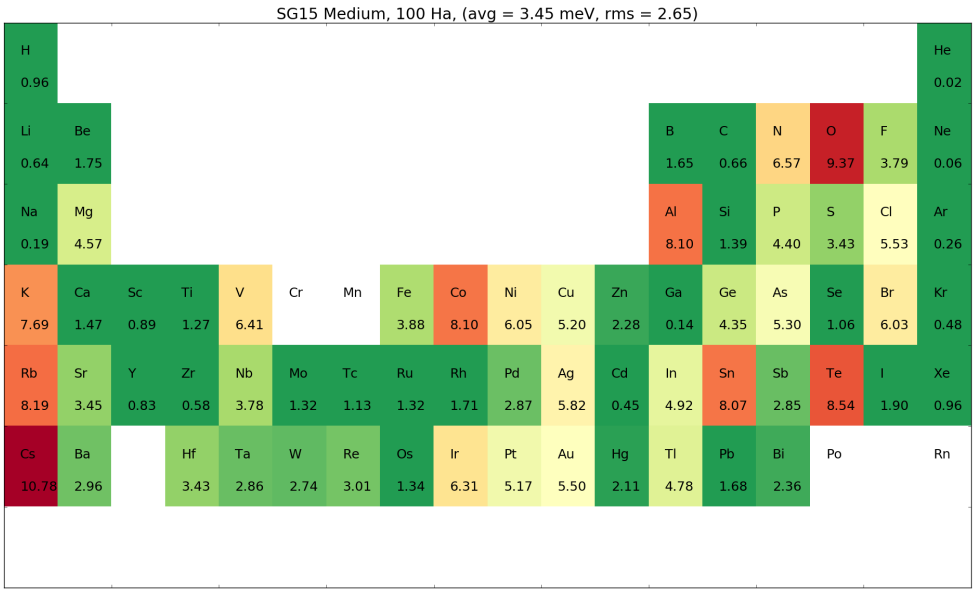

Pseudo-Dojo PW, (avg = 1.04 meV, rms = 1.01)
H 0.30
He
Li 0.15
Be 1.26
B 0.39
C 0.55
N 1.14
O 2.63
F 0.72
Ne 0.96
Na 0.58
Mg 0.41
Al 0.41
Si 0.22
P 0.16
S 0.42
Cl 0.82
Ar 0.05
K 0.11
Ca 0.18
Sc 0.98
Ti 0.86
V 1.14
Cr 1.41
Mn 5.84
Fe 2.42
Co 1.43
Ni 1.16
Cu 1.83
Zn 0.32
Ga 0.49
Ge 0.26
As 0.40
Se 0.70
Br 0.10
Kr 0.06
Rb 0.17
Sr 1.14
Y 0.80
Zr 0.51
Nb 1.55
Mo 1.74
Tc 1.76
Ru 2.08
Rh 2.53
Pd 0.83
Ag 2.39
Cd 3.17
In 0.59
Sn 0.79
Sb 1.15
Te 0.88
I 0.40
Xe 0.06
Cs 0.18
Ba 0.81
Hf 2.18
Ta 0.62
W 0.41
Re 0.71
Os 2.05
Ir 2.01
Pt 2.06
Au 3.75
Hg 0.93
Tl 0.24
Pb 0.53
Bi 0.07
Po
Rn

SG15 Pseudopotentials and basis sets

The SG15 are state of the art norm-conserving pseudopotentials including multiple projectors, semi-core states and non-linear core correction[6][7]. They are only provided for the GGA-PBE functional, and comes in a scalar relativistic and a fully relativistic version. The latter is used for calculations including spin-orbit. The pseudopotentials are smooth and all elements converge with a mesh cutoff below 220 Ha.

A large effort has been put into providing high accuracy basis sets. Three sets are provided: Ultra, High and Medium. The basis sets have been optimized to describe bulk systems, dimers and trimer systems.

Below are given the Δ-tests for the different elements.



PAW data sets

For the Projector Augmented Wave (PAW) method, we need data describing the system inside the augmentation sphere (partial waves, projectors, core densities,). This is gathered in a PAW data set file. We support data set files in the .xml format, as specified on the ESL website. We ship two different data sets, the GPAW and JTH[8] data sets, for which we found default wave function cutoffs that are as low as possible without losing accuracy. We made a combined PAW-Suggested data set for GGA, that gives the best results.

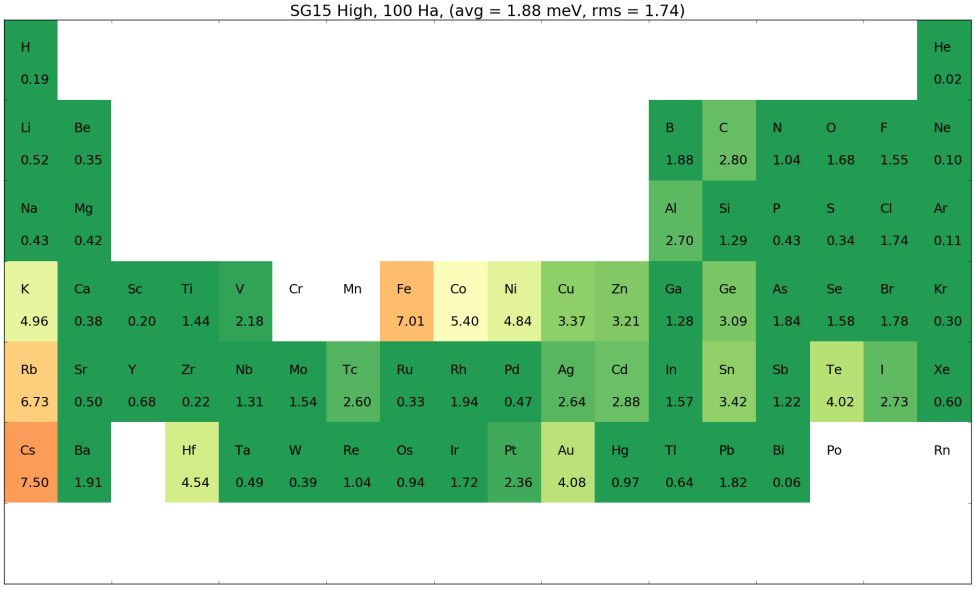
Below are given the Δ-tests for the different elements.

[6] D.ăR. Hamann. Optimized norm-conserving Vanderbilt pseudopotentials. *Phys. Rev. B*, 88:085117, Aug 2013. doi:10.1103/PhysRevB.88.085117.

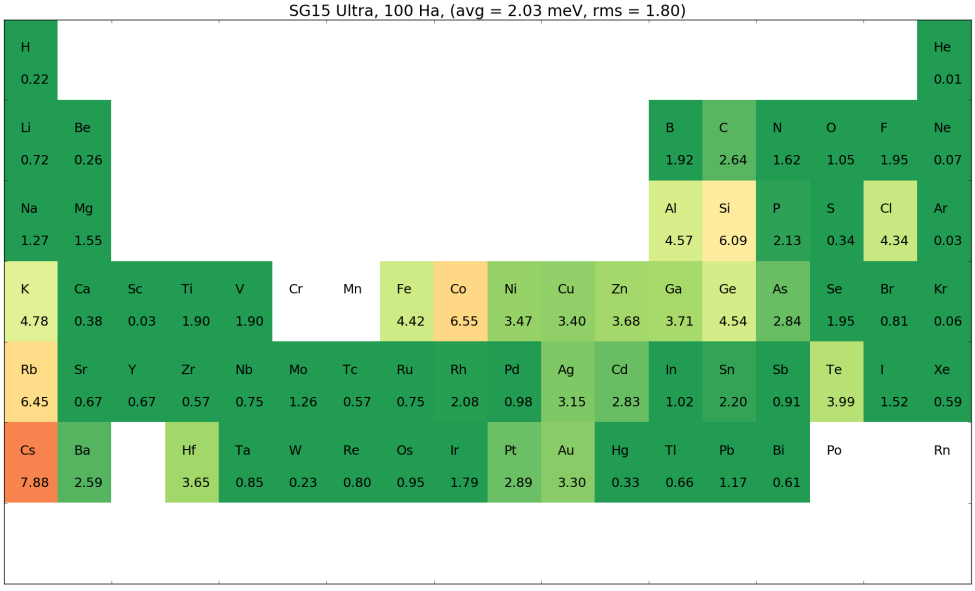
[7] M.ăSchlipf and F.ăGygi. Optimization algorithm for the generation of ONCV pseudopotentials. *Computer Physics Communications*, 196:36–44, 2015. doi:10.1016/j.cpc.2015.05.011.

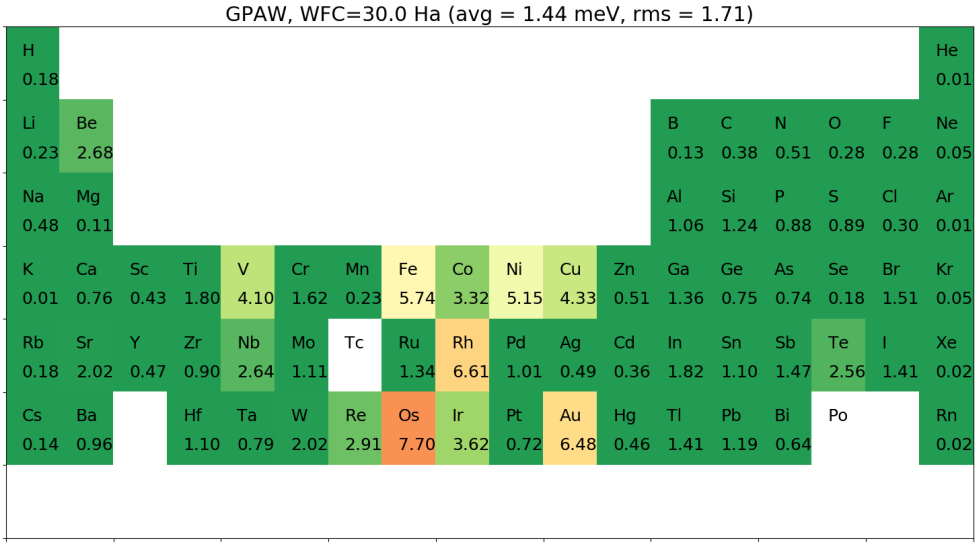
[8] François Jollet, Marc Torrent, and Natalie Holzwarth. Generation of Projector Augmented-Wave atomic data: A 71 element validated table in the XML format. *Computer Physics Communications*, 185(4):1246–1254, 2014. URL: http://www.sciencedirect.com/science/article/pii/S0010465513004359, doi:https://doi.org/10.1016/j.cpc.2013.12.023.

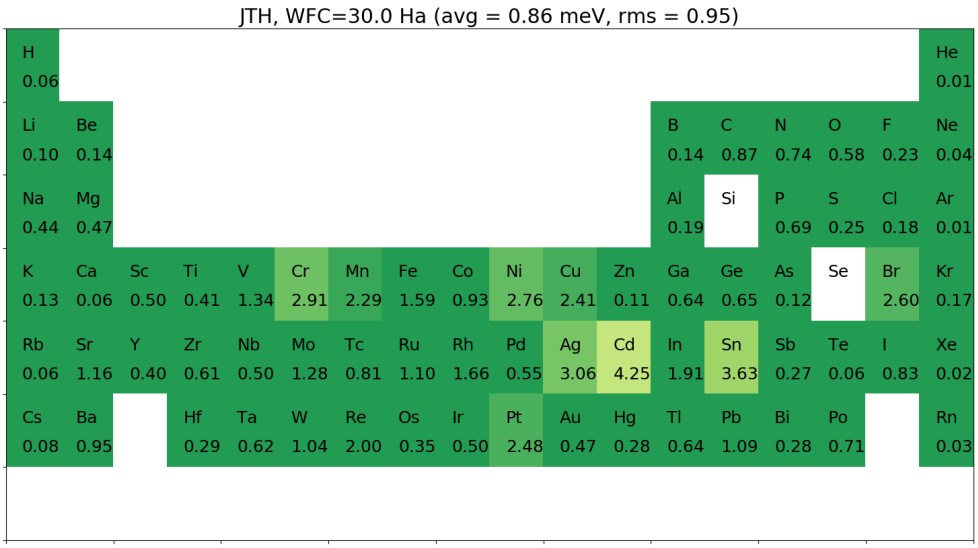
SG15 High, 100 Ha, (avg = 1.88 meV, rms = 1.74)

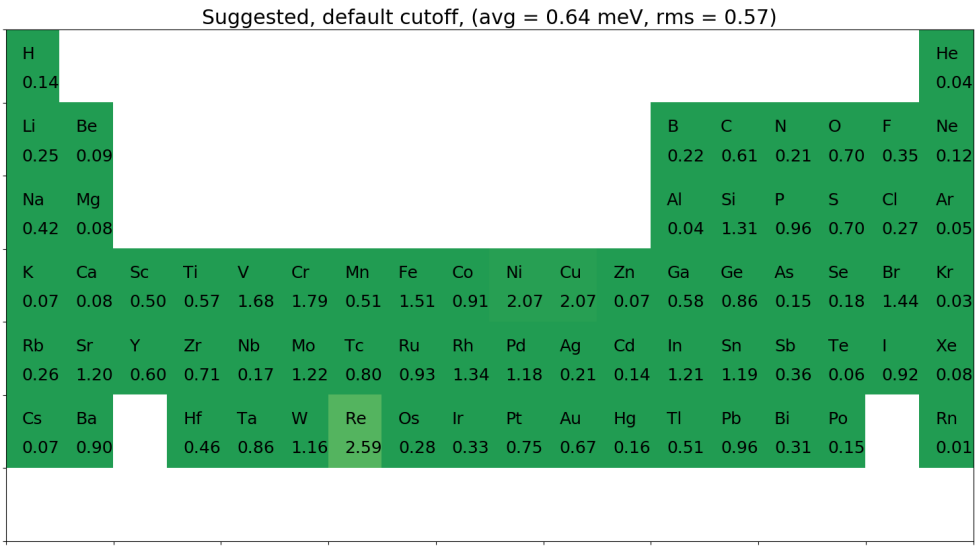


SG15 Ultra, 100 Ha, (avg = 2.03 meV, rms = 1.80)




GPAW, WFC=30.0 Ha (avg = 1.44 meV, rms = 1.71)
H 0.18
He 0.01
Li 0.23
Be 2.68
B 0.13
C 0.38
N 0.51
O 0.28
F 0.28
Ne 0.05
Na 0.48
Mg 0.11
Al 1.06
Si 1.24
P 0.88
S 0.89
Cl 0.30
Ar 0.01
K 0.01
Ca 0.76
Sc 0.43
Ti 1.80
V 4.10
Cr 1.62
Mn 0.23
Fe 5.74
Co 3.32
Ni 5.15
Cu 4.33
Zn 0.51
Ga 1.36
Ge 0.75
As 0.74
Se 0.18
Br 1.51
Kr 0.05
Rb 0.18
Sr 2.02
Y 0.47
Zr 0.90
Nb 2.64
Mo 1.11
Tc
Ru 1.34
Rh 6.61
Pd 1.01
Ag 0.49
Cd 0.36
In 1.82
Sn 1.10
Sb 1.47
Te 2.56
I 1.41
Xe 0.02
Cs 0.14
Ba 0.96
Hf 1.10
Ta 0.79
W 2.02
Re 2.91
Os 7.70
Ir 3.62
Pt 0.72
Au 6.48
Hg 0.46
Tl 1.41
Pb 1.19
Bi 0.64
Po
Rn 0.02


JTH, WFC=30.0 Ha (avg = 0.86 meV, rms = 0.95)
H 0.06
He 0.01
Li 0.10
Be 0.14
B 0.14
C 0.87
N 0.74
O 0.58
F 0.23
Ne 0.04
Na 0.44
Mg 0.47
Al 0.19
Si
P 0.69
S 0.25
Cl 0.18
Ar 0.01
K 0.13
Ca 0.06
Sc 0.50
Ti 0.41
V 1.34
Cr 2.91
Mn 2.29
Fe 1.59
Co 0.93
Ni 2.76
Cu 2.41
Zn 0.11
Ga 0.64
Ge 0.65
As 0.12
Se
Br 2.60
Kr 0.17
Rb 0.06
Sr 1.16
Y 0.40
Zr 0.61
Nb 0.50
Mo 1.28
Tc 0.81
Ru 1.10
Rh 1.66
Pd 0.55
Ag 3.06
Cd 4.25
In 1.91
Sn 3.63
Sb 0.27
Te 0.06
I 0.83
Xe 0.02
Cs 0.08
Ba 0.95
Hf 0.29
Ta 0.62
W 1.04
Re 2.00
Os 0.35
Ir 0.50
Pt 2.48
Au 0.47
Hg 0.28
Tl 0.64
Pb 1.09
Bi 0.28
Po 0.71
Rn 0.03



FHI pseudopotentials and basis sets

The FHI pseudopotentials are the oldest pseudopotentials in QuantumATK. These pseudopotentials are generated with the Fritz-Haber Institute (FHI) pseudopotential code. They are numerical norm-conserving pseudopotentials with a single projector for each angular momentum and without semicore states. For a number of elements the exclusion of semi-core states gives a low accuracy.

The basis sets provided with the FHI pseudopotentials use generic parameters, which are not optimized for each element and generally not of high accuracy. However, the basis sets and pseudopotentials are rather efficient and for some elements can give satisfactory accuracy. We only recommend the SZP and DZP basis sets. Below are given Δ values for these settings and the table can be used to estimate the accuracy of the model for various elements.

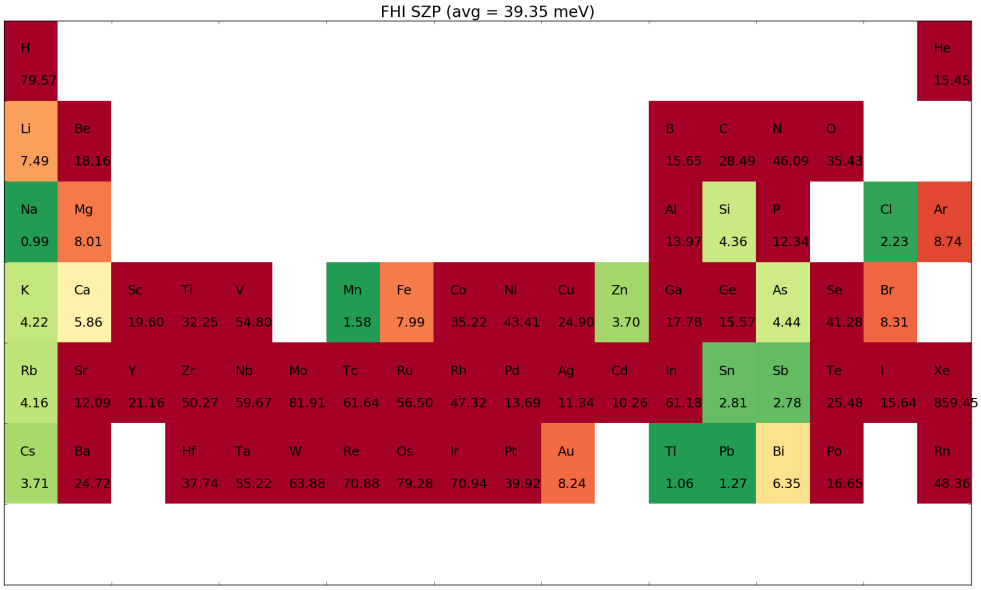
OMX Pseudopotentials and basis sets

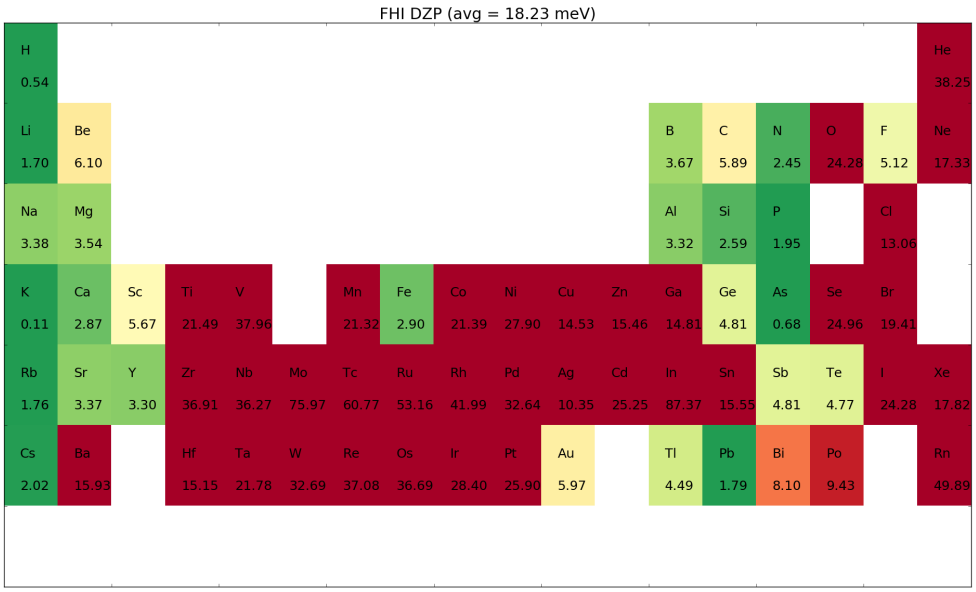
The OMX pseudopotentials are from the OpenMX packages. They are fully relativistic, multiple projectors and with semi-core states. Thus, these are generally high accuracy pseudopotentials which can be used for both LDA, GGA and spin-orbit calculations. The drawback of the pseudopotentials is that they for some elements require very high mesh cutoffs.

The basis sets provided are from the original openMX package. This means that they are not calculated on the fly, and therefore only accurate for the exchange-correlation potential for which they were calculated. Thus, if other exchange-correlation functionals than LDA-PZ or GGA-PBE are used, the accuracy will be lower. For our other pseudopotentials the basis sets are generated on the fly with the selected exchange-correlation potential, thus, even if the pseudopotential is not generated for the exchange-correlation functional the basis set will conform to the exchange-correlation functional.

For the OMX pseudopotentials, we have defined two basis set types, Medium and High. It is also possible for the user to define custom basis sets through the `atomic_species` keyword, see *`OpenMXBasisSet`*.

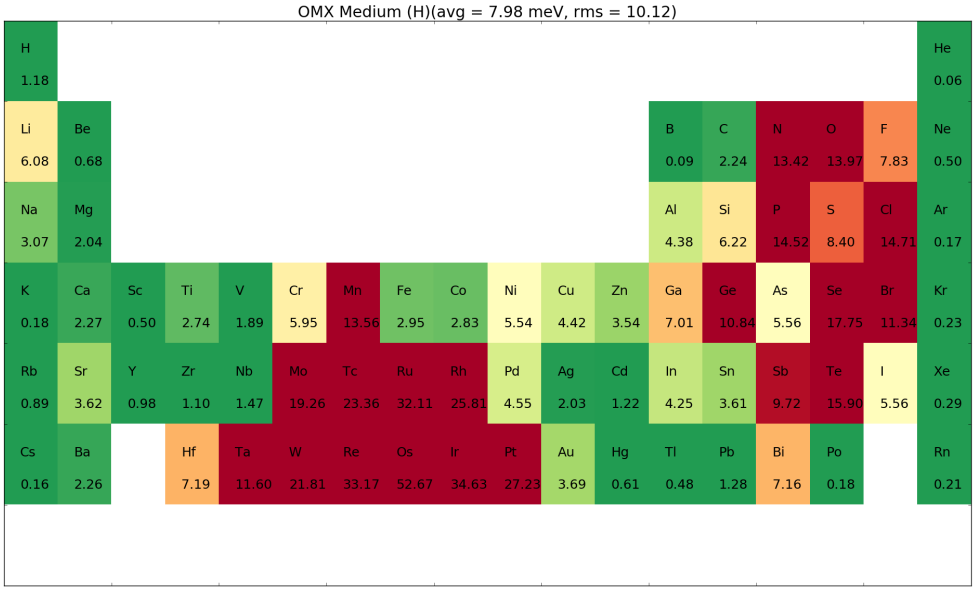
The basis sets was pre-calculated using GGA-PBE or LDA-PZ and included together with the pseudopotential. Thus, if other exchange-correlation functionals than LDA-PZ or GGA-PBE are used, the accuracy will be low.


FHI SZP (avg = 39.35 meV)
H 79.57
He 15.45
Li 7.49
Be 18.16
B 15.65
C 28.49
N 46.09
O 35.43
Na 0.99
Mg 8.01
Al 13.97
Si 4.36
P 12.34
Cl 2.23
Ar 8.74
K 4.22
Ca 5.86
Sc 19.60
Ti 32.25
V 54.80
Mn 1.58
Fe 7.99
Co 35.22
Ni 43.41
Cu 24.90
Zn 3.70
Ga 17.78
Ge 15.57
As 4.44
Se 41.28
Br 8.31
Rb 4.16
Sr 12.09
Y 21.16
Zr 50.27
Nb 59.67
Mo 81.91
Tc 61.64
Ru 56.50
Rh 47.32
Pd 13.69
Ag 11.34
Cd 10.26
In 61.18
Sn 2.81
Sb 2.78
Te 25.48
I 15.64
Xe 859.45
Cs 3.71
Ba 24.72
Hf 37.74
Ta 55.22
W 63.88
Re 70.88
Os 79.28
Ir 70.94
Pt 39.92
Au 8.24
Tl 1.06
Pb 1.27
Bi 6.35
Po 16.65
Rn 48.36


FHI DZP (avg = 18.23 meV)
H 0.54
He 38.25
Li 1.70
Be 6.10
B 3.67
C 5.89
N 2.45
O 24.28
F 5.12
Ne 17.33
Na 3.38
Mg 3.54
Al 3.32
Si 2.59
P 1.95
Cl 13.06
K 0.11
Ca 2.87
Sc 5.67
Ti 21.49
V 37.96
Mn 21.32
Fe 2.90
Co 21.39
Ni 27.90
Cu 14.53
Zn 15.46
Ga 14.81
Ge 4.81
As 0.68
Se 24.96
Br 19.41
Rb 1.76
Sr 3.37
Y 3.30
Zr 36.91
Nb 36.27
Mo 75.97
Tc 60.77
Ru 53.16
Rh 41.99
Pd 32.64
Ag 10.35
Cd 25.25
In 87.37
Sn 15.55
Sb 4.81
Te 4.77
I 24.28
Xe 17.82
Cs 2.02
Ba 15.93
Hf 15.15
Ta 21.78
W 32.69
Re 37.08
Os 36.69
Ir 28.40
Pt 25.90
Au 5.97
Tl 4.49
Pb 1.79
Bi 8.10
Po 9.43
Rn 49.89

For the other pseudo potentials the basis sets are calculated on the fly, using the selected exchange-correlation potential. i.e. only core electrons are treated with the GGA-PBE functional, while valence electrons are treated with the selected exchange-correlation functional.

Below we provide the Δ values for the Medium and High basis sets. For the High basis set we also provide the Δ value when a 100 Ha mesh cutoff is used. This, gives an indication for which elements the default mesh cutoff needs to be increased.

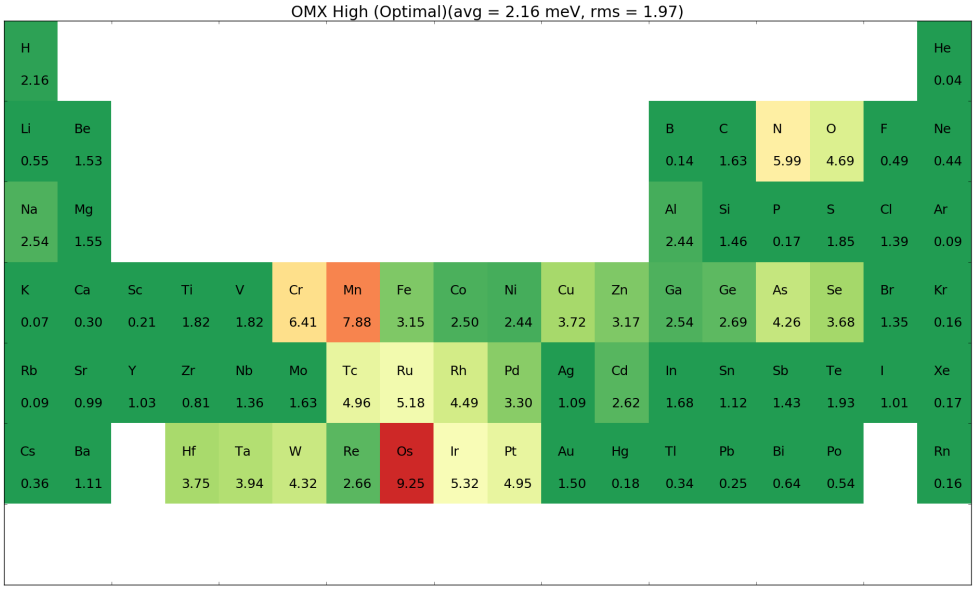


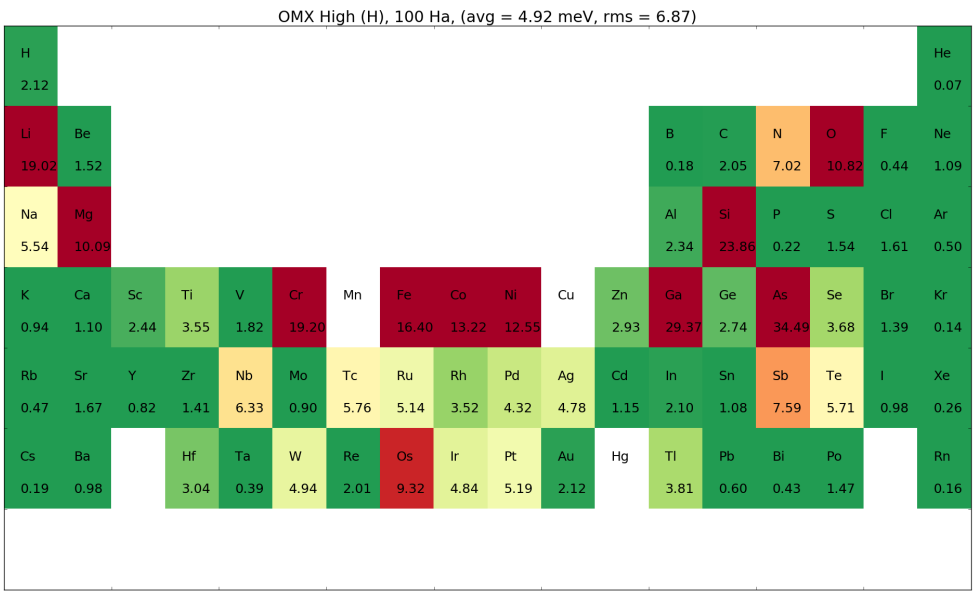
HGH Pseudopotentials and basis sets

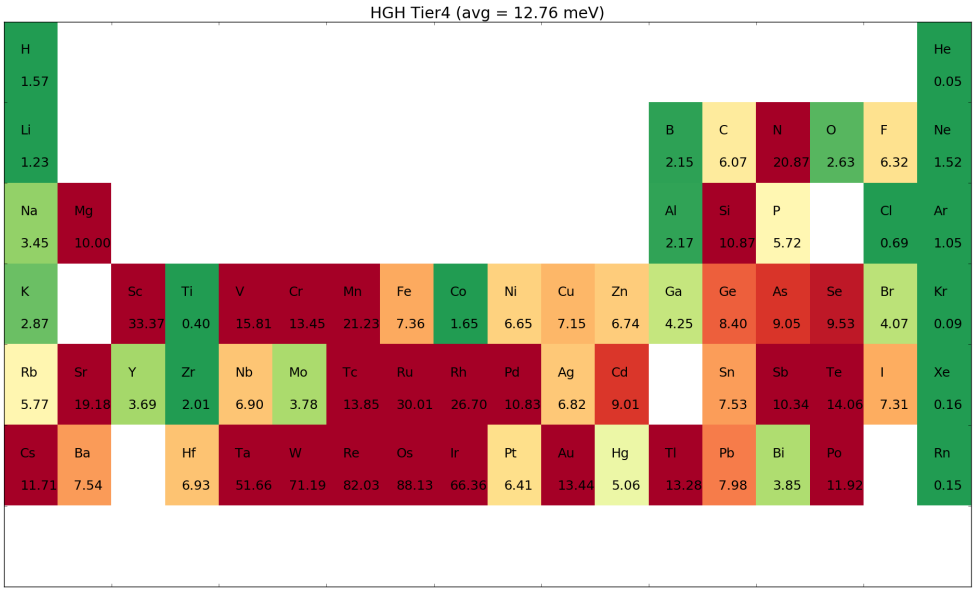
The Hartwigsen–Goedecker–Hutter (HGH) type pseudopotentials are also norm-conserving pseudopotentials, however, include multiple projectors and semi-core states. The pseudopotentials use analytical functions which are slightly lower accuracy than numerical pseudopotentials. These pseudopotentials require high values for the mesh cutoff for some elements.

For each pseudopotential we provide a hierarchy of basis sets. The basis sets were optimized to describe the total energy of dimers with different bondlengths. This was an early version of our basis set optimization tool and not all elements are well described.

Generally the pseudopotentials and basis sets have better accuracy than the FHI pseudopotentials but can be computationally heavy and require high mesh cutoffs. The sets are only provided for backwards compatibility and are no longer recommended.


OMX High (Optimal)(avg = 2.16 meV, rms = 1.97)
H 2.16
He 0.04
Li 0.55
Be 1.53
B 0.14
C 1.63
N 5.99
O 4.69
F 0.49
Ne 0.44
Na 2.54
Mg 1.55
Al 2.44
Si 1.46
P 0.17
S 1.85
Cl 1.39
Ar 0.09
K 0.07
Ca 0.30
Sc 0.21
Ti 1.82
V 1.82
Cr 6.41
Mn 7.88
Fe 3.15
Co 2.50
Ni 2.44
Cu 3.72
Zn 3.17
Ga 2.54
Ge 2.69
As 4.26
Se 3.68
Br 1.35
Kr 0.16
Rb 0.09
Sr 0.99
Y 1.03
Zr 0.81
Nb 1.36
Mo 1.63
Tc 4.96
Ru 5.18
Rh 4.49
Pd 3.30
Ag 1.09
Cd 2.62
In 1.68
Sn 1.12
Sb 1.43
Te 1.93
I 1.01
Xe 0.17
Cs 0.36
Ba 1.11
Hf 3.75
Ta 3.94
W 4.32
Re 2.66
Os 9.25
Ir 5.32
Pt 4.95
Au 1.50
Hg 0.18
Tl 0.34
Pb 0.25
Bi 0.64
Po 0.54
Rn 0.16


OMX High (H), 100 Ha, (avg = 4.92 meV, rms = 6.87)
H 2.12
He 0.07
Li 19.02
Be 1.52
B 0.18
C 2.05
N 7.02
O 10.82
F 0.44
Ne 1.09
Na 5.54
Mg 10.09
Al 2.34
Si 23.86
P 0.22
S 1.54
Cl 1.61
Ar 0.50
K 0.94
Ca 1.10
Sc 2.44
Ti 3.55
V 1.82
Cr 19.20
Mn
Fe 16.40
Co 13.22
Ni 12.55
Cu
Zn 2.93
Ga 29.37
Ge 2.74
As 34.49
Se 3.68
Br 1.39
Kr 0.14
Rb 0.47
Sr 1.67
Y 0.82
Zr 1.41
Nb 6.33
Mo 0.90
Tc 5.76
Ru 5.14
Rh 3.52
Pd 4.32
Ag 4.78
Cd 1.15
In 2.10
Sn 1.08
Sb 7.59
Te 5.71
I 0.98
Xe 0.26
Cs 0.19
Ba 0.98
Hf 3.04
Ta 0.39
W 4.94
Re 2.01
Os 9.32
Ir 4.84
Pt 5.19
Au 2.12
Hg
Tl 3.81
Pb 0.60
Bi 0.43
Po 1.47
Rn 0.16



References

4.5.4 Doping methods available in QuantumATK

The effect of electrostatic doping is generally modelled by either increasing (n-doping) or decreasing (p-doping) the number of electrons of the system with respect to the neutral case. QuantumATK offers two general methods to introduce electrostatic doping. Both lead to the same additional electrons (holes) introduced in the system, but differ in the way these extra electrons (holes) are redistributed within it.

General background

In QuantumATK the *ElectronDensity* of the system is expressed as a sum of two contributions:

$$\rho(\mathbf{r}) = \Delta\rho(\mathbf{r}) + \sum_{i}^{N_{atoms}} \rho_i^{atom}(\mathbf{r}).$$

$\Delta\rho(\mathbf{r})$ is the *ElectronDifferenceDensity*, which for a neutral system integrates to zero:

$$\int \Delta\rho(\mathbf{r})\, dr = 0.$$

$\sum_i^{N_{atoms}} \rho_i^{atom}(\mathbf{r})$ is the sum of the densities associated with the individual atoms. For a neutral system, this sum integrates to the total number of electrons N:

$$\sum_{i}^{N_{atoms}} \int \rho_i^{atom}(\mathbf{r})\, dr = N.$$

It follows that the number of electrons in a generic system with charge Q, which is the integral over the electron density in real space, can be also expressed as:

$$\int \rho(\mathbf{r})\, dr = \tilde{N} + \sum_{i}^{N_{atoms}} \int \rho_i^{atom}(\mathbf{r})\, dr$$

where $\tilde{N}$ are the extra electrons leading to a charge $Q = -e \cdot \tilde{N}$. It follows that

$$\int \Delta\rho(\mathbf{r})\, dr = \tilde{N}.$$

In QuantumATK, the system can be doped in two different ways, either by adding an *Explicit charge* or by adding *Atomic compensation charges*. The former modifies the value of $\tilde{N}$, whereas the latter modifies the value of N. However, both lead to the following result:

$$-e \int \Delta\rho(\mathbf{r})\, dr = Q.$$

Explicit charge

An explicit charge can be added by setting the `charge` parameter to a value Q in the *LCAOCalculator*. In this case, $\tilde{N}$ is set to a non-zero value $\tilde{M} = -Q/e$. The total number of electrons in the system is then determined by

$$\int \rho(\mathbf{r})\, dr = \tilde{N} + \sum_{i}^{N_{atoms}} \int \rho_i^{atom}(\mathbf{r})\, dr = \tilde{M} + N.$$

Note: There are two important consequences of using this method:

- The densities of the individual atoms $\rho_i^{atom}(\mathbf{r})$ are those associated with the individual neutral species;
- The extra charge is not bound to any atom. Therefore, it will redistribute in order to minimize the total energy of the system.

Atomic compensation charges

Using the *AtomicCompensationCharge* method, $\tilde{N}$ is set to zero, but the density of the *i*-th atom is rescaled by a factor:

$$c_i = \frac{\tilde{n}_i + n_i}{n_i}.$$

where n_i and $\tilde{n}_i$ are the number of valence electrons and the extra compensation charge associated with the *i*-th isolated atom. The individual values of $\tilde{n}_i$, and therefore the individual coefficients c_i, can be set arbitrarily from atom to atom, even within the same chemical species. For sake of comparison, we assume here that they are all the same and that they sum up to $\tilde{M}$:

$$\sum_{i}^{N_{atoms}} \tilde{n}_i = \tilde{M}.$$

In this case, the integral of the total density can be expressed as a sum over the individual densities

$$\int \rho(\mathbf{r})\, dr = \sum_{i}^{N_{atoms}} \int \tilde{\rho}_i^{atom}(\mathbf{r})\, dr$$

where

$$\tilde{\rho}_i^{atom}(\mathbf{r}) = c_i \cdot \rho_i^{atom}(\mathbf{r})$$

Due to the use of the rescaled atomic densities $\tilde{\rho}_i^{atom}(\mathbf{r})$, the electronic density of the system will be the same as that obtained using an explicit doping:

$$\sum_i^{N_{atoms}} \int \tilde{\rho}_i^{atom}(\mathbf{r})\, dr = \sum_i^{N_{atoms}} \int \frac{\tilde{n}_i}{n_i} \rho_i^{atom}(\mathbf{r})\, dr + \sum_i^{N_{atoms}} \int \rho_i^{atom}(\mathbf{r})\, dr = \tilde{N} + N$$

Note: Using this method, the extra charge is introduced by modifying the densities of the individual atoms. As a consequence:

- The extra electron density will be bound to those atoms for which $n_i \neq 0$;
- The electrostatic potential will be locally modified in the vicinity of such charged atoms, becoming more attractive (repulsive) if $n_i > 0$ ($n_i < 0$).

4.5.5 Optical response functions

The optical response functions couple an external electric field, E_{ext}, with the internal electric field arising from the response of the crystal. It is convenient to introduce the displacement field, $\mathbf{D}$, which determines the electric field from external charges, ρ_{ext}. The displacement field is related to the internal electric field, $\mathbf{E}$, through the polarization, $\mathbf{P}$:

$$\mathbf{D} = \epsilon_\mathbf{0}\mathbf{E} + \mathbf{P},$$

where $-\mathbf{P}$ is the electric field from the internal charges, ρ_{int}.

The polarization is related to the dipole moment of the material, $\mathbf{p}$,

$$\mathbf{P} \equiv \mathbf{p}/V,$$

where V is the volume of the material.

Linear response coefficients

The linear response coefficients χ (susceptibility), ϵ_r (dielectric constant), and α (polarizability) relate the electrodynamic quantities outlined above to each other:

$$\begin{aligned} \mathbf{P} &= \epsilon_0 \chi \mathbf{E}, \\ \mathbf{D} &= \epsilon_r \epsilon_0 \mathbf{E}, \\ \mathbf{p} &= \alpha \mathbf{E}. \end{aligned}$$

Optical conductivity

For a perturbation $\mathbf{E}(\mathbf{r}) = \mathbf{E}_0 \exp(\mathrm{i}\mathbf{q}\cdot\mathbf{r})$, the linear response current in the long wave-length limit ($q \ll 1/a$, where a is the lattice constant) is given by[1]:

$$\mathbf{j}(\mathbf{r},\omega) = \sigma(q=0,\omega)\mathbf{E}(\mathbf{r},\omega),$$

where σ is the optical conductivity.

[1] N.ăWiser. Dielectric Constant with Local Field Effects Included. *Phys. Rev.*, 129:62–69, Jan 1963. doi:10.1103/PhysRev.129.62.

Units

The unit of the linear response coefficients are:

Coefficient	Unit
α	C^2/Nm^5
ϵ_r	1
χ	1
σ	C^2/Nsm^2

Relation between the linear response coefficients

All the response coefficients follow from the susceptibility, χ:

$$\begin{aligned}\epsilon_r(\omega) &= (1+\chi(\omega)),\\ \alpha(\omega) &= V\epsilon_0\chi(\omega),\\ \sigma(\omega) &= -i\omega\epsilon_0\chi(\omega).\end{aligned}$$

The derivation of the last relation can be found in[2].

4.5.6 Hybrid Functionals

Hybrid functionals are exchange-correlation functionals that take into account exact Fock exchange[1]. There are different approaches as to how to mix in the exact exchange. The method we implemented in our `PlaneWaveCalculator` code is the HSE functional described in[23].

Background

The HSE functional is an adaptation of the PBE0 functional[4], where the exchange-correlation energy is defined as:

$$E_{xc}^{\text{PBE0}} = \frac{1}{4}E_x^{\text{HF}} + \frac{3}{4}E_x^{\text{PBE}} + E_c^{\text{PBE}}\ .$$

The exact-exchange energy E_x^{HF} is defined in terms of the Kohn-Sham orbitals as:

$$E_x^{\text{HF}} = -\frac{e^2}{2}\sum_{\mathbf{k}n}\sum_{\mathbf{k}'n'} w_{\mathbf{k}}f_{\mathbf{k}n}w_{\mathbf{k}}f_{\mathbf{k}'n'}\iint \frac{\psi^*_{\mathbf{k}n}(\mathbf{r})\psi_{\mathbf{k}'n'}(\mathbf{r})\psi^*_{\mathbf{k}'n'}(\mathbf{r}')\psi_{\mathbf{k}n}(\mathbf{r}')}{|\mathbf{r}'-\mathbf{r}|}d\mathbf{r}d\mathbf{r}', \tag{4.8}$$

and is very expensive to evaluate.

[2] RichardăM. Martin. *Electronic structure: Basic theory and practical methods*. Cambridge University Press, New York, 2004. ISBN 0-521-78285-6.

[1] AxelăD. Becke. A new mixing of Hartree–Fock and local densityfunctional theories. *The Journal of Chemical Physics*, 98(2):1372–1377, 1993. URL: https://doi.org/10.1063/1.464304, doi:10.1063/1.464304.

[2] Jochen Heyd, GustavoăE. Scuseria, and Matthias Ernzerhof. Hybrid functionals based on a screened Coulomb potential. *The Journal of Chemical Physics*, 118(18):8207–8215, 2003. URL: https://doi.org/10.1063/1.1564060, doi:10.1063/1.1564060.

[3] Jochen Heyd, GustavoăE. Scuseria, and Matthias Ernzerhof. Erratum: Hybrid functionals based on a screened Coulomb potential [J. Chem. Phys. 118, 8207 (2003)]. *The Journal of Chemical Physics*, 124(21):219906, 2006. URL: https://doi.org/10.1063/1.2204597, arXiv:https://doi.org/10.1063/1.2204597, doi:10.1063/1.2204597.

[4] JohnăP. Perdew, Matthias Ernzerhof, and Kieron Burke. Rationale for mixing exact exchange with density functional approximations. *The Journal of Chemical Physics*, 105(22):9982–9985, 1996. URL: https://doi.org/10.1063/1.472933, doi:10.1063/1.472933.

For periodic systems, the exact-exchange energy converges very slowly with the distance. For this reason Heyd et al.[Page 4318, 2] suggested splitting up the exchange in a long- and short-range part (defined by a screening length parameter μ). In which only for the short range part exact exchange is mixed:

$$E_{xc}^{\mathrm{HSE}} = \frac{1}{4}E_x^{\mathrm{HF,sr},\mu} + \frac{3}{4}E_x^{\mathrm{PBE,sr},\mu} + E_x^{\mathrm{PBE,lr},\mu} + E_c^{\mathrm{PBE}} \, .$$

The most general version of this functional depends on two parameters, the screening length μ, and the mixing fraction α, which was $\frac{1}{4}$ in the PBE0 functional:

$$E_{xc}^{\mathrm{HSE}}(\mu,\alpha) = \alpha E_x^{\mathrm{HF,sr},\mu} + (1-\alpha)E_x^{\mathrm{PBE,sr},\mu} + E_x^{\mathrm{PBE,lr},\mu} + E_c^{\mathrm{PBE}} \, .$$

ACE Implementation

To include the exact exchange energy in the total energy functional in a DFT calculation one needs to add a non-local orbital dependent exact exchange potential, $V_x^{\mathrm{HF}}[\psi_{\mathbf{k}n}(\mathbf{r})] = \frac{\delta E_x^{\mathrm{HF}}}{\delta \psi_{\mathbf{k}n}^*(\mathbf{r})}$ to the Hamiltonian when solving the Kohn-Sham equations. Even though only action of the exchange potential operator is needed when iteratively solving for eigensolutions, the evaluation of the operator is still very expensive compared to the other parts of a DFT calculation, and it has to be carried out several times in the iterative process.

However, using the Adaptively Compressed Exchange operator method[5] we avoid the expensive re-evaluations of the exchange operator in the subsequent steps in the iterative eigensolver. In this approach the exact-exchange operator $\hat{V}_x^{\mathrm{HF}}$ is applied once to the current best estimate for the ground state wave-functions, $|\psi_i\rangle$,

$$|W_i\rangle = \hat{V}_x^{\mathrm{HF}}|\psi_i\rangle, \qquad \text{for} \qquad 1 < i \leq N.$$

From these W states, a set of so-called ACE projectors $|p_i\rangle$ are constructed. The projection operator:

$$\hat{V}_x^{\mathrm{ACE}} = \sum_i |p_i\rangle\langle p_i| \, ,$$

has the same action as $\hat{V}_x^{\mathrm{HF}}$ on states in the subspace spanned by the wave functions and can be proven to a good approximation for states that are close to the subspace. The advantage is that $\hat{V}_x^{\mathrm{ACE}}$ is much cheaper to apply (same computational complexity as the pseudopotential) in the following uses of the Hamiltonian. Note, however, that to construct the ACE projectors, one needs to fully evaluate the action of the full exact exchange operator, $\hat{V}_x^{\mathrm{HF}}$, a process which is still very expensive and scales as

$$T \propto N_{\tilde{\mathbf{k}}} \times N_{\mathbf{k}} \times N_{\mathrm{bands}}^2 \times N_{\mathbf{G}} \log(N_{\mathbf{G}}),$$

where $N_{\tilde{\mathbf{k}}}$ is the number of symmetry-reduced $\mathbf{k}$-points, $N_{\mathbf{k}}$ is total number of unreduced $\mathbf{k}$- points, N_{band} is the number of occupied bands and $N_{\mathbf{G}}$ is the number of plane wave basis functions.

Instead of only using the ACE operator for a single SCF iteration it has proven more efficient to keep the operator fixed and do a full self-consistent cycle before updating the ACE projectors from the most recent Kohn-Sham wave functions.

In QuantumATK ACE hybrid functional calculations therefore employs a double self-consistent loop algorithm, see Fig. 4.64. Starting from a reasonable initial guess for the wave functions $|\psi_i\rangle$, we construct the ACE projectors and perform a regular DFT SCF loop in which the ACE operators are added to the Hamiltonian and are kept constant. The resulting wave functions are then used to update the ACE projectors and a new SCF loop is started. In this way the ACE operator adaptively improves every outer iteration and the process is continued until the exact exchange energy of the system is converged, in which case the ACE operator will be identical to the real exact exchange operator[6]. The tolerance on the exchange energy and the number of outer scf loops taken can be set on `IterationControlParameters`.

[5] Lin Lin. Adaptively Compressed Exchange Operator. *Journal of Chemical Theory and Computation*, 12(5):2242–2249, 2016. PMID: 27045571. URL: http://dx.doi.org/10.1021/acs.jctc.6b00092, doi:10.1021/acs.jctc.6b00092.

[6] Lin Lin and Michael Lindsey. Convergence of adaptive compression methods for Hartree-Fock-like equations. *arXiv:1703.05441 [math]*, mar 2017. arXiv: 1703.05441. URL: http://arxiv.org/abs/1703.05441 (visited on 2018-02-02).

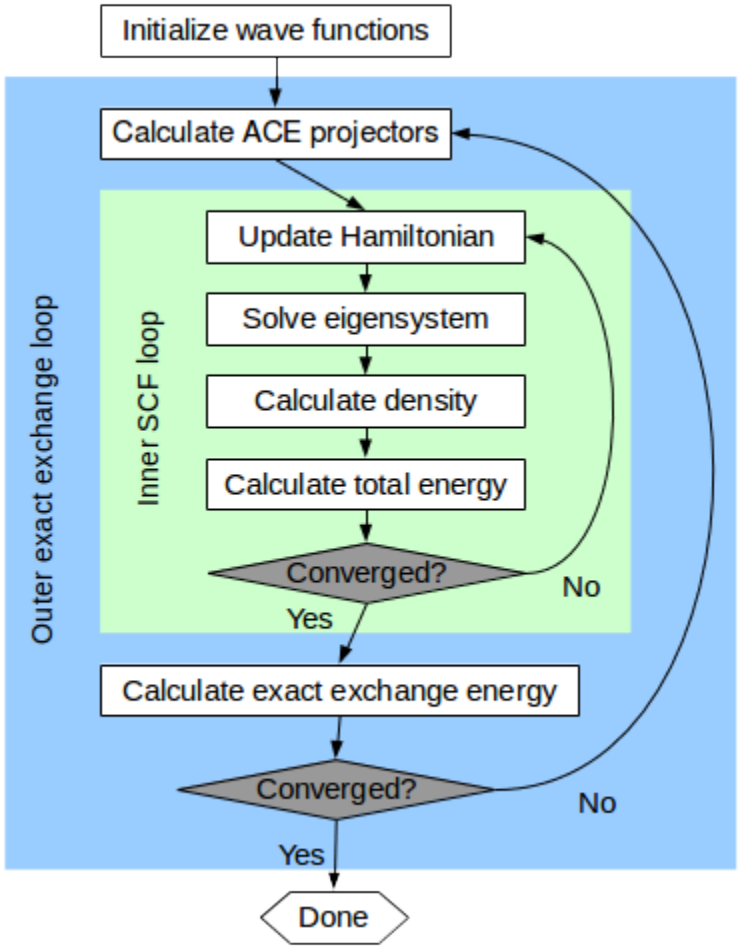


Fig. 4.64: A flowchart diagram showing the double self-consistency loop structure used in ACE hybrid functional calculations.

Usage Examples

Running a calculation with HSE is as simple as setting the correct `exchange_correlation` on the *PlaneWaveCalculator*. By choosing:

```
exchange_correlation = HybridGGA.HSE06
calculator = PlaneWaveCalculator(exchange_correlation=exchange_correlation)
```

the standard `HSE06` parameters are chosen, i.e. `screening_length=0.11*Bohr**-1` and `exx_fraction=0.25`.

One can set custom values for these parameters by choosing:

```
exchange_correlation = HybridGGA.HSECustom(screening_length=0.14*Bohr**-1, exx_
↪fraction=0.20)
calculator = PlaneWaveCalculator(exchange_correlation=exchange_correlation)
```

By default the first SCF loop will have ACE projectors that are set to zero. In a lot of cases it makes sense to start from a PBE calculation, this can be done as follows:

```
calculator = PlaneWaveCalculator(exchange_correlation=GGA.PBE)
configuration.setCalculator(calculator)
configuration.update()

new_calculator = calculator()(exchange_correlation=HybridGGA.HSE06)
configuration.setCalculator(new_calculator, initial_state=configuration)
configuration.update()
```

When restarting, the wave functions of the old configuration will be used to construct ACE projectors, and an exact exchange contribution will already be present in the first SCF loop.

References

4.5.7 NEGF Convergence Guide

Version: 2018

Downloads & Links

Transport calculations with QuantumATK ATK Reference Manual

QuantumATK delivers a market-leading and highly optimized implementation of the non-equilibrium Greens function (NEGF) method for electron transport calculations. Even so, achieving fast and reliable convergence of the NEGF calculation to obtain the electronic ground state can sometimes be challenging. This guide therefore provides a list of **action points** for troubleshooting poorly converging NEGF calculations.

What does it mean that my calculation did not converge?

Just like in standard calculations for periodic bulks, the NEGF device calculator uses the self-consistent field (SCF) method to iteratively search for the electronic ground state. In each step of the SCF cycle, the electronic structure problem is solved and the total energy is computed. The calculation is converged when the energy change between SCF iterations falls below some tolerance, that is, when the self-consistent electron density is found.

If the calculation does not converge, it means that the self-consistent solution to the electronic structure problem (the ground state) was not found within the allowed number of SCF iterations. Your task is then to modify the NEGF calculation such that it will converge, before moving on to analysis.

Tip: QuantumATK issues a warning if the calculation did not converge The QuantumATK calculator log will contain the following warning message whenever a calculation did not properly converge:

```
##############################################################################
#                                                                            #
# Warning: The calculation did not converge to the requested tolerance!      #
#                                                                            #
##############################################################################
```

Checking the QuantumATK log for such warnings is an essential habit, as the results are probably not useful if the calculation was not converged. The log stream is sent to the system `stdout` and/or to a file, depending on how the job is executed. From the O-2018.06 release, the log-file will also contain a **Convergence Report** with detailed information, if the calculation did not converge.

Introduction

As outlined above, it is important that the device calculation converges to the ground state. When this does not happen in a zero-bias calculation, the reason is usually either an improper setup of the device configuration or suboptimal settings in the NEGF device calculator. Finite-bias calculations may sometimes present additional convergence problems that we here address as well.

Troubleshooting steps

The lists below give a troubleshooting guide for both zero-bias and finite-bias calculations.

- *Zero-bias NEGF calculations*
  - *1. Increase the length of the device central region*
  - *2. Increase the length of the electrodes*
  - *3. Check that the boundary conditions are correct*
  - *4. Increase the number of k-points*
  - *5. Use the EquivalentBulk method for the initial density*
  - *6. Increase the electron temperature*
  - *7. Improve the accuracy of the contour integral*

–*8. Try a different Poisson solver*
–*9. Increase the density mesh cut-off energy*
•*Finite-bias NEGF calculations*
–*1. Always restart from a converged calculation*
•*SCF iteration control parameters*
–*1. Maximum steps*
–*2. Pulay mixing*
–*3. Tolerance*

Zero-bias NEGF calculations

1. Increase the length of the device central region

A very common cause of convergence difficulties is that the central region of the device is too short. Specifically, it is too short for the carriers in the central region to screen the electrode from the effects of the interface. In other words, the NEGF formalism requires that the electrostatic potential is equilibrated and become flat before reaching the electrode. If that is not the fulfilled, the NEGF algorithm will not be able to match it to the bulk-like electrode. In this case the remedy is simple – make the central region longer.

Note: For more information on the importance of screening, please see the page describing the *NEGF: Device Calculators*.

In the figure below, we show the Hartree Difference Potential for a device with metal on the left and silicon on the right. The vertical line indicates the interface between the two materials.

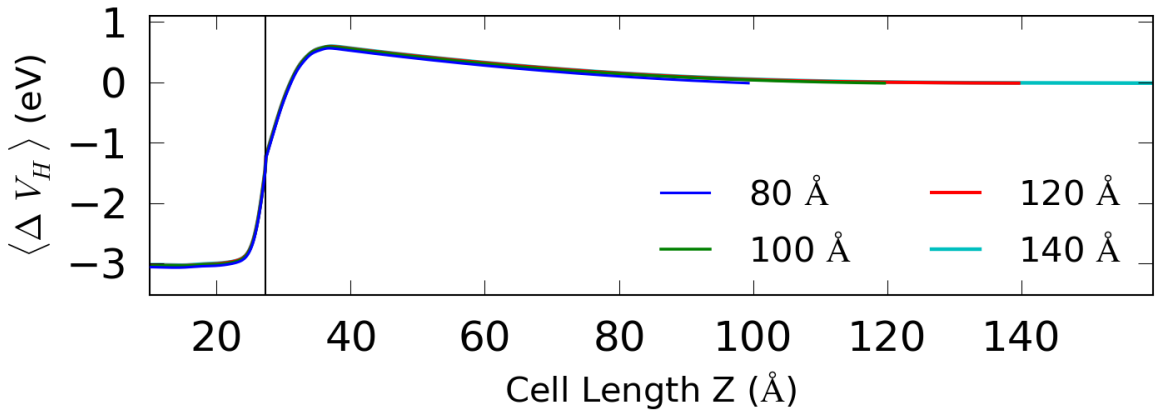


Fig. 4.65: A plot of the Hartree Difference Potential along the silicon screening region length. Note how 80 Å is not sufficient to allow the potential to become flat.

On the plot we see that the potential is not flat 80 Å of silicon away from the interface, but from 100 Å, it is. On the metal side, we see that just a few Ångstrom are enough to screen the interface and make the potential flat, as expected.

It is possible to decrease the length of the depletion region, i.e. the length of the semiconductor where screening takes place, by adding doping, or if already doped, by increasing the doping level. In the figure below, we show the depletion length vs. the doping level for silicon.

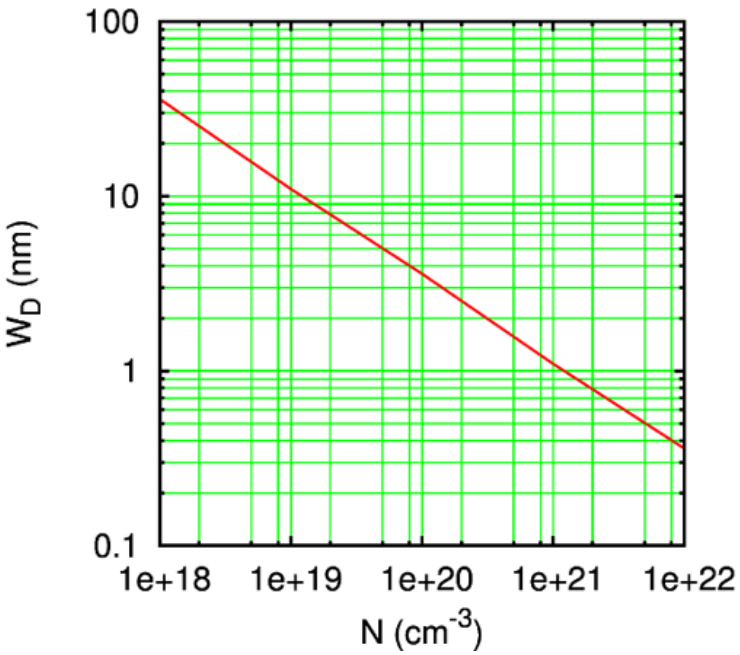


Fig. 4.66: A plot of screening (depletion) length vs. doping concentration. For instance, a doping of 10^{19} cm^{-3} corresponds to a depletion length of around 100 Å.

2. Increase the length of the electrodes

Another common cause of convergence issues is the size of the electrodes. It is very important that the electrodes are long enough to ensure that there is no direct interaction between second-nearest neighboring cells in the transport direction. Phrased another way; there must be no interaction between the electrode extension and the first periodic image of the electrode. A good rule of thumb is that the electrode should be more than twice as long, in the transport direction, as the longest basis function in the basis set. The length of the basis functions can be seen by plotting them in the **New Calculator** widget.

A more thorough test is to use the **Electrode Validator** tool in NanoLab. To use it, first do the self-consistent calculation of the electrode, then select the resulting *BulkConfiguration* on the *LabFloor*, and click on the Electrode Validator tool in the plugin bar on the right. NanoLab will then inform you whether the electrode is valid, i.e. long enough to fulfill the above criterion.

3. Check that the boundary conditions are correct

The next item on the list is to make sure that the calculation uses a Poisson solver with the correct boundary conditions. The boundary conditions should be periodic in A and B if the device is to be modeled in that way, and they should be Neumann if there is a gate in that direction. Finally, there should be Dirichlet boundary conditions in the C-direction.

4. Increase the number of k-points

The next thing to check is the k-point sampling in the transport direction. In order to match the Fermi levels between the electrode and the central region, it is important that the electrode k-point grid is very dense along the C-direction (the transport direction). The default density along the C-direction is 150 (compared to between 2 and 7 otherwise) and it is sometimes necessary to increase it even more. Note that, in most cases, the calculation of the electrodes is an insignificant part of the total NEGF calculation time, so there is in general no need to worry about the performance of this part.

It is also important to check that the number of k-points in the two transverse directions is reasonably converged. If they are too low in a way that leads to unphysical results, this can also hamper convergence. In the figure below, we show an example of how the computed Fermi level may depend on the number of k-points in the transverse directions. In this case, we would expect 1, 2 and possibly 3 k-points to give quite bad results, and possibly have convergence problems, whereas from 5 and up, the results are reasonably converged.

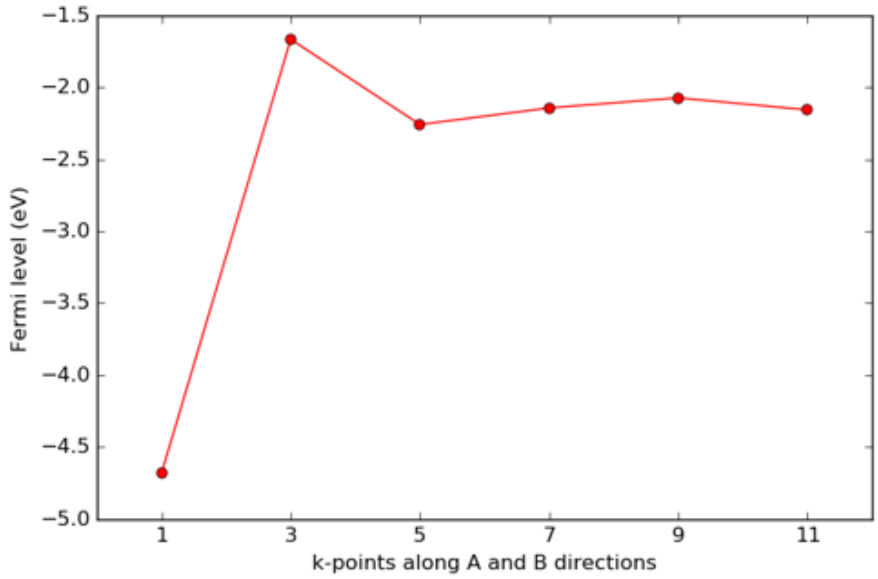


Fig. 4.67: The Fermi level depends on the number of k-points. Here we show the dependence along the non-transport A and B directions.

5. Use the EquivalentBulk method for the initial density

Next option is to try changing the calculation of the initial density to *EquivalentBulk*, instead of the default *NeutralAtom*. The default behavior is to create the initial guess for the density as a direct sum of the neutral atom densities. *EquivalentBulk* does a self-consistent calculation instead, where the central region is treated as a bulk configuration, and uses the resulting density as the initial guess for the NEGF calculation. Note that this calculation can be quite demanding, and may not be as efficiently parallelized over many computing cores as the NEGF calculation itself.

The *NeutralAtom* method works just fine in most cases, but for some systems it is necessary to use *EquivalentBulk* to start the device calculation from a better initial guess. However, if your electrodes are not identical and/or you use a Hückel model, it is most likely not an advantage to use *EquivalentBulk*.

6. Increase the electron temperature

Particularly in metals or small-gap semiconductors, convergence can be hampered if the states close to the Fermi level are not adequately described. If this is the case, these states will shift between being occupied and unoccupied, leading to convergence problems. This can be mitigated by increasing the temperature (width) for the electron occupation function, and/or trying one of the non-default occupation functions. Note that higher temperatures can lead to slight inaccuracies in the results, as described on the main page about *Occupation Methods*.

7. Improve the accuracy of the contour integral

When setting up the calculator in NanoLab, the lower bound of the contour integral is automatically set so low that it encompasses all the valence states in the elements in the configuration. This fact is quite important, and the next thing to do, is to check that this is indeed the case. If you are in doubt, make a DOS calculation of your material to identify the deepest valence states and their energy, and verify that the lower bound of the contour integral is lower in energy than these states.

Sometimes, convergence can be improved by increasing the number of points in the semicircle part of the default *SemiCircleContour* or switching to the *OzakiContour*, where the number of poles (equivalent to contour points) can be increased more easily, as there is only one parameter to adjust. Note that each contour point/pole must be calculated separately, like k-points, and the computational cost is thus approximately linear in the number of contour points/poles. For more information, see the manual pages on the *NEGF: Device Calculators* and the two types of contour integrals: *SemiCircleContour* and *OzakiContour*.

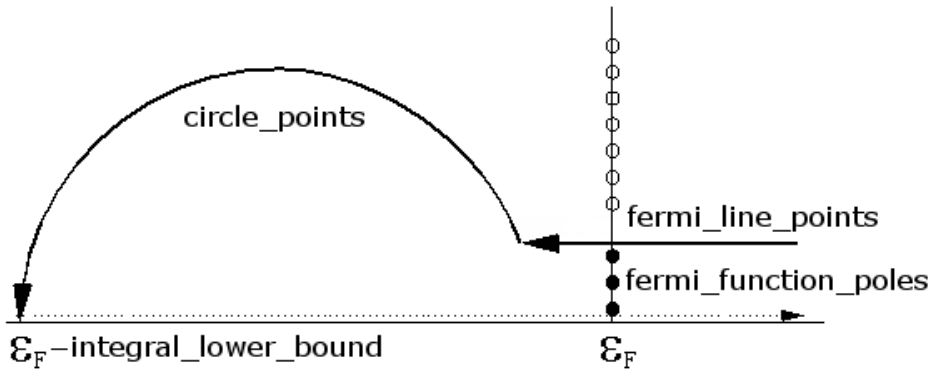


Fig. 4.68: Sketch of the contour integral when using the semicircle contour. See the manual page for more details.

8. Try a different Poisson solver

Convergence can also be helped by changing the Poisson Solver, e.g. from *MultigridSolver* to *ParallelConjugateGradientSolver*, as they use different numerical methods. This means the numerical noise in the calculated electrostatics will be different, which can in turn affect convergence in systems with more complicated electrostatics, such as gates. Note also that for larger calculations, which are run in parallel, *ParallelConjugateGradientSolver* will usually be faster.

9. Increase the density mesh cut-off energy

The final thing to check for a zero-bias calculation is the density mesh cutoff. As with k-points in the transverse direction, it should be high enough to provide physically correct results. Ensuring that it is properly converged with respect to the quantity of interest will also help with convergence. In the figure, we show how the Fermi level depends on the mesh cutoff, and it is seen how very low mesh cutoffs can give an inaccuracy in the determination of the Fermi level. However, note also the difference in scale between this and the figure with k-points above.

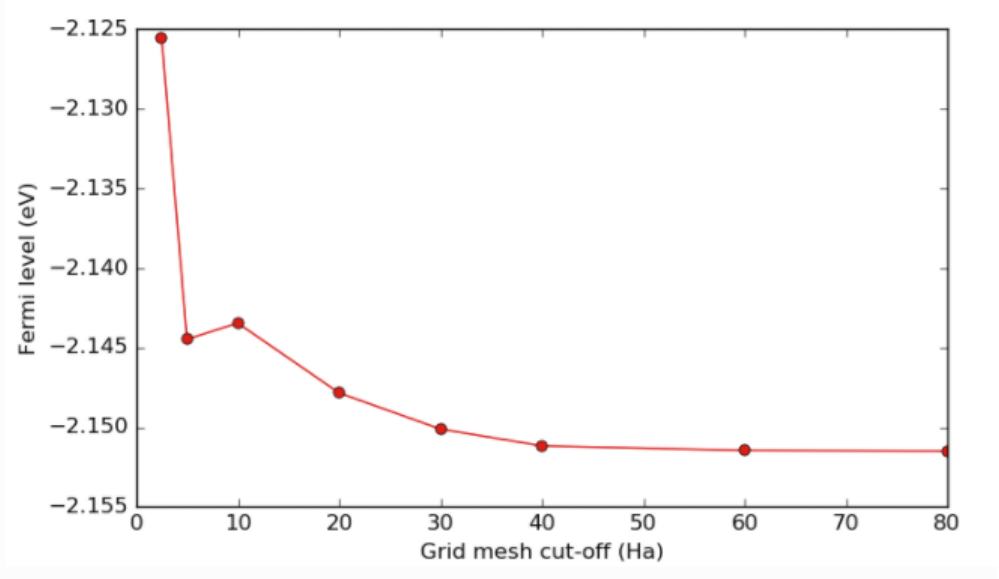


Fig. 4.69: The computed Fermi level depends on the grid mesh cut-off.

Finite-bias NEGF calculations

In general, finite-bias calculations are harder to converge than zero-bias, and all of the points above apply also in this case. However, there is one further thing to try, which will be explained in the next section, and should be tried as the first thing if the zero-bias calculation converged nicely.

1. Always restart from a converged calculation

A finite-bias calculation should always restart from a converged calculation with a lower, or zero, bias. If it does not converge, it will often help to decrease the bias difference. For instance, if you want to do a 0.4 V calculation starting from zero bias, the best approach is to do it incrementally, e.g. in steps of 0.1 V. This example script will do exactly that: `iv.py`

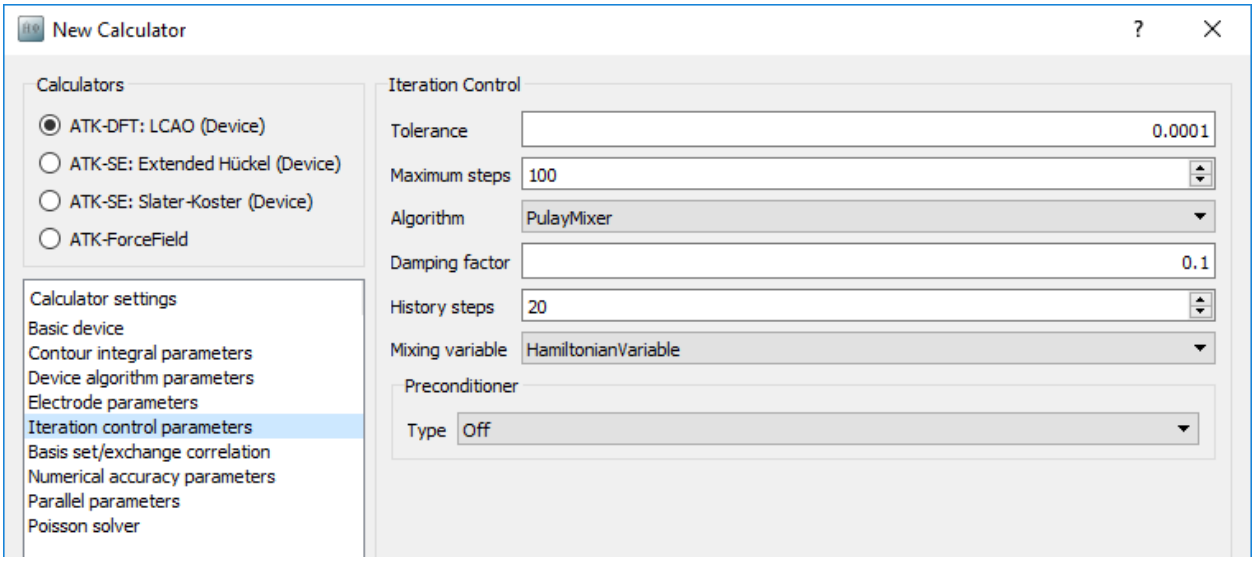
```
# Set bias
# Positive: Forward
# Negative: Reverse

bias_list = [0.10, 0.20, 0.30, 0.40]*Volt

# Read DeviceConfiguration
zero_bias_file = 'device_zero_bias.hdf5'
device_configuration = nlread(zero_bias_file, DeviceConfiguration)[-1]

for bias in bias_list:
    if processIsMaster():
        print("Bias is now: ", bias )
    # Get the calculator
    calculator = device_configuration.calculator()

    # Set the bias voltage
    calculator=calculator(electrode_voltages=(bias/2, -bias/2))

    # Attach the calculator and use the old initial state
    device_configuration.setCalculator(
      calculator(),
      initial_state=device_configuration)
    device_configuration.update()
    nlsave('device_bias_%.2f.hdf5' % bias.inUnitsOf(Volt), device_configuration)
```

This approach becomes even more relevant for higher biases, where shorter increments might be needed to achieve convergence. So in the case above, maybe 0.1 and 0.2 V converge, but 0.3 V do not. It would then be recommended to try 0.25 V, starting from 0.2 V and using increments of 0.05 V from there. Whether this approach is practical, of course depends on the system under study and the final bias you wish to achieve.

SCF iteration control parameters

If you have already verified that the **physical model** employed in your NEGF calculation is sane and appropriate (cf. the preceding sections), but the calculation still fails to converge, tuning the parameters that control the SCF loop is the last option.

The SCF parameters are defined in the `IterationControlParameters` class in **ATK Python** – navigate to the *Iteration control parameters* tab in the **New Calculator** widget to adjust the settings.



1. Maximum steps

Even a healthy and well-behaved NEGF calculation can sometimes converge slower than expected towards the ground state, and may therefore require extra SCF steps to converge correctly.

- In such cases, simply increase the parameter **Maximum steps**.
- However, first consult the QuantumATK log file from the failing calculation to check that the SCF loop is actually well-behaved, that is, converging towards a *reasonable ground state*. If not, additional SCF steps are not likely to solve the problem – consider instead adjusting the *Pulay mixing*.

What is a reasonable ground state?

First of all, **Band energies** and **Hamiltonian matrix elements** should converge steadily towards constant values. Inspect the ATK log file: Look for the lines reporting after each SCF iteration the band energy `E` and the differences in energy (`dE`) and Hamiltonian matrix elements (`dH`) from the previous SCF iteration. In a terminal this may conveniently be done using the *grep* command:

```
user@machine $ grep dE atk.log
|   0 E = -70.6148 dE =  5.719151e-01 dH =  7.266251e-01                    |
|   1 E = -56.8671 dE =  1.374766e+01 dH =  5.675282e-01                    |
|   2 E = -58.5695 dE =  1.702319e+00 dH =  1.737177e-01                    |
|   3 E = -59.6745 dE =  1.105067e+00 dH =  9.875918e-02                    |
|   4 E = -59.3529 dE =  3.216787e-01 dH =  4.048304e-02                    |
|   5 E = -59.0695 dE =  2.833359e-01 dH =  2.447425e-02                    |
|   6 E = -59.2082 dE =  1.386839e-01 dH =  2.978777e-03                    |
```

(continues on next page)

(continued from previous page)

```
|   7 E = -59.0397 dE =  1.684759e-01 dH =  1.492602e-02                    |
|   8 E =  -59.078 dE =  3.826926e-02 dH =  1.691435e-03                    |
|   9 E =   -59.05 dE =  2.800614e-02 dH =  2.041091e-03                    |
|  10 E = -59.0559 dE =  5.931576e-03 dH =  1.624175e-03                    |
|  11 E = -59.0553 dE =  6.427625e-04 dH =  2.825693e-04                    |
|  12 E = -59.0543 dE =  1.008929e-03 dH =  1.880459e-04                    |
|  13 E = -59.0542 dE =  1.964723e-05 dH =  6.034241e-05                    |
```

•The energy and Hamiltonian differences `dE` and `dH` should decrease steadily towards the specified tolerance (10^{-4} by default).

•The energy may oscillate in the beginning of the SCF cycle, but should eventually converge to a constant value.

Secondly, **Mulliken populations** should be physically reasonable. Inspect the *Density Matrix Report* for the last few SCF iterations in the ATK log file. For each atom, the quantities `DM` and `DD` are reported. The former is the trace of the device density matrix (the Mulliken populations) while the latter is the difference between `DM` and the neutral-atom charge on the atom (as given by the pseudopotential).

•Atoms in the same local environment should have approximately the same occupation.

•Variations should be systematic and physical, e.g., in an oxide we would expect extra electrons on the oxygen atoms and fewer on the more electropositive neighbors.

•Charge should not build up near the electrodes.

In the example below, we see how these points are fulfilled:

```
+-------------------------------------------------------------------------------+
| Density Matrix Report                                  DM        DD           |
+-------------------------------------------------------------------------------+
|   0  Si   [    1.907 ,    1.100 ,    1.564 ]    3.89764  -0.10236           |
|   1  Ni   [    0.001 ,   -0.000 ,    2.342 ]   18.21499   0.21499           |
|   2  Si   [    0.001 ,    2.201 ,    3.120 ]    3.89379  -0.10621           |
|   3  Si   [    0.001 ,   -0.000 ,    4.675 ]    3.89556  -0.10444           |
|   4  Ni   [    0.001 ,    2.201 ,    5.453 ]   18.21180   0.21180           |
|   5  Si   [    1.907 ,    1.100 ,    6.231 ]    3.89446  -0.10554           |
|   6  Si   [    0.001 ,    2.201 ,    7.787 ]    3.89564  -0.10436           |
|   7  Ni   [    1.907 ,    1.100 ,    8.565 ]   18.21115   0.21115           |
```

•The Mulliken population is approximately the same for all Ni and all Si atoms, respectively.

•We see that all Si atoms have a slightly negative value of `DD`, while the Ni atoms have a corresponding positive value of twice the size.

•The atoms close to the electrode boundary (those with lowest indexes) do not show unphysically large Mulliken populations.

2. Pulay mixing

The SCF loop uses a *PulayMixer* to generate the electron density in each SCF step, by mixing densities from previous steps into the guess for the current step. The **Damping factor** parameter controls the fraction of the current density guess which is added to the mix in the next step, i.e. a small damping factor will reduce changes in density between SCF steps. **History steps** controls the number of previous steps to use densities from.

The default Pulay mixing parameters should in general give a fast and stable SCF convergence, but sometimes a bit of tweaking can be useful:

- Try first to decrease the **Damping factor** to 0.05 or even 0.01. This should in particular help in case of metallic states at the Fermi level and is helpful when the convergence history exhibits oscillating behavior.
- Try to set the option *restart_strategy* to *AdaptiveHistoryRestart*. This is especially effective when the convergence history shows many contiguous steps where the error does not change significantly.
- Changing the number of **History steps** can improve convergence. A shorter history, e.g. 10 steps, can be especially helpful when doing a bias sweep. Sometimes increasing the number of **History steps** may also slightly improve convergence, but will also increase the memory consumption.

3. Tolerance

The SCF loop keeps going until the accuracy tolerance or the maximum number of steps is reached. This tolerance is measured on both the total band energy and the Hamiltonian matrix elements – in both cases the absolute difference from one SCF step to the other must be below the specified tolerance in order for the calculation to be considered accurate and converged.

- Increasing the **Tolerance** above the default value may in rare cases be needed to get the NEGF calculation to converge, but it may also lead to an inaccurate ground state. *A high tolerance should therefore only be used as a last resort!* In fact, it is more common to tighten the SCF tolerance (lowering it to e.g. 10^{-6}) to improve the accuracy of the converged ground state.

Contact support

If everything fails, write a post on the online QuantumATK Forum (https://forum.quantumatk.com) or contact QuantumATK support (quantumatk-support@synopsys.com). In both cases, please attach your ATK script and log file.

4.5.8 Spin-Polarized NEGF Convergence Guide

Version: O-2018.06

Downloads & Links

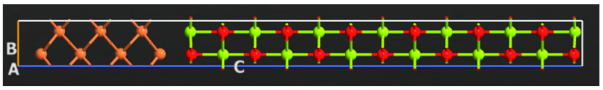
Transport calculations with QuantumATK NEGF Convergence Guide Spin transport in magnetic tunnel junctions ATK Reference Manual

QuantumATK delivers a market-leading and highly optimized implementation of the non-equilibrium Greens function (NEGF) method for spin-polarized electron transport calculations. Achieving fast and reliable convergence of the NEGF calculation to obtain the electronic ground state can sometimes be challenging as discussed in *NEGF Convergence Guide* for spin-unpolarized devices. This guide provides additional guidance when dealing with spin-polarized devices, e.g., magnetic tunnel junctions (MTJ), see also *Spin transport in magnetic tunnel junctions*.

Systems investigated

•Fe(001)|MgO reference structure

•Fe(001)|MgO|Fe(001) magnetic tunnel junction (MTJ)



Computational parameters for the reference Fe|MgO structure

•1×1 lateral unit cell Fe(001)|MgO.

•Dielectric layer: 13 MgO monolayers.

•Ferromagnetic metal: 7 Fe monolayers.

•Norm-conserving pseudopotential: SG15.

•LCAO basis set: `Medium`.

•K-points mesh: $21 \times 21 \times 1$.

•Mesh cutoff: 150 Hartree.

•SCF threshold: 10^5 Hartree

•Forces threshold: 0.05 eV/Å.

•Geometry optimization: full relaxation of Fe and MgO structure and lattice parameters.

Computational parameters for the Fe|MgO|Fe device

•**Basic parameters**

–2×2 lateral unit cell Fe(001)|MgO|Fe(001) junction.

–Dielectric spacer: 13 MgO monolayers.

–Norm-conserving pseudopotential: SG15.

–LCAO basis set: `Medium`.

–Mesh cutoff: 150 Hartree.

–SCF threshold: 10^5 Hartree.

•**Structural parameters taken from a reference Fe(001)|MgO structure**

–Bulk Fe lattice parameter: $a_{\mathrm{Fe}} = 3.00$ Å.

–Fe-O separation distance: $d_{\mathrm{Fe-O}} = 2.11$ Å.

–Mg-O separation distance: $d_{\mathrm{Mg-O}} = 2.18$ Å.

•Parameters that have been varied

–Electrodes

∗Length: 4 or 6 Fe monolayers

–Bias voltage

∗Voltage range: 0.0, 0.05 and 0.1 V, as set in the *IVCharacteristics* study object.

–K-point sampling

∗Device calculations: 11×11 and 15×15.

∗Electrode calculations: $11 \times 11 \times 151$ and $15 \times 15 \times 151$.

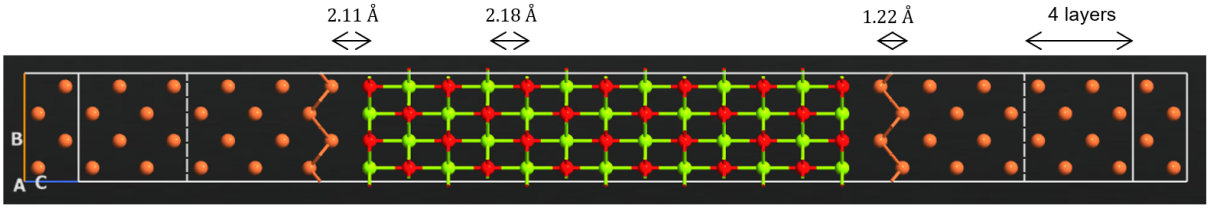
–Mixing parameters

∗Damping factor: 0.1 and 0.05.

∗Number of history steps: 20 and 12.

Calculating the self-energy matrix

•Greens-function matrix



Σ^{L} (Σ^{R}) is the self-energy matrix of the left (right) electrode.

1. Method overview

•Equilibrium part

–*RecursionSelfEnergy*: iterative scheme.

–*SparseRecursionSelfEnergy*: sparse-matrix version of *RecursionSelfEnergy*.

•Non-equilibrium part

–*RecursionSelfEnergy*.

–*SparseRecursionSelfEnergy*.

–*KrylovSelfEnergy*: iterative Krylov subspace method. The parameter λ_{min} determines the size of the subspace - only modes with a decay slower than λ_{min} are included.

–*DirectSelfEnergy*: direct diagonalization scheme - all modes included.

2. Practical methods

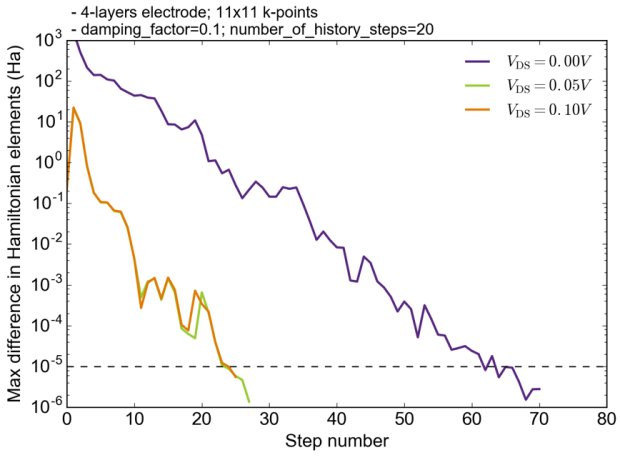
- In practice, *RecursionSelfEnergy* and *SparseRecursionSelfEnergy* are the methods of choice for both equilibrium and non-equilibrium calculations.
- The best method for equilibrium and non-equilibrium calculations in terms of speed and memory footprint can be chosen by using *DevicePerformanceProfile* to analyze the performance of the device calculation.

```
# Create the performance profile and print a report.
device_performance_profile = DevicePerformanceProfile(device_configuration)
nlprint(device_performance_profile)
```

```
+-------------------------------------------------------------------------------+
| Device Performance Profile (1 process)                                        |
+-------------------------------------------------------------------------------+
| Contour point timing (s):                                                     |
|                                    EQ        NEQ                              |
| GreensFunction                   5.10       7.49                              |
| SparseGreensFunction            23.59      11.57                              |
|                                                                               |
| Fastest EQ method (by 4.6 times): GreensFunction                              |
| Fastest NEQ method (by 1.5 times): GreensFunction                             |
+-------------------------------------------------------------------------------+
| Peak memory usage/process (MB):                                               |
|                                    EQ        NEQ                              |
| GreensFunction                1446.13    1913.02                              |
| SparseGreensFunction          1628.67    1589.79                              |
|                                                                               |
| Most memory-efficient EQ method (by 1.1 times): GreensFunction                |
| Most memory-efficient NEQ method (by 1.2 times): SparseGreensFunction         |
+-------------------------------------------------------------------------------+
```

Zero and finite-bias convergence: mixing parameters

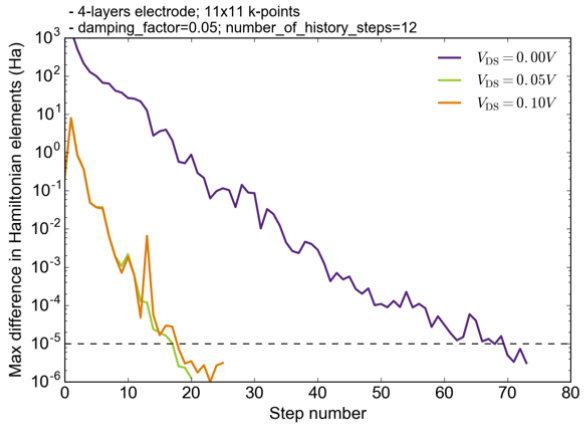
- **General considerations**
  - All calculations converge, independently on the parameters used.
  - The convergence is almost unaffected by the variation of the computational parameters.
  - Convergence at finite bias voltages is much faster than at zero bias, because the density matrix from the most close bias voltage is used as the initial state.

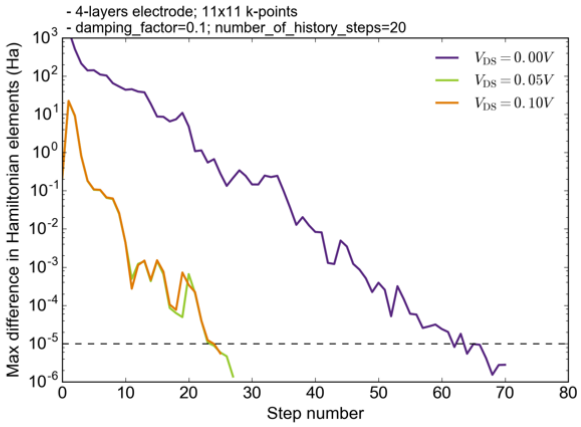


- If slightly less aggressive mixing parameters are used compared to the default ones (lower `damping_factor` and `number_of_history_steps`), the convergence at zero (finite) bias voltage is slightly slower (faster).

$$G(\varepsilon) = [(\varepsilon + i\delta_+)S - H - \Sigma^L(\varepsilon) - \Sigma^R(\varepsilon)]^{-1}.$$

•Increasing the `k_point_sampling` along A and B from 11×11 to 15×15 results in a slightly slower convergence at zero bias voltage, but in a faster convergence at finite bias voltages.

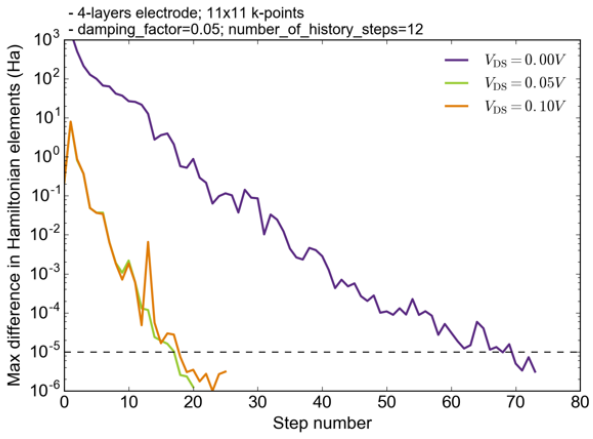


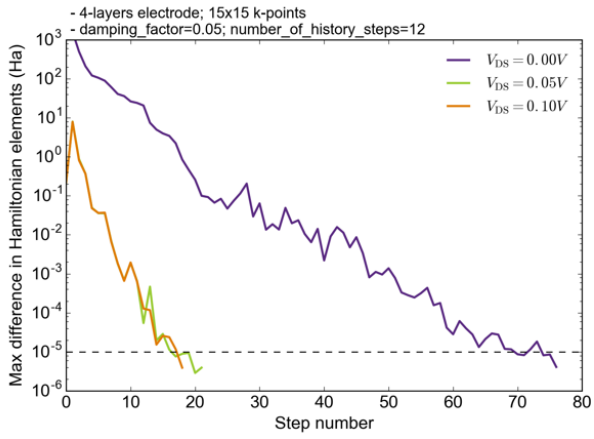


Zero and finite-bias convergence: electrode-length and k-point sampling

•Increasing the electrode length from 4-layers to 6-layers results in a slightly slower convergence at zero-bias voltage. Convergence at finite-bias voltages is very similar.

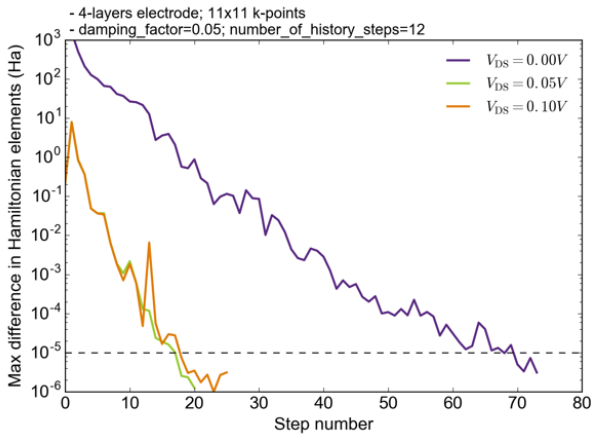
•The 6-layers electrode calculations are more memory intensive than the 4-layers electrode calculations. In case of insufficient memory, it is possible to reduce the memory footprint by re-evaluating the self-energies at each SCF step `storage_strategy=NoStorage()`.

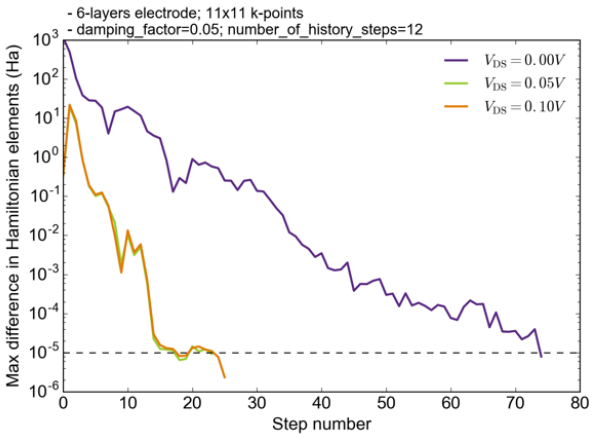




Electrode validator

- The **electrode validator** allows one to check if the electrode repetition is sufficiently long in the C direction. The length of the electrode ultimately depends on the basis set used in the calculations.
- In the present calculations, the four-layers electrode is slightly too short and some matrix elements are discarded. *For production calculations with Fe electrodes and the SG15-Medium pseudopotential/basis set, a 6-layers electrode should be used.*





Contact support

If you failed to converge your device calculation even after following our guidelines, write a post on the online QuantumATK Forum (https://forum.quantumatk.com) or contact QuantumATK support (quantumatk-support@synopsys.com). In both cases, please attach your QuantumATK script and log file.

4.5.9 Noncollinear spins and spin transfer torque in devices

Version: P-2019.03

Downloads & Links

Transport calculations with QuantumATK ATK Reference Manual `Technical note`

Click `here` to download a PDF note that documents how the density functional noncollinear spin formalism is implemented in QuantumATK, and how the spin transfer torque (STT) is computed for spintronic two-probe devices.

4.5.10 Parallelization of QuantumATK calculations

Downloads & Links

Basic QuantumATK Tutorial ATK Reference Manual

ATK uses a range of parallelization techniques to distribute computational workload over a number of computing cores (CPUs). This may either reduce the total wall-clock time or reduce the per-core peak memory requirement. We will here explore the basics of how to parallelize QuantumATK calculations, both for bulk systems and for NEGF device calculations.

ATK offers **multi-level parallelization** and will by **default** try to distribute the computing workload in the best way possible. However, manual tuning of parallelization settings may sometimes be beneficial. In this case, the QuantumATK user needs to be aware of an important distinction between two very different ways to measure the effect of parallelization:

Efficiency
If efficiency is the goal of a particular parallelization scheme (which it often is), the speed-up per extra CPU should be as high as possible. That is, a calculation that takes 1 hour in serial (on a single CPU) should ideally take 15 minutes on 4 CPUs. *Parallelization efficiency is therefore important if efficient use of CPU resources is more important than the wall-clock time spent on the calculation*.

Time-to-result
If the goal of parallelizing over several CPUs is to get the calculation to finish as fast as possible, one will usually choose to compromise on efficiency. For example, parallelizing a bulk calculation with 4 *k*-points over 40 CPUs will rarely give a speed-up of 40 compared to running the calculation on 1 CPU. However, it will in most cases still lead to a significant reduction of the wall-clock time, by a factor of maybe 10-20 depending on the system. *This may therefore result in a less efficient parallelization scheme, when comparing the total CPU-time to the wall-clock time, but may be preferred if getting the result quickly is more important than very efficient use of computing resources*.

Unit-of-work

When distributing an entire QuantumATK calculation over a number of CPUs, the smallest package of computational workload that is conveniently assigned to a single CPU is one unit-of-work. Importantly, this entity is not the same for bulk and device (NEGF) calculations.

Bulks	Devices (NEGF)
1 *k*-point	1 contour point

For a bulk calculation, the natural unit-of-work is one single *k*-point: A bulk calculation with 4 irreducible *k*-points is efficiently parallelized over 4 CPUs, and QuantumATK will **automatically** assign one *k*-point to each CPU, provided 4 CPUs are available.

For device calculations, one NEGF contour integration point is the fundamental unit-of-work. Contributions to the density matrix from all transverse *k*-points (N_k, orthogonal to the transport direction) must be evaluated at each contour energy point (N_e, usually equal to 48), so the total workload consists of $N_k \times N_e$ units of work. QuantumATK will **automatically** distribute these over the available CPUs as efficiently as possible. Obviously, the contour integration is fully distributed if $N_k \times N_e$ CPUs are available (one contour point on each CPU).

Parallelization levels in QuantumATK

The QuantumATK calculator engines contain several layers of algorithms that can benefit from parallelization in different ways. QuantumATK can parallelize over both MPI processes and threads.

MPI parallelization

The message passing interface (MPI) protocol is used to distribute work units as individual computing processes on individual CPUs, and also allows for assigning multiple processes to each work unit. Moreover, each MPI process may be further distributed in a hybrid parallelization scheme by employing shared-memory threading of each process, see below.

Threading

Threading allows for distributing the work load of each computing process (each MPI process) over several CPUs in a shared-memory fashion. Using threading can be advantageous if either

1. There are more CPUs available than units-of-work: If full MPI parallelization uses less than half of the computing resources, threading may be used to utilize the remaining resources.
2. The simulation memory requirement needs to be reduced: Using threading instead of MPI processes can often reduce the memory footprint at the cost of longer time-to-result.

The number of threads is automatically controlled by QuantumATK so as to make the best use of the available resources, taking into account the number of MPI processes used. An example for submission to SLURM is given below:

```
#!/bin/bash

# Job name
#SBATCH --job-name test
#SBATCH --tasks=16                      # Number of MPI ranks
#SBATCH --cpus-per-task=2               # Number of cores per MPI rank
#SBATCH --nodes=2                       # Number of nodes
#SBATCH --tasks-per-node=8              # How many tasks on each node
#SBATCH --ntasks-per-socket=4           # How many tasks on each socket
#SBATCH --partition=xeon16
# Export all environment variables
#SBATCH --export=ALL

export ATK_EXE=/home/user/QuantumATK/QuantumATK-P-2019.03/bin/atkpython
export MPI_EXE=/home/user/QuantumATK/QuantumATK-P-2019.03/libexec/mpiexec.hydra
export SNPSLMD_LICENSE_FILE=27020@hostname

export MKL_DYNAMIC=TRUE

${MPI_EXE} -n 16 -ppn 8 ${ATK_EXE} test.py > test.log
```

The QuantumATK log file lists the number of threads used by each MPI process under the CPU Information header. In general, the number of processes multiplied by the number of threads per process should equal the total number of available physical processors on the machine, for maximum efficiency.

QuantumATK will automatically detect how many threads can be safely used, but if you want to set the maximum number of threads per process to a smaller value, this can be done by exporting the environment variable `MKL_NUM_THREADS`,

e.g., `export MKL_NUM_THREADS=2` if you wish to have no more than 2 threads per process even if more cores are available.

Warning: Setting `MKL_DYNAMIC=FALSE` will not disable threading, but only the ability of MKL to dynamically adjust the number of threads for maximum efficiency. If you wish to disable threading, you should set `export MKL_NUM_THREADS=1`.

ATK-DFT

For regular ATK-DFT calculations, it is very efficient to parallelize over k-points, and this will always be prioritized. In all cases, QuantumATK will automatically determine the optimal parallelization strategy based on the available CPU cores. Additionally, it will print a warning if the specifics of the calculation makes it impossible to use an optimal parallelization with the current number of CPU cores. In this case, it will be more efficient to change the number of CPU cores to better match the calculation, such as making sure the number of CPU cores is a divisor of the total number of k-points.

Warning: The automatic parallelization scheme in QuantumATK is built on the assumption that you have enough (infinite) memory available. You should therefore check the settings for memory-intensive calculations to ensure that the calculations run with optimal efficiency.

In more advanced calculations, such as NEB and AKMC, further levels of parallelization can be used to increase efficiency. We will get back to this at the end of this document.

Bulk calculations

As mentioned above, the fundamental unit-of-work for a bulk calculation is one single k-point. Default parallelization settings will automatically assign 1 k-point per MPI process, if possible. However, 2 k-points cannot be optimally distributed on 3 cores (1 core would be idle), but they can actually be distributed on 4 cores by assigning 2 cores to work on each k-point. This is illustrated in the figure below.

The matrix shows how QuantumATK will **automatically** distribute the workload in the case of a bulk calculation with 2 irreducible k-points. Each filled circle represents a k-point, i.e., one unit-of-work. Blue, red, and green colors indicate that 1, 2, and 3 processors are used per k-point, respectively. The black dashes indicate that one CPU will be idle if the calculation is parallelized over 3 or 5 processors.

Processes per *k*-point

The matrix in Fig. 4.70 shows that choosing the number of computing cores, N_{MPI}, to be either 1 or a multiple of the number of k-points, N_k, leads to a parallelization scheme that utilizes all the allocated cores. With 4, 6, 8, etc. cores, the workload for each k-point is **automatically** shared by several cores. However, this kind of parallelization can also be selected **manually** by the user.

The *DiagonalizationSolver* calculates the density matrix by direct diagonalization of the Hamiltonian matrix, and the solver accepts the parameter `processes_per_kpoint`, which sets the number of MPI processes assigned to work on each k-point. In the matrix in Fig. 4.70, `processes_per_kpoint` is 1, 2, and 3 for blue, red, and green circles, respectively.

While `processes_per_kpoint = Automatic` is the default, it is easy to set it using the **Script Generator**.

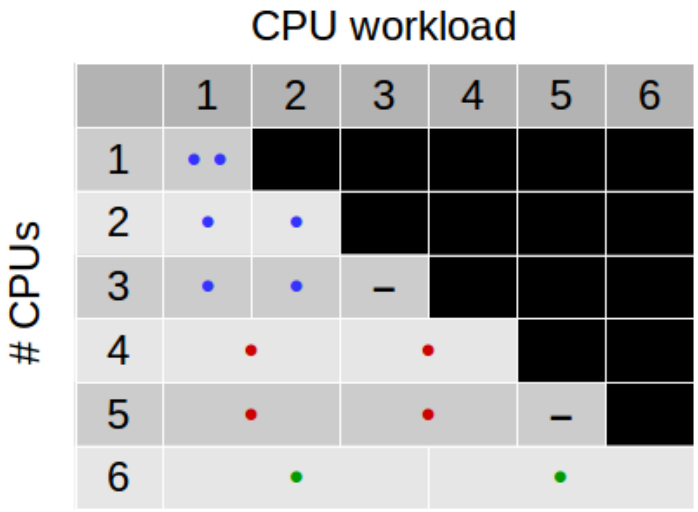
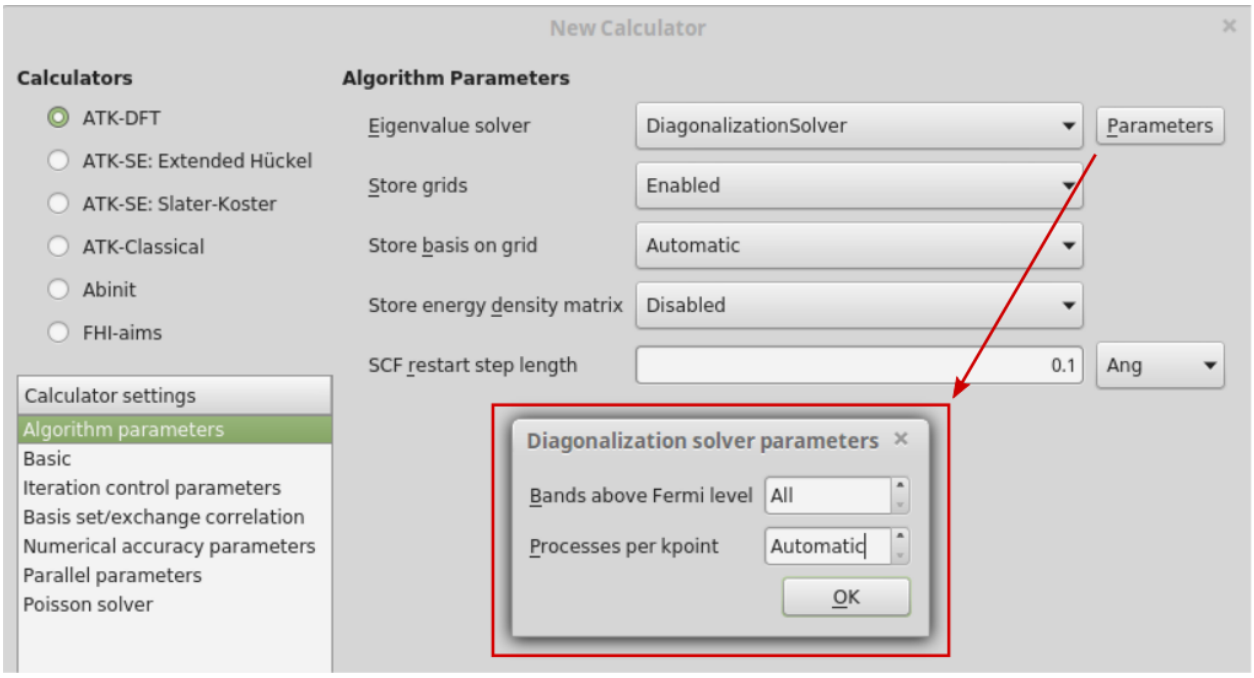


Fig. 4.70: Matrix illustrating how QuantumATK distributes the workload in the case of a bulk calculation with 2 irreducible k-points.

Open the **Calculator** settings window, and select *Algorithm Parameters*. Make sure `DiagonalizationSolver` is chosen as eigenvalue solver, and click the **Parameters** button to open a small window that allows you to manually set the number of processes per k-point.



Exercise I: Non-default number of processes per *k*-point

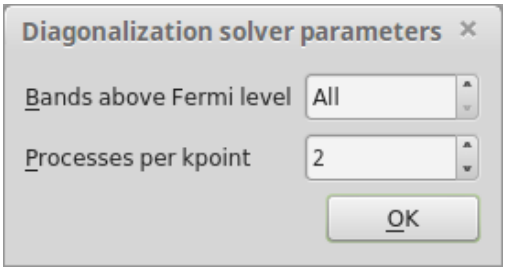
•Open the QuantumATK **Builder**, click *Add* → *From Database*, and add the `Silver` bulk configuration to the Stash.

•Use the *Bulk Tools* → *Repeat* plugin to double the size of the configuration along all directions (2x2x2 repetition).

•Send the silver bulk to the **Script Generator**, and double-click the **New Calculator** icon to add the default ATK-DFT calculator to the script.

•Double-click the added calculator to open the calculator settings window. Select *Algorithm Parameters* and click the **Parameters** button.

•Then change the number of processes per *k*-point from `Automatic` to `2`, and click *OK*. Also close the calculator settings window.



Tip: The *DiagonalizationSolver* also accepts the parameter `bands_above_fermi_level`, which is by default set to `All` bands. Explicitly setting this parameter to a positive integer can often be used to speed up the ATK-DFT calculation. However, be aware that too small values of `bands_above_fermi_level` will lead to inaccuracies in the diagonalization routine.

•Finally, send the script to the **Editor** to see how the parameter `processes_per_kpoint` is set in the script. The script will look like shown below.

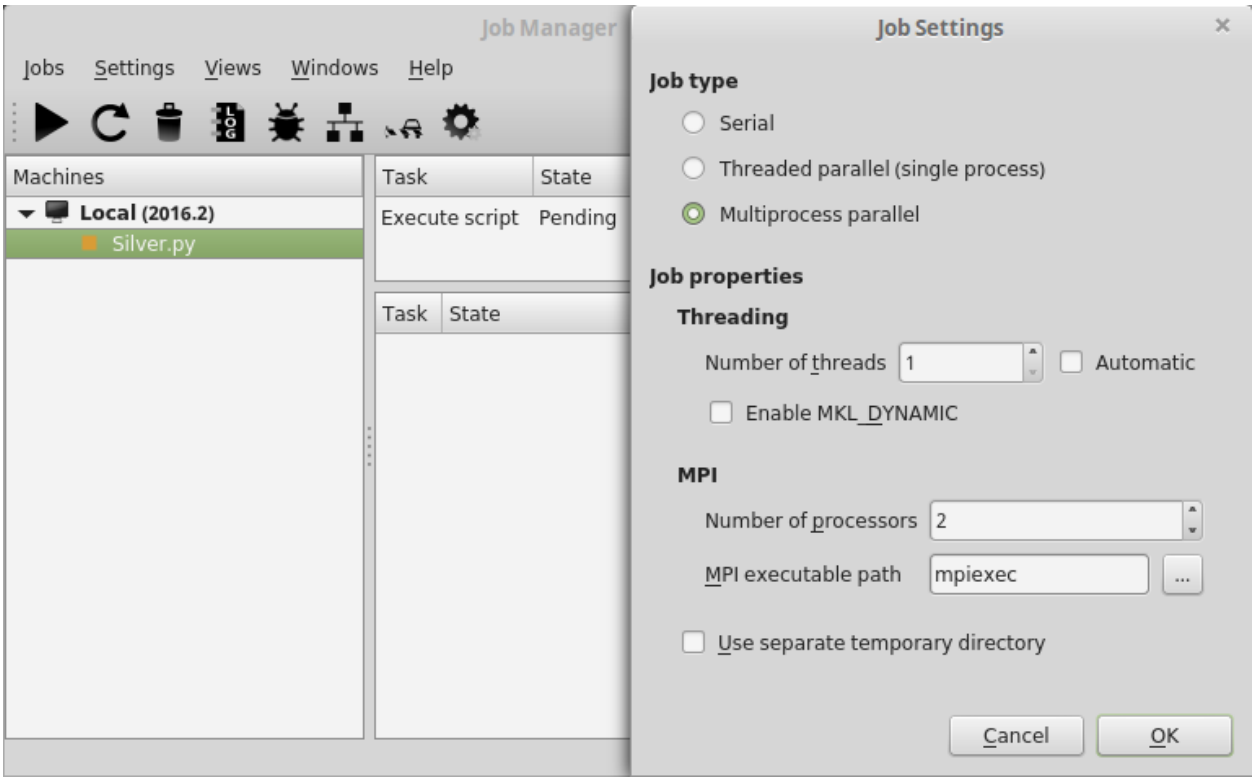
```
# -*- coding: utf-8 -*-
# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(8.1714*Angstrom)

# Define elements
elements = [Silver, Silver, Silver, Silver, Silver, Silver, Silver, Silver]

# Define coordinates
fractional_coordinates = [[ 0. ,  0. ,  0. ],
                          [ 0. , -0. ,  0.5],
                          [ 0. ,  0.5, -0. ],
                          [ 0. ,  0.5,  0.5],
                          [ 0.5,  0. , -0. ],
                          [ 0.5,  0. ,  0.5],
                          [ 0.5,  0.5, -0. ],
```

(continues on next page)

(continued from previous page)

```
                          [ 0.5,  0.5,  0.5]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(
    na=3,
    nb=3,
    nc=3,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=k_point_sampling,
    )

density_matrix_method = DiagonalizationSolver(
    processes_per_kpoint=2,
    )
algorithm_parameters = AlgorithmParameters(
    density_matrix_method=density_matrix_method,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    algorithm_parameters=algorithm_parameters,
    )

bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()
nlsave('Silver.hdf5', bulk_configuration)
```

- Notice lines 41–43, where `processes_per_kpoint=2` is set for the *DiagonalizationSolver*. This defines the density matrix method that is passed on to the *AlgorithmParameters* object, which is eventually given as an option for the *LCAOCalculator*.
- If you have MPI installed on your machine, you should try to run the calculation with parallelization over 2 cores. Use either the **Job Manager**, or run the script directly from **Command-line**.




```
$ mpiexec -n 2 atkpython Silver.py > Silver.log
```

•In any case, you will the following **DiagonalizationSolver parallelization report** in the QuantumATK job logfile, here named `Silver.log`:

```
+-----------------------------------------------------------------------------+
| DiagonalizationSolver parallelization report.                               |
+-----------------------------------------------------------------------------+
| Total number of processes: 2                                                |
| Total number of k-points: 14                                                |
| Processes per k-point: 2                                                    |
| Number of process groups: 1                                                 |
+-----------------------------------------------------------------------------+
```

Speed-up and peak memory reduction

The figure below illustrates how the calculation wallclock time and peak memory requirement of a simple ATK-DFT calculation may both be reduced by using MPI parallelization. The bulk calculation has 4 irreducible k-points.

In Fig. 4.71, blue triangles indicates wallclock time and red circles indicate the peak memory requirement on each core. Horizontal blue lines indicate 100%, 50%, 25%, and 12.5% of the wallclock time without parallelization (1 MPI process). They indicate perfect efficiency for 1, 2, 4, and 8 MPI processes.

It is clear that a significant speed-up is gained with 2–4 MPI processes, and that using 4 cores (such that each k-point fits onto a single core) reduces the wallclock time by almost a factor of 4. However, 5–7 MPI processes bring no further performance gains, because the extra cores are largely idle (there are only 4 k-points to distribute). A significant speed-up occurs with 8 MPI processes (on 8 cores), where the value of `processes_per_kpoint` automatically switches from 1 to 2, such that all 8 cores can do useful work. With this in mind, the green circles indicate the most efficient choices for the number of MPI processes.

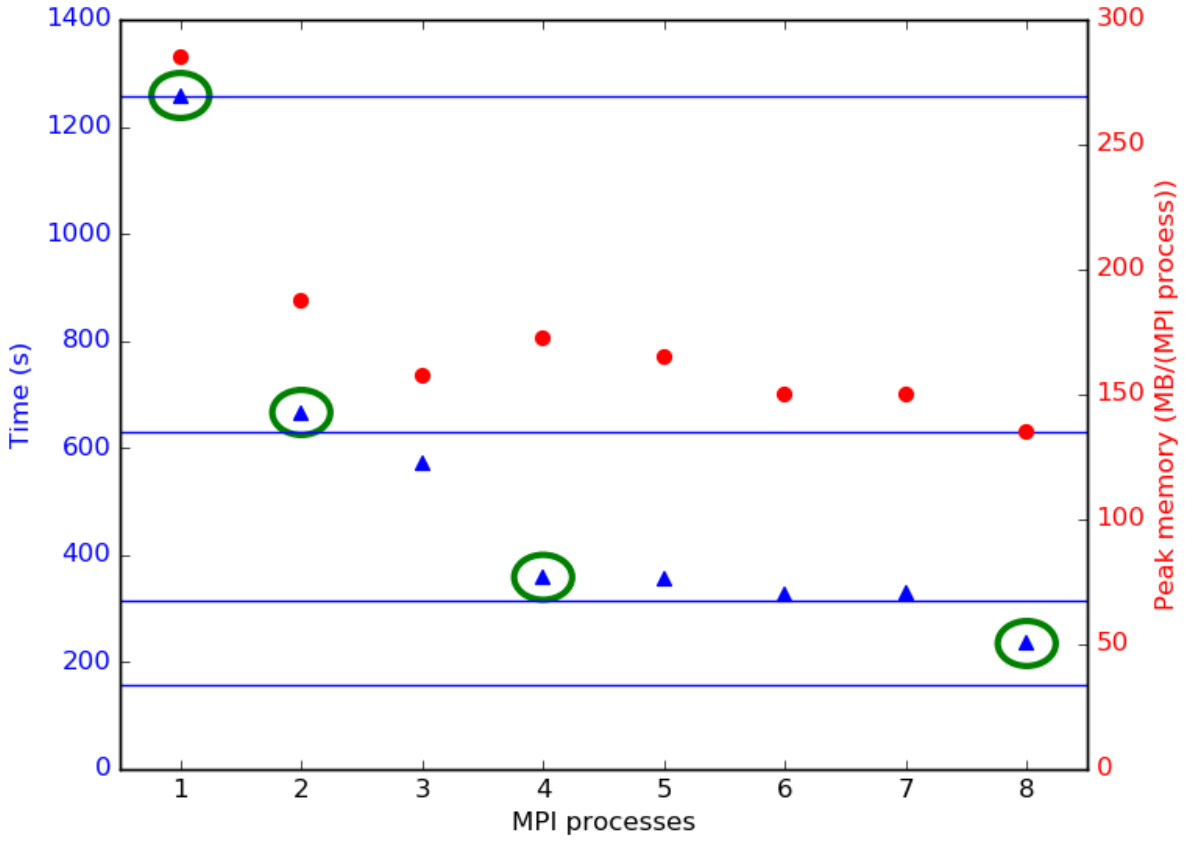


Fig. 4.71: Wallclock time and peak memory requirement when parallelizing a simple bulk calculation using MPI.

Regarding peak memory, the most significant memory reduction happens already with 2 MPI processes. The smallest peak memory is obtained with `processes_per_kpoint` larger than 1, i.e., 8 MPI processes. Bear in mind that the amount of peak memory reduction obtainable by workload distribution is highly system dependent. Large systems will usually offer better opportunities for reducing the memory footprint this way.

NEGF calculations

In a SurfaceConfiguration or in a DeviceConfiguration, the system is effectively treated as infinite along the C cell vector, and as periodic along the A and B cell vectors. The 3D Brillouin zone typical of a BulkConfiguration is thus projected onto a 2D Brillouin zone associated with the periodically repeated system in A and B. The electronic structure of the system is thus solved in a way analogous to a BulkConfiguration in A and B, but in a different way along C.

The *GreensFunction* method uses a block tridiagonal inversion method used to calculate the Greens function and lesser Greens function at each at each k-point and at each energy ε. The lesser Greens function is then used to obtain the full density matrix of the system $\hat{D}$:

$$\hat{D} = -\frac{1}{\pi}\int\int \mathrm{Im}G^{<}(\varepsilon, k)\, d\varepsilon\, dk$$

where $G^{<}(\varepsilon, k)$ is the lesser Greens function. In QuantumATK, the meaning of ε depends on whether the system is at equilibrium or out-of-equilibrium:

- **Equilibrium** : ε is a complex energy and the Greens function is obtained using **complex contour integration**
- **Non-equilibrium** : ε is a real energy and the Greens function is obtained using **real contour integration**

In practice, the integration is carried out considering a double weighted sum over two discrete sets of k-points $N_k = [k_1, \ldots, k_\mathrm{n}]$ and energies $M_\varepsilon = [\varepsilon_1, \ldots, \varepsilon_\mathrm{m}]$

$$\hat{D} = -\frac{1}{\pi}\sum_{i=1}^{N_k} w_{k,i} \sum_{j=1}^{M_\varepsilon} w_{\varepsilon,j}\mathrm{Im}G^{<}(\varepsilon_j, k_i)$$

Note: Since the evaluation of the Greens function at non-equilibrium is done using real contour integration, many more ε-points are needed! This also means the calculation can potentially scale to many more processes.

The evaluation of each element $G^{<}_{i,j}$ is the computationally intensive part of the whole procedure and takes approximatively the same time for both *i* and *j*, so the double sum can be conveniently rewritten as a single sum over a generalized set of **contour points** $N_k + M_\varepsilon = [x_1, \ldots, x_{\mathrm{n+m}}]$, which refer both to energies and k-points:

$$\hat{D} = -\frac{1}{\pi}\sum_{i=1}^{N_k+M_\varepsilon} w_{k,\varepsilon,i}\mathrm{Im}G^{<}(x_i)$$

Each contour point can be regarded as the **unit of work** in a NEGF calculation.

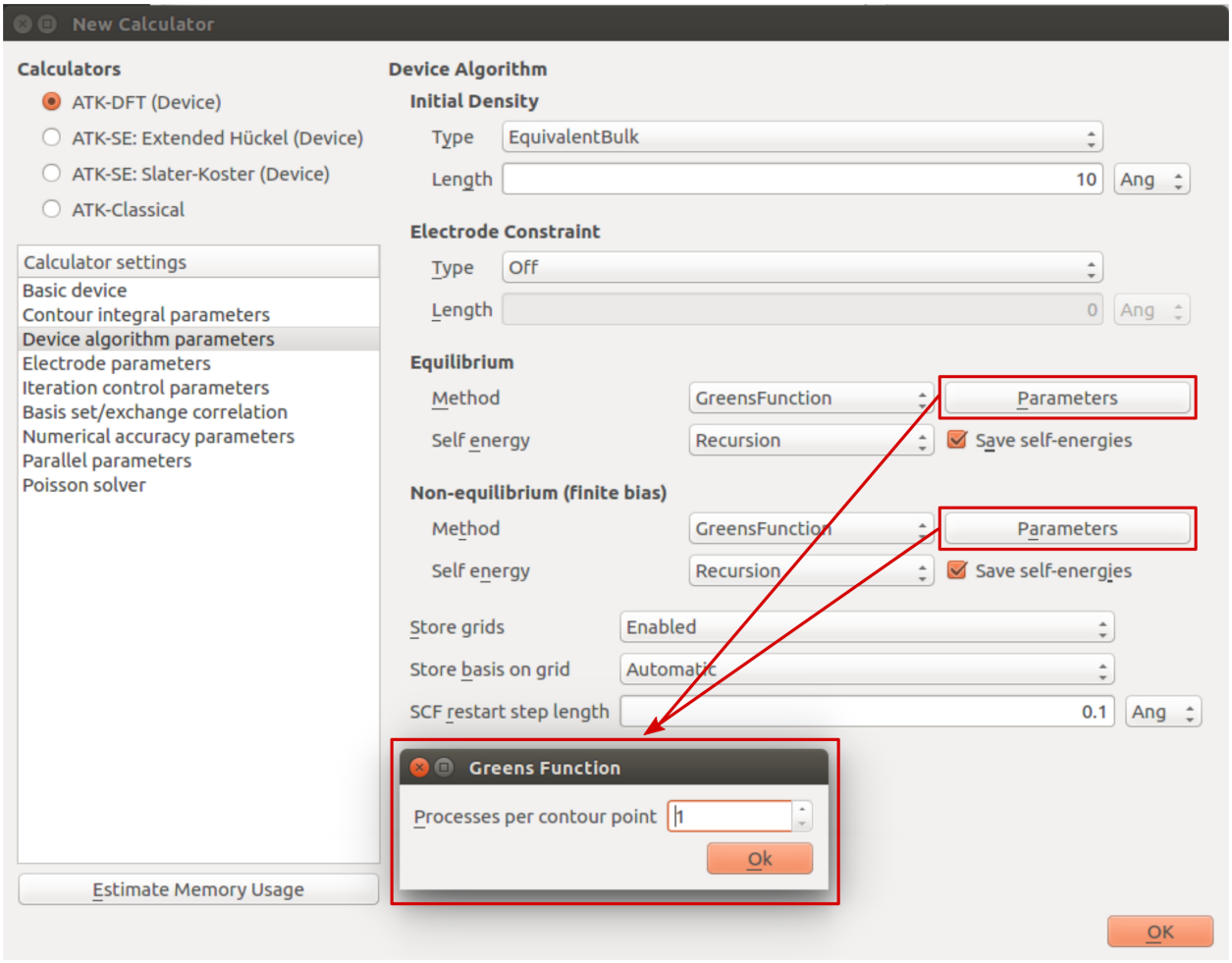
Automatic distribution of contour points

By default, QuantumATK tries to distribute the contour points in a NEGF calculation on as many processes as possible. The evaluation of each contour point is done independently, which implies the following distribution rules for $N_k \times M_\varepsilon$ contour points:

- $N_{MPI} = N_k \times M_\varepsilon$: An optimal distribution of 1 contour point per process is achieved
- $N_{MPI} <= N_k \times M_\varepsilon$: QuantumATK will calculate groups of N_{MPI} processes stepwise. If N_{MPI} is not a dividend of $N_k \times M_\varepsilon$, some processes will be partially idle.
- $N_{MPI} > N_k \times M_\varepsilon$: There will not be any improvement compared to the ideal case, but more memory will be available for each contour point.

Processes per contour point

The option `processes_per_contour_point` allows to control the **number of processes used to solve each individual unit of work**. In a device or surface calculation, it can be set in the **Script Generator** by opening the **Calculator** settings window, and selecting *Device Algorithm Parameters*. Make sure `GreensFunction` is chosen as the method, and click the **Parameters** button to open a small window that allows you to manually set the number of processes per contour point.



The default value `processes_per_contour_point = 1` provides the maximum efficiency in the calculation. For very large calculations, setting `processes_per_contour_point` to a number larger than 1 allows you to increase the available memory per individual unit of work beyond that available for a single process. As an example, suppose you have $N_k \times M_\varepsilon$ contour points and $N_k \times M_\varepsilon$ processes, each one having a maximum memory available which is only half of that required to solve a contour point. In this case, by setting `processes_per_contour_point = 2`, two groups of $(N_k \times M_\varepsilon)/2$ will be solved in a stepwise fashion by using two cores per unit of work, effectively doubling the memory available per contour point.

Speedup in a device calculation

Fig. 4.72 shows the parallel scaling of a NEGF calculation of a Ag/Si(100) interface, similar to the one considered in the paper *General atomistic approach for modeling metal–semiconductor interfaces using density functional theory and non-equilibrium Greens function*[1]. The system is calculated at equilibrium using N_k = 10 k-points in the irreducible 2D Brillouin zone, and M_ε = 48 ε-points on the complex contour for the evaluation of the equilibrium Greens function. This results in a total of $N_k \times M_\varepsilon$ = 480 contour points.

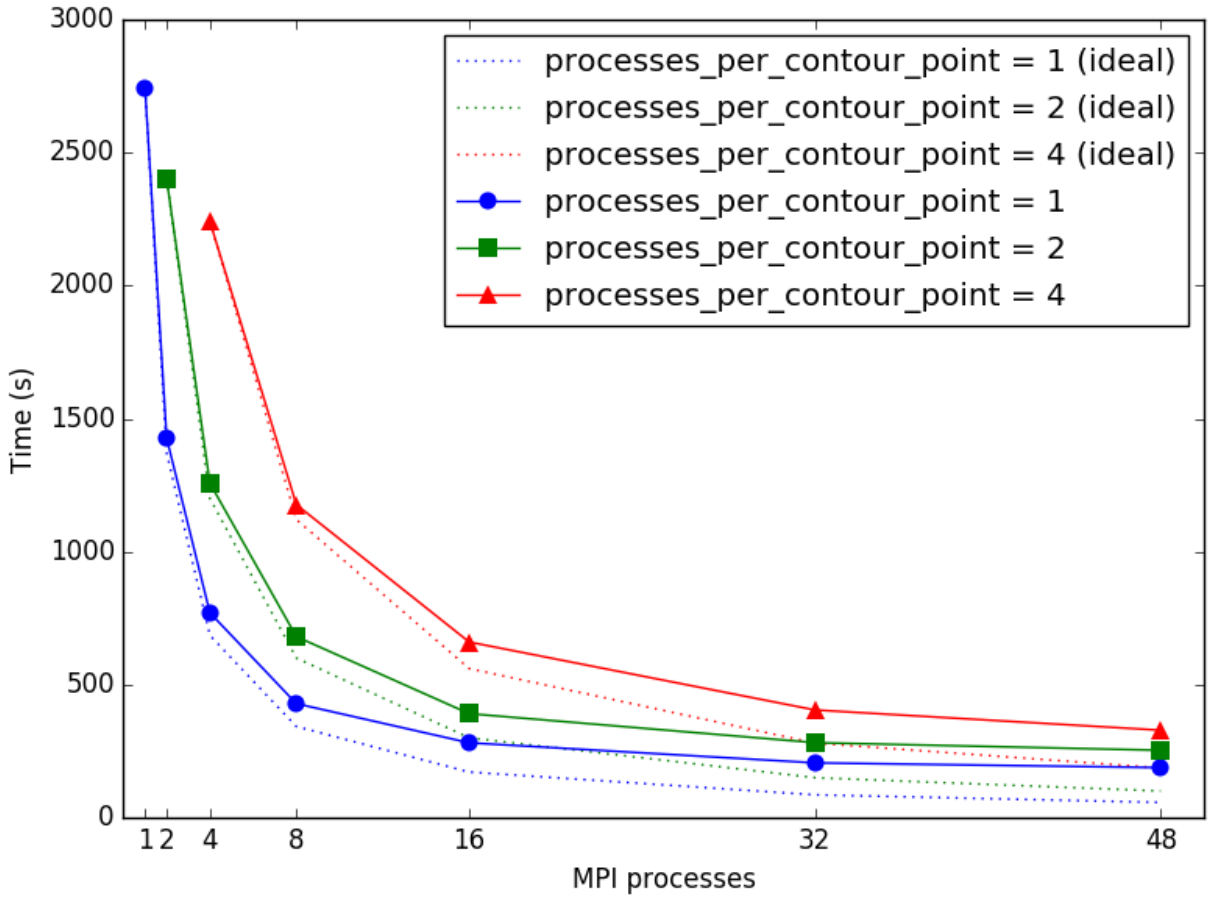


Fig. 4.72: Solid lines: wallclock time when parallelizing a NEGF calculation using MPI, for different values of the option `processes_per_contour_point`. Dashed lines: ideal scaling of the NEGF calculation based on the time obtained for the minimal number of processes for each value of `processes_per_contour_point`.

The number of processes N_{MPI} has been chosen to be always a divisors of 480, to avoid the presence of idle processes during the calculation. It can be seen that the speedup is close ideal for all the different number of processes considered. As expected, the total walltime for `processes_per_contour_point = 2` (`processes_per_contour_point = 4`) is around two (four) times larger that for `processes_per_contour_point = 1` using the same number of processes, because 2 (4) blocks of 240 (120) units of work are solved stepwise. However, the memory available per unit of work is two (four) times larger for `processes_per_contour_point = 2` (`processes_per_contour_point = 4`) compared to `processes_per_contour_point = 1`.

[1] D.ăStradi, U.ăMartinez, A.ăBlom, M.ăBrandbyge, and K.ăStokbro. General atomistic approach for modeling metal-semiconductor interfaces using density functional theory and nonequilibrium Greens function. *Phys. Rev. B*, 93:155302, Apr 2016. doi:10.1103/PhysRevB.93.155302.

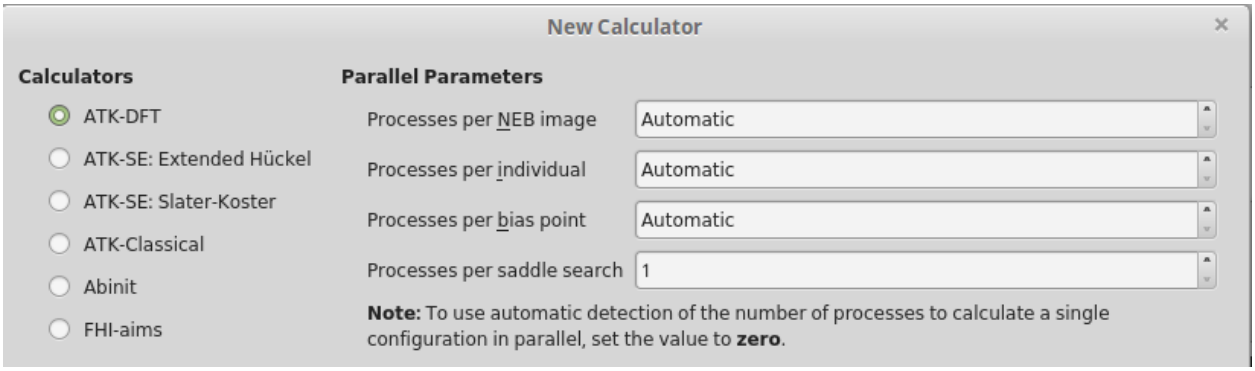
Examples of multi-level parallelisms in QuantumATK

In the preceding sections, we have discussed the basics of how MPI parallelism can be used to distribute the workload of individual QuantumATK calculations for bulks and devices. However, QuantumATK also offers advanced multi-level parallelism, where the workload of **composite calculations** may be distributed from the highest level of a hierarchic workflow all the way down to the lowest level, where only one unit-of-work is left.

Such composite calculations include nudged elastic band (NEB) calculations, IV-curve calculations, adaptive kinetic Monte Carlo (AKMC) simulations, and Crystal Structure Prediction using a genetic algorithm, amongst others.

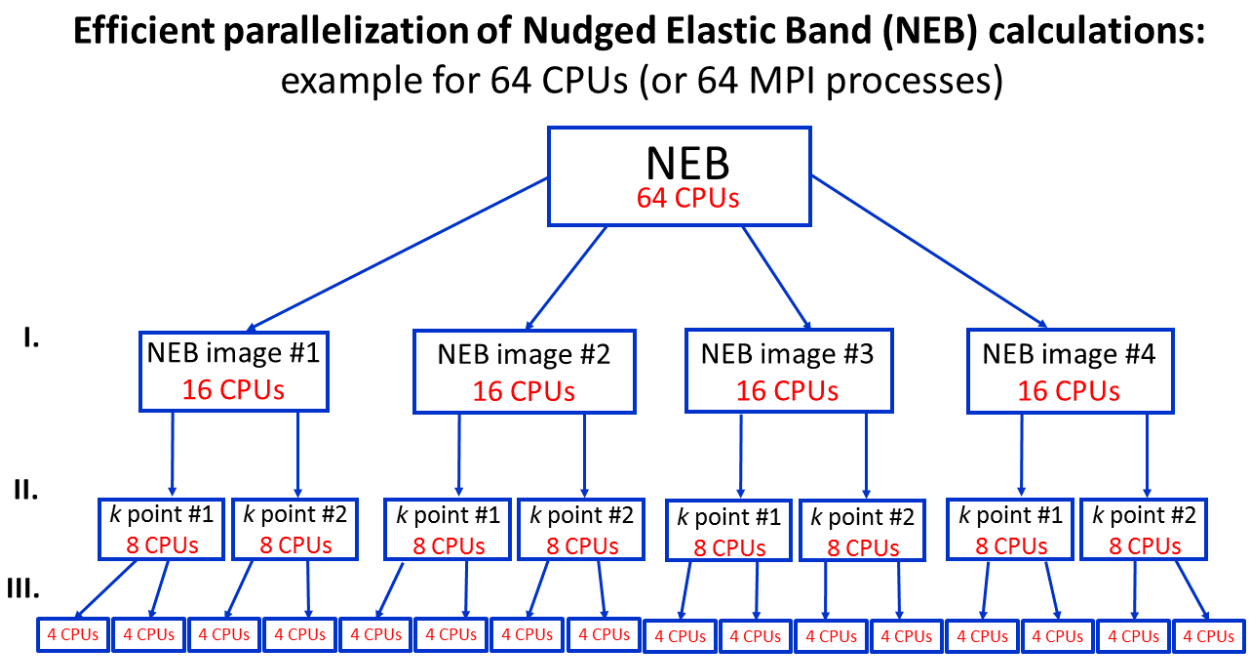
All such methods require a workflow involving calculations for several different structures at the same time, and often quite many of them. Multi-level parallelization makes it possible to run several different calculations for several different structures **at the same time**, thereby reducing the time-to-result.

Multi-level parallelism is of course **automatically enabled** if the number of assigned MPI processes is sufficient. However, the **Parallel Parameters** calculator settings may be used to manually set the desired scheme for multi-level distribution of the workload.



Nudged elastic band calculations

Consider first an example of QuantumATK multi-level parallelization of NEB calculations, here using 64 MPI processes in total. The calculation for each of the 4 NEB images has 2 irreducible k-points. The figure below shows how the full NEB calculation is parallelized over first images and then over k-points for each image.



•**Parallelization level I**: 64 CPUs are distributed over 4 groups with 16 CPUs for each group, where each group is assigned to a single NEB image. Setting the number of CPUs per group is automatically done by QuantumATK.

If needed, you may explicitly set the number of MPI processes per NEB image in the QuantumATK Scripter (Processes per NEB image), or set it in the script as follows: `parallel_parameters = ParallelParameters(processes_per_neb_image=16)`.

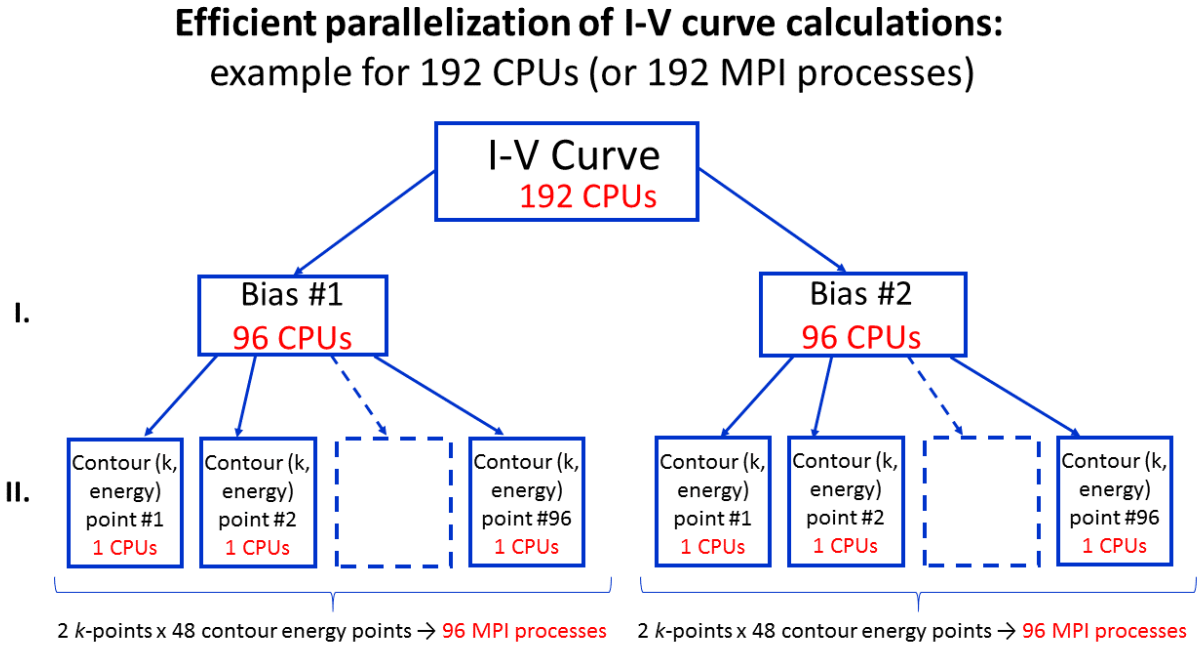
•**Parallelization level II**: If the QuantumATK calculator for the NEB calculation uses N_k k-points for the Brillouin zone integration, e.g., $N_k = 2$, each image group can be divided into 2 groups with 8 CPUs per k-point. QuantumATK automatically sets this parallelization.

If needed, you may also set the number of MPI processes per k-point manually in the QuantumATK Scripter (*New Calculator* → *Algorithm Parameters* → *DiagonalizationSolver Parameters*).

•**Parallelization level III**: QuantumATK automatically sets this parallelization.

I-V curve calculations

Computing an I-V curve requires a number of NEGF device calculations at different biases. Multi-level parallelism is therefore used to distribute the workload over bias points and NEGF contour points for each bias.



•**Parallelization level I**: 192 CPUs are distributed over 2 groups with 96 CPUs per group, where each group is assigned to a single bias calculation. Setting the number of CPUs per group is automatically done by QuantumATK.

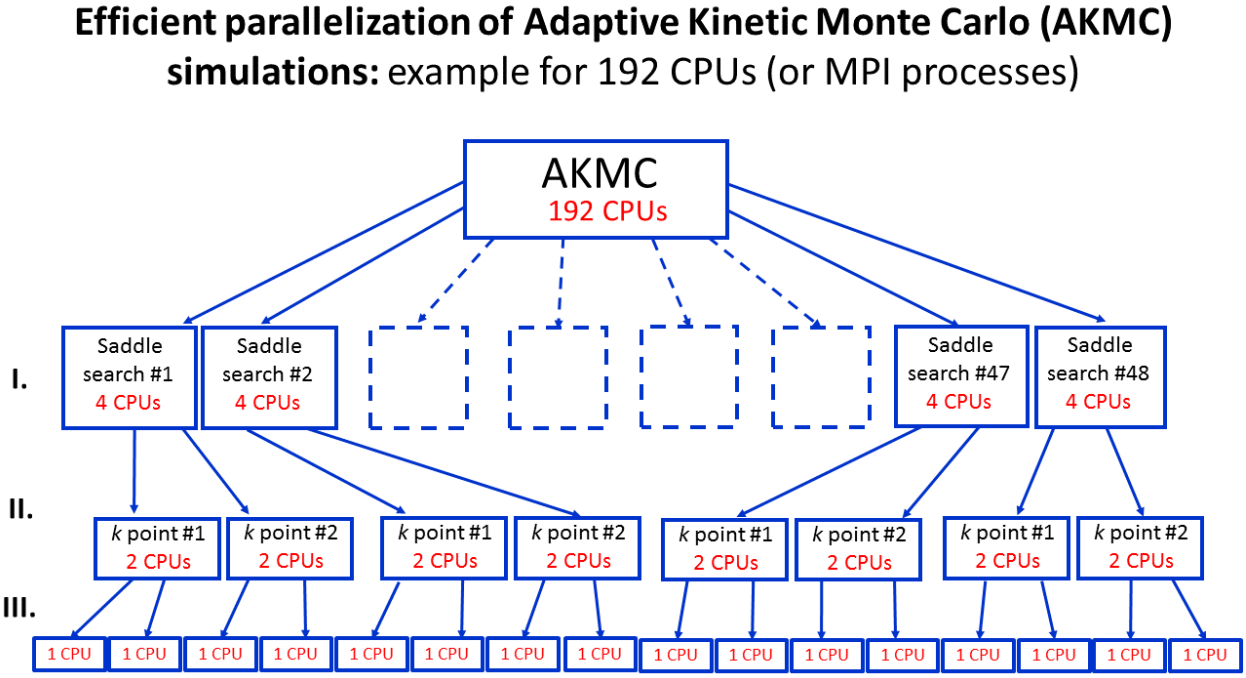
If needed, you may explicitly set the number of MPI processes per bias point in the QuantumATK Scripter (Processes per bias point), or set it in the script as follows: `parallel_parameters = ParallelParameters(processes_per_bias_point=96)`.

•**Parallelization level II**: If the QuantumATK calculator for the I-V Curve calculation uses $N_e = 48$ energy points for the NEGF contour integration, and $N_k = 2$ k-points for the 2D Brillouin zone integration at a given energy, each bias group can be divided into 96 groups with 1 CPU per contour (k, energy) point. QuantumATK automatically sets this parallelization.

If needed, you may also set the number of MPI processes per contour point manually in the script as follows: `parallel_parameters = ParallelParameters(processes_per_bias_point=96, processes_per_contour_point=1)`.

Adaptive Kinetic Monte Carlo simulations

The AKMC method is an obvious candidate for multi-level parallelization. A number of saddle searches must be carried out, and QuantumATK will by default try to fully distribute the workload. We will here consider AKMC calculations using ATK-DFT, as efficient parallelization is most important in this case, though AKMC with ATK-ForceField also benefits from parallelization.



•**Parallelization level I**: 192 CPUs are distributed over 48 groups with 4 CPU per group, where each group is assigned to a single saddle point search. Setting the number of CPUs per group is automatically done by QuantumATK.

If needed, you may explicitly set the number of MPI processes per saddle search in the QuantumATK Scripter (Processes per saddle search), or set it in the script as follows: `parallel_parameters = ParallelParameters(processes_per_saddle_search=4)`.

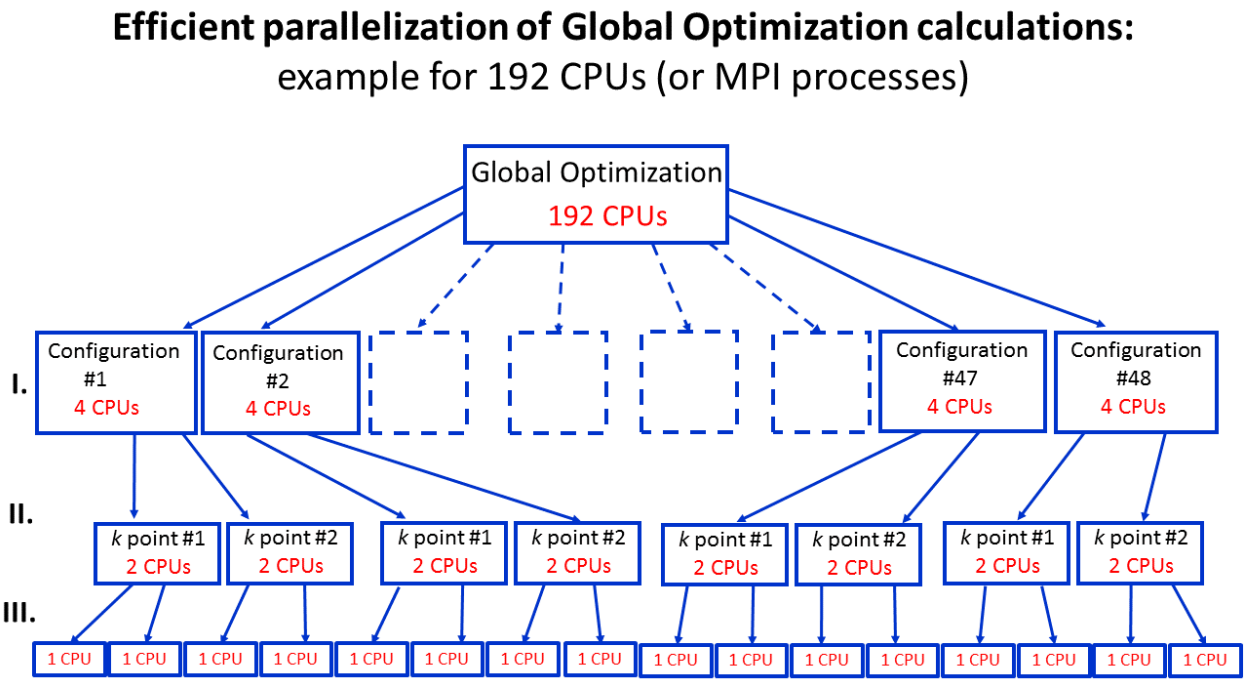
•**Parallelization level II**: If the QuantumATK calculator for the AKMC calculation uses N_k k-points for the Brillouin zone integration, e.g., $N_k = 2$, each image group can be divided into 2 groups with 2 CPUs per k-point. QuantumATK automatically sets this parallelization.

If needed, you may also set the number of MPI processes per k-point manually in the QuantumATK Scripter (*New Calculator* → *Algorithm Parameters* → *DiagonalizationSolver Parameters*).

•**Parallelization level III**: QuantumATK automatically sets this parallelization.

Crystal Structure Prediction

A genetic algorithm relies on continuously evolving and improving improving generations of candidate minimum-energy configurations. Each generation consists of several (sometimes hundreds or thousands) of different configurations that all need to be geometry optimized. Multi-level parallelism will therefore give a significant speed-up for such calculations in most cases.



•**Parallelization level I**: 192 CPUs are distributed over 48 groups with 4 CPUs per group, where each group is assigned to an individual configuration calculation. Setting the number of CPUs per group is automatically done by QuantumATK.

If needed, you may explicitly set the number of MPI processes per individual configuration in the QuantumATK Scripter (Processes per individual), or set it in the script as follows: `parallel_parameters = ParallelParameters(processes_per_individual=4)`.

•**Parallelization level II**: If the QuantumATK calculator for the Crystal Structure Prediction calculation uses N_k k-points for the Brillouin zone integration, e.g., $N_k = 2$, each image group can be divided into 2 groups with 2 CPUs per k-point. QuantumATK automatically sets this parallelization.

If needed, you may also set the number of MPI processes per k-point manually in the QuantumATK Scripter (*New Calculator* → *Algorithm Parameters* → *DiagonalizationSolver Parameters*).

•**Parallelization level III**: QuantumATK automatically sets this parallelization.

References

4.5.11 GPU acceleration of proprietary feature calculations

DFT-LCAO, Semi Empirical, and Force Field calculations can be accelerated using GPU. Currently, NVIDIA GPUs are supported, for detailed requirements refer to the System Requirements.

Note: GPU acceleration of DFT-LCAO, Semi Empirical, and Force Field calculations is NOT supported on Windows. To take advantage of GPU acceleration, you must use the Linux version of QuantumATK.

This document describes how to run with GPU acceleration, which features are supported, and provide some guidance to attain best performance.

Enabling GPU acceleration

GPU acceleration of DFT-LCAO, Semi Empirical, and Force Field calculations is protected by a license key and it is not enabled by default.

There are alternative ways to request GPU acceleration:

- Use the **Job Manager** and check the Enable proprietary GPU acceleration option in the Job settings, or when submitting your job.
- Run the script (or interactive session) using the `atkpython_gpu` command instead of `atkpython`.
- Set the environment variable `QUANTUMATK_ENABLE_GPU_ACC=1` before running `atkpython`.

When GPU acceleration is requested the corresponding license will be checked out at startup. If the license key is not valid or the GPU is not supported, the program will exit with an error message. If GPU acceleration is enabled successfully, the log will display a message:

```
+-------------------------------------------------------------------------------+
|                                                                               |
| GPU Acceleration                                                              |
|                                                                               |
+-------------------------------------------------------------------------------+
| Compatible GPUs have been detected and proprietary CUDA acceleration has      |
| been enabled. All GPU features are supported.                                 |
+-------------------------------------------------------------------------------+
```

If compatible GPUs are detected but GPU acceleration was not requested, a warning is shown and DFT-LCAO, Semi Empirical, and Force Field calculations calculations will run on CPU.

Note: At startup a number of GPU licenses equal to the number of GPUs requested on the system by the user. The number of GPUs is determined by their unique IDs. If MIG (Multi Instance GPU) is enabled, each instance will be counted as a separate GPU.

LCAOCalculator and SemiEmpiricalCalculator

LCAOCalculator and *SemiEmpiricalCalculator* update is accelerated when the density matrix is calculated by direct diagonalization (default, see *DiagonalizationSolver*). Both non-distributed (*processes_per_kpoint=1*) and distributed (*processes_per_kpoint>1*) calculations are supported.

The speedup is more visible for calculation where most of time is spent in the diagonalization step, that is larger systems (100 atoms and above) and/or larger basis, Semi Empirical or DFT calculations with local functionals (LDA or GGA).

Hybrid functionals have a significant overhead and are only partially accelerated, therefore the speedup will be more moderate. *ADMM projections* are accelerated and can speed up significantly calculations for large systems.

There are no restrictions on the spin type, the GPU acceleration works for un-polarized, polarized, non-collinear and spin-orbit calculations.

Bulk Analysis

Bulk analysis objects which rely on direct or iterative diagonalization are accelerated, including:

- *Bandstructure*
- *FatBandstructure*
- *ProjectedDensityOfStates*
- *DensityOfStates*
- *Eigenvalues*
- *BlochState*

Additionally, analysis object based on poisson solvers *ParallelConjugateGradientSolver* and *NonuniformGridConjugateGradientSolver* will also be accelerated, see *dedicated section*.

Other

All more complex workflows which requires frequent or heavy calculator update will be accelerated as a consequence of the calculator acceleration. This includes, among others:

- Geometry optimization
- Molecular dynamics
- Nudged Elastic Band (NEB)
- Defect calculations

DeviceLCAOCalculator and DeviceSemiEmpiricalCalculator

DeviceLCAOCalculator and *DeviceSemiEmpiricalCalculator* update is accelerated when *GreensFunction* method is selected as *equilibrium_method* or *non_equilibrium_method* (both, for best performance), and *RecursionSelfEnergy* as *self_energy_calculator_real* or *self_energy_calculator_complex* (both, for best performance). These are also the default settings, meaning that a default DeviceLCAOCalculator and DeviceSemiEmpiricalCalculator will use GPU acceleration.

Note: Only the serial version of *GreensFunction* is accelerated (*processes_per_contour_point=1*).

There are no restrictions on the spin type, the GPU acceleration works for un-polarized, polarized, non-collinear and spin-orbit calculations.

Device calculations can also use accelerated Poisson solvers, see *dedicated section*. In some cases, when the real space grid is fine and a large vacuum is present, that is for 1D and 2D systems, the Poisson solver can actually become the bottleneck, and the GPU acceleration of the Poisson solver can make a large difference.

The speedup is larger for systems with a large-cross section area, i.e. large number of atoms in the transverse directions (50 atoms or more) and/or large basis. More specifically, the speedup relates to the size of the blocks in the tridiagonal inversion algorithm[1]. For DFT calculations, the speedup is affected moderately by the choice of exchange-correlation functional. Hybrid functionals can exhibit a large speedup, because they lead to larger blocks in the tridiagonal structure.

[1] D.ăE. Petersen, H.ăH.ăB. Sorensen, P.ăC. Hansen, S.ăSkelboe, and K.ăStokbro. Block Tridiagonal Matrix Inversion and Fast Transmission Calculations. *J. Comput. Phys.*, 227(6):3174–3190, 2008. doi:10.1016/j.jcp.2007.11.035.

Device Analysis

The most common device analysis objects are accelerated, including:

- *TransmissionSpectrum*
- *ProjectedLocalDensityOfStates*
- *LocalDeviceDensityOfStates*
- *DeviceDensityOfStates*

Additionally, analysis objects based on poisson solvers *ParallelConjugateGradientSolver* and *NonuniformGridConjugateGradientSolver* will also be accelerated.

IVCharacteristics

Since usually the most time consuming tasks of an IVCharacteristic calculation are the calculator self-consistent update and the transmission spectrum calculation, both of which are accelerated, the IVCharacteristic object can also show large speedup.

Poisson solvers

ParallelConjugateGradientSolver and *NonuniformGridConjugateGradientSolver* are GPU accelerated and can exhibit large speedup, especially DFT calculations which typically have many grid points. When either is used in a bulk or device calculator, the following analysis objects, which use the Poisson solver, are also accelerated:

- *HartreeDifferencePotential*
- *HartreePotential*
- *ElectronDifferenceDensity*
- *ElectronDensity*
- *ElectrostaticDifferencePotential*
- *ElectrostaticPotential*

Note that the *FastFourierSolver* and *FastFourier2DSolver* can be faster regardless of CUDA acceleration. However, *ParallelConjugateGradientSolver* and *NonuniformGridConjugateGradientSolver* support more boundary conditions, metallic and dielectric regions and are therefore commonly used for device simulations.

Force Field

Force Field simulations can be accelerated if the potential set contains at least one potential type which supports GPU acceleration. Potential types that can be accelerated are:

- Long-range pair potentials
- All coulomb solvers (real- and fourier-space)
- Machine-learned Force Field (MTP and TorchX-based)

For details about GPU acceleration, please see the reference manual of the *TremoloXCalculator*.

Machine-learned Force Field Training

Training of machine-learned Force Field training using the *MachineLearnedForceFieldTrainer* can be accelerated with GPUs. This concerns primarily the training of MACE models based on the *MACEFittingParameters*.

Parallelization

GPU acceleration can be used together with MPI parallelization. There are no constraints on the number of MPI processes, and multi-node calculations are supported. The following recommendations apply:

- Use a number of CPU cores proportionate to the number of GPUs on the node. For example, if you have 4 GPUs and 32 CPU cores, use 8 CPU cores per GPU. Otherwise, parts of the code which are not GPU accelerated could be slowed down significantly.
- Using one MPI process per GPU, and threading to occupy the CPU cores, is a good starting point for most calculations. This is the setup which uses the least amount of GPU memory.
- Calculations with multiple MPI processes per GPU are supported, and in some cases they can be faster. Note also that more GPU memory will be used, roughly proportionally to the number of MPI processes. In this case it is **highly recommended to enable NVIDIA MPS (Multi Processing Service)**.

Performance tips for DFT and Semi-Empirical

- **System Size**: Larger systems will have a larger speedup when compared to CPU calculations. This is because the overhead of the GPU is better amortized, and a larger fraction of the time is spent in the algorithms that are accelerated.
- **Bulk Systems**:
  - Use GPU acceleration for systems of about 100 atoms or more.
  - For large systems, speedups of up to 10x can be achieved.
  - The non-distributed diagonalization solver (*processes_per_kpoint=1*) is more efficient and should be preferred if GPU memory is sufficient.
  - Typically, systems up to 1000 atoms or more can be run on a single GPU.
  - For moderate-sized systems (100-1000 atoms), better performance might be achieved by using multiple MPI processes per GPU, if memory allows.
- **Device Systems**:
  - Acceleration depends on the structure of the tri-diagonal matrix (the larger the matrix blocks, the better the speedup), which is determined by:
    - Number of atoms
    - Cross-section size
    - Interaction length of the basis-set (or hamiltonian parametrization for Semi Empirical calculations)
  - **Parallelization tradeoffs**: The optimal configuration depends on your specific system:
    - Medium-sized systems may benefit from multiple MPI processes per GPU for matrix operations
    - However, Poisson solvers achieve best speedup with a single process per GPU
    - Choose processes/threads based on whether your calculation is Poisson solver intensive or density matrix intensive
  - Use the *DevicePerformanceProfile* object to obtain an initial estimate of performance and memory consumption for the density matrix calculation step, which is usually the bottleneck.

Examples

Downloads

`Quartz-scf.py` `GaN-optimization.py` `High-K-stack.py`

This section collects examples with timing comparison between GPU accelerated calculations and their CPU counterpart. Note that performance will depend on the specific hardware and software configuration, as well as the size of the system being simulated.

DFT bulk calculation

In this example we update self-consistently a Quartz supercell with 864 atoms, calculated with DFT using a GGA-PBE functional, the default Medium basis and 8 k-points. For CPU calculations, the fastest diagonalization solver was selected (*optimize_for_speed_over_memory=True* in *DiagonalizationSolver*).

The number of processes was chosen to maximize the number of concurrent k-point:

For the CPU calculation:

•48 CPU cores: 8 processes 6 threads/process

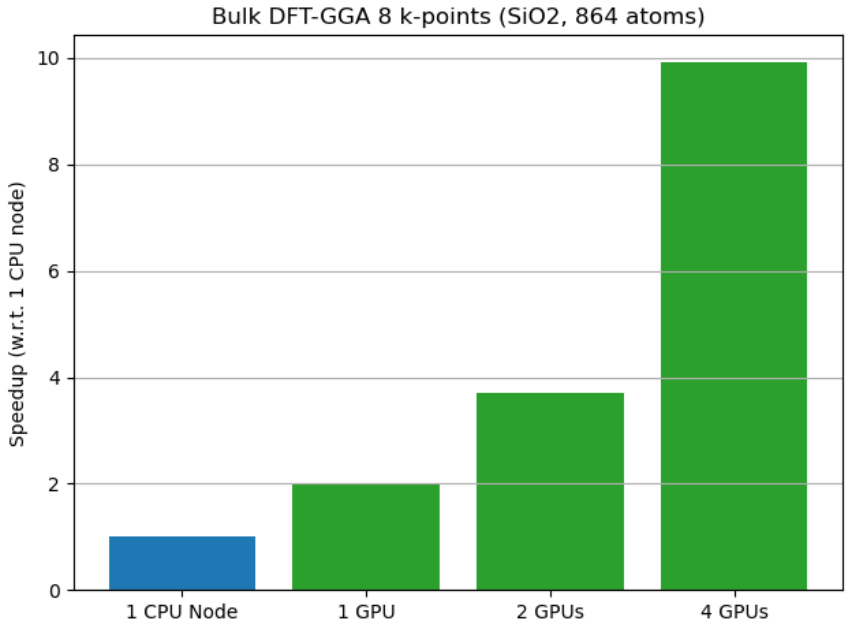
For the GPU calculation:

•1 GPU: 2 processes 6 threads/process (MPS enabled)

•2 GPU: 4 processes 6 threads/process (MPS enabled)

•4 GPU: 8 processes 6 threads/process (MPS enabled)

The results are shown in figure below: a single GPU is 2x faster as the full CPU node and **the full GPU node is 10x faster than the full CPU node.**



All calculations are executed on the same node (with GPU enabled or disabled), equipping 4 NVIDIA A100 with NVLink interconnect, and a two AMD EPYC 7F72 24-Core CPUs (48 cores total).

Geometry optimization

When running a geometry optimization, the largest computational cost is the calculator update. Therefore, the GPU speedup is close to what can be obtained for a self-consistent calculation.

This example is a geometry optimization of a GaN supercell with 768 atoms, using DFT with GGA-PBE functional, PseudoDojo High basis set and default k-point sampling. For CPU calculations, the fastest diagonalization solver was selected (*optimize_for_speed_over_memory=True* in *DiagonalizationSolver*).

The following table shows the total time for the CPU and GPU calculations.

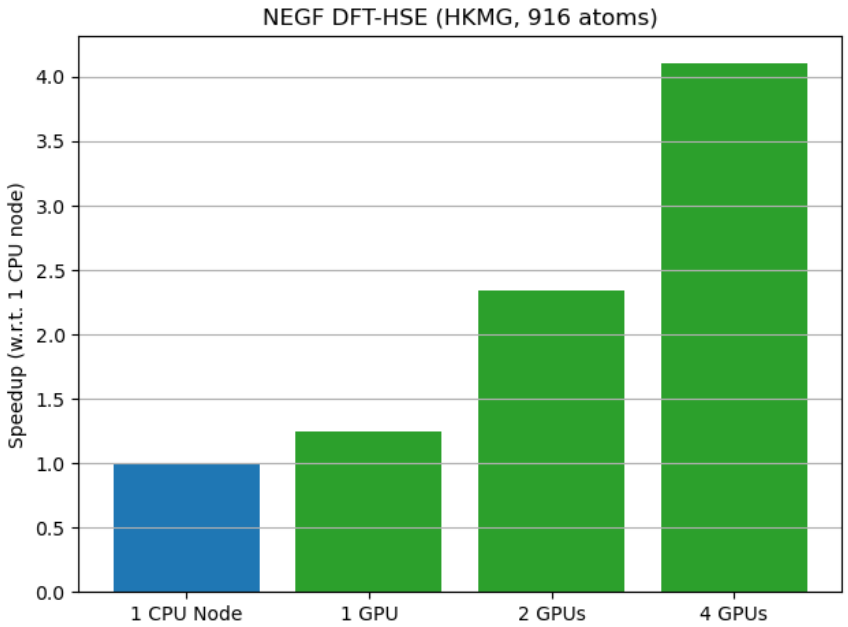
	Turnaround time	Speedup
CPU Node	1d 12h 39m	1
GPU Node	4h 44m	7.74

CPU calculations were performed on an Intel Xeon IceLake node with 56 cores, using 4 MPI processes and 14 threads per process. GPU calculations were performed on 4 NVIDIA A100.

Device calculation

This example is a NEGF calculation of a High-K metal gate stack, using DFT and a hybrid functional (HSE06). The evaluation of the density matrix via *GreensFunction* is accelerated. Since hybrid functionals lead to long range interactions and dense hamiltonian blocks in the transverse direction, the speedup can be significant even for systems with a relatively small cross-section in terms of number of atoms and basis function.

The results are shown in figure below: a single GPU is as fast as a full CPU node, for a system with 916 atoms.



CPU calculations were performed on an Intel Xeon IceLake node with 56 cores, using 8 MPI processes and 7 threads per process. GPU calculations were performed on 4 NVIDIA A100, with 2 MPI processes per GPU and MPS enabled.

For larger cross-sections, even larger speedups can be achieved. On a system with twice the extension in the transverse direction (3364 atoms), speedup up to 9x can be achieved. However, GPUs with large amount of memory (80GB or more, e.g. H100) are required to run such calculation.

4.5.12 Performance troubleshooting guide

Downloads & Links

ATK Reference Manual

In this tutorial, you will learn which are the main parameters to tune for improvimg **ATK** performance for various systems. In order to get the highest possible performance of your calculations, **ATK** provides an array of different algorithms and utilizes both MPI and shared-memory threading for parallelization. As already mentioned in the technical note *Parallelization of QuantumATK calculations*, MPI parallelization will ususlly give the best speedup of calculations, while threading of processes can often reduce the memory footprint.

Getting the absolute peak performance out of any high performance software requires somewhat deep knowledge of both the calculation in question, and the hardware used. It is therefore important to optimize your QuantumATK simulation with respect to hardware used and parallelization method. However, the following sections give some tips for QuantumATK algorithms and options that may also help in fixing issues with memory or speed.

Running out of memory?

General indications

- Try to disable **store_grids** in **AlgorithmParameters**, this will lower the memory slightly, and will reduce the calculation time very little.
- Disabling **store_basis_on_grid** in **AlgorithmParameters** will reduce the memory significant, and if the matrix size of the system is small, it can have negative effect on the performance. This parameter is by default turned off, if the configuration is large, so it might already be disabled. If running in **parallel**, the memory reduction will be lower per core, since the memory is distributed.
  - If running in **parallel** with more than two processers, try setting the **algorithm** in **IterationControlParameters** to **ParallelPulayMixer**.

Specific for molecules and bulk configurations

- If the matrix size is very large, consider setting **density_matrix_method** in **AlgorithmParameters** to **DiagonalizationSolver(processes_per_kpoint=2)**. This will reduce the memory usage per mpi-process by a factor of 2. Try to increase **processes_per_kpoint**, if the problem persist.
- If the calculation is very large, and the calculation is run in **parallel** over many nodes, it can be worth trying setting **density_matrix_method** in **AlgorithmParameters** to **ChebyshevExpansionSolver**. When run in massive parallel, this will reduced the memory usage dramatically, however it is also slow compared to ordinary routines.

Specific for device configurations

- Try setting **storage_strategy** in **SelfEnergyCalculator** to **StoreOnDisk** or **NoStorage**. This will reduce the memory significantly, but will also cost performance for the latter. If running in **parallel**, the memory reduction will be lower per core, since the memory is distributed.
- The device has a wide cross section, try setting **equilibrium_method** and/or **non_equilibrium_method** to **SparseGreensFunction**
- If run in **parallel**, setting **equilibrium_method** and/or **non_equilibrium_method** to **SparseGreensFunction(processes_per_contour_point=2)** will help. If the problem persist, try setting **processes_per_contour_point=4**.

Want to make it run faster?

General indications

- If the calculation requires the usage of the multigrid method, as in the presence of gates, it is possible to consider **DirectSolver** for the **possion_solver**. It will require significant more memory, but the overhead is memory distributed so if running in **parallel** it can be worth to try it.
- Set **store_basis_on_grid** in **AlgorithmParameters** to True. This will make the calculation run faster - and have moderate, but distributed memory overhead.

Specific for molecules and bulk configurations

In **ATK 2015** two new parameters have been introduced for **density_matrix_method** in **AlgorithmParameters**: **processes_per_kpoint** and **bands_above_fermi_level**.

```
algorithm_parameters = AlgorithmParameters(
    density_matrix_method=DiagonalizationSolver(
                                                bands_above_fermi_level=20,
                                                processes_per_kpoint=2
                                                ),
    )
```

- With **processes_per_kpoint** you can specify the number of mpi-processes to use per k-point allowing you for an extra level of parallelization.

  The next figures shows the time for a full SCF run for a InGaAs bulk configuration using 32 k-points. By default, you will be able to parallelized at most over 32 mpi-processes (red line). A quick way to check how many k-points are used in the calculation you can run the following script:

```
mk = MonkhorstPackGrid(4,4,4)
irriducible_kpoints = mk.kpoints()
print(len(irreducible_kpoints), ' irreducible kpoints')
```

  By specifying **processes_per_kpoint** you are now able to push your parallelizatio even further (blue line).

- The total number of bands used in the calculation is defined by the size of the basis set. By defaults all empty bands are included. However, including all these bands in the calculation does not improve the accuracy while it will slow down considerably you simulations. Set **bands_above_fermi_level** to specify the number of empty bands to be used. In the example above, you can see that using all bands (green point) will double the computational time.

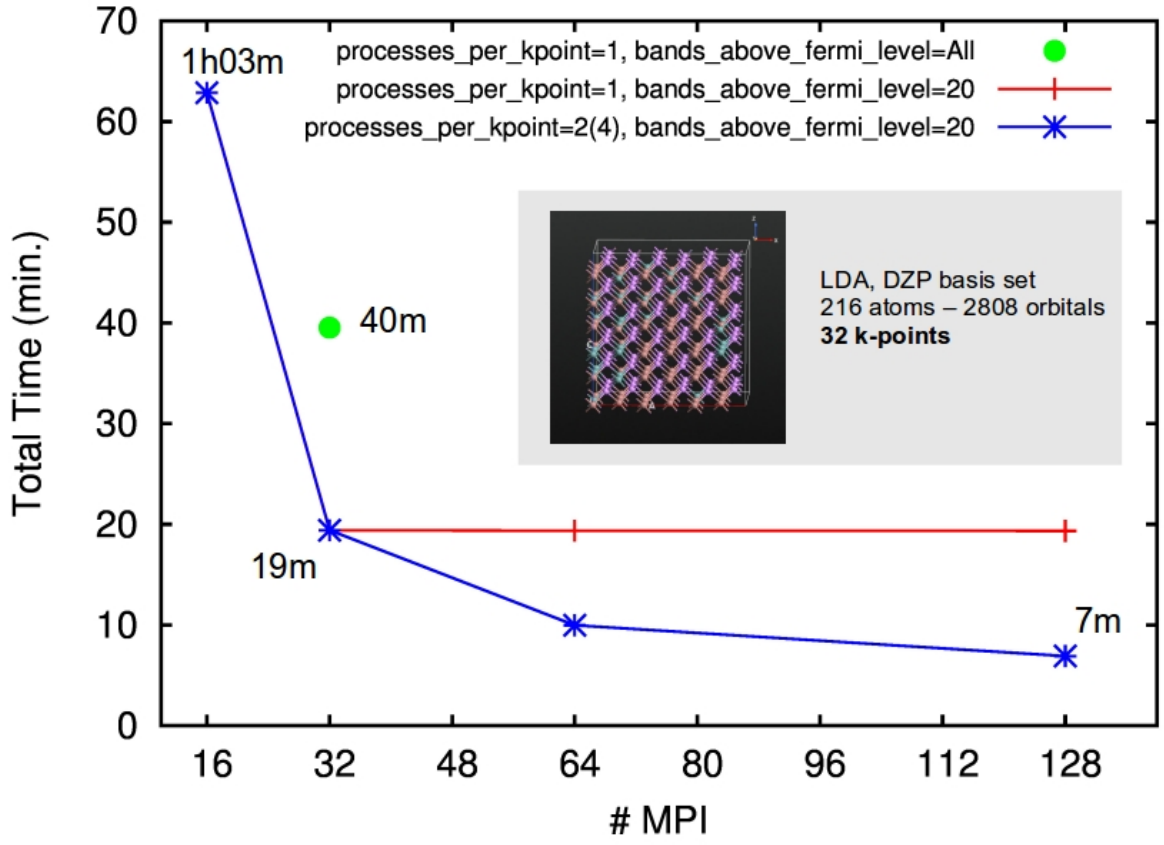


Fig. 4.73: Benchmark plot for InGaAs alloy run on Xeon E5-2687W @ 3.1GHz 16 cure/node cluster. Timing corresponds to a full SCF cycle (converged in 19 steps).

Note:

- The optimal number of empty bands to be used depends on several factor, including the electron temperature specified in the LCAO calculator.
- The performance improvement increase with the system size. For small systems, < 100 atoms, you can keep the default parameter.

Specific for device configurations

There are indeed many parameters that you can tune in order to get the optimal performance for a device configuration. Which are the parameters to tune? This really depends on the system you are investigating and the methods you are using.

4.5.13 Study objects

In *QuantumATK* version 2018, a new type of calculation object has been introduced: the study object. Study objects have been designed to facilitate both long-running calculations and complex studies of a particular type of system. The defining feature of a study object is that it is made up of a number of smaller interdependent tasks. The number and nature of the tasks (e.g., whether they are homogeneous or heterogeneous), and the dependencies between them, will depend on the particular type of study.

The study object can execute its tasks either in serial or in parallel. If the program is terminated before the calculation finishes (e.g., the requested wall time given to a job scheduler was insufficient) then it can be automatically resumed from the already completed tasks by running the same script again.

In order to support this resuming, the study object takes care of writing its results to disk on-the-fly. This means that all study objects require a `filename` and `object_id` (see *nlread* for more details). Since all the results are saved automatically, it is not necessary (or allowed) to call *nlsave* on a study object. Saved study objects can then be read normally with *nlread*.

In addition to providing support for resuming execution, study objects also allow for the user to modify and update existing calculations. When any existing study object is modified the previously calculated data is reused (when appropriate), which greatly improves performance. For example, the *IVCharacteristics* study object allows for additional voltage points to be added to a pre-existing calculation. This means that an initial rough calculation can be quickly refined.

Note: In contrast to *Analysis* objects, study objects do not perform any calculation when the object is constructed; instead, the objects `update` method must be called first. Similarly, when adding or modifying data points in a study object, `update` must be called before the new points will be calculated and the changes will be available for any post-processing methods.

For a full list of study objects see the *Study* section of the reference manual.

Restart example

As an example, consider this simple script `carbon-chain-ivcharacteristics.py`, which applies the *IVCharacteristics* study object to the model system of a carbon chain. It runs in a few minutes, but to show the restart capability of study objects, we purposefully interrupted it. The first log file therefore show the first 3 tasks being executed, as it was interrupted during the third task:

```
+-------------------------------------------------------------------------------+
| Executing task 1 / 22:                                                        |
|   Update configuration                                                        |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.0 V                                              |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.0_Volt.log                     |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Executing task 2 / 22:                                                        |
|   Calculate TransmissionSpectrum                                              |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.0 V                                               |
|   Right electrode voltage: 0.0 V                                              |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.0_Volt.log                     |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Executing task 3 / 22:                                                        |
|   Update configuration                                                        |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.1 V                                               |
|   Right electrode voltage: 0.0 V                                              |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.1_Volt.log                     |
+-------------------------------------------------------------------------------+
```

To continue the calculations, we simply execute the same script in the same directory, giving this information in the log-file:

```
+-------------------------------------------------------------------------------+
| Executing task 1 / 20:                                                        |
|   Update configuration                                                        |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.1 V                                               |
|   Right electrode voltage: 0.0 V                                              |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.1_Volt.log                     |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Executing task 2 / 20:                                                        |
|   Calculate TransmissionSpectrum                                              |
|   Gate voltage: 0.0 V                                                         |
|   Left electrode voltage: 0.1 V                                               |
|   Right electrode voltage: 0.0 V                                              |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.1_Volt.log                     |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
| Executing task 3 / 20:                                                        |
|   Update configuration                                                        |
```

(continues on next page)

(continued from previous page)

```
|   Gate voltage: 0.0 V                                                          |
|   Left electrode voltage: 0.2 V                                                |
|   Right electrode voltage: 0.0 V                                               |
|   Log to: ivcharacteristics_Vgs_0.0_Volt_Vds_0.2_Volt.log                      |
+--------------------------------------------------------------------------------+
```

Note how the total number of tasks has decreased by two, as the first two tasks were successfully completed in the first run, and QuantumATK simply starts from the task that was interrupted. In this case, we chose to interrupt it and restart it a second time, while letting the third run finish. The full log-files can be found here:

- carbon-chain-ivcharacteristics_run_1.log
- carbon-chain-ivcharacteristics_run_2.log
- carbon-chain-ivcharacteristics_run_3.log

4.5.14 Finding Transition States

Finding transition states and thus reaction barriers is a key application of in ab-initio atomistic modelling. Knowing the transition state allows the calculation of reaction rates and gives qualitative insight into the mechanism of chemical reactions, diffusion processes, and phase transitions. Unfortunately, obtaining tightly converged transition states can be challenging, and often requires a lot of computational resources and human expertise.

QuantumATK provides state-of-the-art algorithms and graphical tools to make the process of setting up, running, and analyzing transition state calculations as intuitive and efficient as possible. This document provides an overview of the available methods and gives recommendations on how to use them.

Nudged Elastic Band

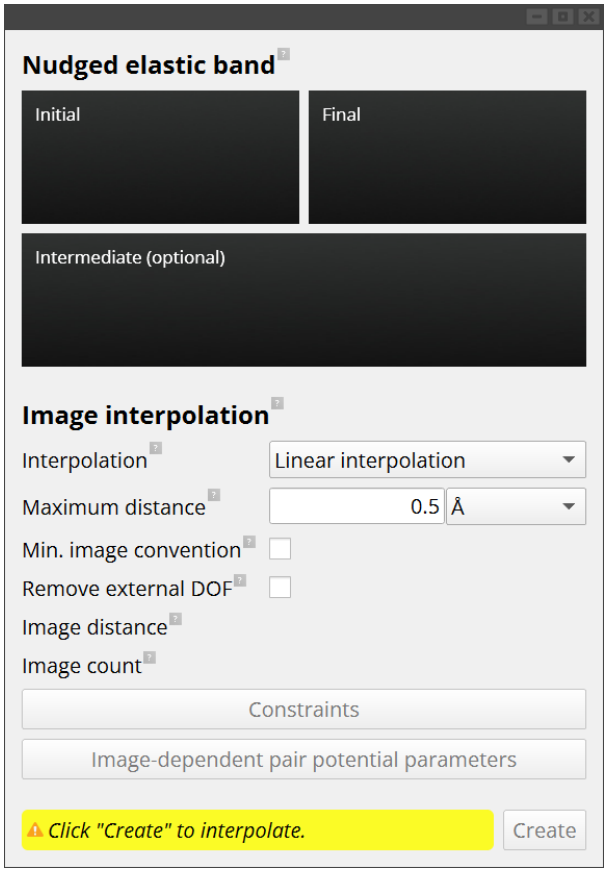
The *NudgedElasticBand* (NEB) method is a widely used technique to find transition states. It is a double-ended method, meaning that it requires the user to provide the initial and final states (i.e. the reactants and products) of the system. The NEB method then finds the minimum energy path (MEP) connecting these states with a chain of images. The images are relaxed to the MEP with the highest energy image being usually a good approximation of the transition state.

This approximate transition state can then be used as a starting point for more accurate or more efficient methods to refine towards the actual transition state (see *Minimum mode following techniques*).

Setting up the NEB Configuration

The easiest way to set up an NEB configuration is to use Nudged Elastic Band builder in NanoLab.

The initial and final states as well as one optional intermediate state can be dropped into the builder from the stash. The NEB can be created with the click of a button. If necessary, some parameters can be adjusted to change the quality of the initial path. The *ImageDependentPairPotential* is a good general purpose interpolation method, while the *SequentialImageDependentPairPotential* can be used for more complex reactions. If neither of these converge to an acceptable path, the *LinearInterpolation* is a fallback option. Remove external degree of freedom to minimize rotations and translations of molecules. The minimum image convention is useful to account for periodic cells.



Optimizing the Path

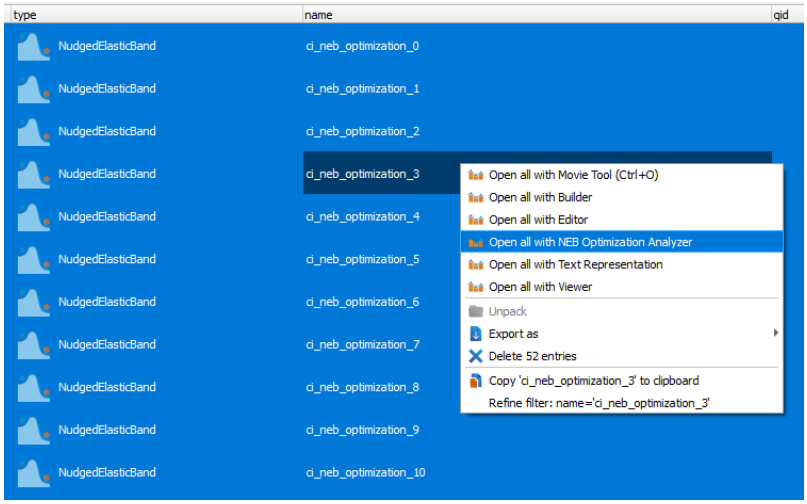
Optimizing the path is done using the *OptimizeNudgedElasticBand* function which implements the *all-images-optimized* algorithm where the whole path is optimized at once. This means that the highest parallel efficiency is achieved when the number of images is equal to the number of processes. Please refer to the documentation of the function for more details.

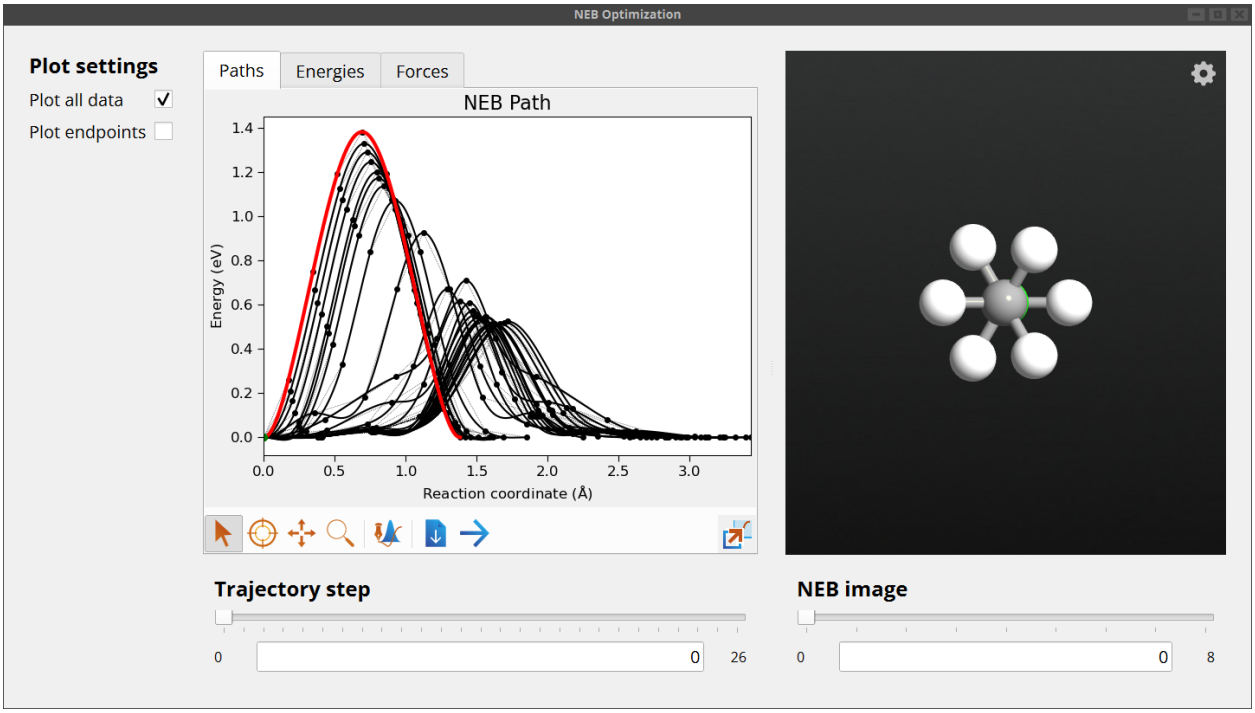
The *AutomaticNEB* method is an alternative to the regular NEB optimization which is more efficient at finding the transition state, as it optimizes the path segment by segment. This is especially useful for users who are not able to run many processes in parallel. One downside is that the path is not fully converged, except for the region around the transition state which is refined at the end of the calculation.

NEB Optimization Analyzer

The NEB optimization analyzer is opened from the Data View by selecting multiple NEB objects and right-clicking on one of them. Then select Open all with NEB Optimization Analyzer.

The analyzer shows all the relevant information, such as the energy and forces of each image throughout the optimization at a glance. Different tabs are available to drill down into the details of the optimization.


type
name
qid
NudgedElasticBand
ci_neb_optimization_0
ci_neb_optimization_1
ci_neb_optimization_2
ci_neb_optimization_3
ci_neb_optimization_4
ci_neb_optimization_5
ci_neb_optimization_6
ci_neb_optimization_7
ci_neb_optimization_8
ci_neb_optimization_9
ci_neb_optimization_10
Open all with Movie Tool (Ctrl+O)
Open all with Builder
Open all with Editor
Open all with NEB Optimization Analyzer
Open all with Text Representation
Open all with Viewer
Unpack
Export as
Delete 52 entries
Copy 'ci_neb_optimization_3' to clipboard
Refine filter: name='ci_neb_optimization_3'


NEB Optimization
Plot settings
Plot all data
Plot endpoints
Paths
Energies
Forces
NEB Path
Energy (eV)
Reaction coordinate (Å)
Trajectory step
NEB image

The analyzer is useful for monitoring the convergence of the NEB optimization and for troubleshooting problems. Some common issues to look out for are:

- Additional minima appearing in the path. It is usually a good idea to separate the path into segments and optimize them individually. For this the `split` method of *NudgedElasticBand* can be used. It takes the indices of the images at which to split the path.
- All images are not converging and forces remain high. This is often accompanied by the reaction coordinate becoming longer or energies of images crossing over. Refer to the troubleshooting section of *OptimizeNudgedElasticBand()* for more information.
- If only a single image is not converging, one might try to switch from NEB to a different method (see *Minimum mode following techniques*).

Minimum Mode Following Techniques

Minimum mode following techniques are a class of methods that start from a single configuration and follow the smallest eigenmode to search for nearby transition states. This can be advantageous in cases where the initial and final states are not known. Another common application is to refine the transition state found by the NEB method.

Two popular minimum mode following techniques are the *Dimer* and the *Lanczos* method. The Dimer method uses a finite difference approximation of two images (hence the name) displaced from the current configuration along the eigenmode and rotations around the axis of displacement to estimate the smallest eigenmode. In this way it avoids the expensive calculation of the Hessian matrix. The Lanczos method is conceptually similar to the Dimer method and differs mainly in the way the smallest eigenmode is estimated. It uses an approximate Hessian that is systematically built up during the optimization.

As both methods use finite differences they can be sensitive to noise in the forces. It is recommended to use a force field with smooth forces or DFT calculators with a tighter SCF tolerance. Generally speaking, both methods have similar performance characteristics overall[1] and are more efficient than the NEB at locating an exact transition state when more than 5-6 images are used for the NEB.

Best Practices and Recommendations

Transition states often require a tighter force tolerance than minima to satisfy the condition of exactly one imaginary frequency. A force tolerance of 10**-3 ev/Ang is usually sufficient, but it is good practice to calculate the vibrational spectrum using the *VibrationalMode* analysis object. This additional step is crucial to get accurate barrier heights since errors scale quadratically.

Usually, finding a transition state is a multi-step process involving several of the methods listed above. A common workflow might look like this:

1. Start with pre-converging the NEB using the default force tolerance and, optionally, a larger number of images.
2. Re-run the NEB optimization with a tighter force tolerance and fewer images while turning on the climbing image option. This will push the highest energy image towards the transition state. If the reaction is relatively simple, steps 1 and 2 may be combined.
3. Refine the transition state using the Dimer or Lanczos method with the final force tolerance. Here, the initial search direction can also be extracted from the NEB path using *dimerFromNEB*.

[1] R.ăA. Olsen, G.ăJ. Kroes, G.ăHenkelman, A.ăArnaldsson, and H.ăJónsson. Comparison of methods for finding saddle points without knowledge of the final states. *The Journal of Chemical Physics*, 121(20):9776–9792, 11 2004. URL: https://doi.org/10.1063/1.1809574, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/121/20/9776/19220457/9776_1_online.pdf, doi:10.1063/1.1809574.

A convenient way to automate the search for transition states is to use the *automaticTransitionState* function.

It is generally advisable to use an odd number of images to ensure that the transition state is in the middle of the path.

References

4.5.15 Molecular Dynamics

Introduction

QuantumATK makes Molecular Dynamics (MD) simulations very simple: Just add the desired calculator and a Molecular Dynamics block to your configuration and start the simulation! There are, however, some guidelines regarding which parameters and settings are the most suitable ones for different types of simulations.

MD simulation is a technique to simulate the motion of atoms and molecules under predefined conditions, such as temperature, pressure, stress, external forces, etc. MD simulations can therefore be used to study dynamical processes at the nanoscale and to calculate a broad range of properties, e.g. phase diagrams, diffusion coefficients, or various response functions, as well as static quantities such as radial distribution functions, coordination numbers, elastic moduli, etc.

Methodology

MD essentially utilizes the numerical solution of Newtons equations of motion for a set of atoms from a given initial configuration. This is commonly achieved via numerical integration by discretizing time into small intervals called the time step using integrators such as the velocity verlet algorithm[1]. The interactions between the atoms, i.e. the interatomic forces, can be calculated based on various methods, ranging from Density Functional Theory (DFT) to classical potentials. These forces determine the acceleration of the atoms and allow the positions and velocities to propagate towards the next time step. Repeating this procedure many times yields a series of snapshots, describing the trajectory of the system in phase space, which can be analyzed to extract the desired properties.

Before setting up the simulation you need to decide what type of calculation you are interested in. Should the total energy be conserved, as in an isolated system? Should the temperature be kept constant to mimic the coupling of the system to a heat bath? Is the system exposed to any external pressure or stress? Based on these considerations, a suitable set of simulation parameters should be selected: Time step size, number of integration steps (duration of simulation), integration algorithm, initial temperature, constraints, etc.

Similar to a real experiment, some empirical knowledge and experience is required for performing MD simulations. In these notes, you will get to know the basic ingredients of MD simulations for running MD simulations with the QuantumATK MolecularDynamics module. A tutorial to try out MD can be found in this link: *How to Setup Basic Molecular Dynamics Simulations*.

NVE Simulations

As mentioned previously, MD simulations are based on the solution of Newtons equations of motion. In the purest form, an MD simulation reproduces an NVE ensemble (also called the microcanonical ensemble) where the number of atoms N, the volume V, and the total energy E are conserved. These conditions correspond to a completely isolated system.

[1] W. C. Swope, H.C. Andersen, P.H. Berens, K.R. Wilson: A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. J. Chem. Phys. 76, 637 (1982)

Setting Up the Geometry

The primitive unit cell of the material to be simulated could have non-orthogonal cell vectors. Although it is possible to perform MD simulation on such cells, it is often more convenient to use orthogonal cells (in particular when the cell size is supposed to change during the simulation).

Furthermore, it is useful to increase the size of the structure to include more atoms. This generally improves the average of the measured observables and reduces finite size effects due to interaction of atoms in a small simulation cell with their periodic images. (Ideally, the lengths of the cell vectors should be larger than twice the interaction range of the potential).

Setting up the Calculation Script

In the `MolecularDynamics`, the value of `Log Interval` determines how often the snapshots of the current configuration are written to disk. A too small value can decrease the speed of the simulation and, in the case of large systems, produce huge output files.

Tip: In most MD applications it is not necessary to record snapshots at a high frequency, unless you explicitly want to analyze high-frequency oscillations.

Setting `Initial Velocity` type to *Maxwell-Boltzmann* at a `Temperature` of T Kelvin will create the random initial velocities of the atoms drawn from a Maxwell-Boltzmann distribution corresponding to a temperature of T Kelvin. It is also useful to remove the centre of mass motion of the cell. The `Time step` is a crucial parameter in MD simulations as it determines the accuracy and efficiency of the numerical integration scheme.

The central point is that a larger time step, although it may at first glance improve the efficiency of the simulation, increases the error in the numerical integration scheme of the equations of motion. The underlying assumption that the atomic forces are approximately constant during one integration step is not valid any more. Essentially, the chosen time step should be small enough to resolve the highest vibrational frequencies of the atoms (i.e. it should be much smaller than the smallest vibrational period), so if you have light atoms (e.g. hydrogen), you will generally be required to use a smaller time step than if you have only heavy atoms (such as gold). A smaller time step size may also be necessary if you have different elements in your calculation, if the temperature is high, or if the atoms are far away from their equilibrium configuration, i.e if large forces act on the particles. For most systems a safe choice to start with, if you do not know what time step to use, is 1 fs. Larger time step values can then be assessed by monitoring the conservation of the total energy in an NVE simulation under the conditions of interest.

Another very important aspect of performing MD simulations is to be aware that it may take some time before a system is equilibrated to the chosen external temperature, pressure, etc. This is important because any measurement of observables should only be carried out after the system is equilibrated (unless you are specifically interested in non-equilibrium phenomena). In general, equilibration times of MD simulations can vary over a wide range up to several hundreds of nanoseconds (e.g. in biological molecules or polymer systems) and depend essentially on the longest relaxation time present in the system. You should therefore always check carefully that the observables of interest have reached a stationary state in your simulations.

NVT Simulations

The *NVT* ensemble is also called the canonical ensemble and the main difference to the microcanonical NVE ensemble is that the system is not isolated any more but can exchange energy with a surrounding virtual heat bath.

Setting Up the Script

QuantumATK offers four possibilities for NVT simulations: 1) NVT Berendsen[2] , 2) NVT Nose Hoover[3] , 3) Langevin[4] and 4) NVT Bussi Donadio Parinello[5]. Each of them implements the coupling to the heat bath in a different way. Although the best choice depends on the system and the particular application at hand, the *Nose Hoover* algorithm can generally be considered the most reliable method for this task.

NVT Nose Hoover Simulations

The `Thermostat timescale` value determines how quickly the system temperature approaches the reservoir temperature. The `Heating rate` parameter can be used to linearly increase or decrease the temperature during the simulation (cf. *NVTNoseHoover* in the reference manual). The `Thermostat chain length` parameter determines how many subsequent thermostats are invoked to control the temperature. The default value of 3 should work well in most cases - only if you experience a strong, persistent oscillation in the temperature should this value be increased to achieve a more stable simulation.

As mentioned, how fast the temperature of the system approaches the target temperature can be controlled via the thermostat timescale parameter. In general, a small thermostat timescale parameter causes a tighter coupling to the virtual heat bath and a system temperature that closely follows the reservoir temperature. Such a tight thermostat coupling is typically accompanied by a more pronounced interference with the natural dynamics of the particles, however. Therefore, when aiming at a precise measurement of dynamical properties, such as diffusion or vibrations, you should use a larger value of the thermal coupling constant or consider running the simulation in the NVE ensemble to avoid artifacts of the thermostat.

NVT Berendsen and Langevin Simulations

Aside from the Nose-Hoover chain thermostat you can also select two other *NVT* algorithms, **NVT Berendsen** (cf. *NVTBerendsen* in the reference manual) and **Langevin** (cf. *Langevin* in the reference manual).

The **Berendsen** thermostat implements an algorithm which is in some situations more stable, as it effectively suppresses temperature oscillations. The drawback is that in contrast to the Nose-Hoover thermostat, it does not exactly reproduce a canonical ensemble, although the deviations are typically small. Effectively this means that observables (such as the velocity distribution) do not have the correct distribution. For this reason it should, if at all, only be used for equilibration simulations, primarily in cases where a more robust temperature control is required.

The third thermostat type that you can choose for NVT simulations is the **Langevin** algorithm. This algorithm solves the Langevin equation[Page 4370, 4], which explicitly includes friction as well as stochastic collision forces into Newtons equations of motion to mimic the interaction with particles of the heat bath.

Unlike the other thermostat types, the Langevin thermostat individually couples each particle to the heat bath. This produces a very tight coupling, as if the system were immersed in a virtual, viscous medium, but it also suppresses

[2] J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak: Molecular-Dynamics with Coupling to an External Bath. J. Chem. Phys. 81, 3684 (1982).

[3] G. J. Martyna, M. L. Klein, M. Tuckerman: Nosé–Hoover chains: The canonical ensemble via continuous dynamics. J. Chem. Phys. 97, 2635 (1992)

[4] W. F. van Gunsteren, H. J. C. Berendsen: A leap-frog algorithm for stochastic dynamics. Mol. Sim. 1, 173 (1988)

[5] Giovanni Bussi, Davide Donadio, and Michele Parrinello: Canonical sampling through velocity rescaling. J. Chem. Phys. 126(1):014101, (2007).

the natural dynamics in a more pronounced way. For this reason, the Langevin thermostat should primarily be used to generate structures or to sample an ensemble rather than to calculate dynamical properties.

In a Langevin NVT simulation, the degree of coupling can be set by the `Friction` parameter. It is defined in inverse time units, meaning that it works in the opposite way compared to the coupling parameters of the Berendsen or Nose Hoover thermostat. Increasing the friction value therefore causes a tighter coupling to the heat bath and a more severe modification of the dynamics.

NVT Bussi Donadio Parrinello

As we mentioned earlier, **NVT Berendsen** thermostat doesnt sample the correct canonical ensemble. However, a stochastic variant of **NVT Berendsen** is the **NVT Bussi Donadio Parrinello** thermostat (cf. *NVTBussiDonadioParrinello* in the reference manual), which correctly samples the canonical ensemble along with the other advantages of **NVT Berendsen** thermostat.

NPT Simulations

There are three possibilities to run isothermic-isobaric (*NPT*) simulations: The **NPT Berendsen**[2] (cf. *NPTBerendsen* in the reference manual), the **NPT Martyna Tobias Klein**[6] (cf. *NPTMartynaTobiasKlein* in the reference manual) and the **NPT Bernetti Bussi**[7] algorithm (cf. *NPTBernettiBussi* in the reference manual). The **NPT Berendsen** method works via the same principle as the corresponding *NVT* integrator and therefore does not exactly reproduce the correct physical ensemble, although, again, the deviations are typically small for large unit cells.

For production simulations, it is recommended not to use the **NPT Berendsen** barostat. The **NPT Bernetti Bussi** method is usually a better choice. It is a stochastic variant of the Berendsen barostat which rescales the unit cell stochasticaly so as to sample the *NPT* ensemble properly even for very small unit cells.

Similar to the thermostat time scale, the `Barostat time scale` parameter can be used to set how quickly the system pressure approaches and oscillates around the target pressure.

In the workflow block in NanoLab, the `Reservoir Pressure` group box allows you to set the pressure and choose whether you want to use isotropic or anisotropic pressure coupling. Isotropic pressure is a suitable choice if isotropic systems, such as liquids or crystals with cubic symmetry, are simulated. Here, the pressure is calculated as $P = (P_{xx} + P_{yy} + P_{zz})/3$, with $P_{ij} = -\sigma_{ij}$ being the components of the pressure tensor, obtained from the stress tensor σ. All cell vectors will be changed by the same factor, which preserves the shape of the cell. For anisotropic systems, which may have different moduli in different directions, you should uncheck the `Isotropic Pressure` box. This means that all active components P_{ij} of the pressure tensor pressure are coupled independently, and the cell vectors can respond separately to the external pressure. Individual components can be switched on and off via the `Pressure Coupling` check boxes.

Moreover, instead of a single external pressure value, you can also specify independent reference stress values for each component to simulate for instance mechanical shear or creep experiments. The *NPTMartynaTobiasKlein* entry in the **QuantumATK** Reference Manual and the tutorial *Simulating a creep experiment of polycrystalline copper* provide a more detailed explanation of this functionality.

A typical application of NPT simulations is to simulate phase transitions, e.g. from α- to β-quartz, by performing a series of simulations at different temperatures and measuring the average cell volume as a function of the temperature, as reported in Ref.[8].

[6] G. J. Martyna, D. J. Tobias, and M. L. Klein: Constant pressure molecular dynamics algorithms. J. Chem. Phys. 101, 4177 (1994)

[7] M. Bernetti, and G. Bussi: Pressure control using stochastic cell rescaling. J. Chem. Phys. 153, 114107 (2020)

[8] M. H. Müser, and K. Binder: Molecular dynamics study of the - transition in quartz: elastic properties, finite size effects, and hysteresis in the local structure. Phys. Chem. Minerals 28, 746 (2001)

Non-Equilibrium Simulations

In non-equilibrium simulations, a temperature gradient is enforced or strain is applied to study a materials behaviour out of equilibrium.

The **NonEquilibriumMomentumExchange** algorithm simulates a heat flow from a hot region to a cold region and can be used to calculate the thermal conductivity (cf. *NonEquilibriumMomentumExchange* in the reference manual for a usage example).

In a stress-strain calculation the cell is extended or strained in one direction and the resulting stress in that direction is then recorded(cf. *StrainConfigurationHook* in the reference manual for a usage example).

MD Simulations with Constraints

It is possible to fix the positions of selected atoms while running MD simulations. The easiest way to set such constraints is to add a tag to the selected atoms to be constrained in the configuration (using the Builder tool or the Configuration block in the Workflow Builder).

Then, in the **Molecular Dynamics** widget, click on `Add Constraints` to select the tag and apply the needed constraints. You can select between `Fixed` atoms and `Rigid` body constraints. The former type fixes all atoms in the respective group to their initial positions or constrains their movement in the *X*, *Y*, or *Z* direction by selecting *Fixed X*, *Fixed Y*, or *Fixed Z*, respectively. The *Rigid* constraint type treats all atoms within this group as a rigid body. This means that these atoms move collectively according to the force on their center of mass. In the current implementation only translational motions are possible, whereas rigid body rotations are deactivated.

When using constraints, at least one atom must remain unconstrained. The thermostat will automatically adapt to the reduced number of degrees of freedom due to the constraints.

Note, however, that some features in the **MD Analyzer** are not designed to work with constraints.

Note: It is generally **not** recommended to perform constant pressure/stress (i.e. *NPT*) simulations in the presence of constrained atoms, as the stress contribution arising within the constrained parts of the system is not removed from the overall stress tensor.

Device Configurations

Performing an *NVE/NVT* simulation of a **DeviceConfiguration** essentially works in the same way as for a **BulkConfiguration**. The atoms inside the electrode extensions in the central region are automatically fixed by constraints to preserve compatibility with the electrodes. Although it is technically also possible to perform constant pressure/stress (i.e. NPT) simulations of **DeviceConfigurations**, such calculations are not recommended, due to the presence of constraints.

Molecules

Isolated molecules in gas phase can be simulated under *NVE/NVT* ensemble. Molecule simulations can be setup within **MoleculeConfiguration** or **BulkConfiguration**. When using **BulkConfiguration**, care must be taken to specify a box size large enough to account for the finite size effects when periodic boundary condition is applied.

References

4.5.16 Charged Point Defects

Table of Contents

•*Charged Point Defects*

–*Defining defects*

∗*Introduction*

∗*Defect generator types*

∗*Defect generator usage*

–*Pristine Reference Configuration*

∗*Charge Correction Schemes*

–*Chemical Potentials in Compound Materials*

–*Charged Point Defect Calculation*

∗*Defining defects*

∗*Defect Relaxation*

∗*Formation energy*

∗*Defect Concentration*

∗*Symmetry of the Defect*

∗*Ghost atoms*

∗*Interfaces and amorphous materials*

–*Analyzing charged point defect calculations*

∗*Opening defects in the analyzer*

∗*Comparing defect configurations*

∗*Analyzing trap levels*

∗*Analyzing defect concentrations*

∗*Checking the validity of charge corrections*

∗*Viewing the defect report*

–*Defect migration*

–*Diffusivity of defects*

–*Analyzing Diffusion Results*

Defining defects

Introduction

The entry point to calculating charged point defect properties with QuantumATK is a set of objects called defect *Generators*. They generate a dictionary of abstract defect definitions (*NamedPointDefect*) based on the symmetry of the supplied reference or *host* *BulkConfiguration*.

Defect generator types

The defect generator types are:

- *VacancyGenerator* for vacancy defects that remove an atom from the host lattice.
- *SubstitutionalGenerator* for substitutional defects that replace an element in a lattice site.
- *InterstitialGenerator* for interstitial defects occupying empty spaces between lattice sites. The positions that atoms can occupy are calculated using Voronoi geometry analysis. Atoms can be placed either on Voronoi vertices, faces or ridges.
- *SplitInterstitialGenerator* for split interstitial defects. These are interstitial defects that make extra room by displacing a lattice atom out their site.
- *DefectPairGenerator* for site defects paired with another point defect. This generator is constructed using the generators for the two point defects in the pair.

Defect generator usage

The defect generators always generate all possible defects of the selected type. Symmetrically equivalent defects are then sorted into groups that are assigned unique symmetry indices. Note that within each group the defects are sorted with respect to their distance to the unit cell center, the closest one being first.

After the initial generation, the defects can be filtered based on various selection criteria:

- `filterBySymmetryIndex` is intended to pick particular defects by symmetry.
- `filterByLatticeSpecies` is intended to select defects by the original element at the defect site.
- `filterByPointDefect` is intended to explicitly include or exclude defects in a provided list of defects.
- `filterByDistinctConfigurations` uses a structural descriptor to select a number of the most distinct defect configurations.
- `filterByZPositionInterval` picks only those defects in the given z coordinate interval.

Pristine Reference Configuration

The *PristineConfiguration* is used to calculate all the reference data of the original, defect-free configuration. This is used in the point defect calculation to provide the reference material and its relevant properties, such as total energy and band gap. Parameters that are common in the calculation of many defects are also stored in the *PristineConfiguration* class. These include parameters such as the supercell size of the defect, the reference calculator and the type of charge correction to apply to the defect. In this way consistency can be assured when calculating a series of defects.

When calculating the band gap, the *PristineConfiguration* has the option to use a different calculator specifically for the band gap. This can be useful where an appropriate calculator for the band gap, such as the *LCAOCalculator* using `HSE06`, may be too computationally expensive for calculating all of the defect properties for the series of defects. While a supercell is often used for the defect calculation, the band gap is calculated using the unit cell of the pristine material.

This means the band gap calculator can be significantly more computationally demanding than the reference energy calculator, as it is applied to a smaller configuration.

Vibrational modes can also be optionally calculated for the pristine material. If calculated, the vibrational modes can be used to estimate the vibrational enthalpy and entropy of the material, which contributes to the formation free energy of the defect. As with the band gap a specific phonon calculator can be given as input. Typically this is used where another less computationally demanding calculator is known to give good estimates of the vibrational modes. This may be a Moment Tensor Potential (MTP) based *TremoloXCalculator* that was trained on the types of defects being studied. Calculating the vibrational modes in the pristine material will also enable calculating the same vibrational modes in the defect.

Charge Correction Schemes

In density functional theory (DFT) calculations the structure in the *BulkConfiguration* is considered to be the fundamental repeat unit of the extended material. Placing a defect in this structure implies that the defect is similarly repeated throughout the material, giving an artificially high concentration of defects. The problem is compounded when adding charges to the defect, as these charges are also repeated throughout the material. In DFT calculations this non-neutral charge is handled by adding a compensating background charge to the material. This conceptual difference between an isolated defect and a repeated periodic defect gives rise to some artifacts in the calculation that need to be corrected.

When performing accurate defect calculations it is first important to make sure that the defect configuration is large enough so that the atomic structure around the defect is not effected by its repeated copies. This applies to both charged and neutral defects. This means that a supercell of the original pristine material is likely needed to remove structural interactions between defects.

In the case of charged defects the added background charge and periodic defect charges also need to be corrected. One method to do this is the scheme of Freysoldt, Neugebauer and Van de Walle (FNV) [FNVanDWalle09]. This correction is specified in the *ChargedPointDefectConfiguration* class by passing as input the *IsotropicFiniteSizeCorrectionParameters* class. In this method the corrections are separated into long-range and short range terms. The long range interactions between defects are modeled assuming a Gaussian model charge at the defect. Interactions between defects are then calculated via Ewald summation. A short-range correction is also made by comparing the model charge to the difference between the defect and pristine potentials. At distances far from the defect this should be constant. The average value of the potential difference then gives an additional alignment correction. Both of these corrections take into account the dielectric constant of the pristine material in determining the electrostatic energies.

Using this correction scheme the formation energy of the defect at a given charge state $E_f(X^q)$ can be given as:

$$E_f(X^q) = E_{tot}(X^q) - E_{bulk} - \sum_i n_i\mu_i + q(E_F + \mu_e + \delta_{q/b}) - E^q_{lat} \tag{4.9}$$

Here E_{tot} gives the DFT energy of the defect and pristine bulk materials respectively, μ_i is the chemical potential of each element, n_i is the change in the number of each element, E_F is the Fermi energy, μ_e is the electronic chemical potential $\delta_{q/b}$ is the short range potential correction term and E^q_{lat} is the long range correction term. The Fermi energy is set to the reference for the chemical potential, generally the valence band maximum. The addition of the electronic chemical potential allows for modeling changes in the Fermi level due to doping or other causes.

In addition to electronic corrections an additional finite size correction can be added to correct for a change in stress [FGH+14]. Adding a defect to the periodic unit cell adds additional stress to the structure. In the isolated defect this stress is normally dissipated in the surrounding pristine material. Given the bulk modulus of the pristine material B, the additional correction due to elastic interactions can be estimated as:

$$E = \frac{V}{2B}(P - P_0)^2$$

Here V is the volume of the defect unit cell and $P - P_0$ is the difference in pressure calculated in the defect and pristine materials.

Chemical Potentials in Compound Materials

Defects in materials often exist in equilibrium with other materials. These materials can be the reservoir for the defect elements. For the formation energy to correctly reflect the equilibrium of elements in defects and in their respective reservoir, the energy of the atoms in this reservoir needs to be included. This is done by calculating the chemical potential of each element in the defect. Using this, the change in free energy between the reference state and the defect can be calculated. For compound materials, the total energy of a compound material is always smaller than the sum of the chemical potentials of the individual elements.

In the defect framework there are two ways to specify a chemical potential for an element. The first way is to calculate the atomic energy based on calculations of a suitable elemental reference material. This is done through the *CalculatedChemicalPotential* class. The choice of this material may be influenced by the kind of element reservoir that may be present in the device. In these calculations the atomic energy is simply defined as the average energy of an atom. This energy includes both the energy required to form a single atom, as well as the formation energy required to form the reference material. Optionally vibrational energies can also be included, so that an entropy per atom can also be calculated. Potentials defined in this way can be used to model element rich environments for defects. For instance in silicon carbide, if the chemical potential for silicon is defined in this way it models a silicon rich environment, where bulk silicon acts as the reservoir for silicon. If all elements are defined in this way it gives elemental reference conditions. Here the total energy of the material is given as the sum of the atomic chemical potentials and the heat of formation. In silicon carbide this leads to the expression:

$$E_{SiC} = \mu^e_{Si} + \mu^e_C + dH_{SiC}$$

Here dH is the heat of formation and μ^e is the elemental chemical potential. In this reference state it is assumed that each element can be exchanged with a corresponding elemental reservoir.

In addition to calculating the chemical potential, the chemical potential can also be directly specified using the *ReferenceChemicalPotential* class. Here the entropy and enthalpy of the element can be specified for a specific temperature. Note that the enthalpy here must include the energy of forming an isolated atom, as well the the reservoir formation energy. Using this definition of the chemical potential it is possible to provide other reference environments. One possible reference environment is the heat of formation reference conditions. Here the heat of formation of the material is evenly distributed among the available atoms. This assumes that each element is exchanged with a reservoir of the same materials as the defect. In silicon carbide the silicon heat of formation chemical potential μ^f would then be:

$$\mu^f_{Si} = \mu^e_{Si} + \frac{1}{2} dH_{SiC}$$

A third possible reference for specific element rich environments would be to assume the potential for the rich element is the elemental potential. The heat of formation for the material is then distributed to the remaining elements. This assumes that one element is exchanged with an element specific reservoir, whereas the other elements are exchanged with the host material. This gives a poor environment for the other elements that do not have their own reservoir. In silicon carbide the carbon poor chemical potential μ^p_C in a silicon rich environment can be given as:

$$\mu^p_C = E_{SiC} - \mu^e_{Si} = \mu^e_C + dH_{SiC}$$

Charged Point Defect Calculation

The charged point defect structure and its properties is calculated with the *ChargedPointDefectConfiguration* object. This object requires a pristine configuration, a defect definition and a sequence of atomic chemical potentials for the elements in the defect. The calculation of the defect is started by calling the `update` method on the object. Once the defect calculation is complete a number of methods on the object are available for querying the results of the calculation. These include the optimized defect configuration, symmetrically equivalent representations of the defect, the defect formation energy and concentration and the electron trap levels of the defect. Alternatively the defect calculation can be analyzed in the *Charged Point Defect Analyzer*.

Defining defects

The definition of a defect required for the *ChargedPointDefectConfiguration* object is a *NamedPointDefect*. These can be generated for a specific host configuration using the defect generators.

Defect Relaxation

In the *ChargedPointDefectConfiguration* object there are two independent relaxation cycles that can be enabled or disabled. This allows doing relaxation away from the minimum with a computationally cheaper method, such as an MTP, and then fully relaxing the structure with the appropriate reference calculator. The first is a pre-relaxation step. This takes a separate pre-relaxation calculator and optionally a *OptimizeGeometryParameters*. The second step is a relaxation step, which is performed with the reference energy calculator. A separate *OptimizeGeometryParameters* can also be given for this stage. Note that during the relaxation it is not possible to relax the cell, as the cell size is set by the pristine material.

During optimization the charge state closest to zero is performed first. This step always performs the requested pre-relaxation and relaxation steps. The subsequent charge state relaxations are then started from the nearest relaxed charge state. If a *TremoloXCalculator* is given as the pre-relaxation calculator this step is ignored, as these calculators cannot model different charge states. For other calculators the pre-relaxation step, if enabled, is performed before the relaxation step.

To stop the defect structure from possibly drifting during the relaxation, it can be beneficial to fix an atom during optimization. The atom selected to be fixed is one of the atoms furthest from the defect, so that the defect itself is able to fully relax. Adding small random displacements to the initial configuration can also aid optimization in breaking symmetry and moving it off potential saddle points.

Formation energy

The formation energy is a measure of the energy required to create the defect in the given host material using the defined atomic reservoirs. The formation energy also depends on the charge state, the temperature and the electronic chemical potential. The general formula for the defect formation internal electronic energy is given in equation (4.9). Adding vibrational modes to the calculation also adds vibrational internal energy and entropy contributions. In this case the total formation free energy δH_f is given as:

$$\delta H_f = \delta U_{elec} + \delta U_{vib} - T\delta S_{vib}$$

Without vibrational corrections the formation free energy is calculated as the change in the electronic formation internal energy.

The formation free energy is calculated using the `formationFreeEnergy` method on the *ChargedPointDefectConfiguration* object. This method has a number of options depending on the specifics of the defect. The electronic chemical potential can be specified with the argument `electronic_chemical_potential`. The reference for the electronic chemical potential can also be given with the `electronic_chemical_potential_reference`. Here the reference for the electronic chemical potential can be either the valence band edge or the conduction band edge. In cases where a different band gap calculator was used, the energy of charged defects needs to be corrected for the different band gap. The alignment is specified the the `defect_type` argument. Here the defect can be specified as a deep level defect, a shallow acceptor or a shallow donor. This changes whether the defect energy is aligned with the Fermi level, valence band maximum or conduction band minimum respectively. Finally the vibrational corrections, where available, can be included or excluded with the argument `include_vibrations`.

Defect Concentration

Where defects are in equilibrium with the surrounding atomic reservoirs, the concentration of the defect is directly related to the formation energy. The concentration of a defect X in charge state q can be calculated with the `concentration` method using the following equation (see also [Meh07b]):

$$C_{X^q} = \left(\frac{n}{V}\right) \theta \exp\left(\frac{S^f_{X^q}}{k_B}\right) \exp\left(-\frac{E^f_{X^q}}{k_B T}\right)$$

where

- n is the number of defect sites in the supercell.
- V is the volume of the supercell.
- θ is the number of internal degree of freedom, for example, the number of possible spin states.
- $S^f_{X^q}$ is the vibrational formation entropy.
- k_B is the Boltzmann constant.
- $E^f_{X^q}$ is the formation internal energy.
- T is the Temperature.

One can also obtain the intrinsic concentration with the `intrinsicConcentration` method which is just the sum of the concentrations of all charge states available on the *ChargedPointDefectConfiguration*.

Symmetry of the Defect

In each point defect calculation the symmetry of the defect is analyzed. Here the symmetry operations of the original pristine material are used to determine the symmetrically equivalent defect structures, as well as the number of symmetry operations that produce each defect configuration. This can be useful for identifying if two seemingly different defects are actually symmetrically equivalent representations of the same defect. The symmetrically equivalent defect configurations can be accessed using the `symmetricDefect` method on the *ChargedPointDefectConfiguration* object. For each defect configuration the center of the defect is also calculated by calculated the center of the modified atoms in the defect material. The symmetrically equivalent positions of the the defect can also be accessed using the `symmetricDefectPosition` method.

When deciding the supercell repetitions of the pristine material for the defect, it is important to preserve the Bravais lattice symmetry of the pristine material. Breaking the symmetry of the Bravais lattice by providing different numbers of repetitions in symmetrically equivalent directions can produce materials with different symmetry, breaking the validity of the symmetry analysis. Such repetitions can also introduce directional artifacts in the calculation. As an example, bulk silicon has a simple cubic conventional Bravais lattice. Here each direction is symmetrically equivalent. Using 1x2x3 repetitions to create the defect supercell breaks this Bravais lattice type, as now each direction is not equivalent. There is a different distance to the repeated defect in each direction. Using a 2x2x2 defect repetition maintains the Bravais lattice symmetry, as there is still an equivalent distance to the repeated defect in each direction. Breaking the Bravais lattice symmetry can also cause the detection of symmetrically equivalent transition paths to fail.

Ghost atoms

The basis set used in the DFT calculations depends on the atoms included. In the case of vacancy defects, this means that the basis set functions centered on the defect are normally removed when the defect is calculated. This can lead to errors in the energy in cases where the surrounding atoms are stabilized by basis functions belonging to the vacancy site atom. To alleviate this problem a ghost atom can be placed at the defect site during the defect DFT calculations. This maintains the full basis set from the pristine material, allowing a more direct comparison. In geometry optimizations the ghost atom is kept at the vacancy defect center.

Ghost atoms are only supported for vacancy defects. Ghost atoms also cannot be used when calculating transition paths. This is because the ghost atom cannot be included in the transition state. For other defects basis functions are added as well as removed, and so the same basis set cannot be re-created with ghost atoms. In these cases increasing the basis set size can give more accurate defect energies as it reduces the dependence of the energy on specific basis functions.

Interfaces and amorphous materials

Modeling defects at interfaces and in amorphous materials brings additional challenges. These materials often have low or no symmetry, meaning that in each configuration a large number of different defects may be possible. In the defect generators there are two specific methods that are useful for filtering defects in low symmetry configurations. The first method, `filterByZPositionInterval` allows selecting defects in a particular slice along the C direction. The enables selecting defects at a particular depth at the interface. Secondly, the method `filterByDistinctConfigurations` allows selecting for the most diverse set of defects up to a certain number in a set of defects. It does this by assigning similarity scores using atom-based descriptors [SGS+13]. This can reduce the number of possible defects while still spanning the space of possible defects.

Another difficulty with calculating defects at interfaces is with the charge correction. This correction assumes an isotropic material. At interfaces, especially where there is a significant change in the dielectric constants between the two materials, this correction may break down. This is a problem for accurate formation energies of charged point defects. Neutral defects do not require the same energy correction.

Analyzing charged point defect calculations

The results of charged point defect calculations can be analyzed in the *Charged Point Defect Analyzer*.

Opening defects in the analyzer

The analyzer can be opened by double-clicking on a *ChargedPointDefectConfiguration* object in the **Data Tool**. Multiple defects can be opened in the analyzer by selecting the defects and right-clicking to bring up the context menu, and then selecting to open in the analyzer. New defects can be easily added by dragging them onto the Defects collection in the *Defects* tab. Once loaded into the analyzer the defects can be renamed by clicking on the name. Allowing multiple defects to be loaded into the analyzer makes it simple to compare the results of calculations on different defects. Defect calculations that only differ in the supercell size can also be merged into one defect record in the analyzer. This is most useful for comparing the scaling behavior of the formation energy with defect supercell size.

An example results file can be downloaded `here`. This file contains two vacancy defects in SiC calculated with LCAO-PBE. These defects can be opened in the analyzer.

Comparing defect configurations

In the *Compare* tab the configurations of two different defects can be shown and inspected. This allows easy visualization of the two defect structures, so that similarities and differences between different defects can be observed. In the analyzer the symmetry representation and unit cell translation within the supercell can be selected. Using these it is possible to visually see if two defects are symmetrically equivalent. This is most useful in the case of interstitial defects, where defects starting from different positions may relax into common structures.

Analyzing trap levels

Trap levels can be analyzed in both the *Formation Energy* and *Trap Levels* tabs. In the *Formation Energy* tab the change in formation Helmholtz energy with electronic chemical potential is plotted for each charge state. The intersections between these lines show the trap levels in the defect. Optionally the lowest stable charge state (SCS) can also be plotted. Different temperatures can be given at which to plot the formation Helmholtz energy. The finite size corrections can also be added or removed from the plotted formation energy. In the plot the valence and conduction bands are also show with blue shaded areas, showing whether or not the trap levels occur in the band gap.

The following plot shows a typical result for a defect calculation in silicon carbide. Here the -2 to +2 charge states are calculated, giving 4 stable transitions shown by the dashed lines. The intersections showing unstable transitions are also marked with a circle.

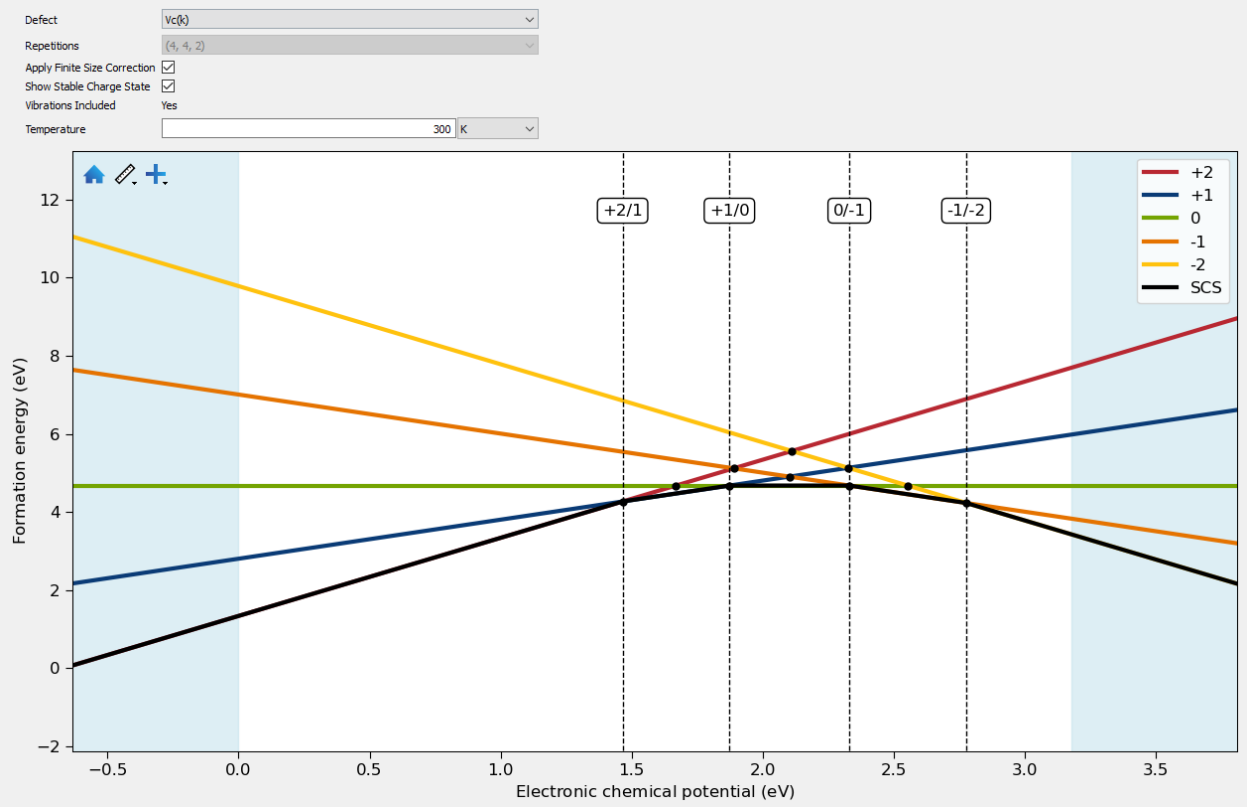


Fig. 4.74: Plot showing the formation energies of a carbon vacancy in silicon carbide.

In the *Trap Levels* tab the trap levels can also be directly plotted for each defect. This allows direct comparison of the trap levels for different defects. Here the relevant supercell size can be selected. Selecting a specific supercell size will show the trap levels of defects with that specific supercell size. Selecting largest will show the trap levels of the largest supercell size for each defect. As with the formation energy, different temperatures can be given and the finite

size effects added or removed. It is also to plot both the stable transitions and unstable integer transitions. Unstable transitions are shown in dashed lines.

The following plot shows a typical result for a defect calculation in silicon carbide. Here the -2 to +2 charge states are calculated for a carbon vacancy at both the h and k sites.

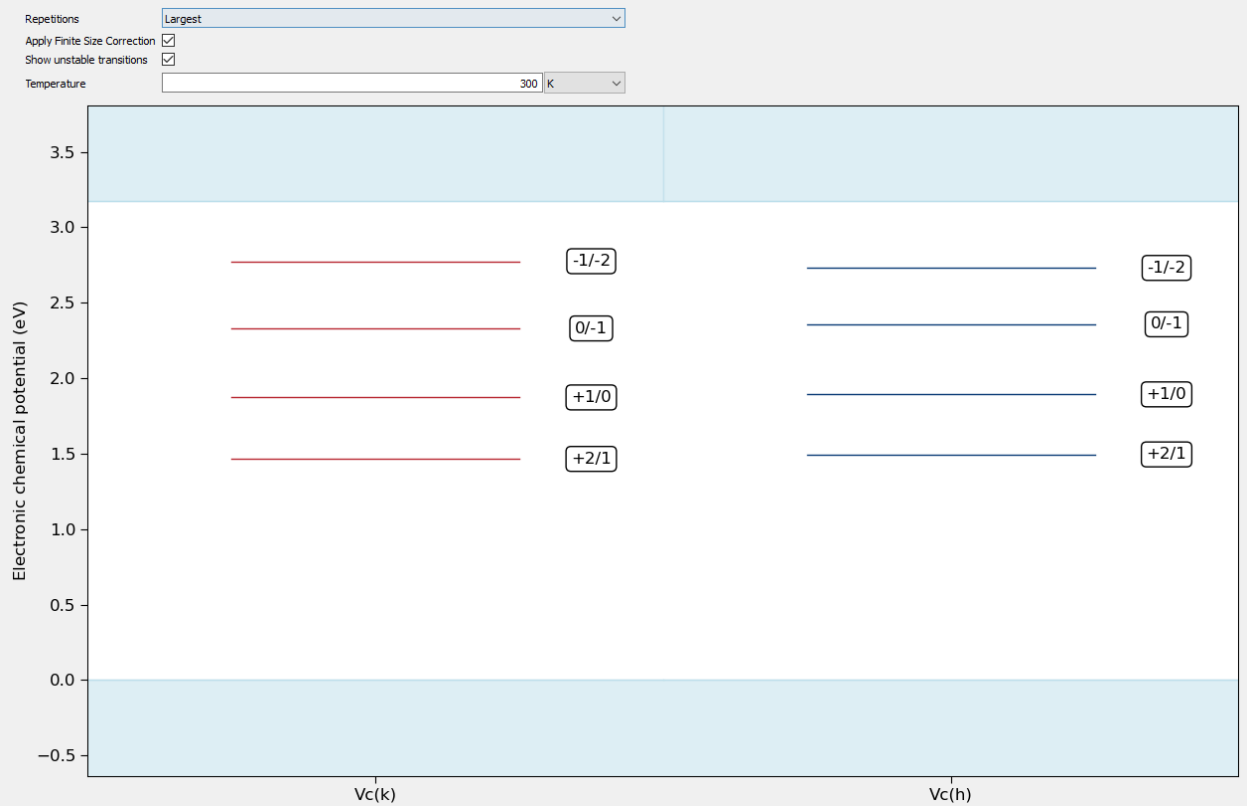


Fig. 4.75: Plot showing the trap levels of carbon vacancies in silicon carbide.

Analyzing defect concentrations

In the *Concentration* tab the concentration of each defect can be plotted with respect to temperature. The intrinsic concentration, which takes into account the concentration of each charge state, is also shown. A temperature range and number of steps can be given to set the temperatures of interest. The relevant electronic chemical potential at which to calculate the concentration can also be given. The concentrations can either be plotted in temperature / concentration or inverse temperature / log concentration units. The latter is often more useful for showing large differences in concentration between defects. In this plot a fit of the data to the Arrhenius equation is also shown.

The following plot shows the calculated concentrations of a carbon vacancy in silicon carbide plotted in inverse temperature / log concentration units. Here the electronic chemical potential set to the mid-gap level.

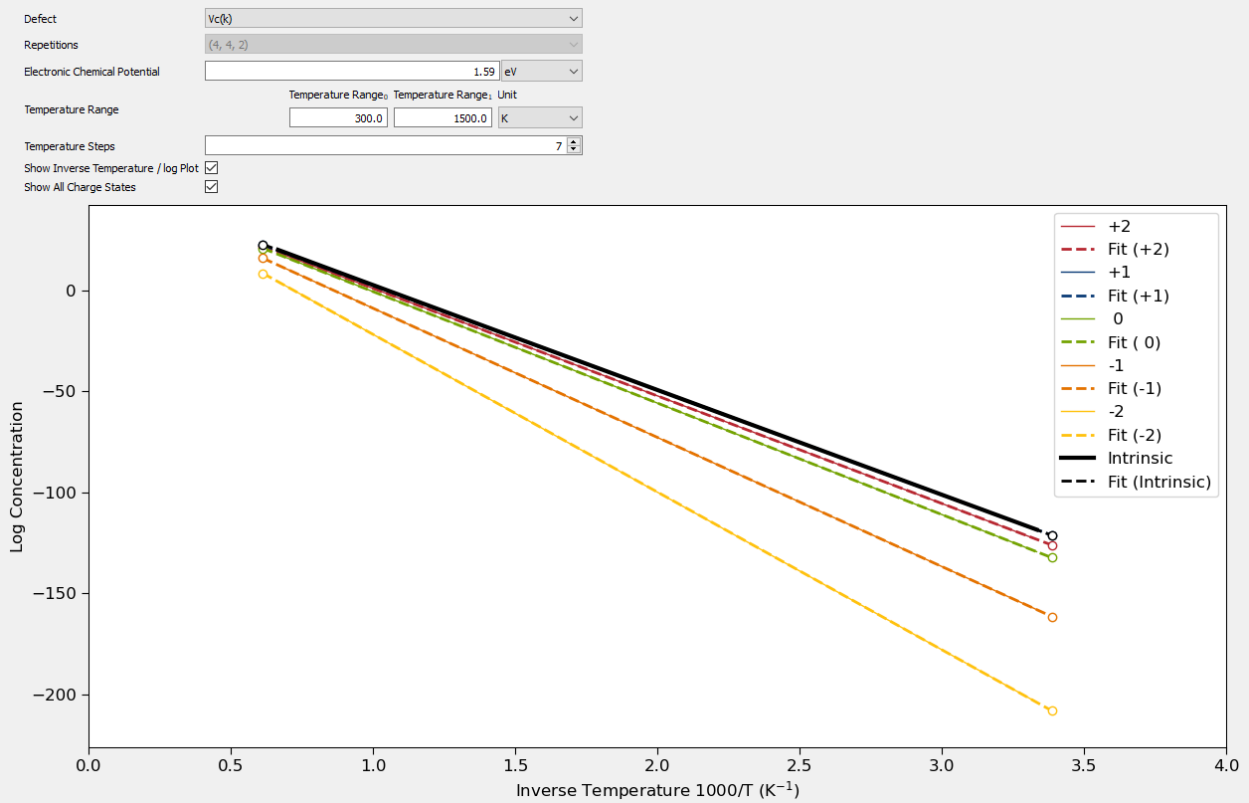


Fig. 4.76: Plot showing the concentrations of each charge state and the intrinsic concentration of a carbon vacancy in silicon carbide.

Checking the validity of charge corrections

When applying a charge correction to the defect formation energy it can be important to check the validity of the approximations in the correction for the individual defect charge states. In the case of the FNV correction it is assumed that the material is isotropic, and that the charge is located primarily at the defect site. The validity of the charge correction can be inspected using the Charged Point Defect Analyzer. By clicking on the *Band Shift Correction* tab you can inspect the value of the short range potential away from the defect. This potential should plateau at the correction value. By calculating the defect at different sizes it is also possible to check the scaling behavior of the defect. The short range potential scales in general as $1/L^3$ while the long range correction scales as $1/L$. If the correction is applied correctly extrapolating the formation energy out to infinite size should give the same formation energy. This can be inspected in the analyzer by selecting the *Finite Size Scaling* tab.

The following plots show some examples of calculations of carbon vacancies in silicon carbide. In the first figure a plot of the defect potential and the Gaussian correction potential is shown. Note that at larger distances from the defect in the middle of the plot, the differences between these two potentials is constant. This indicates that the Gaussian charge distribution is effectively correcting the potential from the defect at long range. The value of the potential here also gives the short range correction for the defect formation energy. In the second plot the scaling of the formation energy with system size is shown. Note that at the extrapolated infinite size, the corrected and uncorrected formation energies agree. This indicates that the corrections are able to effectively calculate the infinite size limit of the defect formation energy.

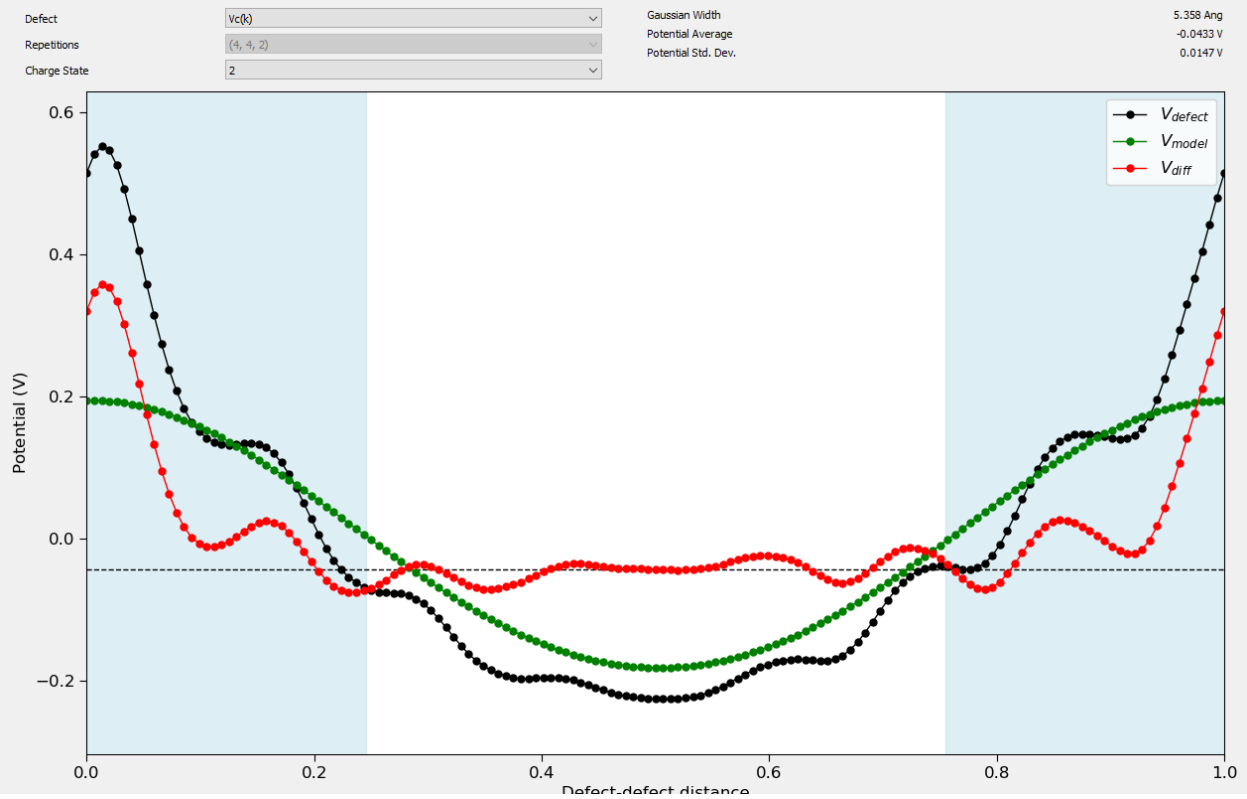


Fig. 4.77: Plot showing the defect potential correction for a +2 carbon vacancy in SiC. The flat potential away from the defect indicates that the short-range potential correction is valid for this defect.

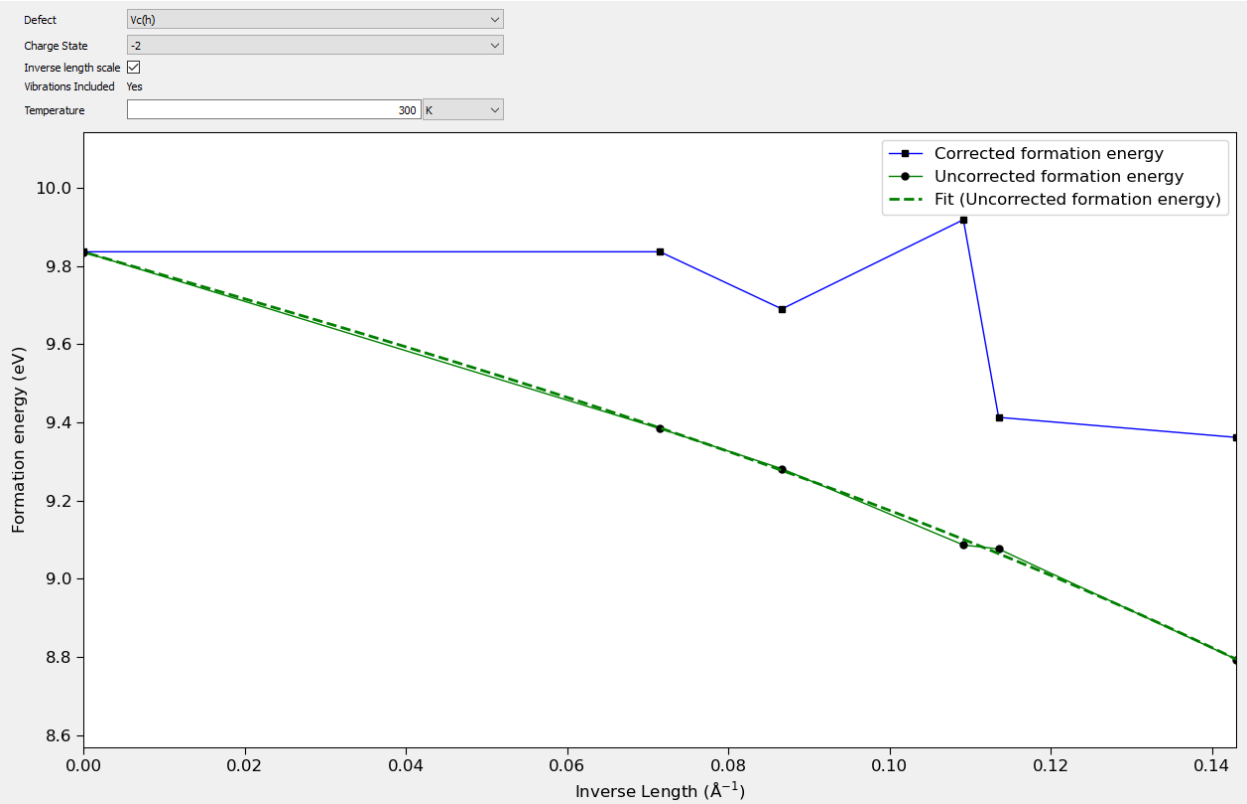


Fig. 4.78: Plot showing the convergence of the formation energy of a -2 carbon vacancy in SiC with defect supercell size. The convergence of the corrected and uncorrected energies at infinite size shows the corrections are effectively correcting the periodic effects in the defect calculation.

Viewing the defect report

In the *Defect Report* tab a textural representation of some information in the defect calculations can be generated. This includes data such as formation energies, concentrations and trap levels. In the analyzer the electronic chemical potential and temperature can be set at which to generate the information. This text output can be used as input for further analyses of the defects.

Defect migration

The migration of a defect within its host material is studied with the *DefectMigrationPaths* class. It takes as input two updated *ChargedPointDefectConfiguration* objects which will serve as the initial and final state of the transition. The object will calculate all possible transitions between the geometry optimized defects using their symmetry information. Optionally, the minimum and maximum travel distance may be specified.

The migration paths are obtained in the form of *NudgedElasticBand* configurations using the `generateNEBs` method. The calculator should be set using the `neb_calculator` argument instead of the `setCalculator` method of the NEB. This will make sure, numerical accuracy parameters stay consistent for DFT calculators. If the default calculator is used, endpoint data (energies, forces, stresses) will be taken from the defects, so they do not get re-calculated as part of the NEB, thus saving time.

The NEBs are then optimized with *OptimizeNudgedElasticBand*. The endpoints need not be optimized if the calculator is the same one used in *ChargedPointDefectConfiguration*, unless the symmetry is broken by the choice of repetitions as this could lead to finite size effects and structural artifacts. If the calculator was changed, e.g. when pre-optimizing with a forcefield calculator, optimizing the endpoints may improve convergence of the *NudgedElasticBand*. Since it is important for the accurate determination of migration barriers to obtain a valid transition state, the `climbing_image` setting should always be set to `True`.

Diffusivity of defects

In order to calculate the diffusivity of a certain defect, the optimized *NudgedElasticBand* and the two *ChargedPointDefectConfiguration* used to construct it need to be passed to the *DefectDiffusivity* class. It will then extract the transition state structure and calculate all data necessary such as the formation energy and vibrational corrections, if requested. These settings are taken from the input *ChargedPointDefectConfiguration*. Note that finite size corrections of the charge are currently not included in the diffusivity, since the position of the charge is ill-defined at the transition state.

To obtain the diffusivity for a defect X in charge state q one calls the `calculateDiffusivity` method which implements the following equation (see also [Meh07a]):

$$D_{X^q} = \frac{a^2}{2d} Z f \nu_0 \exp\left(\frac{S^m_{X^q}}{k_B}\right) \exp\left(-\frac{E^m_{X^q}}{k_B T}\right),$$

where

- a is the migration/hopping distance.
- d is the dimensionality of the diffusion.
- Z is the number of adjacent sites to which the diffusion species can jump.
- f is the correlation factor.
- ν_0 is the attempt rate.
- $S^m_{X^q}$ is the migration entropy.
- $E^m_{X^q}$ is the migration energy.

The attempt rate ν_0 depends on the vibrational spectrum of the initial state $\{\nu_i\}$ and the spectrum at the transition state $\{\nu_i'\}$:

$$\nu_0 = \frac{\prod_i^N \nu_i}{\prod_i^{N-1} \nu_i'}.$$

The migration entropy is calculated as described in *PhononDensityOfStates*.

Analyzing Diffusion Results

The results of a *DefectDiffusionRates* calculation can be shown in the *Defect Diffusion Rates Analyzer*. This can be opened by simply double clicking on the corresponding results object in the **Data Tool**.

An example results file can be downloaded `here`. This file contains the diffusion paths of Si and C vacancies in SiC. This was calculated with a Moment Tensor Potential (MTP) that was trained on defects in SiC. This can be opened in the analyzer.

The *Defect Diffusion Rates Analyzer* shows information regarding the different calculated diffusion paths. The figure below shows the analyzer with data from calculations on silicon and carbon vacancies in silicon carbide.

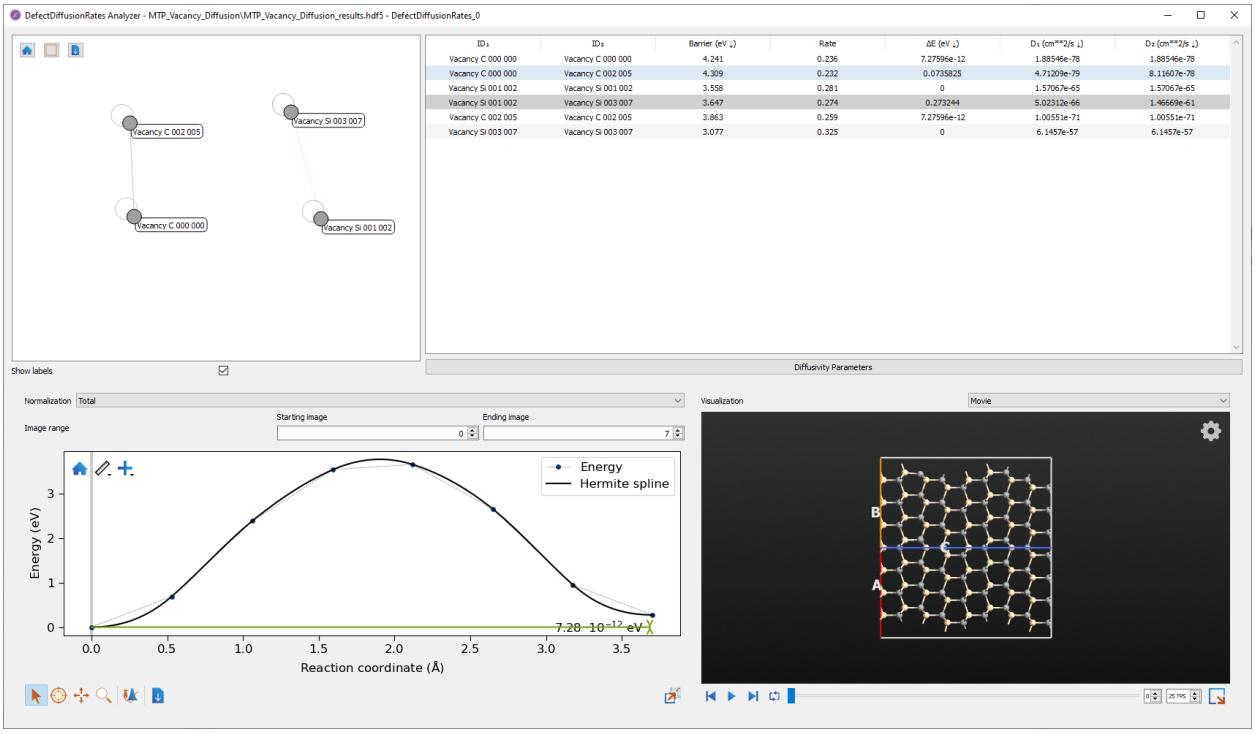


Fig. 4.79: The Defect Diffusion Rates Analyzer with data from calculations on vacancy diffusion in SiC.

In the top left panel is a graph view of the diffusion network. Here, each defect is shown as a filled circle. The lines indicate possible transitions between defects. The open circles represent self diffusion paths, where one defect transitions to the same type of defect at a different site. Depending on the defects selected, there can be one or more isolated graphs showing the connections between defects. In the example, as carbon vacancies cannot transition to silicon vacancies, two isolated graphs are shown.

In the a panel shows information about each defect diffusion path. This shows the forward reaction barrier, the forward transition energy, and the forward and reverse diffusivities for each transition path. These numbers can be used for further calculations on the defect dynamics, such as kinetic Monte Carlo simulations. The parameters used to calculate

the diffusivity can be accessed and modified by pushing the *Diffusivity Parameters* button. This data can also be exported to CSV or XLSX format by right clicking on the panel.

At the bottom of the analyzer is shown the NEB for the selected defect transitions. Transitions can be selected by selecting either one of the transition lines in the transition graph, or selecting the corresponding data row in the data table. Visualizing the NEB allows inspecting the transition path and the likely transition state.

4.5.17 G_0W_0 calculator in QuantumATK

In QuantumATK we implemented a G_0W_0 method, which takes initial Kohn-Sham wave functions and energies, and calculates new quasi-particle energies using many-body perturbation theory[1]. In this document, we will try to explain the particular algorithm that is implemented in QuantumATK and how to use it.

Standard GW implementation work purely in imaginary frequency space, and have a scaling with system size of $O(N^4)$. Recently, an imaginary time to imaginary frequency method has been developed[2], which scales as $O(N^3)$.

We have combined this approach using an LCAO basis with the PARI approximations as described in[3]. Although it formally scales with system size as $O(N^3)$, in practice, for small-to-medium sized systems it scales as $O(N^2)$, since the dominating calculations (polarizability and self-energy) scale as $O(N^2)$.

Imaginary time and frequency grid

First we need to define the imaginary time and frequency grids $\{\tau_i, \omega_k\}$. These are optimized to be a non-homogeneous grid to represent MP2 correlation energy as best as possible given an analytic form of the polarizability in imaginary time and frequency:

$$\eta[\gamma,\omega](x) = \frac{1}{x} - \frac{1}{\pi}\sum_{k=1}^{N_\omega}\gamma_k\left(\frac{2x}{x^2+\omega_k^2}\right)^2$$

$$\eta[\sigma,\tau](x) = \frac{1}{2x} - \sum_{i=1}^{N_\tau}\sigma_k e^{-2x|\tau_i|}$$

The grids are found by minimizing the $L2$ norm of the cost functions over the range $x \in [e_{\text{min}}, e_{\text{max}}]$ where e_{min} is the lowest possible transition and e_{max} is the highest.

Optimization is non-linear, and we use a Levenberg-Marquardt algorithm starting from pretabulated initial guesses to find the best possible grid in the interval. This calculation takes up some time, and this is indicated in the log output when running the calculation.

On the *GWCalculator*, you can use the parameter `tau_precision_parameter` to influence how many frequency and time points will be used. If the parameter, which is a double, is set to 1.0, the optimization will be done starting from the initial value that contains the most points, which will lead to the most accurate result. If 0.0 is chosen for the parameter, the grid with the least number of points will be chosen. For any value in between an intermediate number of points is selected.

[1] Giovanni Onida, Lucia Reining, and Angel Rubio. Electronic excitations: density-functional versus many-body Greens-function approaches. *Rev. Mod. Phys.*, 74:601–659, Jun 2002. URL: https://link.aps.org/doi/10.1103/RevModPhys.74.601, doi:10.1103/RevModPhys.74.601.

[2] Peitao Liu, Merzuk Kaltak, Jiri Klimes, and Georg Kresse. Cubic scaling GW: Towards fast quasiparticle calculations. *Phys. Rev. B*, 94:165109, Oct 2016. URL: https://link.aps.org/doi/10.1103/PhysRevB.94.165109, doi:10.1103/PhysRevB.94.165109.

[3] Xinguo Ren, Florian Merz, Hong Jiang, YiăYao, Markus Rampp, Hermann Lederer, Volker Blum, and Matthias Scheffler. All-electron periodic G0W0 implementation with numerical atomic orbital basis functions: Algorithm and benchmarks. *Phys. Rev. Mater.*, 5:013807, Jan 2021. URL: https://link.aps.org/doi/10.1103/PhysRevMaterials.5.013807, doi:10.1103/PhysRevMaterials.5.013807.

Coulomb integrals and Polarizability using Auxiliary Basis Sets and PARI

The screened ($W_{\alpha\beta;\gamma\delta}$) and unscreened ($V_{\alpha\beta;\gamma\delta}$) Coulomb integrals, and the polarizability ($\Pi_{\alpha\beta;\gamma\delta}$), are 4-index properties. Calculating them directly this way would have unfortunate scaling properties, both for computation time and storage. To avoid this we represent them in auxiliary basis space, in a similar way as in our LCAO-Hybrid code. As auxiliary basis set we use the on-site products of the LCAO basis set, and define the general product of two basis function as:

$$\phi^A_\alpha(\mathbf{r}-\mathbf{R}_A)\phi^B_\alpha(\mathbf{r}-\mathbf{R}_B) = \sum_{M\in[A,B]}\sum_{\mu\in M} C^\mu_{\alpha\beta}P_\mu(\mathbf{r}-\mathbf{R}_M)\ ,$$

where the auxiliary basis functions, $P_\mu(\mathbf{r})$ are centered on either of the sites the LCAO basis functions are centered on. This is called the pair-atom resolution of identity (PARI) approximation[4].

If we use this, the Coulomb integral is reduced from a four-index to a two-index quantity:

$$V_{\alpha\beta;\gamma\delta} \approx C^\mu_{\alpha\beta}V_{\mu\nu}C^\nu_{\gamma\delta}$$

Greens functions in imaginary time

Before starting on the actual GW calculation, we do a normal DFT calculation, using any xc-functional. Afterwards, the Greens functions are constructed from the converged Kohn-Sham orbitals, in imaginary time as:

$$G(i\tau)_{\alpha\beta} = \Theta(\tau)\ G^+(i\tau)_{\alpha\beta} + \Theta(-\tau)\ G^-(i\tau)_{\alpha\beta}\ ,$$

with

$$G^+(i\tau)_{\alpha\beta} = i\sum_{\mathbf{k}n} w_\mathbf{k}\ f(\epsilon^0_{\mathbf{k}n})\ \left[\psi^0_{\mathbf{k}n}\right]_\alpha \left[\psi^0_{\mathbf{k}n}\right]^*_\beta e^{-|\epsilon^0_{\mathbf{k}n}-\epsilon^0_F|\tau},$$
$$G^-(i\tau)_{\alpha\beta} = i\sum_{\mathbf{k}n} w_\mathbf{k}\left[1-f(\epsilon^0_{\mathbf{k}n})\right]\ \left[\psi^0_{\mathbf{k}n}\right]_\alpha \left[\psi^0_{\mathbf{k}n}(\mathbf{r}')\right]^*_\beta e^{-|\epsilon^0_{\mathbf{k}n}-\epsilon^0_F|\tau},$$

where $\{\psi^0_{\mathbf{k}n}, \epsilon^0_{\mathbf{k}n}\}$ are the solutions to the Kohn-Sham problem. This step scales as $O(N^3)$ with system size and linearly, $O(N_k)$, with the number of k-points. Here $\{\alpha,\beta\}$ are indices for the normal LCAO orbitals.

The number of bands that are included in the Greens function calculation is determined by the input parameter `energy_cutoff` on the *GWCalculator*. When the *GWCalculator* is created it checks the eigenvalues of the DFT calculation in the Γ point, and determines the number of bands that are below the energy_cutoff. These bands are then used in the construction of the Greens functions. This is done because our LCAO basis sets are not made to describe highly excited states, and including all of them might give rise to issues.

Polarizability in imaginary time

The polarizability is defined as a four-index object in function of the Greens function in imaginary time:

$$\Pi_{\alpha\beta;\gamma\delta}(i\tau) = iG^+_{\alpha\gamma}(i\tau)G^-_{\beta\delta}(-i\tau)\ .$$

Using the PARI approximation and introducing the auxiliary basis indices as $\{\mu,\nu\}$ this becomes:

$$\Pi_{\mu\nu}(i\tau) = i\sum_{\alpha\beta}\sum_{\gamma\delta} C^\mu_{\alpha\beta}G^+_{\alpha\gamma}(i\tau)G^-_{\beta\delta}(-i\tau)C^\nu_{\gamma\delta}\ .$$

[4] Peize Lin, Xinguo Ren, and Lixin He. Efficient Hybrid Density Functional Calculations for Large Periodic Systems Using Numerical Atomic Orbitals. *Journal of Chemical Theory and Computation*, 17(1):222–239, 2021. PMID: 33307678. URL: https://doi.org/10.1021/acs.jctc.0c00960, arXiv:https://doi.org/10.1021/acs.jctc.0c00960, doi:10.1021/acs.jctc.0c00960.

This is a real space object, which for periodic systems has translations that are not included to keep the notation simple. The whole object is Born-von Karman periodic, so that it scales with the k-point sampling chosen by the user. This means that the calculation scales lineary with the number of k-points. If there is no sparsity, the scaling with system size is $O(N^2)$ because of the PARI approximation, which is much more favorable than the $O(N^4)$ scaling for this step when calculated directly in frequency space.

Transform from imaginary time to imaginary frequency

We now have the polarizability on an imaginary time grid, but we need it in (imaginary) frequency space to calculate the screened Coulomb potential. Both time and frequency grids are non-homogeneous/non-regular and in order to transform, we need custom transforms.

To get the best possible result we optimize a cosine transform that is optimal to transform the analytic form of the polarizability from time to frequency and back[Page 4387, 2], i.e.:

$$\Pi_{\mu\nu}(\omega_k) = \sum_{i=1}^{N_\tau} \gamma_{ki} \cos(\omega_k \tau_i) \Pi_{\mu\nu}(\tau_i) \ .$$

with γ chosen to minimize:

$$\eta(x,\gamma) = \frac{2x}{x^2+\omega_k^2} - \sum_{i=1}^{N_\tau} \gamma_{ki} \cos(\omega_k \tau_i) e^{-x\tau_i} \ .$$

Dielectric tensor and screened Coulomb potential

To calculate the dielectric tensor we transform to frequency space and $\mathbf{k}$ space representation:

$$\epsilon_{\mu\nu}(\mathbf{q}, i\omega) = \delta_{\mu\nu} - V^{\frac{1}{2}}_{\mu\kappa}(\mathbf{q}) \Pi_{\kappa\lambda}(\mathbf{q}, i\omega) V^{\frac{1}{2}}_{\lambda\nu}(\mathbf{q})$$

The inverse of the dielectric tensor is then used compute the screened coulomb potential:

$$W_{\mu\nu}(\mathbf{q}, i\omega) = V^{\frac{1}{2}}_{\mu\kappa}(\mathbf{q}) \epsilon^{-1}_{\kappa\lambda}(\mathbf{q}, i\omega) V^{\frac{1}{2}}_{\lambda\nu}$$

Once this is calculated for all $\mathbf{q}$ points we transform back real space and imaginary time:

$$W_{\mu\nu}(\mathbf{q}, i\omega) \rightarrow W_{\mu\nu}[\mathbf{T}](i\tau)$$

Exchange and correlation Self-energy calculation

Once we have the screened Coulomb potential, we can use it to calculate the self-energies in imaginary time. But we need to split it up in separate exchange and correlation parts:

$$\Sigma^x_{\alpha\beta} = i \sum_{\gamma\delta} \sum_{\mu\nu} C^\mu_{\alpha\gamma} V_{\mu\nu} C^\nu_{\beta\delta} G^+_{\gamma\delta}(\tau = 0) \ ,$$

and

$$\Sigma^c_{\mu\nu}(\tau) = i \sum_{\gamma\delta} \sum_{\mu\nu} C^\mu_{\alpha\gamma} \tilde{W}_{\mu\nu}(i\tau) C^\nu_{\beta\delta} G_{\gamma\delta}(i\tau) \ .$$

where $\tilde{W} = W - V$. This is done because otherwise $W(i\omega)$ would not decay in frequency space, and the cosine transform back to $i\tau$ would be impossible.

This calculation is similar to the one we have in LCAO with Hybrids, the only change is that it needs to be done at all time points. The scaling, for a non-sparse hamiltonian is again $O(N^2)$ with system size, and since we are using Born-von Karman boundary conditions the scaling with the number of k-points is linear.

This is the point where the heavy calculations end, and if you call update() on the calculator, this is where it stops, and the self-energy will be stored in imaginary time and real space for use in post-processing analysis objects.

Solving the quasi-particle equation to get the band energies

To get the correction to the band energies, we first project the self-energies on the Kohn-Sham bands. For this we Fourier transform the self-energy to the desired k-point and calculate:

$$\Sigma^c_{nn}(\mathbf{k}, i\tau) = \sum_{\alpha\beta} \left[\psi^0_{\mathbf{k}n}\right]_\alpha \Sigma^c_{\alpha\beta}(\mathbf{k}, i\tau) \left[\psi^0_{\mathbf{k}n}\right]_\beta \ .$$

We can now symmetrize and anti-symmetrize the band self energies and cosine and sine transform them to imaginary frequency space and add them back together:

$$\begin{aligned}\Sigma^c_{nn}(\mathbf{k}, i\omega) = &\sum_{i=1}^{N_\tau} \gamma_k i \cos(\omega_k \tau_i) \left[\Sigma^c_{nn}(\mathbf{k} i\tau) + \Sigma^c_{nn}(\mathbf{k}, -i\tau)\right] \\ &+ i \sum_{i=1} \lambda_{ki} \sin(\omega_k \tau_i) \left[\Sigma^c_{nn}(\mathbf{k}, i\tau) - \Sigma^c_{nn}(\mathbf{k}, -\mathbf{i}\tau)\right] \ .\end{aligned}$$

To find the correct band-energy for band n we need to solve the following root-finding problem:

$$\omega = \epsilon^0_n - v^{xc}_n + \Sigma^x_{nn} + \mathrm{Re}\left[\Sigma^c_{nn}(\omega)\right] \ ,$$

where ω is on the real axis now. The ω that solves this equation is the G_0W_0 quasi-particle energy. It is important to note that this equation is solved on the real frequency axis. To get the self-energy in real frequency we have to do an analytic continuation, by using the continued fraction representation of the self-energy:

$$\Sigma_{nn}(\omega) \approx \frac{a_1}{1+} \frac{a_2(\omega - \omega_1)}{1+} \cdots \frac{a_{N_\omega}(\omega - \omega_{N\omega-1})}{1 + (\omega - \omega_{N_\omega}) g_{N_\omega+1}(\omega)}$$

This procedure is in the Analysis objects that are support for GW, which are *Bandstructure*, *DensityOfStates*, *ProjectedDensityOfStates*, *FatBandstructure*, *Eigenvalues* and *EffectiveMass*.

Schematic view of the algorithm

Γ point correction for k-point convergence

In GW the problem arises that the Coulomb potential needs to be calculated and screened at the Γ point, but this cannot be done since it diverges as $\frac{1}{q^2}$ for 3D systems and $\frac{1}{q}$ for 2D systems.

The easiest way to avoid this divergence is just not include the Γ point in the k-point sampling. This can be done by setting `include_gamma_point=False` on the *GWCalculator*. The problem is that this will lead to a very slow convergence with k-points for gapped systems, because this assumes that the long range part of the Coulomb potential is perfectly screened by the dielectric tensor, and this will lead to an underestimation of the band gap. To get the correct band gap, one needs to fit a linear curve to the band gap at different k-point sampling, and extrapolate the result to infinite k-point sampling.

We have developed a method to correct for this, both for 3D and 2D systems, so that the contribution at Γ is regularized and the convergence with k-points is much faster, which can be used by setting `include_gamma_point=True`. This is the default setting in QuantumATK.

In the figures below we show the calculation of the band gap for both 3D Si FCC, and 2D MoS_2 with and without the Γ point correction, and show that the extrapolated band gap can be approximated well at small k-point sampling using the Γ point correction.

For metallic systems, the Γ point correction is not important since the long range part of the Coulomb potential is screened by the free electrons. If you are calculating an interface system with both metallic and insulating parts, it is recommended to disable the Γ point correction.

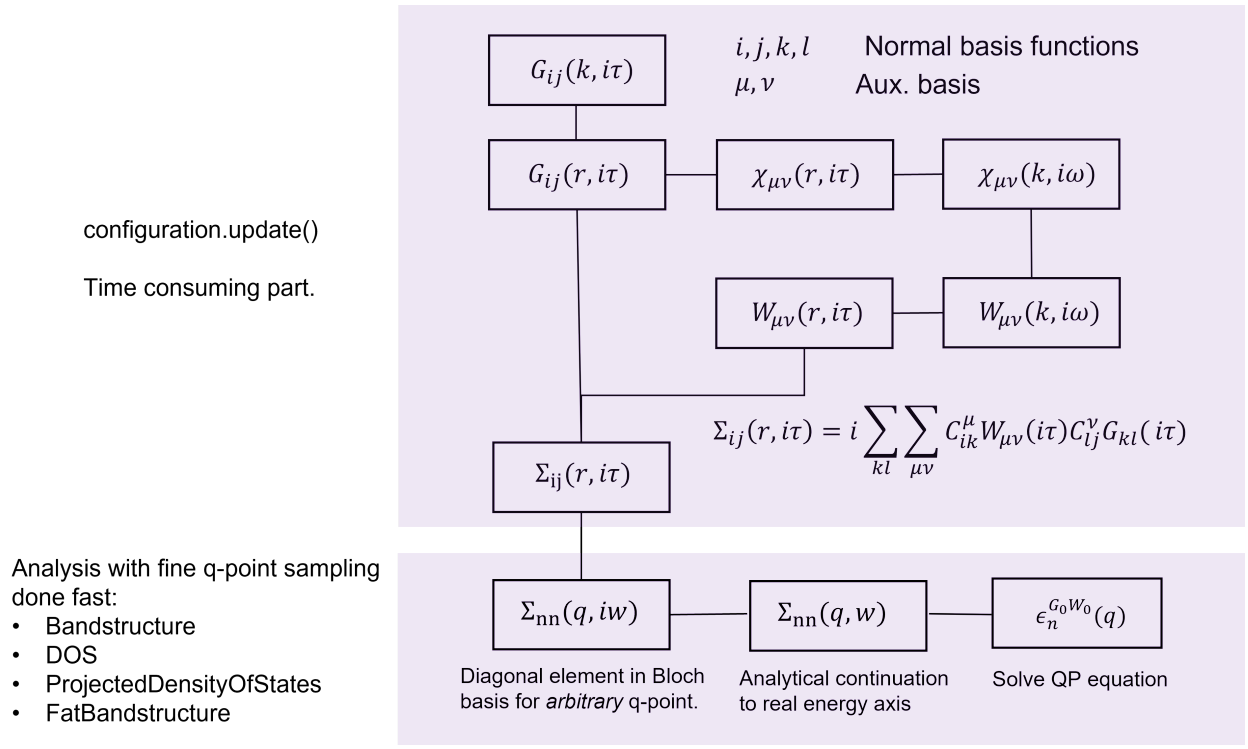


Fig. 4.80: A flowchart diagram showing how the GW calculations is done, and the quasi-particle energies are calculated.

Parallelization of the algorithm

Our implementation of the G_0W_0 method is parallelized using both MPI *and* openMP. We recommend using openMP threads as much as possible, since it also reduced the memory usage of the calculation. The MPI parallelization of the calculation of the polarizability and self-energy is done over the atoms in the system. It is therefore important to never use more MPI processes than there are atoms in the system you are calculating. Doing so will lead to poor performance.

Usage Examples

To run a GW calculation one first has to construct a regular *LCAOCalculator*, which is then set on a *GWCalculator* object:

```
lcao_calculator = LCAOCalculator(
    basis_set=BasisGGAPseudoDojo.High,
    exchange_correlation=HybridGGA.HSE06)

gw_calculator = GWCalculator(
    lcao_calculator,
    energy_cutoff=50*eV,
    polarizability_compression_parameter=1.0e-07,
    w_compression_parameter=1.0e-05,
    include_gamma_point=True)

bulk_configuration.setCalculator(gw_calculator)
bulk_configuration.update()
```

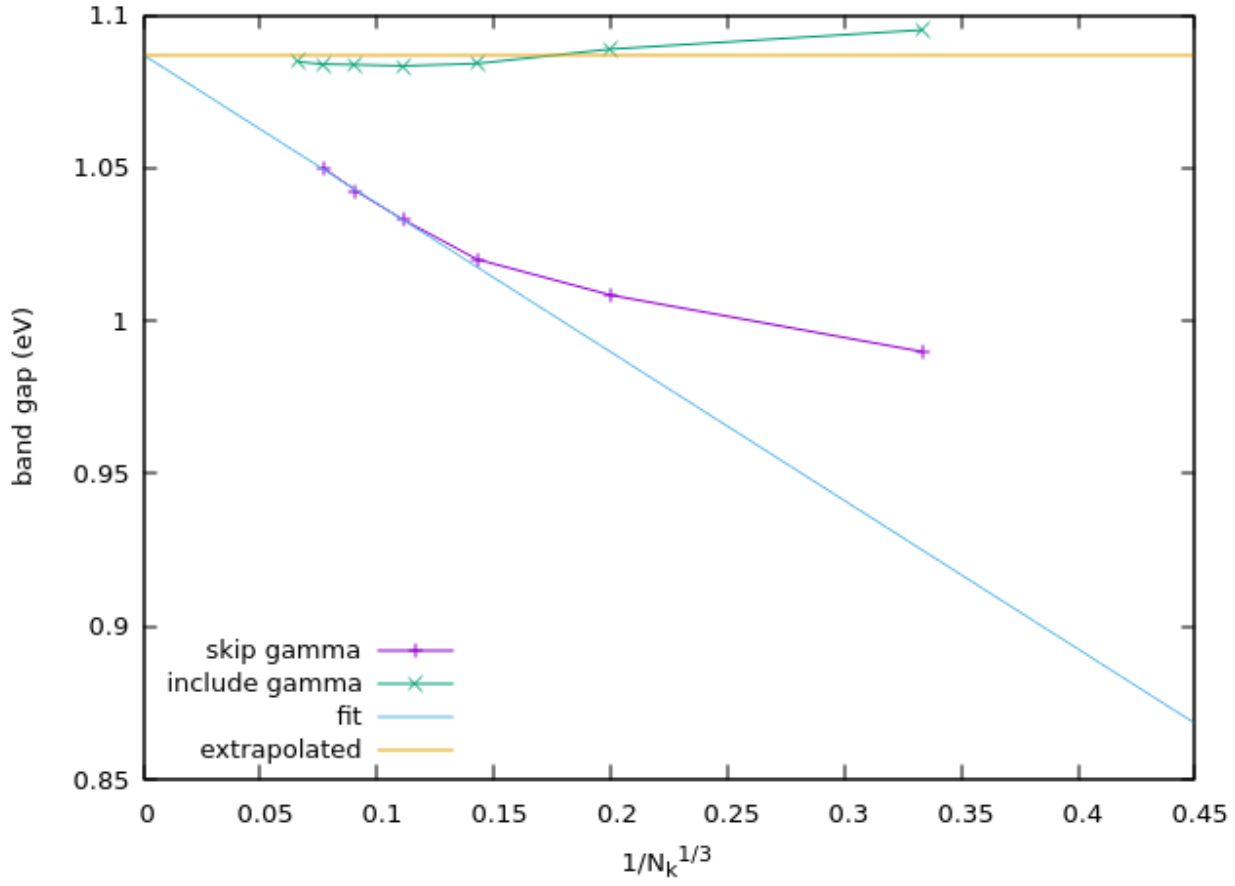


Fig. 4.81: Si-FCC band gap calculated @PBE with different number of k-points, with and without the Γ point correction.

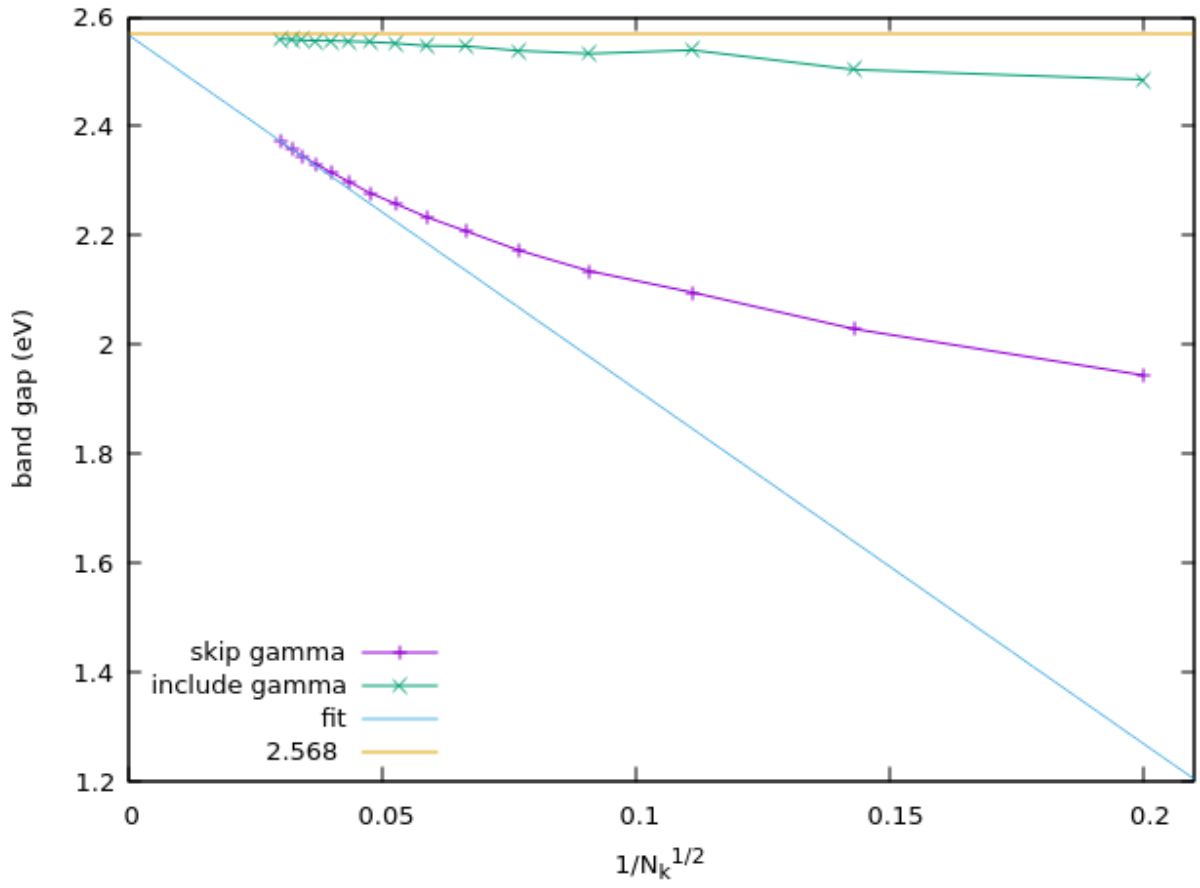


Fig. 4.82: 2D MoS_2 band gap calculated @PBE with different number of k-points, with and without the Γ point correction.

Once the update function is called, the self energies are calculated and stored on the *GWCalculator* object. To get the bandstructure or density of states we just call the standard analysis objects:

```
bandstructure = Bandstructure(bulk_configuration, bands_above_fermi_level=4)
nlsave("gw_bandstructure.hdf5", bandstructure)

pdos= ProjectedDensityOfStates(
    bulk_configuration,
    MonkhorstPackGrid(21,21,21),
    projections=ProjectOnShellsBySite,
    bands_above_fermi_level=4)
nlsave("gw_pdos.hdf5", pdos)
```

For GW calculations it is important to have a good basis set. To get a basis set that matches plane-wave accuracy we have developed the *BasisSetOptimizer* class. For a reference system a plane wave calculation is performed and the LCAO basis set is optimized to match the plane wave calculation. The basis set is then stored in a file:

```
basis_set_optimizer = BasisSetOptimizer(
    bulk_configuration,
    orbitals_per_shell_map={Silicon: [6,6,5,4]},
    energy_cutoff=10*eV,
    tolerance=1.0e-5)

opt_basis_set = basis_set_optimizer.optimize()
nlsave("opt_basis_set.hdf5", opt_basis_set)
```

This can be used in the GW calculation by setting it on the *LCAOCalculator* object:

```
opt_basis_set = nlread("opt_basis_set.hdf5")[0]
lcao_calculator = LCAOCalculator(
    basis_set=opt_basis_set,
    exchange_correlation=HybridGGA.HSE06)

gw_calculator = GWCalculator(
    lcao_calculator,
    energy_cutoff=50*eV,
    polarizability_compression_parameter=1.0e-07,
    w_compression_parameter=1.0e-05,
    include_gamma_point=True)

bulk_configuration.setCalculator(gw_calculator)
bulk_configuration.update()
```

References

4.5.18 External machine-learned force fields in QuantumATK

The field of universal machine-learned force fields (MLFFs) is advancing rapidly, with new models continuously being developed and published. While QuantumATK offers native support for several of these MLFFs (see the list *here*), it is not feasible to include every available model within the package due to the fast pace of innovation and the growing number of MLFFs.

To accommodate this, QuantumATK provides a flexible framework for integrating external MLFFs through the *ExternalMachineLearnedFFCalculator*. This functionality enables the use of any MLFF that implements an

ASE-compatible calculator, including models not yet supported natively in QuantumATK. It also allows users to incorporate their own open-source trained, or fine-tuned MLFFs into their QuantumATK simulation workflows.

Note: Synopsys QuantumATK is not affiliated with, nor does it endorse or promote, any of the external packages mentioned in this guide.

Synopsys QuantumATK assumes no responsibility for the accuracy, safety, or reliability of any third-party packages or files, nor for any issues that may arise from their use. Users are strongly advised to install and use only packages and files obtained from trusted sources.

While the packages listed below have been tested, support is not guaranteed for all versions. Other versions may function correctly, but compatibility and performance cannot be assured.

Please ensure that you comply with the license terms of any external packages you choose to use.

Note: Some MLFFs may have dependencies that are incompatible with the Python environment provided by QuantumATK. This can occur, for example, when a MLFF requires a specific version of a library that differs from the one bundled with QuantumATK. Additionally, the set of supported external MLFFs may change over time as both QuantumATK and the MLFFs themselves are updated.

Note: All arguments passed to the underlying ASE calculator class should be basic python types, such as str, int, float, etc. This will ensure that the *ExternalMachineLearnedFFCalculator* can properly restore the calculator when reading it from disc and in complex workflows, such as *SurfaceProcessSimulation* If any arguments types are not not known to QuantumATK, a placeholder calculator is saved instead, which provides basic functionality, e.g. in the GUI, but cannot be used for further calculations. If needed, this can also be achieved by implementing a custom wrapper function or class that sets up the calculator using only basic types as input arguments arguments, as shown with the Orb example below.

Note: It is recommended to use Linux when running QuantumATK with external MLFFs, as support for Windows is limited for many of these models. Consequently, this guide focuses on installation and usage instructions for Linux environments.

Setting up the virtual environment

Although external MLFF modules can be installed directly into QuantumATK, is always recommended that external MLFF modules are installed in a separate Python virtual environment to be able to completely remove possible conflicting installations without having to re-install QuantumATK. For a detailed guide on how to set up a virtual environment, please refer to the *virtual environment* section of the QuantumATK manual.

The following is a brief example of how to set up a new virtual environment suitable for installing an external MLFF:

```
sh$ QATK_DIR="/path/to/your/QuantumATK/installation"
sh$ VENV_DIR="/path/to/your/custom/venv"
sh$ source "${QATK_DIR}/atkpython/bin/activate_env"
(qatk) sh$ python -m venv --system-site-packages --upgrade-deps "$VENV_DIR"
(qatk) sh$ source "${VENV_DIR}/bin/activate"
```

After running these commands, your command prompt should display the prefix `(venv) (qatk)`, indicating that both environments are active. You are now ready to install the required packages for your external MLFF.

SevenNet

Note: The following guide applies to SevenNet version 0.11.1.

To install the required packages for using SevenNet models, run:

```
(venv) (qatk) sh$ pip install sevenn==0.11.1
```

This command may raise an error related to the version of the `e3nn` library. SevenNet requires `e3nn>=0.5.0`, whereas QuantumATK includes `e3nn==0.4.4`. As a result, you will have `e3nn==0.4.4` in the QuantumATK shell environment and `e3nn>=0.5.0` in the virtual environment.

This version mismatch is generally not an issue, as Python prioritizes packages installed in the active virtual environment. Therefore, the correct version of `e3nn` will be used when running SevenNet. However, this setup does mean that MACE training functionality will not work within the QuantumATK shell environment while the virtual environment is active.

Below is a simple example demonstrating how to use SevenNet. This snippet sets up a calculator using the `7net-mf-ompa model` and runs it on a GPU:

```
from sevenn.calculator import SevenNetCalculator
calculator = ExternalMachineLearnedFFCalculator(
    SevenNetCalculator,
    model='7net-mf-ompa',
    modal='mpa',
    device='cuda'
)
```

Orb models

Note: The following guide applies to Orb models version 0.4.2.

To install the required packages for using Orb models, run the following commands:

```
(venv) (qatk) sh$ pip install orb-models==0.4.2
(venv) (qatk) sh$ pip install "pynanoflann@git+https://github.com/dwastberg/pynanoflann
↪#egg=af434039ae14bedcbb838a7808924d6689274168"
```

Orb models require `scipy>=1.13.1`, while QuantumATK includes `scipy==1.10.1`. Despite this version mismatch, Orb models appear to function correctly with the older version. To avoid conflicts, you can simply uninstall the newer version installed as a dependency:

```
(venv) (qatk) sh$ pip uninstall scipy
```

To fully support Orb models with QuantumATK, you need to define a custom wrapper function, as the underlying ASE calculator expects an object as an argument (see note above). This wrapper function must be placed in a separate file in the same working directory, for example, `wrap_orb.py`

```
# wrap_orb.py
from orb_models.forcefield import pretrained
from orb_models.forcefield.calculator import ORBCalculator
```

(continues on next page)

(continued from previous page)

```
def wrap_orb_v2(device):
    orbff = pretrained.orb_v2(device=device)
    return ORBCalculator(orbff, device=device)
```

You can then use this wrapper in your workflow script to initialize the *ExternalMachineLearnedFFCalculator*:

```
from wrap_orb import wrap_orb_v2
calculator = ExternalMachineLearnedFFCalculator(wrap_orb_v2, device='cuda')
```

DeePMD-kit

Note: The following guide applies to DeePMD-kit version 3.0.2.

To install DeePMD-kit, use the following command. Note that due to incompatibilities in the precompiled binary packages, the library must be built from source using the `--no-binary deepmd-kit` flag:

```
(venv) (qatk) sh$ pip install deepmd-kit==3.0.2 --no-binary deepmd-kit
```

To use a DeePMD-kit model, you first need to download it from AIS Square and follow the specific instructions provided for that model. In this example, we use the model DPA-2.3.1-v3.0.0rc0 as an example. We will extract and freeze the `Domains_SemiCond` using the following command:

```
(venv) (qatk) sh$ dp --pt freeze -c DPA2_medium_28_10M_rc0.pt -o DPA2_SemiCond.pth --
↪head Domains_SemiCond
```

Once the model is prepared, it can be used with the *ExternalMachineLearnedFFCalculator* as shown below:

```
from deepmd.calculator import DP
calculator = ExternalMachineLearnedFFCalculator(
    DP,
    model='DPA2_SemiCond.pth'
)
```

MatterSim

Note: The following guide applies to MatterSim version 1.1.2.

MatterSim lists `atomate2` as a dependency, but this is not required for using the force field itself (as of version 1.1.2). Installing `atomate2` introduces numerous additional dependencies, many of which conflict with packages included in QuantumATK. To avoid these issues, we recommend installing only the necessary dependencies manually:

```
(venv) (qatk) sh$ pip install "e3nn>=0.5.0" "torch_geometric>=2.5.3" "torch_runstats>=0.
↪2.0" deprecated loguru
(venv) (qatk) sh$ pip install --no-deps mattersim==1.1.2
```

This command may raise an error related to the version of the `e3nn` library. MatterSim requires `e3nn>=0.5.0`, whereas QuantumATK includes `e3nn==0.4.4`. As a result, you will have `e3nn==0.4.4` in the QuantumATK shell environment and `e3nn>=0.5.0` in the virtual environment.

This version mismatch is generally not an issue, as Python prioritizes packages installed in the active virtual environment. Therefore, the correct version of `e3nn` will be used when running MatterSim. However, this setup does mean that MACE training functionality will not work within the QuantumATK shell environment while the virtual environment is active.

Below is a simple example demonstrating how to use MatterSim. This snippet sets up a calculator using the default MatterSim model and runs it on a GPU:

```
from mattersim.forcefield import MatterSimCalculator
calculator = ExternalMachineLearnedFFCalculator(
    MatterSimCalculator,
    device='cuda'
)
```

CHGnet

Note: The following guide applies to CHGNet version 0.3.8.

To install CHGNet, run the following command:

```
(venv) (qatk) sh$ pip install chgnet==0.3.8
```

Below is a simple example demonstrating how to use CHGNet. This snippet sets up a calculator using the default CHGNet model and runs it on a GPU:

```
from chgnet.model.dynamics import CHGNetCalculator
calculator = ExternalMachineLearnedFFCalculator(
    CHGNetCalculator,
    use_device='cuda'
)
```

FAIRChem UMA

Note: The following guide applies to FAIRChem version 2.2.0.

FAIRChem lists `numpy>2.0` as a dependency, but QuantumATK includes `numpy==1.26.4` as of the X-2025.06 release. Despite this version mismatch, it appears that FAIRChem models function correctly with the older version of NumPy when just using the force field itself. To install FAIRChem, run the following command:

```
(venv) (qatk) sh$ pip install "e3nn>=0.5.0" "numba>=0.61.2" "huggingface_hub>=0.27.1"␣
↪wandb lmdb tensorboard orjson submitit hydra-core torchtnt
(venv) (qatk) sh$ pip install --no-deps fairchem-core==2.2.0
```

Similar to MatterSim and SevenNet, this command may raise an error related to the version of the `e3nn` library, which can be ignored. FAIRChem UMA models require a custom wrapper function to be used with the *ExternalMachineLearnedFFCalculator*, as the underlying ASE calculator expects an object as an argument (see note above). This wrapper function must be placed in a separate file in the same working directory - for example, `wrap_uma.py`

```
# wrap_uma.py
from fairchem.core import FAIRChemCalculator
from fairchem.core.units.mlip_unit import load_predict_unit

def wrap_uma(ckpt="uma-s-1.pt", task_name="omat", device="cuda"):
    predictor = load_predict_unit(ckpt, device=device)
    calc = FAIRChemCalculator(predictor, task_name=task_name)
    return calc
```

The FAIRChem UMA model is available for download on the Hugging Face Hub, but access must be requested beforehand.

After downloading the model checkpoint, you can then use the wrapper in your workflow script to initialize the calculator:

```
from wrap_uma import wrap_uma
calculator = ExternalMachineLearnedFFCalculator(
    wrap_uma,
    ckpt='path/to/uma_model.pt',
    task_name='omat',
    device='cuda'
)
```

Other Models

The above list presents a selection of tested models. Other models may work as well, similar to the examples shown above. In many cases they can be installed using pip and the installation instructions that come with the open-source module. Incompatibilities during the installation process may in some cases require some extra technical workarounds, which can be tested safely in a virtual environment.

4.6 Atomic data

This appendix collects atomic data for elements and parameters for built-in basis sets in QuantumATK.

4.6.1 Element data

Table 4.29: Atomic data.

Z	Symbol	Name	Electron configuration	Valence configuration	Valence charge	U / eV
1	H	Hydrogen	$1s^1$	$1s^1$	1	12.848
2	He	Helium	$1s^2$	$1s^2$	2	9.0
3	Li	Lithium	[He]$2s^1$	$2s^1$	1	3.469
4	Be	Beryllium	[He]$2s^2$	$2s^2$	2	5.935
5	B	Boron	[He]$2s^2$ $2p^1$	$2s^2$ $2p^1$	3	8.0
6	C	Carbon	[He]$2s^2$ $2p^2$	$2s^2$ $2p^2$	4	10.207
7	N	Nitrogen	[He]$2s^2$ $2p^3$	$2s^2$ $2p^3$	5	11.052
8	O	Oxygen	[He]$2s^2$ $2p^4$	$2s^2$ $2p^4$	6	13.625
9	F	Fluorine	[He]$2s^2$ $2p^5$	$2s^2$ $2p^5$	7	15.054
10	Ne	Neon	[He]$2s^2$ $2p^6$	$2s^2$ $2p^6$	8	9.0

continues on next page

Table 4.29 – continued from previous page

Z	Symbol	Name	Electron configuration	Valence configuration	Valence charge	U / eV
11	Na	Sodium	[Ne]$3s^1$	$3s^1$	1	2.982
12	Mg	Magnesium	[Ne]$3s^2$	$3s^2$	2	4.623
13	Al	Aluminium	[Ne]$3s^2$ $3p^1$	$3s^2$ $3p^1$	3	5.682
14	Si	Silicon	[Ne]$3s^2$ $3p^2$	$3s^2$ $3p^2$	4	6.964
15	P	Phosphorus	[Ne]$3s^2$ $3p^3$	$3s^2$ $3p^3$	5	9.878
16	S	Sulfur	[Ne]$3s^2$ $3p^4$	$3s^2$ $3p^4$	6	9.205
17	Cl	Chlorine	[Ne]$3s^2$ $3p^5$	$3s^2$ $3p^5$	7	10.292
18	Ar	Argon	[Ne]$3s^2$ $3p^6$	$3s^2$ $3p^6$	8	9.0
19	K	Potassium	[Ar]$3d^0$ $4s^1$	$4s^1$	1	3.702
20	Ca	Calcium	[Ar]$3d^0$ $4s^2$	$4s^2$	2	3.977
21	Sc	Scandium	[Ar]$3d^1$ $4s^2$	$4s^2$ $3d^1$	3	9.0
22	Ti	Titanium	[Ar]$3d^2$ $4s^2$	$4s^2$ $3d^2$	4	9.0
23	V	Vanadium	[Ar]$3d^3$ $4s^2$	$4s^2$ $3d^3$	5	9.0
24	Cr	Chromium	[Ar]$3d^4$ $4s^2$	$4s^2$ $3d^4$	6	9.0
25	Mn	Manganese	[Ar]$3d^5$ $4s^2$	$4s^2$ $3d^5$	7	9.0
26	Fe	Iron	[Ar]$3d^6$ $4s^2$	$4s^2$ $3d^6$	8	9.0
27	Co	Cobalt	[Ar]$3d^7$ $4s^2$	$4s^2$ $3d^7$	9	9.0
28	Ni	Nickel	[Ar]$3d^8$ $4s^2$	$4s^2$ $3d^8$	10	9.0
29	Cu	Copper	[Ar]$3d^{10}$ $4s^1$	$4s^1$ $3d^{10}$	11	9.0
30	Zn	Zinc	[Ar]$3d^{10}$ $4s^2$	$4s^2$ $3d^{10}$	12	9.0
31	Ga	Gallium	[Ar]$3d^{10}$ $4s^2$ $4p^1$	$4s^2$ $4p^1$	3	5.936
32	Ge	Germanium	[Ar]$3d^{10}$ $4s^2$ $4p^2$	$4s^2$ $4p^2$	4	6.608
33	As	Arsenic	[Ar]$3d^{10}$ $4s^2$ $4p^3$	$4s^2$ $4p^3$	5	8.399
34	Se	Selenium	[Ar]$3d^{10}$ $4s^2$ $4p^4$	$4s^2$ $4p^4$	6	9.121
35	Br	Bromine	[Ar]$3d^{10}$ $4s^2$ $4p^5$	$4s^2$ $4p^5$	7	8.823
36	Kr	Krypton	[Ar]$3d^{10}$ $4s^2$ $4p^6$	$4s^2$ $4p^6$	8	9.0
37	Rb	Rubidium	[Kr]$4d^0$ $5s^1$	$5s^1$	1	2.495
38	Sr	Strontium	[Kr]$4d^0$ $5s^2$	$5s^2$	2	3.749
39	Y	Yttrium	[Kr]$4d^1$ $5s^2$	$5s^2$ $4d^1$	3	9.0
40	Zr	Zirconium	[Kr]$4d^2$ $5s^2$	$5s^2$ $4d^2$	4	9.0
41	Nb	Niobium	[Kr]$4d^4$ $5s^1$	$5s^2$ $4d^3$	5	9.0
42	Mo	Molybdenum	[Kr]$4d^5$ $5s^1$	$5s^1$ $4d^5$	6	9.0
43	Tc	Technetium	[Kr]$4d^5$ $5s^2$	$5s^2$ $4d^5$	7	9.0
44	Ru	Ruthenium	[Kr]$4d^7$ $5s^1$	$5s^2$ $4d^6$	8	9.0
45	Rh	Rhodium	[Kr]$4d^8$ $5s^1$	$5s^1$ $4d^8$	9	9.0
46	Pd	Palladium	[Kr]$4d^{10}$ $5s^0$	$4d^{10}$	10	9.0
47	Ag	Silver	[Kr]$4d^{10}$ $5s^1$	$5s^1$ $4d^{10}$	11	9.0
48	Cd	Cadmium	[Kr]$4d^{10}$ $5s^2$	$5s^2$ $4d^{10}$	12	9.0
49	In	Indium	[Kr]$4d^{10}$ $5s^2$ $5p^1$	$5s^2$ $5p^1$	3	5.53
50	Sn	Tin	[Kr]$4d^{10}$ $5s^2$ $5p^2$	$5s^2$ $5p^2$	4	4.297
51	Sb	Antimony	[Kr]$4d^{10}$ $5s^2$ $5p^3$	$5s^2$ $5p^3$	5	7.657
52	Te	Tellurium	[Kr]$4d^{10}$ $5s^2$ $5p^4$	$5s^2$ $5p^4$	6	8.985
53	I	Iodine	[Kr]$4d^{10}$ $5s^2$ $5p^5$	$5s^2$ $5p^5$	7	9.448
54	Xe	Xenon	[Kr]$4d^{10}$ $5s^2$ $5p^6$	$5s^2$ $5p^6$	8	9.0
55	Cs	Caesium	[Xe]$5d^0$ $6s^1$	$6s^1$ $5p^6$	7	9.0
56	Ba	Barium	[Xe]$5d^0$ $6s^2$	$6s^2$ $5p^6$	8	9.0
57	La	Lanthanum	[Xe]$5d^1$ $6s^2$	$6s^2$ $5d^1$	3	9.0
58	Ce	Cerium	[Xe]$4f^2$ $5d^0$ $6s^2$	$6s^2$ $4f^2$	4	9.0
59	Pr	Praseodymium	[Xe]$4f^3$ $5d^0$ $6s^2$	$6s^2$ $4f^3$	5	9.0
60	Nd	Neodymium	[Xe]$4f^4$ $5d^0$ $6s^2$	$6s^2$ $4f^4$	6	9.0

continues on next page

Table 4.29 – continued from previous page

Z	Symbol	Name	Electron configuration	Valence configuration	Valence charge	U / eV
61	Pm	Promethium	[Xe]$4f^5$ $5d^0$ $6s^2$	$6s^2$ $4f^5$	7	9.0
62	Sm	Samarium	[Xe]$4f^6$ $5d^0$ $6s^2$	$6s^2$ $4f^6$	8	9.0
63	Eu	Europium	[Xe]$4f^7$ $5d^0$ $6s^2$	$6s^2$ $4f^7$	9	9.0
64	Gd	Gadolinium	[Xe]$4f^7$ $5d^1$ $6s^2$	$6s^2$ $5d^1$ $4f^7$	10	9.0
65	Tb	Terbium	[Xe]$4f^9$ $5d^0$ $6s^2$	$6s^2$ $4f^9$	11	9.0
66	Dy	Dysprosium	[Xe]$4f^{10}$ $5d^0$ $6s^2$	$6s^2$ $4f^{10}$	12	9.0
67	Ho	Holmium	[Xe]$4f^{11}$ $5d^0$ $6s^2$	$6s^2$ $4f^{11}$	13	9.0
68	Er	Erbium	[Xe]$4f^{12}$ $5d^0$ $6s^2$	$6s^2$ $4f^{12}$	14	9.0
69	Tm	Thulium	[Xe]$4f^{13}$ $5d^0$ $6s^2$	$6s^2$ $4f^{13}$	15	9.0
70	Yb	Ytterbium	[Xe]$4f^{14}$ $5d^0$ $6s^2$	$6s^2$ $4f^{14}$	16	9.0
71	Lu	Lutetium	[Xe]$4f^{14}$ $5d^1$ $6s^2$	$6s^2$ $5d^1$ $4f^{14}$	17	9.0
72	Hf	Hafnium	[Xe]$4f^{14}$ $5d^2$ $6s^2$	$6s^2$ $5d^2$	4	9.0
73	Ta	Tantalum	[Xe]$4f^{14}$ $5d^3$ $6s^2$	$6s^2$ $5d^3$	5	9.0
74	W	Tungsten	[Xe]$4f^{14}$ $5d^4$ $6s^2$	$6s^1$ $5d^5$	6	9.0
75	Re	Rhenium	[Xe]$4f^{14}$ $5d^5$ $6s^2$	$6s^2$ $5d^5$	7	9.0
76	Os	Osmium	[Xe]$4f^{14}$ $5d^6$ $6s^2$	$6s^2$ $5d^6$	8	9.0
77	Ir	Iridium	[Xe]$4f^{14}$ $5d^7$ $6s^2$	$6s^2$ $5d^7$	9	9.0
78	Pt	Platinum	[Xe]$4f^{14}$ $5d^{10}$ $6s^0$	$6s^1$ $5d^9$	10	8.604
79	Au	Gold	[Xe]$4f^{14}$ $5d^{10}$ $6s^1$	$6s^1$ $5d^{10}$	11	8.604
80	Hg	Mercury	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$	$6s^2$ $5d^{10}$	12	9.0
81	Tl	Thallium	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^1$	$6s^2$ $6p^1$	3	9.0
82	Pb	Lead	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^2$	$6s^2$ $6p^2$	4	9.0
83	Bi	Bismuth	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^3$	$6s^2$ $6p^3$	5	9.0
84	Po	Polonium	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^4$	$6s^2$ $6p^4$	6	9.0
85	At	Astatine	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^5$	$6s^2$ $6p^5$	7	9.0
86	Rn	Radon	[Xe]$4f^{14}$ $5d^{10}$ $6s^2$ $6p^6$	$6s^2$ $6p^6$	8	9.0
87	Fr	Francium	[Rn]$7s^1$	$7s^1$ $6p^6$	7	9.0
88	Ra	Radium	[Rn]$7s^2$	$7s^2$ $6p^6$	8	9.0
89	Ac	Actinium	[Rn]$7s^2$ $6d^1$	$7s^2$ $6d^1$	3	9.0
90	Th	Thorium	[Rn]$7s^2$ $6d^2$	$7s^2$ $6d^2$	4	9.0
91	Pa	Protactinium	[Rn]$7s^2$ $5f^2$ $6d^1$	$7s^2$ $6d^1$ $5f^2$	5	9.0
92	U	Uranium	[Rn]$7s^2$ $5f^3$ $6d^1$	$7s^2$ $5f^3$ $6d^1$	6	9.0
93	Np	Neptunium	[Rn]$7s^2$ $5f^4$ $6d^1$	$7s^2$ $5f^4$ $6d^1$	7	9.0
94	Pu	Plutonium	[Rn]$7s^2$ $5f^6$	$7s^2$ $5f^6$	8	9.0
95	Am	Americium	[Rn]$7s^2$ $5f^7$	$7s^2$ $5f^7$	9	9.0
96	Cm	Curium	[Rn]$7s^2$ $5f^7$ $6d^1$	$7s^2$ $5f^7$ $6d^1$	10	9.0
97	Bk	Berkelium	[Rn]$7s^2$ $5f^9$	$7s^2$ $5f^9$	11	9.0
98	Cf	Californium	[Rn]$7s^2$ $5f^{10}$	$7s^2$ $5f^{10}$	12	9.0
99	Es	Einsteinium	[Rn]$7s^2$ $5f^{11}$	$7s^2$ $5f^{11}$	13	9.0
100	Fm	Fermium	[Rn]$7s^2$ $5f^{12}$	$7s^2$ $5f^{12}$	14	9.0
101	Md	Mendelevium	[Rn]$7s^2$ $5f^{13}$	$7s^2$ $5f^{13}$	15	9.0
102	No	Nobelium	[Rn]$7s^2$ $5f^{14}$	$7s^2$ $5f^{14}$	16	9.0

4.6.2 Built-in semiempirical parameter sets in QuantumATK

Slater–Koster basis sets

Table 4.30: Self-consistent Slater–Koster DFTB style parameters for electronic structure and total energy calculations. The basis set directory is used together with the functions DFTBDirectory and PairPotential. Some parameter sets are extensions to a base set, given in the Requires column; the base set is loaded automatically.

Basis set directory	Requires	Elements	Reference
cp2k/scc/		H, C, N, O, P, S, Zn	cp2k.org
cp2k/si-d/		H, O, Si	cp2k.org
dftb/hfo-1-1/		Hf, O	Elstner *et al.*[1]; Franke *et al.*[2]
dftb/matsci-0-3/		Al, O, H | Al, Si, O, H | Cu, Si, Al, Na, O, H | Ti, P, O, N, C, H | O, N, C, B, H | Al, O, C, H | Si, P, N, O, C, H	dftb.org
dftb/mio-1-1/		H, C, N, O, S, P	dftb.org
dftb/pbc-0-3/		Si, F, O, N, C, H | Fe - Fe	dftb.org
dftb/auorg-1-1/	mio	Au - (H, C, N, O, S, Au)	dftb.org
dftb/borg-0-1/	mio	B - (H, B)	dftb.org
dftb/chalc-0-1/	mio	As - (S, H, As)	dftb.org
dftb/halorg-0-1/	mio	F, Cl, Br, I, C, N, O, H, P, S	dftb.org
dftb/hyb-0-2/	mio	Ag - (O, C, H, Ag) | Ga - (As, S, O, C, H, Ga) | Si - (Ag, Ga, As, S, Si) | As - (S, H, As)	dftb.org
dftb/magsil-1-1/	matsci	Mg - (Si, O, H, Mg)	dftb.org
dftb/miomod-hh-0-1/	mio	H - H	dftb.org
dftb/miomod-nh-0-1/	mio	N - H	dftb.org
dftb/siband-1-1/		Si, O, H	dftb.org
dftb/tiorg-0-1/	mio	Ti - (C, H, N, O, S, Ti)	dftb.org
dftb/trans3d-0-1/	mio	Sc - (H, C, N, O, Sc) | Ti - (H, C, N, O, Ti) | Fe - (H, C, N, O, Fe) | Co - (H, C, N, O, Co) | Ni - (H, C, N, O, Ni)	dftb.org
dftb/znorg-0-1/	mio	Zn - (C, H, N, O, S, Zn)	dftb.org

Important: The parameters in the dftb directory are distributed by the dftb.org consortium under the Creative Commons Attribution-ShareAlike 4.0 International License. The use in part or in whole of this data is permitted only under the condition that the scientific background of the Licensed Material will be cited in any publications arising from its use. The required references are specified in the links to the corresponding dftb.org page, for each of the parameter sets in the table above.

[1] M.ăElstner, D.ăPorezag, G.ăJungnickel, J.ăElsner, M.ăHaugk, Th. Frauenheim, S.ăSuhai, and G.ăSeifert. Self-consistent-charge density-functional tight-binding method for simulations of complex materials properties. *Phys. Rev. B*, 58:7260–7268, Sep 1998. doi:10.1103/PhysRevB.58.7260.

[2] Dennis Franke, Christian Hettich, Thomas Köhler, Marcus Turowski, Henrik Ehlers, Detlev Ristau, and Thomas Frauenheim. Density functional based tight-binding parametrization of hafnium oxide: Simulations of amorphous structures. *Phys. Rev. B*, 98:075207, Aug 2018. URL: https://link.aps.org/doi/10.1103/PhysRevB.98.075207, doi:10.1103/PhysRevB.98.075207.

Table 4.31: Self-consistent Slater–Koster DFTB style parameters for electronic structure and total energy calculations using the Hotbit format. The basis set directory is used together with the functions HotbitDirectory and PairPotential.

Basis set directory	Elements	Reference
hotbit/standard	H, C, N	https://github.com/pekkosk/hotbit
hotbit/standard	H, C, O	https://github.com/pekkosk/hotbit
hotbit/standard	C, Au	https://github.com/pekkosk/hotbit
hotbit/standard	Na	https://github.com/pekkosk/hotbit
hotbit/inofficial	Na, Cl	https://github.com/pekkosk/hotbit
hotbit/inofficial	Mg	https://github.com/pekkosk/hotbit
hotbit/CC_pi	C (π-model faked as H)	https://github.com/pekkosk/hotbit

Table 4.32: NRL style parameters for electronic structure and total energy calculations.

Basis set name	Elements	Reference
NRLParameters.Si_sp	Si	Bernstein *et al.*[3]
NRLParameters.Si_spd	Si	Bernstein *et al.*[Page 4403, 3]
NRLParameters.Au	Au	Kirchhoff *et al.*[4]
NRLParameters.C	C	Papaconstantopoulos *et al.*[5]
NRLParameters.Ge_sp	Ge	Bernstein *et al.*[6]
NRLParameters.Ge_spd	Ge	Bernstein *et al.*[6]
NRLParameters.Al	Al	Yang *et al.*[7]
NRLParameters.SiC_sp	Si, C	Bernstein *et al.*[8]
NRLParameters.SiC_spd	Si, C	Bernstein *et al.*[8]

[3] N.ăBernstein, M.ăJ. Mehl, D.ăA. Papaconstantopoulos, N.ăI. Papanicolaou, MartinăZ. Bazant, and Efthimios Kaxiras. Energetic, vibrational, and electronic properties of silicon using a nonorthogonal tight-binding model. *Phys. Rev. B*, 62:4477–4487, Aug 2000. doi:10.1103/PhysRevB.62.4477.

[4] F.ăKirchhoff, M.ăJ. Mehl, N.ăI. Papanicolaou, D.ăA. Papaconstantopoulos, and F.ăS. Khan. Dynamical properties of Au from tight-binding molecular-dynamics simulations. *Phys. Rev. B*, 63:195101, Apr 2001. doi:10.1103/PhysRevB.63.195101.

[5] D.ăA. Papaconstantopoulos, M.ăJ. Mehl, S.ăC. Erwin, and M.ăR. Pederson. Tight-Binding Hamiltonians for Carbon and Silicon. *MRS Proc.*, 491:221, 1997. doi:10.1557/PROC-491-221.

[6] N.ăBernstein, M.ăJ. Mehl, and D.ăA. Papaconstantopoulos. Nonorthogonal tight-binding model for germanium. *Phys. Rev. B*, 66:075212, Aug 2002. doi:10.1103/PhysRevB.66.075212.

[7] SangăH. Yang, MichaelăJ. Mehl, and D.ăA. Papaconstantopoulos. Application of a tight-binding total-energy method for Al, Ga, and In. *Phys. Rev. B*, 57:R2013–R2016, Jan 1998. doi:10.1103/PhysRevB.57.R2013.

[8] N.ăBernstein, H.ăJ. Gotsis, D.ăA. Papaconstantopoulos, and M.ăJ. Mehl. Tight-binding calculations of the band structure and total energies of the various polytypes of silicon carbide. *Phys. Rev. B*, 71:075203, Feb 2005. doi:10.1103/PhysRevB.71.075203.

Non-self-consistent Slater–Koster parameters for electronic structure calculations

Basis set name	Elements	Reference
Bassani.AlP_Basis	Al, P	Jancu *et al.*[9]
Bassani.InAs_Basis	In, As	Jancu *et al.*[Page 4405, 9]
Bassani.GaP_Basis	Ga, P	Jancu *et al.*[Page 4405, 9]
Bassani.GeH_Basis	Ge, H	Jancu *et al.*[Page 4405, 9]
Bassani.GaSbH_Basis	Ga, Sb, H	Jancu *et al.*[Page 4405, 9]
Bassani.InPH_Basis	In, P, H	Jancu *et al.*[Page 4405, 9]
Bassani.InSb_Basis	In, Sb	Jancu *et al.*[Page 4405, 9]
Bassani.AlPH_Basis	Al, P, H	Jancu *et al.*[Page 4405, 9]
Bassani.Ge_Basis	Ge	Jancu *et al.*[Page 4405, 9]
Bassani.AlAs_Basis	Al, As	Jancu *et al.*[Page 4405, 9]
Bassani.InP_Basis	In, P	Jancu *et al.*[Page 4405, 9]
Bassani.GaAsH_Basis	Ga, As, H	Jancu *et al.*[Page 4405, 9]
Bassani.SiH_Basis	Si, H	Jancu *et al.*[Page 4405, 9]
Bassani.CH_Basis	C, H	Jancu *et al.*[Page 4405, 9]
Bassani.AlAsH_Basis	Al, As, H	Jancu *et al.*[Page 4405, 9]
Bassani.AlSbH_Basis	Al, Sb, H	Jancu *et al.*[Page 4405, 9]
Bassani.GaPH_Basis	Ga, P, H	Jancu *et al.*[Page 4405, 9]
Bassani.InAsH_Basis	In, As, H	Jancu *et al.*[Page 4405, 9]
Bassani.AlSb_Basis	Al, Sb	Jancu *et al.*[Page 4405, 9]
Bassani.GaSb_Basis	Ga, Sb	Jancu *et al.*[Page 4405, 9]
Bassani.GaAs_Basis	Ga, As	Jancu *et al.*[Page 4405, 9]
Bassani.C_Basis	C	Jancu *et al.*[Page 4405, 9]
Bassani.InSbH_Basis	In, Sb, H	Jancu *et al.*[Page 4405, 9]
Bassani.Si_Basis	Si	Jancu *et al.*[Page 4405, 9]
Boykin.SiH_Basis	Si, H	Boykin *et al.*[10], Zheng *et al.*[11]
Boykin.InAs_Basis	In, As	Boykin *et al.*[12]
Boykin.Si_Basis	Si	Boykin *et al.*[Page 4405, 10]
Boykin.GaAs_Basis	Ga, As	Boykin *et al.*[Page 4405, 12]
Boykin.Ge_Basis	Ge	Boykin *et al.*[Page 4405, 12]
Hancock.C_basis	C (π-model)	Hancock *et al.*[13]
Niquet.SiGeH_Basis, Niquet.SiGe_StrainParameters	Si, Ge, H	Niquet *et al.*[14]
Shinomiya.SiGe_Basis	Si, Ge	Shinomiya *et al.*[15]
Smidstrup.ZnO_H	Zn, O, H	Smidstrup[16]
Vogl.InAsH_Basis	In, As, H	Vogl *et al.*[17]
Vogl.InAs_Basis	In, As	Vogl *et al.*[Page 4405, 17]
Vogl.GaP_Basis	Ga, P	Vogl *et al.*[Page 4405, 17]
Vogl.GeH_Basis	Ge, H	Vogl *et al.*[Page 4405, 17]
Vogl.GaSbH_Basis	Ga, Sb, H	Vogl *et al.*[Page 4405, 17]
Vogl.InPH_Basis	In, P, H	Vogl *et al.*[Page 4405, 17]
Vogl.AlP_Basis	Al, P	Vogl *et al.*[Page 4405, 17]
Vogl.InSb_Basis	In, Sb	Vogl *et al.*[Page 4405, 17]
Vogl.ZnSeH_Basis	Zn, Se, H	Vogl *et al.*[Page 4405, 17]
Vogl.AlPH_Basis	Al, P, H	Vogl *et al.*[Page 4405, 17]
Vogl.Ge_Basis	Ge	Vogl *et al.*[Page 4405, 17]
Vogl.AlAs_Basis	Al, As	Vogl *et al.*[Page 4405, 17]
Vogl.ZnTe_Basis	Zn, Te	Vogl *et al.*[Page 4405, 17]
Vogl.InP_Basis	In, P	Vogl *et al.*[Page 4405, 17]

continues on next page

Table 4.33 – continued from previous page

Basis set name	Elements	Reference
Vogl.GaAsH_Basis	Ga, As, H	Vogl *et al.*[Page 4405, 17]
Vogl.SiH_Basis	Si, H	Vogl *et al.*[17]
Vogl.CH_Basis	C, H	Vogl *et al.*[17]
Vogl.AlAsH_Basis	Al, As, H	Vogl *et al.*[17]
Vogl.ZnTeH_Basis	Zn, Te, H	Vogl *et al.*[17]
Vogl.GaAs_Basis	Ga, As	Vogl *et al.*[17]
Vogl.AlSbH_Basis	Al, Sb, H	Vogl *et al.*[17]
Vogl.ZnSe_Basis	Zn, Se	Vogl *et al.*[17]
Vogl.Sn_Basis	Sn	Vogl *et al.*[17]
Vogl.GaPH_Basis	Ga, P, H	Vogl *et al.*[17]
Vogl.SnH_Basis	Sn, H	Vogl *et al.*[17]
Vogl.AlSb_Basis	Al, Sb	Vogl *et al.*[17]
Vogl.GaSb_Basis	Ga, Sb	Vogl *et al.*[17]
Vogl.C_Basis	C	Vogl *et al.*[17]
Vogl.InSbH_Basis	In, Sb, H	Vogl *et al.*[17]
Vogl.Si_Basis	Si	Vogl *et al.*[17]

Extended Hückel basis sets

- *Hoffmann basis set parameters*
- *Müller basis set parameters*
- *Cerda basis set parameters*

Hoffmann Hückel basis set parameters

The Hoffmann Hückel basis set parameters. The unit of E^{ion} is eV, while the units of w_i and η_i are Bohr$^{-3/2}$ and Bohr^{-1}, respectively.

H	Z = 1		CerdaHuckelParameters.Hydrogen_C2H4_Basis					N_V = 1.1988
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	1	0	-17.8384	0.50494	0.87223			U = [12.848] eV
								E_{VAC} = -6.2568 eV

[9] J.-M. Jancu, R.ăScholz, F.ăBeltram, and F.ăBassani. Empirical spds* tight-binding calculation for cubic semiconductors: General method and material parameters. *Phys. Rev. B*, 57:6493–6507, Mar 1998. doi:10.1103/PhysRevB.57.6493.

[10] T.ăB. Boykin, G.ăKlimeck, and F.ăOyafuso. Valence band effective-mass expressions in the sp3d5s* empirical tight-binding model applied to a Si and Ge parametrization. *Phys. Rev. B*, 69:115201, 2004. doi:10.1103/PhysRevB.69.115201.

[11] Y.ăZheng, C.ăRivas, R.ăLake, K.ăAlam, T.ăB. Boykin, and G.ăKlimeck. Electronic properties of silicon nanowires. *IEEE Transactions on Electron Devices*, 52(6):1097–1103, 2005. doi:10.1109/TED.2005.848077.

[12] TimothyăB. Boykin, Gerhard Klimeck, R.ăChris Bowen, and Fabiano Oyafuso. Diagonal parameter shifts due to nearest-neighbor displacements in empirical tight-binding theory. *Phys. Rev. B*, 66:125207, Sep 2002. doi:10.1103/PhysRevB.66.125207.

[13] Y.ăHancock, K.ăSaloriutta, A.ăUppstu, A.ăHarju, and M.ăJ. Puska. Spin-Dependence in Asymmetric, V-Shaped-Notched Graphene Nanoribbons. *J. Low Temp. Phys.*, 153(5):393–398, 2008. doi:10.1007/s10909-008-9838-y.

[14] Y.ăM. Niquet, D.ăRideau, C.ăTavernier, H.ăJaouen, and X.ăBlase. Onsite matrix elements of the tight-binding Hamiltonian of a strained crystal: Application to silicon, germanium, and their alloys. *Phys. Rev. B*, 79:245201, Jun 2009. URL: https://link.aps.org/doi/10.1103/PhysRevB.79.245201, doi:10.1103/PhysRevB.79.245201.

[15] TimothyăB. Boykin, Gerhard Klimeck, R.ăChris Bowen, and Fabiano Oyafuso. Nonorthogonal Tight-Binding Molecular Dynamics for Si(1-x)Ge(x) Alloys. *Mem. Fac. Engrg. Okayama Uni.*, 35:63–75, 2001. URL: http://ousar.lib.okayama-u.ac.jp/15358.

[16] Soren Smidstrup. Nam donec per potenti aptent etiam aliquet. Masters thesis, Aalborg University, 2005.

[17] P.ăVogl, HaroldăP. Hjalmarson, and J.ăD. Dow. A Semi-empirical tight-binding theory of the electronic structure of semiconductors. *J. Phys. Chem. Solids*, 44(5):365–378, jan 1983. URL: http://www.sciencedirect.com/science/article/pii/0022369783900641.

B	Z = 5		CerdaHuckelParameters.Boron_BN_wurtz_Basis					N_V = 3.82181
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	2	0	-22.6243	0.66811	1.60645			U = [8, 8, 8] eV
	2	1	-16.6839	0.87705	1.63627			E_{VAC} = -6.0 eV
	3	2	-9.36241	0.70894	0.89284			

B	Z = 5		CerdaHuckelParameters.Boron_BN_hexag_Basis					N_V = 3.82181
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	2	0	-21.7174	0.66811	1.60645			U = [8, 8, 8] eV
	2	1	-14.1676	0.99022	1.63627			E_{VAC} = -6.0 eV
	3	2	-5.55352	0.6544	0.89284			

C	Z = 6		CerdaHuckelParameters.Carbon_graphite_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.8
	2	0	-19.8892	0.76422	2.0249			U = [10.0082, 10.0082, 10.0082] eV
	2	1	-13.08	0.27152	1.62412	0.73886	2.17687	E_{VAC} = -7.36576789 eV
	3	2	-2.04759	0.49066	1.1944			

C	Z = 6		CerdaHuckelParameters.Carbon_diamond_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.8
	2	0	-21.3959	0.71927	2.0029			U = [9.99619, 9.99619, 9.99619] eV
	2	1	-15.2271	0.55108	1.63935	0.53135	3.48317	E_{VAC} = -9.432846 eV
	3	2	-4.06791	0.88307	0.86366			

C	Z = 6		CerdaHuckelParameters.Carbon_GW_diamond_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.8
	2	0	-22.5994	0.77991	2.10199			U = [10.009, 10.009, 10.009] eV
	2	1	-14.7745	0.09615	1.00122	0.93068	2.16332	E_{VAC} = -9.4142981 eV
	3	2	-3.42637	0.69317	0.91626			

C	Z = 6		CerdaHuckelParameters.Carbon_GW_SiC_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.8
	2	0	-19.4265	0.77991	2.10199			U = [10.207, 10.207, 10.207] eV
	2	1	-12.8029	0.09614	1.00122	0.93069	2.16335	E_{VAC} = -6.14175 eV
	3	2	-1.62663	0.69317	0.91626			

C	Z = 6		CerdaHuckelParameters.Carbon_C2H4_Basis					N_V = 3.60238
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.8
	2	0	-18.4006	0.7449	2.04927			U = [10.207, 10.207, 10.207] eV
	2	1	-12.3453	0.74031	1.71703	0.3554	3.31214	E_{VAC} = -6.2568 eV
	3	2	-3.53572	0.76455	0.94043			

N	Z = 7		CerdaHuckelParameters.Nitrogen_BN_wurtz_Basis					N_V = 4.17819
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	2	0	-27.127	1	2.74251			U = [11.052, 11.052, 11.052] eV
	2	1	-17.5276	1	2.26145			E_{VAC} = -6.0 eV
	3	2	-5.60857	0.37978	0.62169			

N	Z = 7		CerdaHuckelParameters.Nitrogen_BN_hexag_Basis					N_V = 4.17819
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	2	0	-26.4216	1	2.74251			U = [11.052, 11.052, 11.052] eV
	2	1	-17.0156	1	2.26145			E_{VAC} = -6.0 eV
	3	2	-3.59026	0.51048	0.62169			

Al	Z = 13		CerdaHuckelParameters.Aluminium_fcc_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-12.9807	0.72564	1.62763			U = [5.9724, 5.9724, 5.9724] eV
	3	1	-8.90927	0.71507	1.28508			E_{VAC} = -5.74486 eV
	3	2	-4.66958	0.77928	0.88276			

Al	Z = 13		CerdaHuckelParameters.Aluminium_AlP_GW_Basis					N_V = 2.82759
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-14.3073	0.72564	1.62763			U = [5.682, 5.682, 5.682] eV
	3	1	-9.67525	0.71507	1.28508			E_{VAC} = -6.4482586 eV
	3	2	-4.88276	0.77928	0.88276			

Al	Z = 13		CerdaHuckelParameters.Aluminium_AlAs_GW_Basis					N_V = 2.81319
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-14.7568	0.72564	1.62763			U = [5.682, 5.682, 5.682] eV
	3	1	-10.2301	0.71507	1.28508			E_{VAC} = -6.8946 eV
	3	2	-4.7515	0.77928	0.88276			

Si	Z = 14		CerdaHuckelParameters.Silicon_diamond_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-18.0388	0.65715	1.78044			U = [7.08, 7.08, 7.08] eV
	3	1	-11.8203	0.81839	1.61427			E_{VAC} = -7.756917 eV
	3	2	-6.3232	0.56273	0.79668	0.65807	2.49718	

Si	Z = 14		CerdaHuckelParameters.Silicon_GW_diamond_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-18.1026	0.70366	1.83611			U = [7.08252, 7.08252, 7.08252] eV
	3	1	-11.253	0.0277	0.78901	0.98313	1.70988	E_{VAC} = -7.67047 eV
	3	2	-5.34706	0.68383	0.68292	0.46957	1.72102	

Si	Z = 14		CerdaHuckelParameters.Silicon_GW_SiC_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-19.4736	0.70366	1.83611			U = [6.964, 6.964, 6.964] eV
	3	1	-11.9665	0.0277	0.78901	0.98313	1.70988	E_{VAC} = -6.14175 eV
	3	2	-6.45414	0.68383	0.68292	0.46957	1.72102	

P	Z = 15		CerdaHuckelParameters.Phosphorus_AlP_GW_Basis					N_V = 5.172410000000001
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-20.9897	0.64007	1.9419			U = [9.878, 9.878, 9.878] eV
	3	1	-12.8889	0.71876	1.68957	0.344	3.05784	E_{VAC} = -6.4482586 eV
	3	2	-3.60739	0.79107	0.89643			

P	Z = 15		CerdaHuckelParameters.Phosphorus_GaP_GW_Basis					N_V = 5.13542
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-21.3273	0.64007	1.9419			U = [9.88788, 9.88788, 9.88788] eV
	3	1	-13.1613	0.71876	1.68957	0.344	3.05784	E_{VAC} = -6.88753 eV
	3	2	-4.58537	0.79107	0.89643			

P	Z = 15		CerdaHuckelParameters.Phosphorus_InP_GW_Basis					N_V = 5.44879
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	3	0	-21.2204	0.64007	1.9419			U = [9.878, 9.878, 9.878] eV
	3	1	-13.2123	0.71876	1.68957	0.344	3.05784	E_{VAC} = -6.92207 eV
	3	2	-4.89407	0.79107	0.89643			

Sc	Z = 21		CerdaHuckelParameters.Scandium_hcp_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-8.36905	0.65566	1.41412			U = [4.36767, 4.36767, 8.58] eV
	4	1	-5.82003	0.66104	1.16027			E_{VAC} = -6.4737072 eV
	3	2	-8.36623	0.58803	1.39314			

Sc	Z = 21		CerdaHuckelParameters.Scandium_bcc_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-8.27398	0.65566	1.41412			U = [4.39116, 4.39116, 8.58] eV
	4	1	-5.68802	0.66104	1.16027			E_{VAC} = -6.470326 eV
	3	2	-8.43933	0.58803	1.39314			

Ti	Z = 22		CerdaHuckelParameters.Titanium_hcp_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-7.84958	0.45558	0.89722			U = [4.11888, 4.11888, 9.386] eV
	4	1	-7.32609	0.49494	1.00909			E_{VAC} = -5.5560624 eV
	3	2	-9.42441	0.28536	1.15327	0.81231	2.63915	

Ti	Z = 22		CerdaHuckelParameters.Titanium_fcc_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-7.70802	0.45558	0.89722			U = [4.13142, 4.13142, 9.386] eV
	4	1	-7.21524	0.49494	1.00909			E_{VAC} = -5.4606502 eV
	3	2	-9.27229	0.28536	1.15327	0.81231	2.63915	

Ti	Z = 22		CerdaHuckelParameters.Titanium_bcc_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-7.61918	0.45558	0.89722			U = [4.11491, 4.11491, 9.386] eV
	4	1	-7.129	0.49494	1.00909			E_{VAC} = -5.881899 eV
	3	2	-9.08015	0.28536	1.15327	0.81231	2.63915	

V	Z = 23		CerdaHuckelParameters.Vanadium_bcc_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-7.902	0.88449	1.51793			U = [4.8865, 4.8865, 10.1] eV
	4	1	-5.37821	0.66078	1.30693			E_{VAC} = -5.8156347 eV
	3	2	-9.44229	0.62062	1.68218			

Cr	Z = 24		CerdaHuckelParameters.Chromium_bcc_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-8.07927	0.64199	0.99961			U = [4.41663, 4.41663, 10.75] eV
	4	1	-7.26305	0.58547	1.10362			E_{VAC} = -5.6996914 eV
	3	2	-10.7296	0.33576	1.45656	0.7694	3.27523	

Mn	Z = 25		CerdaHuckelParameters.Manganese_fcc_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-8.93247	0.60538	1.39823			U = [4.98099, 4.98099, 11.35] eV
	4	1	-6.56812	0.59765	1.20449			E_{VAC} = -5.921038 eV
	3	2	-10.8763	0.3231	1.52037	0.78115	3.44592	

Mn	Z = 25		CerdaHuckelParameters.Manganese_bcc_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-9.14276	0.60538	1.39823			U = [4.91258, 4.91258, 11.35] eV
	4	1	-6.92114	0.59765	1.20449			E_{VAC} = -5.91948471 eV
	3	2	-11.2151	0.3231	1.52037	0.78115	3.44592	

Fe	Z = 26		CerdaHuckelParameters.Iron_bcc_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-9.39204	0.5892	1.4884			U = [5.13145, 5.13145, 11.92] eV
	4	1	-6.66953	0.59596	1.25262			E_{VAC} = -5.20999981 eV
	3	2	-11.4024	0.26494	1.48912	0.82466	3.3483	

Co	Z = 27		CerdaHuckelParameters.Cobalt_fcc_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-9.90499	0.53969	1.48379			U = [5.18117, 5.18117, 12.47] eV
	4	1	-6.97352	0.59446	1.2556			E_{VAC} = -4.95579 eV
	3	2	-11.7395	0.24393	1.4872	0.84379	3.41646	

Co	Z = 27		CerdaHuckelParameters.Cobalt_hcp_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-10.0487	0.53969	1.48379			U = [5.14334, 5.14334, 12.47] eV
	4	1	-7.18148	0.59446	1.2556			E_{VAC} = -5.0868314 eV
	3	2	-11.7495	0.24393	1.4872	0.84379	3.41646	

Ni	Z = 28		CerdaHuckelParameters.Nickel_fcc_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-9.7934	0.60557	1.58729			U = [5.36667, 5.36667, 12.99] eV
	4	1	-6.69928	0.61588	1.29812			E_{VAC} = -4.6469427 eV
	3	2	-11.7794	0.29586	1.63992	0.81374	3.94689	

Cu	Z = 29		CerdaHuckelParameters.Copper_fcc_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-10.5742	0.6063	1.69318			U = [4.34198, 4.34198, 11.22] eV
	4	1	-6.86172	0.63888	1.33731			E_{VAC} = -5.1780533 eV
	3	2	-12.8961	0.32382	1.79662	0.82655	5.14331	

Zn	Z = 30		CerdaHuckelParameters.Zinc_fcc_Basis					N_V = 12.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-12.9625	1	2.36615			U = [4.98932, 4.98932, 13.99] eV
	4	1	-9.23345	0.4167	1.63016			E_{VAC} = -6.67286 eV
	3	2	-18.035	0.15524	2.01067	0.87781	3.565	

Ga	Z = 31		CerdaHuckelParameters.Gallium_fcc_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-14.5101	0.67038	2.03484			U = [6.32012, 6.32012, 6.32012] eV
	4	1	-8.56214	0.71632	1.63083			E_{VAC} = -5.6833707 eV
	4	2	-3.16883	0.89901	0.90717			

Ga	Z = 31		CerdaHuckelParameters.Gallium_GaP_GW_Basis					N_V = 2.86458
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-16.3708	0.67038	2.03484			U = [5.936, 5.936, 5.936] eV
	4	1	-10.0152	0.71632	1.63083			E_{VAC} = -6.88753 eV
	4	2	-3.79622	0.89901	0.90717			

Ga	Z = 31		CerdaHuckelParameters.Gallium_GaAs_Basis					N_V = 3.7153800000000006
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-17.2549	0.57998	1.93753			U = [5.936, 5.936, 5.936] eV
	4	1	-11.2889	0.64383	1.58044			E_{VAC} = -7.60579 eV
	4	2	-6.68155	0.65154	0.9494			

Ga	Z = 31		CerdaHuckelParameters.Gallium_GaAs_GW_Basis					N_V = 2.97286
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-16.9187	0.67038	2.03484			U = [5.936, 5.936, 5.936] eV
	4	1	-10.7539	0.71632	1.63083			E_{VAC} = -7.41 eV
	4	2	-3.96693	0.89901	0.90717			

As	Z = 33		CerdaHuckelParameters.Arsenic_AlAs_GW_Basis					N_V = 5.1868099999999995
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-21.9456	0.54624	2.19335			U = [8.399, 8.399, 8.399] eV
	4	1	-13.0744	0.7307	1.9686			E_{VAC} = -6.8946024 eV
	4	2	-3.38857	0.9902	0.94359			

As	Z = 33		CerdaHuckelParameters.Arsenic_GaAs_Basis					N_V = 4.28462
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-20.6851	0.62857	2.2775			U = [8.399, 8.399, 8.399] eV
	4	1	-12.1021	0.999	2.28904			E_{VAC} = -7.60579 eV
	4	2	-3.36392	0.9902	1.67268			

As	Z = 33		CerdaHuckelParameters.Arsenic_GaAs_GW_Basis					N_V = 5.02714
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-22.1472	0.54624	2.19335			U = [8.399, 8.399, 8.399] eV
	4	1	-13.1706	0.7307	1.9686			E_{VAC} = -7.41 eV
	4	2	-4.137	0.9902	0.94359			

As	Z = 33		CerdaHuckelParameters.Arsenic_InAs_GW_Basis					N_V = 5.40319
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	4	0	-22.0247	0.54624	2.19335			U = [8.399, 8.399, 8.399] eV
	4	1	-13.2078	0.7307	1.9686			E_{VAC} = -7.5043 eV
	4	2	-4.6414	0.9902	0.94359			

Y	Z = 39		CerdaHuckelParameters.Yttrium_fcc_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-8.45573	0.66414	1.62475			U = [4.21408, 4.21408, 6.529] eV
	5	1	-5.03686	0.72293	1.30361			E_{VAC} = -6.0 eV
	4	2	-8.01171	0.68038	1.58038			

Y	Z = 39		CerdaHuckelParameters.Yttrium_hcp_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-8.47712	0.66414	1.62475			U = [4.18435, 4.18435, 6.529] eV
	5	1	-5.08407	0.72293	1.30361			E_{VAC} = -6.0 eV
	4	2	-7.99731	0.68038	1.58038			

Zr	Z = 40		CerdaHuckelParameters.Zirconium_hcp_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-8.21277	0.66589	1.56164			U = [4.48138, 4.48138, 7.307] eV
	5	1	-6.00133	0.53606	1.2688			E_{VAC} = -5.9781231 eV
	4	2	-9.03884	0.61574	1.66473	0.64065	5.0542	

Zr	Z = 40		CerdaHuckelParameters.Zirconium_fcc_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-8.08839	0.66589	1.56164			U = [4.32906, 4.32906, 7.307] eV
	5	1	-5.73102	0.53606	1.2688			E_{VAC} = -5.9683992 eV
	4	2	-8.94122	0.61574	1.66473	0.64065	5.0542	

Zr	Z = 40		CerdaHuckelParameters.Zirconium_bcc_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-7.8852	0.66589	1.56164			U = [4.55723, 4.55723, 7.307] eV
	5	1	-5.58256	0.53606	1.2688			E_{VAC} = -6.026187 eV
	4	2	-8.74286	0.61574	1.66473	0.64065	5.0542	

Nb	Z = 41		CerdaHuckelParameters.Niobium_bcc_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-7.83515	0.78125	1.66071			U = [4.34576, 4.34576, 6.92] eV
	5	1	-5.03085	0.68753	1.27372			E_{VAC} = -5.79099796 eV
	4	2	-9.3306	0.68669	1.85049			

Mo	Z = 42		CerdaHuckelParameters.Molybdenum_bcc_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-8.36662	0.83018	1.72218			U = [4.26433, 4.26433, 7.689] eV
	5	1	-5.39352	0.72696	1.3365			E_{VAC} = -5.194213 eV
	4	2	-10.8671	0.67217	1.9978			

Tc Z = 43 CerdaHuckelParameters.Technetium_hcp_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-8.50569	0.77154	1.64956		
5	1	-5.74739	0.63636	1.36162		
4	2	-11.4215	0.45903	1.87926	0.66939	3.90392

N_V = 7.0
β = 2.3
U = [5.02966, 5.02966, 9.207] eV
E_{VAC} = -7.5148955 eV

Tc Z = 43 CerdaHuckelParameters.Technetium_fcc_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-8.2267	0.77154	1.64956		
5	1	-5.70711	0.63636	1.36162		
4	2	-11.3582	0.45903	1.87926	0.66939	3.90392

N_V = 7.0
β = 2.3
U = [5.02277, 5.02277, 9.207] eV
E_{VAC} = -7.4558211 eV

Ru Z = 44 CerdaHuckelParameters.Ruthenium_hcp_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-9.32602	0.60076	1.63558		
5	1	-6.34065	0.58839	1.37066		
4	2	-12.4218	0.27066	1.71196	0.80993	3.31356

N_V = 8.0
β = 2.3
U = [4.16207, 4.16207, 8.907] eV
E_{VAC} = -5.5914432 eV

Ru Z = 44 CerdaHuckelParameters.Ruthenium_fcc_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-8.84268	0.60076	1.63558		
5	1	-6.06874	0.58839	1.37066		
4	2	-11.9364	0.27066	1.71196	0.80993	3.31356

N_V = 8.0
β = 2.3
U = [4.19692, 4.19692, 8.907] eV
E_{VAC} = -5.1774836 eV

Rh Z = 45 CerdaHuckelParameters.Rhodium_fcc_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-9.22928	0.72992	1.86894		
5	1	-5.15799	0.71084	1.40344		
4	2	-12.433	0.41591	2.00068	0.70707	4.15195

N_V = 9.0
β = 2.3
U = [4.56908, 4.56908, 9.455] eV
E_{VAC} = -4.9623528 eV

Pd Z = 46 CerdaHuckelParameters.Palladium_fcc_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-9.07916	0.63338	1.77352		
5	1	-5.8727	0.64064	1.43651		
4	2	-12.3746	0.31704	1.9082	0.78406	3.88573

N_V = 10.0
β = 2.3
U = [5.772, 5.772, 9.047] eV
E_{VAC} = -4.387488 eV

Ag Z = 47 CerdaHuckelParameters.Silver_fcc_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-10.3592	0.61985	1.91661		
5	1	-6.20001	0.66651	1.50982		
4	2	-14.5109	0.28692	2.01944	0.80921	4.14345

N_V = 11.0
β = 2.3
U = [5.06366, 5.06366, 10.48] eV
E_{VAC} = -5.43566 eV

Cd	Z = 48		CerdaHuckelParameters.Cadmium_fcc_Basis					N_V = 12.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-12.3344	0.60994	2.10439			U = [5.21713, 5.21713, 11.64] eV
	5	1	-7.85805	0.5758	1.66558			E_{VAC} = -5.7843316 eV
	4	2	-19.8661	0.93688	14.8833	0.32073	2.31681	

In	Z = 49		CerdaHuckelParameters.Indium_fcc_Basis					N_V = 3.0000000000000004
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-13.9371	0.59101	2.11757			U = [5.83205, 5.83205, 5.83205] eV
	5	1	-8.22964	0.71392	1.84269			E_{VAC} = -5.904437 eV
	5	2	-3.54891	0.61732	1.02658			

In	Z = 49		CerdaHuckelParameters.Indium_InP_GW_Basis					N_V = 2.55121
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-15.9796	0.59101	2.11757			U = [5.53, 5.53, 5.53] eV
	5	1	-9.05671	0.71392	1.84269			E_{VAC} = -6.92207 eV
	5	2	-3.77804	0.61732	1.02658			

In	Z = 49		CerdaHuckelParameters.Indium_InAs_GW_Basis					N_V = 2.5968099999999996
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	5	0	-16.4496	0.59101	2.11757			U = [5.53, 5.53, 5.53] eV
	5	1	-9.83157	0.71392	1.84269			E_{VAC} = -7.5043 eV
	5	2	-3.6995	0.61732	1.02658			

W	Z = 74		CerdaHuckelParameters.Tungsten_bcc_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	6	0	-10.5819	0.80244	2.30427			U = [4.69614, 4.69614, 8.228] eV
	6	1	-5.53905	0.68049	1.72283			E_{VAC} = -5.6645015 eV
	5	2	-10.7394	0.69067	2.3028			

Re	Z = 75		CerdaHuckelParameters.Rhenium_hcp_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	6	0	-10.004	0.20733	1.6507	0.84382	2.6424	U = [4.80643, 4.80643, 8.704] eV
	6	1	-5.98158	0.76128	1.58733			E_{VAC} = -5.453634 eV
	5	2	-11.3579	0.65008	2.33586			

Ir	Z = 77		CerdaHuckelParameters.Iridium_fcc_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	6	0	-12.1196	0.64865	2.32336			U = [4.88371, 4.88371, 9.555] eV
	6	1	-6.44135	0.69422	1.80105			E_{VAC} = -4.2382022 eV
	5	2	-13.1059	0.32075	2.16073	0.77521	4.01172	

Pt	Z = 78		CerdaHuckelParameters.Platinum_fcc_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	6	0	-11.7091	0.65212	2.32872			U = [3.97472, 3.97472, 8.579] eV
	6	1	-6.08105	0.7288	1.77159			E_{VAC} = -4.0783659 eV
	5	2	-13.301	0.38501	2.30384	0.74491	4.63467	

Au	Z = 79		CerdaHuckelParameters.Gold_fcc_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 2.3
	6	0	-12.1342	0.60326	2.31605			U = [4.37908, 4.37908, 9.852] eV
	6	1	-6.73967	0.62661	1.74458			E_{VAC} = -4.8533 eV
	5	2	-14.0257	0.37598	2.32731	0.79437	5.44496	

Müller Hückel basis set parameters

The Müller Hückel basis set parameters. The unit of E^{ion} is eV, while the units of w_i and η_i are Bohr$^{-3/2}$ and Bohr^{-1}, respectively.

H	Z = 1		HoffmannHuckelParameters.Hydrogen_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	1	0	-13.6	1	1.3			U = [12.848] eV
								E_{VAC} = 0.0 eV

He	Z = 2		HoffmannHuckelParameters.Helium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	1	0	-23.4	1	1.688			U = [9] eV
								E_{VAC} = 0.0 eV

Li	Z = 3		HoffmannHuckelParameters.Lithium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-5.4	1	0.65			U = [3.469, 3.469] eV
	2	1	-3.5	1	0.65			E_{VAC} = 0.0 eV

Be	Z = 4		HoffmannHuckelParameters.Beryllium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-10	1	0.975			U = [5.935, 5.935] eV
	2	1	-6	1	0.975			E_{VAC} = 0.0 eV

B	Z = 5		HoffmannHuckelParameters.Boron_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-15.2	1	1.3			U = [8, 8] eV
	2	1	-8.5	1	1.3			E_{VAC} = 0.0 eV

C Z = 6 HoffmannHuckelParameters.Carbon_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
2	0	-21.4	1	1.625		
2	1	-11.4	1	1.625		

N_V = 4.0
β = 1.75
U = [10.207, 10.207] eV
E_{VAC} = 0.0 eV

N Z = 7 HoffmannHuckelParameters.Nitrogen_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
2	0	-26	1	1.95		
2	1	-13.4	1	1.95		

N_V = 5.0
β = 1.75
U = [11.052, 11.052] eV
E_{VAC} = 0.0 eV

O Z = 8 HoffmannHuckelParameters.Oxygen_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
2	0	-32.3	1	2.275		
2	1	-14.8	1	2.275		

N_V = 6.0
β = 1.75
U = [13.625, 13.625] eV
E_{VAC} = 0.0 eV

F Z = 9 HoffmannHuckelParameters.Fluorine_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
2	0	-40	1	2.425		
2	1	-18.1	1	2.425		

N_V = 7.0
β = 1.75
U = [15.054, 15.054] eV
E_{VAC} = 0.0 eV

Ne Z = 10 HoffmannHuckelParameters.Neon_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
2	0	-43.2	1	2.879		
2	1	-20	1	2.879		

N_V = 8.0
β = 1.75
U = [9, 9] eV
E_{VAC} = 0.0 eV

Na Z = 11 HoffmannHuckelParameters.Sodium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
3	0	-5.1	1	0.733		
3	1	-3	1	0.733		

N_V = 1.0
β = 1.75
U = [2.982, 2.982] eV
E_{VAC} = 0.0 eV

Mg Z = 12 HoffmannHuckelParameters.Magnesium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
3	0	-9	1	1.1		
3	1	-4.5	1	1.1		

N_V = 2.0
β = 1.75
U = [4.623, 4.623] eV
E_{VAC} = 0.0 eV

Al Z = 13 HoffmannHuckelParameters.Aluminium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
3	0	-12.3	1	1.167		
3	1	-6.5	1	1.167		

N_V = 3.0
β = 1.75
U = [5.682, 5.682] eV
E_{VAC} = 0.0 eV

Si	Z = 14		HoffmannHuckelParameters.Silicon_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-17.3	1	1.383			U = [6.964, 6.964] eV
	3	1	-9.2	1	1.383			E_{VAC} = 0.0 eV

P	Z = 15		HoffmannHuckelParameters.Phosphorus_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-18.6	1	1.75			U = [9.878, 9.878] eV
	3	1	-14	1	1.3			E_{VAC} = 0.0 eV

S	Z = 16		HoffmannHuckelParameters.Sulfur_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-20	1	2.122			U = [9.205, 9.205] eV
	3	1	-11	1	1.827			E_{VAC} = 0.0 eV

Cl	Z = 17		HoffmannHuckelParameters.Chlorine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-26.3	1	2.183			U = [10.292, 10.292] eV
	3	1	-14.2	1	1.733			E_{VAC} = 0.0 eV

K	Z = 19		HoffmannHuckelParameters.Potassium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-4.34	1	0.874			U = [3.702, 3.702] eV
	4	1	-2.73	1	0.874			E_{VAC} = 0.0 eV

Ca	Z = 20		HoffmannHuckelParameters.Calcium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7	1	1.2			U = [9, 9] eV
	4	1	-4	1	1.2			E_{VAC} = 0.0 eV

Sc	Z = 21		HoffmannHuckelParameters.Scandium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-8.87	1	1.3			U = [9, 9, 9] eV
	4	1	-2.75	1	1.3			E_{VAC} = 0.0 eV
	3	2	-8.51	0.422783	4.35	0.72757	1.7	

Ti	Z = 22		HoffmannHuckelParameters.Titanium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-8.97	1	1.075			U = [9, 9, 9] eV
	4	1	-5.44	1	1.075			E_{VAC} = 0.0 eV
	3	2	-10.81	0.420607	4.55	0.783913	1.4	

V	Z = 23		HoffmannHuckelParameters.Vanadium_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-8.81	1	1.3			U = [9, 9, 9] eV
	4	1	-5.52	1	1.3			E_{VAC} = 0.0 eV
	3	2	-11	0.475509	4.75	0.705213	1.7	

Cr	Z = 24		HoffmannHuckelParameters.Chromium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-8.66	1	1.7			U = [9, 9, 9] eV
	4	1	-5.24	1	1.7			E_{VAC} = 0.0 eV
	3	2	-11.22	0.505792	4.95	0.674723	1.8	

Mn	Z = 25		HoffmannHuckelParameters.Manganese_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-9.75	1	0.97			U = [9, 9, 9] eV
	4	1	-5.89	1	0.97			E_{VAC} = 0.0 eV
	3	2	-11.67	0.513906	5.15	0.692909	1.7	

Fe	Z = 26		HoffmannHuckelParameters.Iron_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-9.1	1	1.9			U = [9, 9, 9] eV
	4	1	-5.32	1	1.9			E_{VAC} = 0.0 eV
	3	2	-12.6	0.5505	5.35	0.626	2	

Co	Z = 27		HoffmannHuckelParameters.Cobalt_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-9.21	1	2			U = [9, 9, 9] eV
	4	1	-5.29	1	2			E_{VAC} = 0.0 eV
	3	2	-13.18	0.567865	5.55	0.605856	2.1	

Ni	Z = 28		HoffmannHuckelParameters.Nickel_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-10.95	1	2.1			U = [9, 9, 9] eV
	4	1	-6.27	1	2.1			E_{VAC} = 0.0 eV
	3	2	-14.2	0.549306	5.75	0.60817	2.3	

Cu	Z = 29		HoffmannHuckelParameters.Copper_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-11.4	1	2.2			U = [9, 9, 9] eV
	4	1	-6.06	1	2.2			E_{VAC} = 0.0 eV
	3	2	-14	0.593322	5.95	0.574421	2.3	

Zn	Z = 30		HoffmannHuckelParameters.Zinc_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-12.41	1	2.01			U = [9, 9] eV
	4	1	-6.53	1	1.7			E_{VAC} = 0.0 eV

Ga	Z = 31		HoffmannHuckelParameters.Gallium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-14.58	1	1.77			U = [5.936, 5.936] eV
	4	1	-6.75	1	1.55			E_{VAC} = 0.0 eV

Ge	Z = 32		HoffmannHuckelParameters.Germanium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-16	1	2.16			U = [6.608, 6.608] eV
	4	1	-9	1	1.85			E_{VAC} = 0.0 eV

As	Z = 33		HoffmannHuckelParameters.Arsenic_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-16.22	1	2.23			U = [8.399, 8.399] eV
	4	1	-12.16	1	1.89			E_{VAC} = 0.0 eV

Se	Z = 34		HoffmannHuckelParameters.Selenium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-20.5	1	2.44			U = [9.121, 9.121] eV
	4	1	-14.4	1	2.07			E_{VAC} = 0.0 eV

Br	Z = 35		HoffmannHuckelParameters.Bromine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-22.07	1	2.588			U = [8.823, 8.823] eV
	4	1	-13.1	1	2.131			E_{VAC} = 0.0 eV

Rb	Z = 37		HoffmannHuckelParameters.Rubidium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-4.18	1	0.997			U = [2.495, 2.495] eV
	5	1	-2.6	1	0.997			E_{VAC} = 0.0 eV

Sr	Z = 38		HoffmannHuckelParameters.Strontium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.62	1	1.214			U = [9, 9] eV
	5	1	-3.92	1	1.214			E_{VAC} = 0.0 eV

Zr	Z = 40		HoffmannHuckelParameters.Zirconium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-9.87	1	1.817			U = [9, 9, 9] eV
	5	1	-6.76	1	1.776			E_{VAC} = 0.0 eV
	4	2	-11.18	0.622416	3.835	0.578216	1.505	

Nb	Z = 41		HoffmannHuckelParameters.Niobium_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-10.1	1	1.89			U = [9, 9, 9] eV
	5	1	-6.86	1	1.85			E_{VAC} = 0.0 eV
	4	2	-12.1	0.640101	4.08	0.551601	1.64	

Mo	Z = 42		HoffmannHuckelParameters.Molybdenum_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-8.34	1	1.96			U = [9, 9, 9] eV
	5	1	-5.24	1	1.9			E_{VAC} = 0.0 eV
	4	2	-10.5	0.589879	4.54	0.589879	1.9	

Tc	Z = 43		HoffmannHuckelParameters.Technetium_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-10.07	1	2.018			U = [9, 9, 9] eV
	5	1	-5.4	1	1.984			E_{VAC} = 0.0 eV
	4	2	-12.82	0.5715	4.9	0.6012	2.094	

Ru	Z = 44		HoffmannHuckelParameters.Ruthenium_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-10.4	1	2.08			U = [9, 9, 9] eV
	5	1	-6.87	1	2.04			E_{VAC} = 0.0 eV
	4	2	-14.9	0.534242	5.38	0.636789	2.3	

Rh	Z = 45		HoffmannHuckelParameters.Rhodium_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-8.09	1	2.135			U = [9, 9, 9] eV
	5	1	-4.57	1	2.1			E_{VAC} = 0.0 eV
	4	2	-12.5	0.580698	4.29	0.568498	1.97	

Pd	Z = 46		HoffmannHuckelParameters.Palladium_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-7.32	1	2.19			U = [9, 9, 9] eV
	5	1	-3.75	1	2.152			E_{VAC} = 0.0 eV
	4	2	-12.02	0.526436	5.983	0.637334	2.613	

Cd	Z = 48		HoffmannHuckelParameters.Cadmium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-11.8	1	1.64			U = [9, 9] eV
	5	1	-8.2	1	1.6			E_{VAC} = 0.0 eV

In	Z = 49		HoffmannHuckelParameters.Indium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-12.6	1	1.903			U = [5.53, 5.53] eV
	5	1	-6.19	1	1.677			E_{VAC} = 0.0 eV

Sn	Z = 50		HoffmannHuckelParameters.Tin_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-16.16	1	2.12			U = [4.297, 4.297] eV
	5	1	-8.32	1	1.82			E_{VAC} = 0.0 eV

Sb	Z = 51		HoffmannHuckelParameters.Antimony_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-18.8	1	2.323			U = [7.657, 7.657] eV
	5	1	-11.7	1	1.999			E_{VAC} = 0.0 eV

Te	Z = 52		HoffmannHuckelParameters.Tellurium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-20.8	1	2.51			U = [8.985, 8.985] eV
	5	1	-14.8	1	2.16			E_{VAC} = 0.0 eV

I	Z = 53		HoffmannHuckelParameters.Iodine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-18	1	2.679			U = [9.448, 9.448] eV
	5	1	-12.7	1	2.322			E_{VAC} = 0.0 eV

Cs	Z = 55		HoffmannHuckelParameters.Caesium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-3.88	1	1.06			U = [9, 9] eV
	6	1	-2.49	1	1.06			E_{VAC} = 0.0 eV

La	Z = 57		HoffmannHuckelParameters.Lanthanum_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-7.67	1	2.14			U = [9, 9, 9] eV
	6	1	-5.01	1	2.08			E_{VAC} = 0.0 eV
	5	2	-8.21	0.776515	3.78	0.458609	1.381	

Ce Z = 58 HoffmannHuckelParameters.Cerium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.97	1	1.799		
6	1	-4.97	1	1.799		
5	2	-6.43	1	2.747		
4	3	-11.28	1	3.907		

N_V = 4.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Sm Z = 62 HoffmannHuckelParameters.Samarium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.86	1	1.4		
6	1	-4.86	1	1.4		
5	2	-6.06	0.718435	2.747	0.444722	1.267
4	3	-11.28	0.735406	6.907	0.459703	2.639

N_V = 8.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Gd Z = 64 HoffmannHuckelParameters.Gadolinium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-5.44	1	1.369		
5	2	-6.06	0.718435	2.747	0.444722	1.267
4	3	-11.28	0.735406	6.907	0.459703	2.639

N_V = 10.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Yb Z = 70 HoffmannHuckelParameters.Ytterbium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-5.35	1	1.54		
6	1	-5.35	1	1.54		
5	2	-5.21	0.706344	2.81	0.48343	1.216
4	3	-13.86	0.746226	8.629	0.456538	3.198

N_V = 16.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Lu Z = 71 HoffmannHuckelParameters.Lutetium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-6.05	1	1.666		
6	1	-6.05	1	1.666		
5	2	-5.12	0.704411	2.813	0.488008	1.21
4	3	-22.4	0.733001	9.136	0.445901	3.666

N_V = 17.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Ta Z = 73 HoffmannHuckelParameters.Tantalum_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-10.1	1	2.28		
6	1	-6.86	1	2.241		
5	2	-12.1	0.610612	4.762	0.610612	1.938

N_V = 5.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

W Z = 74 HoffmannHuckelParameters.Tungsten_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-8.26	1	2.341		
6	1	-5.17	1	2.309		
5	2	-10.37	0.668535	4.982	0.542438	2.068

N_V = 6.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Re Z = 75 HoffmannHuckelParameters.Rhenium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-9.36	1	2.398		
6	1	-5.96	1	2.372		
5	2	-12.66	0.637752	5.343	0.565763	2.277

N_V = 7.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Os Z = 76 HoffmannHuckelParameters.Osmium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-8.17	1	2.452		
6	1	-4.81	1	2.429		
5	2	-11.84	0.637195	5.571	0.559795	2.416

N_V = 8.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Ir Z = 77 HoffmannHuckelParameters.Iridium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-11.36	1	2.5		
6	1	-4.5	1	2.2		
5	2	-12.17	0.635059	5.796	0.555605	2.557

N_V = 9.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Pt Z = 78 HoffmannHuckelParameters.Platinum_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-9.077	1	2.554		
6	1	-5.475	1	2.554		
5	2	-12.59	0.633378	6.013	0.551281	2.696

N_V = 10.0
β = 1.75
U = [8.604, 8.604, 8.604] eV
E_{VAC} = 0.0 eV

Au Z = 79 HoffmannHuckelParameters.Gold_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-10.92	1	2.602		
6	1	-5.55	1	2.584		
5	2	-15.07	0.644177	6.163	0.535576	2.794

N_V = 11.0
β = 1.75
U = [8.604, 8.604, 8.604] eV
E_{VAC} = 0.0 eV

Hg Z = 80 HoffmannHuckelParameters.Mercury_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-13.68	1	2.649		
6	1	-8.47	1	2.631		
5	2	-17.5	0.643776	6.436	0.521481	3.032

N_V = 12.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Tl Z = 81 HoffmannHuckelParameters.Thallium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-11.6	1	2.3		
6	1	-5.8	1	1.6		

N_V = 3.0
β = 1.75
U = [9, 9] eV
E_{VAC} = 0.0 eV

Pb	Z = 82		HoffmannHuckelParameters.Lead_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-15.7	1	2.35			U = [9, 9] eV
	6	1	-8	1	2.06			E_{VAC} = 0.0 eV

Bi	Z = 83		HoffmannHuckelParameters.Bismuth_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-15.19	1	2.56			U = [9, 9] eV
	6	1	-7.79	1	2.072			E_{VAC} = 0.0 eV

Th	Z = 90		HoffmannHuckelParameters.Thorium_Basis					N_V = 0.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	3	-9.64	0.768182	4.477	0.42669	1.837	U = [9, 9, 9, 9, 9] eV
	7	0	-5.39	1	1.834			E_{VAC} = 0.0 eV
	7	1	-5.39	1	1.834			
	6	2	-10.11	0.761171	2.461	0.407085	1.165	
	5	3	-9.64	0.768182	4.477	0.42669	1.837	

U	Z = 92		HoffmannHuckelParameters.Uranium_Basis					N_V = 0.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	7	0	-5.5	1	1.914			U = [9, 9, 9, 9] eV
	7	1	-5.5	1	1.914			E_{VAC} = 0.0 eV
	6	2	-9.19	0.760793	2.581	0.412596	1.207	
	5	3	-10.62	0.784411	4.943	0.390806	2.106	

Cerda Hückel basis set parameters

The Cerda Hückel basis set parameters. The unit of E^{ion} is eV, while the units of w_i and η_i are Bohr$^{-3/2}$ and Bohr^{-1}, respectively.

H	Z = 1		MullerHuckelParameters.Hydrogen_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	1	0	-13.606	1	1			U = [12.848] eV
								E_{VAC} = 0.0 eV

He	Z = 2		MullerHuckelParameters.Helium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	1	0	-24.979	1	1.618			U = [9] eV
								E_{VAC} = 0.0 eV

Li	Z = 3		MullerHuckelParameters.Lithium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-5.342	1	0.645			U = [3.469, 3.469] eV
	2	1	-3.499	1	0.524			E_{VAC} = 0.0 eV

Be	Z = 4		MullerHuckelParameters.Beryllium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-8.416	1	0.944			U = [5.935, 5.935] eV
	2	1	-5.632	1	0.875			E_{VAC} = 0.0 eV

B	Z = 5		MullerHuckelParameters.Boron_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-13.462	1	1.265			U = [8, 8] eV
	2	1	-8.432	1	1.134			E_{VAC} = 0.0 eV

C	Z = 6		MullerHuckelParameters.Carbon_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-19.376	1	1.576			U = [10.207, 10.207] eV
	2	1	-11.072	1	1.435			E_{VAC} = 0.0 eV

N	Z = 7		MullerHuckelParameters.Nitrogen_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-26.223	1	1.885			U = [11.052, 11.052] eV
	2	1	-13.841	1	1.728			E_{VAC} = 0.0 eV

O	Z = 8		MullerHuckelParameters.Oxygen_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-34.024	1	2.192			U = [13.625, 13.625] eV
	2	1	-16.768	1	2.018			E_{VAC} = 0.0 eV

F	Z = 9		MullerHuckelParameters.Fluorine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-42.791	1	2.497			U = [15.054, 15.054] eV
	2	1	-19.865	1	2.305			E_{VAC} = 0.0 eV

Ne	Z = 10		MullerHuckelParameters.Neon_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	2	0	-52.529	1	2.802			U = [9, 9] eV
	2	1	-23.141	1	2.59			E_{VAC} = 0.0 eV

Na	Z = 11		MullerHuckelParameters.Sodium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-4.955	1	0.832			U = [2.982, 2.982] eV
	3	1	-2.976	1	0.611			E_{VAC} = 0.0 eV

Mg	Z = 12		MullerHuckelParameters.Magnesium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-6.886	1	1.076			U = [4.623, 4.623] eV
	3	1	-4.169	1	0.862			E_{VAC} = 0.0 eV

Al	Z = 13		MullerHuckelParameters.Aluminium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-10.706	1	1.347			U = [5.682, 5.682] eV
	3	1	-5.713	1	1.019			E_{VAC} = 0.0 eV

Si	Z = 14		MullerHuckelParameters.Silicon_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-14.792	1	1.588			U = [6.964, 6.964] eV
	3	1	-7.585	1	1.256			E_{VAC} = 0.0 eV

P	Z = 15		MullerHuckelParameters.Phosphorus_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-19.221	1	1.816			U = [9.878, 9.878] eV
	3	1	-9.539	1	1.478			E_{VAC} = 0.0 eV

S	Z = 16		MullerHuckelParameters.Sulfur_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-24.019	1	2.035			U = [9.205, 9.205] eV
	3	1	-11.601	1	1.691			E_{VAC} = 0.0 eV

Cl	Z = 17		MullerHuckelParameters.Chlorine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-29.196	1	2.25			U = [10.292, 10.292] eV
	3	1	-13.78	1	1.9			E_{VAC} = 0.0 eV

Ar	Z = 18		MullerHuckelParameters.Argon_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	3	0	-34.759	1	2.461			U = [9, 9] eV
	3	1	-16.083	1	2.105			E_{VAC} = 0.0 eV

K	Z = 19		MullerHuckelParameters.Potassium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-4.013	1	0.858			U = [3.702, 3.702, 3.702] eV
	4	1	-2.599	1	0.671			E_{VAC} = 0.0 eV
	3	2	-1.581	1	0.364			

Ca	Z = 20		MullerHuckelParameters.Calcium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-5.321	1	1.067			U = [9, 9, 9] eV
	4	1	-3.573	1	0.893			E_{VAC} = 0.0 eV
	3	2	-3.337	1	1.906			

Sc	Z = 21		MullerHuckelParameters.Scandium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-5.717	1	1.136			U = [9, 9, 9] eV
	4	1	-3.739	1	0.951			E_{VAC} = 0.0 eV
	3	2	-9.353	0.4599	3.807	0.701847	1.455	

Ti	Z = 22		MullerHuckelParameters.Titanium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-6.039	1	1.195			U = [9, 9, 9] eV
	4	1	-3.863	1	0.998			E_{VAC} = 0.0 eV
	3	2	-11.043	0.468935	4.218	0.685904	1.664	

V	Z = 23		MullerHuckelParameters.Vanadium_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-6.325	1	1.248			U = [9, 9, 9] eV
	4	1	-3.963	1	1.041			E_{VAC} = 0.0 eV
	3	2	-12.549	0.474002	4.6	0.677004	1.846	

Cr	Z = 24		MullerHuckelParameters.Chromium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-6.583	1	1.296			U = [9, 9, 9] eV
	4	1	-4.042	1	1.077			E_{VAC} = 0.0 eV
	3	2	-13.909	0.474986	4.978	0.67298	2.022	

Mn	Z = 25		MullerHuckelParameters.Manganese_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-6.84	1	1.344			U = [9, 9, 9] eV
	4	1	-4.12	1	1.113			E_{VAC} = 0.0 eV
	3	2	-15.27	0.480857	5.318	0.665802	2.176	

Fe	Z = 26		MullerHuckelParameters.Iron_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7.079	1	1.388			U = [9, 9, 9] eV
	4	1	-4.183	1	1.145			E_{VAC} = 0.0 eV
	3	2	-16.541	0.484872	5.653	0.660825	2.325	

Co	Z = 27		MullerHuckelParameters.Cobalt_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7.309	1	1.431			U = [9, 9, 9] eV
	4	1	-4.237	1	1.174			E_{VAC} = 0.0 eV
	3	2	-17.77	0.485886	5.996	0.658846	2.476	

Ni	Z = 28		MullerHuckelParameters.Nickel_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7.532	1	1.473			U = [9, 9, 9] eV
	4	1	-4.284	1	1.203			E_{VAC} = 0.0 eV
	3	2	-18.969	0.486001	6.339	0.658001	2.625	

Cu	Z = 29		MullerHuckelParameters.Copper_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7.755	1	1.515			U = [9, 9, 9] eV
	4	1	-4.331	1	1.232			E_{VAC} = 0.0 eV
	3	2	-20.188	0.486895	6.676	0.656859	2.768	

Zn	Z = 30		MullerHuckelParameters.Zinc_Basis					N_V = 12.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-7.96	1	1.553			U = [9, 9, 9] eV
	4	1	-4.361	1	1.251			E_{VAC} = 0.0 eV
	3	2	-21.294	0.48724	7.015	0.656323	2.911	

Ga	Z = 31		MullerHuckelParameters.Gallium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-11.554	1	1.808			U = [5.936, 5.936] eV
	4	1	-5.674	1	1.314			E_{VAC} = 0.0 eV

Ge	Z = 32		MullerHuckelParameters.Germanium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-15.158	1	2.024			U = [6.608, 6.608] eV
	4	1	-7.329	1	1.55			E_{VAC} = 0.0 eV

As	Z = 33		MullerHuckelParameters.Arsenic_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-18.916	1	2.222			U = [8.399, 8.399] eV
	4	1	-8.984	1	1.757			E_{VAC} = 0.0 eV

Se	Z = 34		MullerHuckelParameters.Selenium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	4	0	-22.862	1	2.409			U = [9.121, 9.121] eV
	4	1	-10.681	1	1.949			E_{VAC} = 0.0 eV

Br Z = 35 MullerHuckelParameters.Bromine_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
4	0	-27.013	1	2.588		
4	1	-12.438	1	2.131		

N_V = 7.0
β = 1.75
U = [8.823, 8.823] eV
E_{VAC} = 0.0 eV

Kr Z = 36 MullerHuckelParameters.Krypton_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
4	0	-31.373	1	2.762		
4	1	-14.264	1	2.306		

N_V = 8.0
β = 1.75
U = [9, 9] eV
E_{VAC} = 0.0 eV

Rb Z = 37 MullerHuckelParameters.Rubidium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-3.752	1	0.977		
5	1	-2.452	1	0.774		
4	2	-1.636	1	0.523		

N_V = 1.0
β = 1.75
U = [2.495, 2.495, 2.495] eV
E_{VAC} = 0.0 eV

Sr Z = 38 MullerHuckelParameters.Strontium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-4.856	1	1.187		
5	1	-3.299	1	1		
4	2	-3.179	1	1.694		

N_V = 2.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Y Z = 39 MullerHuckelParameters.Yttrium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-5.337	1	1.279		
5	1	-3.515	1	1.079		
4	2	-6.799	0.601963	2.554	0.577964	1.068

N_V = 3.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Zr Z = 40 MullerHuckelParameters.Zirconium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-5.735	1	1.36		
5	1	-3.694	1	1.151		
4	2	-8.461	0.650366	2.769	0.508286	1.224

N_V = 4.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Nb Z = 41 MullerHuckelParameters.Niobium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
5	0	-5.923	1	1.403		
5	1	-3.757	1	1.186		
4	2	-10.001	0.685854	2.955	0.461902	1.333

N_V = 5.0
β = 1.75
U = [9, 9, 9] eV
E_{VAC} = 0.0 eV

Mo	Z = 42		MullerHuckelParameters.Molybdenum_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.112	1	1.446			U = [9, 9, 9] eV
	5	1	-3.819	1	1.221			E_{VAC} = 0.0 eV
	4	2	-11.541	0.71711	3.126	0.426065	1.408	

Tc	Z = 43		MullerHuckelParameters.Technetium_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.3	1	1.49			U = [9, 9, 9] eV
	5	1	-3.882	1	1.256			E_{VAC} = 0.0 eV
	4	2	-13.08	0.742534	3.293	0.39875	1.468	

Ru	Z = 44		MullerHuckelParameters.Ruthenium_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.488	1	1.533			U = [9, 9, 9] eV
	5	1	-3.945	1	1.291			E_{VAC} = 0.0 eV
	4	2	-14.62	0.779822	3.429	0.367916	1.453	

Rh	Z = 45		MullerHuckelParameters.Rhodium_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.677	1	1.576			U = [9, 9, 9] eV
	5	1	-4.008	1	1.326			E_{VAC} = 0.0 eV
	4	2	-16.16	0.80486	3.577	0.347939	1.452	

Pd	Z = 46		MullerHuckelParameters.Palladium_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-6.865	1	1.619			U = [9, 9, 9] eV
	5	1	-4.07	1	1.362			E_{VAC} = 0.0 eV
	4	2	-17.7	0.814894	3.746	0.337956	1.501	

Ag	Z = 47		MullerHuckelParameters.Silver_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-7.054	1	1.662			U = [9, 9, 9] eV
	5	1	-4.133	1	1.397			E_{VAC} = 0.0 eV
	4	2	-19.24	0.824441	3.912	0.328777	1.545	

Cd	Z = 48		MullerHuckelParameters.Cadmium_Basis					N_V = 12.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-7.242	1	1.706			U = [9, 9, 9] eV
	5	1	-4.196	1	1.432			E_{VAC} = 0.0 eV
	4	2	-20.78	0.823957	4.094	0.324983	1.64	

In	Z = 49		MullerHuckelParameters.Indium_Basis				N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-10.141	1	1.934			U = [5.53, 5.53] eV
	5	1	-5.368	1	1.456			E_{VAC} = 0.0 eV

Sn	Z = 50		MullerHuckelParameters.Tin_Basis				N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-13.043	1	2.129			U = [4.297, 4.297] eV
	5	1	-6.764	1	1.674			E_{VAC} = 0.0 eV

Sb	Z = 51		MullerHuckelParameters.Antimony_Basis				N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-16.026	1	2.305			U = [7.657, 7.657] eV
	5	1	-8.143	1	1.863			E_{VAC} = 0.0 eV

Te	Z = 52		MullerHuckelParameters.Tellurium_Basis				N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-19.122	1	2.47			U = [8.985, 8.985] eV
	5	1	-9.54	1	2.036			E_{VAC} = 0.0 eV

I	Z = 53		MullerHuckelParameters.Iodine_Basis				N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-22.344	1	2.626			U = [9.448, 9.448] eV
	5	1	-10.971	1	2.198			E_{VAC} = 0.0 eV

Xe	Z = 54		MullerHuckelParameters.Xenon_Basis				N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	5	0	-25.699	1	2.776			U = [9, 9] eV
	5	1	-12.444	1	2.352			E_{VAC} = 0.0 eV

Cs	Z = 55		MullerHuckelParameters.Caesium_Basis				N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-3.365	1	1.031			U = [9, 9, 9] eV
	6	1	-2.286	1	0.845			E_{VAC} = 0.0 eV
	5	2	-1.844	1	0.906			

Ba	Z = 56		MullerHuckelParameters.Barium_Basis				N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.287	1	1.236			U = [9, 9, 9] eV
	6	1	-3.063	1	1.071			E_{VAC} = 0.0 eV
	5	2	-4.143	1	1.96			

La	Z = 57		MullerHuckelParameters.Lanthanum_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.637	1	1.318			U = [9, 9, 9] eV
	6	1	-3.233	1	1.142			E_{VAC} = 0.0 eV
	5	2	-7.316	0.593989	3.153	0.611989	1.338	

Ce	Z = 58		MullerHuckelParameters.Cerium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.697	1	1.335			U = [9, 9, 9, 9] eV
	6	1	-3.261	1	1.157			E_{VAC} = 0.0 eV
	5	2	-7.359	1	2.027			
	4	3	-11.95	1	4.22			

Pr	Z = 59		MullerHuckelParameters.Praseodymium_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.466	1	1.288			U = [9, 9, 9, 9] eV
	6	1	-3.29	1	1.127			E_{VAC} = 0.0 eV
	5	2	-7.376	1	2.055			
	4	3	-12.94	1	4.415			

Nd	Z = 60		MullerHuckelParameters.Neodymium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.518	1	1.303			U = [9, 9, 9, 9] eV
	6	1	-3.318	1	1.187			E_{VAC} = 0.0 eV
	5	2	-7.373	1	2.08			
	4	3	-13.892	1	4.619			

Pm	Z = 61		MullerHuckelParameters.Promethium_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.586	1	1.317			U = [9, 9, 9, 9] eV
	6	1	-3.346	1	1.201			E_{VAC} = 0.0 eV
	5	2	-7.353	1	2.103			
	4	3	-14.901	1	4.81			

Sm	Z = 62		MullerHuckelParameters.Samarium_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.616	1	1.331			U = [9, 9, 9, 9] eV
	6	1	-3.375	1	1.216			E_{VAC} = 0.0 eV
	5	2	-7.32	1	2.122			
	4	3	-15.719	1	4.991			

Eu Z = 63 MullerHuckelParameters.Europium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.663	1	1.345		
6	1	-3.403	1	1.231		
5	2	-7.276	1	2.14		
4	3	-16.451	1	5.165		

N_V = 9.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Gd Z = 64 MullerHuckelParameters.Gadolinium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-5.033	1	1.427		
6	1	-3.431	1	1.246		
5	2	-7.22	1	2.155		
4	3	-17.079	1	5.33		

N_V = 10.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Tb Z = 65 MullerHuckelParameters.Terbium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.753	1	1.371		
6	1	-3.46	1	1.261		
5	2	-7.156	1	2.169		
4	3	-17.707	1	5.495		

N_V = 11.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Dy Z = 66 MullerHuckelParameters.Dysprosium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.797	1	1.383		
6	1	-3.488	1	1.276		
5	2	-7.084	1	2.181		
4	3	-18.246	1	5.654		

N_V = 12.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Ho Z = 67 MullerHuckelParameters.Holmium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.84	1	1.396		
6	1	-3.516	1	1.291		
5	2	-7.004	1	2.19		
4	3	-18.733	1	5.808		

N_V = 13.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Er Z = 68 MullerHuckelParameters.Erbium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
6	0	-4.882	1	1.408		
6	1	-3.545	1	1.306		
5	2	-6.917	1	2.199		
4	3	-19.174	1	5.96		

N_V = 14.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Tm	Z = 69		MullerHuckelParameters.Thulium_Basis					N_V = 15.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.924	1	1.42			U = [9, 9, 9, 9] eV
	6	1	-3.573	1	1.32			E_{VAC} = 0.0 eV
	5	2	-6.824	1	2.205			
	4	3	-19.571	1	6.109			

Yb	Z = 70		MullerHuckelParameters.Ytterbium_Basis					N_V = 16.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-4.965	1	1.432			U = [9, 9, 9, 9] eV
	6	1	-3.601	1	1.335			E_{VAC} = 0.0 eV
	5	2	-6.726	1	2.21			
	4	3	-19.929	1	6.256			

Lu	Z = 71		MullerHuckelParameters.Lutetium_Basis					N_V = 17.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-5.411	1	1.526			U = [9, 9, 9, 9] eV
	6	1	-3.63	1	1.35			E_{VAC} = 0.0 eV
	5	2	-6.622	1	2.213			
	4	3	-20.313	1	6.404			

Hf	Z = 72		MullerHuckelParameters.Hafnium_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-5.725	1	1.598			U = [9, 9, 9] eV
	6	1	-3.658	1	1.365			E_{VAC} = 0.0 eV
	5	2	-8.141	0.636594	3.337	0.545652	1.505	

Ta	Z = 73		MullerHuckelParameters.Tantalum_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-5.979	1	1.658			U = [9, 9, 9] eV
	6	1	-3.757	1	1.417			E_{VAC} = 0.0 eV
	5	2	-9.569	0.673894	3.478	0.495922	1.606	

W	Z = 74		MullerHuckelParameters.Tungsten_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-6.196	1	1.711			U = [9, 9, 9] eV
	6	1	-3.836	1	1.463			E_{VAC} = 0.0 eV
	5	2	-10.963	0.705262	3.609	0.456169	1.683	

Re	Z = 75		MullerHuckelParameters.Rhenium_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-6.386	1	1.76			U = [9, 9, 9] eV
	6	1	-3.902	1	1.503			E_{VAC} = 0.0 eV
	5	2	-12.348	0.732016	3.734	0.424009	1.742	

Os	Z = 76		MullerHuckelParameters.Osmium_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-6.556	1	1.804			U = [9, 9, 9] eV
	6	1	-3.958	1	1.54			E_{VAC} = 0.0 eV
	5	2	-13.737	0.757831	3.851	0.394912	1.782	

Ir	Z = 77		MullerHuckelParameters.Iridium_Basis					N_V = 9.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-6.71	1	1.845			U = [9, 9, 9] eV
	6	1	-4.006	1	1.574			E_{VAC} = 0.0 eV
	5	2	-15.136	0.779917	3.968	0.37096	1.813	

Pt	Z = 78		MullerHuckelParameters.Platinum_Basis					N_V = 10.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-6.875	1	1.888			U = [8.604, 8.604, 8.604] eV
	6	1	-4.034	1	1.593			E_{VAC} = 0.0 eV
	5	2	-16.528	0.79785	4.084	0.351934	1.84	

Au	Z = 79		MullerHuckelParameters.Gold_Basis					N_V = 11.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-7.037	1	1.931			U = [8.604, 8.604, 8.604] eV
	6	1	-4.068	1	1.615			E_{VAC} = 0.0 eV
	5	2	-17.922	0.813667	4.2	0.335863	1.861	

Hg	Z = 80		MullerHuckelParameters.Mercury_Basis					N_V = 12.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-7.104	1	1.953			U = [9, 9, 9] eV
	6	1	-4.117	1	1.66			E_{VAC} = 0.0 eV
	5	2	-19.434	0.809955	4.353	0.331981	1.979	

Tl	Z = 81		MullerHuckelParameters.Thallium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-9.827	1	2.191			U = [9, 9] eV
	6	1	-5.235	1	1.656			E_{VAC} = 0.0 eV

Pb	Z = 82		MullerHuckelParameters.Lead_Basis					N_V = 4.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-12.486	1	2.386			U = [9, 9] eV
	6	1	-6.527	1	1.88			E_{VAC} = 0.0 eV

Bi	Z = 83		MullerHuckelParameters.Bismuth_Basis					N_V = 5.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-15.189	1	2.56			U = [9, 9] eV
	6	1	-7.788	1	2.072			E_{VAC} = 0.0 eV

Po	Z = 84		MullerHuckelParameters.Polonium_Basis					N_V = 6.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-17.965	1	2.72			U = [9, 9] eV
	6	1	-9.053	1	2.245			E_{VAC} = 0.0 eV

At	Z = 85		MullerHuckelParameters.Astatine_Basis					N_V = 7.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-20.827	1	2.87			U = [9, 9] eV
	6	1	-10.337	1	2.45			E_{VAC} = 0.0 eV

Rn	Z = 86		MullerHuckelParameters.Radon_Basis					N_V = 8.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	6	0	-23.783	1	3.014			U = [9, 9] eV
	6	1	-11.647	1	2.556			E_{VAC} = 0.0 eV

Fr	Z = 87		MullerHuckelParameters.Francium_Basis					N_V = 1.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	7	0	-3.209	1	1.131			U = [9, 9, 9] eV
	7	1	-2.205	1	0.939			E_{VAC} = 0.0 eV
	6	2	-1.931	1	1.258			

Ra	Z = 88		MullerHuckelParameters.Radium_Basis					N_V = 2.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	7	0	-4.048	1	1.346			U = [9, 9, 9] eV
	7	1	-2.934	1	1.177			E_{VAC} = 0.0 eV
	6	2	-4.108	1	2.113			

Ac	Z = 89		MullerHuckelParameters.Actinium_Basis					N_V = 3.0
	n	l	E_{ion}	W_1	η_1	W_2	η_2	β = 1.75
	7	0	-4.381	1	1.433			U = [9, 9, 9] eV
	7	1	-3.098	1	1.253			E_{VAC} = 0.0 eV
	6	2	-6.843	1	2.12			

Th Z = 90 MullerHuckelParameters.Thorium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.31	1	1.742		
7	1	-3.345	1	1.256		
6	2	-4.98	0.624003	2.811	0.503002	1.554
5	3	-6.03	0.58187	5.475	0.547877	2.863

N_V = 4.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Pa Z = 91 MullerHuckelParameters.Protactinium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.41	1	1.699		
7	1	-3.375	1	1.286		
6	2	-5.07	0.617317	2.894	0.511263	1.595
5	3	-7.63	0.594817	5.659	0.530836	2.991

N_V = 5.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

U Z = 92 MullerHuckelParameters.Uranium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.51	1	1.728		
7	1	-3.406	1	1.315		
6	2	-5.1	0.622063	2.944	0.509052	1.611
5	3	-9.045	0.592502	5.885	0.530449	3.135

N_V = 6.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Np Z = 93 MullerHuckelParameters.Neptunium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.61	1	1.756		
7	1	-3.436	1	1.345		
6	2	-5.095	0.615803	3.017	0.516835	1.646
5	3	-10.35	0.519698	3.254	0.600651	6.068

N_V = 7.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Pu Z = 94 MullerHuckelParameters.Plutonium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.71	1	1.681		
7	1	-3.467	1	1.375		
6	2	-5.06	0.609204	3.085	0.525176	1.677
5	3	-11.585	0.509646	3.364	0.608577	6.241

N_V = 8.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

Am Z = 95 MullerHuckelParameters.Americium_Basis

n	l	E_{ion}	W_1	η_1	W_2	η_2
7	0	-5.8	1	1.704		
7	1	-3.497	1	1.405		
6	2	-5.01	0.596362	3.169	0.539327	1.719
5	3	-12.75	0.496199	3.457	0.620249	6.392

N_V = 9.0
β = 1.75
U = [9, 9, 9, 9] eV
E_{VAC} = 0.0 eV

4.7 Sentaurus Materials Workbench Reference Manual

Table of Contents

•*Sentaurus Materials Workbench Reference Manual*
–*Introduction*
–*Material specifications*
–*Single defect specification and convergence studies*
–*Multilayer Builder*
–*GrainBoundaryScattering*
–*Full SMW package*

4.7.1 Introduction

Sentaurus Materials Workbench helps you to create material references and to set up input files for atomistic calculations using DFT calculations, or empirical potentials, or both. It can automate defect generation and includes several techniques and recipes to increase the accuracy of calculations. Furthermore, from the atomistic calculation results, Sentaurus Materials Workbench analyzes the results and generates the Sentaurus model parameters that can be used for different TCAD tools, but in particular for band structure generation needed for Sentaurus Device.

Note: The SMW scripts need to import the package explicitly. In other words, be sure to include a `from SMW import *` at the beginning of the scripts.

There are different modules or workflows under Sentaurus Materials Workbench. The following content describes its main capabilities.

4.7.2 Material specifications

Using *MaterialSpecifications* objects you can define computational settings for your calculations. Sentaurus Materials Workbench comes with a *MaterialSpecificationsDatabase* with predefined *MaterialSpecifications* for industry relevant materials. This database should be the starting point for most applications.

•*MaterialSpecifications*

•*MaterialSpecificationsDatabase*

4.7.3 Single defect specification and convergence studies

Formation energies and trap energy levels for a variety of defects and supercell sizes can be done with *ChargedPointDefect*. An overview of how to specify particular defects and perform convergence studies can be read at the `ChargedPointDefect` *Notes*.

The available defect classes are:

•*DefectCluster*

•*Interstitial*

•*SplitInterstitial*

•*Substitutional*

•*Vacancy*

All to be run with:

•*ChargedPointDefect*

4.7.4 Multilayer Builder

For an overall explanation on the multilayer builder features see *Notes*.

The available classes are:

•*AmorphousLayer*

•*CrystalLayer*

•*MultilayerBuilder*

•*VacuumLayer*

4.7.5 GrainBoundaryScattering

The *GrainBoundaryScattering* calculates grain boundary resistances and specific resistivities of metal grain boundaries. For an overall explanation on the grain boundary scattering features see *Notes*.

The available classes are:

•*GrainBoundaryGenerator*

•*GrainBoundaryScattering*

4.7.6 Full SMW package

SMW

Classes

AlloyLayer

class AlloyLayer(*material_specifications*, *length=None*, *cleave_plane=None*, *plane_indices=None*, *tolerance=None*, *displacement_vector=None*, *percentage=None*, *substitutional_element=None*, *alloy_method=None*, *number_of_generations=None*, *population_size=None*, *random_seed=None*, ***kwargs*)

Class to specify a crystalline alloy layer materials in the MultilayerBuilder.

Parameters

•**material_specifications** (*MaterialSpecifications*) – A descriptor of a bulk material, plus the information needed to perform the calculations.

•**length** (*PhysicalQuantity* of type length | None) – Deprecated. use plane indices instead. **Default:** None

•**cleave_plane** (*sequence of length 3*) – The Miller indices of the layer.

- **plane_indices** (*tuple*) – The start and end atomic plane indices from the cleaved configuration, e.g., [0, 1], both inclusive. 0 always corresponds to the bottom layer of the configuration.
- **tolerance** (*float*) – The (logarithmic) detection tolerance used to define atomic planes. Range [-10, 4].
- **displacement_vector** (*sequence of length 2 | None*) – This surface vector is added to the fractional coordinates of the layer configuration.
- **percentage** (*float*) – The percentages of the atoms that should be changed.
- **substitutional_element** (*PeriodicTableElement*) – The element to change to.
- **alloy_method** (*EvolutionarySQS | SubstitutionalAlloyNormalDistribution | SubstitutionalAlloyFixedFraction | None*) – The alloy method to generate the alloy configuration. Options are EvolutionarySQS or SubstitutionalAlloyFixedFraction or SubstitutionalAlloyNormalDistribution. In EvolutionarySQS method, a special quasi-random structure (SQS) is generated using genetic optimization. NormalDistribution means you assign a number [0,1] from a normal distribution across all atoms and only pick those that have a value below the given percentage. FixedFraction means if you set 25%, you get 25% of the total number of atoms changed. The size of the random sample are the percentage of atoms provided, rounded to nearest integer. SubstitutionalAlloy method. **Default:** SubstitutionalAlloyFixedFraction.
- **number_of_generations** (*int*) – This parameter is valid if alloy_method is EvolutionarySQS. The number of times the population should be evolved.
- **population_size** (*int*) – This parameter is valid if alloy_method is EvolutionarySQS. The number of individuals which are part of a population. The population size must be at least 5, but it is recommended to use a large population size (>= 20) to obtain higher quality results.
- **random_seed** (*int | array of ints | None*) – Random seed used to initialize the pseudo-random number generator. Can be any integer between 0 and 2**32 - 1 inclusive, an array (or other sequence) of such integers, or None. If random_seed is None, then it will be generated using values from the system entropy pool.
- **kwargs** (*dict*) – Deprecated keyword arguments.

alloyMethod()

Returns
The alloy method to generate the alloy configuration. Options are EvolutionarySQS or SubstitutionalAlloyFixedFraction or SubstitutionalAlloyNormalDistribution. In EvolutionarySQS method, a special quasi-random structure (SQS) is generated using genetic optimization. NormalDistribution means you assign a number [0,1] from a normal distribution across all atoms and only pick those that have a value below the given percentage. FixedFraction means if you set 25%, you get 25% of the total number of atoms changed. The size of the random sample are the percentage of atoms provided, rounded to nearest integer. SubstitutionalAlloy method.

Return type
EvolutionarySQS | SubstitutionalAlloyNormalDistribution | SubstitutionalAlloyFixedFraction | None

displacementVector()

Returns
This surface vector is added to the fractional coordinates of the layer configuration.

Return type
sequence of length 2 | None

entry()

Returns
The entry.

Return type
HKMGEntry

classmethod entryType()

The type of entry to instantiate.

Returns
The entry type.

Return type
InterfacesEntry-based

length()

Returns
The target length (thickness) of the layer.

Return type
PhysicalQuantity of type length

materialSpecifications()

Returns
The descriptor of a bulk material, plus the information needed to perform the calculations.

Return type
MaterialSpecifications

newElement()

Returns
The element to change to.

Return type
PeriodicTableElement | None (vacancy)

numberOfGenerations()

Returns
This parameter is valid if alloy_method is EvolutionarySQS. The number of times the population should be evolved.

Return type
int

percentage()

Returns
The percentages of the atoms that should be changed.

Return type
float | sequence of floats

populationSize()

Returns
This parameter is valid if alloy_method is EvolutionarySQS. The number of individuals which are part of a population. The population size must be at least 5, but it is recommended to use a large population size (>= 20) to obtain higher quality results.

Return type
int

randomSeed()

Returns
Random seed used to initialize the pseudo-random number generator. Can be any integer between 0 and 2**32 - 1 inclusive, an array (or other sequence) of such integers, or None. If random_seed is None, then it will be generated using values from the system entropy pool.

Return type
int | array of ints | None

substitutionalElement()

Returns
The element to change to.

Return type
PeriodicTableElement | None (vacancy)

uniqueString()

Return a unique string representing the state of the object.

Usage Example

Example of a customized *AlloyLayer* object to generate an Iron cobalt alloy layer with 30% of Cobalt concentration.

```
custom_layer = AlloyLayer(
    MaterialSpecificationsDatabase.MRAM_MATERIALS['Iron cobalt (alloy)'],
    percentage=30,
    substitutional_element=Cobalt,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    alloy_method=EvolutionarySQS,
    )
```

alloy_layer_example.py

Notes

The *AlloyLayer* class defines how an alloy layer should be generated by the *MultilayerBuilder*. The alloy layer can be generated using the *EvolutionarySQS* or *substitutionalAlloy* method. The *EvolutionarySQS* is based on the special quasi-random structure (SQS) method[1]. Currently an alloy layer can only have binary composition, i.e. at maximum be composed of two elements.

Note: The grometry optimization is performed using the `material_specifications.relaxationCalculator()` calculator. This is normally a machine-learned moment tensor potential (MTP) force-field calculator.

---

[1] A.ăvanăde Walle, P.ăTiwary, M.ădeăJong, D.L. Olmsted, M.ăAsta, A.ăDick, D.ăShin, Y.ăWang, L.-Q. Chen, and Z.-K. Liu. Efficient stochastic generation of special quasirandom structures. *Calphad*, 42:13–18, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0364591613000540.

AmorphousLayer

class AmorphousLayer(*material_specifications*, *length*, *method=None*, *random_seed=None*, *melt_temperature=None*, *melt_time=None*, *quench_temperature=None*, *quench_rate=None*, *time_step=None*, *max_retries=None*)

Class to represent an amorphous layer material in MultilayerBuilder.

The amorphous layer is generated using a melt-quench MD procedure. First the system is melted and equilibrated at a high temperature. Then it is quenched from the melt_temperature to the quench_temperature at a rate of quench_rate. Then the system is quenched to zero Kelvin using a geometry optimization.

Parameters

- **material_specifications** (*MaterialSpecifications*) – A descriptor of a bulk material, plus the information needed to perform the calculations.
- **length** (*PhysicalQuantity* of type length) – The target length (thickness) of the layer. A length of zero will generate the smallest possible layer. **Default:** 10 * Angstrom.
- **method** (*MolecularDynamicsMeltQuench* | *TimeStampedForceBiasMonteCarloMeltQuench*) – Technique to perform the amorphization and associated parameters. **Default:** *MolecularDynamicsMeltQuench*.
- **random_seed** (*int | array of ints | None*) – The random seed for the initial structure generation via packmol. Can be any integer between 0 and 2**32 - 1 inclusive, an array (or other sequence) of such integers, or None.
- **melt_temperature** (*PhysicalQuantity* of type temperature) – Suggested melt temperature. **Default:** 3500 * Kelvin.
- **melt_time** (*PhysicalQuantity* of type time) – The length of the melt MD simulation. **Default:** 40 * ps.
- **quench_temperature** (*PhysicalQuantity* of type temperature) – The target temperature during the quenching MD simulation. **Default:** 300 * Kelvin.
- **quench_rate** (*PhysicalQuantity* of type temperature per time) – The rate of temperature change during the quenching. This rate will be used to calculate the length of the quenching MD simulation. **Default:** 64 * Kelvin / ps.
- **time_step** (*PhysicalQuantity* of type time) – Time step per iteration. **Default:** 1.0 * fs.
- **max_retries** (*int*) – Maximum number of tries to satisfy amorphous quality. **Default:** 30.

amorphousDensity()

Returns
The mass density of the amorphous material.

Return type
PhysicalQuantity of type mass / volume

length()

Returns
The target length (thickness) of the layer.

Return type
PhysicalQuantity of type length

materialSpecifications()

Returns
The descriptor of a bulk material, plus the information needed to perform the calculations.

Return type
MaterialSpecifications

method()

Returns
The method to perform amorphization

Return type
TimeStampedForceBiasMonteCarloMeltQuench | *MolecularDynamicsMeltQuench*

stoichiometry()

Returns
The dict to map element to stoichiometric number.

Return type
dict

uniqueString()
Return a unique string representing the state of the object.

Usage Example

Example of a customized *AmorphousLayer* object to generate an amorphous silica layer.

```
custom_layer = AmorphousLayer(
    MaterialSpecificationsDatabase.MULTILAYER_MATERIALS['Silica'],
    length=10.0*Angstrom,
    melt_temperature=6000*Kelvin,
    melt_time=60*picosecond,
    quench_temperature=1000*Kelvin,
    quench_rate=60*Kelvin/picosecond,
    time_step=0.5*femtosecond,
)
```

amorphous_layer_example.py

Notes

The *AmorphousLayer* class defines how an amorphous layer should be generated by the *MultilayerBuilder*. The amorphous structure is generated using *runPackmol* based on `material_specifications.amorphousDensity()` and `material_specifications.elementsWithStoichiometry()`. A melt quench procedure is carried out starting from the optimized packmol configuration. The parameters in the melt-quench protocol can be specified via the `method` parameter, which can take a *MolecularDynamicsMeltQuench*, *TimeStampedForceBiasMonteCarloMeltQuench*, or *EmpiricalAmorphizationMethod* object.

Note: An initial optimization is done on the configuration obtained from *runPackmol* using `material_specifications.quickOptimizerCalculator()`. This is usually a robust force-field calculator. Another geometry optimization is performed using `material_specifications.amorphizationCalculator()`. The MD simulations and final optimization of the amorphous configuration are performed using the `material_specifications.amorphizationCalculator()` calculator. This is normally a machine-learned moment tensor potential (MTP) force-field calculator.

The first step is to melt the structure. This is done by running a NVT MD simulation at a temperature of `melt_temperature` for time of `melt_time`. This temperature should be hot enough to ensure that the material can thoroughly melt.

The second step is quenching. A NVT MD simulation is run starting at a temperature of `melt_time` and cooling at a rate of `quench_rate` until the temperature is equal to the `quench_temperature`.

Note: A slower quenching rate may give more realistic amorphous structures, but it is not often possible to simulate experimental cooling rates due to the number of MD steps that would be required.

The final step is geometry optimization, which represents instantaneous quenching from the `quench_temperature` to zero Kelvin.

The melt-quench protocol is repeated for up to `max_retries` iterations, specified in the amorphous method, until a final configuration without coordination defects is obtained.

CrystalLayer

class CrystalLayer(*material_specifications*, *length=None*, *cleave_plane=None*, *plane_indices=None*, *tolerance=None*, *displacement_vector=None*, ***kwargs*)

Class to specify layer materials in MultilayerBuilder.

Parameters

- **material_specifications** (*MaterialSpecifications*) – A descriptor of a bulk material, plus the information needed to perform the calculations.
- **length** (*PhysicalQuantity* of type length | None) – Deprecated. use plane indices instead. **Default:** None
- **cleave_plane** (*sequence of length 3*) – The Miller indices of the layer.
- **plane_indices** (*tuple*) – The start and end atomic plane indices from the cleaved configuration, e.g., [0, 1], both inclusive. 0 always corresponds to the bottom layer of the configuration.
- **tolerance** (*float*) – The (logarithmic) detection tolerance used to define atomic planes. Range [-10, 4].
- **displacement_vector** (sequence | *PhysicalQuantity* of type length | None) – This surface vector is added to the layer configuration.
- **kwargs** (*dict*) – Any deprecated keyword arguments.

displacementVector()

Returns
This surface vector is added to the fractional coordinates of the layer configuration.

Return type
sequence of length 2 | None

entry()

Returns
The entry.

Return type
HKMGEntry

classmethod entryType()

The type of entry to instantiate.

Returns
The entry type.

Return type
InterfacesEntry-based

`length()`

Returns
The target length (thickness) of the layer.

Return type
PhysicalQuantity of type length

`materialSpecifications()`

Returns
The descriptor of a bulk material, plus the information needed to perform the calculations.

Return type
MaterialSpecifications

`uniqueString()`

Return a unique string representing the state of the object.

EmpiricalAmorphizationMethod

`class EmpiricalAmorphizationMethod(`*amorphous_quality=None*, *max_retries=None*, *melt_quality=None*, *max_melt_retries=None*, *melt_temperature=None*, *melt_time=None*, *quench_temperature=None*, *quench_rate=None*, *defect_optimization_retries=None*, *time_step=None*, *cut_off=None*, *random_seed=None*`)`

Base class for melt/quench amorphization methods

Parameters

- **amorphous_quality** (*float*) – Expected quality for the amorphous material, 0 to 1. 1 being free of defects. **Default:** 0.75
- **max_retries** (*int*) – Maximum number of tries to satisfy amorphous quality **Default:** 2
- **melt_quality** (*float*) – Requested percentage of melted material **Default:** 0.35
- **max_melt_retries** (*float*) – Maximum number of tries to melt the material with the requested quality **Default:** same as `max_retries`
- **melt_temperature** (*PhysicalQuantity* of type temperature) – Suggested melt temperature **Default:** 5000*Kelvin
- **melt_time** (*PhysicalQuantity* of type time) – The length of the melt MD simulation. **Default:** 40*picosecond
- **quench_temperature** (*PhysicalQuantity* of type temperature) – The target temperature during the quenching MD simulation. **Default:** 300*Kelvin
- **quench_rate** (*PhysicalQuantity* of type temperature per time) – The rate of temperature change during the quenching. This rate will be used to calculate the length of the quenching MD simulation. **Default:** 400*Kelvin/picosecond
- **defect_optimization_retries** (*int*) – Number of optimization tries requested after melt to decrease the number of defects in the amorphous sample. **Default:** 20
- **time_step** (*PhysicalQuantity* of time time) – Time step per iteration **Default:** 1*femtosecond
- **cut_off** (*PhysicalQuantity* of type length.) – Cut off radius to consider when building the empirical potentials. **Default:** Second nearest neighbor in associated bulk configuration.
- **random_seed** (*int*) – Random seed **Default:** 1234

amorphousQuality()

Returns
requested amorphous quality

Return type
float

cutOff()

Returns
Che cut off value defining the local environment

Return type
PhysicalQuantity of type length

defectOptimizationRetries()

Returns
Number of tries requested to decrease defects.

Return type
int

maxMeltRetries()

Returns
requested maximum number of melt tries

Return type
int

maxRetries()

Returns
requested number of maximum tries

Return type
int

meltQuality()

Returns
requested melt quality

Return type
float

meltSteps()

Returns
The calculated number of melt steps.

Return type
int

meltTemperature()

Returns
requested melt temperature

Return type
PhysicalQuantity of type temperature

meltTime()

Returns
requested melt time

Return type
PhysicalQuantity of type time

quenchRate()

Returns
requested quench rate

Return type
PhysicalQuantity of type temperature per time

quenchTemperature()

Returns
requested quench temperature

Return type
PhysicalQuantity of type temperature

randomSeed()

Returns
random seed

Return type
int

timeStep()

Returns
returns time step

Return type
int

uniqueString()
Return a unique string representing the state of the object.

FET2DAtomisticParameterExtraction

class FET2DAtomisticParameterExtraction(*filename*, *object_id*, *channel_material_name*, *contact_material_name*, *channel_thickness*, *maxE*, *contact_orientation*, *channel_configuration*, *contact_configuration*, *run_pe=True*, *run_sb=True*, *number_of_processes_per_task=None*, *log_filename_prefix=''*)

Check configuration input.

Parameters

- **filename** (*str*) – The full or relative filename path the study object should be saved to.
- **object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file.
- **channel_material_name** (*str*) – The name of the channel material.

- **contact_material_name** (*str*) – The name of the contact material.
- **channel_thickness** (*int*) – The thickness of the channel in monolayers.
- **maxE** (*PhysicalQuantity* of type energy) – The upper energy limit for valley detection. Determines the number of LocalBandstructures (NBands) that will be used for band parameter extraction. The algorithm will try to find at least one set of band parameters before this limit is used to break off the search.
- **contact_orientation** (*sequence of int*) – The Miller indices for the contact surface towards the channel.
- **channel_configuration** (*BulkConfiguration*) – The bulk configuration for the channel material.
- **contact_configuration** (*BulkConfiguration*) – The bulk configuration for the contact/metal material.
- **run_pe** (*bool*) – If True, the parameter extraction task will be performed. **Default:** True
- **run_sb** (*bool*) – If True, the Schottky barrier extraction task will be performed. **Default:** True
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively.
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of the study. If `LogToStdOut`, all logging will instead be sent to standard output.

bandGap()

Returns
Calculated bandgap.

Return type
PhysicalQuantity of type energy

channelMaterialName()

Returns
The name of the channel material.

Return type
str

channelThickness()

Returns
The thickness of the channel in monolayers.

Return type
int

contactMaterialName()

Returns
The name of the contact material.

Return type
str

contactOrientation()

Returns
The Miller indices for the contact surface towards the channel.

Return type
sequence of int

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

electronAffinity()

Returns
Calculated electron affinity.

Return type
PhysicalQuantity of type energy

extractionResults(*heavyweight=False*)

Returns all results for finished tasks.

Parameters
heavyweight (*bool*) – If True, reads heavyweight data from workflow tasks.

Returns
Results of finished tasks.

Return type
dict

filename()

Returns
The filename where the study object is stored.

Return type
str

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

maxE()

Returns
The upper energy limit for valley detection.

Return type
PhysicalQuantity of type energy

metatext()

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

nlinfo()

Get only the printable results from the extraction results - plain text.

Returns
The study information.

Return type
dict

`nlinfoHtml()`

Get only the printable results from the extraction results - rich text.

Returns
The study information.

Return type
dict

`nlprint(stream=None)`

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** `NLPrintLogger()`

`numberOfBands()`

Returns
Calculated number of bands.

Return type
int

`numberOfProcessesPerTask()`

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

`numberOfProcessesPerTaskResolved()`

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

`objectId()`

Returns
The name of the study object in the file.

Return type
str

`permittivity()`

Returns
Calculated dielectric constant / permittivity.

Return type
float

qatkBands()

Returns
Calculated QATK bands - input for SWB.

Return type
str

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

schottkyBarrier()

Returns
Calculated Schottky barrier.

Return type
PhysicalQuantity of type energy

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

tasknameForResult(*name*)

Check for a particular result from this object, including anything in the workflow tasks.

Parameters
name (*str*) – The name of the result to get.

Returns
The name of the task that contains the result for the given name.

Return type
str

thickness()

Returns
Calculated layer thickness in micrometers.

Return type
PhysicalQuantity of type length

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Usage Examples

Example 1. Perform an extraction of parameters and material properties in QuantumATK needed to run the Sentaurus Workbench model calibration and device characterization. The materials used are MoS2 for the 2D channel and gold for the contacts. The channel width is 1 ML.

```
from QuantumATK import *
from AddOns.FET2D.Study.FET2DAtomisticParameterExtraction import␣
↪FET2DAtomisticParameterExtraction

# Channel configuration
# Set up lattice
lattice = Hexagonal(3.1881795204035988*Angstrom, 11.057859406824896*Angstrom)

# Define elements
elements = [Sulfur, Molybdenum, Sulfur, Sulfur, Molybdenum, Sulfur]

# Define coordinates
fractional_coordinates = [[ 0.333333333333,  0.666666666667,  0.609057255744],
                          [ 0.666666666667,  0.333333333333,  0.75          ],
                          [ 0.333333333333,  0.666666666667,  0.890942744256],
                          [ 0.666666666667,  0.333333333333,  0.109057255744],
                          [ 0.333333333333,  0.666666666667,  0.25          ],
                          [ 0.666666666667,  0.333333333333,  0.390942744256]]

# Set up configuration
channel_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Contact configuration
# Set up lattice
lattice = FaceCenteredCubic(4.07825*Angstrom)

# Define elements
elements = [Gold]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
contact_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# Define the Parameter Extraction Study Object.
parameter_extraction = FET2DAtomisticParameterExtraction(
    filename='fet2d_parameter_extraction_1642514327.22.hdf5',
    object_id='FET2DAtomisticParameterExtraction_0',
```

(continues on next page)

(continued from previous page)

```
    channel_material_name='MoS2',
    contact_material_name='Gold',
    channel_thickness=1,
    maxE=1.1 * eV,
    contact_orientation=(0, 0, 1),
    channel_configuration=channel_configuration,
    contact_configuration=contact_configuration,
    log_filename_prefix_default='fet2d_parameter_extraction_',
)

# Start the calculations.
parameter_extraction.update()

# Extract the results.
parameter_extraction_results = parameter_extraction.extractionResults()
```

Example1.py

Starting configurations

In principle, any *BulkConfiguration* can be used as the `channel_configuration` and `contact_configuration`, but care should be taken that the channel configuration can be meaningfully repeated in the C-direction and periodic in the A-B plane. Ideally, both the channel and contact configurations should also be kept to a minimal size for maximum computational efficiency.

Extended results

In addition to numerical material properties and model calibration parameters, *FET2DAtomisticParameterExtraction* also contains extended results in the form of NLObjects. These can be queried from the *FET2DAtomisticParameterExtraction* object using the *extractionResults()* method, which returns a dictionary:

```
mos2_band_structure = parameter_extraction.extractionResults()['bandstructure']
```

These can also be unpacked in Nanolab via the right-click context menu when a *FET2DAtomisticParameterExtraction* is selected, and can be sent for further analysis in, e.g., the Builder, Viewer or a dedicated analyzer plugin. Available NLObjects:

- **PE_bulk_configuration** [*BulkConfiguration*] - The configuration of the 2D channel with an LCAO calculator, used for extracting most of the channel material parameters.
- **bandstructure** [*Bandstructure*] - The band structure of the PE_bulk_configuration.
- **eigenvalues** [*Eigenvalues*] - The eigenvalues of the PE_bulk_configuration.
- **electron_density** [*ElectronDensity*] - The electron density of PE_bulk_configuration, used to estimate the real channel thickness.
- **hartree_difference_potential** [*HartreeDifferencePotential*] - The potential used to calculate the dielectric constant of the 2D channel.
- **PW_bulk_configuration** [*BulkConfiguration*] - The configuration used to calculate the electron affinity of the 2D channel, with a plane-wave calculator.

•**PW_DOS** [*DensityOfStates*] - The density of states used to calculate the electron affinity.

•**Ch_Ct_initial** [*BulkConfiguration*] - The initial channel-contact interface configuration used calculate the Schottky barrier.

•**Ch_Ct_interface** [*BulkConfiguration*] - The geometry-optimized version of Ch_Ct_initial.

•**fat_bandstructure** [*FatBandstructure*] - The band structure used to calculate the Schottky barrier.

See also fet2d_general.

FET2DDeviceSetupParameters

class FET2DDeviceSetupParameters(*gate_type*, *lgate*, *gate_material_name*, *ttox*, *oxide_material_name*, *til*, *lcontact*, *lspacer*, *lbackgate=PhysicalQuantity(0.0, mum)*, *tbox=PhysicalQuantity(0.0, mum)*, *ch_dop=PhysicalQuantity(0.0, 1 / cm**3)*, *sd_dop=PhysicalQuantity(0.0, 1 / cm**3)*, *nit=PhysicalQuantity(0.0, 1 / cm**3)*, *nit_bg=PhysicalQuantity(0.0, 1 / cm**3)*, *mumax=PhysicalQuantity(50.0, cm**2 / V / s)*)

Serializable container for parameters to set up a 2DFET device.

Parameters

•**gate_type** (*enum*) – The gate type from GATE_TYPES.

•**lgate** (*PhysicalQuantity* of type length) – The length of the top gate.

•**gate_material_name** (*str*) – The name of the gate material.

•**ttox** (*PhysicalQuantity* of type length) – The thickness of the top oxide.

•**oxide_material_name** (*str*) – The name of the oxide material.

•**til** (*PhysicalQuantity* of type length) – The thickness of the insulating layer between the top oxide and the channel.

•**lcontact** (*PhysicalQuantity* of type length) – The length of the contact.

•**lspacer** (*PhysicalQuantity* of type length) – The length of the spacer.

•**lbackgate** (*PhysicalQuantity* of type length) – The length of the back gate.

•**tbox** (*PhysicalQuantity* of type length) – The thickness of the bottom oxide.

•**ch_dop** (*PhysicalQuantity* of type inverse volume) – The doping concentration of the channel.

•**sd_dop** (*PhysicalQuantity* of type inverse volume) – The doping concentration of the source and drain.

•**nit** (*PhysicalQuantity* of type inverse volume) – The interface trap concentration of the top oxide.

•**nit_bg** (*PhysicalQuantity* of type inverse volume) – The interface trap concentration of the bottom oxide.

•**mumax** (*int*) – The scattering limited mobility, a parameter used in SDevice.

chDop()

Returns
The doping concentration of the channel.

Return type
PhysicalQuantity of type inverse volume

gateMaterialName()

Returns
The name of the gate material.

Return type
str

gateType()

Returns
The gate type.

Return type
enum

lbackgate()

Returns
The length of the back gate.

Return type
PhysicalQuantity of type length

lcontact()

Returns
The length of the contact.

Return type
PhysicalQuantity of type length

lgate()

Returns
The length of the top gate.

Return type
PhysicalQuantity of type length

lspacer()

Returns
The length of the spacer.

Return type
PhysicalQuantity of type length

mumax()

Returns
The scattering limited mobility, a parameter used in SDevice.

Return type
int

nit()

Returns
The interface trap concentration of the top oxide.

Return type
PhysicalQuantity of type inverse volume

`nitBg()`

Returns
The interface trap concentration of the bottom oxide.

Return type
PhysicalQuantity of type inverse volume

`nlinfo()`

Create a nicely formatted dict for presentation in the gui.

Returns
The study information.

Return type
dict

`oxideMaterialName()`

Returns
The name of the oxide material.

Return type
str

`sdDop()`

Returns
The doping concentration of the source and drain.

Return type
PhysicalQuantity of type inverse volume

`tbox()`

Returns
The thickness of the bottom oxide.

Return type
PhysicalQuantity of type length

`til()`

Returns
The thickness of the insulating layer between the top oxide and the channel.

Return type
PhysicalQuantity of type length

`ttox()`

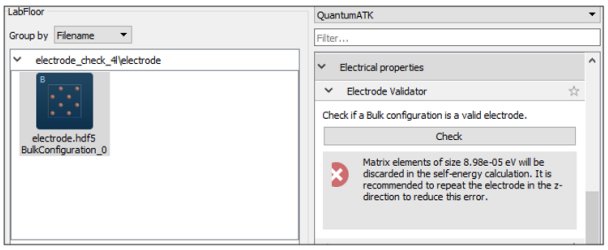
Returns
The thickness of the top oxide.

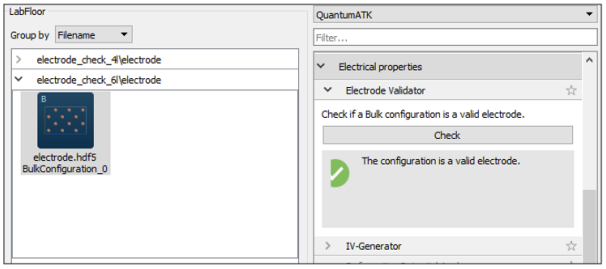
Return type
PhysicalQuantity of type length

`uniqueString()`

Return a unique string representing the state of the object.

Usage Examples





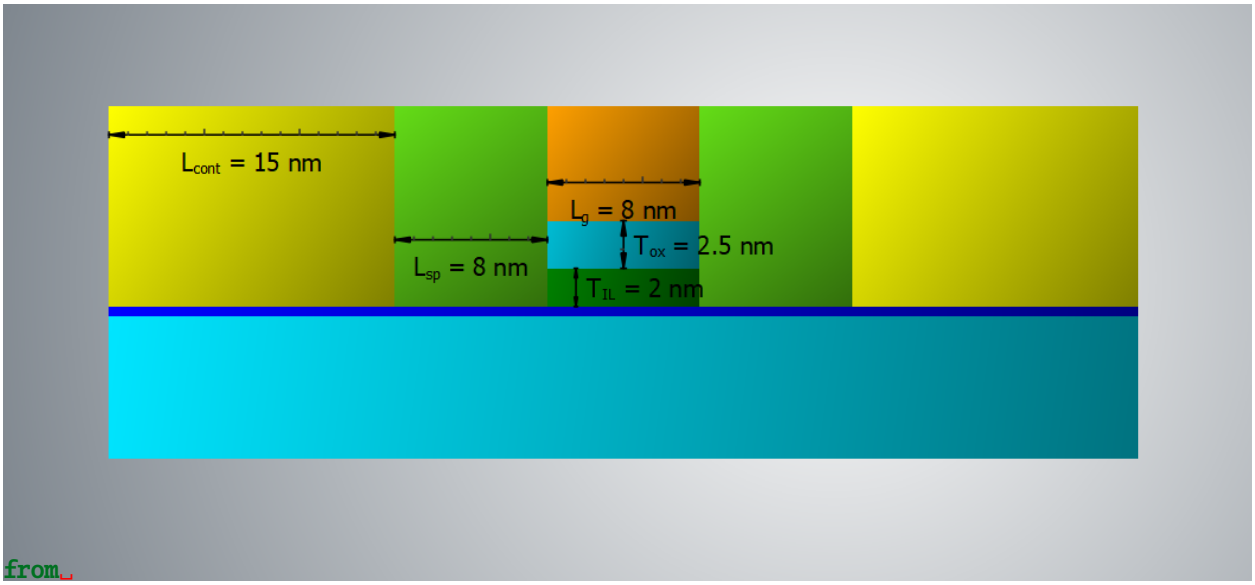


Fig. 4.82 Visualization of the device setup used in Example 1. Yellow areas are contacts (metals), bright green areas are spacers (none/vacuum), the orange area is the gate (metal), blue areas are oxides, the dark-green area is the representative insulating layer (none/vacuum) between the oxide and the channel and the dark blue area is the 2D channel.

Example 1. Set up a custom device design based on Figure 1.

```
from␣
↪QuantumATK␣
↪import␣
↪*
from␣
↪Addons.
↪FET2D.
↪Study.
↪FET2DDeviceSetupParameters␣
↪import␣
↪FET2DDeviceSetupParameters


↪#␣
↪Define␣
```

```
↪the␣
↪device␣
↪setup␣
↪parameters.
↪
device_
↪setup␣
↪=␣
↪FET2DDeviceSetupParameters(
␣
↪␣
↪␣
↪␣
↪gate_
↪type=GATE_
↪TYPES.
↪SINGLE_
↪GATE,
↪
␣
↪␣
↪␣
↪␣
↪lgate=0.
↪008␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪gate_
↪material_
↪name=
↪'Gold
↪',
↪
␣
↪␣
↪␣
↪␣
↪ttox=0.
↪0025␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪oxide_
```

```
↪material_
↪name=
↪'SiO2
↪',
↪
␣
↪␣
↪␣
↪␣
↪til=0.
↪002␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪lcontact=0.
↪015␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪lspacer=0.
↪008␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪lbackgate=0.
↪034␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪back_
↪gate_
↪material_
↪name=
↪'Gold
↪',
↪
```

```
␣
↪␣
↪␣
↪␣
↪tbox=0.
↪0025␣
↪*␣
↪um,
↪
␣
↪␣
↪␣
↪␣
↪ch_
↪dop=0.
↪0␣
↪*␣
↪cm**-
↪3,
↪
␣
↪␣
↪␣
↪␣
↪sd_
↪dop=0.
↪0␣
↪*␣
↪cm**-
↪3,
↪
␣
↪␣
↪␣
↪␣
↪nit=0.
↪0␣
↪*␣
↪cm**-
↪3,
↪
␣
↪␣
↪␣
↪␣
↪nit_
↪bg=0.
↪0␣
↪*␣
↪cm**-
↪3,
↪
```

(continued from previous page)

```
␣
→␣
→␣
→␣
→mumax=50.
→0␣
→*␣
→cm**2␣
→/
→␣
→V␣
→/
→␣
→s,
→
)
```

Example1.py

Gate types

Variable name	Description
GATE_TYPES.SI GLE_GATE	An FET with a single gate on top of the channel.
GATE_TYPES.DO BLE_GATE	An FET with one gate on top of the channel and a longer back gate below the channel. This choice also enables additional device dimensions, such as the back gate overlap length, to be set.

See also fet2d_general.

FET2DSentaurusDeviceCharacteristics

class FET2DSentaurusDeviceCharacteristics(*filename*, *object_id*, *device_setup*, *model_calibration_results*, *workfunction=None*, *number_of_processes_per_task=None*, *log_filename_prefix=None*)

Executes device characteristics task.

Parameters

- **filename** (*str*) – The full or relative filename path the study object should be saved to.
- **object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file.
- **device_setup** (*FET2DDeviceSetupParameters*) – The device setup.
- **model_calibration_results** (*dict*) – The output of a FET2DSentaurusModelCalibration.extractionResults() call.
- **workfunction** (*PhysicalQuantity of type energy*) – The effective work function of the gate/backgate in the device.

- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively.
- **log_filename_prefix** (str | LogToStdOut) – Filename prefix for the logging output of the study. If LogToStdOut, all logging will instead be sent to standard output.

channelMaterialName()

Returns
The name of the channel material.

Return type
str

channelThickness()

Returns
The thickness of the channel in monolayers.

Return type
int

contactMaterialName()

Returns
The name of the contact material.

Return type
str

contactOrientation()

Returns
The Miller indices for the contact surface towards the channel.

Return type
sequence of int

cv()
Getter for cv

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

deviceSetup()

Returns
The device setup.

Return type
FET2DDeviceSetupParameters

extractionResults()

Returns
All results excluding NL objects.

Return type
dict

`filename()`

Returns
The filename where the study object is stored.

Return type
str

`idvd_0()`
Getter for idvd_0

`idvd_1()`
Getter for idvd_1

`idvd_2()`
Getter for idvd_2

`idvd_3()`
Getter for idvd_3

`idvg()`
Getter for idvg

`idvgVariables()`

Returns
The Id-Vg variables.

Return type
dict

`logFilenamePrefix()`

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

`maxE()`

Returns
The upper energy limit for valley detection.

Return type
PhysicalQuantity of type energy

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nlinfo()`
Get only the printable results from the extraction results - plain text.

Returns
The study information.

Return type
dict

nlinfoHtml()

Get only the printable results from the extraction results.

Returns
The study information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

processVariables()

Returns
The process variables.

Return type
dict

resultsToTransfer()

Returns
A copy of the relevant results from model calibration calculation.

Return type
dict

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

workfunction()

Getter for workfunction

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Usage Examples

Example 1. Perform device characterization based on manually defined model calibration results.

```
from QuantumATK import *
from AddOns.FET2D.Study.FET2DDeviceSetupParameters import FET2DDeviceSetupParameters
from AddOns.FET2D.Study.FET2DSentaurusDeviceCharacteristics import␣
↪FET2DSentaurusDeviceCharacteristics

# Define previous model calibration results.
model_calibration_results = {
    'channel_material_name': 'MoS2',
    'contact_material_name': 'Gold',
    'channel_thickness': 1,
    'maxE': 1.1 * eV,
    'contact_orientation': (0, 0, 1),
    'NBands': 8,
    'chi': 4.25 * eV,
    'eps': 3.38,
    'H': 0.0006516 * um,
    'BG': 1.699 * eV,
    'PhiSB': 0.276 * eV,
    'WF': 4.5 * eV,
    'NPparLEV': '1.215:1.215',
    'UpperEnergyValley': 'valley0:valley1:valley2:valley3:valley6:valley7',
    'EMparUEV': '0.801:0.801:0.801:0.801:1.311:1.311',
    'g_cal': '0.210575',
```

(continues on next page)

(continued from previous page)

```
    'LowerEnergyValley': 'valley4:valley5',
    'Eshift': '0.139',
    'EMperpUEV': '1.081:1.081:1.081:1.081:0.714:0.714',
    'V0f_cal': '1',
    'NUpBands': '6',
    'EMperpLEV': '0.484:0.484',
    'EMparLEV': '0.484:0.484',
    'NLowBands': '2',
    'Bf_cal': '0.987581',
    'NPparUEV': '1.773:1.773:0.433:0.433:1.155:1.155',
    'Tf_cal': '1.75296',
    'k_cal': '17.7495',
}

# Define the device setup parameters.
device_setup = FET2DDeviceSetupParameters(
    gate_type=GATE_TYPES.SINGLE_GATE,
    lgate=0.008 * um,
    gate_material_name='Gold',
    ttox=0.0025 * um,
    oxide_material_name='SiO2',
    til=0.002 * um,
    lcontact=0.015 * um,
    lspacer=0.008 * um,
    lbackgate=0.034 * um,
    back_gate_material_name='Gold',
    tbox=0.0025 * um,
    ch_dop=0.0 * cm**-3,
    sd_dop=0.0 * cm**-3,
    nit=0.0 * cm**-3,
    nit_bg=0.0 * cm**-3,
    mumax=50.0 * cm**2 / V / s,
)

# Define the Device Characteristics Study Object.
device_characteristics = FET2DSentaurusDeviceCharacteristics(
    filename='fet2d_device_characterization_1642515686.31.hdf5',
    object_id='FET2DSentaurusDeviceCharacteristics_0',
    device_setup=device_setup,
    model_calibration_results=model_calibration_results,
    log_filename_prefix_default='fet2d_device_characterization_',
)

# Start the calculations.
device_characteristics.update()

# Extract the results.
device_characteristics_results = device_characteristics.extractionResults()
```

Example1.py

If a *FET2DSentaurusModelCalibration* is available, the model calibration results can be defined as

```
model_calibration_results = model_calibration.extractionResults()
```

where `model_calibration` is an instance of *FET2DSentaurusModelCalibration* that has completed all of its calculations.

Example 2. Perform a chained model calibration and device characterization study based on manually defined parameter extraction results.

```
from QuantumATK import *
from AddOns.FET2D.Study.FET2DDeviceSetupParameters import FET2DDeviceSetupParameters
from AddOns.FET2D.Study.FET2DSentaurusDeviceCharacteristics import␣
↪FET2DSentaurusDeviceCharacteristics
from AddOns.FET2D.Study.FET2DSentaurusModelCalibration import␣
↪FET2DSentaurusModelCalibration

# Define previous parameter extraction results.
parameter_extraction_results = {
    'channel_material_name': 'MoS2',
    'contact_material_name': 'Gold',
    'channel_thickness': 1,
    'maxE': 1.1 * eV,
    'contact_orientation': (0, 0, 1),
    'NBands': 8,
    'chi': 4.25 * eV,
    'eps': 3.38,
    'H': 0.0006516 * um,
    'BG': 1.699 * eV,
    'qatk_bands': '0.367\n1.081\n1.923\n0.989\n0.346\n0.801\n1.773\n0.989\n0.254\n1.081\
↪n0.618\n0.989\n0.346\n0.801\n1.773\n0.989\n0.367\n1.081\n1.923\n0.989\n0.344\n0.801\n0.
↪433\n0.989\n0.254\n1.081\n0.618\n0.989\n0.344\n0.801\n0.433\n0.989\n0.452\n0.484\n0.
↪911\n0.85\n0.485\n0.484\n1.215\n0.85\n0.527\n0.484\n1.589\n0.85\n0.485\n0.484\n1.215\
↪n0.85\n0.312\n0.714\n0.542\n0.989\n0.204\n1.311\n1.155\n0.989\n0.713\n0.714\n1.153\n0.
↪989\n0.204\n1.311\n1.155\n0.989',
    'PhiSB': 0.276 * eV,
}

# Define the Model Calibration Study Object.
model_calibration = FET2DSentaurusModelCalibration(
    filename='fet2d_model_calibration_1642515594.46.hdf5',
    object_id='FET2DSentaurusModelCalibration_0',
    workfunction=7.835 * eV,
    parameter_extraction_results=parameter_extraction_results,
    log_filename_prefix_default='fet2d_model_calibration_',
)

# Start the calculations.
model_calibration.update()

# Extract the results.
model_calibration_results = model_calibration.extractionResults()

# Define the device setup parameters.
device_setup = FET2DDeviceSetupParameters(
```

(continues on next page)

(continued from previous page)

```
    gate_type=GATE_TYPES.SINGLE_GATE,
    lgate=0.008 * um,
    gate_material_name='Gold',
    ttox=0.0025 * um,
    oxide_material_name='SiO2',
    til=0.002 * um,
    lcontact=0.015 * um,
    lspacer=0.008 * um,
    lbackgate=0.034 * um,
    back_gate_material_name='Gold',
    tbox=0.0025 * um,
    ch_dop=0.0 * cm**-3,
    sd_dop=0.0 * cm**-3,
    nit=0.0 * cm**-3,
    nit_bg=0.0 * cm**-3,
    mumax=50.0 * cm**2 / V / s,
)

# Define the Device Characteristics Study Object.
device_characteristics = FET2DSentaurusDeviceCharacteristics(
    filename='fet2d_device_characterization_1642515596.36.hdf5',
    object_id='FET2DSentaurusDeviceCharacteristics_0',
    device_setup=device_setup,
    model_calibration_results=model_calibration_results,
    log_filename_prefix_default='fet2d_device_characterization_',
)

# Start the calculations.
device_characteristics.update()

# Extract the results.
device_characteristics_results = device_characteristics.extractionResults()
```

Example2.py

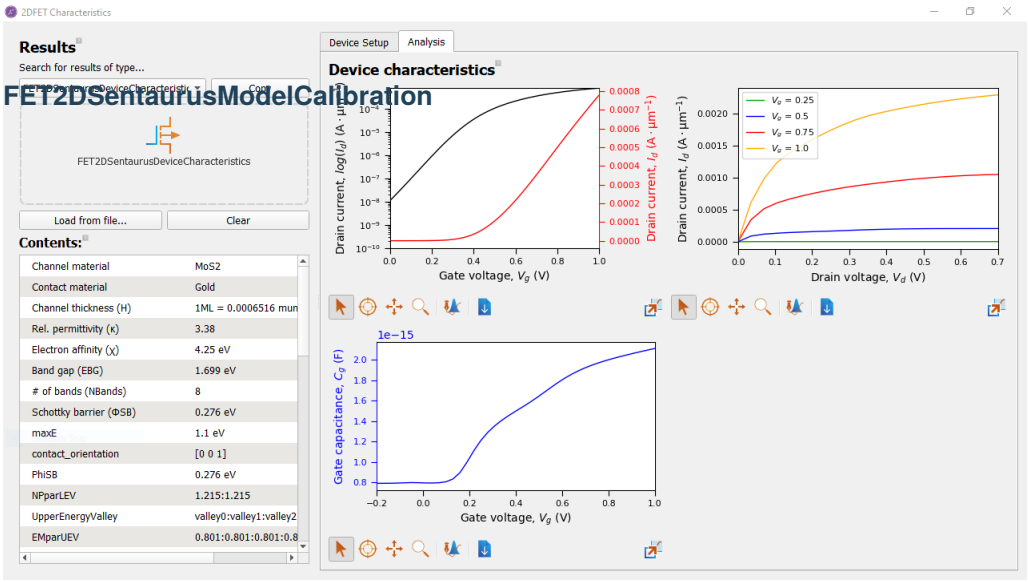
If a *FET2DAtomisticParameterExtraction* is available, the parameter extraction results can be defined as

```
parameter_extraction_results = parameter_extraction.extractionResults()
```

where `parameter_extraction` is an instance of *FET2DAtomisticParameterExtraction* that has completed all of its calculations.

Analysis

The results of a device characterization can be summarized in Nanolab using the 2DFET Characteristics plugin. The summary includes plots of the drain current to gate voltage ($I_D - V_G$) relation in both logarithmic and linear scale, the drain current to drain voltage ($I_D - V_D$) relation at four different gate voltages and the gate capacitance to gate voltage ($C_G - V_G$) relation. Using the plotting framework, these plots can be stored, edited or combined across different calculation runs. Additional calculation results from the Sentaurus Workbench calculation, such as .plt files and the actual Sentaurus Workbench project, will also be available at the end of a calculation, but are not handled by QuantumATK or Nanolab.



FET2DSentaurusModelCalibration

Fig. 4.84: Example of the *Analysis* tab in the 2DFET Characteristics plugin.

See also fet2d_general.

class FET2DSentaurusModelCal

Check configuration input.

Parameters

- **filename** (*str*) – The full or relative filename path the study object should be saved to.
- **object_id** (*str*) – The name of the study that the study object should be saved to within the file. This needs to be a unique name in this file.

- **parameter_extraction_results** (*dict*) – The output of a FET2DAtomisticParameterExtraction.extractionResults() call.
- **number_of_processes_per_task** (int | None | *ProcessesPerNode*) – The number of processes that will be used to execute each task. If the total number of process does not divide evenly into the tasks, some tasks may have less than this number of processes. If None, all available processes execute each task collaboratively.
- **log_filename_prefix** (str | LogToStdOut) – Filename prefix for the logging output of the study. If LogToStdOut, all logging will instead be sent to standard output.

bfcal()

Getter for Bf_cal

channelMaterialName()

Returns
The name of the channel material.

Return type
str

channelThickness()

Returns
The thickness of the channel in monolayers.

Return type
int

`contactMaterialName()`

Returns
The name of the contact material.

Return type
str

`contactOrientation()`

Returns
The Miller indices for the contact surface towards the channel.

Return type
sequence of int

`dependentStudies()`

Returns
The list of dependent studies.

Return type
list of `Study`

`deviceStructureVariables()`

Returns
Device structure variables.

Return type
dict

`eMparLEV()`
Getter for EMparLEV

`eMparUEV()`
Getter for EMparUEV

`eMperpLEV()`
Getter for EMperpLEV

`eMperpUEV()`
Getter for EMperpUEV

`eshift()`
Getter for Eshift

`extractionResults()`

Returns
All results excluding NL objects.

Return type
dict

`filename()`

Returns
The filename where the study object is stored.

Return type
str

`gcal()`

Getter for g_cal

`kcal()`

Getter for k_cal

`logFilenamePrefix()`

Returns
The filename prefix for the logging output of the study.

Return type
str | `LogToStdOut`

`lowerEnergyValley()`

Getter for LowerEnergyValley

`maxE()`

Returns
The upper energy limit for valley detection.

Return type
PhysicalQuantity of type energy

`metatext()`

Returns
The metatext of the object or None if no metatext is present.

Return type
str | None

`nLowBands()`

Getter for NLowBands

`nPparLEV()`

Getter for NPparLEV

`nPparUEV()`

Getter for NPparUEV

`nUpBands()`

Getter for NUpBands

`negfVariables()`

Returns
Input variables for NEGF.

Return type
dict

nlinfo()

Get only the printable results from the extraction results - plain text.

Returns
The study information.

Return type
dict

nlinfoHtml()

Get only the printable results from the extraction results.

Returns
The study information.

Return type
dict

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

qatkBands()

Returns
QATK bands.

Return type
dict

resultsToTransfer()

Returns
A copy of the relevant results from parameter extraction calculation.

Return type
dict

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

setMetatext(*metatext*)

Set a given metatext string on the object.

Parameters
metatext (*str | None*) – The metatext string that should be set. A value of None can be given to remove the current metatext.

tfcal()

Getter for Tf_cal

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

upperEnergyValley()

Getter for UpperEnergyValley

v0f_cal()

Getter for V0f_cal

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

Usage Examples

Example 1. Perform model calibration to extract the parameters used to run device characterization, with manually defined parameter extraction results.

```
from QuantumATK import *
from AddOns.FET2D.Study.FET2DSentaurusModelCalibration import␣
↪FET2DSentaurusModelCalibration

# Define previous parameter extraction results.
parameter_extraction_results = {
    'channel_material_name': 'MoS2',
    'contact_material_name': 'Gold',
    'channel_thickness': 1,
    'maxE': 1.1 * eV,
    'contact_orientation': (0, 0, 1),
    'NBands': 8,
    'chi': 4.25 * eV,
    'eps': 3.38,
```

(continues on next page)

(continued from previous page)

```
    'H': 0.0006516 * um,
    'BG': 1.699 * eV,
    'qatk_bands': '0.367\n1.081\n1.923\n0.989\n0.346\n0.801\n1.773\n0.989\n0.254\n1.081\
↪n0.618\n0.989\n0.346\n0.801\n1.773\n0.989\n0.367\n1.081\n1.923\n0.989\n0.344\n0.801\n0.
↪433\n0.989\n0.254\n1.081\n0.618\n0.989\n0.344\n0.801\n0.433\n0.989\n0.452\n0.484\n0.
↪911\n0.85\n0.485\n0.484\n1.215\n0.85\n0.527\n0.484\n1.589\n0.85\n0.485\n0.484\n1.215\
↪n0.85\n0.312\n0.714\n0.542\n0.989\n0.204\n1.311\n1.155\n0.989\n0.713\n0.714\n1.153\n0.
↪989\n0.204\n1.311\n1.155\n0.989',
    'PhiSB': 0.276 * eV,
}

# Define the Model Calibration Study Object.
model_calibration = FET2DSentaurusModelCalibration(
    filename='fet2d_model_calibration_1642514480.76.hdf5',
    object_id='FET2DSentaurusModelCalibration_0',
    workfunction=7.835 * eV,
    parameter_extraction_results=parameter_extraction_results,
    log_filename_prefix_default='fet2d_model_calibration_',
)

# Start the calculations.
model_calibration.update()

# Extract the results.
model_calibration_results = model_calibration.extractionResults()
```

Example1.py

If a *FET2DAtomisticParameterExtraction* is available, the parameter extraction results can be defined as

```
parameter_extraction_results = parameter_extraction.extractionResults()
```

where `parameter_extraction` is an instance of *FET2DAtomisticParameterExtraction* that has completed all of its calculations.

See also fet2d_general.

GrainBoundaryScattering

class GrainBoundaryScattering(*configuration*, *filename*, *object_id*, *grain_boundary_generators*, *kpoint_density=None*, *optimization_region_length=None*, *optimize_geometry_parameters=None*, *energies=None*, *transmission_kpoint_density=None*, *self_energy_calculator=None*, *infinitesimal=None*, *passivate_electrode_surfaces=None*, *device_algorithm_parameters=None*, *contour_parameters=None*, *relaxation_calculator=None*, *log_filename_prefix=None*, *number_of_processes_per_task=None*)

Class for performing a grain boundary scattering study.

Parameters

- **configuration** (*DeviceConfiguration*) – The device configuration with attached calculator for which to perform the study. The configuration must include at least one metallic region acting as the gate.

- **filename** (*str*) – The full or relative filename path the Study object should be saved to. See *nlsave()*.
- **object_id** (*str*) – The name of the study that the Study object should be saved to within the file. This needs to be a unique name in this file. See *nlsave()*.
- **grain_boundary_generators** (list or *Table* of *GrainBoundaryGenerator*) – List or Table of *GrainBoundaryGenerator* to generate the device grain boundary structures.
- **kpoint_density** (*PhysicalQuantity* of type length.) – The k-point density to be used for the self-consistent DFT-NEGF calculation. **Default:** `4.0 * Ang`
- **optimization_region_length** (*PhysicalQuantity* of type length) – The length along the transport direction of the region to be optimized within the central region of the device. Note that the length of the optimization region is automatically capped to keep the given electrode extensions fixed on either side. **Default:** `10.0 * Ang`
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters to use for optimizing the geometry. Note that `constraints`, `pre_step_hook` and `post_step_hook` must be left unset. Also note that `max_stress`, `target_stress`, `restart_strategy` and `enable_optimization_stop_file` do not have any effect since stress is not optimized, and the restart mechanism within *OptimizeGeometry()* is disabled. **Default:** `OptimizeGeometryParameters()`
- **energies** (list of *PhysicalQuantity* of type energy) – A list of the energies for which the *TransmissionSpectrum* at each voltage should be calculated. **Default:** Energy range that covers the bias window, plus $30k_BT$.
- **transmission_kpoint_density** (*PhysicalQuantity* of type length.) – The k-point density to be used for the transmission spectrum calculations. **Default:** `7.0 * Ang`
- **self_energy_calculator** (*DirectSelfEnergy* | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*) – The SelfEnergyCalculator to be used for the *TransmissionSpectrum* calculation at each voltage. **Default:** `RecursionSelfEnergy(storage_strategy=NoStorage())`
- **infinitesimal** (*PhysicalQuantity* of type energy) – Small positive energy, used to move the *TransmissionSpectrum* calculation at each voltage away from the real axis. This is only relevant for recursion-style self-energy calculators. **Default:** `1.0e-6 * eV`
- **passivate_electrode_surfaces** (*bool*) – Whether the left and right surfaces of the central region of the device along the transport direction should be passivated when extracting the central region for optimization. **Default:** `False`.
- **device_algorithm_parameters** (*DeviceAlgorithmParameters*) – The DeviceAlgorithmParameters used for the device simulation **Default:** The default of the calculator.
- **contour_parameters** (*ContourParameters*) – The parameters used for the complex contour integration for the device calculation. **Default:** The default of the calculator.
- **relaxation_calculator** (*Calculator*) – The calculator used for relaxing the atomic coordinates of the bulk configuration and of the grain boundary configurations. The calculator must contain a basis set for the elements in the configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference.
- **log_filename_prefix** (str | `LogToStdOut`) – Filename prefix for the logging output of the calculations, each to be stored in a separate file. If `LogToStdOut`, all logging will instead be sent to standard output. **Default:** `'grain_boundary_scattering_'`
- **number_of_processes_per_task** (int | None | *ProcessesPerNode* | `AllProcesses`) – The number of processes that will be used to execute each task. If this value is greater than or equal to the total number of available processes, each single task will be executed collaboratively over all processes. Otherwise, a delegator-worker scheme is used; in this case, one process will be set aside as the delegator, and the remaining ones will be grouped into workers and execute tasks concurrently. **Default:** All available processes execute each task collaboratively.

addGrainBoundaryGenerator(*grain_boundary_generator=None*)

Add a grain boundary generator to the list of grain boundary generators.

Parameters
grain_boundary_generator (*GrainBoundaryGenerator*) – The grain boundary generator to generate the device grain boundary structures.

calculateMayadasShatzkesResistivityVsGrainBoundarySize(*bulk_resistivity*, *bulk_mean_free_path*, *grain_boundary_lengths*, *grain_boundary_weights=None*, *temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Calculate the resistivity vs. grain boundary size based on the Mayadas and Shatzkes model. Returns a list of resistivities corresponding to the valus of the GB lengths.

Parameters

- **bulk_resistivity** (*PhysicalQuantity* of type Meter / Siemens) – The bulk resistivity of the material without grain boundaries.
- **bulk_mean_free_path** (*PhysicalQuantity* of type length) – The bulk resistivity of the material without grain boundaries.
- **grain_boundary_lengths** (*PhysicalQuantity* of type length) – List of grain boundary lengths. The GB lengths are the average characteristic dimension of the grain boundaries.
- **grain_boundary_weights** – The weight factors for the individual grain boundary specific resistances. **Default:** list of equal weights, i.e. numpy.ones(number_of_grain_boundaries)
- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
The resistivity vs. grain boundary size based on the Mayadas and Shatzkes model.

Return type
PhysicalQuantity of type Meter/Siemens

calculatedGrainBoundaries()

Returns
The list GrainBoundaryGenerator descriptions for which all the calculation results are present in the cache.

Return type
list of *GrainBoundaryGenerator*

dependentStudies()

Returns
The list of dependent studies.

Return type
list of `Study`

electrodeTotalEnergies(*contribution=None*)

Retrieve the total energies for the specified electrode.

Parameters
contribution (Left | Right) – The electrode for which to get the energy.

Returns
The total energies. If not available, returns list of None.

Return type
list of *TotalEnergy* | None

electrodeTransmissionSpectra(*contribution=None*)

Retrieve the transmission spectra for the specified electrode.

Parameters
contribution (Left | Right) – The electrode for which to get the transmission.

Returns
The transmission spectra. If not available, returns list of None.

Return type
list of *TransmissionSpectrum* | None

energies()

Returns
The list of energies used for the *TransmissionSpectrum* calculation.

Return type
list of *PhysicalQuantity* of type energy

filename()

Returns
The filename where the study object is stored.

Return type
str

formationEnergies()

Returns
A list of GB formation energies for all the grain boundaries. If a grain boundary calculation is not finished, a None will be returned in the list for that grain boundary. The GB formation energy calculated as (E_device / N_device - E_elec / N_elec), where E_device is the total energy of the device configuration, E_elec is the energy of the electrodes, and N_device, N_elec are the number of atoms in the device and electrodes.

Return type
PhysicalQuantity of type energy | None

formationEnergiesBulk()

Returns
A list of GB formation energies for all the grain boundaries calculated from the relaxed bulk grain boundary configuration. If a grain boundary calculation is not finished, a None will be returned in the list for that grain boundary. The GB formation energy calculated as (E_device / N_device - E_elec / N_elec), where E_device is the total energy of the device configuration, E_elec is the energy of the electrodes, and N_device, N_elec are the number of atoms in the device and electrodes.

Return type
PhysicalQuantity of type energy | None

grainBoundaryGenerators()

Returns
All the grain boundary generators given as input.

Return type
GrainBoundaryGenerator

infinitesimal()

Returns
The small positive energy used to move the *TransmissionSpectrum* calculation away from the real axis for the case of a recursion-style self-energy calculator.

Return type
PhysicalQuantity of type energy

kpointDensity()

Returns
The kpoint density given as input.

Return type
PhysicalQuantity of type length.

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizationRegionLength()

Returns
The length along the transport direction of the optimization region. Note that the optimization region might be capped in order to keep at least one repeat of the minimal electrode fixed on either side.

Return type
PhysicalQuantity of type length

optimizeGeometryParameters()

Returns
The parameters used for optimizing the geometry.

Return type
OptimizeGeometryParameters

passivateElectrodeSurfaces()

Returns
Whether the left and right surfaces of the central region of the device along the transport direction are passivated when extracting the central region for optimization.

Return type
bool

reflectionCoefficients(*temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Retrieve the GB reflection coefficient for all the grain boundary generators.

Parameters

- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
The GB reflection coefficients calculated as r = 1 - R_elec / R_device, where R_device, R_elec = (R_left + R_right)/2 are the resistances of the device, and the average of left and right electrodes, respectively.

Return type
list of float | None

relaxedGrainBoundaryDevice(*grain_boundary_generator*)

Retrieve the relaxed device configuration.

Parameters
grain_boundary_generator (*GrainBoundaryGenerator*) – The grain boundary generator for which to get the device configuration.

Returns
The relaxed device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

resistances(*temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Retrieve the GB scattering resistance for all the grain boundary generators.

Parameters

- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
The GB scattering resistance calculated as R_device - 0.5*(R_left + R_right), where R_device, R_left, R_right are the resistances of the device, and left and right electrodes, respectively.

Return type
list *PhysicalQuantity* of type 1/Siemens | None

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

selfEnergyCalculator()

Returns
The SelfEnergyCalculator used for the *TransmissionSpectrum* calculation.

Return type
DirectSelfEnergy | *RecursionSelfEnergy* | *SparseRecursionSelfEnergy* | *KrylovSelfEnergy*

specificResistivities(*temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Retrieve the GB area specific resistivity for all the grain boundary generators.

Parameters

- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
A list with the specific resistivity for each grain boundary, calculated as (R_device - 0.5*(R_left + R_right)) * A, where A is the cross sectional area, and R_device, R_left, R_right are the resistances of the device, and left and right electrodes, respectively.

Return type
list *PhysicalQuantity* of type Meter^2/Siemens | None

totalEnergies()

Retrieve the total energies for the device configurations.

Returns
The total energies. If not available, returns list of None.

Return type
list of *TotalEnergy* | None

totalEnergiesBulk()

Retrieve the total energies for the bulk GB configurations.

Returns
The total energies. If not available, returns list of None.

Return type
list of *TotalEnergy* | None

totalEnergyInitialBulk()

Retrieve the total energies for the initial bulk configuration.

Returns
The total energy If not available, returns list of None.

Return type
list of *TotalEnergy* | None

transmissionKpointDensity()

Returns
The kpoint density for the transmissson calculations given as input.

Return type
PhysicalQuantity of type length.

transmissionSpectra()

Retrieve the transmission spectra

Returns
The transmission spectra. If not available, returns list of None.

Return type
list of *TransmissionSpectrum* | None

uniqueString()

Return a unique string representing the state of the object.

unrelaxedGrainBoundaryDevice(*grain_boundary_generator*)

Retrieve the unrelaxed device configuration.

Parameters
grain_boundary_generator (*GrainBoundaryGenerator*) – The grain boundary generator for which to get the device configuration.

Returns
The unrelaxed device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

update()

Run the calculations for the study object.

updatedGrainBoundaryDevice(*grain_boundary_generator*)

Retrieve the updated, and optimized device configuration

Parameters
grain_boundary_generator (*GrainBoundaryGenerator*) – The grain boundary generator for which to get the device configuration.

Returns
The optimized device configuration. If not available, returns None.

Return type
DeviceConfiguration | None

weightedReflectionCoefficient(*grain_boundary_weights=None*, *temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Retrieve the weighted reflection coefficient.

Parameters

- **grain_boundary_weights** – The weight factors for the individual grain boundary specific resistances. **Default:** list of equal weights, i.e. numpy.ones(number_of_grain_boundaries)
- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy.) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
The weighted GB specific resistance, calculated as sum(r_i * w_i) / sum(w_i), where r_i is the specific resistances and w_i are the weight factors.

Return type
PhysicalQuantity of type Meter^2/Siemens

weightedSpecificResistivity(*grain_boundary_weights=None*, *temperature=PhysicalQuantity(300.0, K)*, *spin=<class 'NL.ComputerScienceUtilities.NLFlag.Spin.Sum'>*, *fermi_shift=PhysicalQuantity(0.0, eV)*)

Retrieve the weighted specific resistivity.

Parameters

- **grain_boundary_weights** – The weight factors for the individual grain boundary specific resistivities. **Default:** list of equal weights, i.e. numpy.ones(number_of_grain_boundaries)
- **temperature** (*PhysicalQuantity* of type Kelvin) – The electrode temperatures.
- **spin** (`Spin.Up` | `Spin.Down` | `Spin.Sum`) – The spin component to query result for. **Default:** `Spin.Sum`
- **fermi_shift** (*PhysicalQuantity* of type energy) – Manuel shift of the Fermi level shift. **Default:** `0.0 * eV`

Returns
The weighted GB specific resistivity, calculated as sum(r_i * w_i) / sum(w_i), where r_i is the specific resistivities and w_i are the weight factors.

Return type
PhysicalQuantity of type Meter^2/Siemens

Usage Example

This example script shows how to perform a grain boundary scattering calculation of Copper with two different grain boundaries considered. The grain boundaries are specified using the *GrainBoundaryGenerator* objects.

```
from QuantumATK import *
from SMW import *

# ------------------------------------------------------------
# Bulk Configuration
# ------------------------------------------------------------

# Set up lattice
lattice = FaceCenteredCubic(3.61496*Angstrom)

# Define elements
elements = [Copper]

# Define coordinates
fractional_coordinates = [[ 0.,  0.,  0.]]

# Set up configuration
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

# ------------------------------------------------------------
# Calculator
# ------------------------------------------------------------
k_point_sampling = MonkhorstPackGrid(
    na=11,
    nb=11,
    nc=11,
    )
numerical_accuracy_parameters = NumericalAccuracyParameters(
    density_mesh_cutoff=85.0*Hartree,
    k_point_sampling=k_point_sampling,
    )

calculator = LCAOCalculator(
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)

# ------------------------------------------------------------
# Grain Boundary Scattering
# ------------------------------------------------------------
# Grain boundary generators.
grain_boundary_generators = []
```

(continues on next page)

(continued from previous page)

```
# Grain Boundary Generator 0
grain_boundary_generator = GrainBoundaryGenerator(
    rotation_axis=[1, 1, 1],
    rotation_angle=60.00*Degrees,
    sigma=3,
    boundary_plane=[-1, -1, -1],
    minimum_electrode_length=5.0*Angstrom,
    buffer_layer=15.0*Angstrom,
    overlap_distance=0.5*Angstrom,
)
grain_boundary_generators.append(grain_boundary_generator)

# Grain Boundary Generator 1
grain_boundary_generator = GrainBoundaryGenerator(
    rotation_axis=[0, 0, 1],
    rotation_angle=36.87*Degrees,
    sigma=5,
    boundary_plane=[-2, -1, 0],
    minimum_electrode_length=5.0*Angstrom,
    buffer_layer=15.0*Angstrom,
    overlap_distance=0.5*Angstrom,
)
grain_boundary_generators.append(grain_boundary_generator)

# Optimization parameters.
optimize_geometry_parameters = OptimizeGeometryParameters(
     max_forces=0.05*eV/Angstrom,
     max_steps=200,
     max_step_length=0.2*Angstrom,
     optimizer_method=LBFGS(),
)

# Calculate scattering.
grain_boundary_scattering = GrainBoundaryScattering(
    configuration=bulk_configuration,
    filename='Copper-Grain-Boundary-Scattering.hdf5',
    object_id='grainboundaryscattering',
    grain_boundary_generators=grain_boundary_generators,
    kpoint_density=4.0*Angstrom,
    optimization_region_length=10.0*Angstrom,
    optimize_geometry_parameters=optimize_geometry_parameters,
    energies=numpy.linspace(-0.1, 0.1, 21)*eV,
    transmission_kpoint_density=12.0*Angstrom,
    self_energy_calculator=RecursionSelfEnergy(),
    infinitesimal=1e-06*eV,
    log_filename_prefix='grainboundaryscattering_',
    number_of_processes_per_task=1,
)
grain_boundary_scattering.update()
```

grain_boundary_scattering.py

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *GrainBoundaryScattering* object allows the user to study electonic transport and scattering properties of metal grain boundaries in an automated work flow. The purpose is to bundle a number of individual tasks into a single work flow. The grain boundaries are specified using *GrainBoundaryGenerator* objects, which can generate a device configuration from a bulk configuration of a cubic lattice type. As seen in the example above, the *GrainBoundaryScattering* is thus constructed with a bulk configuration and a list of *GrainBoundaryGenerator* objects.

The work flow is the following:

•Initially the bulk configuration is optimized in order to minimize the forces and stress.

For each grain boundary structure the following tasks will be performed:

•Construct the device configuration using the GB generator and the relaxed bulk configuration.

•Convert the device configuration to a bulk configuration as in *OptimizeDeviceConfiguration*.

•Perform structural relaxation of the bulk configuration as in *OptimizeDeviceConfiguration*. Only the central atoms around the grain boundary will be relaxed. The length of the region with structural relaxations is determined by the input parameters *optimization_region_length*.

•Convert the relaxed bulk configuration to a device configuration.

•Perform a self-consistent DFT-NEGF calculation for the relaxed device configuration.

•Total energy of relaxed DeviceConfiguration.

•Total energy of electrode configuration.

•TransmissionSpectrum calculation for the device configuration, $T_{dev}(E)$

•TransmissionSpectrum calculation for the pristine (electrode) structure (Sharvin resistance), $T_{elec}(E)$

The grain boundary resistance is calculated as[1]

$$R_{GB} = R_{dev} - R_{elec} = \frac{1}{G_{dev}} - \frac{1}{G_{elec}}$$

where

$$G_{dev} = \int T_{dev}(E) \left(-\frac{\partial f(E)}{\partial E} \right) dE$$

is the device conductance obtained from the device transmission function, $T_{dev}(E)$, and the energy derivative of the Fermi-Dirac distribution function. Similarly the electrode conductance and resistance can be obtained from the electrode transmission function. The electrode transmission represents the transmission through a pure metal without any defects or grain boundaries.

The specific resistivity is determined as

$$\gamma = R_{GB} \cdot A$$

where A is the cross-sectional area.

The grain boundary reflection coefficient is defined as

$$r = 1 - \frac{G_{dev}}{G_{elec}}$$

[1] Mathieu Cesar, Dongping Liu, Daniel Gall, and Hong Guo. Calculated Resistances of Single Grain Boundaries in Copper. *Phys. Rev. Applied*, 2:044007, Oct 2014. URL: https://link.aps.org/doi/10.1103/PhysRevApplied.2.044007, doi:10.1103/PhysRevApplied.2.044007.

A weighted reflection coefficient can be obtained as

$$\langle r \rangle = \frac{\sum_i w_i r_i}{\sum w_i}$$

where the sum runs over all the considered grain boundaries. r_i and w_i are the reflection coefficient and relative weight of the ith grain boundary. The weighted resistivity is defined by a similar equation. The weights, or distribution, of different defects depends on the experimental conditions (metal type, growth conditions etc.) and will be user determined.

The conductivity as function of grain boundary size can finally be obtained from the Mayadas and Shatzkes model as

$$\sigma = \sigma_0 \cdot 3 \left(\frac{1}{3} - \frac{\alpha}{2} + \alpha^2 - \alpha^3 \ln(1 + \frac{1}{\alpha}) \right)$$

$$\alpha = \frac{l_0}{L_g} \frac{\langle r \rangle}{1 - \langle r \rangle}$$

where l_0 is the bulk mean free path, L_g the average grain size, and σ_0 is the bulk conductivity in the absence of grain boundaries. The bulk conductivity and mean free path are user inputs, but can be calculated from the *Mobility* object.

References

MaterialSpecifications

class MaterialSpecifications(*pristine_configuration*, *supercell_repetitions=None*, *formation_energy_calculator=None*, *relaxation_calculator=None*, *band_gap_calculator=None*, *phonon_calculator=None*, *quick_optimizer_calculator=None*, *amorphization_calculator=None*, *amorphous_density=None*, *atomic_chemical_potentials=None*, *dielectric_constant=None*, *bulk_modulus=None*, *assumed_formation_entropy=None*, *assumed_transition_prefactor=None*, *first_nearest_neighbor=None*, *second_nearest_neighbor=None*, *color=None*, *sentaurus_material_name=None*, *optimize_geometry_parameters=None*, *symmetry_tolerance=None*, *layer_indices=None*, *use_ghost_atoms=None*, *random_seed=None*, *periodic_charge_correction=None*, *stress_correction=None*, *elements_with_stoichiometry=None*, *substitutional_percentage=None*, *substitutional_element=None*)

A class defining a bulk material to use in an SMW simulation. The class encapsulates the structure of the pristine unit cell and all the technical options needed to perform the calculations.

Parameters

- **pristine_configuration** (*BulkConfiguration*) – The reference bulk unit cell configuration to be used to generate the supercell configuration.
- **supercell_repetitions** (*sequence (size 3) of int*) – The number of repetitions of the reference bulk unit cell along the (a, b, c) directions. **Default:** `(1, 1, 1)`
- **formation_energy_calculator** (*Calculator*) – The calculator used for calculating the formation energy of the defect charge states. All total energies, band positions and finite-size corrections are calculated using this calculator. The calculator must contain a basis set for the elements in both the defect and pristine configurations. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `relaxation_calculator`. At least one of the two calculators must be provided.

- **relaxation_calculator** (*Calculator*) – The calculator used for relaxing the atomic coordinates of the defect supercells. The calculator must contain a basis set for the elements in the defect configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `formation_energy_calculator`. At least one of the two calculators must be provided.

- **band_gap_calculator** – The calculator used for calculating the band gap in which the transition levels will be placed. The calculator is only used for the pristine configuration, so only needs to contain a basis set for the elements in this configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `formation_energy_calculator`.

- **phonon_calculator** – The calculator used for performing phonon calculations. The calculator must contain a basis set for the elements in the defect configuration. Note that the calculator is taken as a reference corresponding to the bulk unit cell given in `bulk_configuration`; the `density_mesh_cutoff` and `k_point_sampling` parameters of the *NumericalAccuracyParameters* of the calculator will be scaled consistently with the supercell size. The `k_point_sampling` will be scaled to maintain the k-point density approximately equal to the reference. The k-point grid will always be shifted such that the Gamma point is included, i.e., `shift_to_gamma=True`. **Default:** The calculator given in `relaxation_calculator`.

- **quick_optimizer_calculator** (*Calculator*) – The calculator used for fast initial optimization of amorphous packmol structures. This should typically be a force field calculator since the optimization of packmol structure procedure requires many force evaluations. **Default:** None

- **amorphization_calculator** (*Calculator*) – The calculator used for producing amorphous structures. This should typically be a force field calculator since the amorphization procedure requires many force evaluations. **Default:** None

- **amorphous_density** (*PhysicalQuantity* of type mass per volume) – The target density of this material when producing amorphous structures. If not value is given, then the density will be the same as the pristine configuration. **Default:** None

- **atomic_chemical_potentials** (list of *AtomicChemicalPotential* | None) – The atomic chemical potential for each element of interest. See the documentation of *AtomicChemicalPotential* for details on how to define the atomic chemical potential for a species, and how the default values are calculated. **Default:** None; all necessary atomic chemical potentials calculated using the default configurations.

- **dielectric_constant** (*float*) – The dielectric constant of the pristine material. Only needed to calculate the finite-size electrostatic energy correction terms. **Default:** `1.0`

- **bulk_modulus** (*PhysicalQuantity* of type pressure) – The bulk modulus of the pristine material. Only needed to calculate the finite-size elastic energy correction term. **Default:** `100.0 * GPa`

- **assumed_formation_entropy** (*PhysicalQuantity* of type entropy) – The assumed formation entropy of all defects calculated for this material. Setting this parameter will disable all phonon contributions to the defects. **Default:** Phonon contributions will be calculated.

- **assumed_transition_prefactor** (*PhysicalQuantity* of type frequency) – The assumed prefactor for all transition paths calculated for this material. Setting this parameter will disable the prefactor calculation from the vibrational frequencies at the minima and saddle point for all transition paths. **Default:** The prefactors will be calculated from the vibrational frequencies at the minima and saddle point.

- **first_nearest_neighbor** (*PhysicalQuantity* of type length) – The distance for first nearest neighbors in the material **Default:** Distance automatically calculated.

- **second_nearest_neighbor** (*PhysicalQuantity* of type length) – The distance for second nearest neighbors in the material **Default:** Distance automatically calculated.
- **color** (*Valid color in hexadecimal format as str*) – A representative color for this material. **Default:** #FFFFFF
- **sentaurus_material_name** (*str*) – The material name for Sentaurus tools. A typical use is when writing parameters for other Sentaurus (Process, Device) tools. This parameter is only needed when providing parameters for other tools. Otherwise it can be left empty. But if a name is not provided, an exception/error will be raised when used by functions providing parameters for other Sentaurus tools. **Default:** None
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The parameters to use for optimizing the geometry. Note that `pre_step_hook` and `post_step_hook` must be left unset. **Default:** `OptimizeGeometryParameters()`
- **symmetry_tolerance** (*PhysicalQuantity* of type length.) – The tolerance to be used when determining the symmetries. **Default:** 0.1 * Angstrom
- **layer_indices** (*list of type list of type int*) – A list of lists that contain the indices of the atoms that belong in each layer. If `None` is given (default), then this is not a multilayered material. **Default:** None
- **use_ghost_atoms** (*bool*) – If set to True ghost atoms are to be used. **Default:** False
- **random_seed** (*int*) – Random seed to be used when needed for simulations. **Default:** 1234
- **periodic_charge_correction** (*ModelChargeCorrection* | False) – Charge correction flag. False to disable. **Default:** ModelChargeCorrection() (Correction is on, gaussian width is fitted)
- **stress_correction** (*ImageStressCorrection* | False) – Type of stress correction applied to defects. False to disable **Default:** False
- **elements_with_stoichiometry** (*dict*) – A dict of elements and stoichiometry of the elements that is used for amorphorization.
- **substitutional_percentage** (*float*) – The percentages of the atoms that should be changed.
- **substitutional_element** (*PeriodicTableElement*) – The element to change to.

amorphizationCalculator()

Returns
The reference calculator used for performing amorphization calculations.

Return type
Calculator | None

amorphousDensity()

Returns
The target density of this material when producing amorphous structures.

Return type
PhysicalQuantity of type mass per volume

assumedFormationEntropy()

Returns
The assumed formation entropy of all defects calculated for this material. If None, the phonon contributions are calculated explicitly.

Return type
PhysicalQuantity of type entropy | None

assumedTransitionPrefactor()

Returns
The assumed prefactor for all transition paths calculated for this material. If None, the prefactors are calculated explicitly from the vibrational frequencies at the minima and saddle point.

Return type
PhysicalQuantity of type frequency | None

atomicChemicalPotentials()

Returns
The list of atomic chemical potentials for each element.

Return type
list of *AtomicChemicalPotential*

bandGapCalculator()

Returns
The reference calculator used for calculating the band gap in which the transition levels will be placed.

Return type
Calculator

bulkModulus()

Returns
The bulk modulus of the pristine material.

Return type
PhysicalQuantity of type pressure

color()

Returns
The color of the material.

Return type
str

description()

Returns
The description of the material.

Return type
str

dielectricConstant()

Returns
The dielectric constant of the pristine material.

Return type
float

elementsWithStoichiometry()

Returns
Different elements with stoichiometry.

Return type
dict | None

firstNearestNeighbor()

Returns
The first nearest neighbor distance associated to this system.

Return type
PhysicalQuantity of type length

formationEnergyCalculator()

Returns
The reference calculator used for calculating the formation energy of the defect charge states.

Return type
Calculator

layerIndices()

Returns
The indices of the atoms in each layer.

Return type
list of type list of type int

name()

Returns
The name of the material, mostly used for displaying purposes. This name might not be unique for different specifications. Do not use for indexing purposes.

Return type
str

nlinfo()

Returns
The nlinfo.

Return type
dict

nlprint(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Prints a summary of material specifications.

optimizeGeometryParameters()

Returns
Parameters required to optimize the geometry.

Return type
OptimizeGeometryParameters

periodicChargeCorrection()

Returns
The type of charge correction used for calculations

Return type
ModelChargeCorrection

phononCalculator()

Returns
The reference calculator used for performing phonon calculations.

Return type
Calculator

pristineConfiguration()

Returns
The reference bulk unit cell configuration used to generate the supercell configuration.

Return type
BulkConfiguration

quickOptimizerCalculator()

Returns
The reference calculator used for performing amorphization calculations.

Return type
Calculator | None

randomSeed()

Returns
The random seed. None if it changes between simulations.

Return type
int

relaxationCalculator()

Returns
The reference calculator used for relaxing the atomic coordinates of the defect supercells.

Return type
Calculator

secondNearestNeighbor()

Returns
The second nearest neighbor distance associated to this system.

Return type
PhysicalQuantity of type length

sentaurusMaterialName()

Returns
The name to be used for this material in Sentarus tools

Return type
str

setDescription(*description*)

Parameters
description (*str*) – The description of the material.

stressCorrection()

Returns
The Type of image stress correction used for calculations

Return type
ImageStressCorrection | None

substitutionalElement()

Returns
The substitutional element for this layer as defined by the material specification.

Return type
class:~.*PeriodicTableElement*

substitutionalPercentage()

Returns
The percentage of the element in the pristine material that will be replaced.

Return type
float

supercellRepetitions()

Returns
The number of repetitions of the reference bulk unit cell along the (a, b, c) directions.

Return type
tuple (size 3) of int

symmetryTolerance()

Returns
The tolerance to be used when determining the symmetries.

Return type
PhysicalQuantity of type length.

uniqueString()
Return a unique string representing the state of the object.

useGhostAtoms()

Returns
whether ghost atoms are being used

Return type
bool

MaterialSpecificationsDatabase

Table 4.34: Database of MaterialSpecifications.

Name	Structure	# atoms	Calculator	Calculate phonons	Details
Alumina	Al2O3	240	TremoloX	✓	
Hafnia	HfO2	96	TremoloX	✓	
Hafnia_MTP	HfO2	96	TremoloX	✓	

Table 4.34 – continued from previous page

Name	Structure	# atoms	Calculator	Calculate phonons	Details
Hafnia_cubic	HfO2	24	TremoloX	✓	
Hafnia_orthorhombic	HfO2	24	TremoloX	✓	
Hafnia_tetragonal	HfO2	48	TremoloX	✓	
Hafnia_triclinic	HfO2	102	TremoloX	✓	
Ruthenium_MTP	Ru	16	TremoloX	✓	
Scandium_MTP	Sc	16	TremoloX	✓	
Silica	SiO2	72	TremoloX	✓	
Silica_Cristobalite	SiO2	48	TremoloX	✓	
Silica_MTP	SiO2	72	TremoloX	✓	
Silicon_Forcefield	Si	16	TremoloX	✓	
TitaniumNitride_MTP	TiN	64	TremoloX	✓	
Tungsten	W	8	TremoloX	✓	
AlAs_216_DFT_SCAN	AlAs	216	LCAO		Generally applicable high-precision p
AlN_300_DFT_SCAN	AlN	300	LCAO		Generally applicable high-precision p
Aluminum_256_DFT_SCAN	Al	256	LCAO		Generally applicable high-precision p
Cobalt_256_DFT_SCAN	Co	256	LCAO		Generally applicable high-precision p
Copper_256_DFT_SCAN	Cu	256	LCAO		Generally applicable high-precision p
GaAs_216_DFT_SCAN	GaAs	216	LCAO		Generally applicable high-precision p
GaN_300_DFT_SCAN	GaN	300	LCAO		Generally applicable high-precision p
Germanium_216_DFT_SCAN	Ge	216	LCAO		Generally applicable high-precision p
Gold_256_DFT_SCAN	Au	256	LCAO		Generally applicable high-precision p
HfO2_324_DFT_SCAN	HfO2	324	LCAO		Generally applicable high-precision p
InAs_216_DFT_SCAN	InAs	216	LCAO		Generally applicable high-precision p
InP_216_DFT_SCAN	InP	216	LCAO		Generally applicable high-precision p
Iridium_256_DFT_SCAN	Ir	256	LCAO		Generally applicable high-precision p
Iron_250_DFT_SCAN	Fe	250	LCAO		Generally applicable high-precision p
MgO_216_DFT_SCAN	MgO	216	LCAO		Generally applicable high-precision p
Palladium_256_DFT_SCAN	Pd	256	LCAO		Generally applicable high-precision p
Platinum_256_DFT_SCAN	Pt	256	LCAO		Generally applicable high-precision p
Ruthenium_288_DFT_SCAN	Ru	288	LCAO		Generally applicable high-precision p
SiC_2H_300_DFT_SCAN	SiC	300	LCAO		Generally applicable high-precision p
SiC_4H_256_DFT_SCAN	SiC	256	LCAO		Generally applicable high-precision p
SiC_6H_384_DFT_SCAN	SiC	300	LCAO		Generally applicable high-precision p
SiO2_243_DFT_SCAN	SiO2	243	LCAO		Generally applicable high-precision p
Silicon_216_DFT_SCAN	Si	216	LCAO		Generally applicable high-precision p
TaN_270_DFT_SCAN	TaN	270	LCAO		Generally applicable high-precision p
Tantalum_250_DFT_SCAN	Ta	250	LCAO		Generally applicable high-precision p
TiN_216_DFT_SCAN	TiN	216	LCAO		Generally applicable high-precision p
Titanium_200_DFT_SCAN	Ti	200	LCAO		Generally applicable high-precision p
Tungsten_250_DFT_SCAN	W	250	LCAO		Generally applicable high-precision p
ZrO2_324_DFT_SCAN	ZrO2	324	LCAO		Generally applicable high-precision p

MolecularDynamicsMeltQuench

class MolecularDynamicsMeltQuench(*amorphous_quality=None*, *max_retries=None*, *melt_quality=None*, *max_melt_retries=None*, *melt_temperature=None*, *melt_time=None*, *quench_temperature=None*, *quench_rate=None*, *time_step=None*, *random_seed=None*)

Base class for melt/quench amorphization methods

Parameters

- **amorphous_quality** (*float*) – Expected quality for the amorphous material, 0 to 1. 1 being free of defects. **Default:** 0.75
- **max_retries** (*int*) – Maximum number of tries to satisfy amorphous quality **Default:** 30
- **melt_quality** (*float*) – Requested percentage of melted material **Default:** 0.35
- **max_melt_retries** (*float*) – Maximum number of tries to melt the material with the requested quality **Default:** same as `max_retries`
- **melt_temperature** (*PhysicalQuantity* of type temperature) – Suggested melt temperature **Default:** 5000*Kelvin
- **melt_time** (*PhysicalQuantity* of type time) – The length of the melt MD simulation. **Default:** 40*picosecond
- **quench_temperature** (*PhysicalQuantity* of type temperature) – The target temperature during the quenching MD simulation. **Default:** 300*Kelvin
- **quench_rate** (*PhysicalQuantity* of type temperature per time) – The rate of temperature change during the quenching. This rate will be used to calculate the length of the quenching MD simulation. **Default:** 400*Kelvin/picosecond
- **time_step** (*PhysicalQuantity* of time time) – Time step per iteration **Default:** 1*femtosecond
- **random_seed** (*int*) – Random seed **Default:** 1234

amorphousQuality()

Returns
requested amorphous quality

Return type
float

maxMeltRetries()

Returns
requested maximum number of melt tries

Return type
int

maxRetries()

Returns
requested number of maximum tries

Return type
int

meltQuality()

Returns
requested melt quality

Return type
float

`meltSteps()`

Returns
The calculated number of melt steps.

Return type
int

`meltTemperature()`

Returns
requested melt temperature

Return type
PhysicalQuantity of type temperature

`meltTime()`

Returns
requested melt time

Return type
PhysicalQuantity of type time

`quenchRate()`

Returns
requested quench rate

Return type
PhysicalQuantity of type temperature per time

`quenchTemperature()`

Returns
requested quench temperature

Return type
PhysicalQuantity of type temperature

`randomSeed()`

Returns
random seed

Return type
int

`timeStep()`

Returns
returns time step

Return type
int

`uniqueString()`
Return a unique string representing the state of the object.

Example

This class is to be used as a method of *amorphize*:

```
method = MolecularDynamicsMeltQuench(amorphous_quality=0.75)
amorphous = amorphize(reference, method, 'amorphous-Si.hdf5', 'amorphous')
```

MultilayerBuilder

class MultilayerBuilder(*filename*, *object_id*, *layers*, *surface_vectors*, *relaxation_calculator=None*, *kpoint_density=None*, *relaxation_length=None*, *interaction_length=None*, *optimize_geometry_parameters=None*, *annealing_parameters=None*, *max_iteration_annealing_parameters=1*, *select_from_different_cut_plane=False*, *log_filename_prefix=None*, *add_passivation=False*, *add_bridging_oxygen_atoms=True*, *defect_fuzz_factor=None*)

Constructor for the MultilayerBuilder object. A multilayer builder is a multilayer configuration composed of crystalline, alloy, amorphous, or vacuum layers. The lateral dimensions (a and b unit cell vectors) are specially chosen to minimize the strain while meeting certain constraints (e.g. minimum and maximum sizes).

Parameters

- **filename** (*str*) – The full or relative path to save the results to. See *nlsave()*.
- **object_id** (*str*) – The object id to use when saving. See *nlsave()*.
- **layers** (list of type *CrystalLayer* | *AmorphousLayer* | *VacuumLayer* | *AlloyLayer*) – The list of layers to create the multilayer builder from.
- **surface_vectors** (*PhysicalQuantity* of type length) – The pair of vectors the define the lateral dimensions of the multilayer builder. The vectors may be given as 2d or 3d vectors, but the z components must be zero. If no surface vectors are given, then lowest strain surface vectors are selected automatically.
- **relaxation_calculator** (*Calculator | list | None*) – The calculators to use when relaxing the atomic positions at the interfaces. If a single calculator is given it is used for all interfaces. Giving a list of calculators allows a different calculator for each interface and skips optimization of each material. If no additional relaxation is desired, then None can be given. **Default:** None
- **kpoint_density** (*PhysicalQuantity* of type length) – The target k-point sampling density. **Default:** 4.0 * Angstrom
- **relaxation_length** (*PhysicalQuantity* of type length | list of *PhysicalQuantity* of type length | None) – The distance around the interfacial plane that is relaxed with the relaxation calculator. Can be specified as one value, or a list of values for each interface. **Default:** 4.0 * Angstrom
- **interaction_length** (*PhysicalQuantity* of type length | list of *PhysicalQuantity* of type length | None) – The distance around the interfacial plane within which atomic interactions will be included during the interface optimization. Can be specified as one value, or a list of values for each interface. **Default:** 8.0 * Angstrom
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – The optimize geometry parameters to use when optimizing the atoms at each interface. Note that `restart_strategy` does not have any effect since the restart mechanism within *OptimizeGeometry()* is disabled. **Default:** `OptimizeGeometryParameters()`
- **annealing_parameters** (*list of type AnnealingParameters*) – The parameters for the interface annealing workflow. If None, no annealing MD is run.
- **max_iteration_annealing_parameters** (*list or single of type int*) – The maximum iteration parameter which is used to run multiple interface annealing workflow. If provided one value then it will use it for different interface. If a list is provided then it will use according to the order of the interface. If nothing is provided then it is 1. **Default:** 1

- **select_from_different_cut_plane** (*Bool*) – If True, the algorithm will scan over different cut-planes in the amorphous layer and use the one which results in the lowest number of coordination defects at the interface. **Default:** False
- **add_passivation** (*Bool*) – If True, the algorithm will passivate Si atoms in Si and SiO2 layer. **Default:** False
- **log_filename_prefix** (*str or None*) – Prefix for the filenames where the logging output for each calculation is stored. If a value of None is given then all logging output is done to stdout. **Default:** "multilayer_builder_"
- **defect_fuzz_factor** (*float*) – The fuzz factor used for the bond calculation to determine the coordination defects. **Default:** 1.15

Parm add_bridging_oxygen_atoms
The parameter adds oxygen atoms between neighboring under-coordinated Silicon atoms.

addBridgingOxygenAtoms()

Returns
If True, it adds oxygen atoms between neighboring under-coordinated Silicon atoms.

Return type
bool

addPassivation()

Returns
If true, the under coordinated silicon atoms is passivated with hydrogen.

Return type
bool

blocks()

Returns
A list of the block configurations.

Return type
list of type BulkConfiguration

configuration()

Returns
The multilayer builder configuration

Return type
BulkConfiguration

defectFuzzFactor()

Returns
The fuzz factor used for the bond calculation to determine the coordination defects.

Return type
float

dependentStudies()

Returns
The list of dependent studies.

Return type
list of Study

filename()

Returns
The filename where the study object is stored.

Return type
str

interactionLength()

Returns
The interaction length.

Return type
list of *PhysicalQuantity* of type length

kpointDensity()

Returns
The k-point density of the relaxation calculator.

Return type
PhysicalQuantity of type length

layerIndices()

Returns
A list containing the indices of each layer.

Return type
list of type list of type int

layers()

Returns
The layers of the multilayer builder.

Return type
list of type *CrystalLayer* | *AmorphousLayer* | *AlloyLayer* | *VacuumLayer*

logFilenamePrefix()

Returns
The filename prefix for the logging output of the study.

Return type
str | LogToStdOut

materialSpecifications(***kwargs*)

Parameters
kwargs (*dict*) – A list of keyword arguments to pass to MaterialSpecifications. By default the relaxation_calculator will be used for the formation_energy_calculator, relaxation_calculator, and phonon_calculator. The optimize_geometry_parameters will default to the values passed to MultilayerBuilder. The supercell_repetitions defaults to (1, 1, 1). The pristine_configuration will be the multilayer configuration.

Returns
The MaterialSpecifications for this multilayer configuration.

Return type
MaterialSpecifications

nlprint(*stream=None*)

Print a string containing an ASCII table useful for plotting the Study object.

Parameters
stream (*python stream*) – The stream the table should be written to. **Default:** NLPrintLogger()

numberOfProcessesPerTask()

Returns
The number of processes to be used to execute each task. If None, all available processes execute each task collaboratively.

Return type
int | None | *ProcessesPerNode*

numberOfProcessesPerTaskResolved()

Returns
The number of processes to be used to execute each task. Default values are resolved based on the current execution settings.

Return type
int

objectId()

Returns
The name of the study object in the file.

Return type
str

optimizeGeometryParameters()

Returns
The optimize geometry parameters.

Return type
OptimizeGeometryParameters

relaxationCalculator()

Returns
The relaxation calculators, with one calculator per interface.

Return type
list

relaxationLength()

Returns
The relaxation length.

Return type
list of *PhysicalQuantity* of type length

saveToFileAfterUpdate()

Returns
Whether the study is automatically saved after it is updated.

Return type
bool

uniqueString()

Return a unique string representing the state of the object.

update()

Run the calculations for the study object.

Usage Example

Use the *MultilayerBuilder* object to generate a Si-SiO2-HfO2-TiN High-k metal gate (HKMG) stack:

```
from SMW import *

# Define each layer in the multilayer structure.
layers = []
annealing_parameters = []
relaxation_calculator = []
amorphous_method = MolecularDynamicsMeltQuench(
    melt_temperature=3500*Kelvin,
    melt_time=40.0*ps,
    quench_temperature=300*Kelvin,
    quench_rate=2*37*Kelvin/ps,
    time_step=1.0*fs,
    max_retries=30,
    random_seed=1234,
)

material = MaterialSpecificationsDatabase.HKMG_MATERIALS['Silicon']()
layer_0 = CrystalLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_0)

material = MaterialSpecificationsDatabase.HKMG_MATERIALS['Silica'](
    amorphous_density=2.200*gram/cm**3,
    elements_with_stoichiometry={Silicon: 1, Oxygen: 2},
    )
layer_1 = AmorphousLayer(
    material,
    length=10.0*Angstrom,
    method=amorphous_method,
    )
layers.append(layer_1)

material = MaterialSpecificationsDatabase.HKMG_MATERIALS['Hafnia (monoclinic)'](
    amorphous_density=9.390*gram/cm**3,
    elements_with_stoichiometry={Hafnium: 1, Oxygen: 2},
    )
layer_2 = AmorphousLayer(
    material,
    length=10.0*Angstrom,
```

(continues on next page)

(continued from previous page)

```
    method=amorphous_method,
    )
layers.append(layer_2)

material = MaterialSpecificationsDatabase.HKMG_MATERIALS['Titanium nitride'](
    amorphous_density=5.210*gram/cm**3,
    elements_with_stoichiometry={Titanium: 1, Nitrogen: 1},
    )
layer_3 = AmorphousLayer(
    material,
    length=10.0*Angstrom,
    method=amorphous_method,
    )
layers.append(layer_3)


# ------------------------------------------------------------
# Setup annealing parameter used for optimizing interfaces.
# ------------------------------------------------------------
initial_velocity = MaxwellBoltzmannDistribution(
    temperature=600.0*Kelvin,
    remove_center_of_mass_momentum=True,
    random_seed=None,
    enforce_temperature=True,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=600*Kelvin,
    friction=0.05*femtoSecond**-1,
    initial_velocity=initial_velocity,
)
md_steps = 40000
annealing_parameters.append(
    AnnealingParameters(
        md_method=method,
        md_steps=md_steps
    )
)

initial_velocity = MaxwellBoltzmannDistribution(
    temperature=600.0*Kelvin,
    remove_center_of_mass_momentum=True,
    random_seed=None,
    enforce_temperature=True,
)

method = Langevin(
    time_step=1*femtoSecond,
    reservoir_temperature=600*Kelvin,
    friction=0.05*femtoSecond**-1,
    initial_velocity=initial_velocity,
```

(continues on next page)

(continued from previous page)

```
    heating_rate=-50*Kelvin/picoSecond,
)
md_steps = 10000
annealing_parameters.append(
    AnnealingParameters(
        md_method=method,
        md_steps=md_steps
    )
)


Si_SiO2_calculator = TremoloXCalculator(QuantumATK_MTP_Si_SiO2_2022())
relaxation_calculator.append(Si_SiO2_calculator)
HfO2_SiO2_calculator = TremoloXCalculator(QuantumATK_MTP_HfO2_SiO2_2022())
relaxation_calculator.append(HfO2_SiO2_calculator)
HfO2_TiN_calculator = TremoloXCalculator(QuantumATK_MTP_HfO2_TiN_2022())
relaxation_calculator.append(HfO2_TiN_calculator)

surface_vectors = [[ 11.520042257735,   0.            ],
                   [  7.680028171823,  -7.680028171823]]*Angstrom


multilayer_builder = MultilayerBuilder(
    filename='HKMG_builder.hdf5',
    object_id='multilayer_builder',
    layers=layers,
    surface_vectors=surface_vectors,
    relaxation_calculator=relaxation_calculator,
    annealing_parameters=annealing_parameters,
    max_iteration_annealing_parameters=1,
    select_from_different_cut_plane=True,
)
multilayer_builder.update()
nlsave('HKMG_builder.hdf5', multilayer_builder.configuration())
```

HKMG_builder_example.py

Use the *MultilayerBuilder* object to generate a MgO-FeCo-Ta-FeCo-MgO Magnetoresistive RAM (MRAM) stack:

```
from SMW import *

# Define each layer in the multilayer structure.
layers = []
relaxation_calculator = []

material = MaterialSpecificationsDatabase.MRAM_MATERIALS['Magnesium oxide']()
layer_0 = CrystalLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_0)
```

(continues on next page)

(continued from previous page)

```
material = MaterialSpecificationsDatabase.MRAM_MATERIALS['Iron cobalt (alloy)'](
    substitutional_percentage=30.000,
    substitutional_element=Cobalt,
    )
layer_1 = AlloyLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_1)

material = MaterialSpecificationsDatabase.MRAM_MATERIALS['Tantalum']()
layer_2 = CrystalLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_2)

material = MaterialSpecificationsDatabase.MRAM_MATERIALS['Iron cobalt (alloy)'](
    substitutional_percentage=30.000,
    substitutional_element=Cobalt,
    )
layer_3 = AlloyLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_3)

material = MaterialSpecificationsDatabase.MRAM_MATERIALS['Magnesium oxide']()
layer_4 = CrystalLayer(
    material,
    length=10.0*Angstrom,
    miller_indices=(1, 0, 0),
    )
layers.append(layer_4)


FeCo_MgO_calculator = TremoloXCalculator(QuantumATK_MTP_FeCo_MgO_2022())
relaxation_calculator.append(FeCo_MgO_calculator)
FeCo_Ta_calculator = TremoloXCalculator(QuantumATK_MTP_FeCo_Ta_2022())
relaxation_calculator.append(FeCo_Ta_calculator)
FeCo_Ta_calculator = TremoloXCalculator(QuantumATK_MTP_FeCo_Ta_2022())
relaxation_calculator.append(FeCo_Ta_calculator)
FeCo_MgO_calculator = TremoloXCalculator(QuantumATK_MTP_FeCo_MgO_2022())
relaxation_calculator.append(FeCo_MgO_calculator)

surface_vectors = [[ 2.977768076933,  0.            ],
                   [ 0.            ,  2.977768076933]]*Angstrom
```

(continues on next page)

(continued from previous page)

```
mram_builder = MultilayerBuilder(
    filename='MRAM_builder.hdf5',
    object_id='multilayer_builder',
    layers=layers,
    surface_vectors=surface_vectors,
    relaxation_calculator=relaxation_calculator,
    annealing_parameters=None,
    max_iteration_annealing_parameters=0,
)
mram_builder.update()
nlsave('MRAM_builder.hdf5', mram_builder.configuration())
```

MRAM_builder_example.py

Notes

Note: Study objects behave differently from analysis objects. See the *Study object* overview for more details.

The *MultilayerBuilder* class is designed to create an optimized multilayered material of low strain. The following types of layers are supported: *CrystalLayer*, *AlloyLayer*, *AmorphousLayer*.

This study object supports resuming tasks from the geometry optimization step that they left off at. This means that a job should resume from the previous step in case it is terminated in the middle of a geometry optimization.

The `relaxation_length` and `interaction_length` can be tuned according to the need of the interface. If the layer thickness is less than the `relaxation_length` then the optimization will not be done for the interface.

Since in particular the generation of amorphous layers and interfaces is stochastic, the recommended workflow is to create a series of candidate multilayer configurations, inspect these for structural defects, and use the best resulting configuration for further simulations.

To run the MultilayerBuilder efficiently in parallel we recommend using the OpenMP parallelization.

High-k metal gate (HKMG) stack

The HKMG configuration is created in three main steps. In the first step, a configuration is created for each layer. For crystal layers, the surface vectors and user given length are used to make a crystal of low strain. For amorphous layers, a random amorphous structure is generated in a simulated melt-quench process. This protocol uses the `amorphization_calculator` specified in the *MaterialSpecifications* of the layer multiple times and picks the configuration which has the lowest number of defects. In the second step, the layers are combined to form interfaces. The atoms near the interfaces (layer boundaries) are optimized using the `relaxation_calculator`. Additionally, a series of annealing simulations can be run on the layer boundaries to reduce the number of coordination defects at the interfaces to create more realistic interface structures. In the third step the optimized interfaces are combined into the final gate stack configuration, which can be accessed using the `configuration()` method.

Magnetoresistive RAM (MRAM) stack

The MRAM configuration can be created, using a similar protocol as described above. The builder for this type of stack is primarily designed to work with *CrystalLayer* and *AlloyLayer* types. By default, interfaces are not annealed, but this can be activated if needed. The final MRAM stack configuration can be accessed using the `configuration()` method.

Note: To achieve both high accuracy and short simulation times the multilayer builder primarily uses the machine-learned moment tensor potentials (MTP). MTP calculators are used for amorphization as well as optimizing and annealing the alloy layer and interfaces. The implementation and supported materials are mainly targeted at HKMG stack configurations and MRAM stack configuration. For HKMG, pre-trained models for the amorphization of silica, hafnia and titanium nitride, as well as the interfaces between these materials are provided. For MRAM, pre-trained models of iron cobalt alloy with cobalt concentration up to 50% and the interfaces between MgO-FeCo, FeCo-Ta and Feco-W are provided. These models can be extended by other force fields in a custom script.

TimeStampedForceBiasMonteCarloMeltQuench

class TimeStampedForceBiasMonteCarloMeltQuench(*amorphous_quality=None*, *max_retries=None*, *melt_quality=None*, *max_melt_retries=None*, *melt_temperature=None*, *melt_steps=None*, *quench_temperature=None*, *quench_rate=None*, *random_seed=None*)

Parameters

- **amorphous_quality** (*float*) – Expected quality for the amorphous material, 0 to 1. 1 being free of defects. **Default:** 0.75
- **max_retries** (*int*) – Maximum number of tries to satisfy amorphous quality **Default:** 2
- **melt_quality** (*float*) – Requested percentage of melted material **Default:** 0.35
- **max_melt_retries** (*float*) – Maximum number of tries to melt the material with the requested quality **Default:** same as `max_retries`
- **melt_temperature** (*PhysicalQuantity* of type temperature) – Suggested melt temperature **Default:** 5000*Kelvin
- **melt_steps** (*int*) – Number of Monte Carlo steps during melting **Default:** 2000
- **quench_temperature** (*PhysicalQuantity* of type temperature) – The target temperature during the quenching MD simulation. **Default:** 300*Kelvin
- **quench_rate** (*PhysicalQuantity* of type temperature) – The rate of temperature change per step during the quenching. This rate will be used to calculate the length of the quenching MC simulation. **Default:** 0.3 Kelvin
- **random_seed** (*int*) – Random seed **Default:** 1234

amorphousQuality()

Returns
requested amorphous quality

Return type
float

maxMeltRetries()

Returns
requested maximum number of melt tries

Return type
int

maxRetries()

Returns
requested number of maximum tries

Return type
int

meltQuality()

Returns
requested melt quality

Return type
float

meltSteps()

Returns
The melt Monte Carlo steps

Return type
int

meltTemperature()

Returns
requested melt temperature

Return type
PhysicalQuantity of type temperature

quenchRate()

Returns
requested quench rate

Return type
PhysicalQuantity of type temperature

quenchTemperature()

Returns
requested quench temperature

Return type
PhysicalQuantity of type temperature

randomSeed()

Returns
random seed

Return type
int

uniqueString()

Return a unique string representing the state of the object.

Example

This class is to be used as a method of *amorphize*:

```
method = TimeStampedForceBiasMonteCarlo(amorphous_quality=0.75)
amorphous = amorphize(reference, method, 'amorphous-Si.hdf5', 'amorphous')
```

VacuumLayer

class VacuumLayer(*length*)

Class to represent a vacuum layer in MultilayerBuilder.

Parameters
length (*PhysicalQuantity* of type length) – The target length (thickness) of the layer. A length of zero will generate the smallest possible layer.

length()

Returns
The target length (thickness) of the layer.

Return type
PhysicalQuantity of type length

uniqueString()

Return a unique string representing the state of the object.

Functions

amorphize

amorphize(*configuration*, *method*, *filename*, *object_id*, *amorphous_density=None*, *optimize_geometry_parameters=None*)

Amorphizes the configuration using a particular method and saving the temporal results (for restarting or visualization) in filename

Parameters

- **configuration** (*BulkConfiguration*) – The configuration to amorphize, with an attached calculator
- **method** (*TimeStampedForceBiasMonteCarloMeltQuench* | *MolecularDynamicsMeltQuench*) – Amorphization method.
- **filename** (*str*) – File name to save restart information
- **object_id** (*str*) – Identification (label) for objects saved in the filename
- **amorphous_density** (*PhysicalQuantity* of type mass per volume) – The target density of the amorphous structure. If not value is given, then the density will be the same as the configuration. **Default:** None
- **optimize_geometry_parameters** (*OptimizeGeometryParameters*) – Options to be used during final optimization step **Default:** *OptimizeGeometryParameters* defaults

Returns
An amorphized configuration

Return type
`BulkConfiguration`

Example

This function requires the specification of a particular method to produce the amorphous material:

```
method = MolecularDynamicsMeltQuench(amorphous_quality=0.75)
amorphous = amorphize(reference, method, 'amorphous-Si.hdf5', 'amorphous')
```

CHAPTER

FIVE

PUBLICATIONS

5.1 Referencing

- If you are preparing an article using QuantumATK, please see the *Referencing QuantumATK* page for instructions on how to quote the software properly. Here you will also find links to reference articles and background information on the methods used in QuantumATK.
- If you know of a QuantumATK-related publication that is not included, please send us an email with the full reference and we will be glad to add it to the the list!

5.2 List of Publications

Full, searchable list of QuantumATK papers, complete with abstracts and links to the full text papers.

BIBLIOGRAPHY

[FGH+14] Christoph Freysoldt, Blazej Grabowski, Tilmann Hickel, Jörg Neugebauer, Georg Kresse, Anderson Janotti, and ChrisăG. Van De Walle. First-principles calculations for point defects in solids. *Rev. Mod. Phys.*, 86(1):253–305, 2014. doi:10.1103/RevModPhys.86.253.

[FNVanDWalle09]Christoph Freysoldt, Jörg Neugebauer, and ChrisăG. Van De Walle. Fully Ab Initio finite-size corrections for charged-defect supercell calculations. *Phys. Rev. Lett.*, 102:016402, 2009. doi:10.1103/PhysRevLett.102.016402.

[Meh07a] Helmut Mehrer. *Dependence of Diffusion on Temperature and Pressure*, pages 127–149. Springer, 2007.

[Meh07b] Helmut Mehrer. *Point Defects in Crystals*, pages 69–91. Springer, 2007.

[SGS+13] Ali Sadeghi, S.ăAlireza Ghasemi, Bastian Schaefer, Stephan Mohr, MarkusăA. Lill, and Stefan Goedecker. Metrics for measuring distances in configuration spaces. *J. Chem. Phys*, 139:184118, 2013. URL: https://doi.org/10.1063/1.4828704, doi:10.1063/1.4828704.

PYTHON MODULE INDEX

n

NL.Calculators.GenericParameters.ElectrodeConstraint, 2798

s

SentaurusMaterialsWorkbench.MaterialSpecifications.MaterialSpecificationsDatabase, 4494